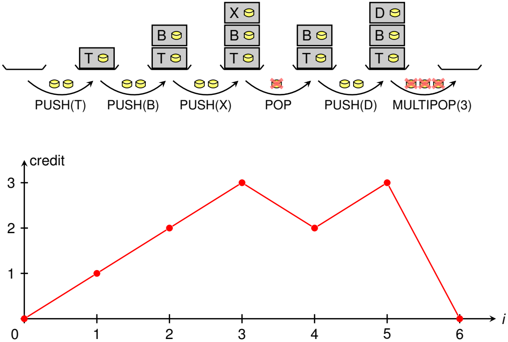


5.1: Amortized Analysis

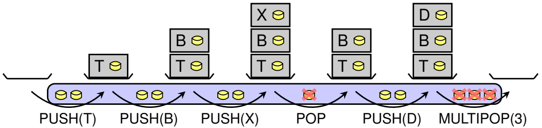
Frank Stajano

Thomas Sauerwald

Lent 2016

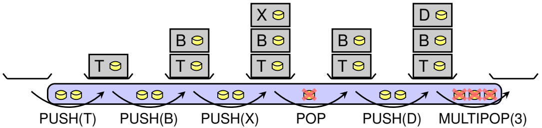
UNIVERSITY OF CAMBRIDGE

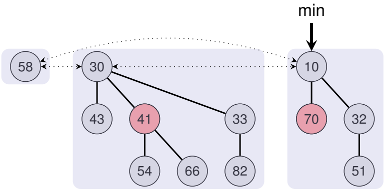
Use of Amortized Analysis

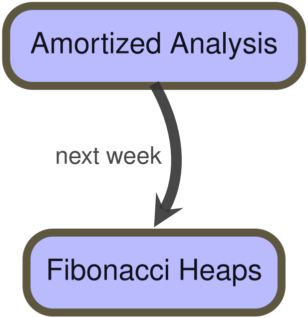


Amortized Analysis

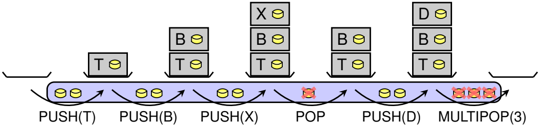
Use of Amortized Analysis

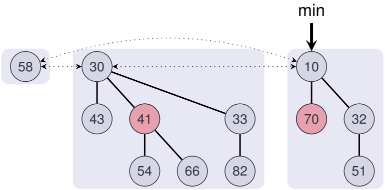


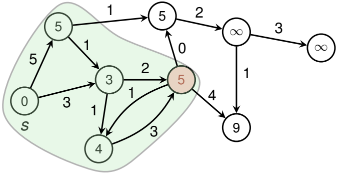


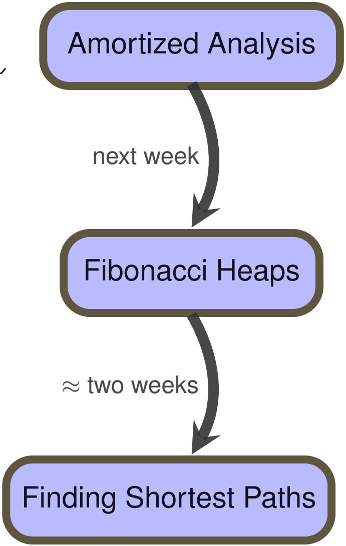


Use of Amortized Analysis









Motivating Example: Stack

Stack Operations

Motivating Example: Stack

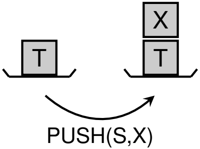
Stack Operations

- **`PUSH(S,x)`**

Motivating Example: Stack

Stack Operations

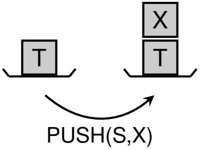
- **`PUSH(S,x)`**
  - pushes object x onto stack S



Motivating Example: Stack

Stack Operations

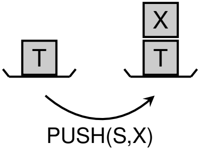
- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1



Motivating Example: Stack

Stack Operations

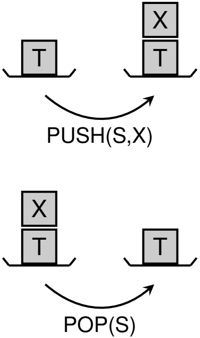
- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**



Motivating Example: Stack

Stack Operations

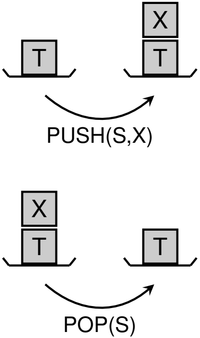
- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S



Motivating Example: Stack

Stack Operations

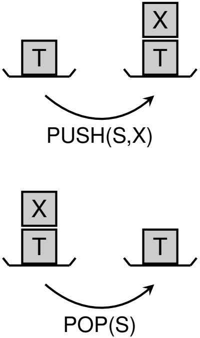
- **`PUSH(S,x)`**
  - pushes object *x* onto stack *S*
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack *S*
  - total cost of 1



Motivating Example: Stack

Stack Operations

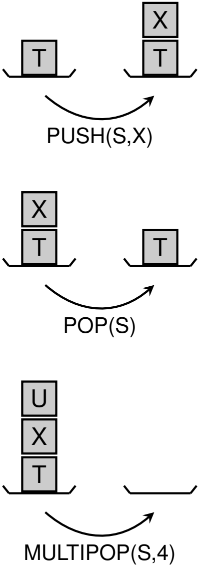
- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**



Motivating Example: Stack

Stack Operations

- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**
  - pops the k top objects (S non-empty)

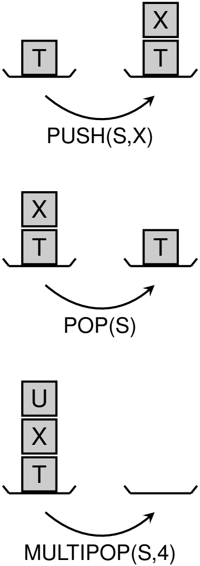


Motivating Example: Stack

Stack Operations

- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**
  - pops the k top objects (S non-empty)

  $\Rightarrow$ total cost of $\min\{|S|, k\}$



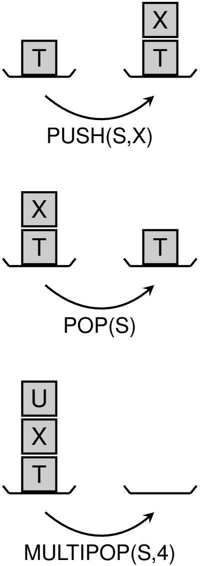
Motivating Example: Stack

Stack Operations

- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**
  - pops the k top objects (S non-empty)

$\Rightarrow$ total cost of $\min\{|S|, k\}$

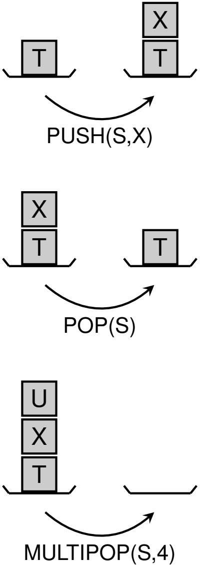
```
MULTIPOP(S,k)
while not S.empty() and k > 0
    POP(S)
    k = k - 1
```



Motivating Example: Stack

Stack Operations

- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**
  - pops the k top objects (S non-empty)
  - $\Rightarrow$ total cost of $\min\{|S|, k\}$

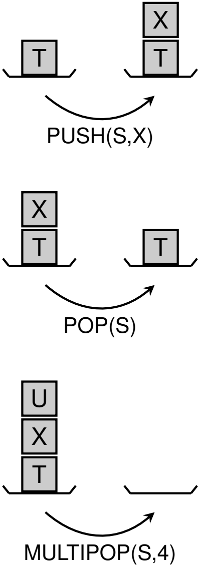


What is the largest possible cost of a sequence of n stack operations (starting from an empty stack)?

Motivating Example: Stack

Stack Operations

- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**
  - pops the k top objects (S non-empty)
  - $\Rightarrow$ total cost of $\min\{|S|, k\}$



What is the largest possible cost of a sequence of n stack operations (starting from an empty stack)?

Simple Worst-Case Bound (stack is initially empty):

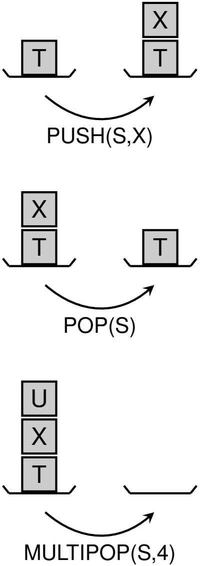
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$

Motivating Example: Stack

Stack Operations

- **`PUSH(S,x)`**
  - pushes object x onto stack S
  - total cost of 1
- **`POP(S)`**
  - pops the top of (a non-empty) stack S
  - total cost of 1
- **`MULTIPOP(S,k)`**
  - pops the k top objects (S non-empty)

  $\Rightarrow$ total cost of $\min\{|S|, k\}$



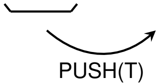
What is the largest possible cost of a sequence of n stack operations (starting from an empty stack)?

Simple Worst-Case Bound (stack is initially empty):

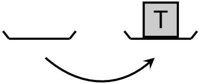
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**

Sequence of Stack Operations

Sequence of Stack Operations

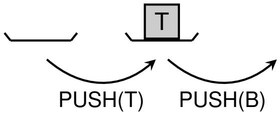


Sequence of Stack Operations

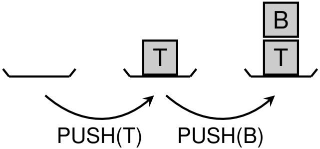


PUSH(T)

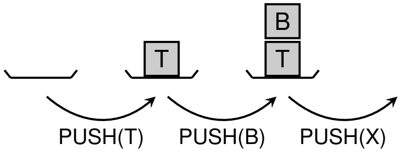
Sequence of Stack Operations



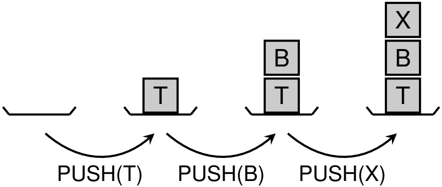
Sequence of Stack Operations



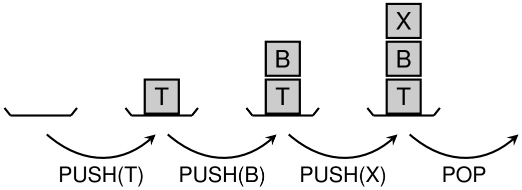
Sequence of Stack Operations



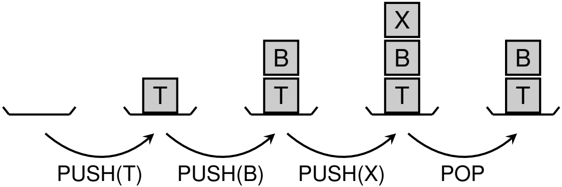
Sequence of Stack Operations



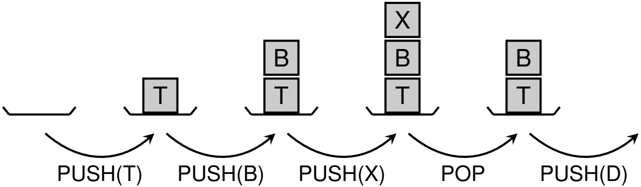
Sequence of Stack Operations



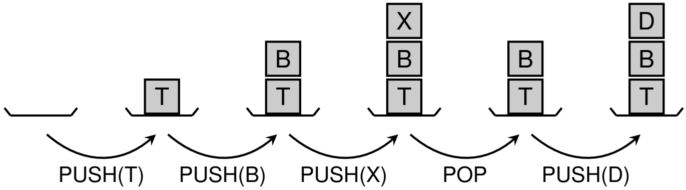
Sequence of Stack Operations



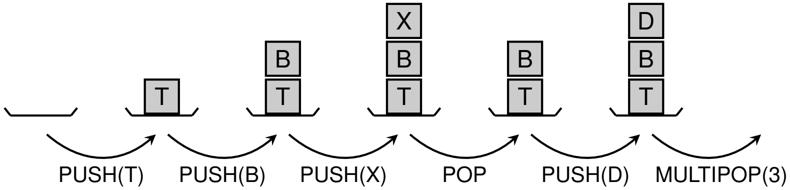
Sequence of Stack Operations



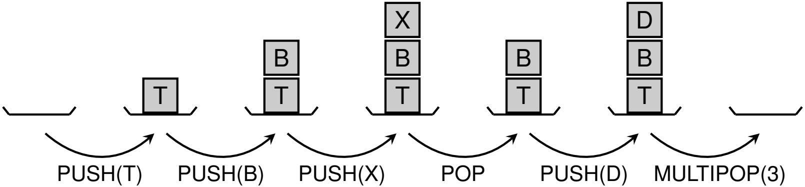
Sequence of Stack Operations



Sequence of Stack Operations



Sequence of Stack Operations



A new Analysis Tool: Amortized Analysis

Amortized Analysis

A new Analysis Tool: Amortized Analysis

Amortized Analysis

- analyse a sequence of operations

A new Analysis Tool: Amortized Analysis

Data structure operations (Heap, Stack, Queue etc.)

Amortized Analysis

- analyse a sequence of operations

A new Analysis Tool: Amortized Analysis

Amortized Analysis

- analyse a sequence of operations
- show that average cost of an operation is small

A new Analysis Tool: Amortized Analysis

Amortized Analysis

- analyse a sequence of operations
- show that average cost of an operation is small

This is **not** average case analysis!

A new Analysis Tool: Amortized Analysis

Amortized Analysis

- analyse a sequence of operations
- show that average cost of an operation is small
- concrete techniques
  - Aggregate Analysis
  - Potential Method

A new Analysis Tool: Amortized Analysis

Amortized Analysis

- analyse a sequence of operations
- show that average cost of an operation is small
- concrete techniques
  - Aggregate Analysis
  - Potential Method

Aggregate Analysis

- Determine an upper bound $T(n)$ for the total cost of any sequence of n operations
- amortized cost of each operation is the average $\frac{T(n)}{n}$

A new Analysis Tool: Amortized Analysis

Amortized Analysis

- analyse a sequence of operations
- show that average cost of an operation is small
- concrete techniques
  - Aggregate Analysis
  - Potential Method

Aggregate Analysis

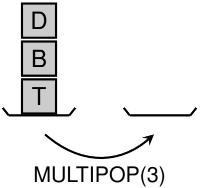
- Determine an upper bound $T(n)$ for the total cost of any sequence of n operations
- amortized cost of each operation is the average $\frac{T(n)}{n}$

Even though operations may be of different types/costs

Stack: Aggregate Analysis

Simple Worst-Case Bound:

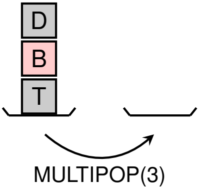
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



Stack: Aggregate Analysis

Simple Worst-Case Bound:

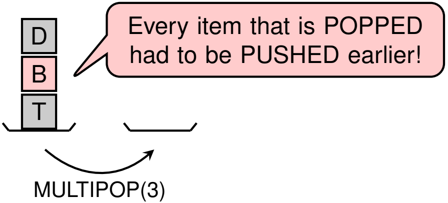
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



Stack: Aggregate Analysis

Simple Worst-Case Bound:

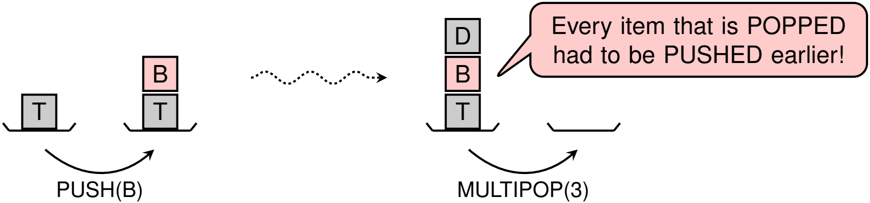
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



Stack: Aggregate Analysis

Simple Worst-Case Bound:

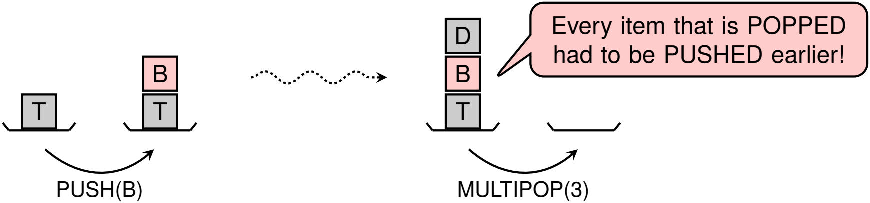
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



Stack: Aggregate Analysis

Simple Worst-Case Bound:

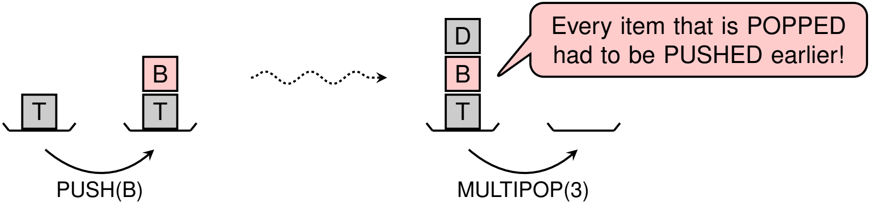
- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



Stack: Aggregate Analysis

Simple Worst-Case Bound:

- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**

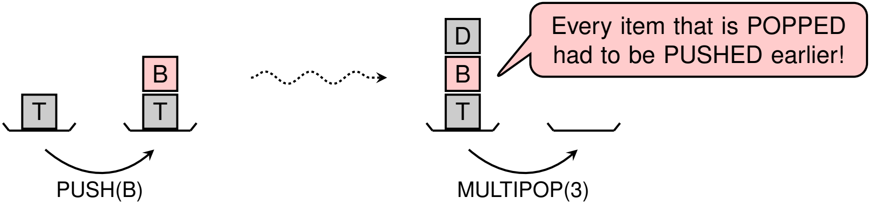


$$T(n) \leq$$

Stack: Aggregate Analysis

Simple Worst-Case Bound:

- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**

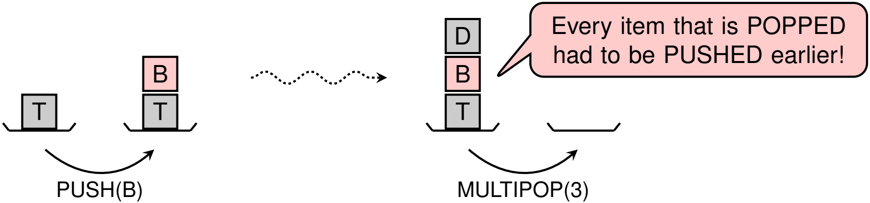


$$T(n) \leq T_{POP}(n) + T_{PUSH}(n)$$

Stack: Aggregate Analysis

Simple Worst-Case Bound:

- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



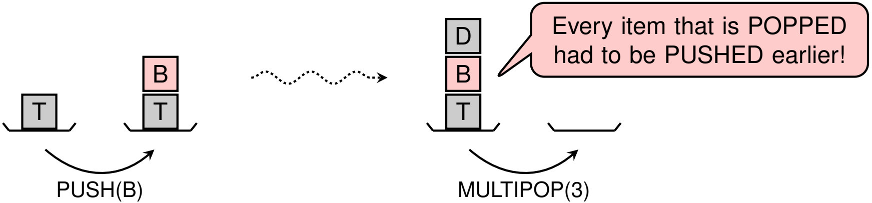
MULTIPOP(k) contributes $\min\{k, |S|\}$ to $T_{POP}(n)$

$$T(n) \leq T_{POP}(n) + T_{PUSH}(n)$$

Stack: Aggregate Analysis

Simple Worst-Case Bound:

- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**

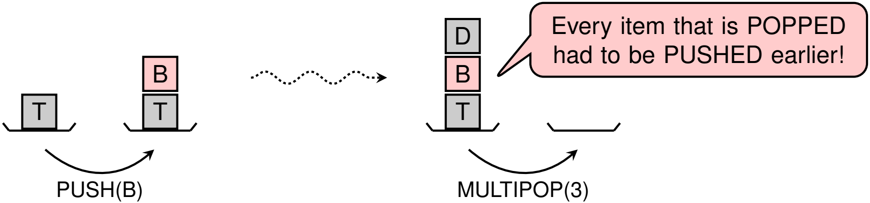


$$T(n) \leq T_{POP}(n) + T_{PUSH}(n) \leq 2 \cdot T_{PUSH}(n)$$

Stack: Aggregate Analysis

Simple Worst-Case Bound:

- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**

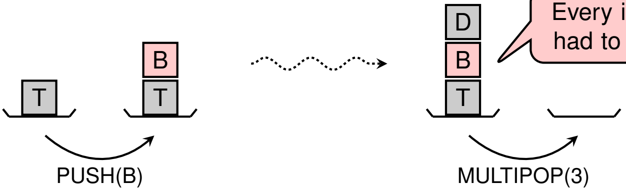


$$T(n) \leq T_{POP}(n) + T_{PUSH}(n) \leq 2 \cdot T_{PUSH}(n) \leq 2 \cdot n.$$

Stack: Aggregate Analysis

Simple Worst-Case Bound:

- largest cost of an operation: n
- cost is at most $n \cdot n = n^2$ **(correct, but not tight!)**



Every item that is POPPED had to be PUSHED earlier!

$$T(n) \leq T_{POP}(n) + T_{PUSH}(n) \leq 2 \cdot T_{PUSH}(n) \leq 2 \cdot n.$$

Aggregate Analysis: The amortized cost per operation is $\frac{T(n)}{n} \leq 2$

Second Technique: Potential Method

Potential Method

Second Technique: Potential Method

Potential Method

- allow different amortized costs

$\rightsquigarrow$ store **(fictitious)** credit in the data structure to cover up for expensive operations

Second Technique: Potential Method

Potential Method

- allow different amortized costs

$\leadsto$ store **(fictitious)** credit in the data structure to cover up for expensive operations

Potential of a data structure can be also thought of as

- amount of potential energy stored
- distance from an ideal state

Second Technique: Potential Method

Potential Method

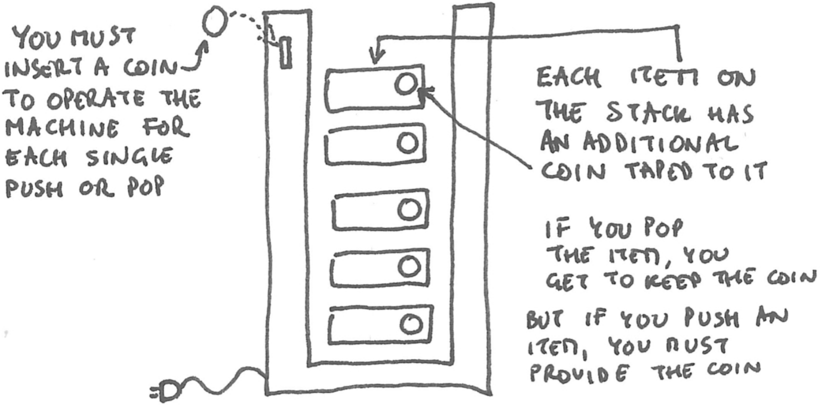
- allow different amortized costs

$\rightsquigarrow$ store **(fictitious)** credit in the data structure to cover up for expensive operations

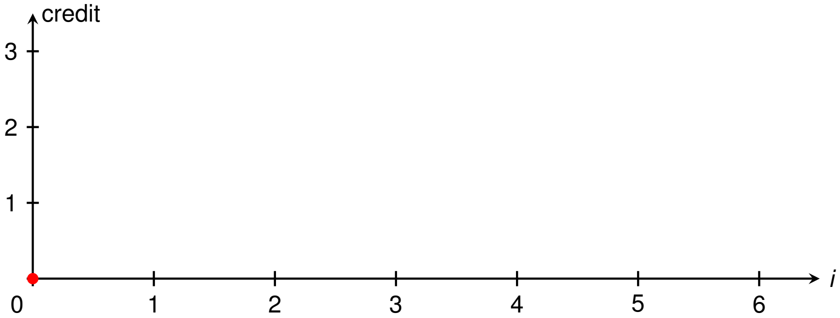
Potential of a data structure can be also thought of as

- amount of potential energy stored
- distance from an ideal state

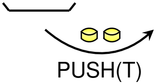
Stack as a coin-operated machine (p. 83)

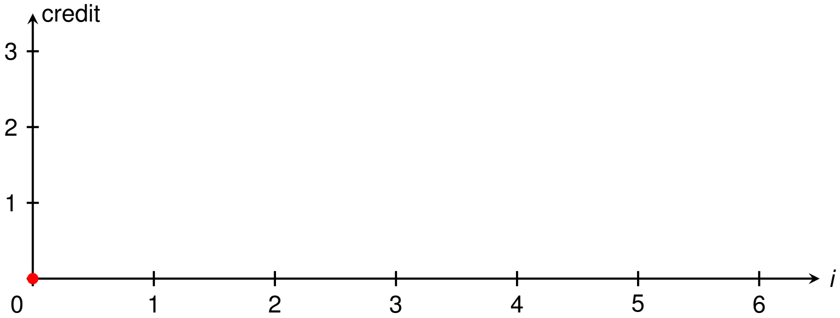


Stack and Coins

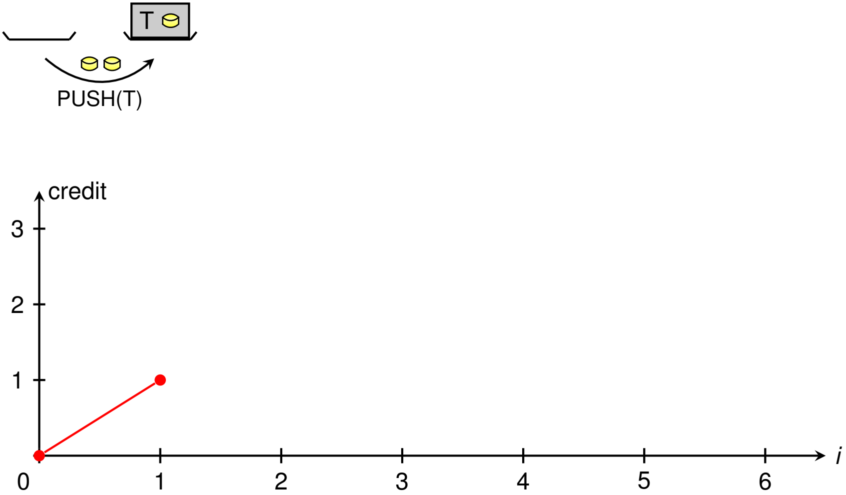


Stack and Coins

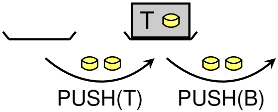


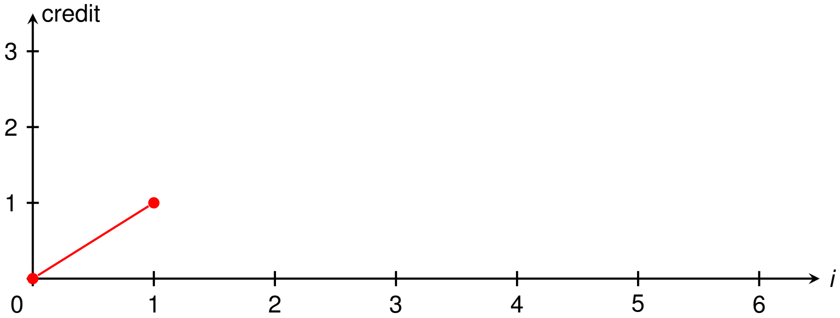


Stack and Coins

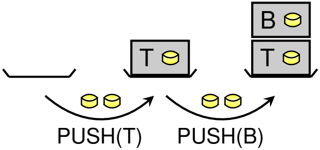


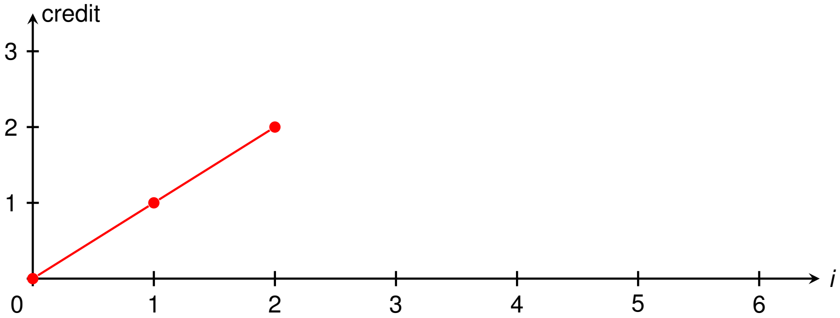
Stack and Coins



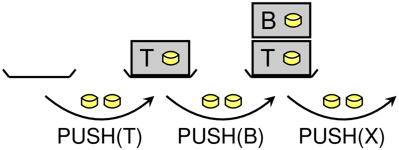


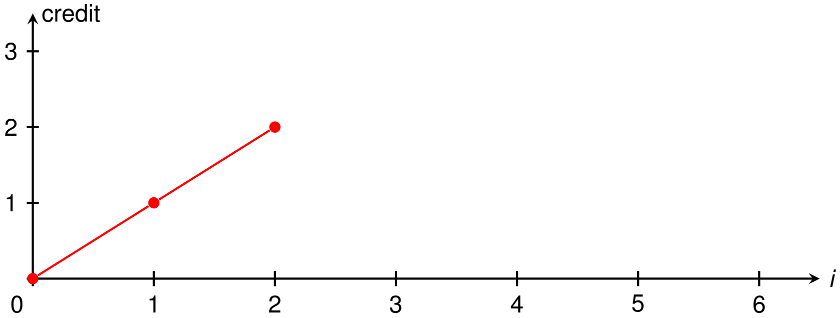
Stack and Coins



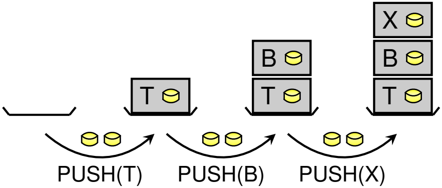


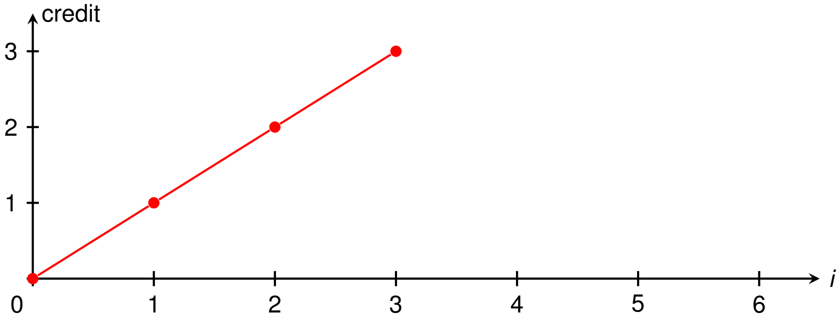
Stack and Coins



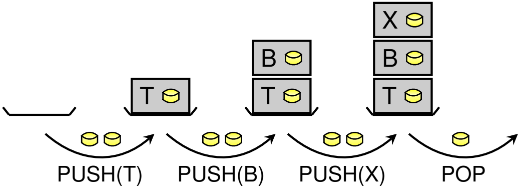


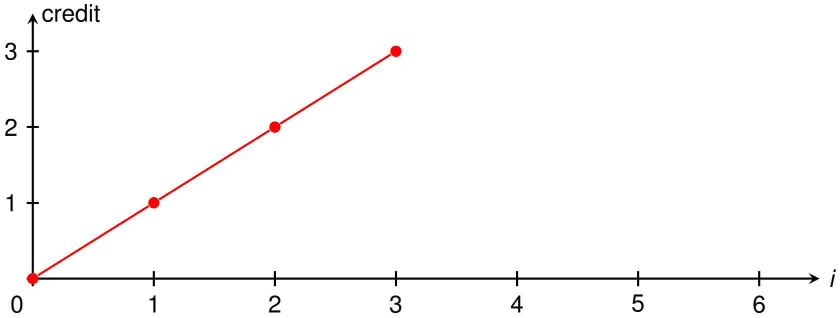
Stack and Coins



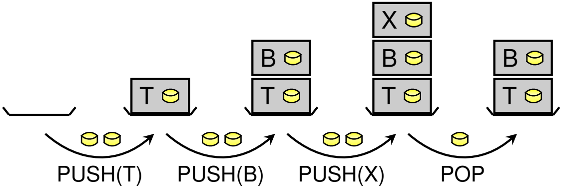


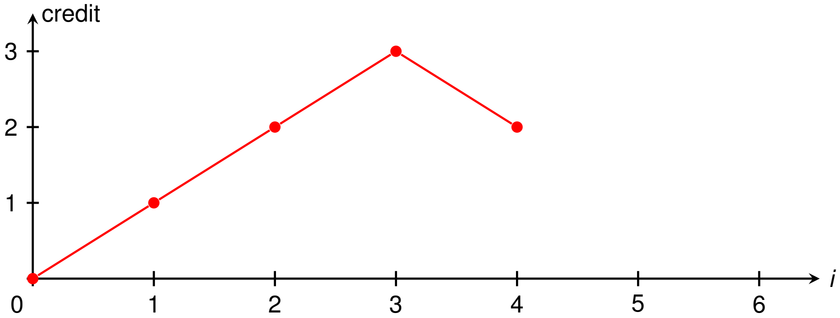
Stack and Coins



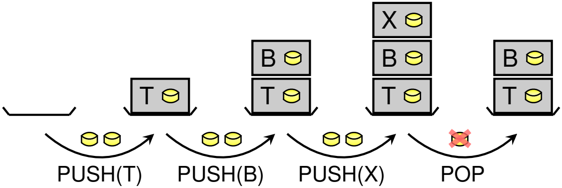


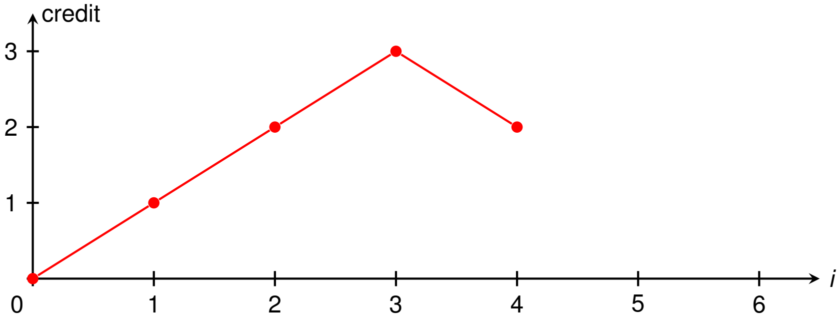
Stack and Coins



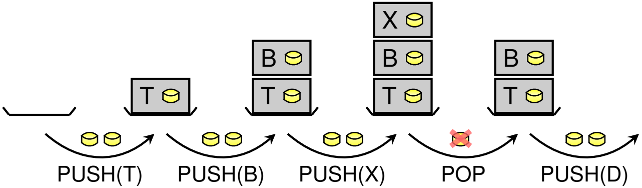


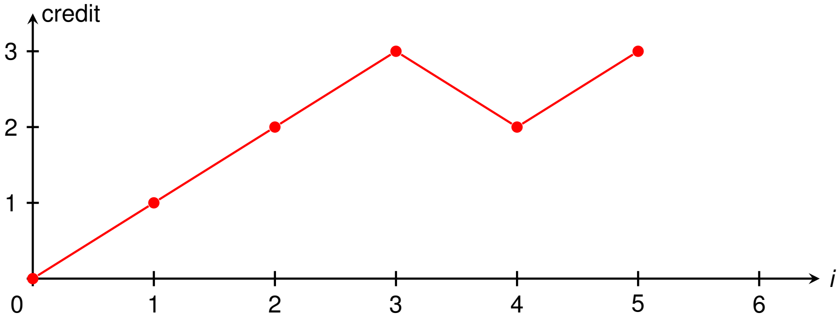
Stack and Coins



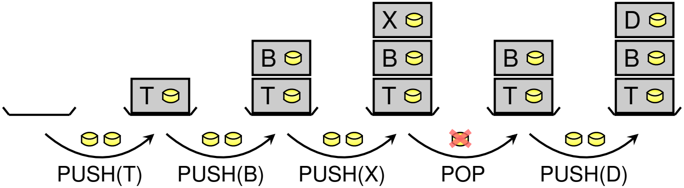


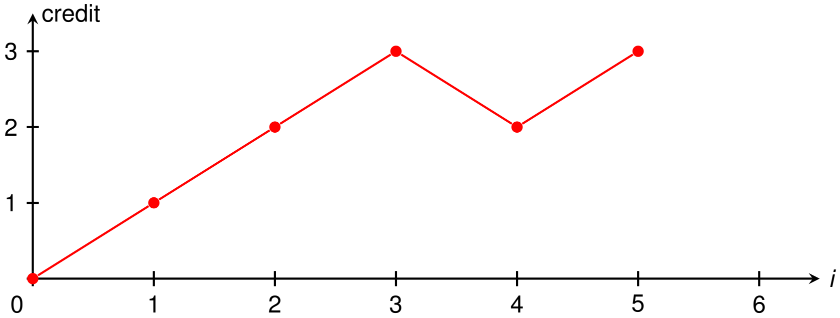
Stack and Coins



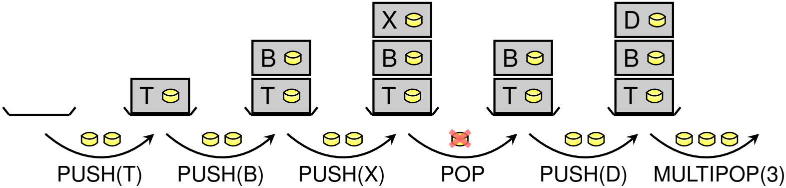


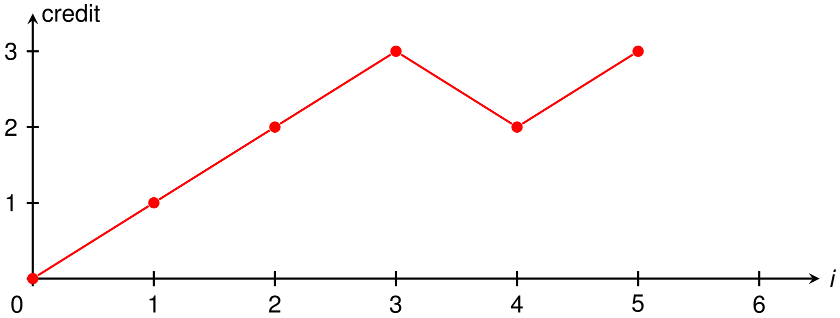
Stack and Coins



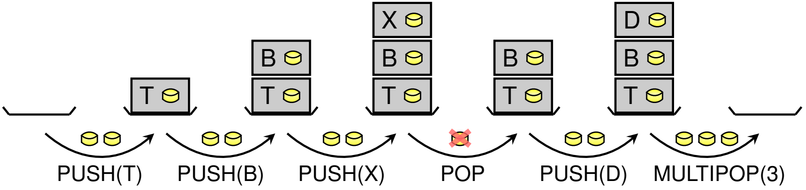


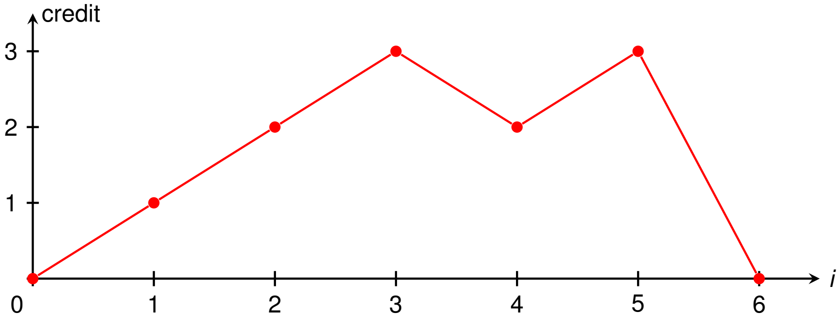
Stack and Coins



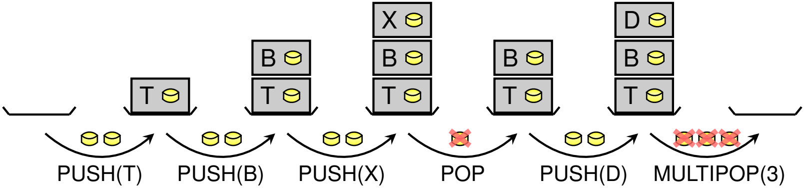


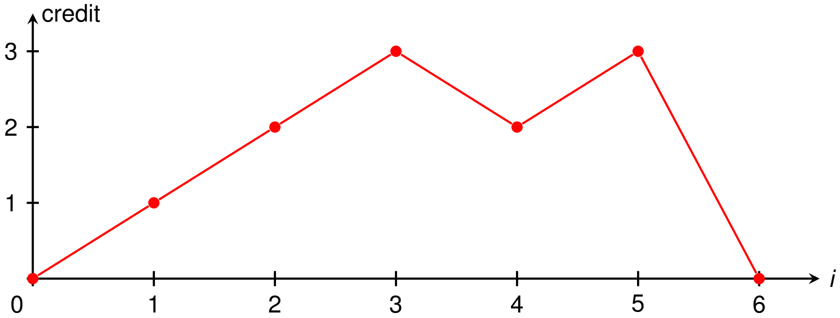
Stack and Coins



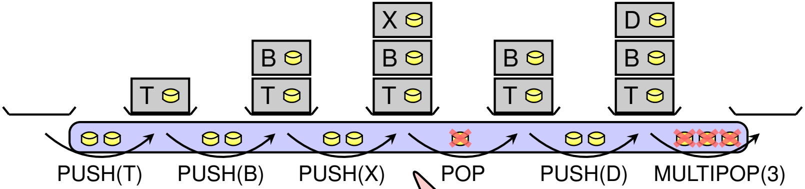


Stack and Coins

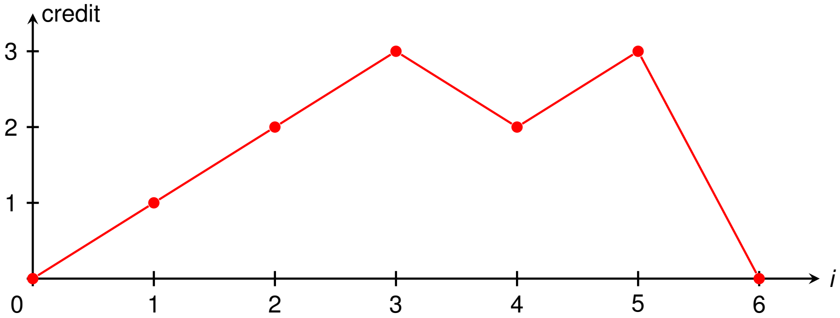




Stack and Coins



Every operation costs at most **two coins**!



Potential Method in Detail

- c_i is the actual cost of operation i

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i

$c_i < \widehat{c}_i$, $c_i = \widehat{c}_i$ or $c_i > \widehat{c}_i$ are all possible!

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c_i}$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

Function that maps states of the data structure to some value

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + \left(\Phi_i - \Phi_{i-1}\right)$$

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1})$$

- PUSH(): $c_i = 1$
- POP: $c_i = 1$

- PUSH(): $\Phi_i - \Phi_{i-1} = 1$
- POP: $\Phi_i - \Phi_{i-1} = -1$

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1})$$

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + \left(\Phi_i - \Phi_{i-1}\right)$$

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) =$$

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + \left(\Phi_i - \Phi_{i-1}\right)$$

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0$$

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + \left(\Phi_i - \Phi_{i-1}\right)$$

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n$$

Potential Method in Detail

- c_i is the actual cost of operation i
- $\widehat{c}_i$ is the amortized cost of operation i
- Φ_i is the potential stored after operation i ($\Phi_0 = 0$)

$$\widehat{c}_i = c_i + \left(\Phi_i - \Phi_{i-1}\right)$$

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi_i - \Phi_{i-1}\right) = \sum_{i=1}^{n} c_i + \Phi_n$$

If $\Phi_n \geq 0$ for all n, sum of amortized costs is an upper bound for the sum of actual costs!

Stack: Analysis via Potential Method

$\Phi_i =$

Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $(= \#$ coins$)$

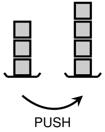
PUSH

Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

- actual cost: $c_i = 1$

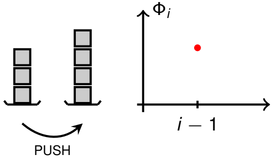


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} =$

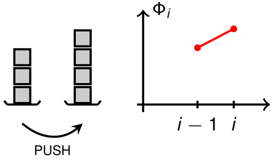


Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $(= \#$ coins$)$

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$

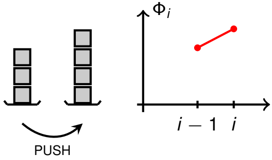


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c}_i =$

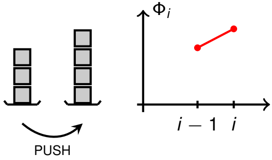


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) =$

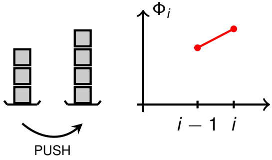


Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $(= \#$ coins$)$

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

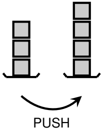


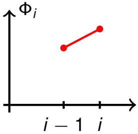
Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$





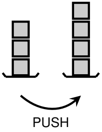
POP

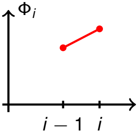
Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $(= \#$ coins$)$

PUSH

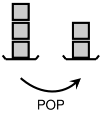
- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$





POP

- $c_i = 1$

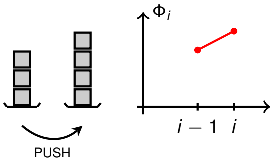


Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $(= \#$ coins$)$

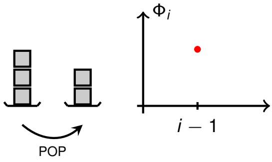
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} =$

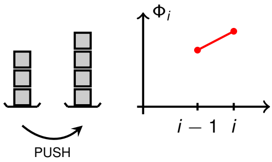


Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $(= \#$ coins$)$

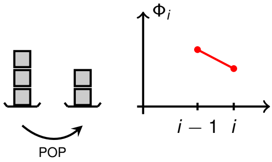
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$

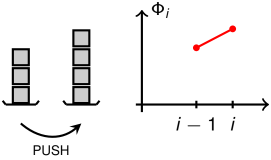


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

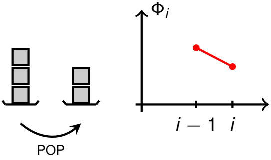
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) =$

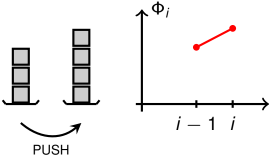


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

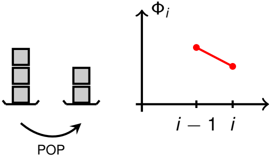
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$

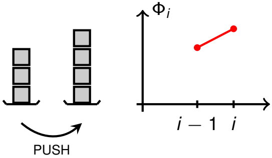


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

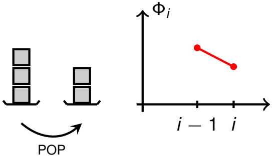
- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$

Stack is non-empty!

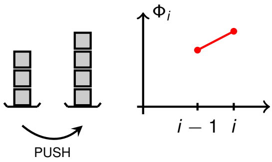


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

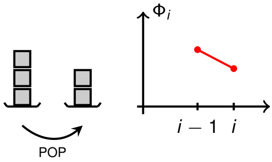
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



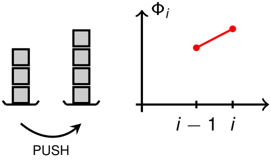
MULTIPOP(k)

Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $\;(= \#$ coins$)$

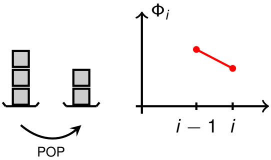
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



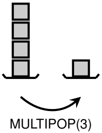
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$

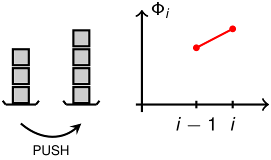


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

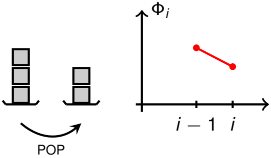
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



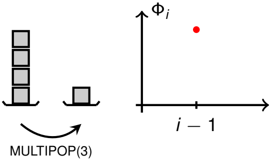
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$
- $\Phi_i - \Phi_{i-1} =$

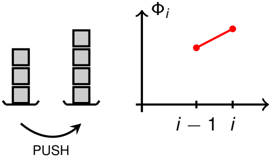


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

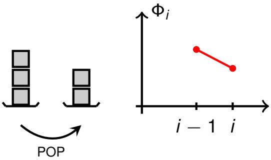
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



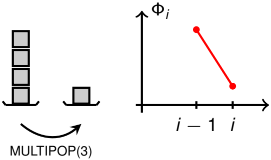
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$
- $\Phi_i - \Phi_{i-1} = -\min\{k, |S|\}$

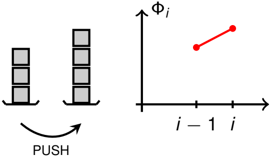


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

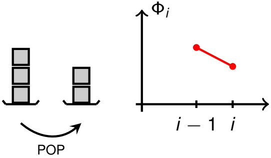
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



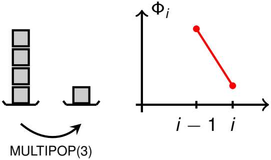
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$
- $\Phi_i - \Phi_{i-1} = -\min\{k, |S|\}$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) =$

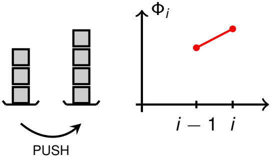


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

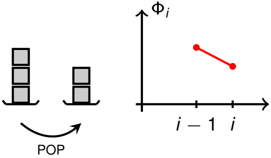
PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



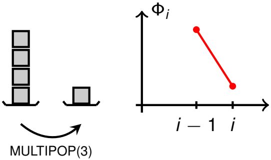
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$
- $\Phi_i - \Phi_{i-1} = -\min\{k, |S|\}$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = \min\{k, |S|\} - \min\{k, |S|\} = 0$

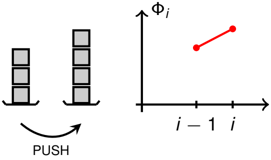


Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation (= # coins)

PUSH

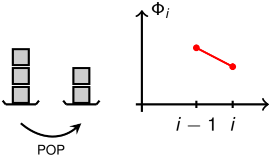
- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



Amortized Cost $\leq 2 \Rightarrow T(n) \leq 2n$

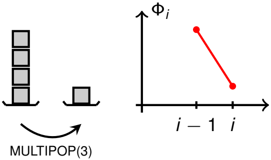
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$
- $\Phi_i - \Phi_{i-1} = -\min\{k, |S|\}$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = \min\{k, |S|\} - \min\{k, |S|\} = 0$

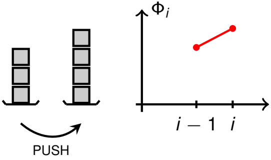


Stack: Analysis via Potential Method

$\Phi_i = \#$ objects in the stack after ith operation $\;(= \#$ coins$)$

PUSH

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

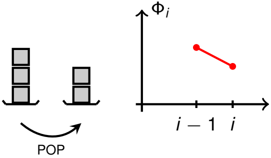


Amortized Cost $\leq 2 \Rightarrow T(n) \leq 2n$

$n/2$ PUSH, $n/2$ POP $\Rightarrow T(n) \leq n$

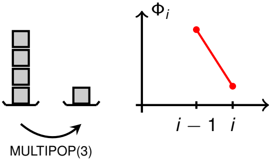
POP

- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$



MULTIPOP(k)

- $c_i = \min\{k, |S|\}$
- $\Phi_i - \Phi_{i-1} = -\min\{k, |S|\}$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = \min\{k, |S|\} - \min\{k, |S|\} = 0$



Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	0	1	1	11

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **`INC`**

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	0	1	1	11

INC $\downarrow$

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **`INC`**
  - increases the counter by one

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	11

$\downarrow$ INC

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **`INC`**
  - increases the counter by one

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	11

$\big\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	12

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **`INC`**
    - increases the counter by one

```
INC(A)
i = 0
while i < k and A[i]==1
    A[i] = 0
    i = i + 1
A[i] = 1
```

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	0	1	1	11

$\big\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	1	0	0	12

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **INC**
  - increases the counter by one
  - total cost: ??

```
INC(A)
i = 0
while i < k and A[i]==1
    A[i] = 0
    i = i + 1
A[i] = 1
```

$A[3]\ A[2]\ A[1]\ A[0]$

$\boxed{1}\ \boxed{0}\ \boxed{1}\ \boxed{1}$ 11

$\big\downarrow$ INC

$A[3]\ A[2]\ A[1]\ A[0]$

$\boxed{1}\ \boxed{1}\ \boxed{0}\ \boxed{0}$ 12

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **`INC`**
  - increases the counter by one
  - total cost: $\leq k$

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	11

$\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	12

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **`INC`**
  - increases the counter by one
  - total cost: $\leq k$

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	0	1	1	11

$\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	1	0	0	12

What is the total cost of a sequence of n INC operations?

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **INC**
  - increases the counter by one
  - total cost: $\leq k$

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	0	1	1	11

$\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	1	0	0	12

What is the total cost of a sequence of n INC operations?

Simple Worst-Case Bound:

- largest cost of an operation: k
- cost is at most $n \cdot k$

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: **INC**
  - increases the counter by one
  - total cost: $\leq k$

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	0	1	1	11

$\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
1	1	0	0	12

What is the total cost of a sequence of n INC operations?

Simple Worst-Case Bound:

- largest cost of an operation: k
- cost is at most $n \cdot k$ **(correct, but not tight!)**

Second Example: Binary Counter

Binary Counter

- Array $A[k-1], A[k-2], \ldots, A[0]$ of k bits
- Use array for counting from 0 to $2^k - 1$
- only operation: `INC`
  - increases the counter by one
  - total cost: $\cancel{\leq k}$ number of flips (smallest index of a zero)

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	11

$\downarrow$ INC

$A[3]$	$A[2]$	$A[1]$	$A[0]$	
$\boxed{1}$	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	12

What is the total cost of a sequence of n INC operations?

Simple Worst-Case Bound:

- largest cost of an operation: k
- cost is at most $n \cdot k$ **(correct, but not tight!)**

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Incrementing a Binary Counter

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

$$T(n) \leq$$

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

$$T(n) \leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor$$

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

$$T(n) \leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i}$$

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

$$T(n) \leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} = n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right)$$

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

$$T(n) \leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} = n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right) \leq 2 \cdot n.$$

Incrementing a Binary Counter: Aggregate Analysis

Counter Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total Cost
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	3
3	0	0	1	1	4
4	0	1	0	0	7
5	0	1	0	1	8
6	0	1	1	0	10
7	0	1	1	1	11

- Bit $A[i]$ is only flipped every 2^i increments
- In a sequence of n increments from 0, bit $A[i]$ is flipped $\lfloor \frac{n}{2^i} \rfloor$ times

Aggregate Analysis: The amortized cost per operation is $\frac{T(n)}{n} \leq 2$.

$$T(n) \leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} = n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right) \leq 2 \cdot n.$$

Binary Counter: Analysis via Potential Function

$$\Phi_i =$$

Binary Counter: Analysis via Potential Function

$\Phi_i = \#$ ones in the binary representation of i

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i = \#$ ones in the binary representation of i

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i = \#$ ones in the binary representation of i

Increment without Carry-Over

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i = \#$ ones in the binary representation of i

Increment without Carry-Over

- actual cost: $c_i = 1$

1 1 0 0

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

Increment without Carry-Over

- actual cost: $c_i = 1$

$$\begin{array}{cccc} 1 & 1 & 0 & \boxed{0} \end{array}$$
$$\Big\downarrow \text{INC}$$
$$\begin{array}{cccc} 1 & 1 & 0 & 1 \end{array}$$

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

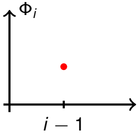
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} =$

1 1 0 $\boxed{0}$

$\downarrow$ INC

1 1 0 1



Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

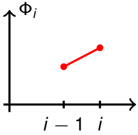
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$

1 1 0 0

↓ INC

1 1 0 1



Binary Counter: Analysis via Potential Function

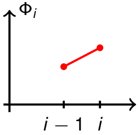
$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c}_i =$

$$\begin{array}{cccc} 1 & 1 & 0 & \boxed{0} \\ & & \big\downarrow \text{INC} & \\ 1 & 1 & 0 & 1 \end{array}$$



Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

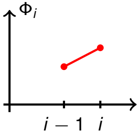
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) =$

1 1 0 0

↓ INC

1 1 0 1



Binary Counter: Analysis via Potential Function

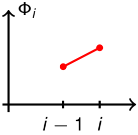
$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i =$ # ones in the binary representation of i

Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

$$\begin{array}{c} 1\ 1\ 0\ \boxed{0} \\ \Big\downarrow \text{INC} \\ 1\ 1\ 0\ 1 \end{array}$$



Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

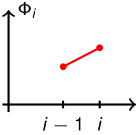
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

$$\begin{array}{cccc} 1 & 1 & 0 & \boxed{0} \end{array}$$

$$\downarrow \text{INC}$$

$$\begin{array}{cccc} 1 & 1 & 0 & 1 \end{array}$$



Increment with Carry-Over

Binary Counter: Analysis via Potential Function

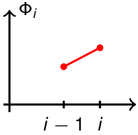
$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ & & \downarrow \text{INC} & \\ 1 & 1 & 0 & 1 \end{array}$$



0 1 1 1

Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

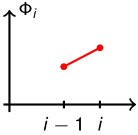
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

1 1 0 0

↓ INC

1 1 0 1



Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)

0 1 1 1

↓ INC

1 0 0 0

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i =$ # ones in the binary representation of i

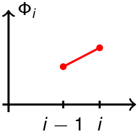
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

1 1 0 0

$\downarrow$ INC

1 1 0 1



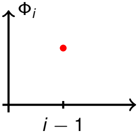
Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} =$

0 1 1 1

$\downarrow$ INC

1 0 0 0



Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i = \#$ ones in the binary representation of i

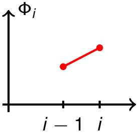
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

1 1 0 0

↓ INC

1 1 0 1



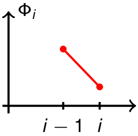
Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$

0 1 1 1

↓ INC

1 0 0 0



Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i =$ # ones in the binary representation of i

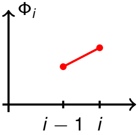
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

1 1 0 0

$\downarrow$ INC

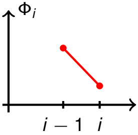
1 1 0 1



0 1 1 1

$\downarrow$ INC

1 0 0 0



Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) =$

Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i =$ # ones in the binary representation of i

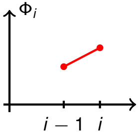
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

1 1 0 0

INC

1 1 0 1



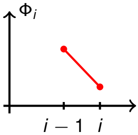
Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + x - x + 1$

0 1 1 1

INC

1 0 0 0



Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

Φ_i = # ones in the binary representation of i

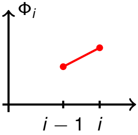
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$

1 1 0 0

↓ INC

1 1 0 1



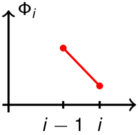
Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + x - x + 1 \ = 2$

0 1 1 1

↓ INC

1 0 0 0



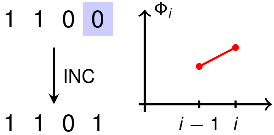
Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\quad \Phi_i \geq 0$ ✓

$\Phi_i =$ # ones in the binary representation of i

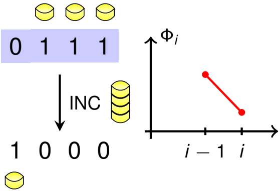
Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + x - x + 1 \ = 2$



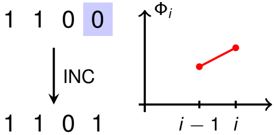
Binary Counter: Analysis via Potential Function

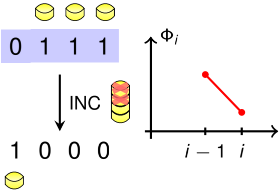
$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i =$ # ones in the binary representation of i

Increment without Carry-Over

- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$





Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + x - x + 1 = 2$

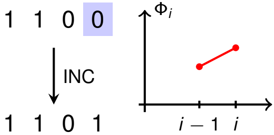
Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

$\Phi_i = \#$ ones in the binary representation of i

Increment without Carry-Over

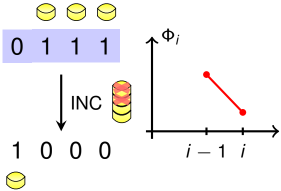
- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



Amortized Cost $= 2 \Rightarrow T(n) \leq 2n$

Increment with Carry-Over

- $c_i = x + 1$, (x lowest index of a zero)
- $\Phi_i - \Phi_{i-1} = -x + 1$
- $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + x - x + 1 \; = 2$



Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Aggregate Analysis

Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Aggregate Analysis

- Determine an absolute upper bound $T(n)$

Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

E.g. by bounding the number of expensive operations

Aggregate Analysis

- Determine an absolute upper bound $T(n)$

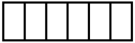
Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Aggregate Analysis

- Determine an absolute upper bound $T(n)$
- every operation has amortized cost $\frac{T(n)}{n}$

$T(n)$ 

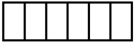
Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Aggregate Analysis

- Determine an absolute upper bound $T(n)$
- every operation has amortized cost $\frac{T(n)}{n}$

$T(n)$ 

Potential Method

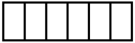
Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Aggregate Analysis

- Determine an absolute upper bound $T(n)$
- every operation has amortized cost $\frac{T(n)}{n}$

$T(n)$ 

Potential Method

- use savings from cheap operations to compensate for expensive ones

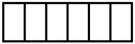
Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

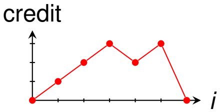
Aggregate Analysis

- Determine an absolute upper bound $T(n)$
- every operation has amortized cost $\frac{T(n)}{n}$

$T(n)$ 

Potential Method

- use savings from cheap operations to compensate for expensive ones



Summary

Amortized Analysis

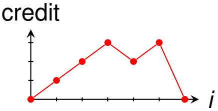
- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

Aggregate Analysis

- Determine an absolute upper bound $T(n)$
- every operation has amortized cost $\frac{T(n)}{n}$

Potential Method

- use savings from cheap operations to compensate for expensive ones
- operations may have different amortized cost



Summary

Amortized Analysis

- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

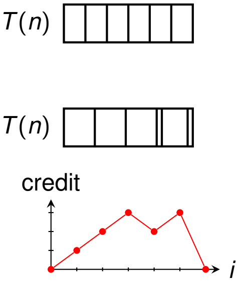
Aggregate Analysis

- Determine an absolute upper bound $T(n)$
- every operation has amortized cost $\frac{T(n)}{n}$

Full power of this method will become clear later!

Potential Method

- use savings from cheap operations to compensate for expensive ones
- operations may have different amortized cost



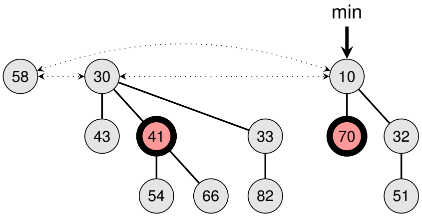
Next Lecture: Fibonacci Heap

Operation	Binomial heap worst-case cost
MAKE-HEAP	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$
MINIMUM	$\mathcal{O}(\log n)$
EXTRACT-MIN	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(\log n)$
DECREASE-KEY	$\mathcal{O}(\log n)$
DELETE	$\mathcal{O}(\log n)$

Next Lecture: Fibonacci Heap

Operation	Binomial heap worst-case cost	Fibonacci heap amortized cost
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Crucial for many applications including shortest paths and minimum spanning trees!



5.2 Fibonacci Heaps

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Priority Queues Overview

Operation	Linked list	Binary heap	Binomial heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Priority Queues Overview

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap	Fibonacci heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap: $k/2$ DECREASE-KEY
+ $k/2$ INSERT

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap: $k/2$ DECREASE-KEY
+ $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$

$\Rightarrow \sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$

$\Rightarrow \sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$

Fibonacci Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$

$\Rightarrow \sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$

Fibonacci Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $\widehat{c}_1 = \widehat{c}_2 = \cdots = \widehat{c}_k = \mathcal{O}(1)$

Binomial Heap vs. Fibonacci Heap: Costs

Operation	Binomial heap **actual cost**	Fibonacci heap **amortized cost**
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

n is the number of items in the heap when the operation is performed.

Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$

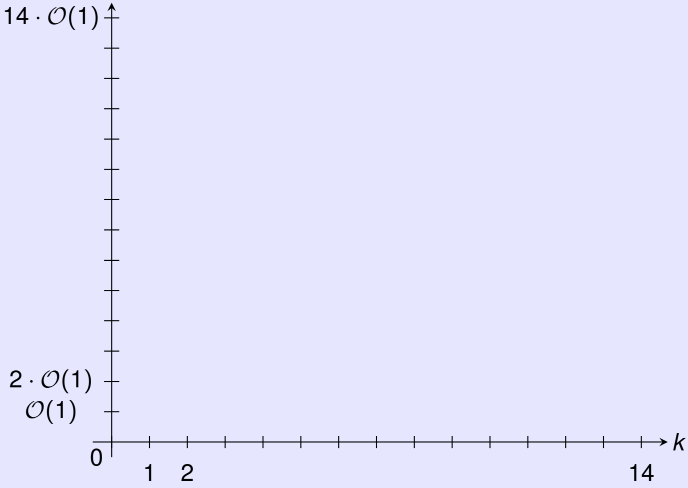
$\Rightarrow \sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$

Fibonacci Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

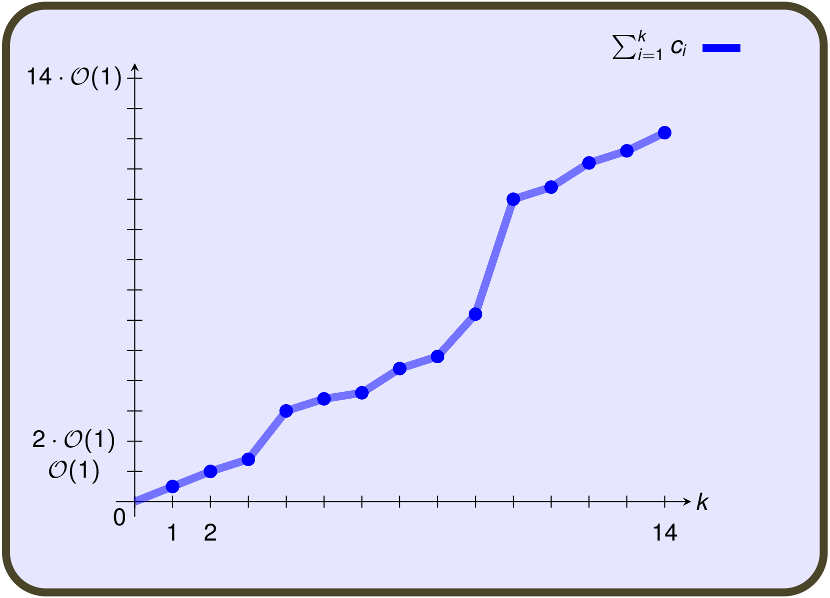
- $\widehat{c}_1 = \widehat{c}_2 = \cdots = \widehat{c}_k = \mathcal{O}(1)$

$\Rightarrow \sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c}_i = \mathcal{O}(k)$

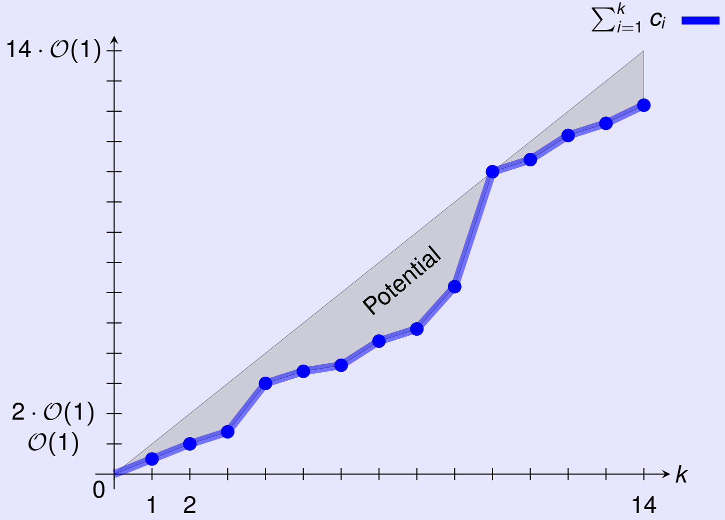
Actual vs. Amortized Cost



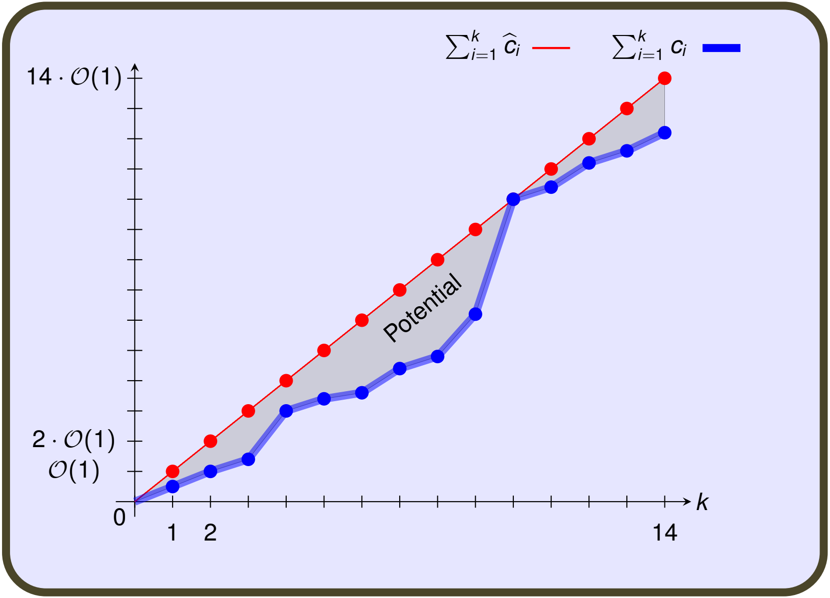
Actual vs. Amortized Cost



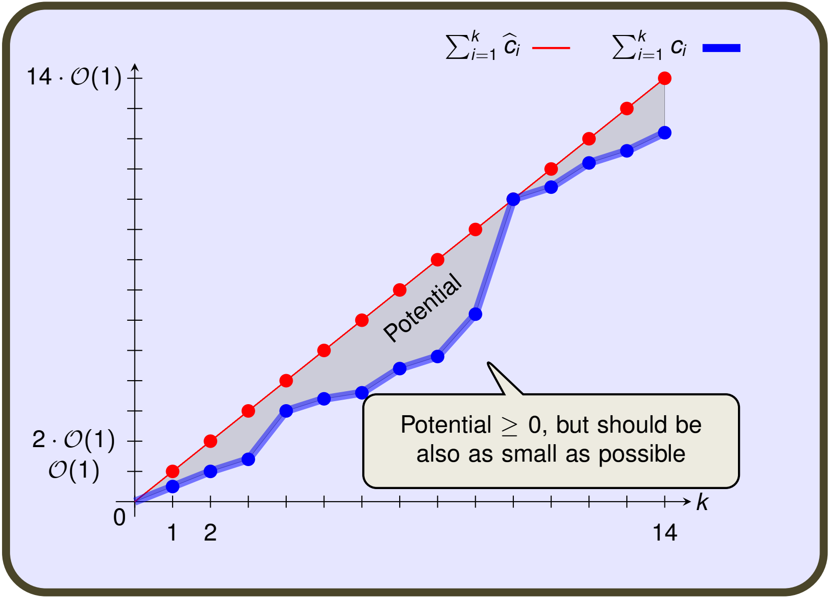
Actual vs. Amortized Cost



Actual vs. Amortized Cost



Actual vs. Amortized Cost



Outline

Structure

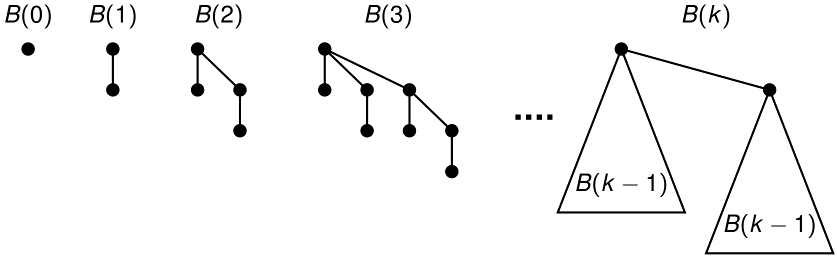
Operations

Glimpse at the Analysis

Amortized Analysis

Reminder: Binomial Heaps

Binomial Trees

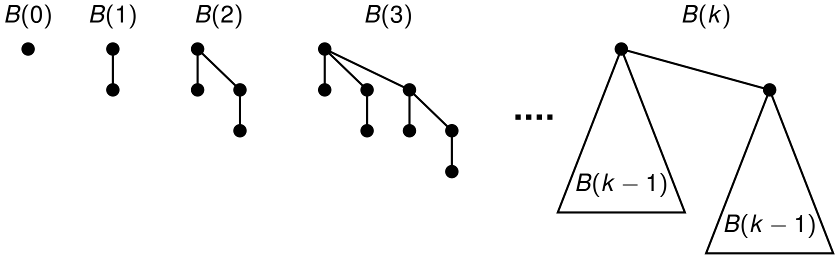


Binomial Heaps

- Binomial Heap is a collection of binomial trees of different orders, each of which obeys the heap property

Reminder: Binomial Heaps

Binomial Trees

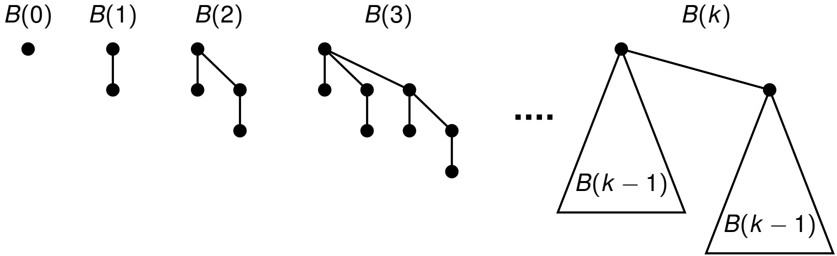


Binomial Heaps

- Binomial Heap is a collection of binomial trees of different orders, each of which obeys the heap property
- Operations:

Reminder: Binomial Heaps

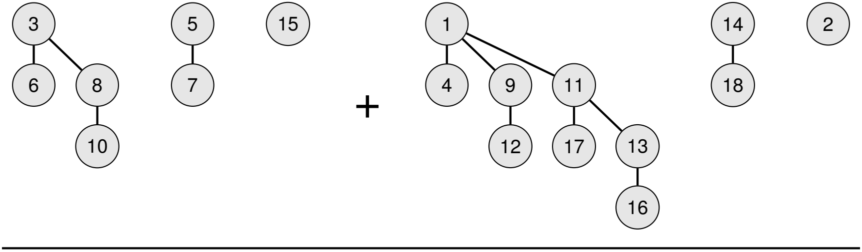
Binomial Trees



Binomial Heaps

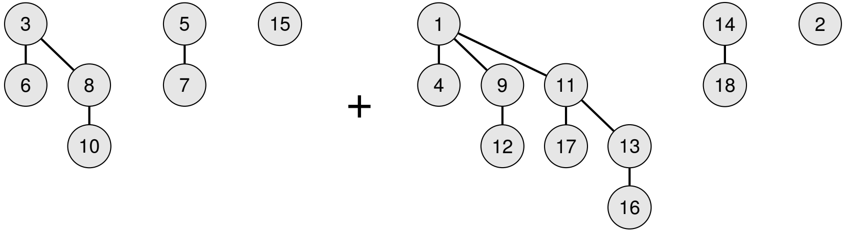
- Binomial Heap is a collection of binomial trees of different orders, each of which obeys the heap property
- Operations:
  - MERGE: Merge two binomial heaps using Binary Addition Procedure
  - INSERT: Add $B(0)$ and perform a MERGE
  - EXTRACT-MIN: Find tree with minimum key, cut it and perform a MERGE
  - DECREASE-KEY: The same as in a binary heap

Merging two Binomial Heaps



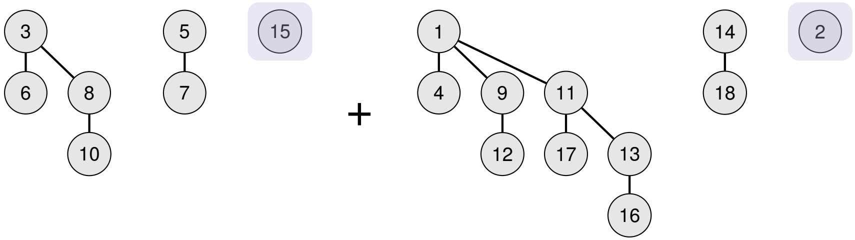
0	0	1	1	1		= 7
0	1	0	1	1		= 11
1	1	1	1			
1	0	0	1	0		= 18

Merging two Binomial Heaps



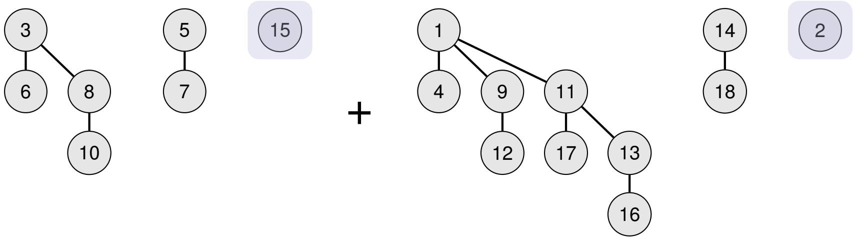
$$\begin{array}{ccccccl} & 0 & 0 & 1 & 1 & 1 & = 7 \\ & 0 & 1 & 0 & 1 & 1 & = 11 \\ & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\ \hline & 1 & 0 & 0 & 1 & 0 & = 18 \end{array}$$

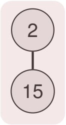
Merging two Binomial Heaps



$$
\begin{array}{cccccl}
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
{\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

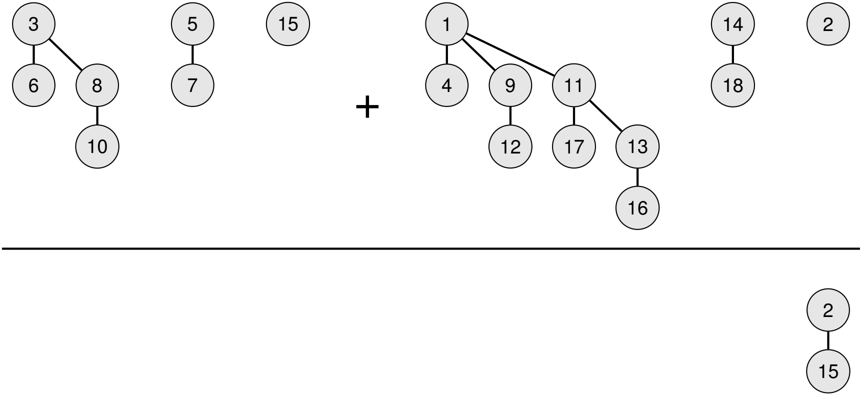
Merging two Binomial Heaps





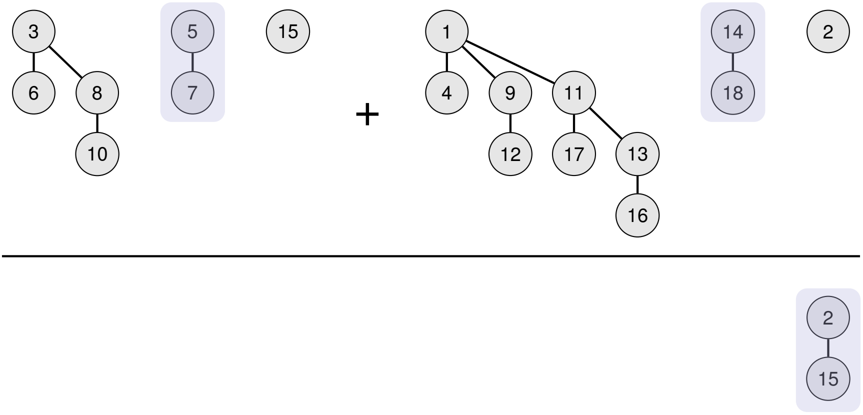
$$
\begin{array}{rrrrrl}
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
{\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

Merging two Binomial Heaps



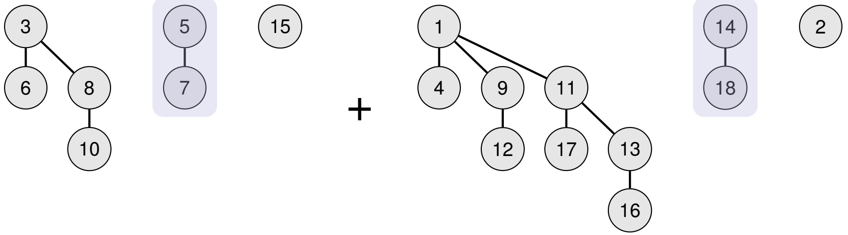
$$
\begin{array}{ccccccl}
 & 0 & 0 & 1 & 1 & 1 & = 7 \\
 & 0 & 1 & 0 & 1 & 1 & = 11 \\
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & 1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

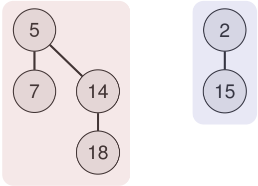
Merging two Binomial Heaps



$$
\begin{array}{ccccccl}
 & 0 & 0 & 1 & 1 & 1 & = 7 \\
 & 0 & 1 & 0 & 1 & 1 & = 11 \\
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & 1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

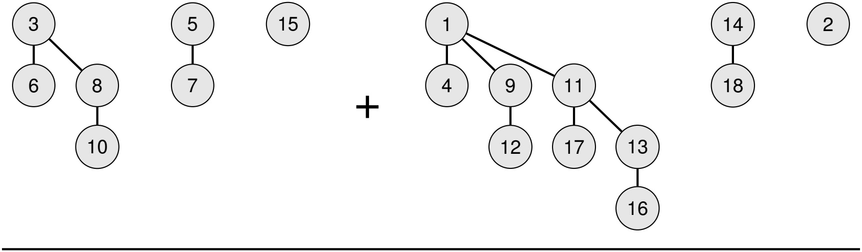
Merging two Binomial Heaps

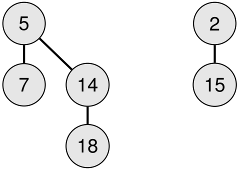




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

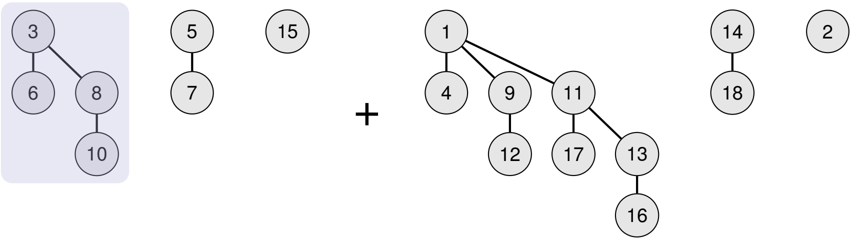
Merging two Binomial Heaps

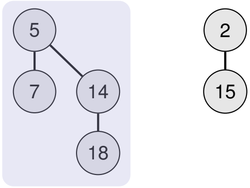




$$
\begin{array}{ccccccl}
 & 0 & 0 & 1 & 1 & 1 & = 7 \\
 & 0 & 1 & 0 & 1 & 1 & = 11 \\
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & 1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

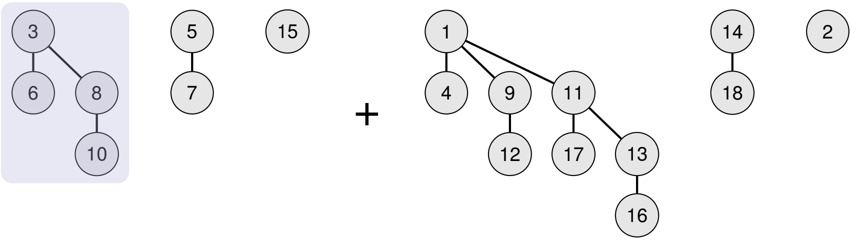
Merging two Binomial Heaps

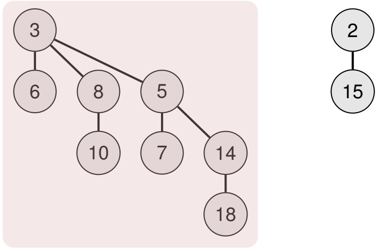




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

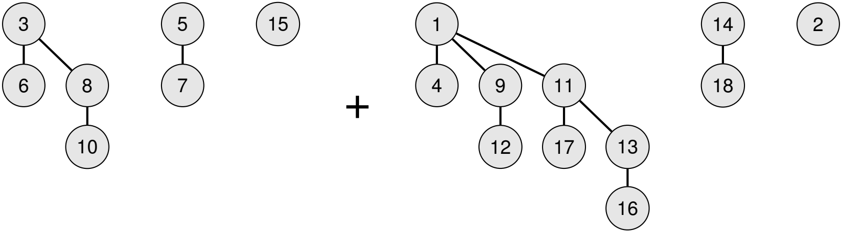
Merging two Binomial Heaps

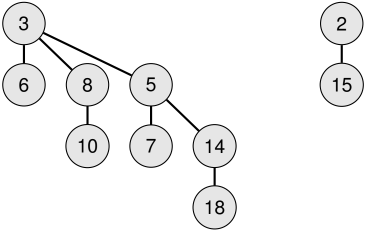




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

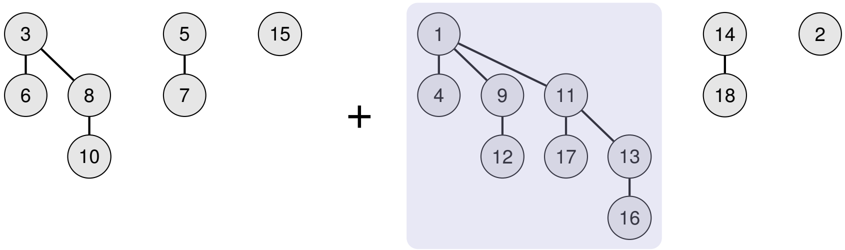
Merging two Binomial Heaps

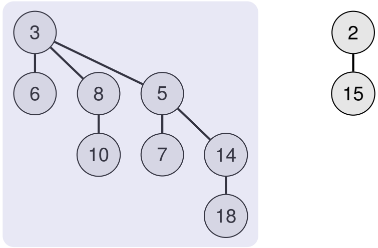




$$
\begin{array}{ccccccl}
 & 0 & 0 & 1 & 1 & 1 & = 7 \\
 & 0 & 1 & 0 & 1 & 1 & = 11 \\
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & 1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

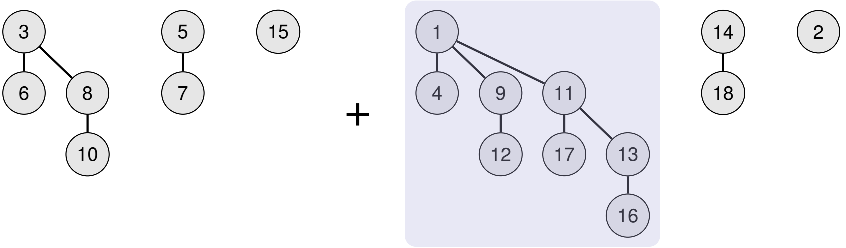
Merging two Binomial Heaps



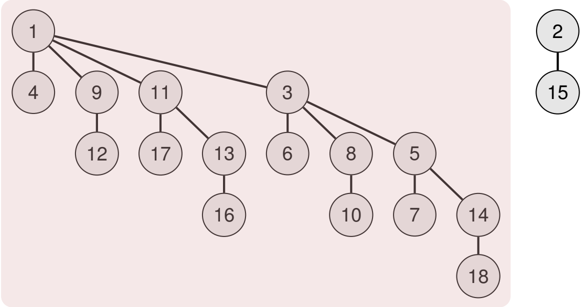


0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

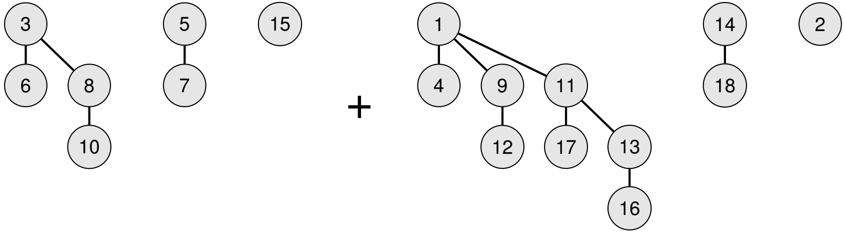
Merging two Binomial Heaps

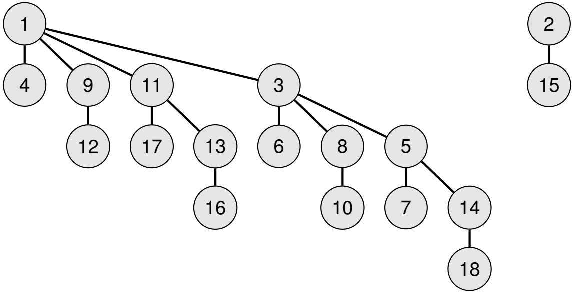


$$
\begin{array}{cccccl}
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
{\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$



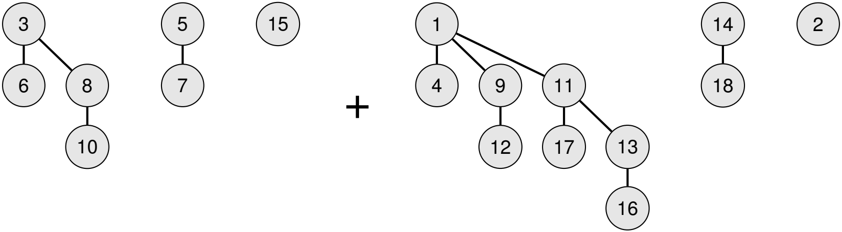
Merging two Binomial Heaps

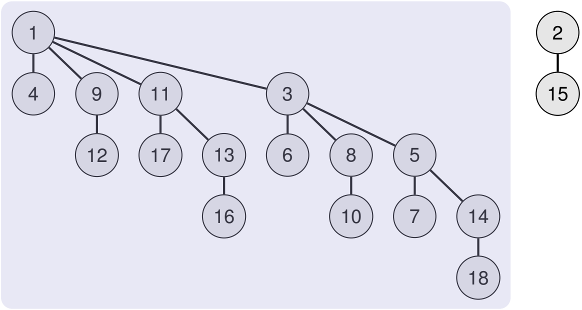




$$
\begin{array}{cccccl}
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
{\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
1 & 0 & 0 & 1 & 0 & = 18
\end{array}
$$

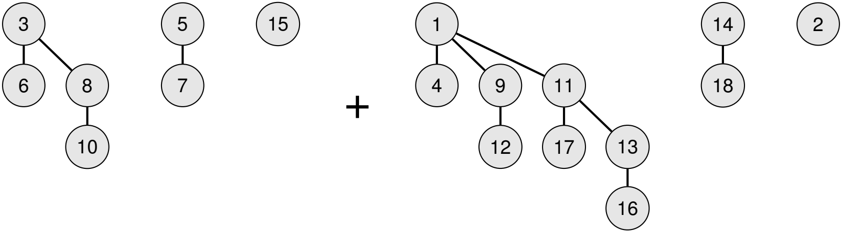
Merging two Binomial Heaps

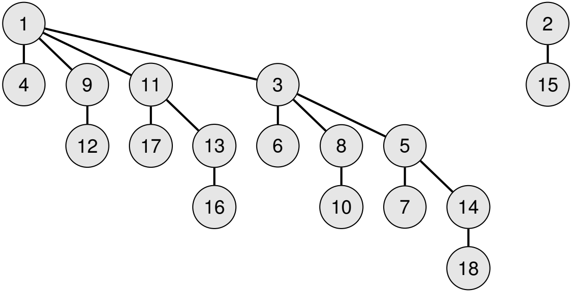




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

Merging two Binomial Heaps



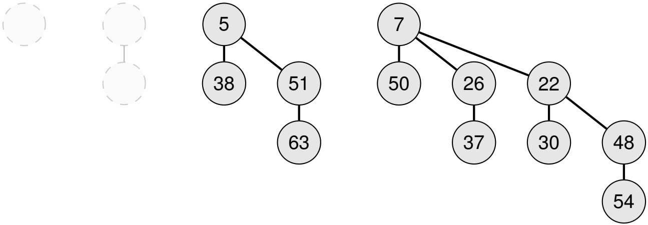


0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

Binomial Heap vs. Fibonacci Heap: Structure

Binomial Heap:

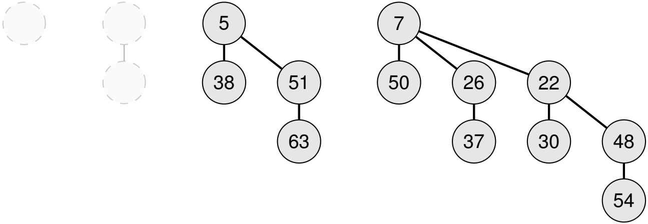
- consists of binomial trees, and every order appears at most once
- immediately tidy up after INSERT or MERGE



Binomial Heap vs. Fibonacci Heap: Structure

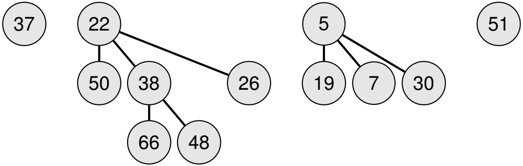
Binomial Heap:

- consists of binomial trees, and every order appears at most once
- immediately tidy up after INSERT or MERGE



Fibonacci Heap:

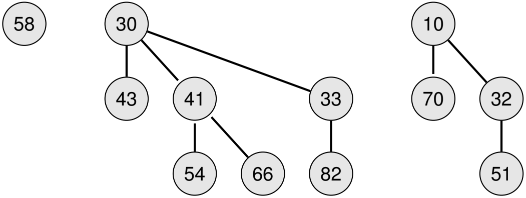
- forest of MIN-HEAPs
- lazily defer tidying up; do it on-the-fly when search for the MIN



Structure of Fibonacci Heaps

Fibonacci Heap

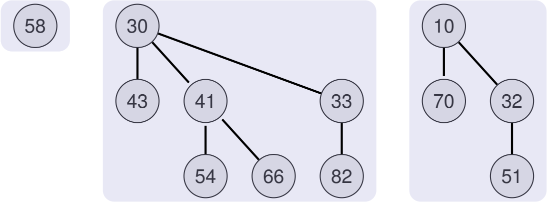
- Forest of MIN-HEAPs



Structure of Fibonacci Heaps

Fibonacci Heap

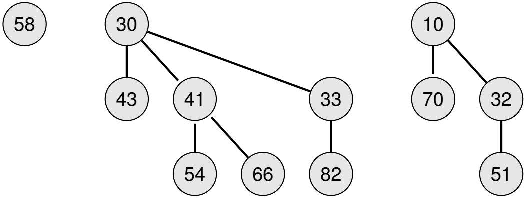
- Forest of MIN-HEAPs



Structure of Fibonacci Heaps

Fibonacci Heap

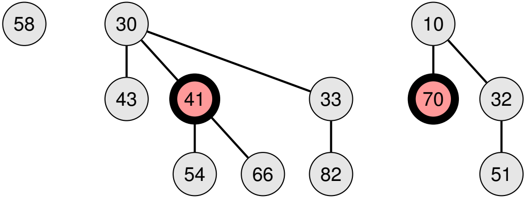
- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)



Structure of Fibonacci Heaps

Fibonacci Heap

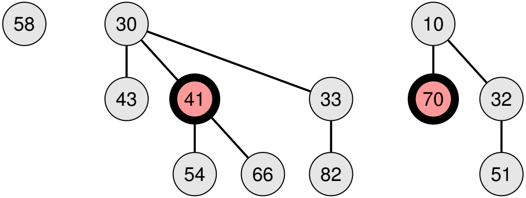
- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)



Structure of Fibonacci Heaps

Fibonacci Heap

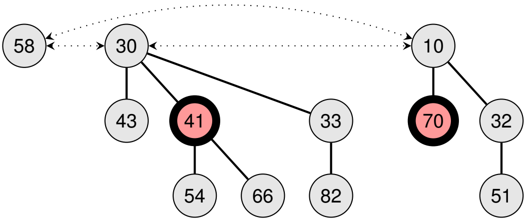
- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list



Structure of Fibonacci Heaps

Fibonacci Heap

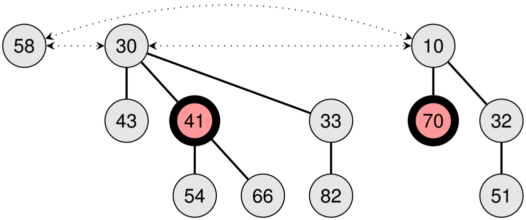
- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list



Structure of Fibonacci Heaps

Fibonacci Heap

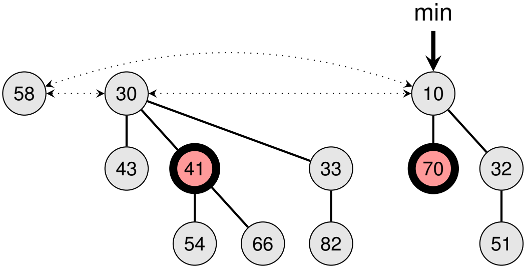
- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list
- Min-Pointer pointing to the smallest element



Structure of Fibonacci Heaps

Fibonacci Heap

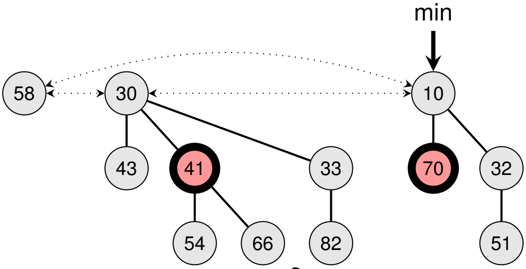
- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list
- Min-Pointer pointing to the smallest element



Structure of Fibonacci Heaps

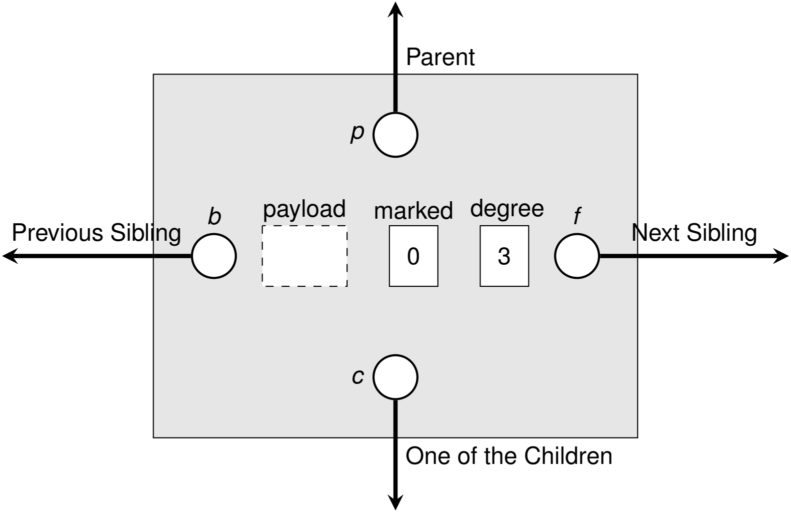
Fibonacci Heap

- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list
- Min-Pointer pointing to the smallest element

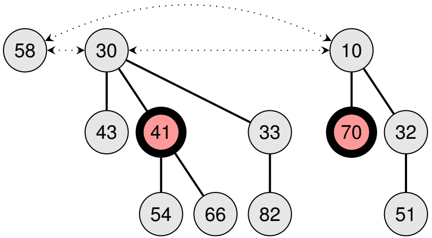


How do we implement a Fibonacci Heap?

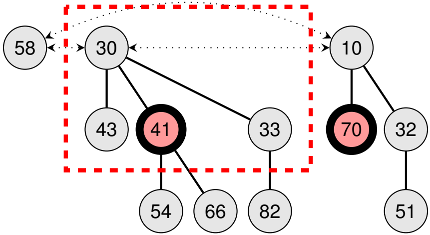
A single Node



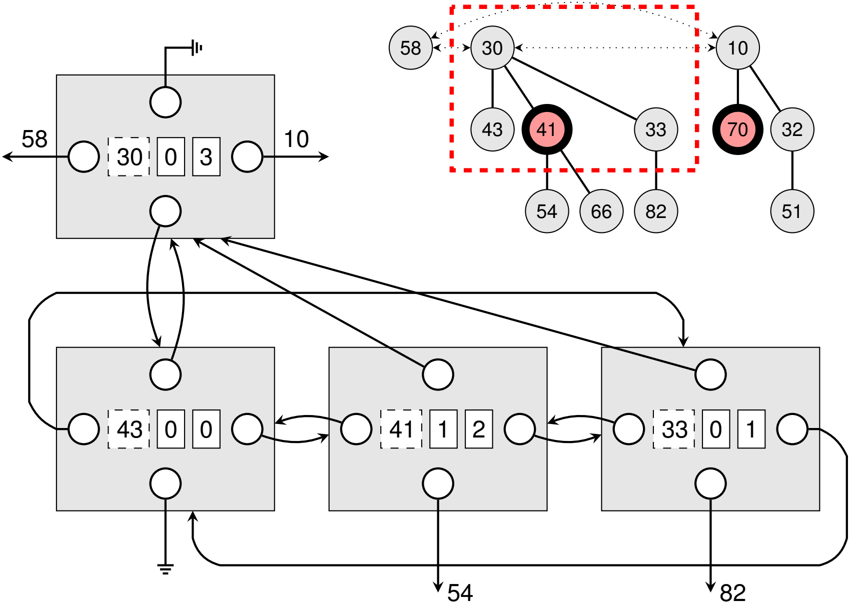
Magnifying a Four-Node Portion



Magnifying a Four-Node Portion



Magnifying a Four-Node Portion



Outline

Structure

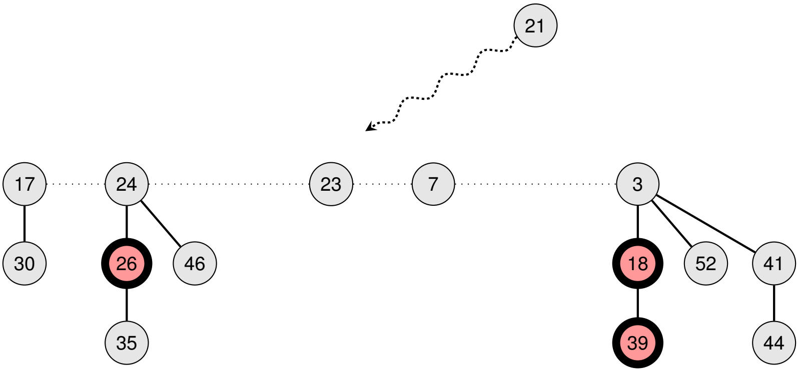
Operations

Glimpse at the Analysis

Amortized Analysis

Fibonacci Heap: INSERT

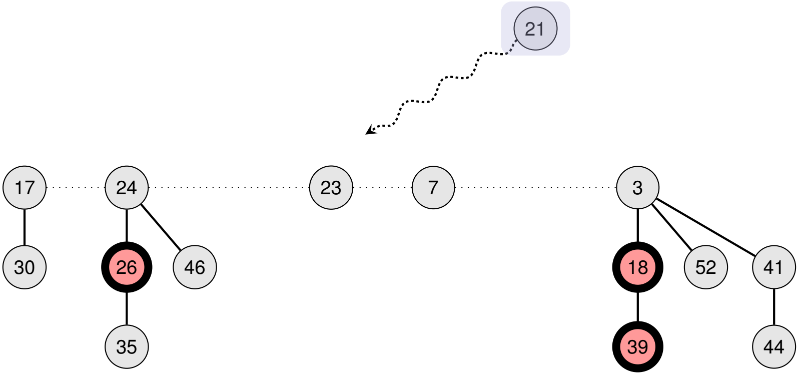
INSERT



Fibonacci Heap: Insert

Insert

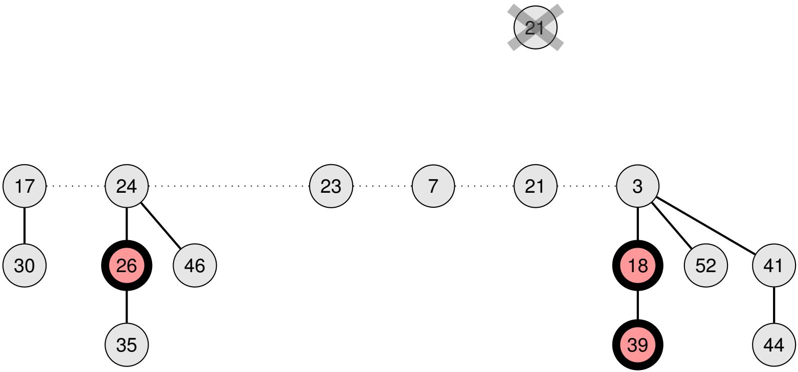
- Create a singleton tree



Fibonacci Heap: INSERT

INSERT

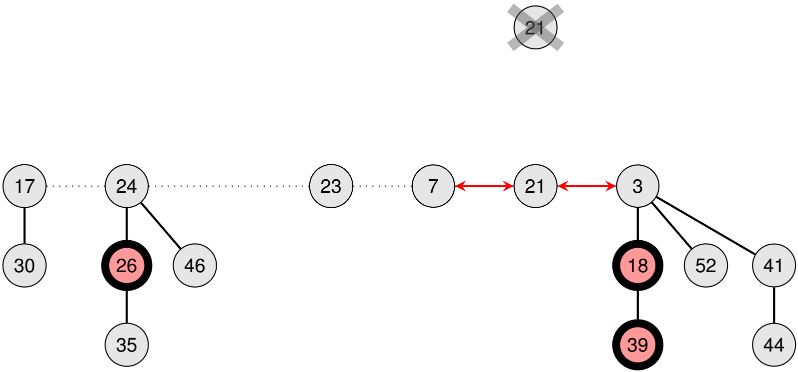
- Create a singleton tree
- Add to root list



Fibonacci Heap: INSERT

INSERT

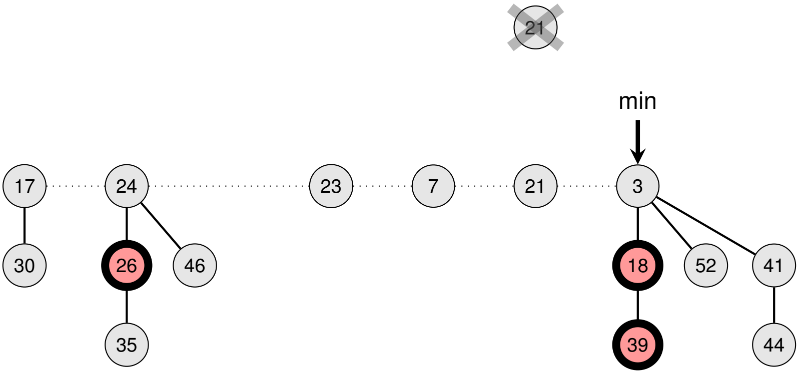
- Create a singleton tree
- Add to root list



Fibonacci Heap: INSERT

INSERT

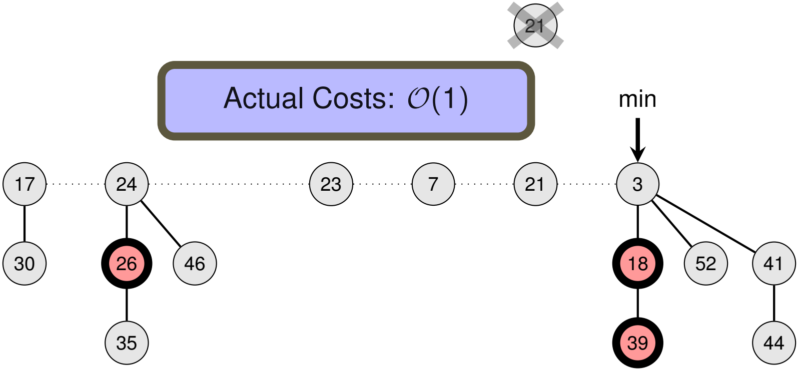
- Create a singleton tree
- Add to root list and update min-pointer (if necessary)



Fibonacci Heap: INSERT

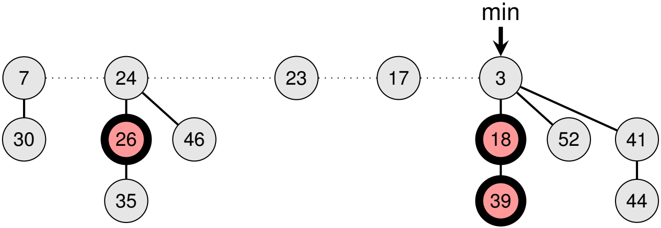
INSERT

- Create a singleton tree
- Add to root list and update min-pointer (if necessary)



Fibonacci Heap: Extract-Min

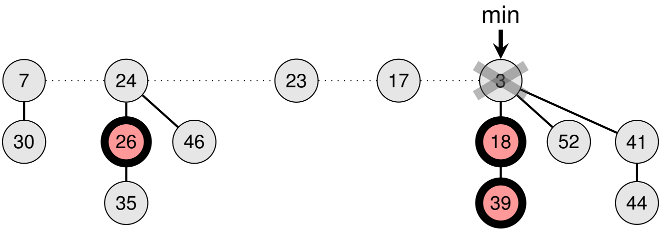
Extract-Min



Fibonacci Heap: Extract-Min

Extract-Min

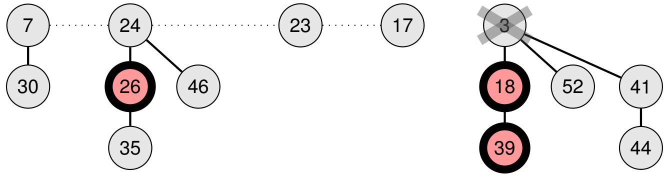
- Delete min



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

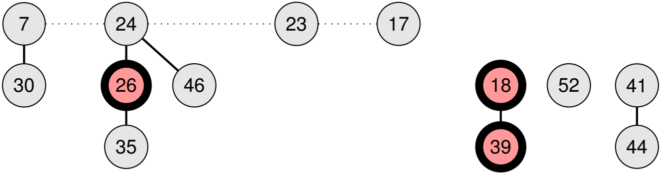
- Delete min ✓



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

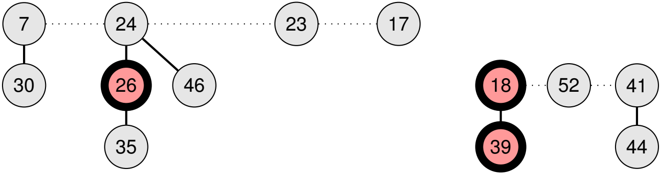
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

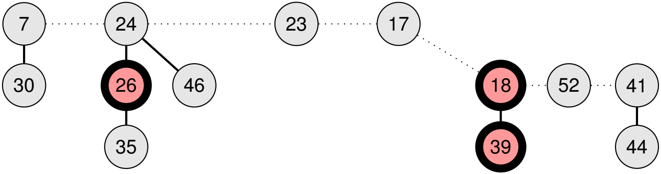
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

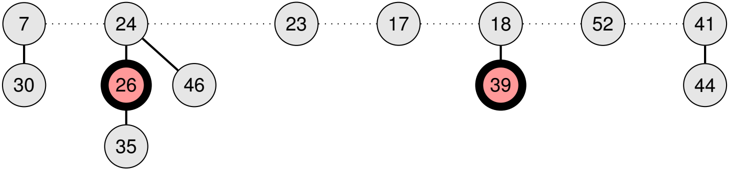
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

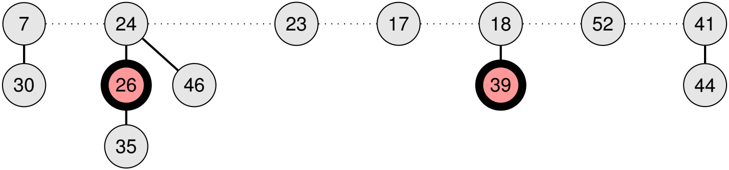
- Delete min ✓
- Meld childen into root list and unmark them ✓



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

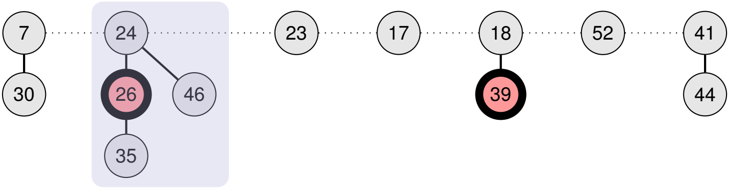
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree



Fibonacci Heap: Extract-Min

Extract-Min

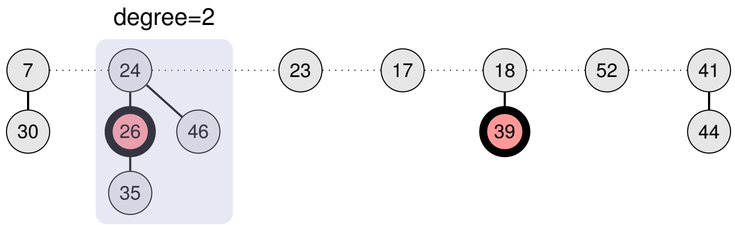
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

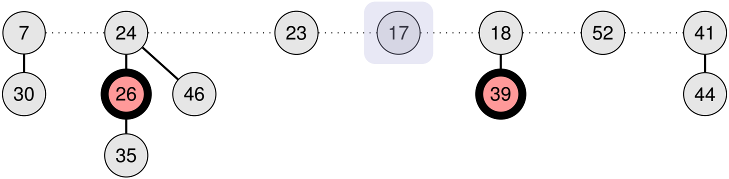
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

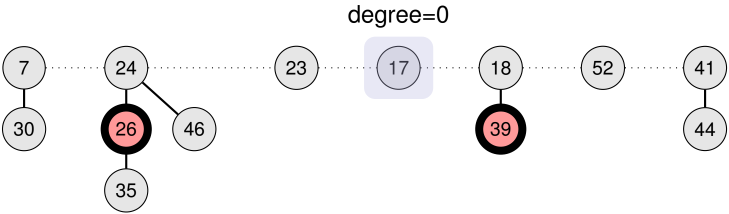
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

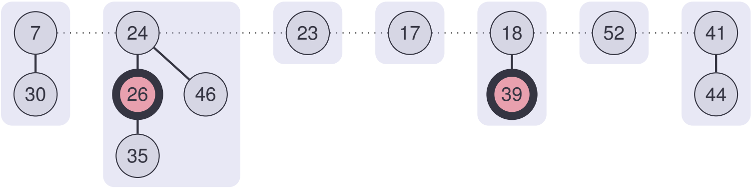
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

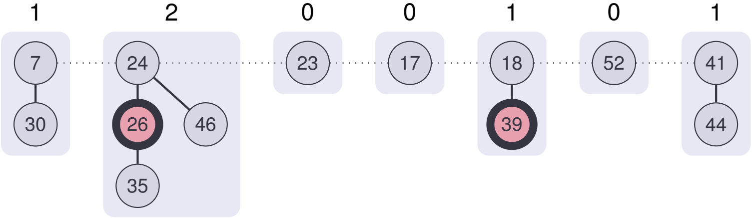
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



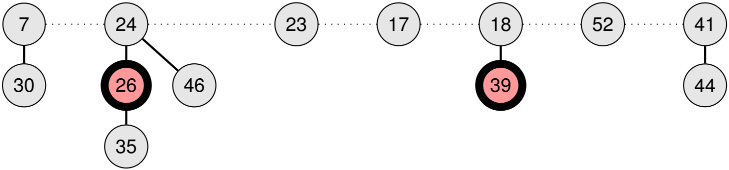
Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

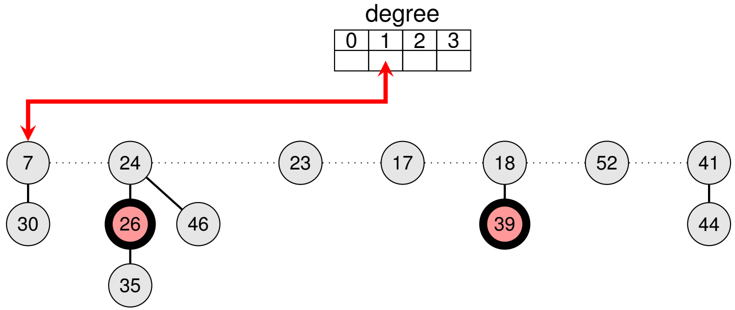
0	1	2	3



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

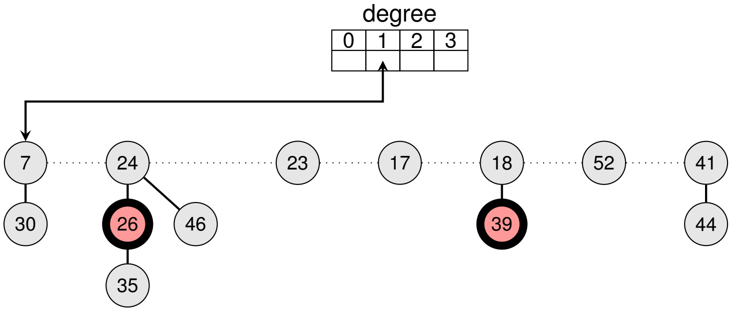
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

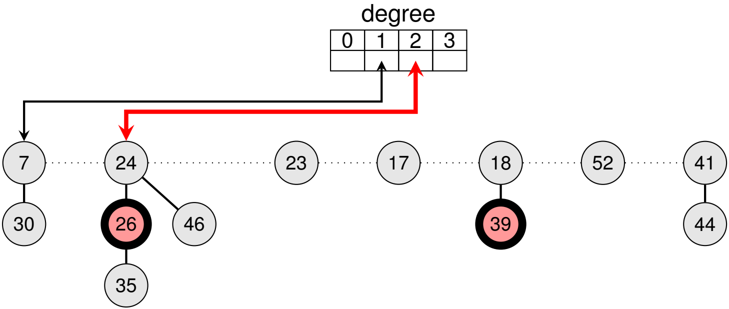
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

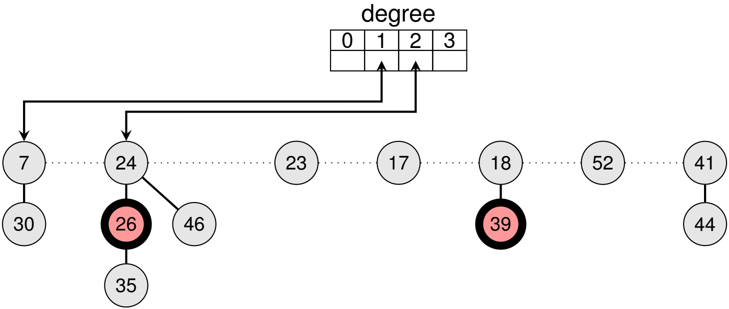
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

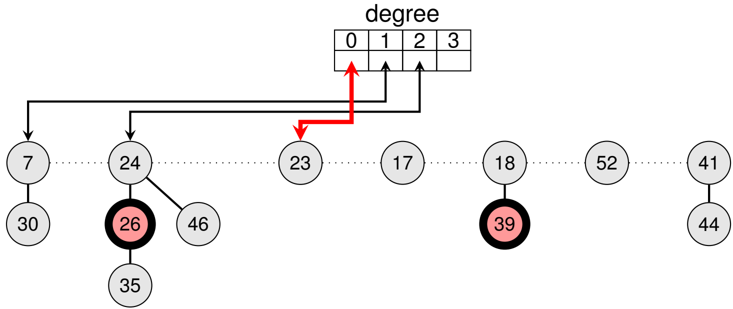
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

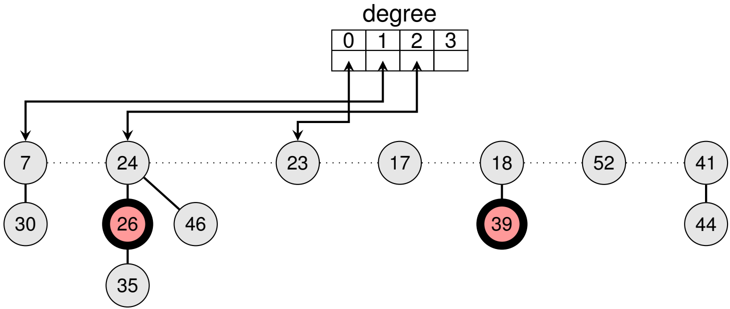
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

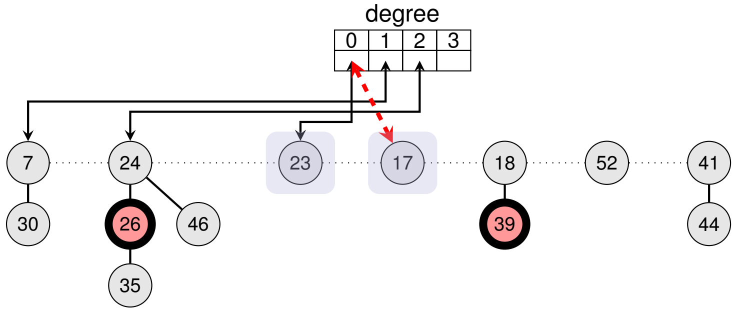
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

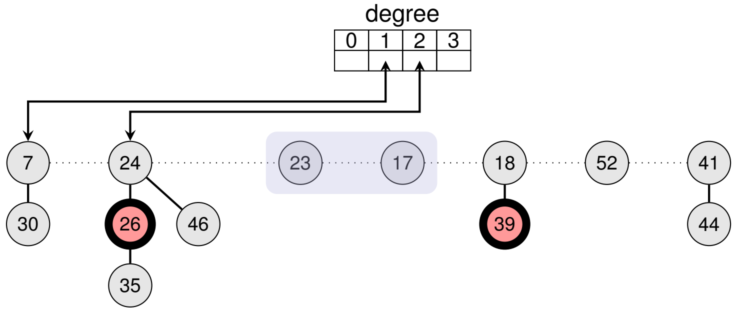
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

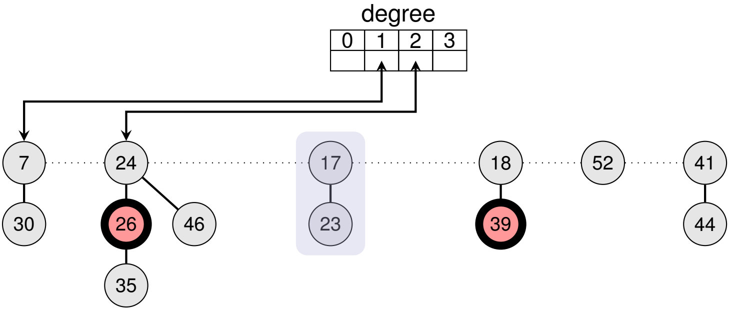
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

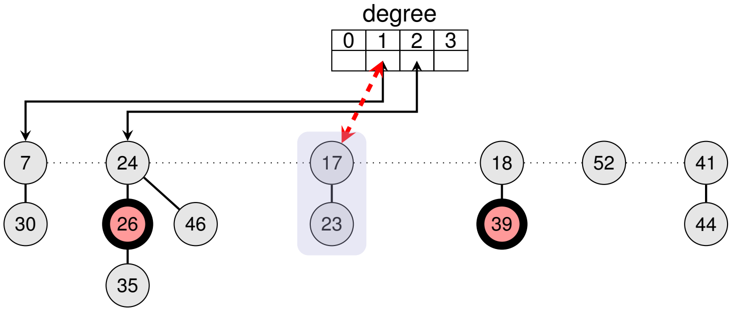
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

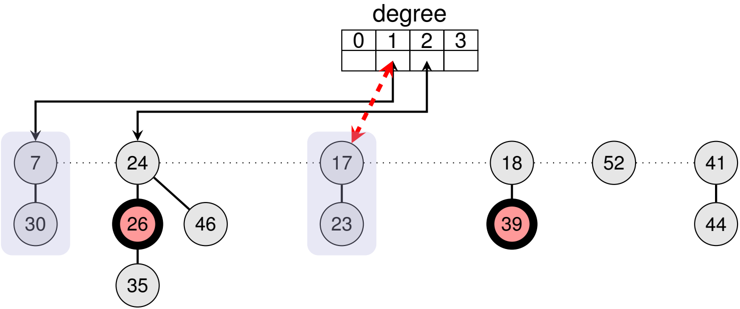
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

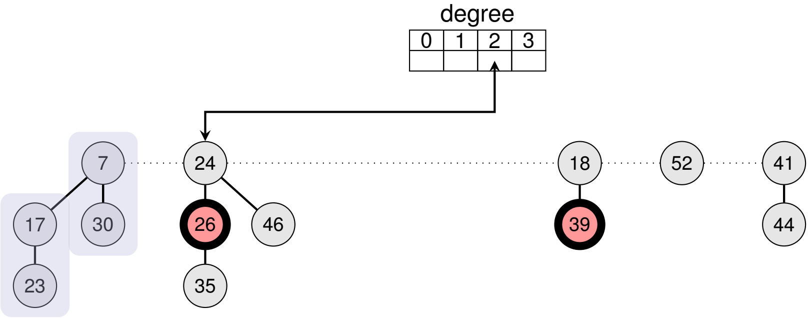
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

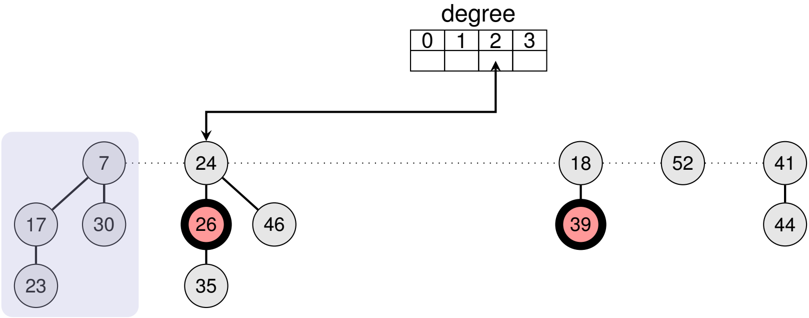
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

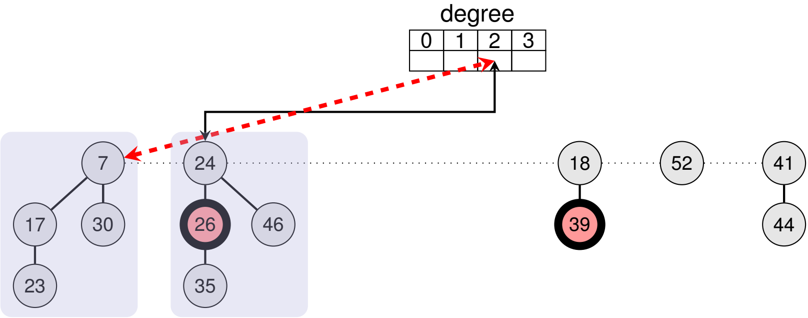
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



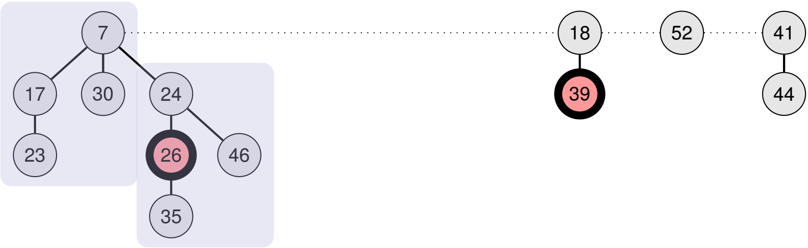
Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

0	1	2	3



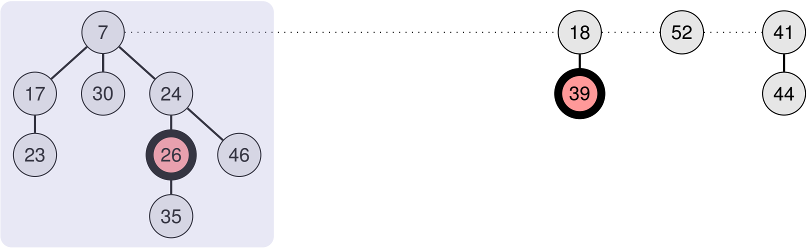
Fibonacci Heap: Extract-Min

Extract-Min

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

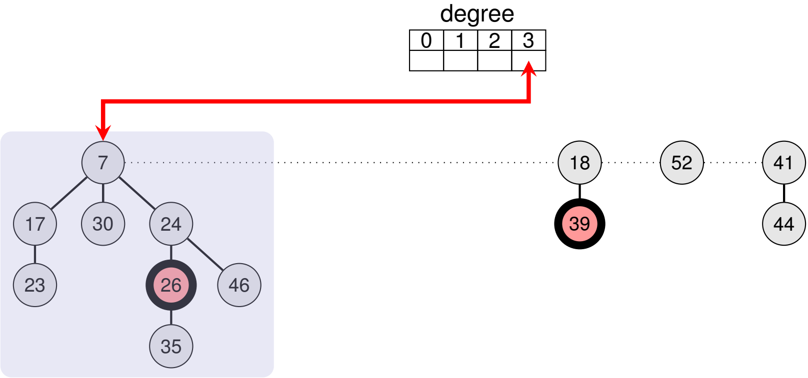
0	1	2	3



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

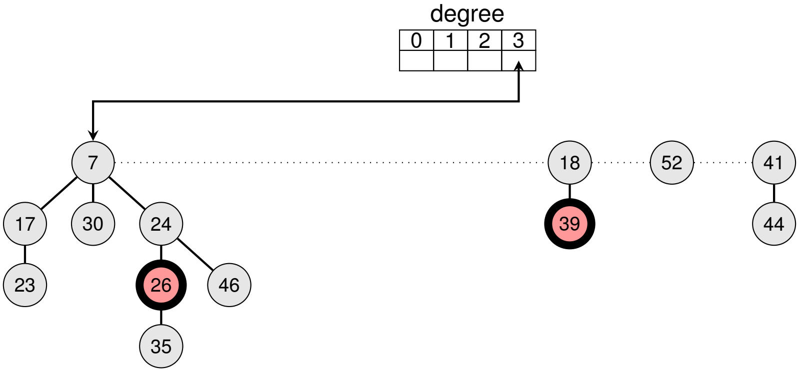
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

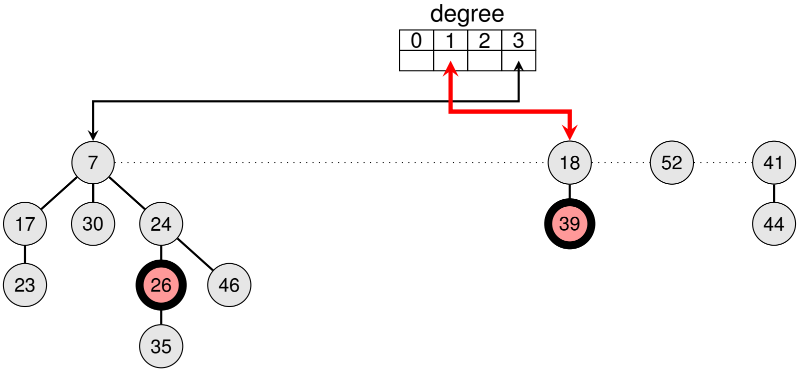
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

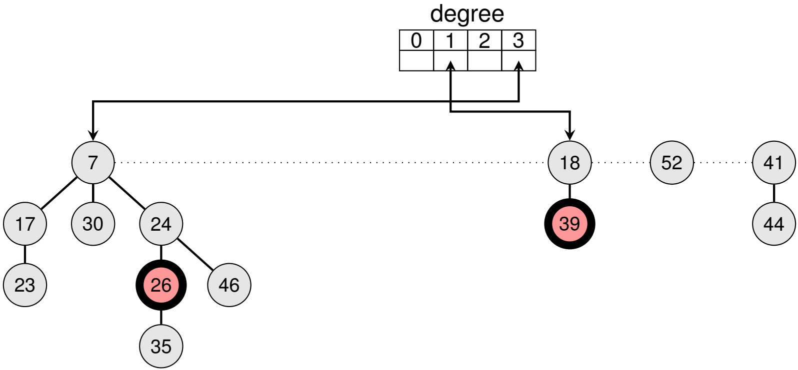
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

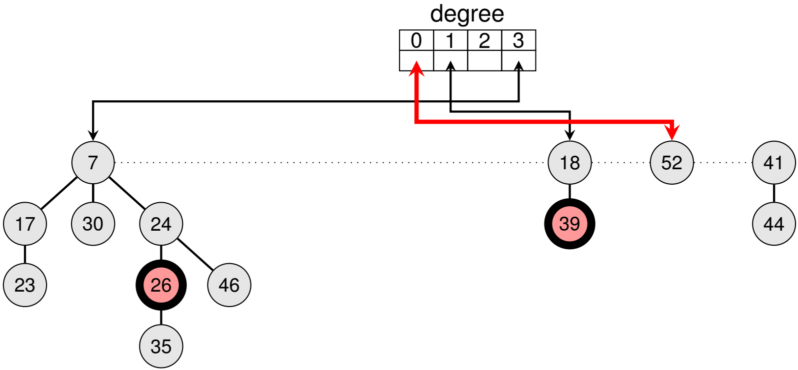
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

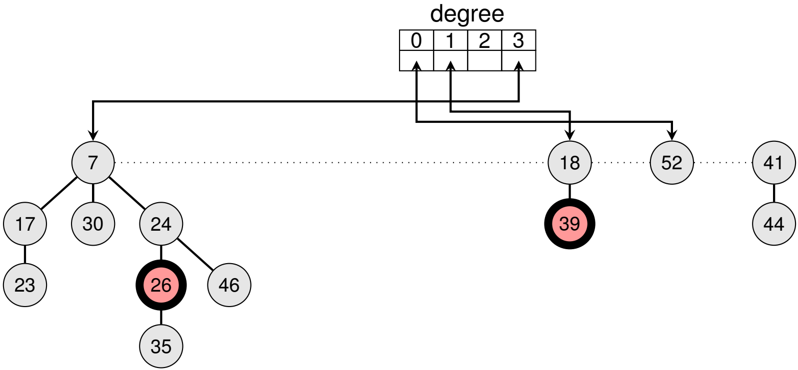
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

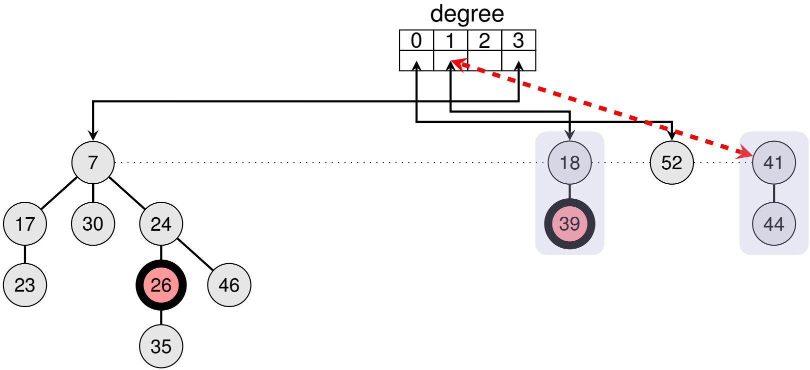
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

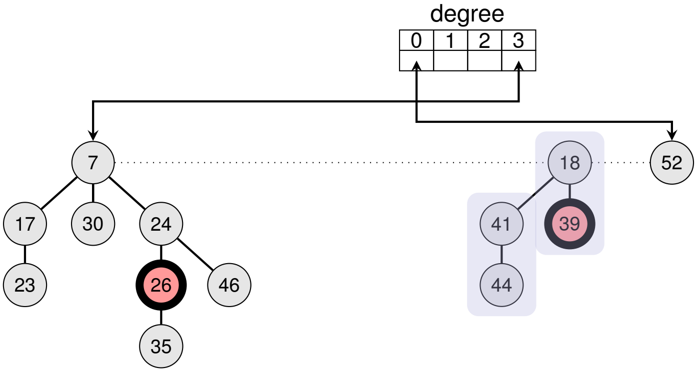
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

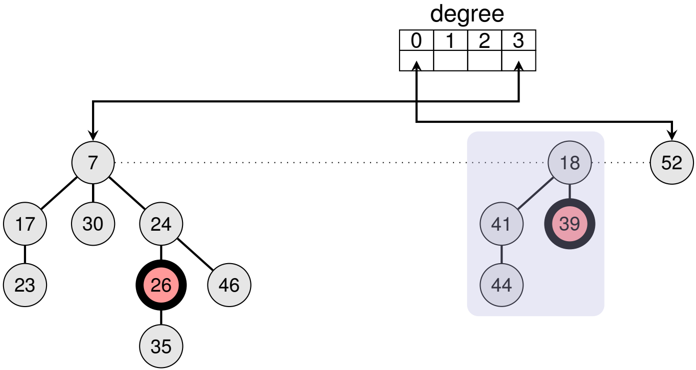
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

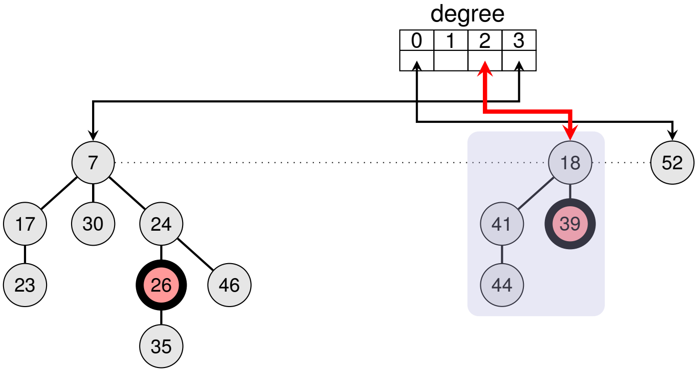
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

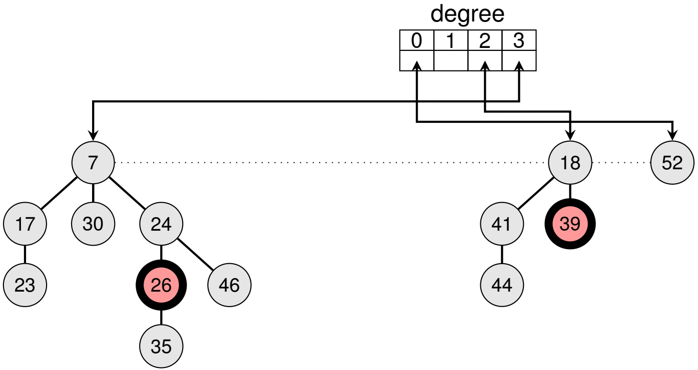
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

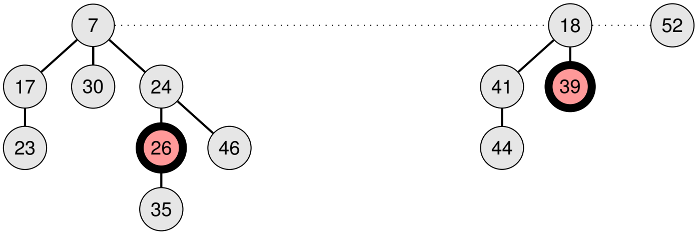
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓



Fibonacci Heap: Extract-Min

Extract-Min

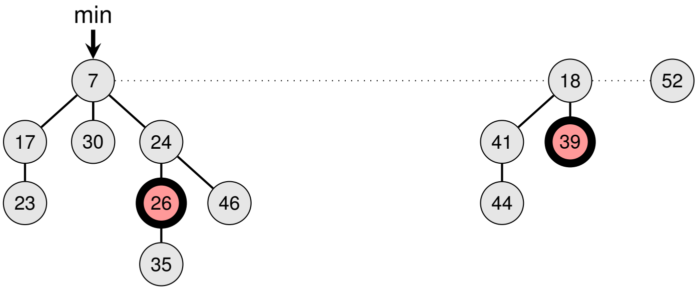
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

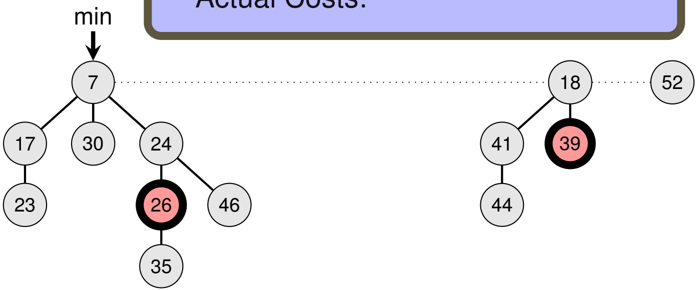


Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Actual Costs:



Fibonacci Heap: EXTRACT-MIN

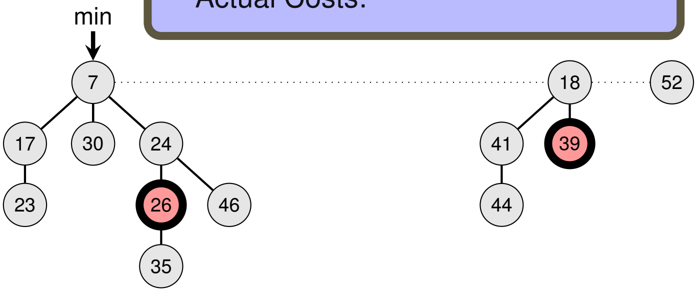
EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size n

Actual Costs:



Fibonacci Heap: EXTRACT-MIN

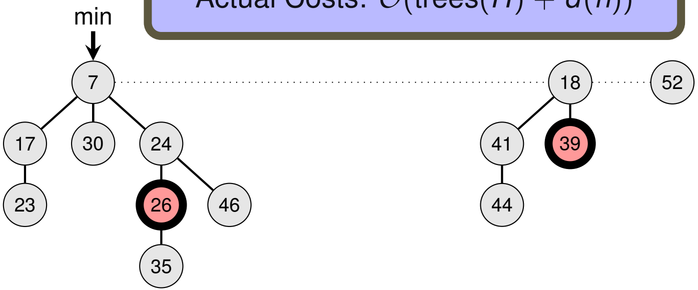
EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size n

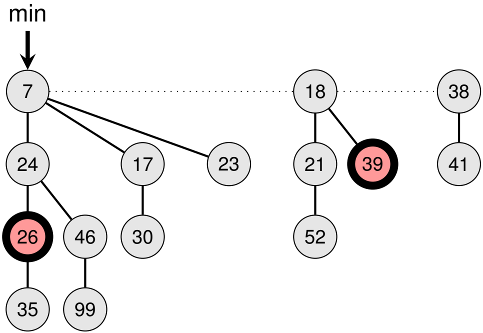
Actual Costs: $\mathcal{O}(\text{trees}(H) + d(n))$



Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

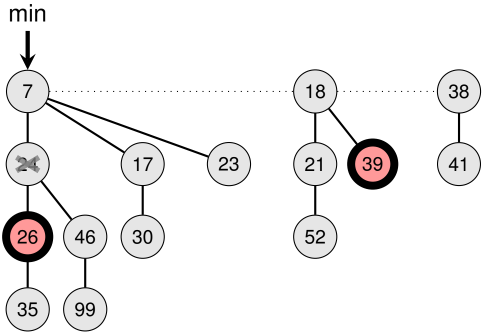
- Decrease the key of x (given by a pointer)



Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)

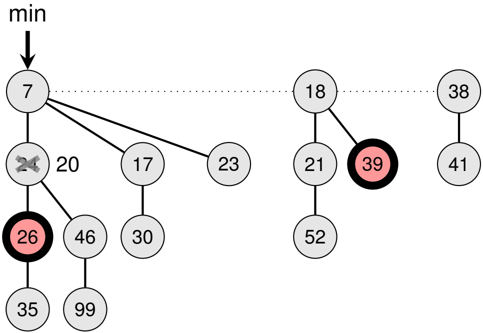


1. DECREASE-KEY 24 $\rightsquigarrow$ 20

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)

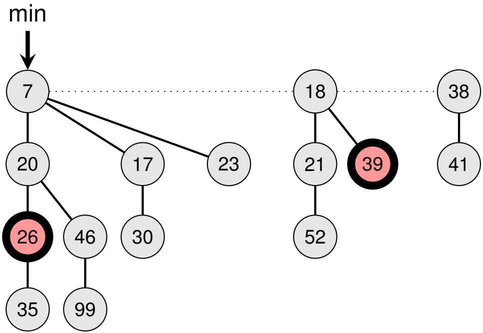


1. DECREASE-KEY 24 $\rightsquigarrow$ 20

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated

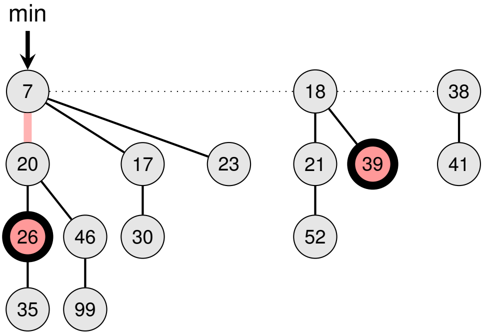


1. DECREASE-KEY 24 $\rightsquigarrow$ 20

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated

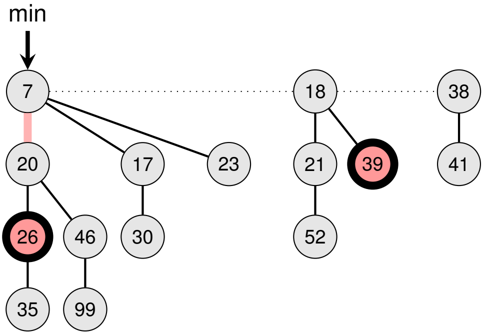


1. DECREASE-KEY 24 $\rightsquigarrow$ 20

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated
  - If not

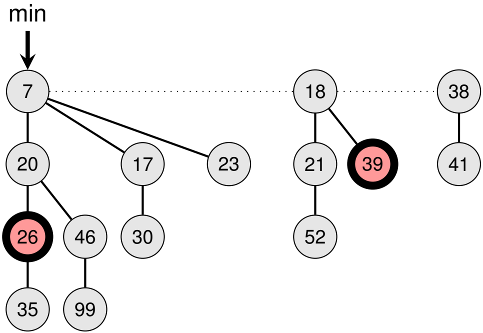


1. DECREASE-KEY 24 $\rightsquigarrow$ 20

Fibonacci Heap: Decrease-Key (First Try)

Decrease-Key of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.

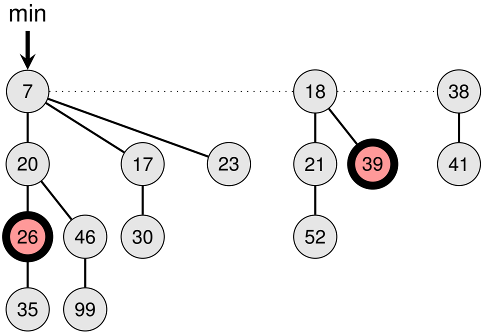


1. Decrease-Key 24 $\rightsquigarrow$ 20

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

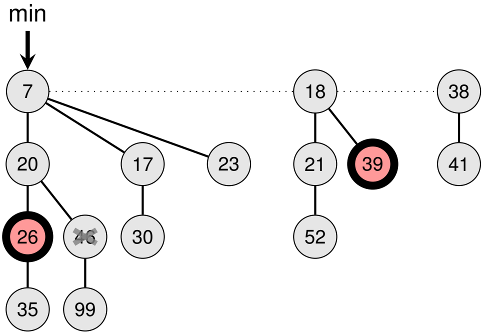


1. DECREASE-KEY 24 $\rightsquigarrow$ 20

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

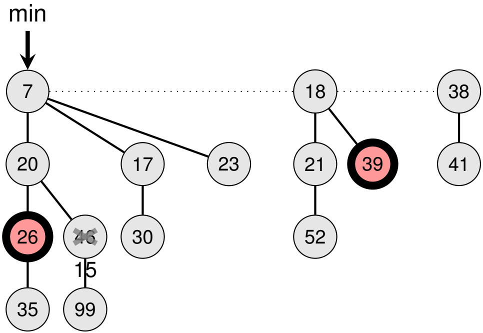


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

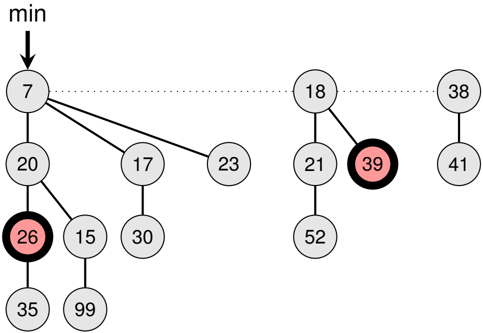


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

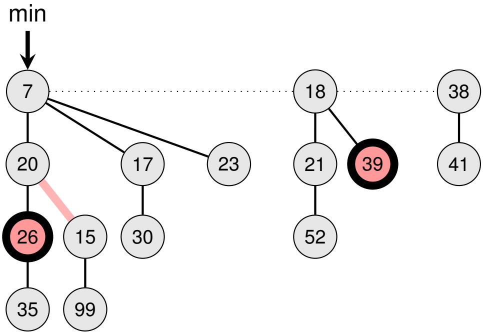


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

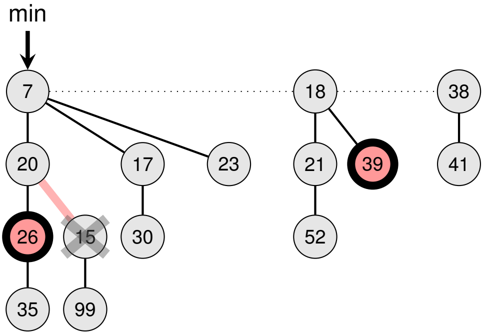


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

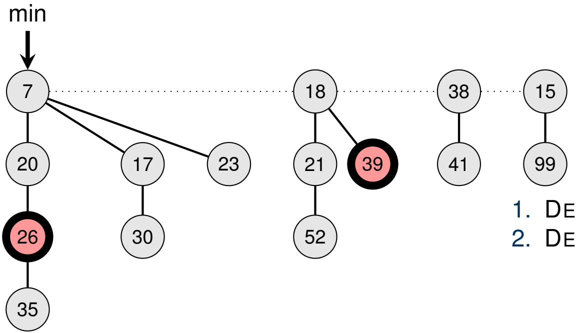


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

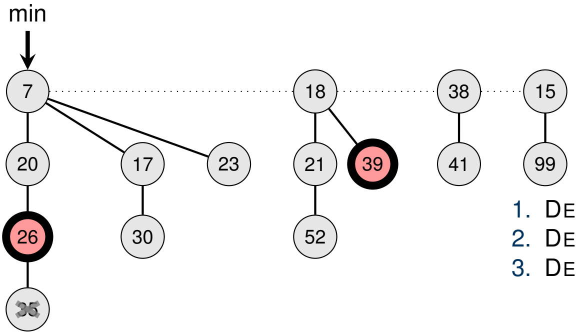


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

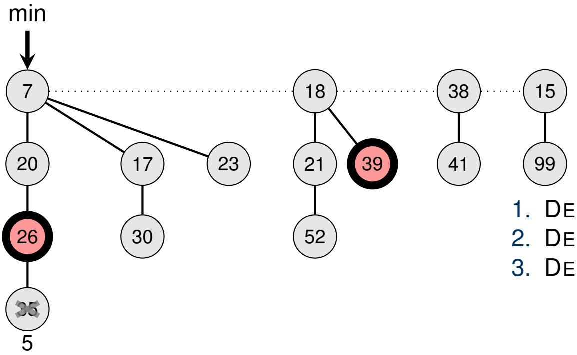


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

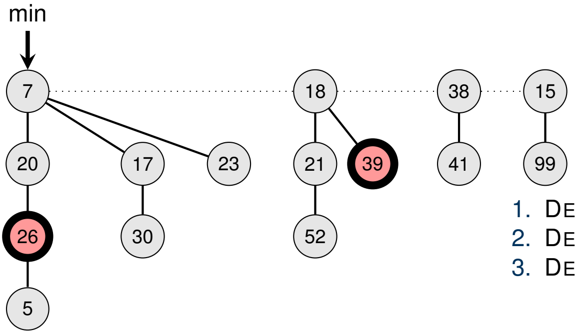


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

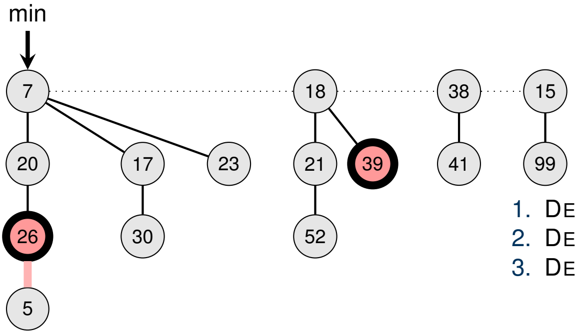


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

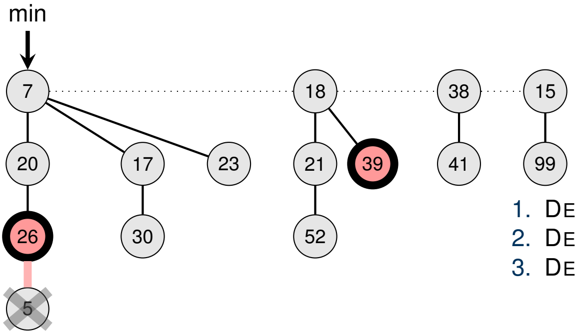


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

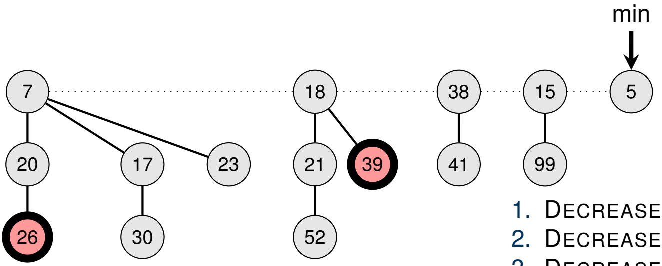


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

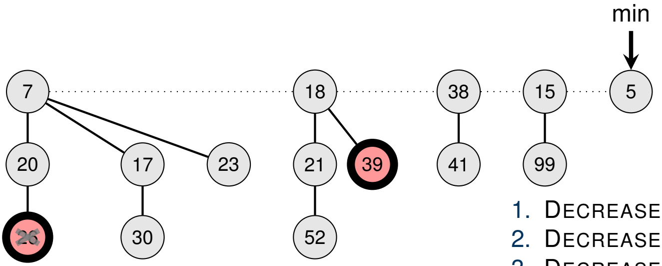


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

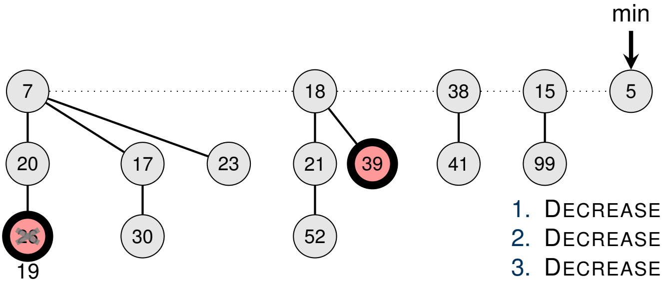


1. DECREASE-KEY 24 ⇝ 20
2. DECREASE-KEY 46 ⇝ 15
3. DECREASE-KEY 35 ⇝ 5
4. DECREASE-KEY 26 ⇝ 19

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

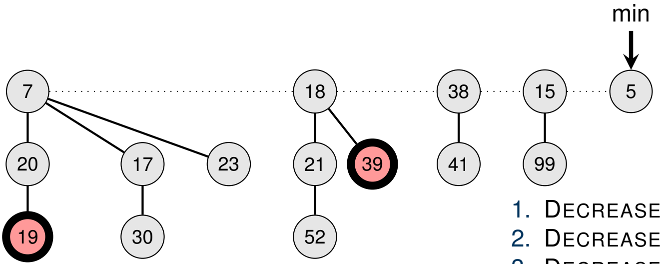


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at *x* and meld into root list (update min).

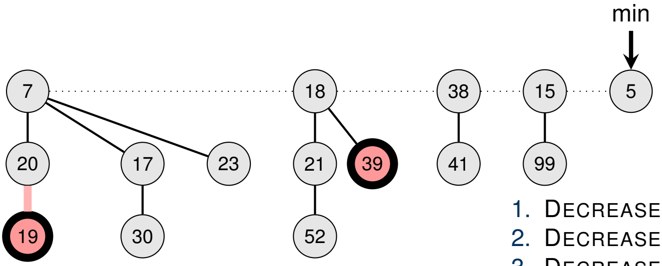


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

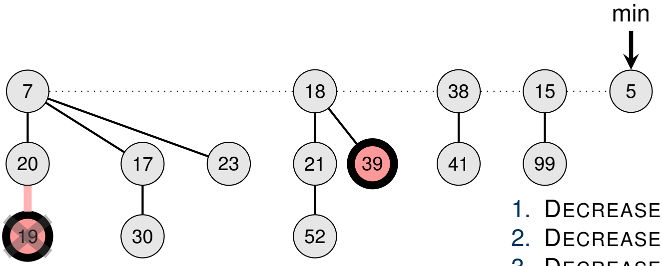


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

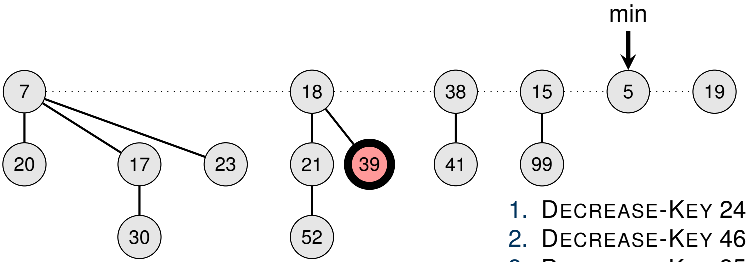


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

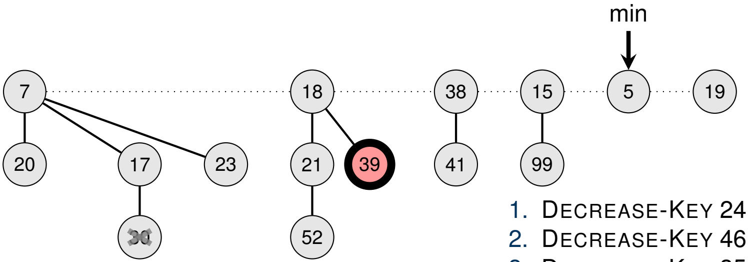


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19

Fibonacci Heap: Decrease-Key (First Try)

Decrease-Key of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

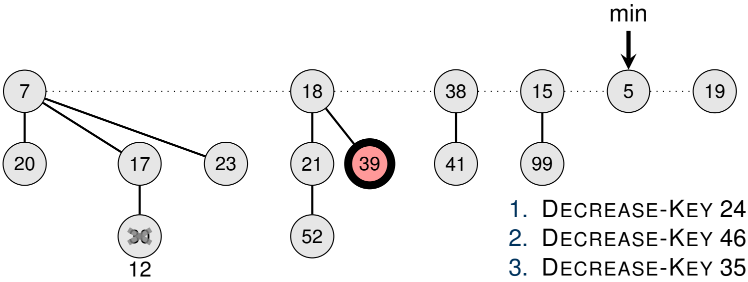


1. Decrease-Key 24 $\rightsquigarrow$ 20
2. Decrease-Key 46 $\rightsquigarrow$ 15
3. Decrease-Key 35 $\rightsquigarrow$ 5
4. Decrease-Key 26 $\rightsquigarrow$ 19
5. Decrease-Key 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

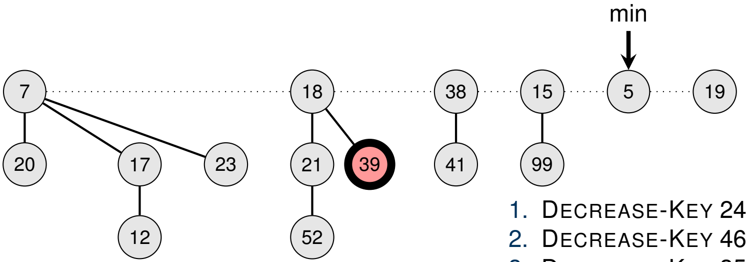


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at *x* and meld into root list (update min).

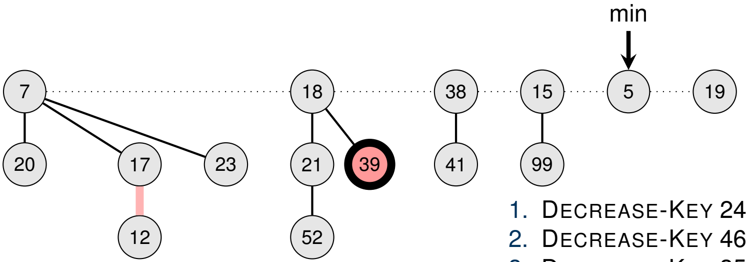


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

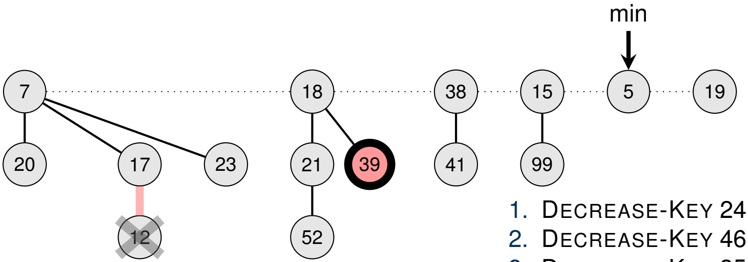


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

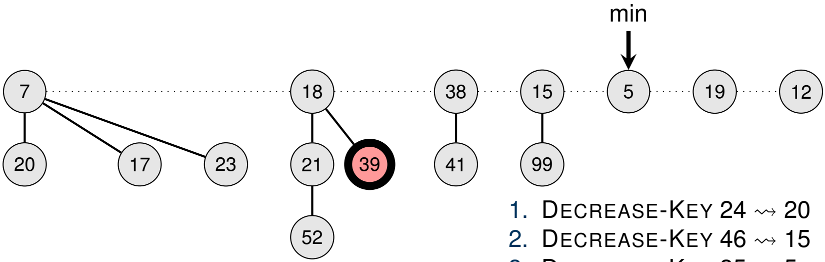


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

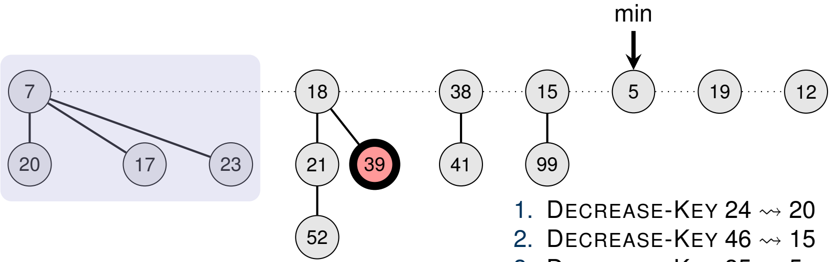


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node *x*

- Decrease the key of *x* (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at *x* and meld into root list (update min).

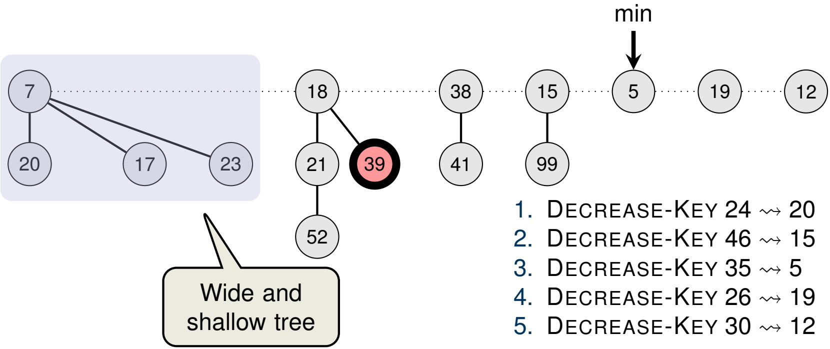


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

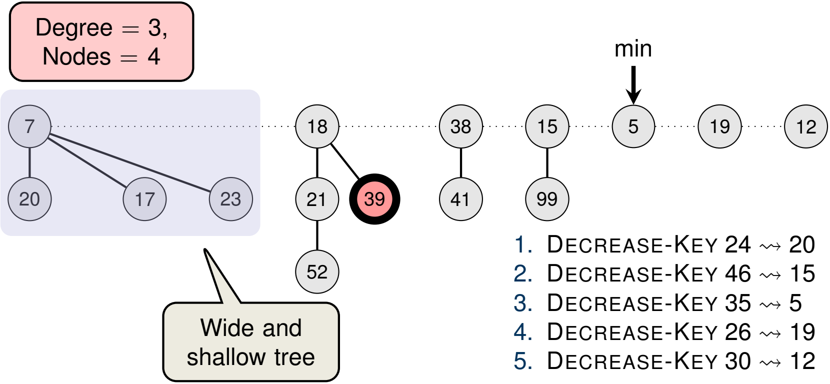


1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).



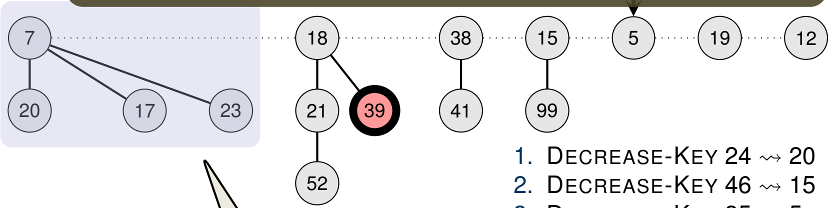
1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY (First Try)

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root



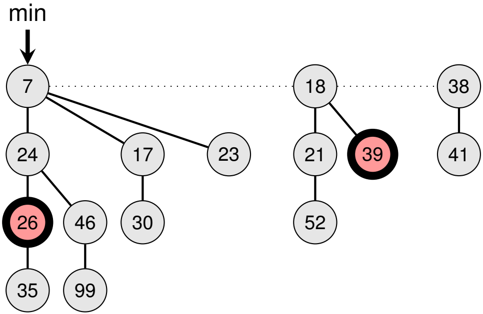
Wide and shallow tree

1. DECREASE-KEY 24 $\rightsquigarrow$ 20
2. DECREASE-KEY 46 $\rightsquigarrow$ 15
3. DECREASE-KEY 35 $\rightsquigarrow$ 5
4. DECREASE-KEY 26 $\rightsquigarrow$ 19
5. DECREASE-KEY 30 $\rightsquigarrow$ 12

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

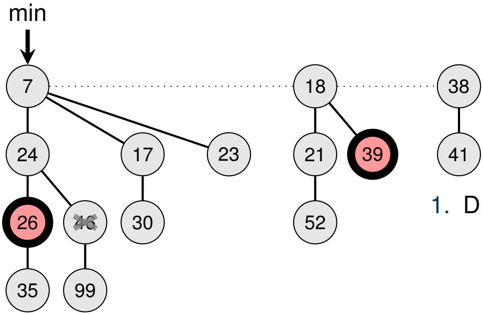
- Decrease the key of x (given by a pointer)



Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)



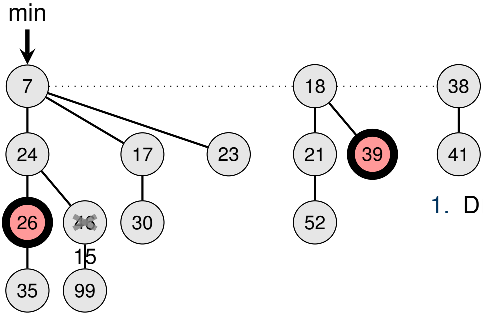
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



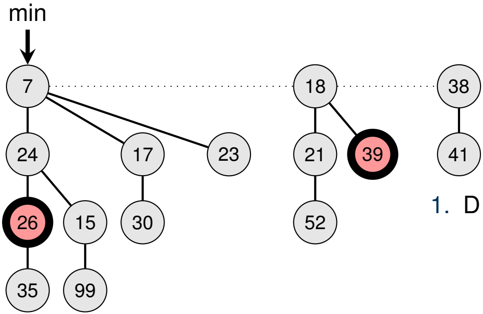
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



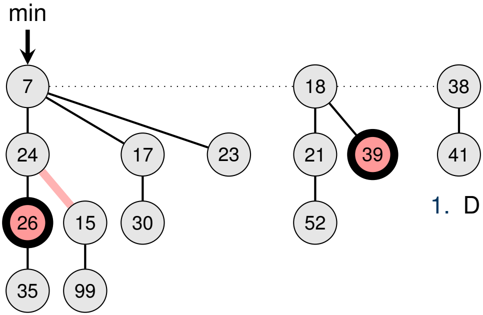
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



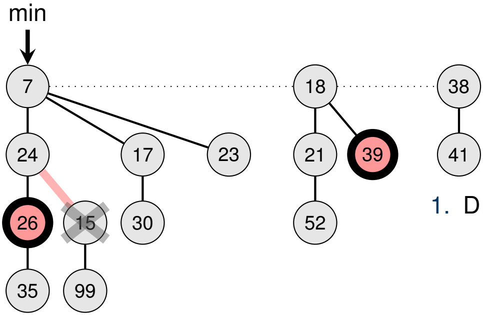
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



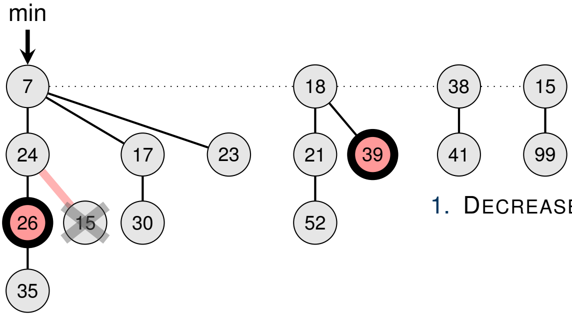
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

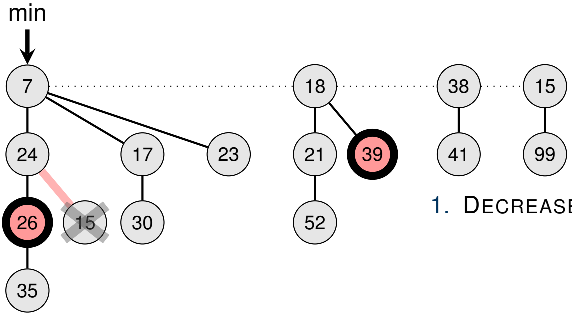
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

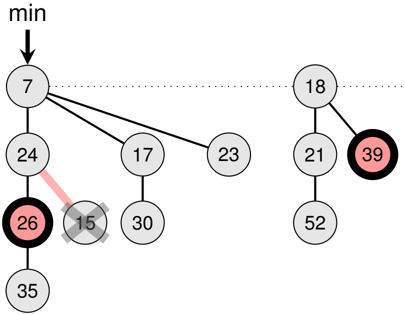
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

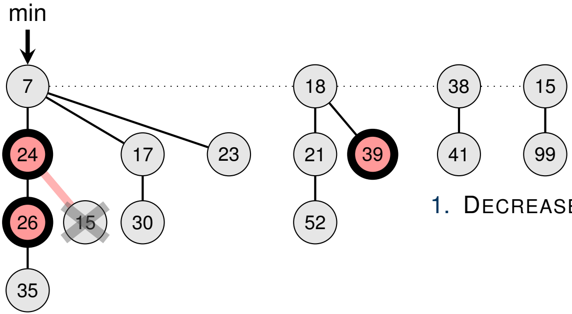
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

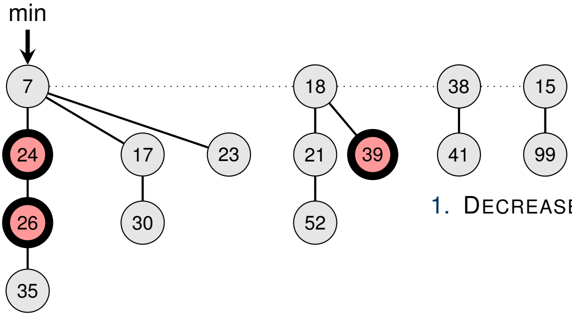
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓

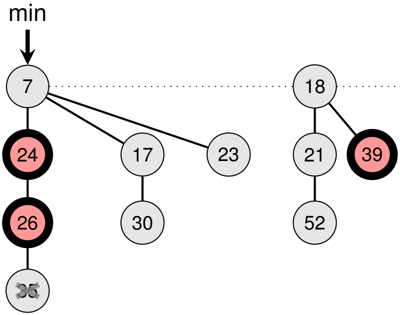
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

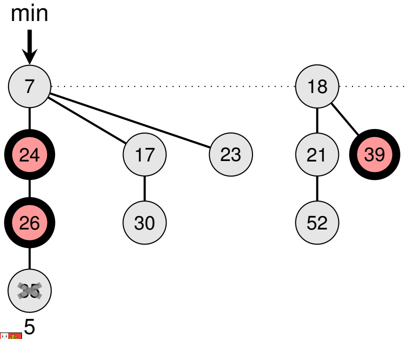
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

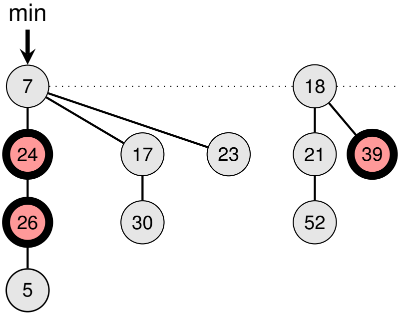
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

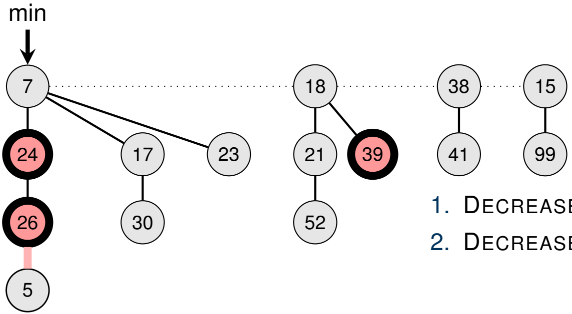
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

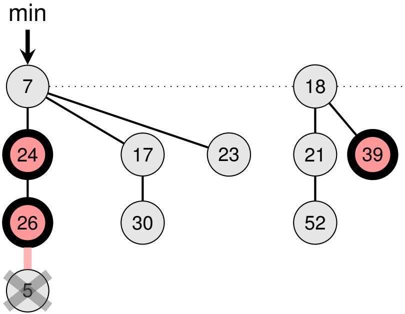
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

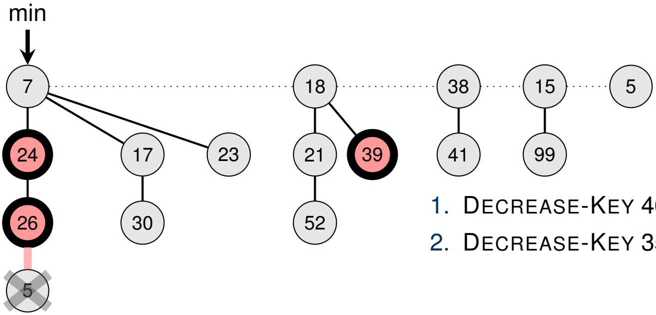
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

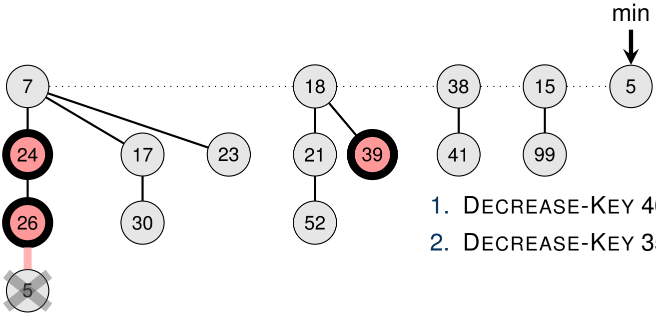
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked,



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

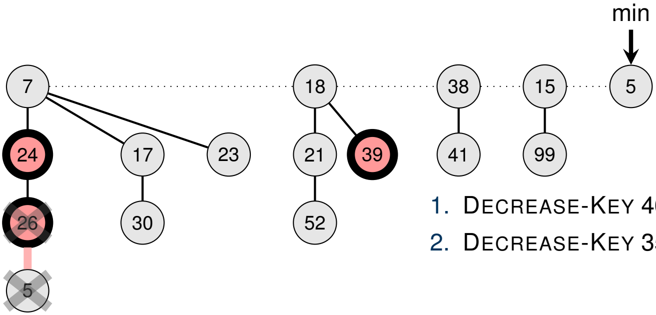
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

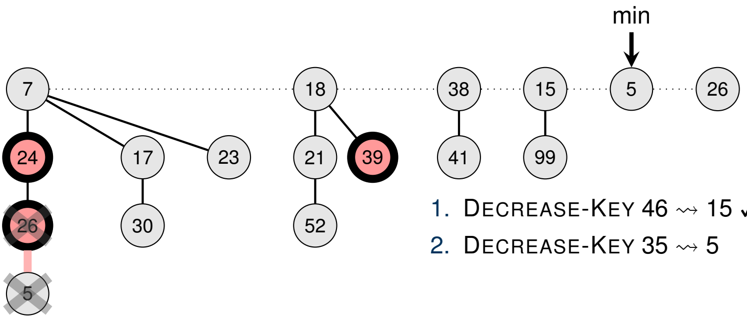
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

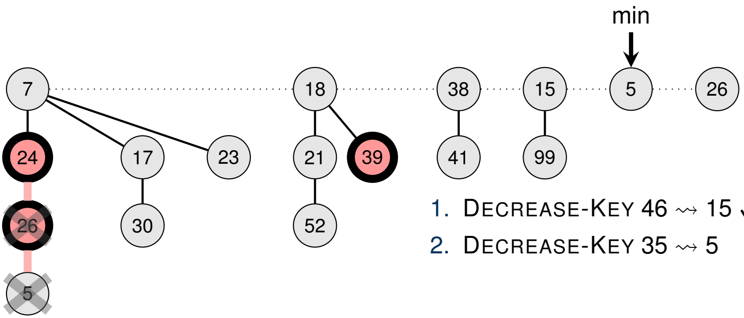
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

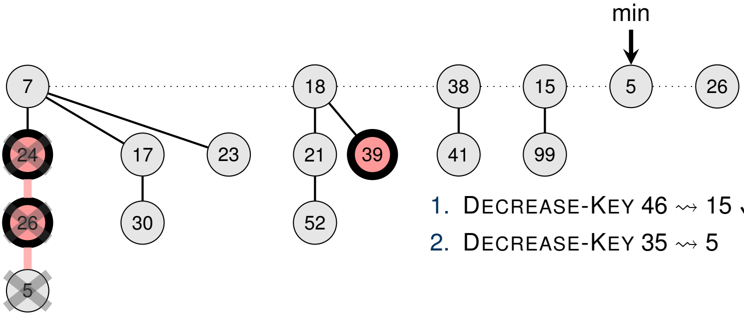
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

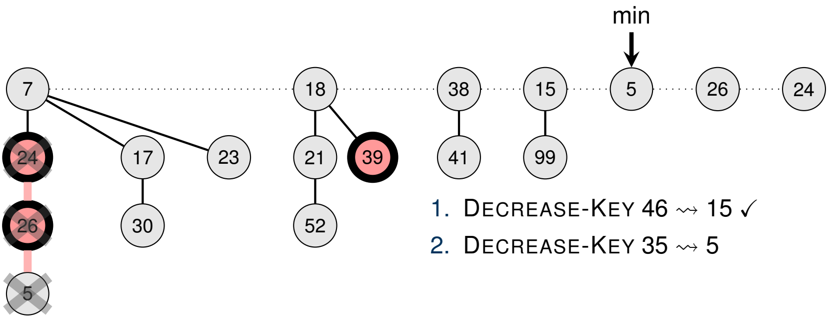
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 ⇝ 15 ✓
2. DECREASE-KEY 35 ⇝ 5

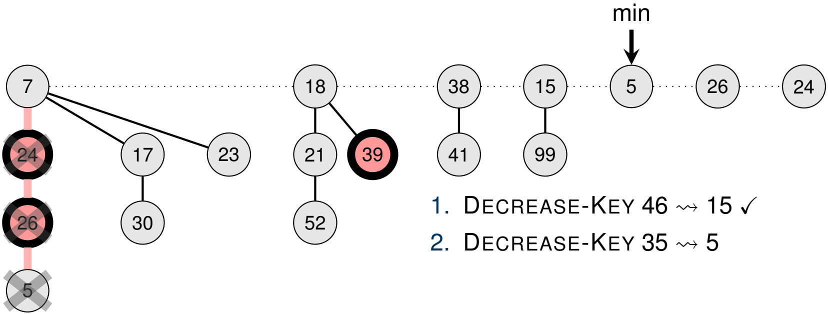
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

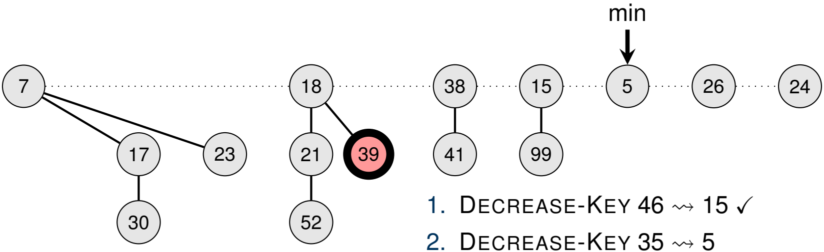
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

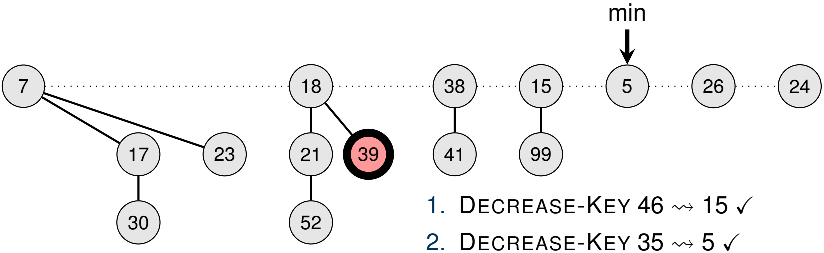
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5 ✓

Fibonacci Heap: DECREASE-KEY

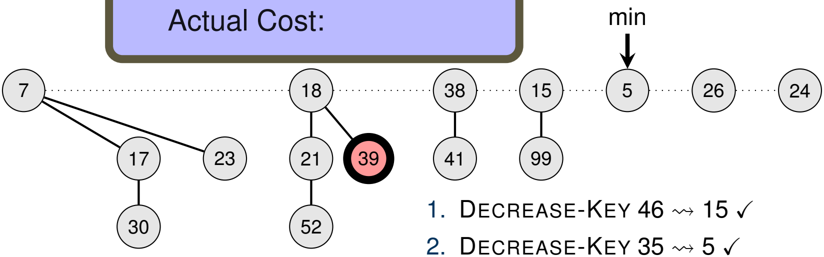
DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

Actual Cost:



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5 ✓

Fibonacci Heap: DECREASE-KEY

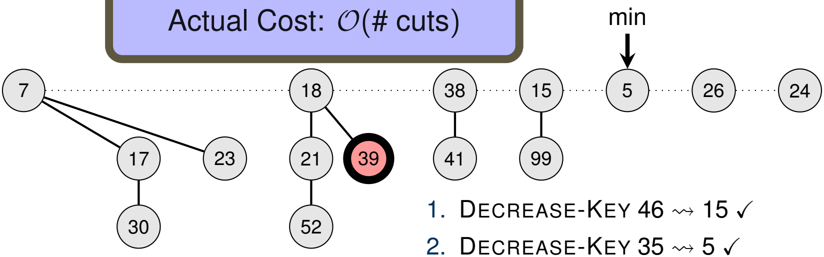
DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

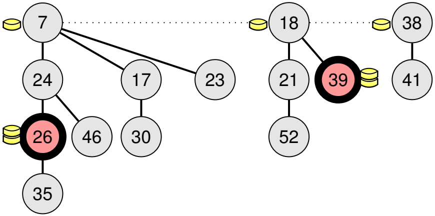
$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

Actual Cost: $\mathcal{O}(\text{\# cuts})$



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5 ✓



5.2 Fibonacci Heaps (Analysis)

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Structure

Operations

Glimpse at the Analysis

Amortized Analysis

Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\text{\# cuts}) \leq \mathcal{O}(\text{marks}(H))$

Amortized Analysis via Potential Method

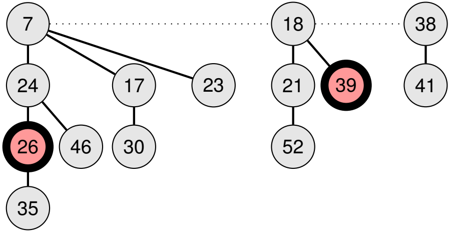
- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

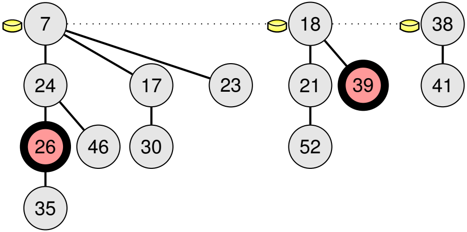
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

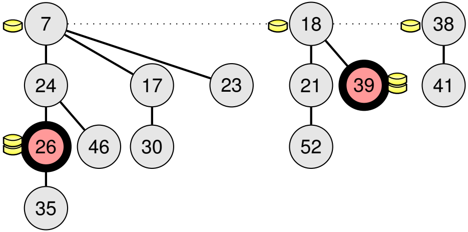
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

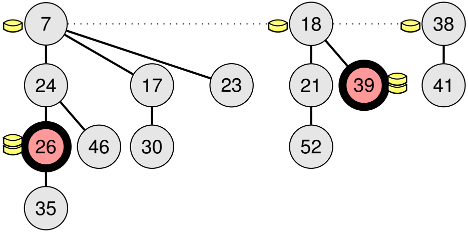
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



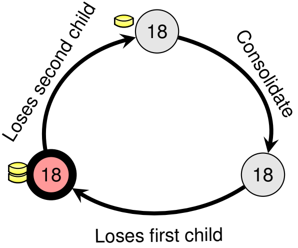
Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$ amortized $\mathcal{O}(d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$ amortized $\mathcal{O}(1)$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



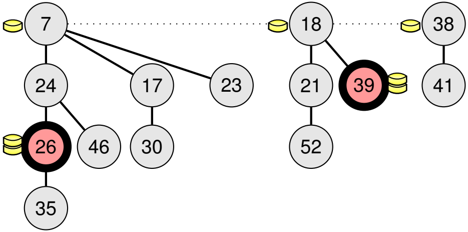
Lifecycle of a node



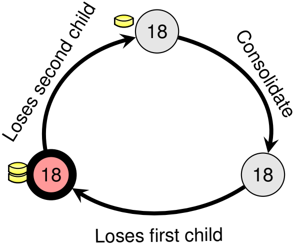
Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$ ✓
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$ amortized $\mathcal{O}(d(n))$?
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$ amortized $\mathcal{O}(1)$?

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Lifecycle of a node



Outline

Structure

Operations

Glimpse at the Analysis

Amortized Analysis

Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x + 1)$, where x is the number of cuts.

Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

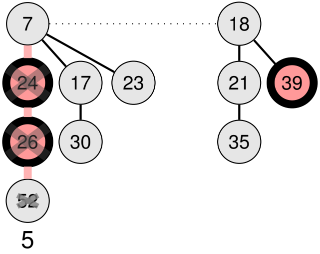
Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential



Amortized Analysis of DECREASE-KEY

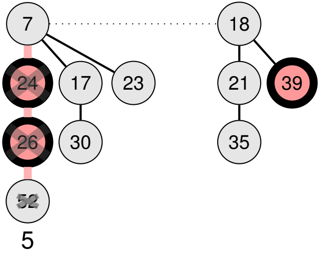
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') =$



Amortized Analysis of DECREASE-KEY

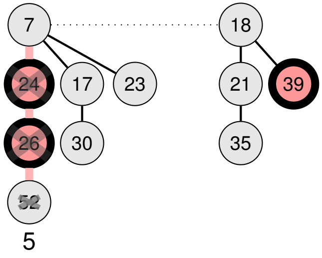
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$



Amortized Analysis of DECREASE-KEY

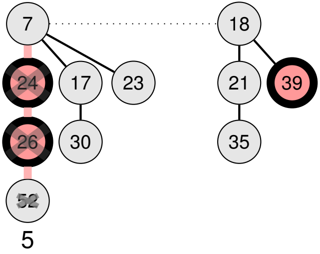
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq$



Amortized Analysis of DECREASE-KEY

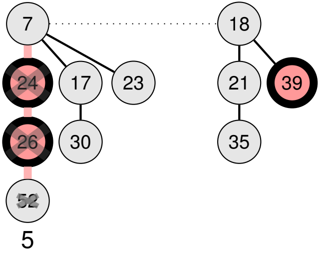
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x + 1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$



Amortized Analysis of DECREASE-KEY

Actual Cost

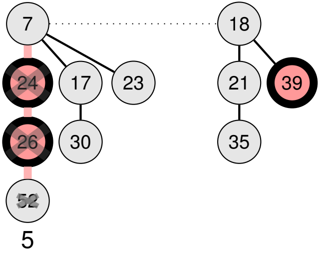
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$



Amortized Analysis of DECREASE-KEY

Actual Cost

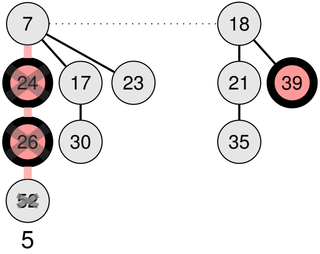
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$



Amortized Cost

$$\widehat{c_i} = c_i + \Delta\Phi$$

Amortized Analysis of DECREASE-KEY

Actual Cost

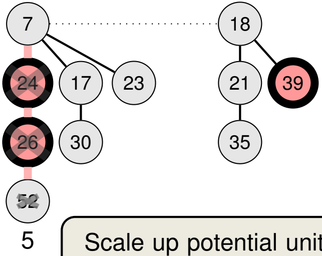
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$



Scale up potential units

Amortized Cost

$$\widehat{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(x+1) + 4 - x$$

Amortized Analysis of DECREASE-KEY

Actual Cost

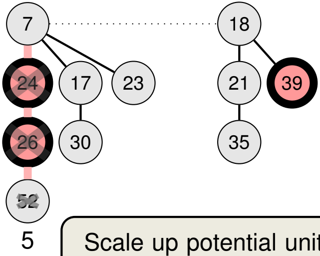
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x+2) = 4 - x.$



Scale up potential units

Amortized Cost

$$\widehat{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(x+1) + 4 - x = \mathcal{O}(1)$$

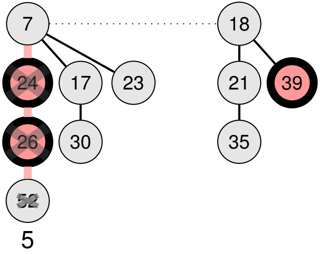
Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

First Coin $\rightsquigarrow$ pays cut
Second Coin $\rightsquigarrow$ increase of trees(H)



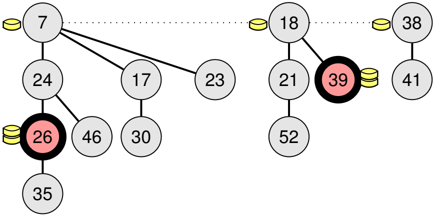
Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$

Amortized Cost

$$\widehat{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(x+1) + 4 - x = \mathcal{O}(1)$$



5.2 Fibonacci Heaps (Analysis)

Frank Stajano

Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Recap of INSERT, EXTRACT-MIN and DECREASE-KEY

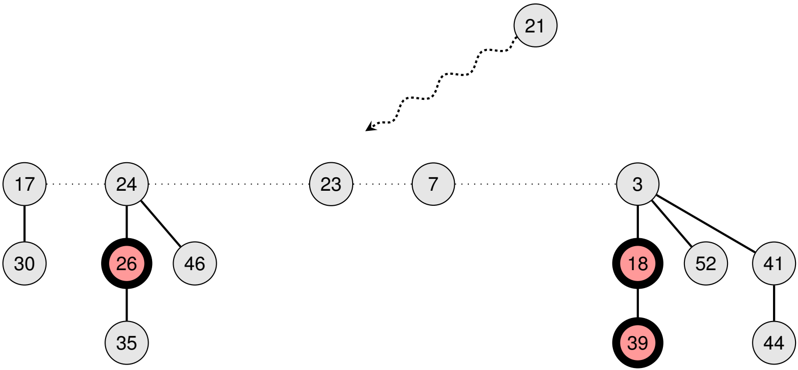
Glimpse at the Analysis

Amortized Analysis

Bounding the Maximum Degree

Fibonacci Heap: Insert

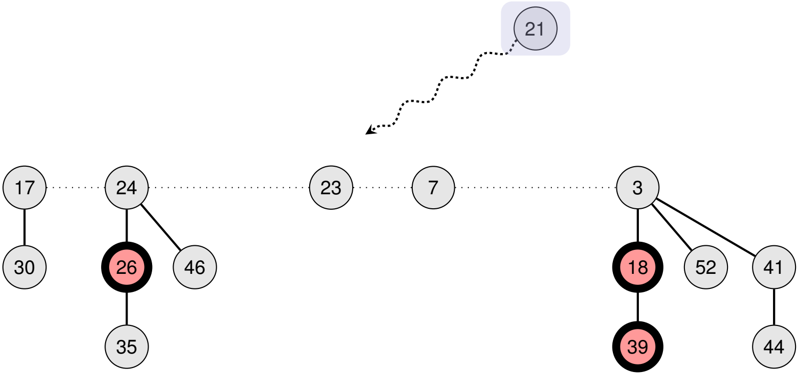
Insert



Fibonacci Heap: Insert

Insert

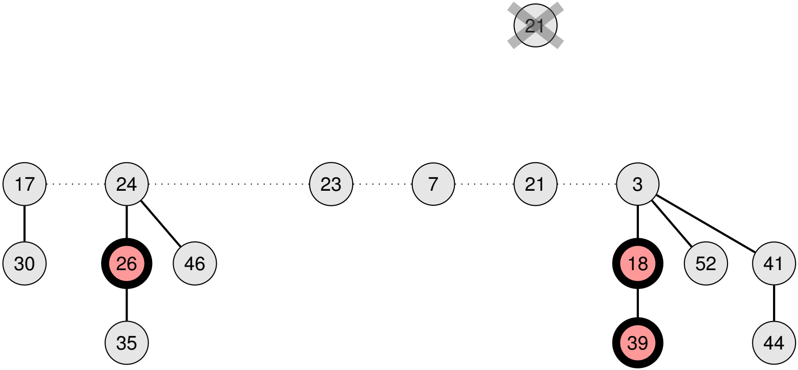
- Create a singleton tree



Fibonacci Heap: INSERT

INSERT

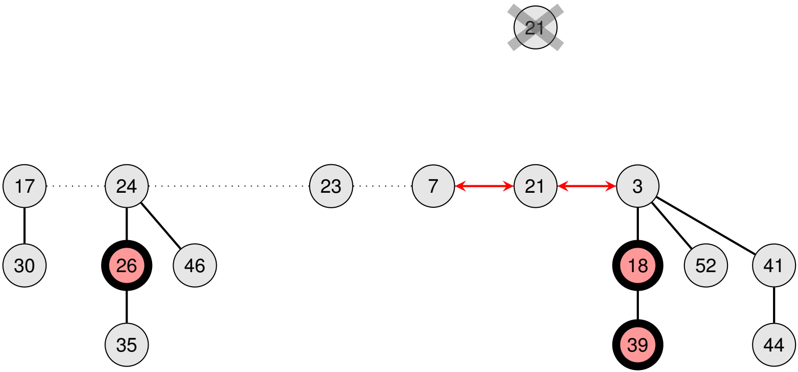
- Create a singleton tree
- Add to root list



Fibonacci Heap: INSERT

INSERT

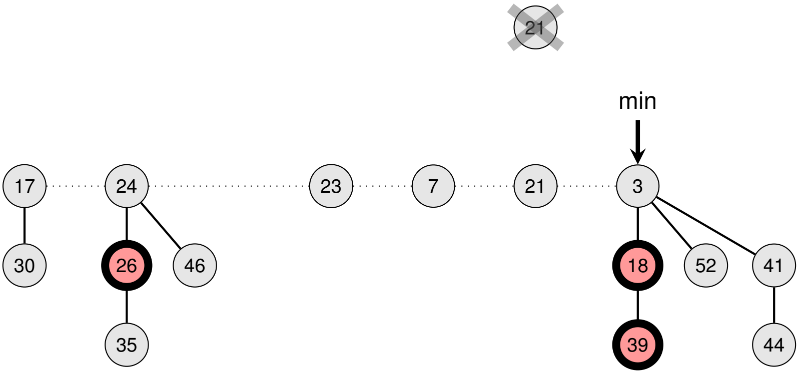
- Create a singleton tree
- Add to root list



Fibonacci Heap: INSERT

INSERT

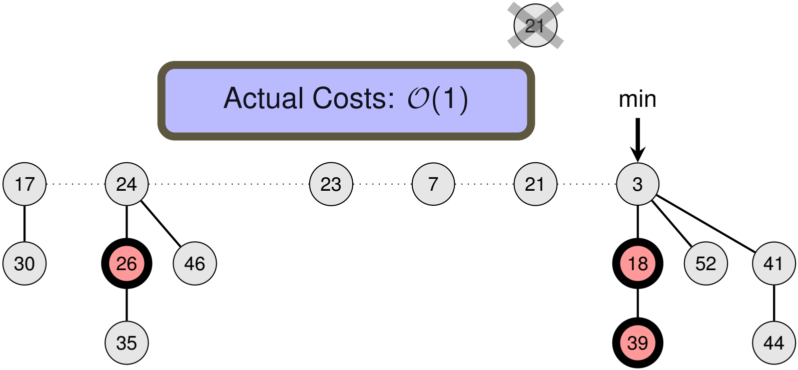
- Create a singleton tree
- Add to root list and update min-pointer (if necessary)



Fibonacci Heap: INSERT

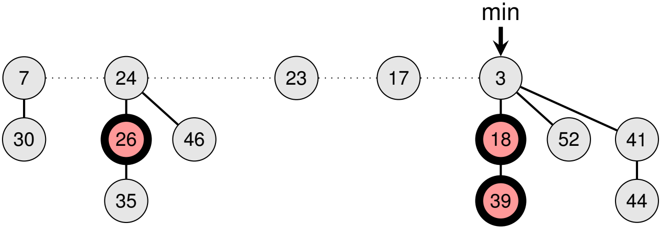
INSERT

- Create a singleton tree
- Add to root list and update min-pointer (if necessary)



Fibonacci Heap: EXTRACT-MIN

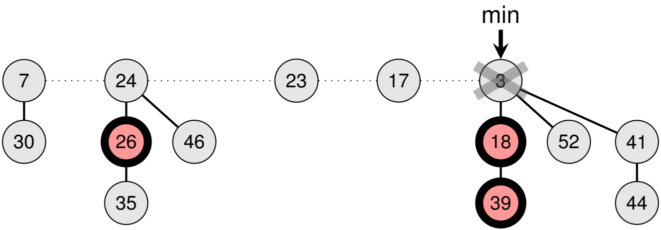
EXTRACT-MIN



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

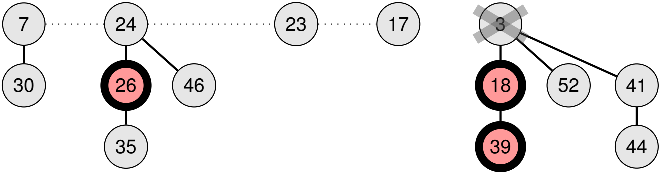
- Delete min



Fibonacci Heap: Extract-Min

Extract-Min

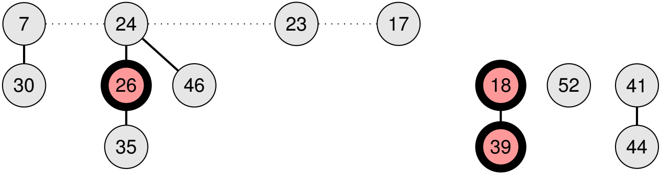
- Delete min ✓



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

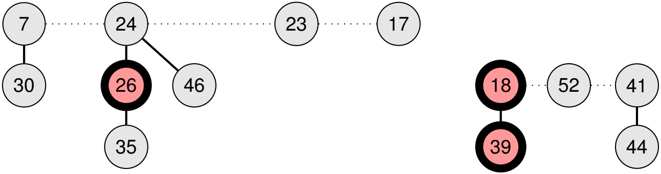
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

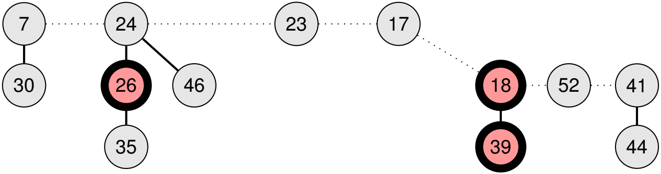
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

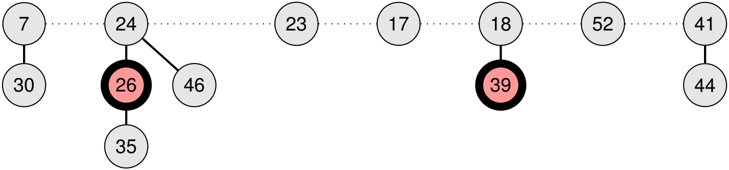
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

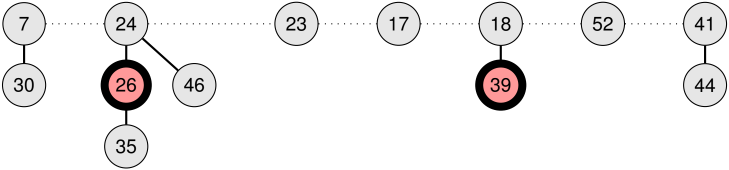
- Delete min ✓
- Meld childen into root list and unmark them ✓



Fibonacci Heap: Extract-Min

Extract-Min

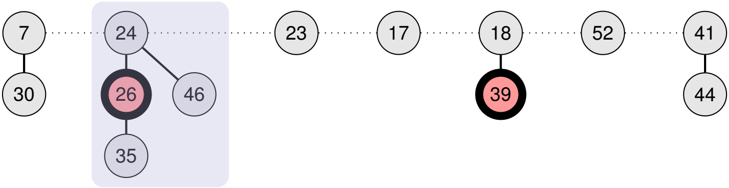
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

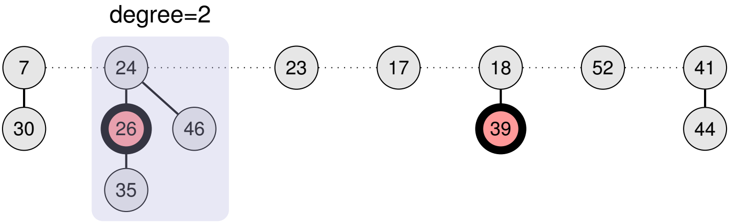
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

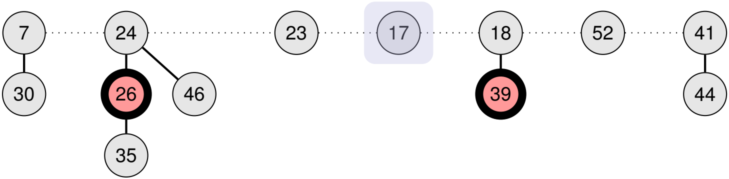
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

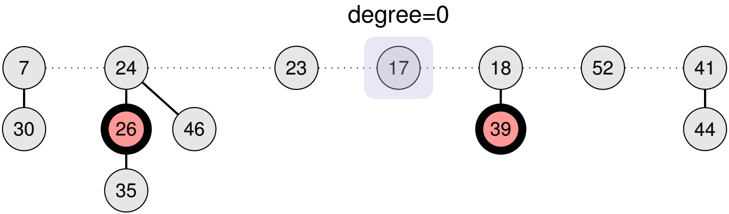
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

EXTRACT-MIN

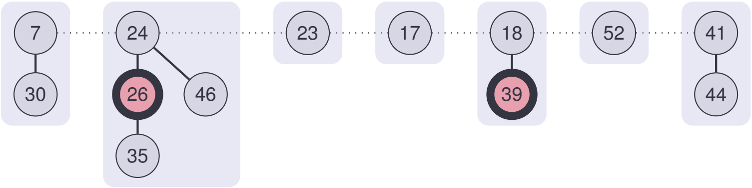
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

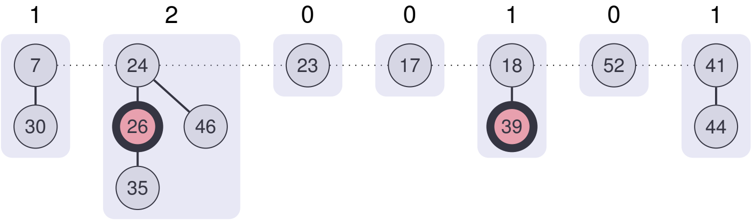
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



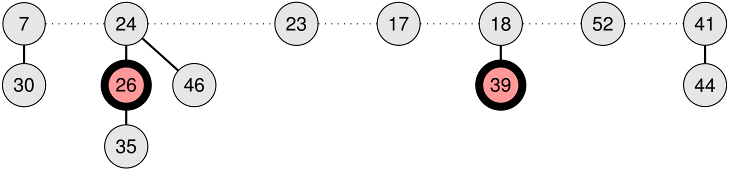
Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

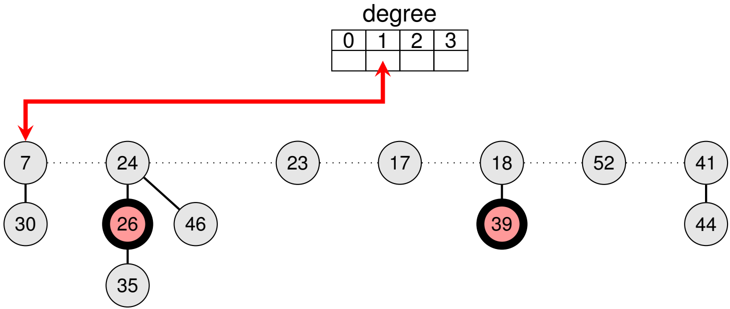
0	1	2	3



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

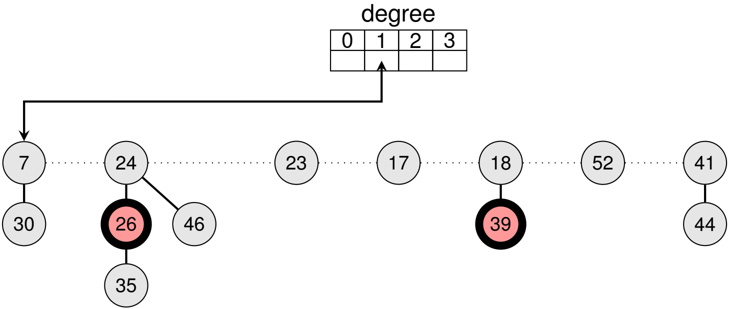
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

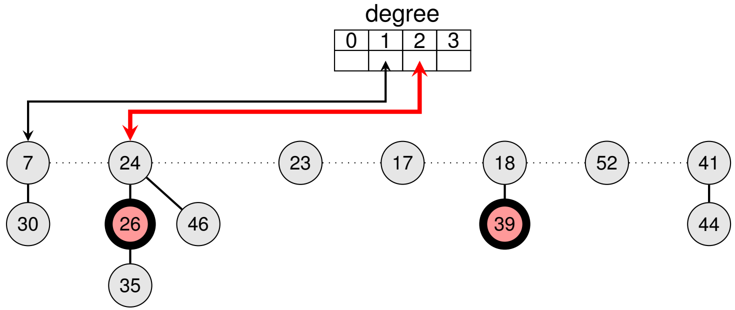
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

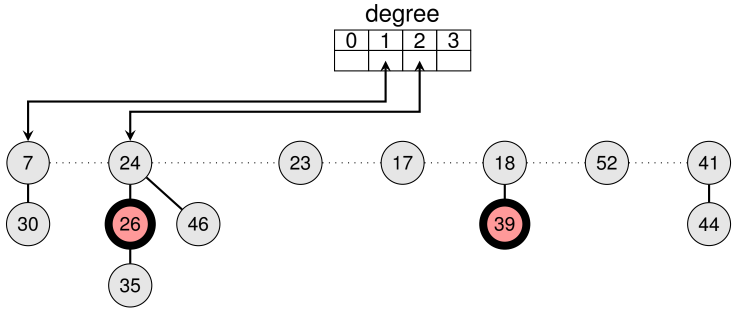
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

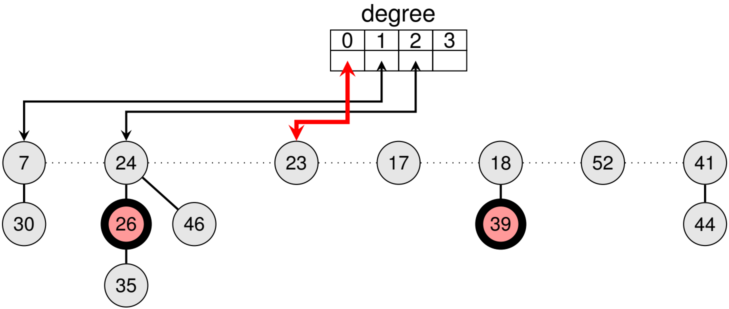
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

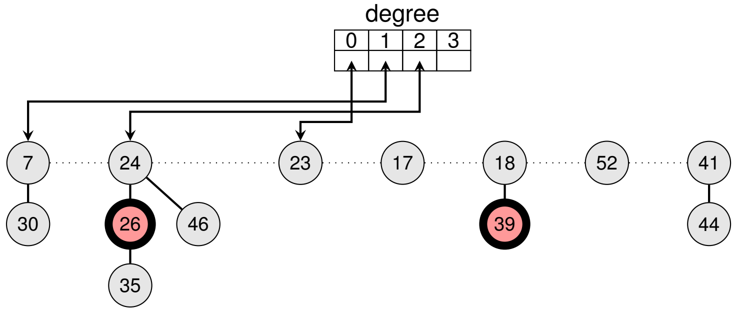
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

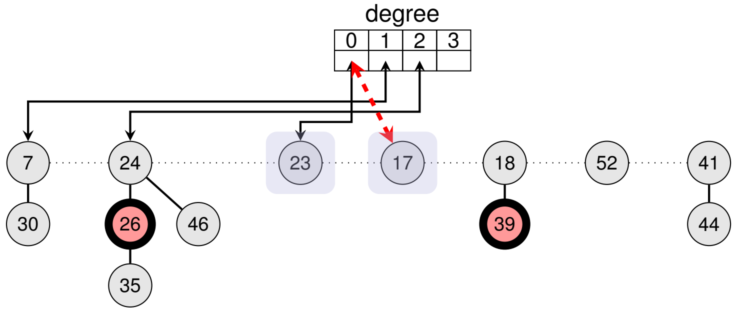
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

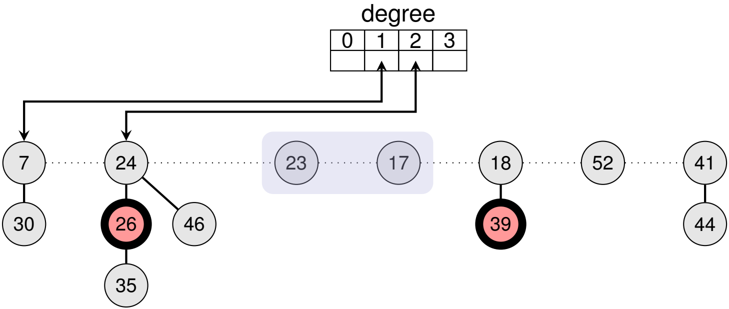
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

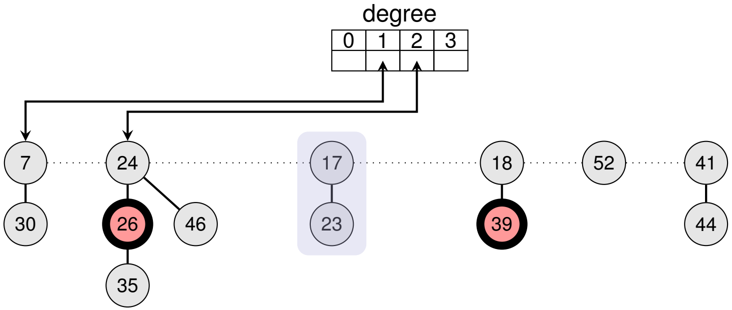
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

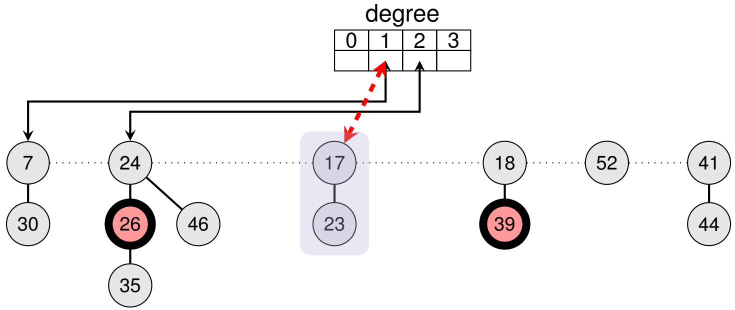
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

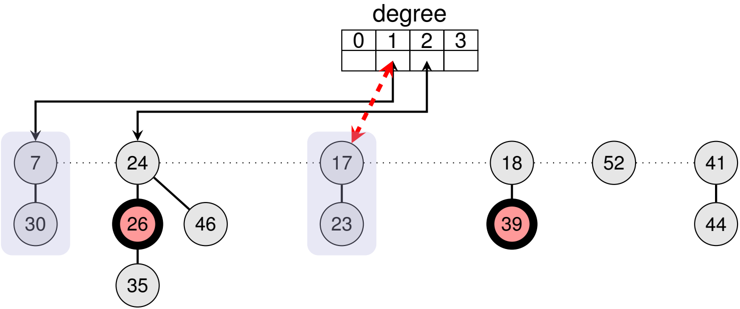
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

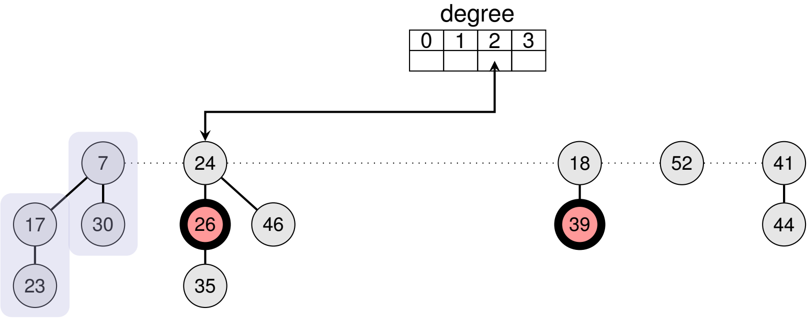
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

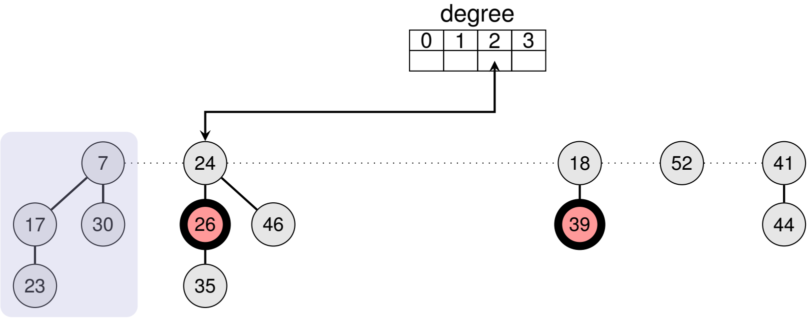
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

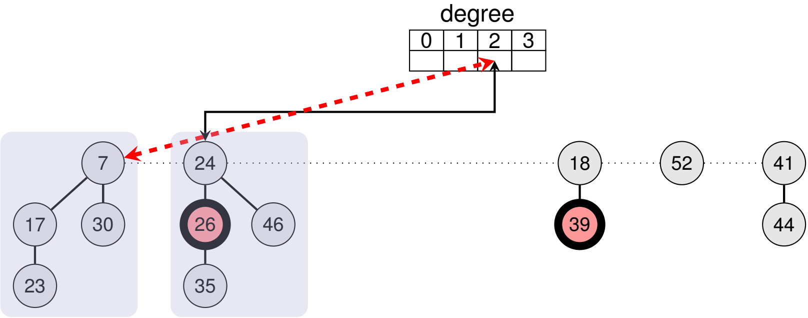
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



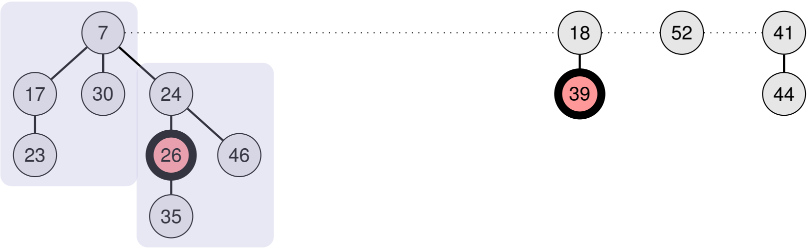
Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

0	1	2	3



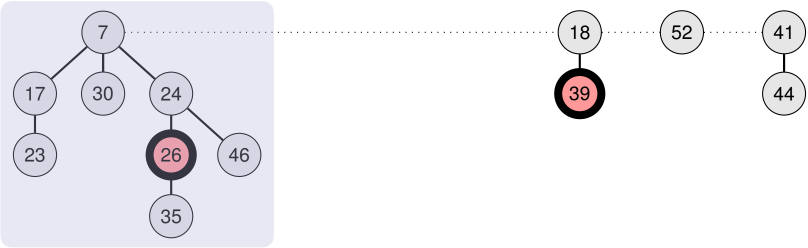
Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

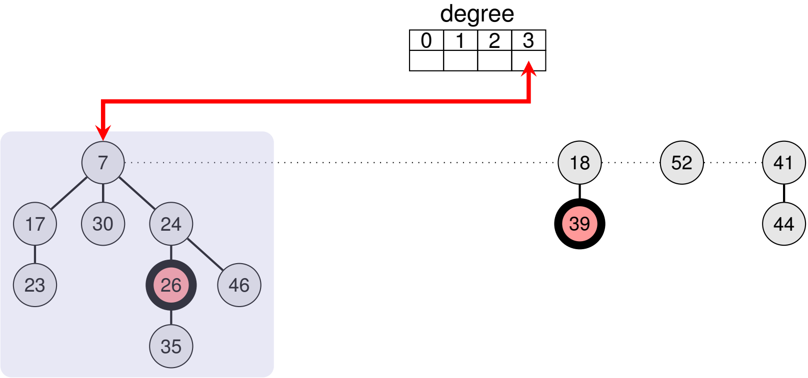
0	1	2	3



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

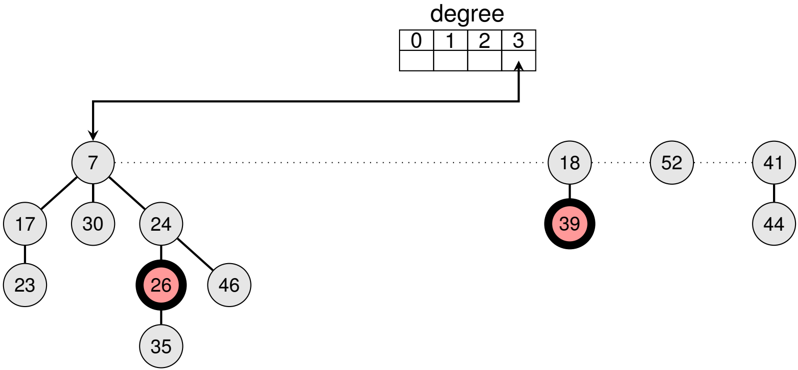
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

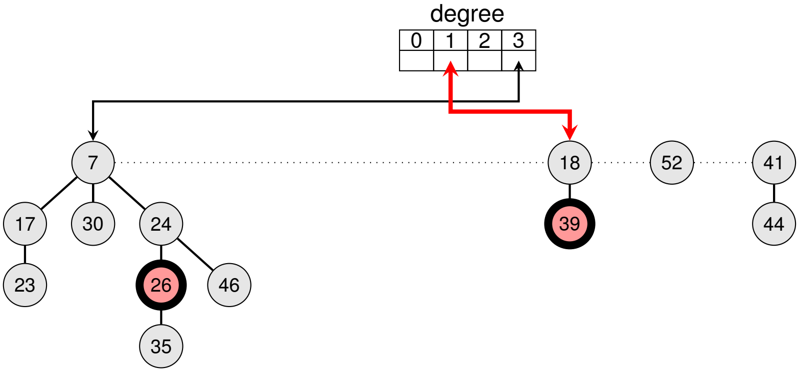
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

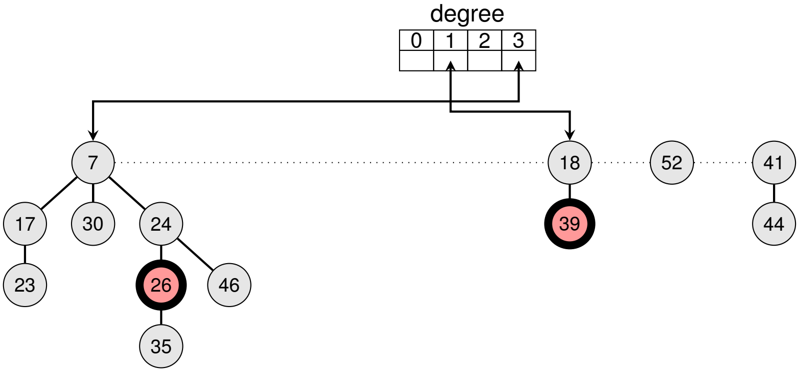
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

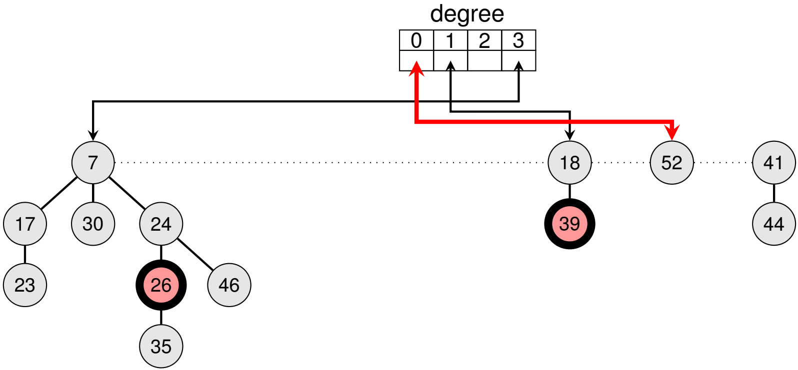
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

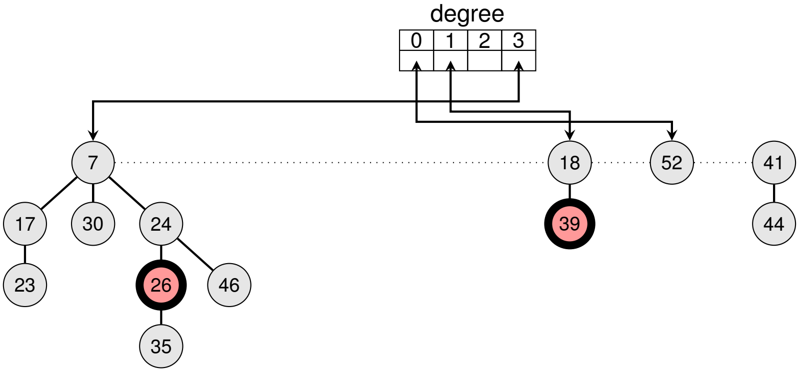
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

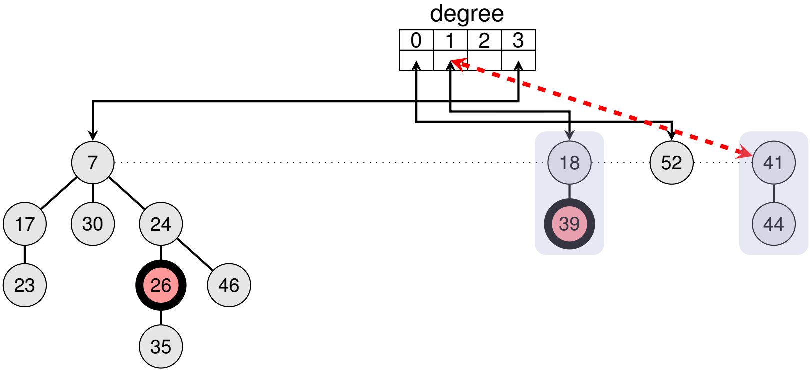
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

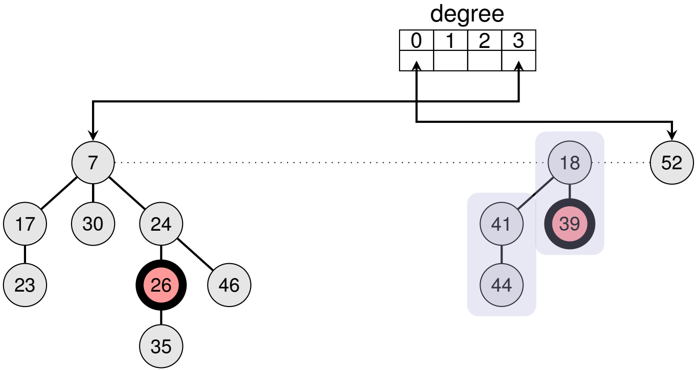
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

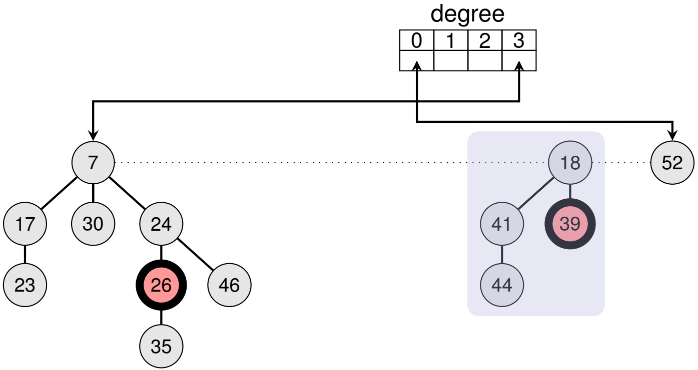
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

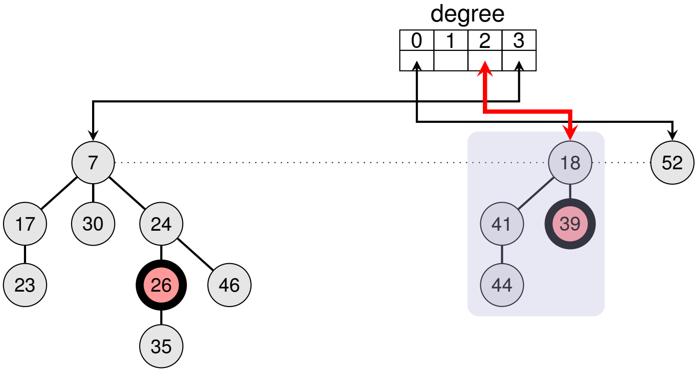
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

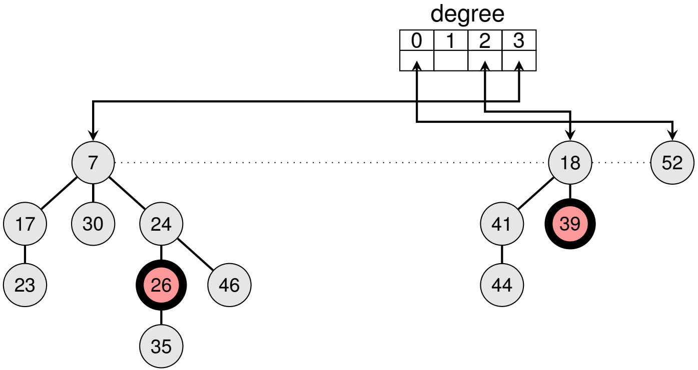
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

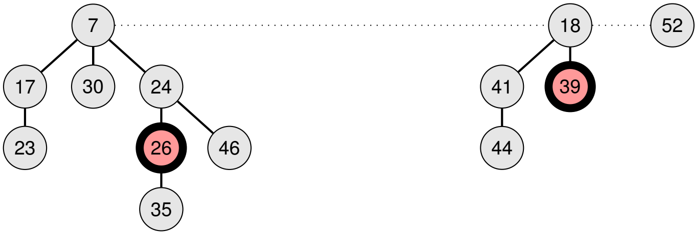
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

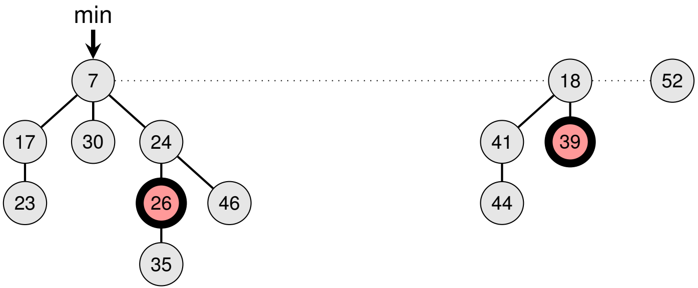
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum



Fibonacci Heap: Extract-Min

Extract-Min

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

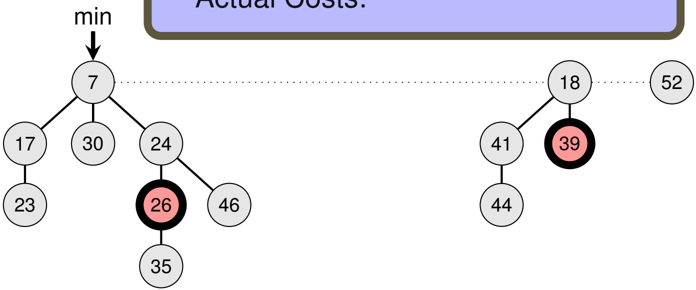


Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Actual Costs:



Fibonacci Heap: EXTRACT-MIN

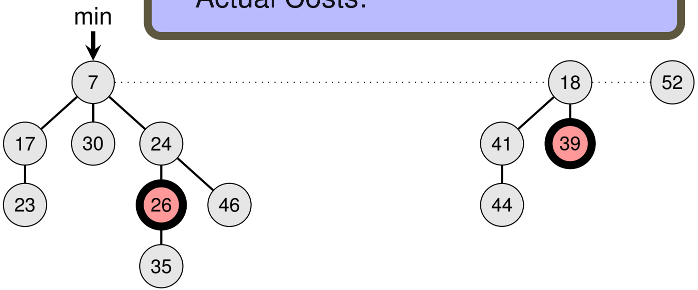
EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size n

Actual Costs:



Fibonacci Heap: EXTRACT-MIN

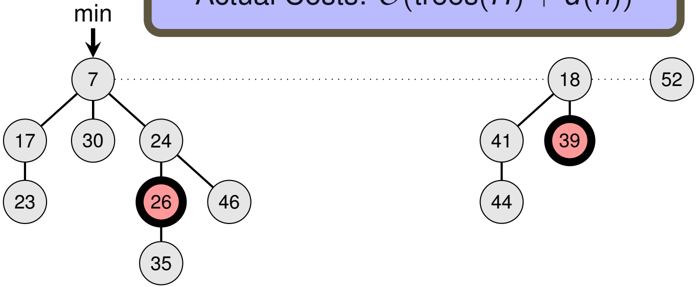
EXTRACT-MIN

- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size n

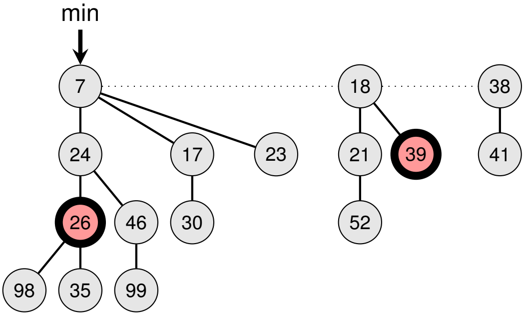
Actual Costs: $\mathcal{O}(\text{trees}(H) + d(n))$



Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

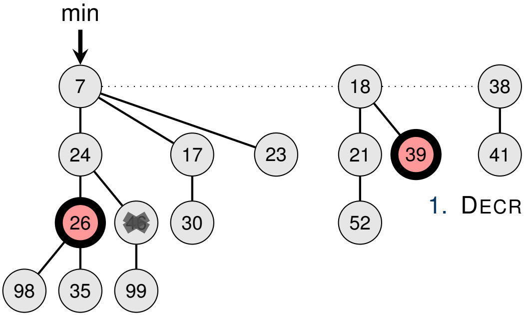
- Decrease the key of x (given by a pointer)



Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)



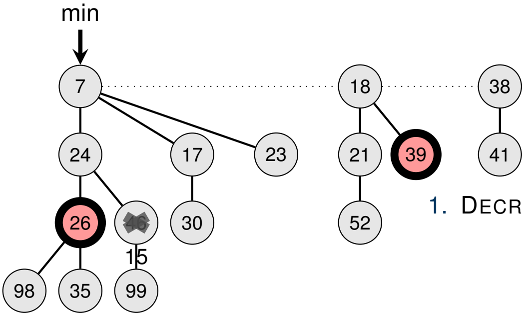
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



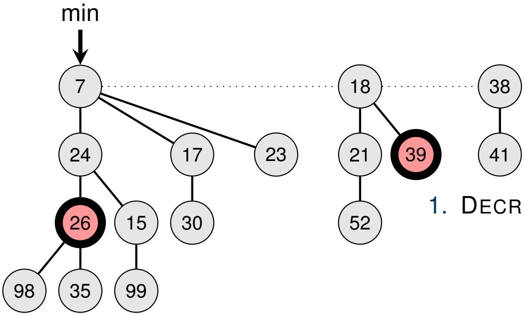
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



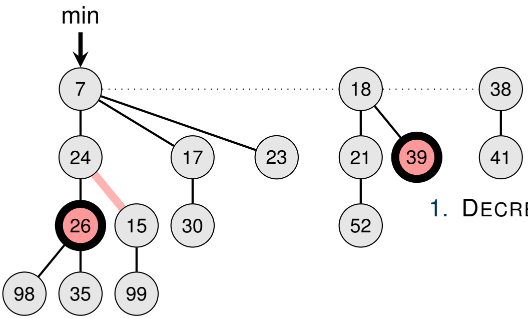
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



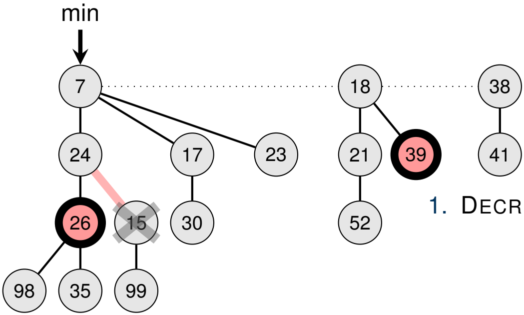
1. DECREASE-KEY 46 $\rightsquigarrow$ 15

Fibonacci Heap: Decrease-Key

Decrease-Key of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list



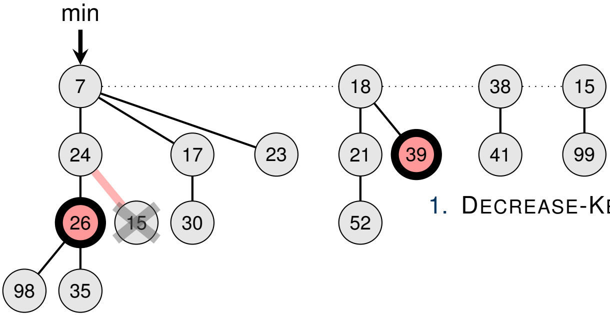
1. Decrease-Key 46 $\rightsquigarrow$ 15

Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

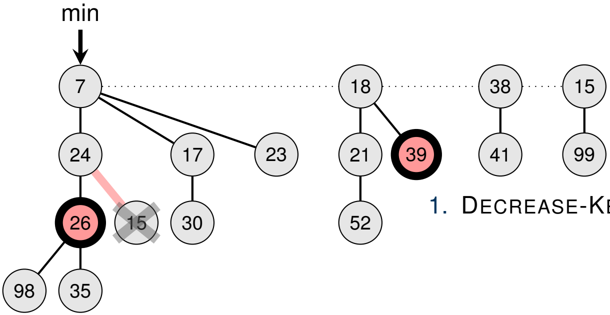
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

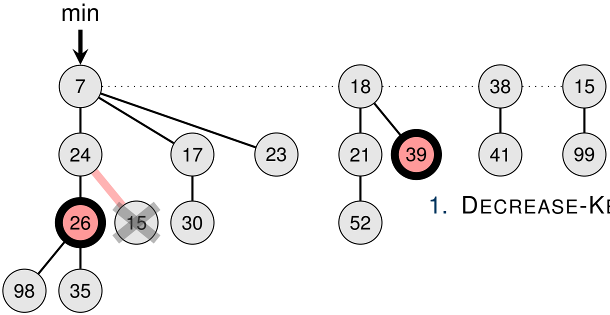
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

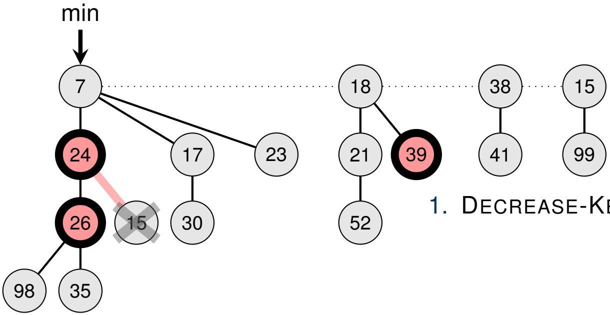
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15

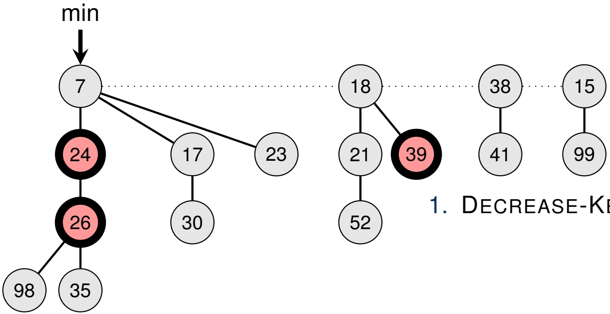
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
    - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓

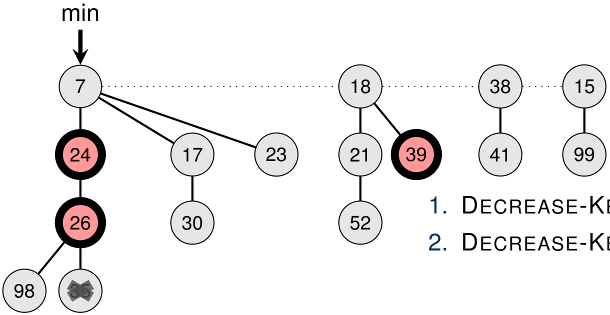
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

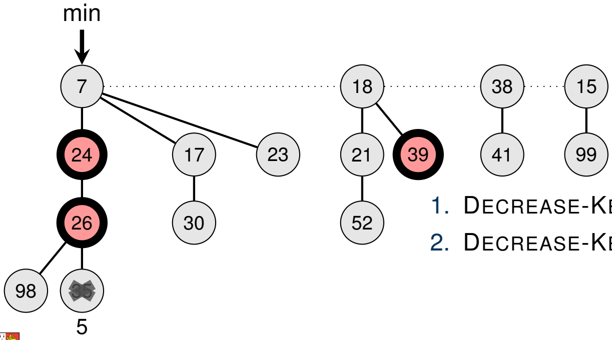
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

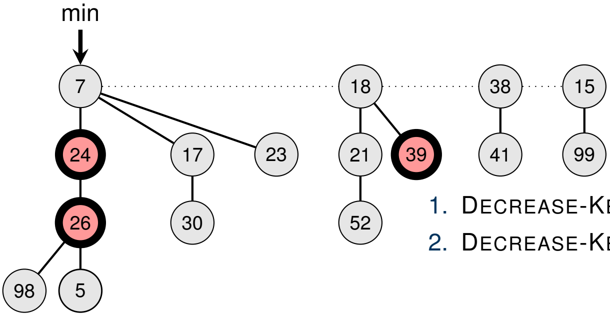
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

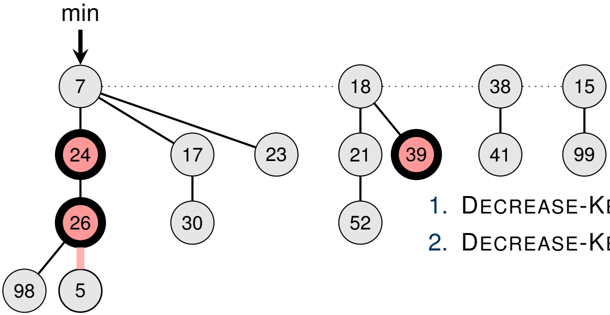
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

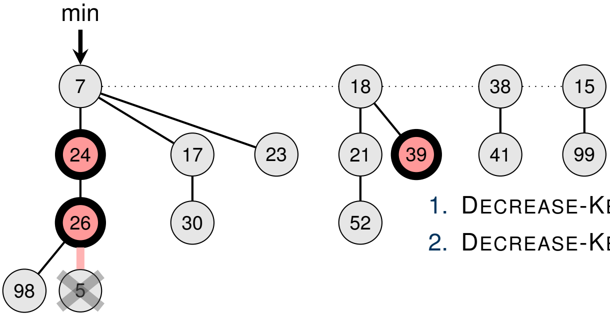
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

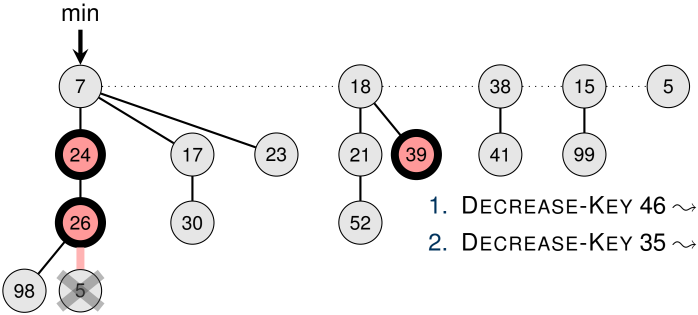
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)



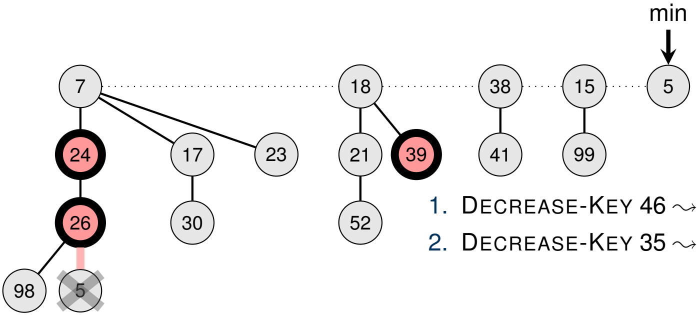
1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked,



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

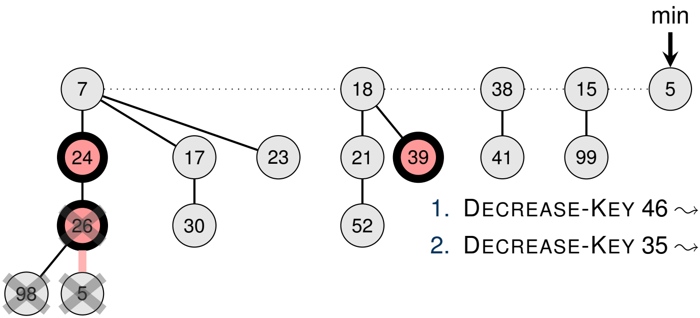
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

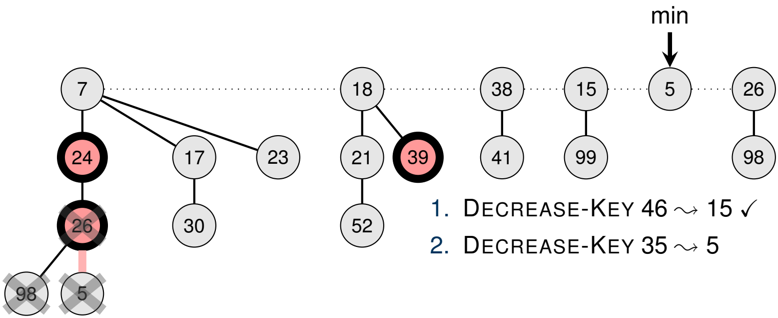
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

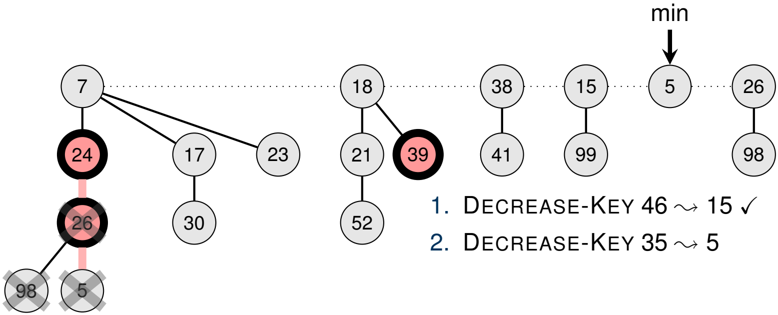
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

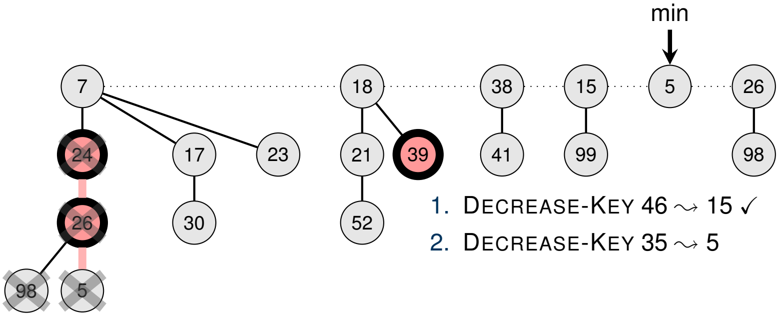
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

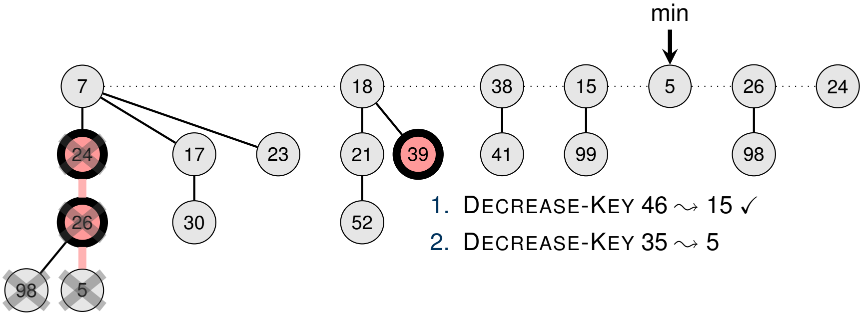
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

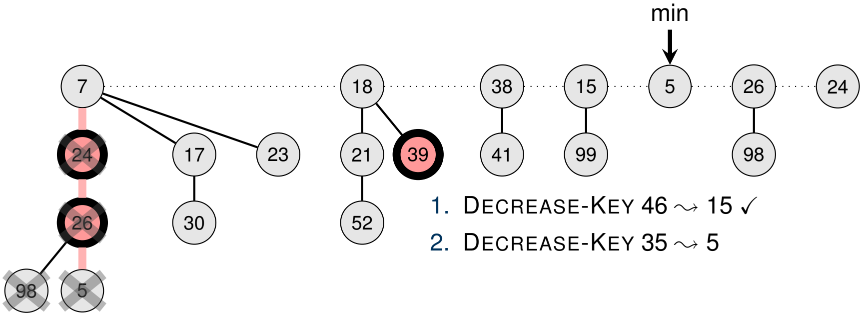
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

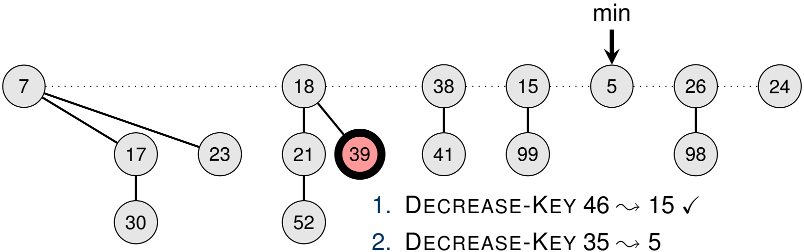
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5

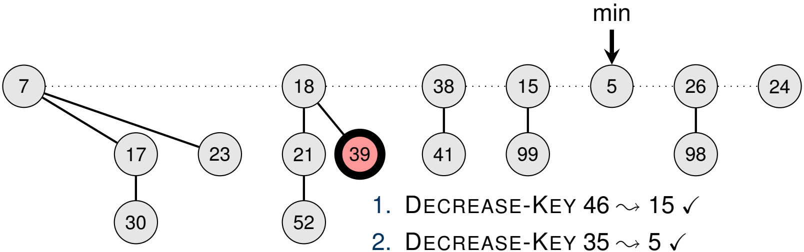
Fibonacci Heap: DECREASE-KEY

DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5 ✓

Fibonacci Heap: DECREASE-KEY

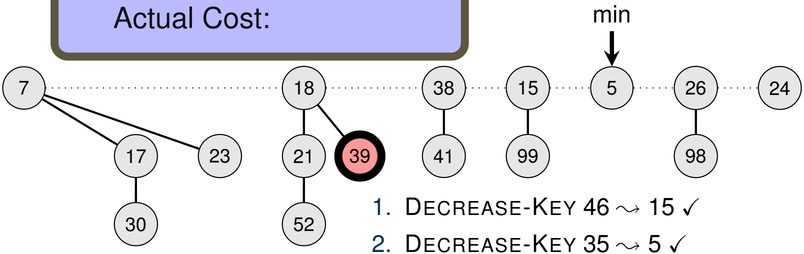
DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

Actual Cost:



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5 ✓

Fibonacci Heap: DECREASE-KEY

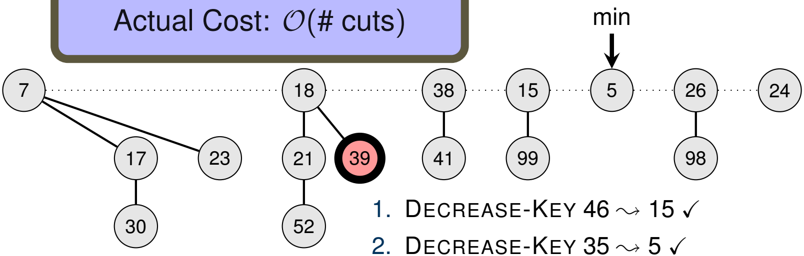
DECREASE-KEY of node x

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

Actual Cost: $\mathcal{O}$(# cuts)



1. DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightsquigarrow$ 5 ✓

Outline

Recap of INSERT, EXTRACT-MIN and DECREASE-KEY

Glimpse at the Analysis

Amortized Analysis

Bounding the Maximum Degree

Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

Amortized Analysis via Potential Method

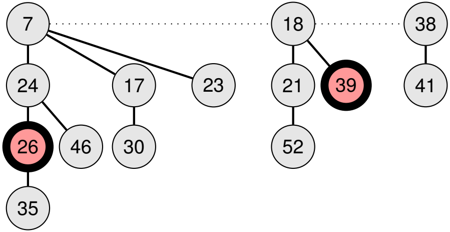
- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

$$\boxed{\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)}$$

Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

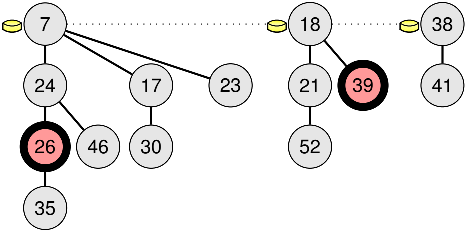
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

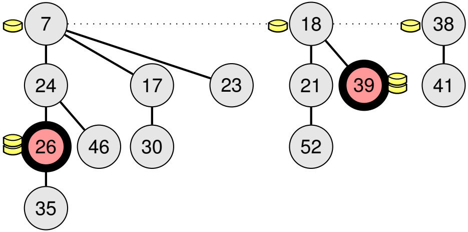
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$

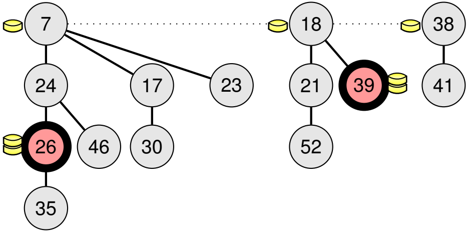
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



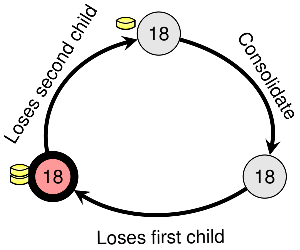
Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$ amortized $\mathcal{O}(d(n))$
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$ amortized $\mathcal{O}(1)$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



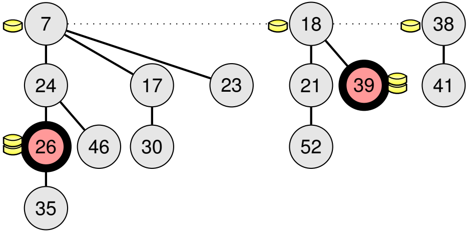
Lifecycle of a node



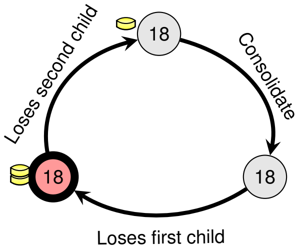
Amortized Analysis via Potential Method

- INSERT: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$ ✓
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$ amortized $\mathcal{O}(d(n))$?
- DECREASE-KEY: actual $\mathcal{O}(\#\text{ cuts}) \leq \mathcal{O}(\text{marks}(H))$ amortized $\mathcal{O}(1)$?

$$\boxed{\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)}$$



Lifecycle of a node



Outline

Recap of INSERT, EXTRACT-MIN and DECREASE-KEY

Glimpse at the Analysis

Amortized Analysis

Bounding the Maximum Degree

Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

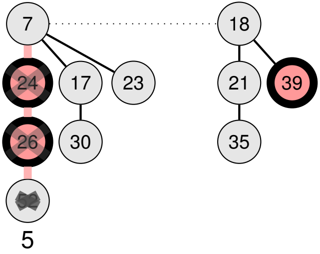
Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x + 1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential



Amortized Analysis of DECREASE-KEY

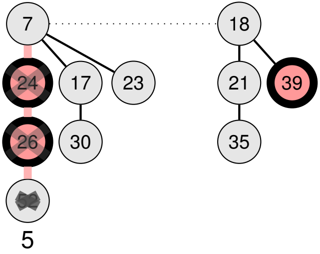
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') =$



Amortized Analysis of DECREASE-KEY

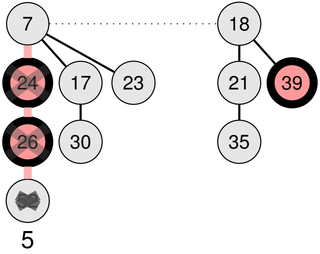
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$



Amortized Analysis of DECREASE-KEY

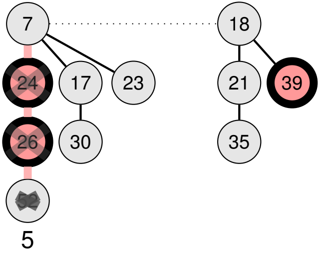
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq$



Amortized Analysis of DECREASE-KEY

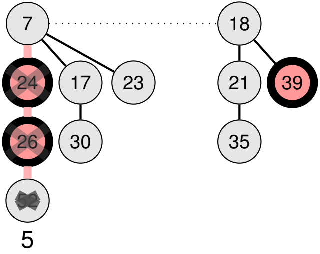
Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$



Amortized Analysis of DECREASE-KEY

Actual Cost

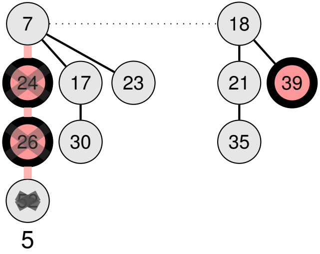
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$



Amortized Analysis of DECREASE-KEY

Actual Cost

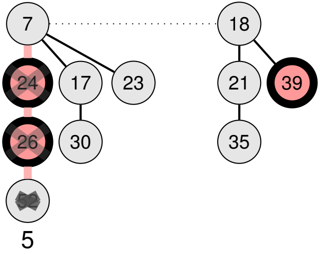
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$



Amortized Cost

$$\widetilde{c_i} = c_i + \Delta\Phi$$

Amortized Analysis of DECREASE-KEY

Actual Cost

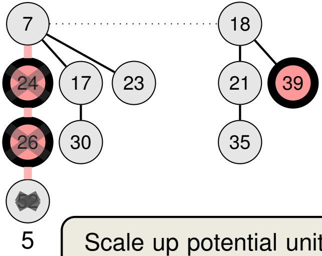
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x+2) = 4 - x.$



Scale up potential units

Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(x+1) + 4 - x$$

Amortized Analysis of DECREASE-KEY

Actual Cost

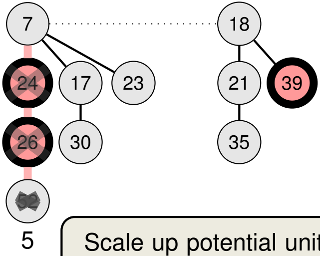
- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \le \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \le x + 2 \cdot (-x + 2) = 4 - x.$



Scale up potential units

Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi \le \mathcal{O}(x+1) + 4 - x = \mathcal{O}(1)$$

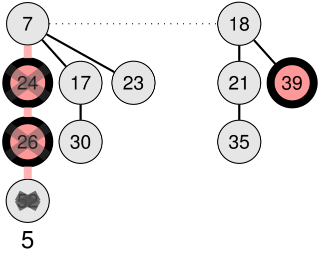
Amortized Analysis of DECREASE-KEY

Actual Cost

- DECREASE-KEY: $\mathcal{O}(x+1)$, where x is the number of cuts.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

First Coin $\rightsquigarrow$ pays cut
Second Coin $\rightsquigarrow$ increase of trees(H)



Change in Potential

- $\text{trees}(H') = \text{trees}(H) + x$
- $\text{marks}(H') \leq \text{marks}(H) - x + 2$

$\Rightarrow \Delta\Phi \leq x + 2 \cdot (-x + 2) = 4 - x.$

Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(x+1) + 4 - x = \mathcal{O}(1)$$

Amortized Analysis of EXTRACT-MIN

Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

Amortized Analysis of EXTRACT-MIN

Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Amortized Analysis of EXTRACT-MIN

Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

Amortized Analysis of EXTRACT-MIN

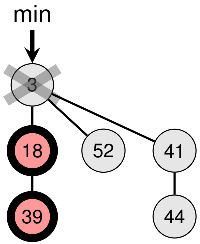
Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H')$? $\text{marks}(H)$



Amortized Analysis of EXTRACT-MIN

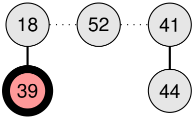
Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \ ? \ \text{marks}(H)$



Amortized Analysis of EXTRACT-MIN

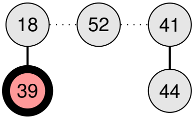
Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$



Amortized Analysis of EXTRACT-MIN

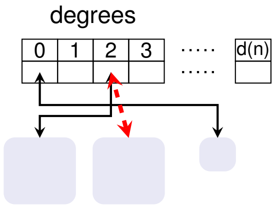
Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq$



Amortized Analysis of EXTRACT-MIN

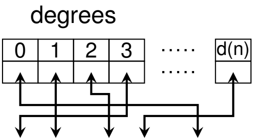
Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq$



Amortized Analysis of EXTRACT-MIN

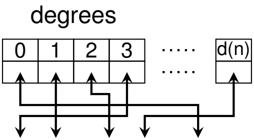
Actual Cost

- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq \ d(n) + 1$



Amortized Analysis of EXTRACT-MIN

Actual Cost

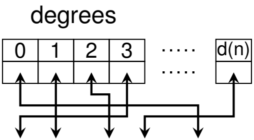
- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq d(n) + 1$

$\Rightarrow \Delta\Phi \leq d(n) + 1 - \text{trees}(H)$



Amortized Analysis of EXTRACT-MIN

Actual Cost

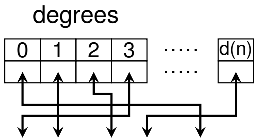
- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq d(n) + 1$

$\Rightarrow$ $\Delta\Phi \leq d(n) + 1 - \text{trees}(H)$



Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi$$

Amortized Analysis of Extract-Min

Actual Cost

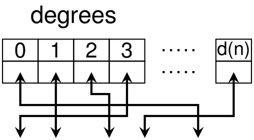
- Extract-Min: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq \; d(n) + 1$

$\Rightarrow \; \Delta\Phi \leq d(n) + 1 - \text{trees}(H)$



Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(\text{trees(H)} + d(n)) + d(n) + 1 - \text{trees(H)}$$

Amortized Analysis of EXTRACT-MIN

Actual Cost

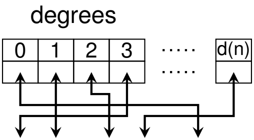
- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq d(n) + 1$

$\Rightarrow$ $\Delta\Phi \leq d(n) + 1 - \text{trees}(H)$



Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(\text{trees(H)} + d(n)) + d(n) + 1 - \text{trees(H)} = \mathcal{O}(d(n))$$

Amortized Analysis of EXTRACT-MIN

Actual Cost

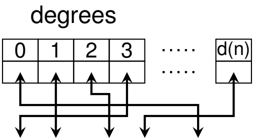
- EXTRACT-MIN: $\mathcal{O}(\text{trees}(H) + d(n))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
- $\text{trees}(H') \leq d(n) + 1$

$\Rightarrow \Delta\Phi \leq d(n) + 1 - \text{trees}(H)$



Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi \leq \mathcal{O}(\text{trees(H)} + d(n)) + d(n) + 1 - \text{trees(H)} = \mathcal{O}(d(n))$$

How to bound $d(n)$?

Outline

Recap of INSERT, EXTRACT-MIN and DECREASE-KEY

Glimpse at the Analysis

Amortized Analysis

Bounding the Maximum Degree

Bounding the Maximum Degree

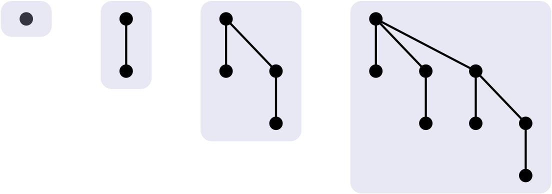
Binomial Heap

Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.

Bounding the Maximum Degree

Binomial Heap

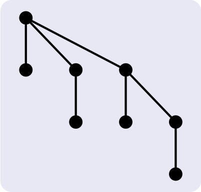
Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.



Bounding the Maximum Degree

Binomial Heap

Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.

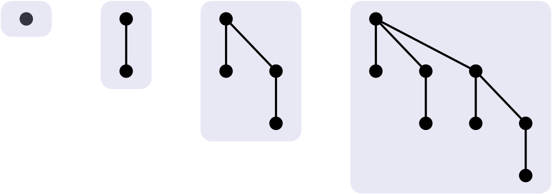


$d = 3, n = 2^3$

Bounding the Maximum Degree

Binomial Heap

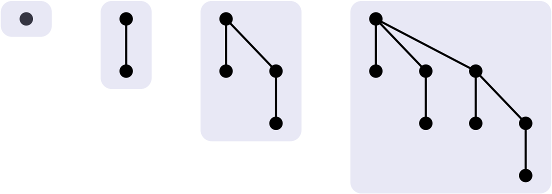
Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.



Bounding the Maximum Degree

Binomial Heap

Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.



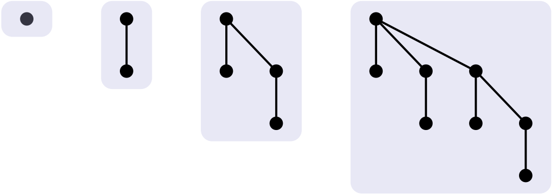
Fibonacci Heap

Not all trees are binomial trees, but still $d(n) \leq \log_\varphi n$, where $\varphi \approx 1.62$.

Bounding the Maximum Degree

Binomial Heap

Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.



Fibonacci Heap

Not all trees are binomial trees, but still $d(n) \leq \log_\varphi n$, where $\varphi \approx 1.62$.

▸ Skip Analysis

Lower Bounding Degrees of Children

$$d(n) \leq \log_{\varphi} n$$

Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

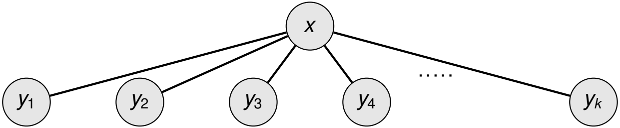
- Consider any node x of degree k (not necessarily a root) at the final state

Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment



Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

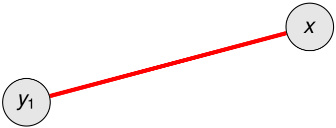
- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

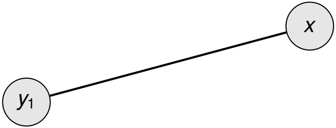


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

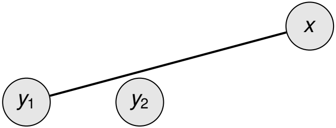


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

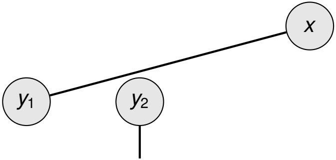


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

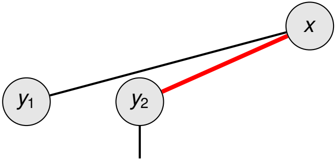


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

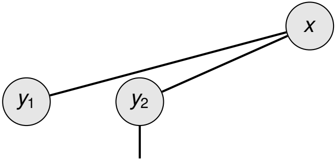


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

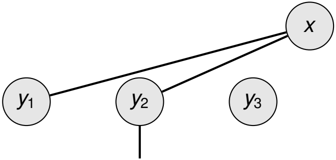


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

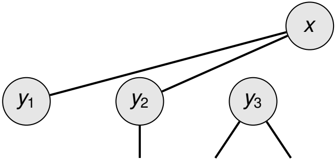


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

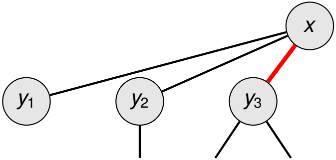


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

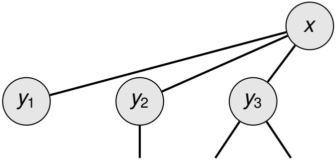


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

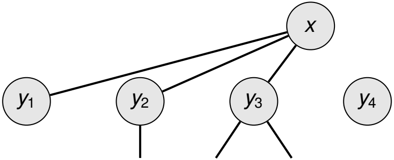


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

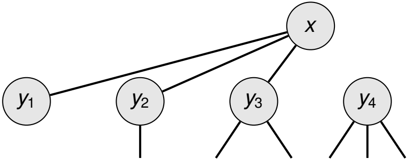


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

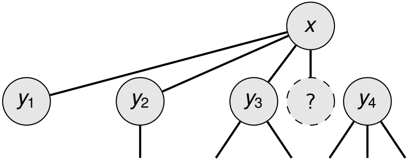


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

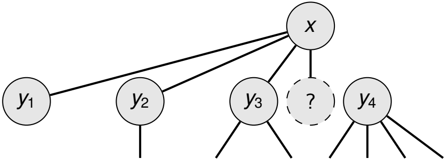


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

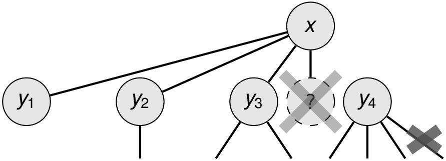


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

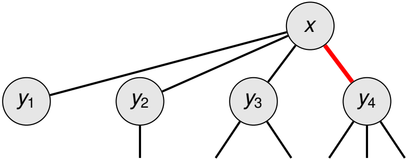


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

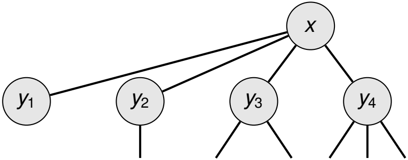


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

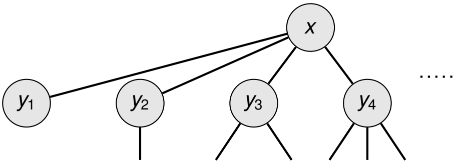


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

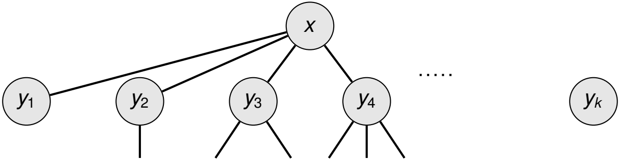


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

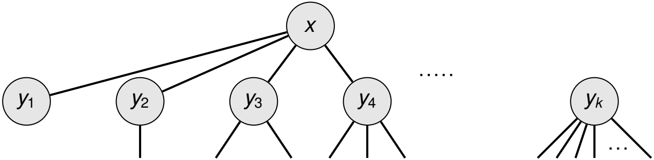


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

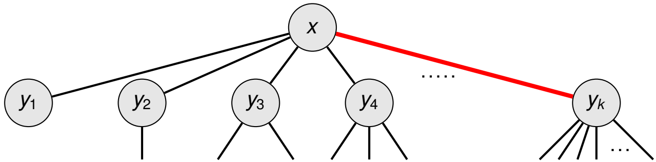


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

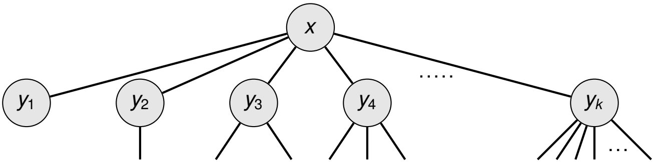


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

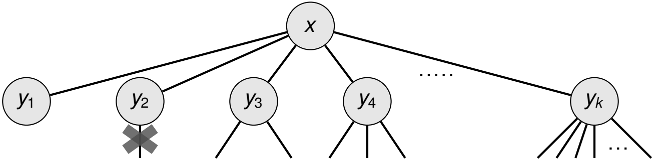


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

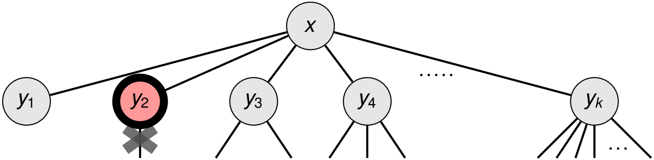


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

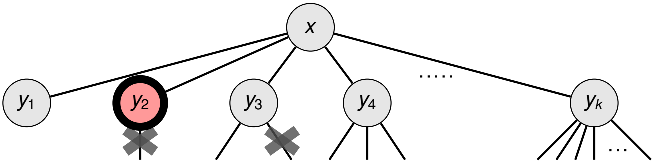


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

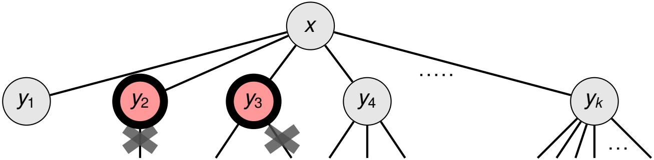


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

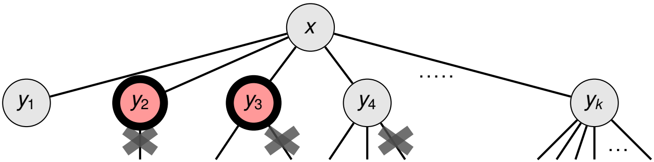


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

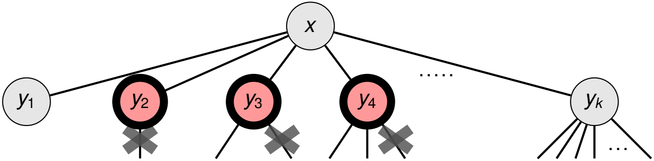


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

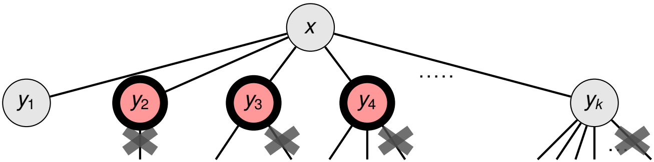


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

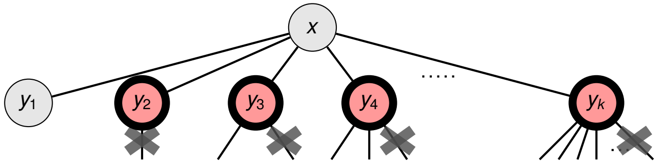


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_{\varphi} n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment

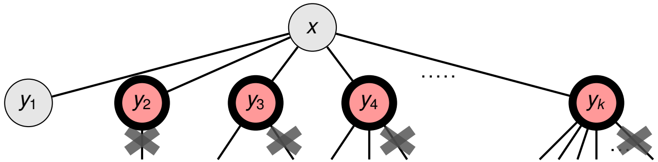


Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment and $d_1, d_2, \ldots, d_k$ be their degrees



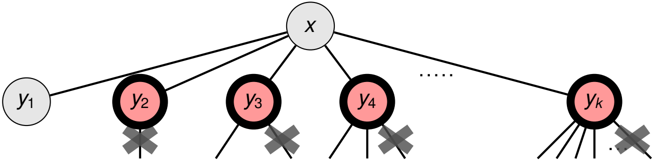
Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

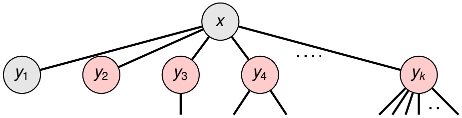
$$d(n) \leq \log_\varphi n$$

- Consider any node x of degree k (not necessarily a root) at the final state
- Let $y_1, y_2, \ldots, y_k$ be the children in the order of attachment and $d_1, d_2, \ldots, d_k$ be their degrees

$\Rightarrow$ $\boxed{\forall 1 \leq i \leq k: \quad d_i \geq i - 2}$

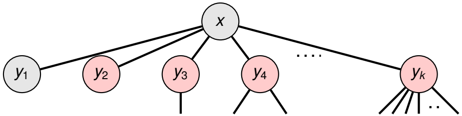


From Degrees to Minimum Subtree Sizes



$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

From Degrees to Minimum Subtree Sizes

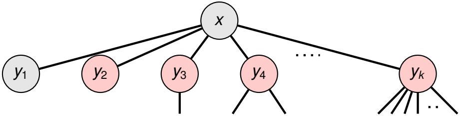


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

From Degrees to Minimum Subtree Sizes



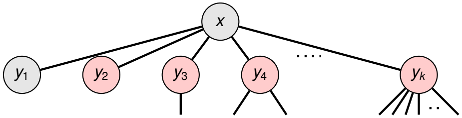
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$

From Degrees to Minimum Subtree Sizes



$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

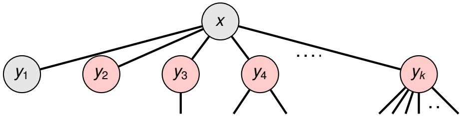
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$

- 0

From Degrees to Minimum Subtree Sizes



$$\forall 1 \leq i \leq k : \quad d_i \geq i - 2$$

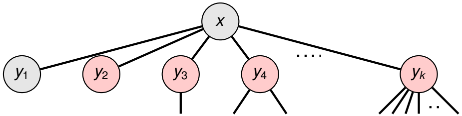
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$ $\qquad$ $N(1)$

- 0

From Degrees to Minimum Subtree Sizes



$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

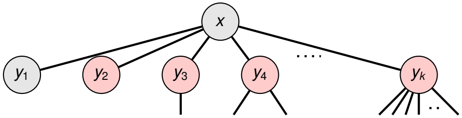
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$ $N(1)$

- 0
- 1

From Degrees to Minimum Subtree Sizes



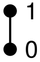
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

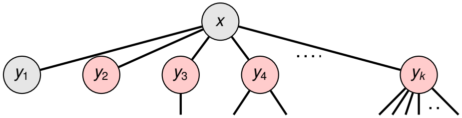
Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$

$N(1)$



From Degrees to Minimum Subtree Sizes

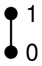


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

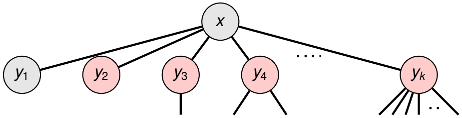
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$ $N(1)$ $N(2)$



From Degrees to Minimum Subtree Sizes

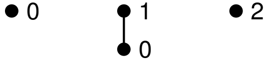


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

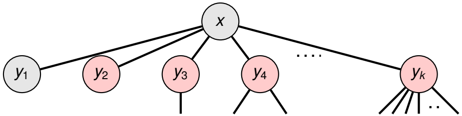
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$ $N(1)$ $N(2)$



From Degrees to Minimum Subtree Sizes

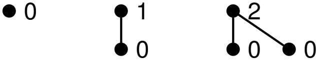


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

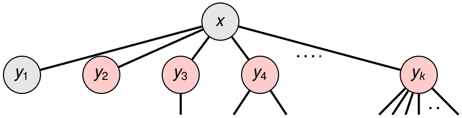
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$ $N(1)$ $N(2)$



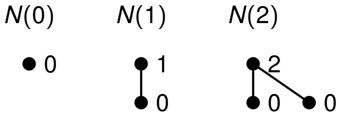
From Degrees to Minimum Subtree Sizes



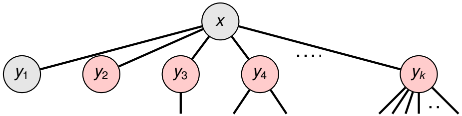
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



From Degrees to Minimum Subtree Sizes

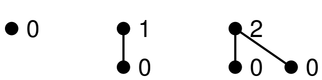


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

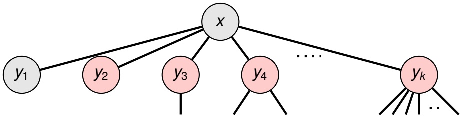
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0)$ $N(1)$ $N(2)$ $N(3)$



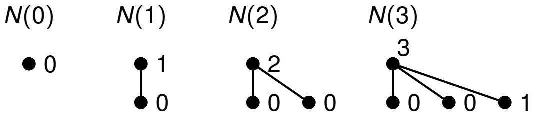
From Degrees to Minimum Subtree Sizes



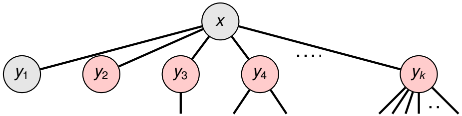
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



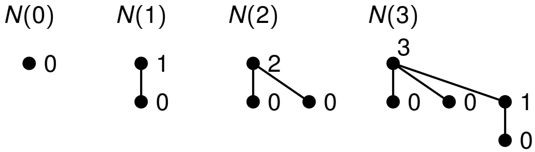
From Degrees to Minimum Subtree Sizes



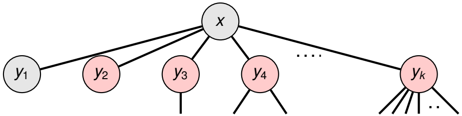
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



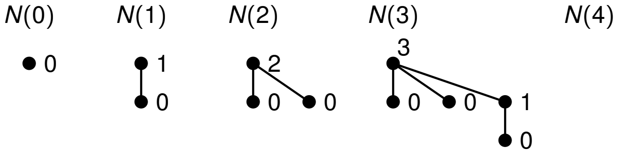
From Degrees to Minimum Subtree Sizes



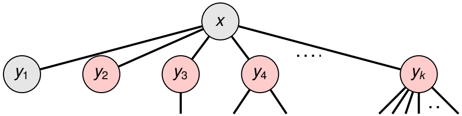
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



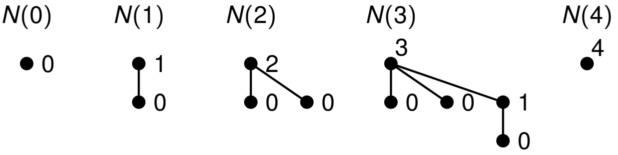
From Degrees to Minimum Subtree Sizes



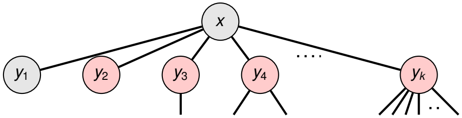
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



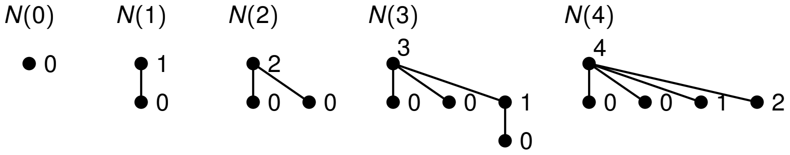
From Degrees to Minimum Subtree Sizes



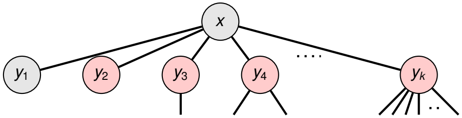
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



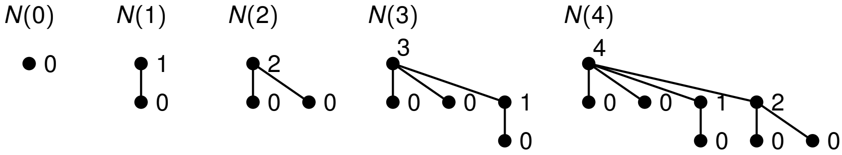
From Degrees to Minimum Subtree Sizes



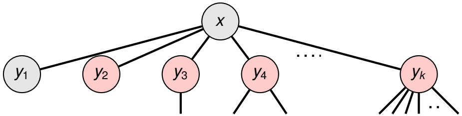
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.



From Degrees to Minimum Subtree Sizes

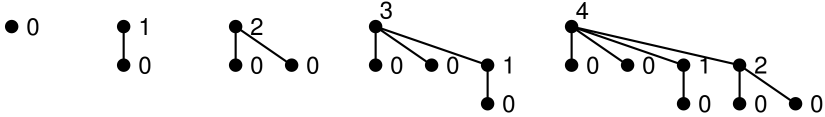


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

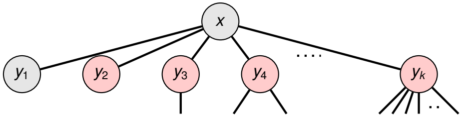
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $\quad N(1) \quad N(2) \quad N(3) \quad N(4)$



From Degrees to Minimum Subtree Sizes

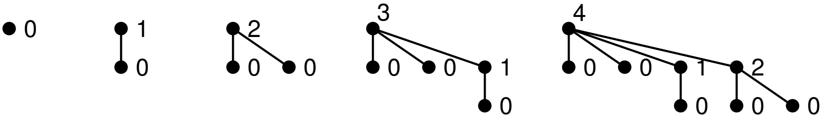


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

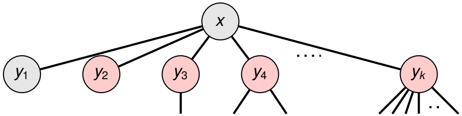
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $N(1) = 2$ $N(2)$ $N(3)$ $N(4)$



From Degrees to Minimum Subtree Sizes

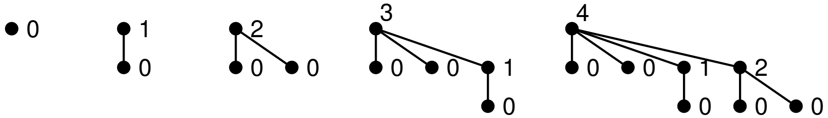


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

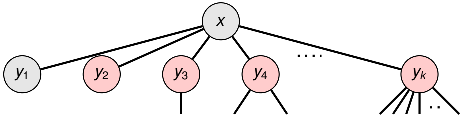
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $N(1) = 2$ $N(2) = 3$ $N(3)$ $N(4)$



From Degrees to Minimum Subtree Sizes

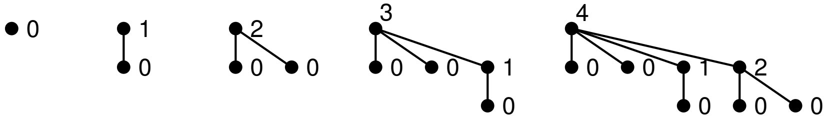


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

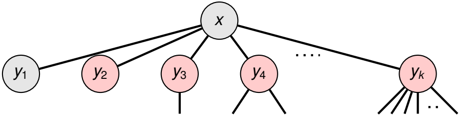
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $\quad N(1) = 2$ $\quad N(2) = 3$ $\quad N(3) = 5$ $\quad N(4)$



From Degrees to Minimum Subtree Sizes

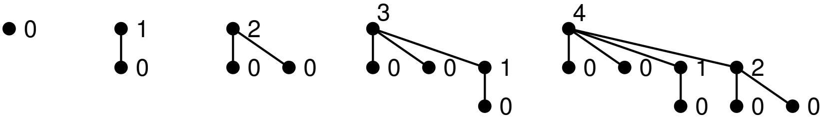


$$\forall 1 \leq i \leq k: \quad d_i \geq i-2$$

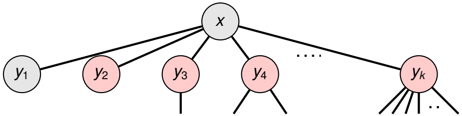
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $N(1) = 2$ $N(2) = 3$ $N(3) = 5$ $N(4) = 8$



From Degrees to Minimum Subtree Sizes

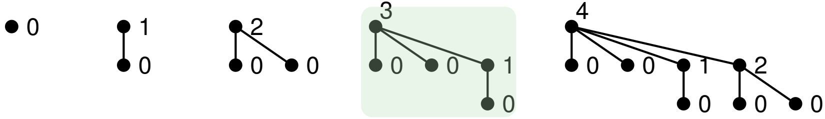


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

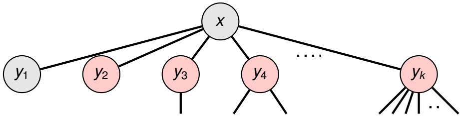
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $N(1) = 2$ $N(2) = 3$ $N(3) = 5$ $N(4) = 8$



From Degrees to Minimum Subtree Sizes

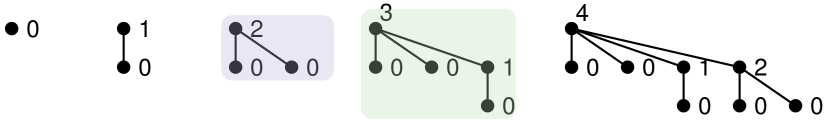


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

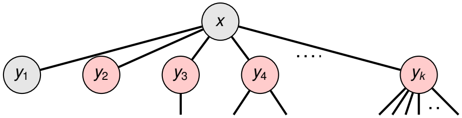
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $\quad N(1) = 2$ $\quad N(2) = 3$ $\quad N(3) = 5$ $\quad N(4) = 8$



From Degrees to Minimum Subtree Sizes

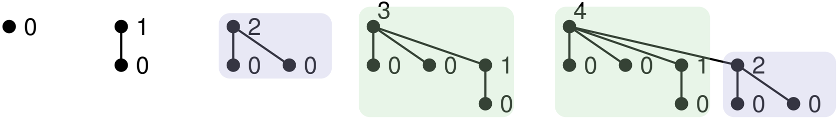


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

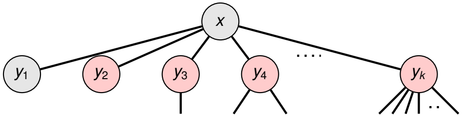
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $\quad N(1) = 2$ $\quad N(2) = 3$ $\quad N(3) = 5$ $\quad N(4) = 8$



From Degrees to Minimum Subtree Sizes

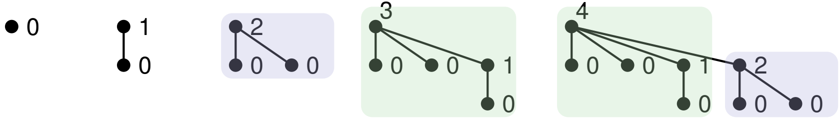


$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

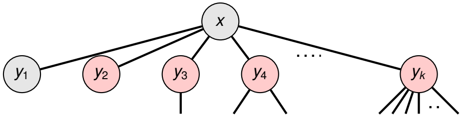
Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$N(0) = 1$ $\quad N(1) = 2$ $\quad N(2) = 3$ $\quad N(3) = 5$ $\quad N(4) = 8 = 5 + 3$



From Degrees to Minimum Subtree Sizes



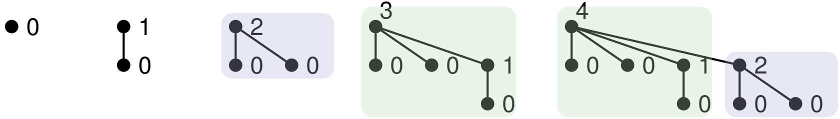
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$$N(k) = F(k+2)?$$

$N(0) = 1$ $N(1) = 2$ $N(2) = 3$ $N(3) = 5$ $N(4) = 8 = 5 + 3$



From Minimum Subtree Sizes to Fibonacci Numbers

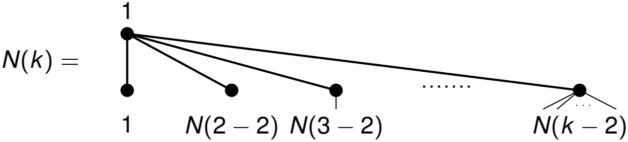
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

$$N(k) \ = \ F(k+2)?$$

From Minimum Subtree Sizes to Fibonacci Numbers

$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

$$N(k) = F(k+2)?$$

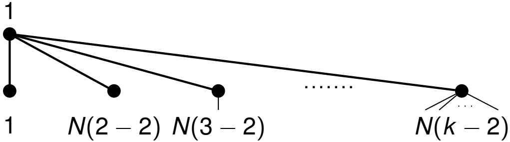


From Minimum Subtree Sizes to Fibonacci Numbers

$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$

$N(k) \ = \ F(k+2)?$

$N(k) =$



$$
\begin{aligned}
N(k) &= 1 + 1 + N(2-2) + N(3-2) + \cdots + N(k-2) \\
&= 1 + 1 + \sum_{\ell=0}^{k-2} N(\ell) \\
&= 1 + 1 + \sum_{\ell=0}^{k-3} N(\ell) + N(k-2) \\
&= N(k-1) + N(k-2) \\
&= F(k+1) + F(k) = F(k+2) \qquad \square
\end{aligned}
$$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓
- Inductive Step ($k \geq 2$):

$$F(k+2) =$$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓
- Inductive Step ($k \geq 2$):

$$F(k+2) = F(k+1) + F(k)$$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓
- Inductive Step $(k \geq 2)$:

$$\begin{aligned} F(k+2) &= F(k+1) + F(k) \\ &\geq \varphi^{k-1} + \varphi^{k-2} \qquad \text{(by the inductive hypothesis)} \end{aligned}$$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓
- Inductive Step $(k \geq 2)$:

$$\begin{aligned} F(k+2) &= F(k+1) + F(k) \\ &\geq \varphi^{k-1} + \varphi^{k-2} && \text{(by the inductive hypothesis)} \\ &= \varphi^{k-2} \cdot (\varphi + 1) \end{aligned}$$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓
- Inductive Step ($k \geq 2$):

$$\begin{aligned} F(k+2) &= F(k+1) + F(k) \\ &\geq \varphi^{k-1} + \varphi^{k-2} && \text{(by the inductive hypothesis)} \\ &= \varphi^{k-2} \cdot (\varphi + 1) \\ &= \varphi^{k-2} \cdot \varphi^2 && (\varphi^2 = \varphi + 1) \end{aligned}$$

Exponential Growth of Fibonacci Numbers

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fib. number satisfies $F(k+2) \geq \varphi^k$, where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

Fibonacci Numbers grow at least exponentially fast in k.

Proof by induction on k:

- Base $k = 0$: $F(2) = 1$ and $\varphi^0 = 1$ ✓
- Base $k = 1$: $F(3) = 2$ and $\varphi^1 \approx 1.619 < 2$ ✓
- Inductive Step ($k \geq 2$):

$$\begin{aligned}
F(k+2) &= F(k+1) + F(k) \\
&\geq \varphi^{k-1} + \varphi^{k-2} && \text{(by the inductive hypothesis)} \\
&= \varphi^{k-2} \cdot (\varphi + 1) \\
&= \varphi^{k-2} \cdot \varphi^2 && (\varphi^2 = \varphi + 1) \\
&= \varphi^k && \square
\end{aligned}$$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\mathcal{O}(d(n))$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\mathcal{O}(d(n))$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

$N(k)$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\mathcal{O}(d(n))$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

$$N(k) = F(k+2)$$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\mathcal{O}(d(n))$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

$$N(k) = F(k+2) \geq \varphi^k$$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\mathcal{O}(d(n))$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

$$n \geq N(k) = F(k+2) \geq \varphi^k$$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\mathcal{O}(d(n))$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

$$
\begin{aligned}
& n \geq N(k) = F(k+2) \geq \varphi^k \\
\Rightarrow \quad & \log_\varphi n \geq k
\end{aligned}
$$

Putting the Pieces Together

Amortized Analysis

- INSERT: amortized cost $\mathcal{O}(1)$
- EXTRACT-MIN: amortized cost $\cancel{\mathcal{O}(d(n))}$ $\mathcal{O}(\log n)$
- DECREASE-KEY: amortized cost $\mathcal{O}(1)$

$$
\begin{aligned}
& n \geq N(k) = F(k+2) \geq \varphi^k \\
\Rightarrow \quad & \log_{\varphi} n \geq k
\end{aligned}
$$

What if we don't have marked nodes?

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(1)$

What if we don't have marked nodes?

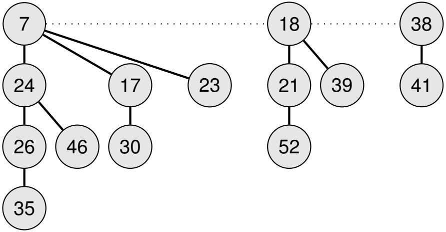
- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(1)$

$$\Phi(H) = \text{trees}(H)$$

What if we don't have marked nodes?

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(1)$

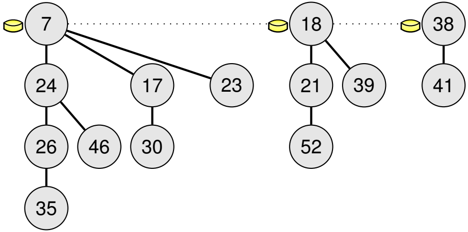
$$\Phi(H) = \text{trees}(H)$$



What if we don't have marked nodes?

- INSERT: actual $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$
- DECREASE-KEY: actual $\mathcal{O}(1)$

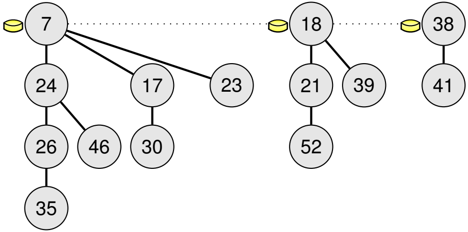
$$\Phi(H) = \text{trees}(H)$$



What if we don't have marked nodes?

- INSERT: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$ amortized $\mathcal{O}(d(n))$
- DECREASE-KEY: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$

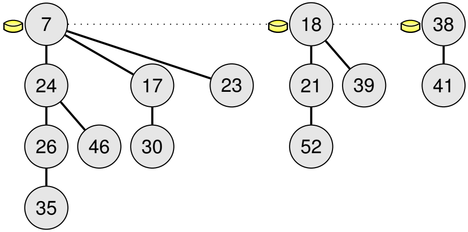
$$\Phi(H) = \text{trees}(H)$$



What if we don't have marked nodes?

- INSERT: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$
- EXTRACT-MIN: actual $\mathcal{O}(\text{trees}(H) + d(n))$ amortized $\mathcal{O}(d(n)) \neq \mathcal{O}(\log n)$
- DECREASE-KEY: actual $\mathcal{O}(1)$ amortized $\mathcal{O}(1)$

$$\Phi(H) = \text{trees}(H)$$



Summary

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Summary

Can we perform EXTRACT-MIN in $o(\log n)$?

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Summary

If this was possible, then there would be a sorting algorithm with runtime $o(n \log n)$!

Can we perform EXTRACT-MIN in $o(\log n)$?

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Summary

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Summary

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

DELETE = DECREASE-KEY + EXTRACT-MIN

Summary

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

DELETE = DECREASE-KEY + EXTRACT-MIN

EXTRACT-MIN = MIN + DELETE

Summary

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Crucial for many applications including shortest paths and minimum spanning trees!

Recent Studies

- Fibonacci Numbers were discovered >800 years ago
- Fibonacci Heaps were developed by Fredman and Tarjan in 1984

Recent Studies

- Fibonacci Numbers were discovered >800 years ago
- Fibonacci Heaps were developed by Fredman and Tarjan in 1984

Brodal, Lagogiannis, Tarjan: Strict Fibonacci Heap (STOC'12)

Strict Fibonacci Heap:

- pointer-based heap implementation similar to Fibonacci Heaps
- achieves the same cost as Fibonacci Heaps, but actual costs!

Recent Studies

- Fibonacci Numbers were discovered >800 years ago
- Fibonacci Heaps were developed by Fredman and Tarjan in 1984

Brodal, Lagogiannis, Tarjan: Strict Fibonacci Heap (STOC'12)

Strict Fibonacci Heap:

- pointer-based heap implementation similar to Fibonacci Heaps
- achieves the same cost as Fibonacci Heaps, but actual costs!

Li, Peebles: Replacing Mark Bits with Randomness in Fibonacci Heap (ICALP'15)

- Queries to marked bits are intercepted and responded with a random bit

Recent Studies

- Fibonacci Numbers were discovered >800 years ago
- Fibonacci Heaps were developed by Fredman and Tarjan in 1984

Brodal, Lagogiannis, Tarjan: Strict Fibonacci Heap (STOC'12)

Strict Fibonacci Heap:

- pointer-based heap implementation similar to Fibonacci Heaps
- achieves the same cost as Fibonacci Heaps, but actual costs!

Li, Peebles: Replacing Mark Bits with Randomness in Fibonacci Heap (ICALP'15)

- Queries to marked bits are intercepted and responded with a random bit
- several lower bounds on the amortized cost in terms of the size of the heap **and** the number of operations

Recent Studies

- Fibonacci Numbers were discovered >800 years ago
- Fibonacci Heaps were developed by Fredman and Tarjan in 1984

Brodal, Lagogiannis, Tarjan: Strict Fibonacci Heap (STOC'12)

Strict Fibonacci Heap:

- pointer-based heap implementation similar to Fibonacci Heaps
- achieves the same cost as Fibonacci Heaps, but actual costs!

Li, Peebles: Replacing Mark Bits with Randomness in Fibonacci Heap (ICALP'15)

- Queries to marked bits are intercepted and responded with a random bit
- several lower bounds on the amortized cost in terms of the size of the heap **and** the number of operations

$\Rightarrow$ less efficient than the original Fibonacci heap

Recent Studies

- Fibonacci Numbers were discovered >800 years ago
- Fibonacci Heaps were developed by Fredman and Tarjan in 1984

Brodal, Lagogiannis, Tarjan: Strict Fibonacci Heap (STOC'12)

Strict Fibonacci Heap:

- pointer-based heap implementation similar to Fibonacci Heaps
- achieves the same cost as Fibonacci Heaps, but actual costs!

Li, Peebles: Replacing Mark Bits with Randomness in Fibonacci Heap (ICALP'15)

- Queries to marked bits are intercepted and responded with a random bit
- several lower bounds on the amortized cost in terms of the size of the heap **and** the number of operations

$\Rightarrow$ less efficient than the original Fibonacci heap

$\Rightarrow$ marked bit is not redundant!

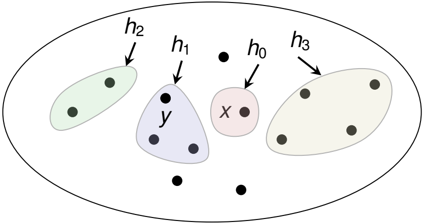
Outlook: A More Efficient Priority Queue for fixed Universe

Operation	Fibonacci heap amortized cost	Van Emde Boas Tree actual cost
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log\log u)$
MINIMUM	$\mathcal{O}(1)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log\log u)$
MERGE/UNION	$\mathcal{O}(1)$	-
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log\log u)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log\log u)$
SUCC	-	$\mathcal{O}(\log\log u)$
PRED	-	$\mathcal{O}(\log\log u)$
MAXIMUM	-	$\mathcal{O}(1)$

Outlook: A More Efficient Priority Queue for fixed Universe

Operation	Fibonacci heap amortized cost	Van Emde Boas Tree actual cost
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log\log u)$
MINIMUM	$\mathcal{O}(1)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(\log n)$	$\mathcal{O}(\log\log u)$
MERGE/UNION	$\mathcal{O}(1)$	-
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log\log u)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log\log u)$
SUCC	-	$\mathcal{O}(\log\log u)$
PRED	-	$\mathcal{O}(\log\log u)$
MAXIMUM	-	$\mathcal{O}(1)$

all this requires key values to be in a universe of size u!



5.3: Disjoint Sets

Frank Stajano

Thomas Sauerwald

Lent 2016

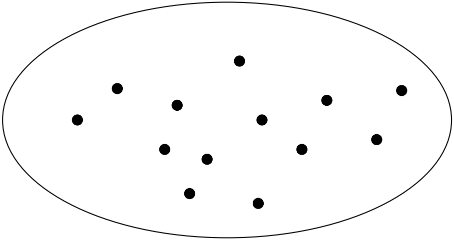
UNIVERSITY OF CAMBRIDGE

Outline

Disjoint Sets

Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

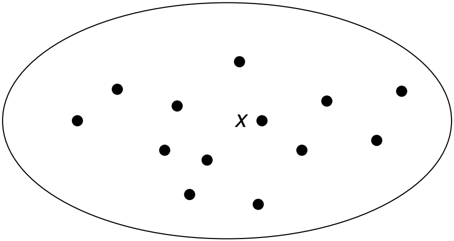


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

h_0=**`MakeSet(x)`**

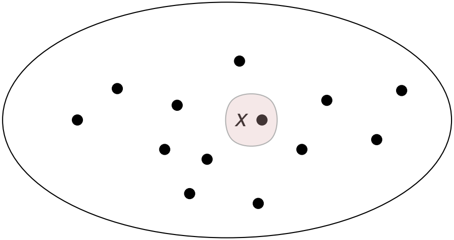


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

h_0=**`MakeSet(x)`**

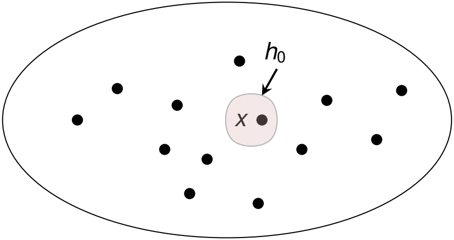


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

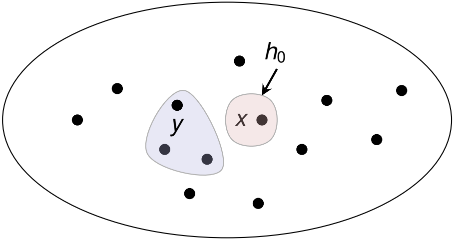
h_0=**`MakeSet(x)`**



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

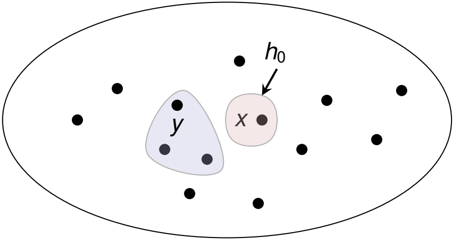


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

h_1=**`FindSet(y)`**

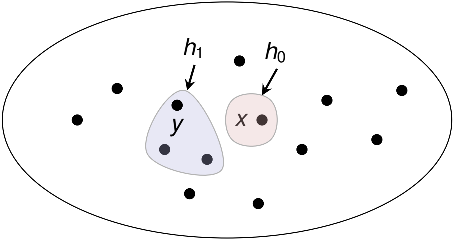


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

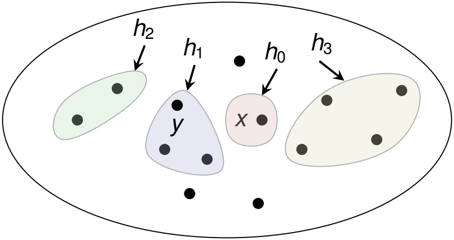
h_1=**`FindSet(y)`**



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

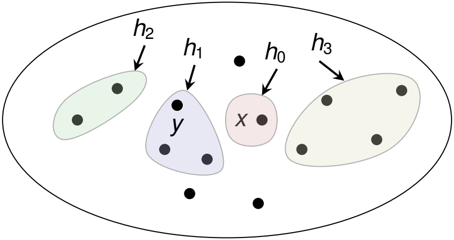


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union`(h_0, h_3)

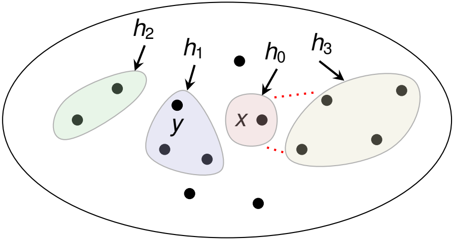


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union(`h_0`,`h_3`)`

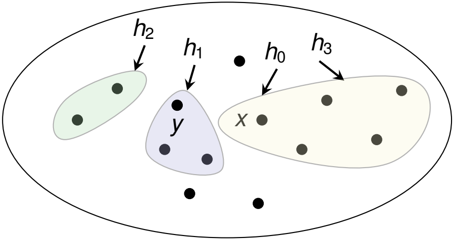


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union(`h_0`,`h_3`)`

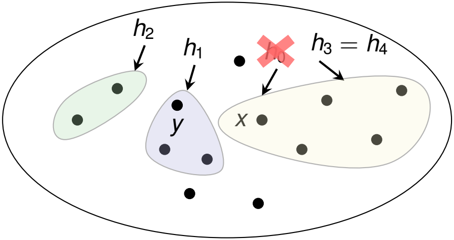


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

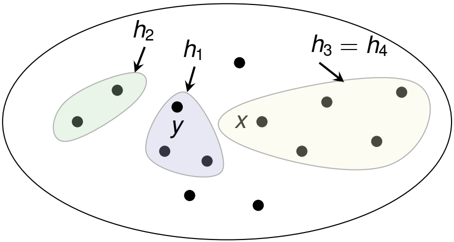
h_4=`Union(`h_0`,`h_3`)`



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

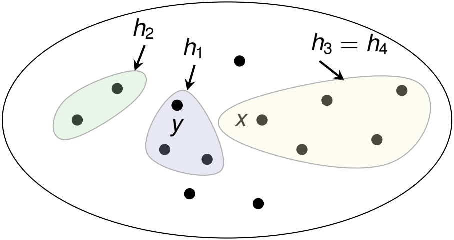


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`

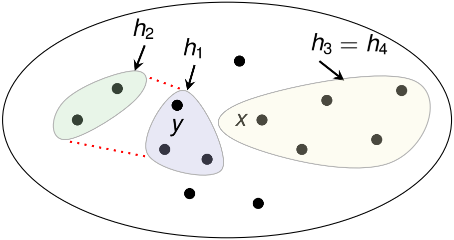


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union`(h_1,h_2)

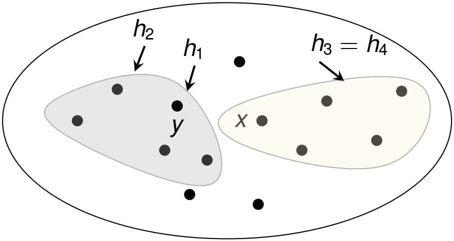


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`

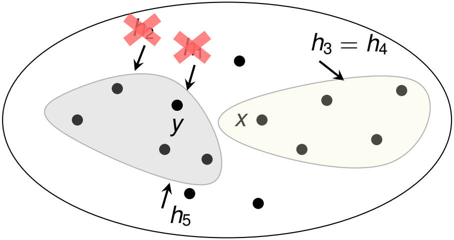


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`

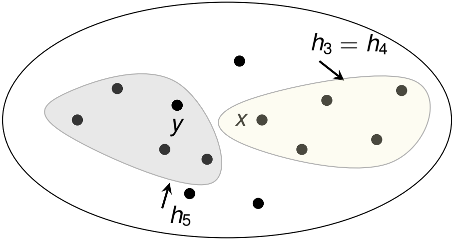


Disjoint Sets (aka Union Find)

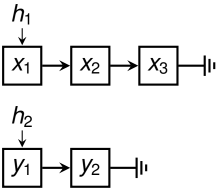
Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`



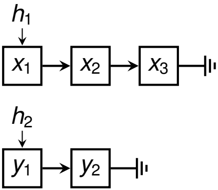
First Attempt: List Implementation



First Attempt: List Implementation

UNION-Operation

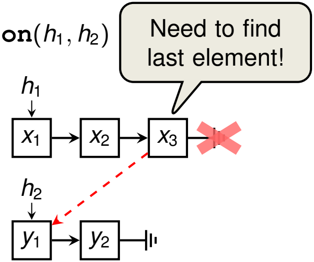
`Union`(h_1, h_2)



First Attempt: List Implementation

UNION-Operation

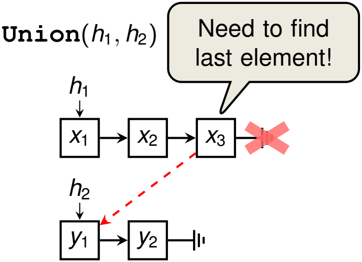
$\texttt{Union}(h_1, h_2)$



First Attempt: List Implementation

UNION-Operation

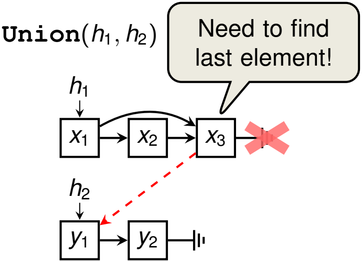
- Add extra pointer to the last element in each list



First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list



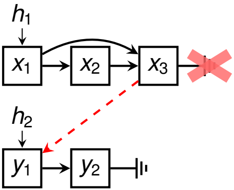
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

`Union`(h_1, h_2)



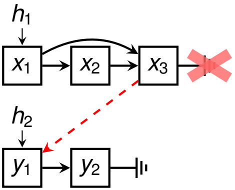
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

Union(h_1, h_2)



FINDSET-Operation

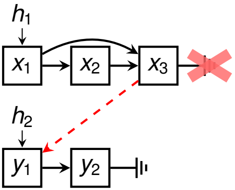
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

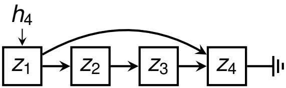
$\Rightarrow$ UNION takes constant time

Union(h_1, h_2)



FINDSET-Operation

FindSet(z_3)



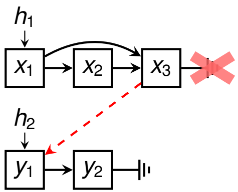
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

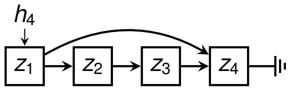
Union(h_1, h_2)



FINDSET-Operation

- Add backward pointer to the list head from everywhere

FindSet(z_3)



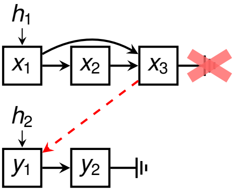
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

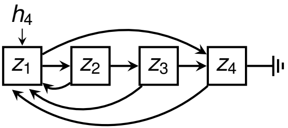
Union(h_1, h_2)



FINDSET-Operation

- Add backward pointer to the list head from everywhere

FindSet(z_3)



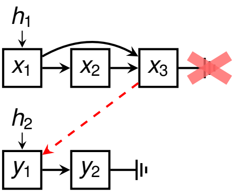
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

`Union`(h_1, h_2)

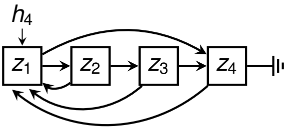


FINDSET-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FINDSET takes constant time

`FindSet`(z_3)



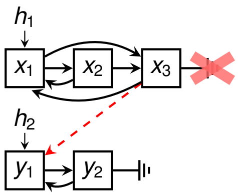
First Attempt: List Implementation

Union-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ Union takes constant time

Union(h_1, h_2)

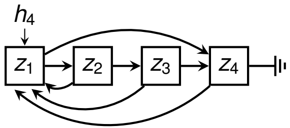


FindSet-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FindSet takes constant time

FindSet(z_3)



First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

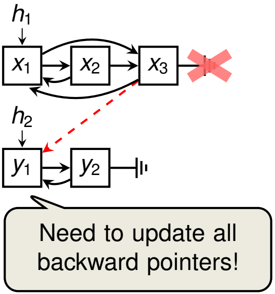
$\Rightarrow$ UNION takes constant time

FINDSET-Operation

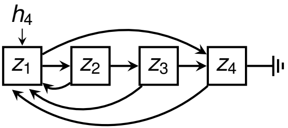
- Add backward pointer to the list head from everywhere

$\Rightarrow$ FINDSET takes constant time

Union(h_1, h_2)



FindSet(z_3)



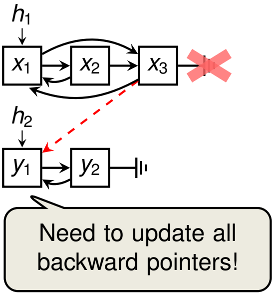
First Attempt: List Implementation

Union-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ ~~Union takes constant time~~

$\texttt{Union}(h_1, h_2)$

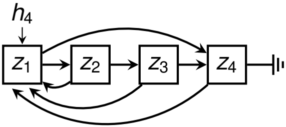


FindSet-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FindSet takes constant time

$\texttt{FindSet}(z_3)$

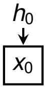


First Attempt: List Implementation (Analysis)

$d = \texttt{\textbf{DisjointSet}}()$

First Attempt: List Implementation (Analysis)

$d = \texttt{\textbf{DisjointSet}}()$
$h_0 = d.\texttt{\textbf{MakeSet}}(x_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$

First Attempt: List Implementation (Analysis)

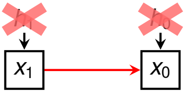
$d = \texttt{\textbf{DisjointSet}}()$
$h_0 = d.\texttt{\textbf{MakeSet}}(x_0)$

$h_1 = d.\texttt{\textbf{MakeSet}}(x_1)$
$h_0 = d.\texttt{\textbf{Union}}(h_1, h_0)$

First Attempt: List Implementation (Analysis)

$d = \mathtt{DisjointSet}()$
$h_0 = d.\mathtt{MakeSet}(x_0)$

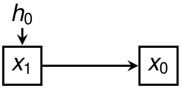
$h_1 = d.\mathtt{MakeSet}(x_1)$
$h_0 = d.\mathtt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

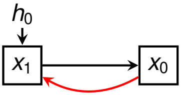
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

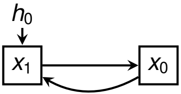
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

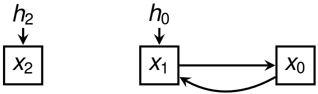
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

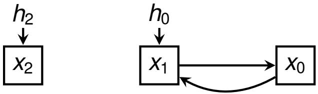
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

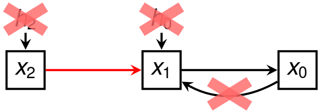
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

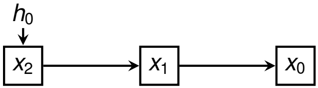
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

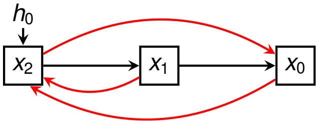
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

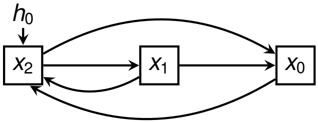
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

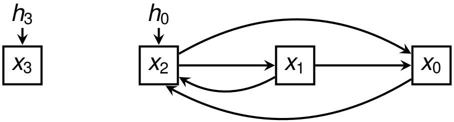
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

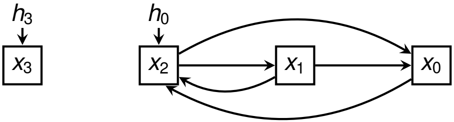
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

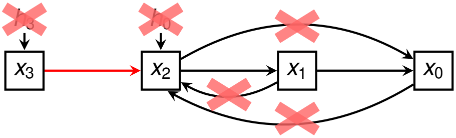
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

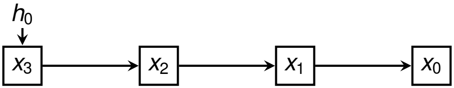
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

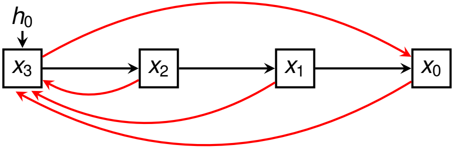
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

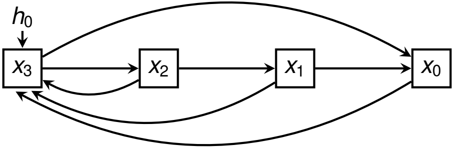
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

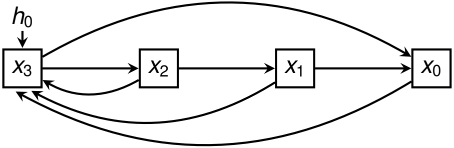
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$

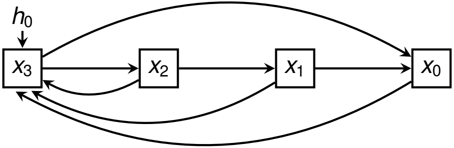


Cost for n UNION operations: $\sum_{i=1}^{n} i = \Theta(n^2)$

First Attempt: List Implementation (Analysis)

$d = \mathtt{DisjointSet}()$
$h_0 = d.\mathtt{MakeSet}(x_0)$

$h_1 = d.\mathtt{MakeSet}(x_1)$
$h_0 = d.\mathtt{Union}(h_1, h_0)$
$h_2 = d.\mathtt{MakeSet}(x_2)$
$h_0 = d.\mathtt{Union}(h_2, h_0)$
$h_3 = d.\mathtt{MakeSet}(x_3)$
$h_0 = d.\mathtt{Union}(h_3, h_0)$



better to append shorter list to longer $\rightsquigarrow$ Weighted-Union Heuristic

Cost for n UNION operations: $\sum_{i=1}^{n} i = \Theta(n^2)$

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Amortized Analysis: Every operation has amortized cost $\mathcal{O}(\log n)$, but there may be operations with total cost $\Theta(n)$.

Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations

Analysis of Weighted-Union Heuristic

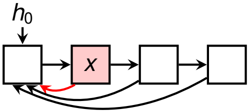
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic

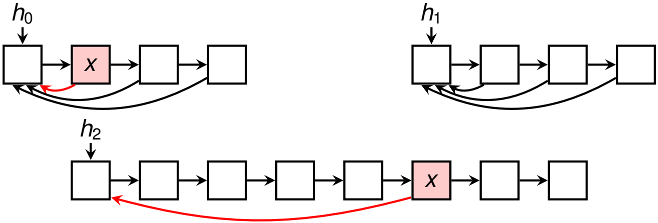
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic



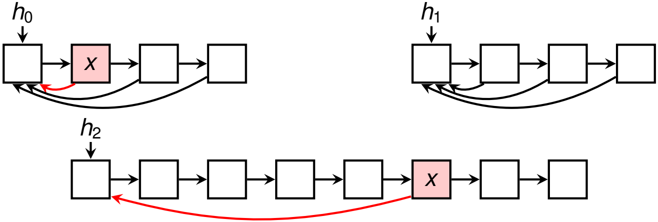
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n-1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic



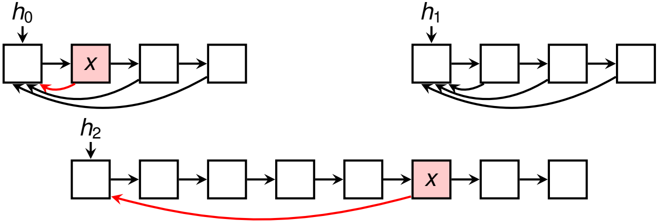
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

Analysis of Weighted-Union Heuristic



Theorem 21.1

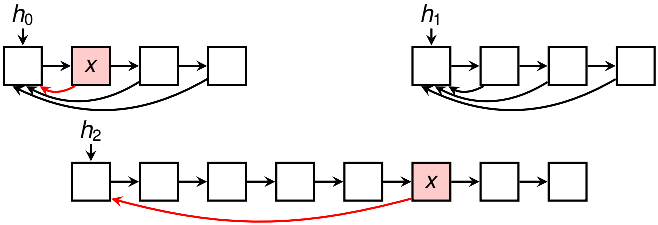
Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

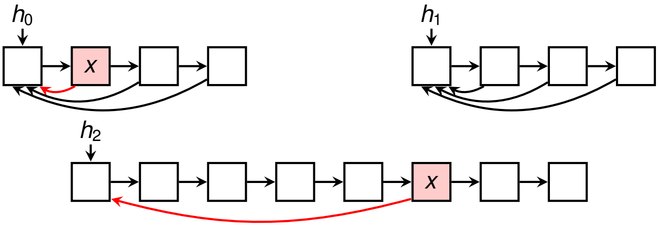
Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

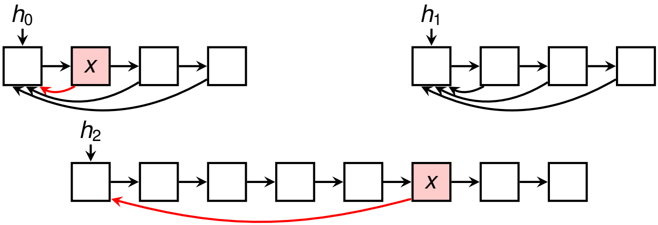
Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Can we improve on this further?

Proof:

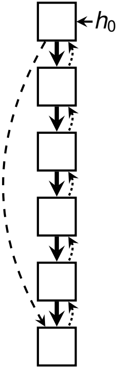
- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

$\square$

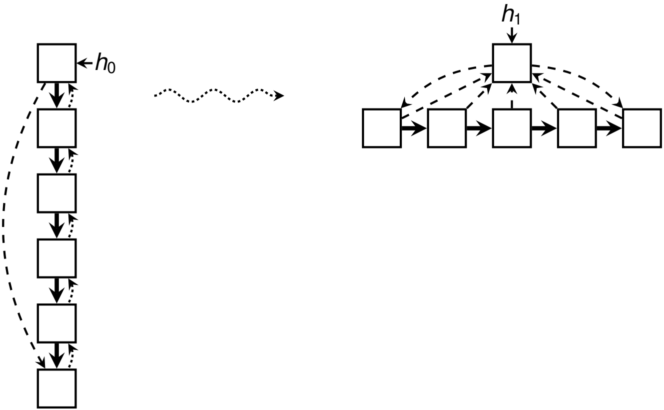
How to Improve?



Doubly-Linked List

- MakeSet: $\mathcal{O}(1)$
- FindSet: $\mathcal{O}(n)$
- Union: $\mathcal{O}(1)$

How to Improve?



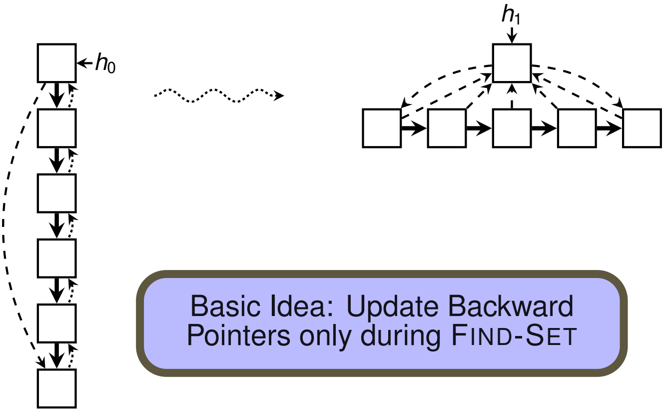
Doubly-Linked List

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

How to Improve?



Basic Idea: Update Backward Pointers only during FIND-SET

Doubly-Linked List

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

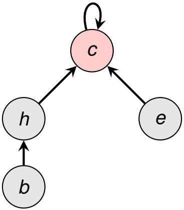
Disjoint Sets via Forests

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)

Disjoint Sets via Forests

$\{b, c, e, h\}$

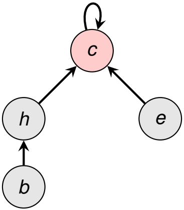


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)

Disjoint Sets via Forests

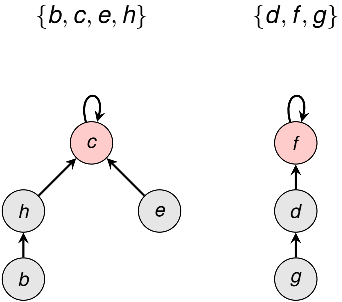
$\{b, c, e, h\}$



Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

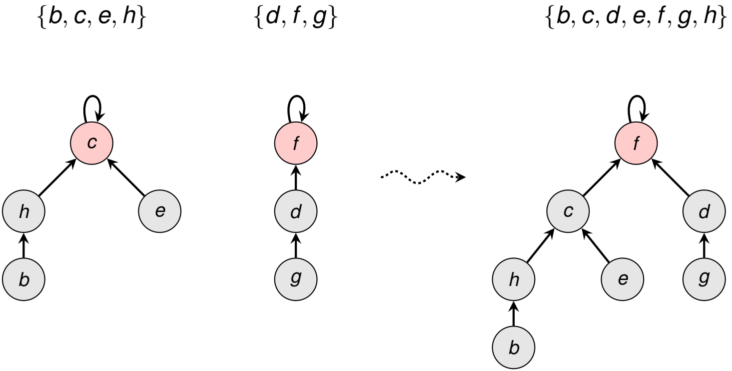
Disjoint Sets via Forests



Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

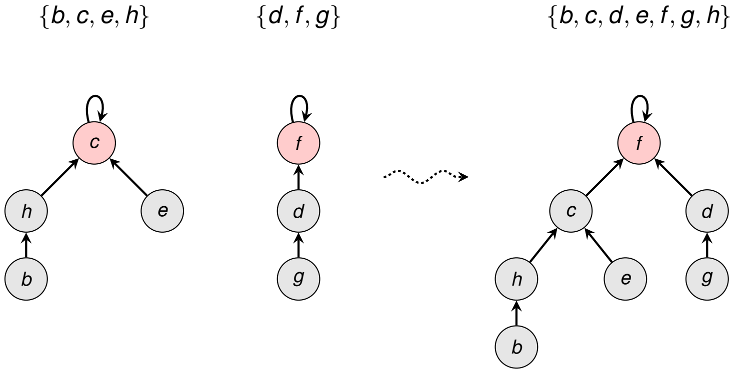
Disjoint Sets via Forests



Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Disjoint Sets via Forests

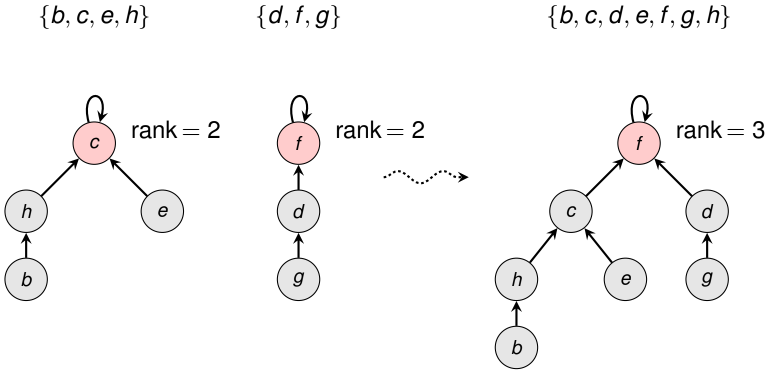


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Disjoint Sets via Forests

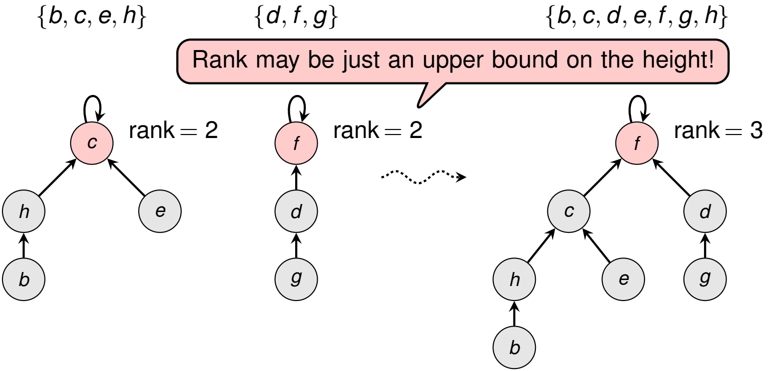


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Disjoint Sets via Forests

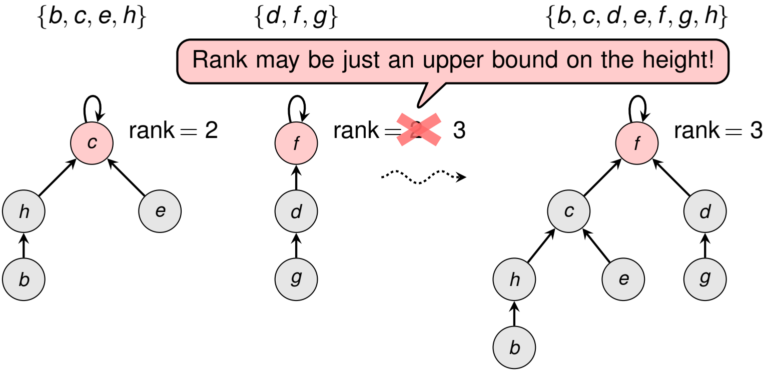


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Disjoint Sets via Forests



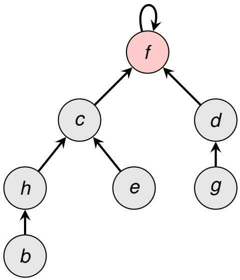
Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Path Compression during FINDSET

FindSet(b):

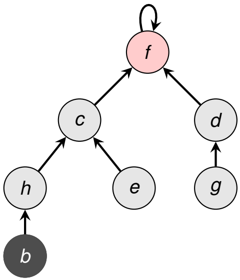



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

$\boxed{b}$

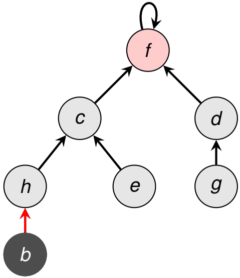



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet (b):

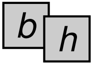
$\boxed{b}$

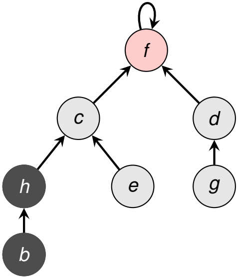



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

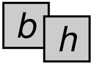


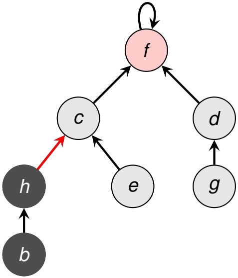



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FindSet

FindSet(b):

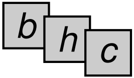


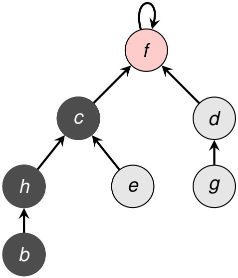



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

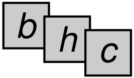
FindSet(b):

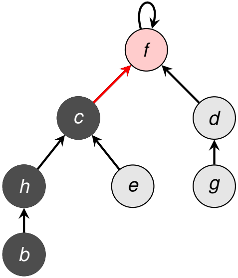





```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

FindSet(b):

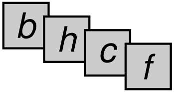


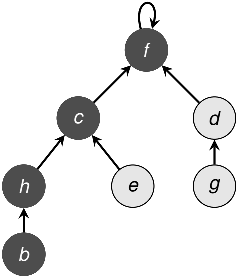



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

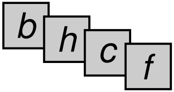


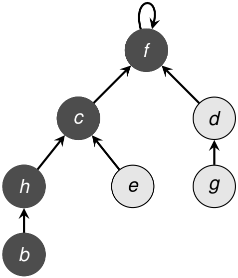



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FindSet

$\texttt{FindSet}(b)$:

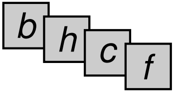


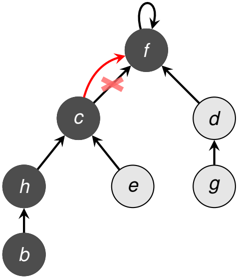



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

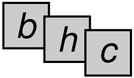


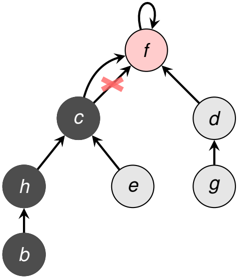



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

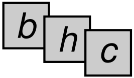


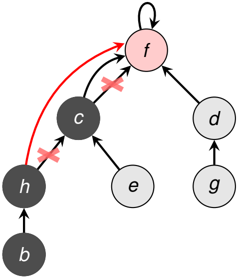



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

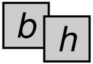


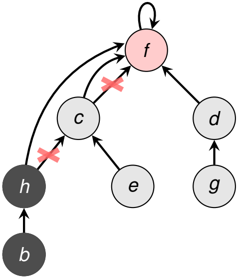



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FindSet

FindSet(b):

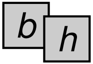


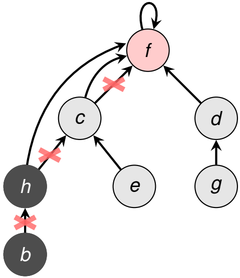



```
0: FindSet(x)
1:    if x ≠ x.p
2:       x.p = FindSet(x.p)
3:    return x.p
```

Path Compression during FINDSET

FindSet(b):



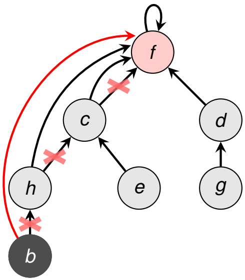



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet (b):

$\boxed{b}$

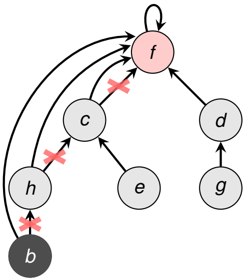



```
0: FindSet(x)
1:    if x ≠ x.p
2:       x.p = FindSet(x.p)
3:    return x.p
```

Path Compression during FINDSET

FindSet (b):

$\boxed{b}$

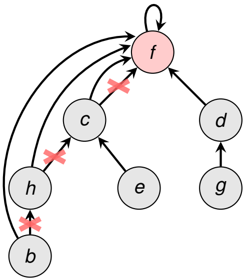



```
0: FindSet(x)
1:    if x ≠ x.p
2:       x.p = FindSet(x.p)
3:    return x.p
```

Path Compression during FINDSET

FindSet(*b***):**

$\boxed{b}$

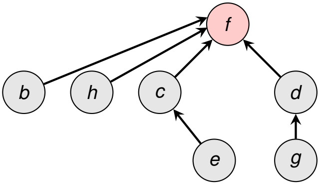



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

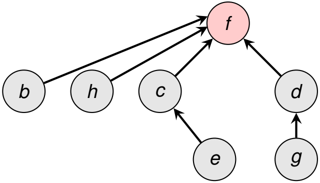
$\boxed{b}$




```
0: FindSet(x)
1:    if x ≠ x.p
2:       x.p = FindSet(x.p)
3:    return x.p
```

Path Compression during FINDSET

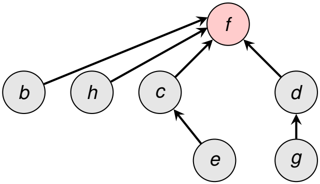
FindSet(b):




```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):



Maintaining the exact height would be costly, hence rank is only an **upper bound**!

```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

More than the number of atoms in the universe!

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

$\log^*(n)$, **the iterated logarithm**, satifies $\alpha(n) \leq \log^*(n)$, but still $\log^*(10^{80}) = 5$.

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

In practice, $\alpha(n)$ is a small constant

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

Combining Union by Rank and Path Compression

Data Structure is essentially optimal! (for more details see CLRS)

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

In practice, $\alpha(n)$ is a small constant

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

Simulating the Effects of Union by Rank and Path Compression

Simulating the Effects of Union by Rank and Path Compression

Experimental Setup

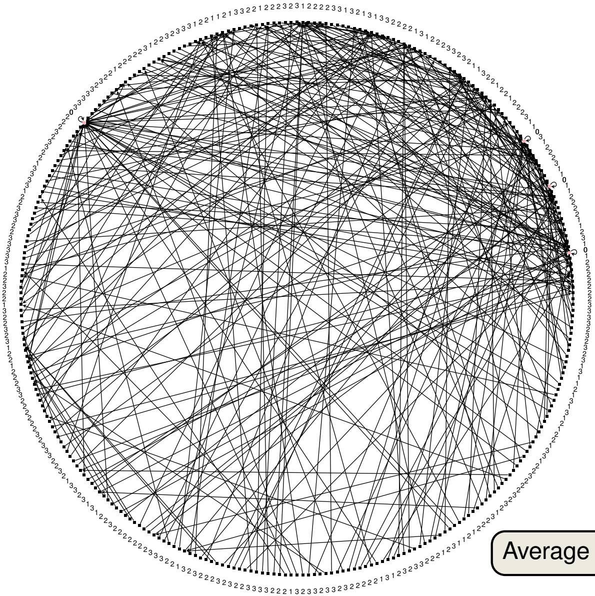
1. Initialise singletons $1, 2, \ldots, 300$
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform $\textsc{Union}(\textsc{FindSet}(i), \textsc{FindSet}(r))$

Simulating the Effects of Union by Rank and Path Compression

Experimental Setup

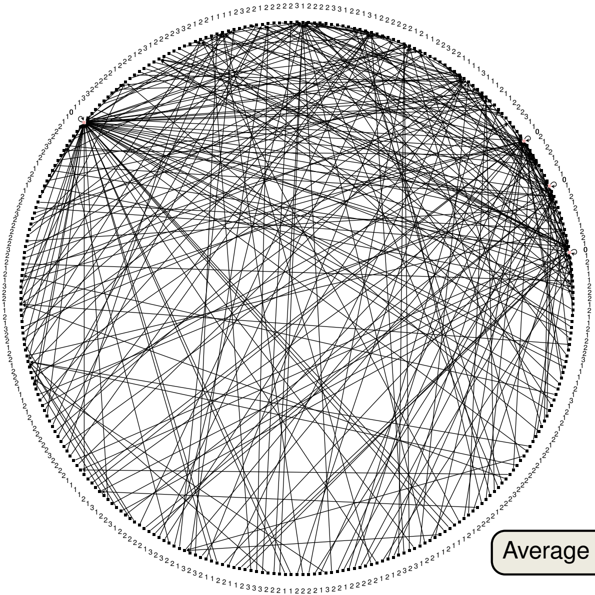
1. Initialise singletons $1, 2, \ldots, 300$
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform $\textsc{Union}(\textsc{FindSet}(i), \textsc{FindSet}(r))$
3. Perform $j \in \{0, 100, 200, 300, 600, 900, 1200, 1500, 1800\}$ many additional $\textsc{FindSet}(r)$, where $1 \leq r \leq 300$ is random

Union by Rank without Path Compression



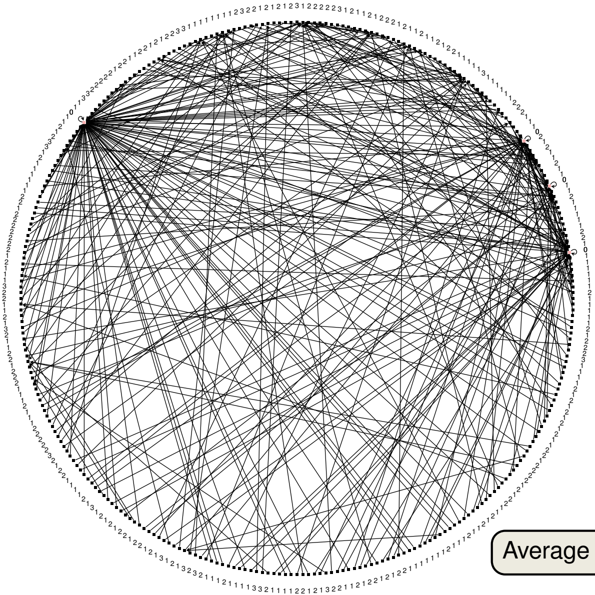
Average Height: 2.12

Union by Rank with Path Compression



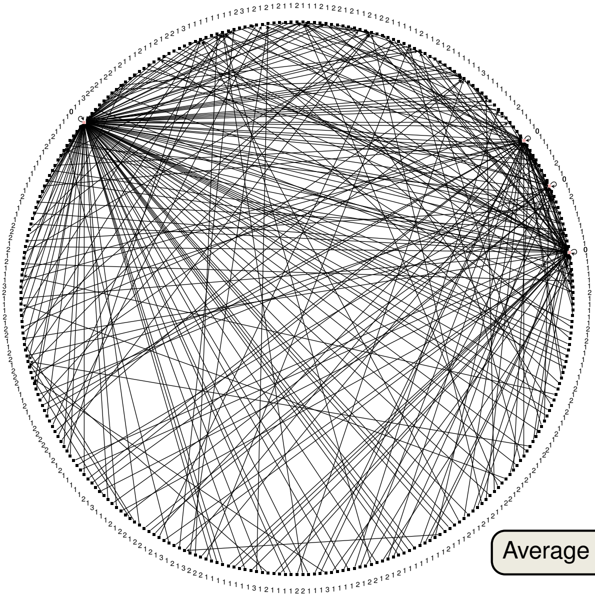
Average Height: 1.75

Union by Rank with Path Compression (100 additional FINDSET)



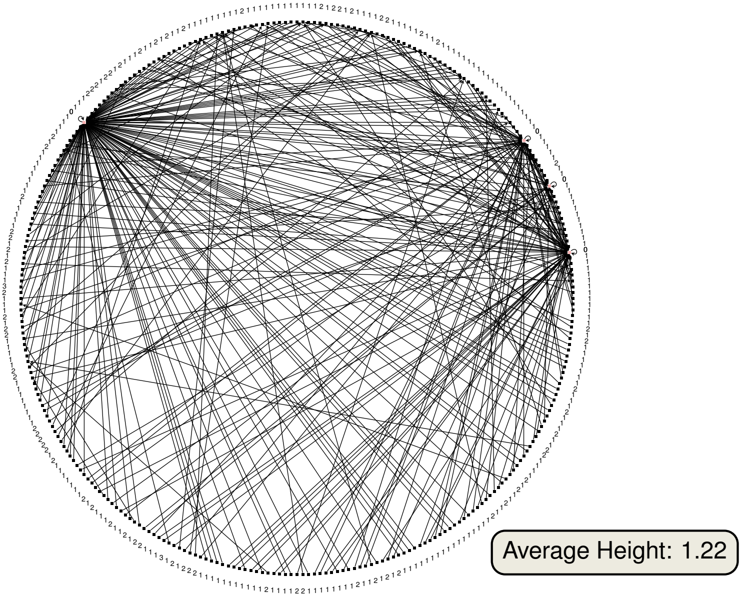
Average Height: 1.53

Union by Rank with Path Compression (200 additional FINDSET)

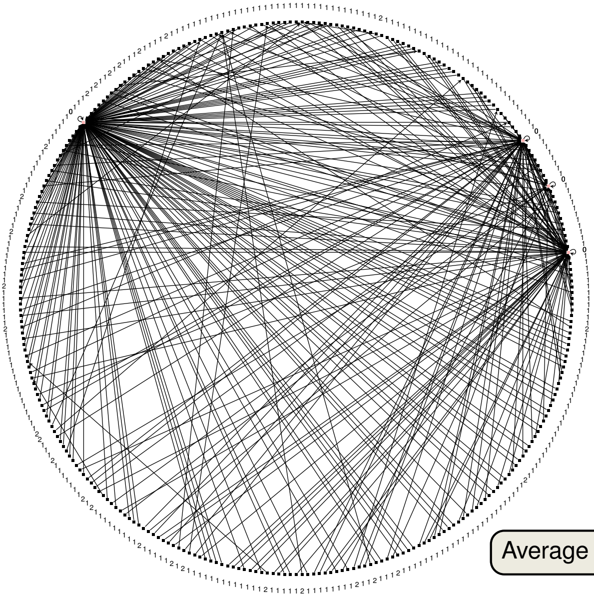


Average Height: 1.35

Union by Rank with Path Compression (300 additional FINDSET)

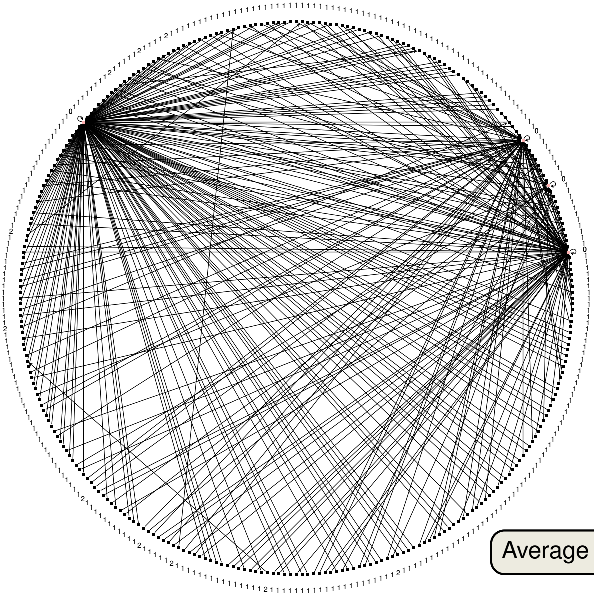


Union by Rank with Path Compression (600 additional FINDSET)



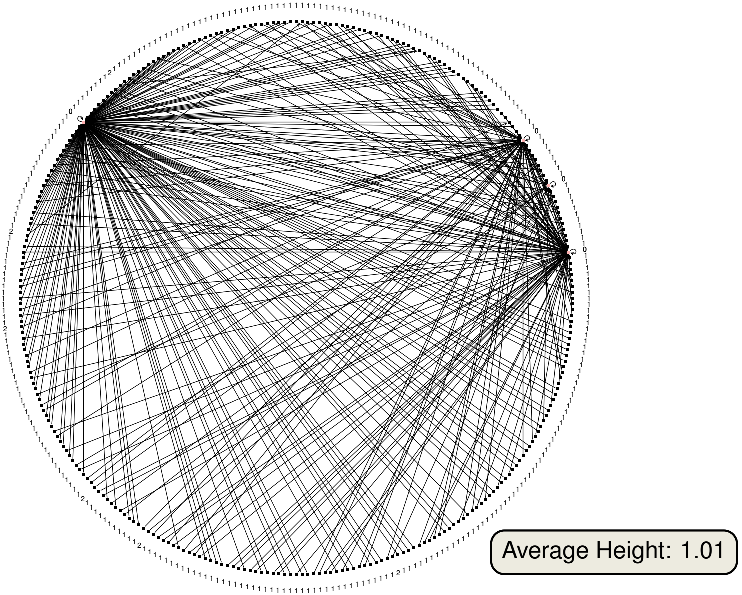
Average Height: 1.08

Union by Rank with Path Compression (900 additional FindSet)

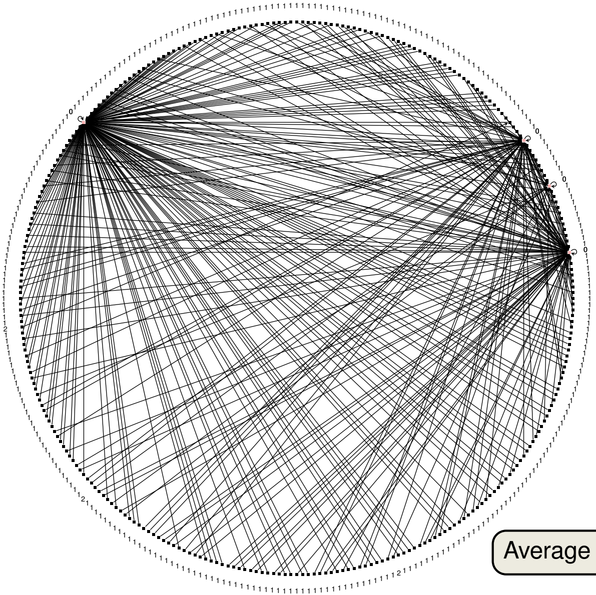


Average Height: 1.02

Union by Rank with Path Compression (1200 additional FINDSET)

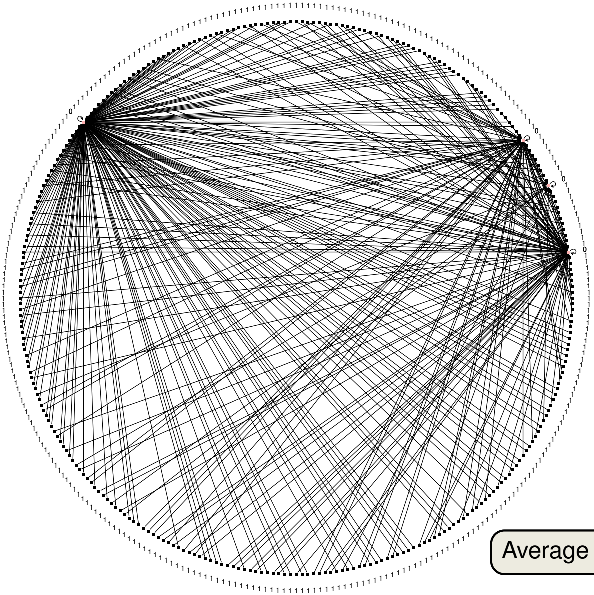


Union by Rank with Path Compression (1500 additional FINDSET)



Average Height: 1.00

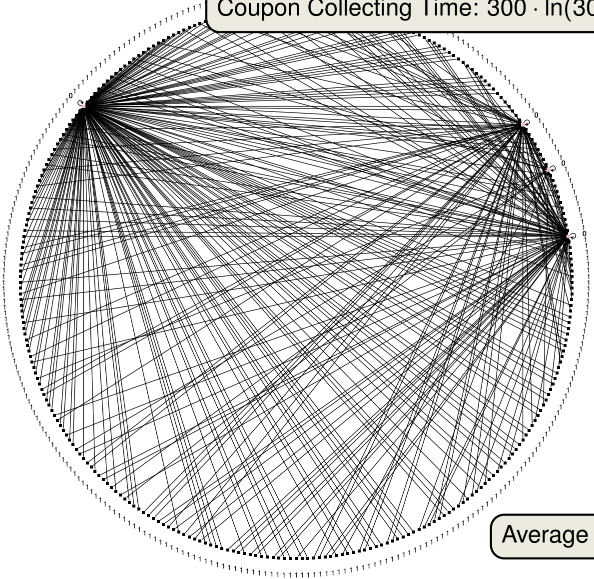
Union by Rank with Path Compression (1800 additional FindSet)



Average Height: 0.98

Union by Rank with Path Compression (1800 additional FINDSET)

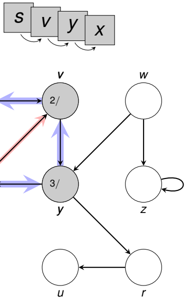
Coupon Collecting Time: $300 \cdot \ln(300) \approx 1711$



Average Height: 0.98

Overview

	Union by Rank	Union by Rank & Path Compression
300 MakeSet & 300 Union	2.12	1.75
100 extra FindSet	2.12	1.53
200 extra FindSet	2.12	1.35
300 extra FindSet	2.12	1.22
600 extra FindSet	2.12	1.08
900 extra FindSet	2.12	1.02
1200 extra FindSet	2.12	1.01
1500 extra FindSet	2.12	1.00
1800 extra FindSet	2.12	0.98



6.1 & 6.2: Graph Searching

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

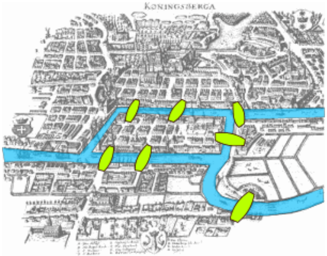
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

Topological Sort

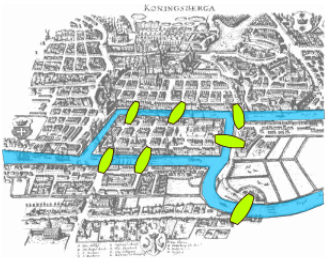
Origin of Graph Theory



Source: Wikipedia

Seven Bridges at Königsberg 1737

Origin of Graph Theory



Source: Wikipedia

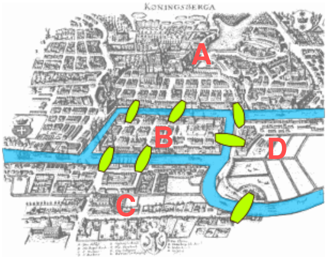
Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory



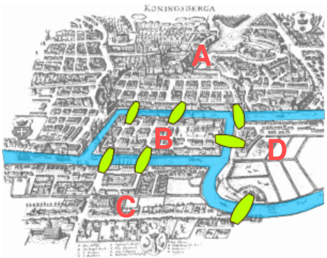
Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?



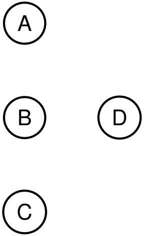
Source: Wikipedia

Seven Bridges at Königsberg 1737

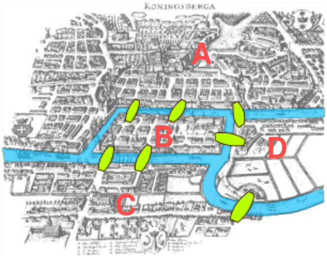
Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?



Origin of Graph Theory

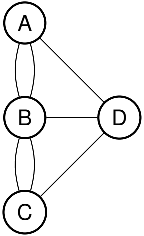


Source: Wikipedia

Seven Bridges at Königsberg 1737

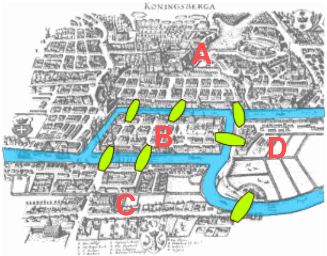
Source: Wikipedia

Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory

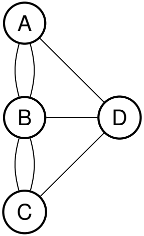


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

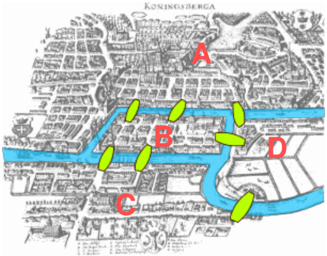
Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Is there a tour which visits every island **exactly once**?

Origin of Graph Theory

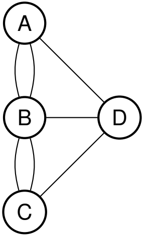


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Is there a tour which visits every island **exactly once**?
⇝ 1B course: Complexity Theory

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

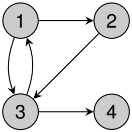
- V: the set of vertices
- E: the set of edges (arcs)

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

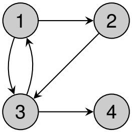


What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)



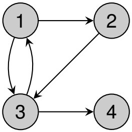
$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

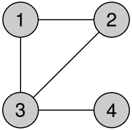


Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

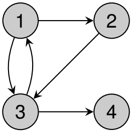


What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

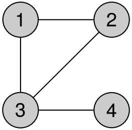


$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



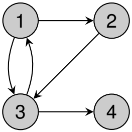
$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

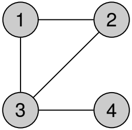


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

Undirected Graph

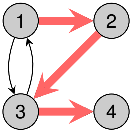
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

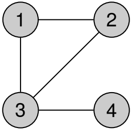
Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 4)$



$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$



$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

Undirected Graph

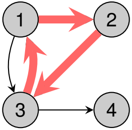
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

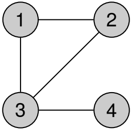
Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 1)$, which is a cycle



$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$



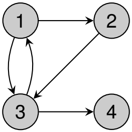
$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

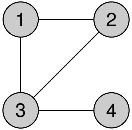


$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

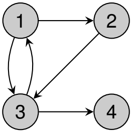
What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

G is not connected



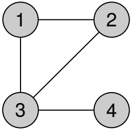
$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

G is connected



Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

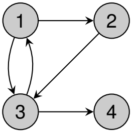
What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

G is not connected



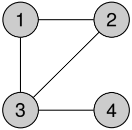
$$V = \{1, 2, 3, 4\}$$
$$E = \{(1,2), (1,3), (2,3), (3,1), (3,4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

G is connected



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1,2\}, \{1,3\}, \{2,3\}, \{3,4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

Later: edge-weighted graphs $G = (V, E, w)$

Representations of Directed and Undirected Graphs

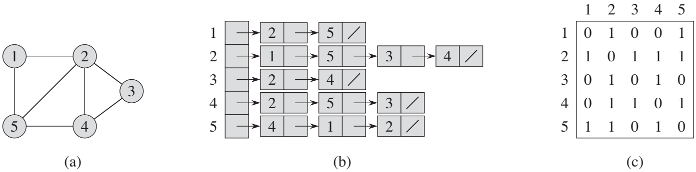
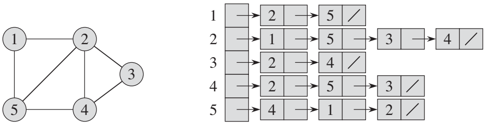


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Representations of Directed and Undirected Graphs



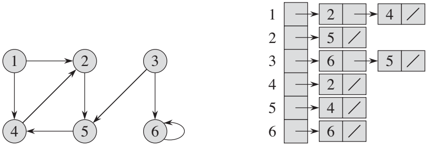
(a) (b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Most times we will use the adjacency-list representation!



(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Overview

Amortized Analysis

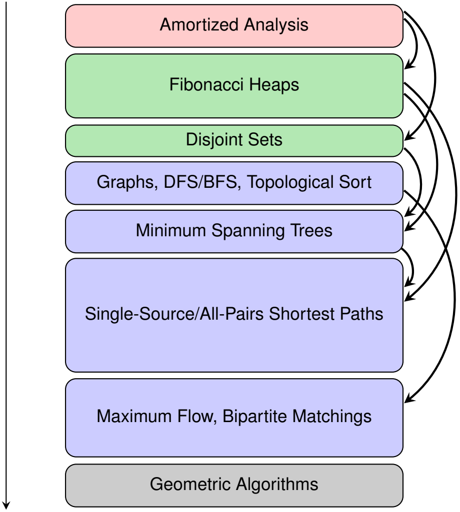
Fibonacci Heaps

Disjoint Sets

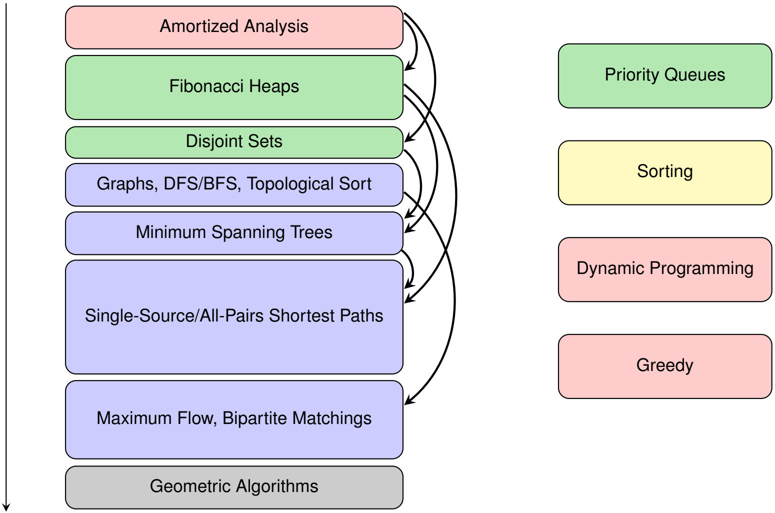
Overview

- Amortized Analysis
- Fibonacci Heaps
- Disjoint Sets
- Graphs, DFS/BFS, Topological Sort
- Minimum Spanning Trees
- Single-Source/All-Pairs Shortest Paths
- Maximum Flow, Bipartite Matchings
- Geometric Algorithms

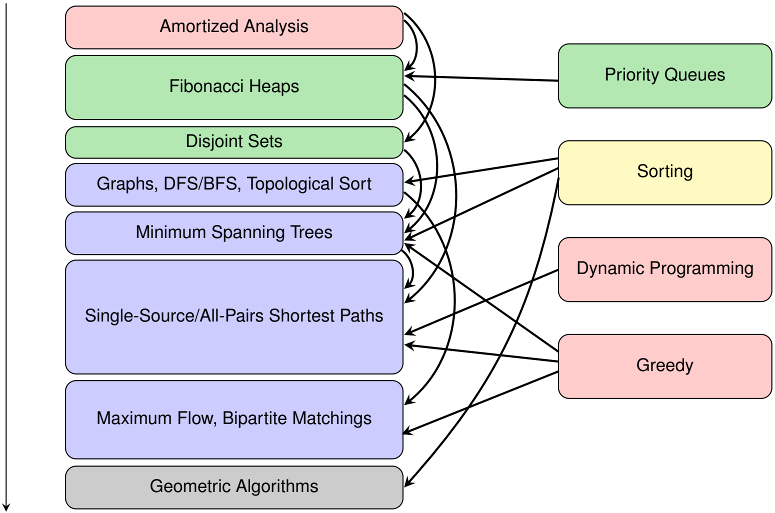
Overview


Amortized Analysis
Fibonacci Heaps
Disjoint Sets
Graphs, DFS/BFS, Topological Sort
Minimum Spanning Trees
Single-Source/All-Pairs Shortest Paths
Maximum Flow, Bipartite Matchings
Geometric Algorithms

Overview

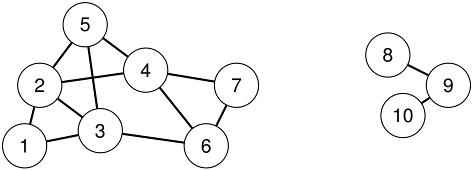

Amortized Analysis
Fibonacci Heaps
Disjoint Sets
Graphs, DFS/BFS, Topological Sort
Minimum Spanning Trees
Single-Source/All-Pairs Shortest Paths
Maximum Flow, Bipartite Matchings
Geometric Algorithms
Priority Queues
Sorting
Dynamic Programming
Greedy

Overview



Amortized Analysis
Fibonacci Heaps
Disjoint Sets
Graphs, DFS/BFS, Topological Sort
Minimum Spanning Trees
Single-Source/All-Pairs Shortest Paths
Maximum Flow, Bipartite Matchings
Geometric Algorithms
Priority Queues
Sorting
Dynamic Programming
Greedy

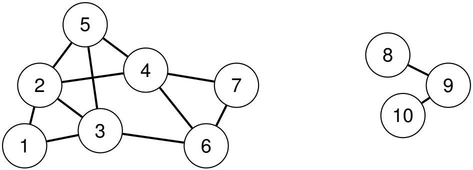
Graph Searching



Overview

- Graph searching means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.

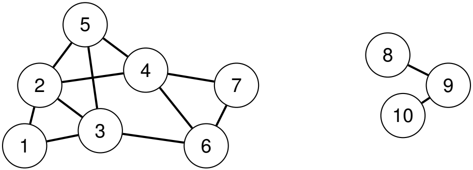
Graph Searching



Overview

- Graph searching means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.
- Two strategies: Breadth-First-Search and Depth-First-Search

Graph Searching



Overview

- Graph searching means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.
- Two strategies: Breadth-First-Search and Depth-First-Search

Measure time complexity in terms of the size of V and E (often write just V instead of $|V|$, and E instead of $|E|$)

Outline

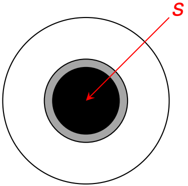
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

Topological Sort

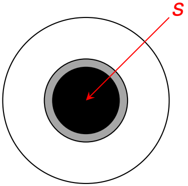
Breadth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s

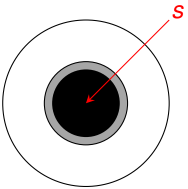
Breadth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- BFS sends out a wave from $s \rightsquigarrow$ compute distances/shortest paths

Breadth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- BFS sends out a wave from $s \rightsquigarrow$ compute distances/shortest paths
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors (=adjacent vertices)

  Black = Visited and all neighbors

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
     v.predecessor = None
     v.d = Infinity # .d = distance from s
     v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
   u = Q.extract()
   assert (u.colour == "grey")
   for v in u.adjacent()
      if v.colour = "white"
         v.colour = "grey"
         v.d = u.d+1
         v.predecessor = u
         Q.insert(v)
   u.colour = "black"
```

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

   assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
      v.predecessor = None
      v.d = Infinity # .d = distance from s
      v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

   assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
      v.predecessor = None
      v.d = Infinity # .d = distance from s
      v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

   assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
      v.predecessor = None
      v.d = Infinity # .d = distance from s
      v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime ???

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
   Run BFS on the given graph G
   starting from source s
  assert(s in G.vertices())
# Initialize graph and queue
for v in G.vertices():
    v.predecessor = None
    v.d = Infinity # .d = distance from s
    v.colour = "white"
Q = Queue()
# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)
# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime ???

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

   assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
       v.predecessor = None
       v.d = Infinity # .d = distance from s
       v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime ???

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
     v.predecessor = None
     v.d = Infinity # .d = distance from s
     v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - White = Unvisited
  - Grey = Visited, but not all neighbors
  - Black = Visited and all neighbors
- Runtime ???

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

$$\sum_{u \in V} |u.adj| = 2|E|$$

Breadth-First-Search: Pseudocode

```
def bfs(G,s)
    Run BFS on the given graph G
    starting from source s

  assert(s in G.vertices())

# Initialize graph and queue
for v in G.vertices():
    v.predecessor = None
    v.d = Infinity # .d = distance from s
    v.colour = "white"
Q = Queue()

# Visit source vertex
s.d = 0
s.colour = "grey"
Q.insert(s)

# Visit the adjacents of each vertex in Q
while not Q.isEmpty():
    u = Q.extract()
    assert (u.colour == "grey")
    for v in u.adjacent()
        if v.colour = "white"
            v.colour = "grey"
            v.d = u.d+1
            v.predecessor = u
            Q.insert(v)
    u.colour = "black"
```

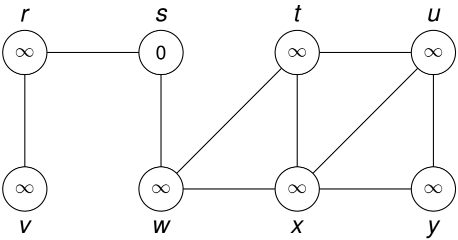
- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - $\boxed{\text{White}}$ = Unvisited
  - Grey = Visited, but not all neighbors
  - Black = Visited and all neighbors
- Runtime $O(V + E)$

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

$$\sum_{u \in V} |u.adj| = 2|E|$$

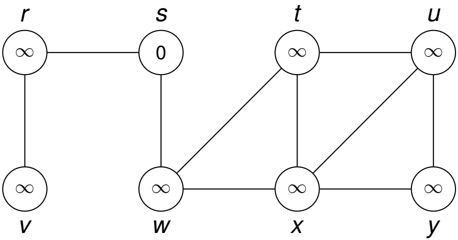
Execution of BFS (Figure 22.3)

Queue:



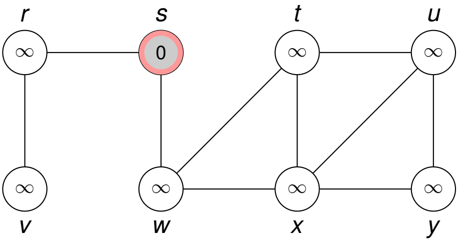
Execution of BFS (Figure 22.3)

Queue: s



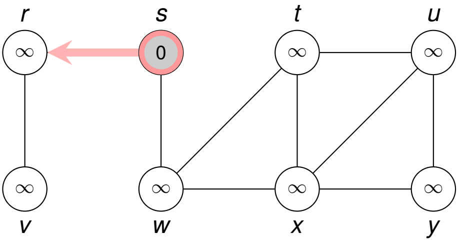
Execution of BFS (Figure 22.3)

Queue: ~~s~~



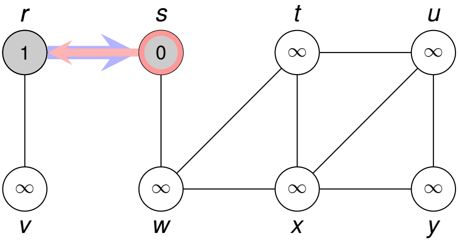
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~



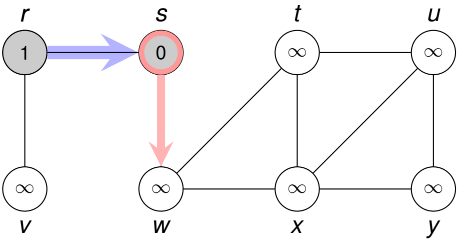
Execution of BFS (Figure 22.3)

Queue: ~~s~~ r



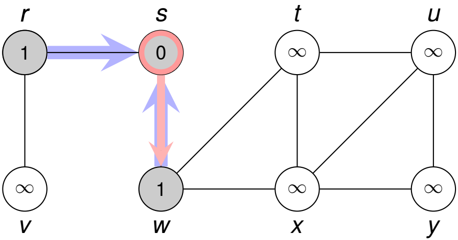
Execution of BFS (Figure 22.3)

Queue: ~~s~~ r



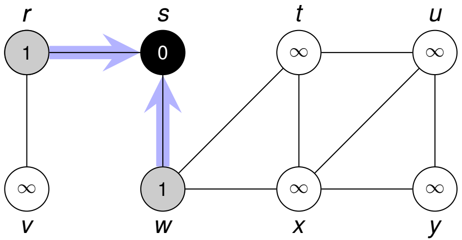
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ r w



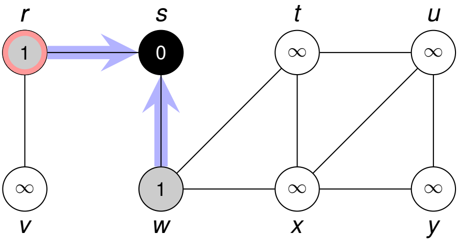
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ r w



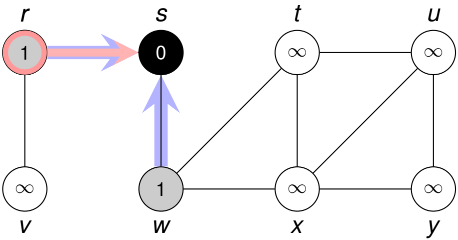
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ w



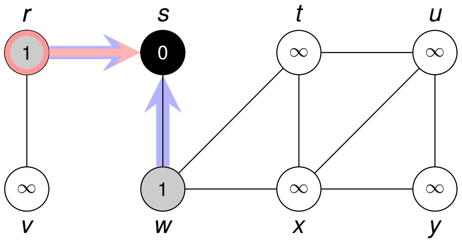
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ w



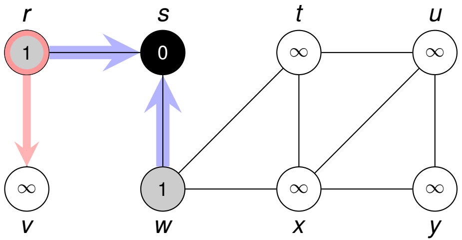
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ w



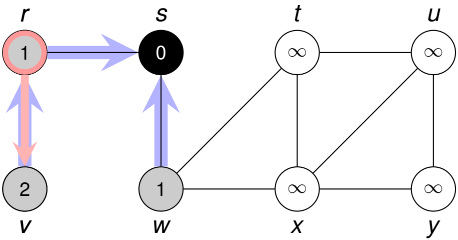
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ w



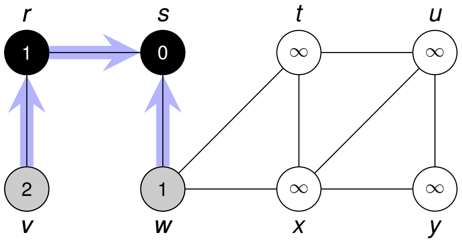
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ w v



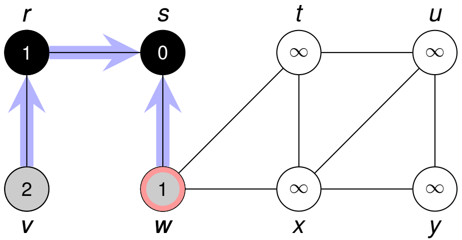
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ w v



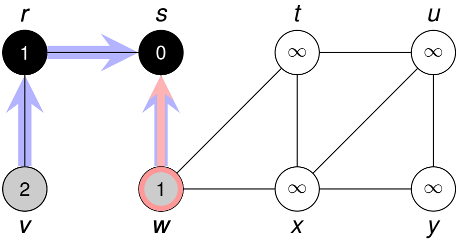
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v



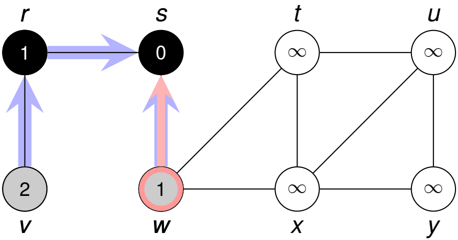
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ v



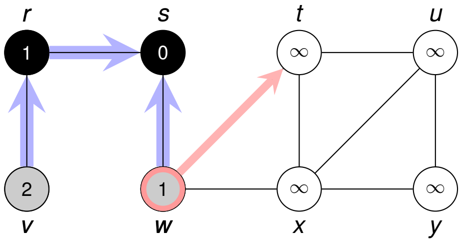
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v



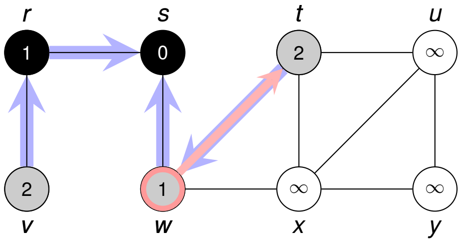
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ *v*



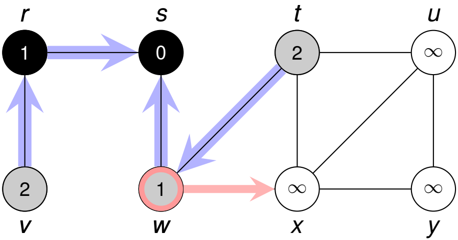
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ v t



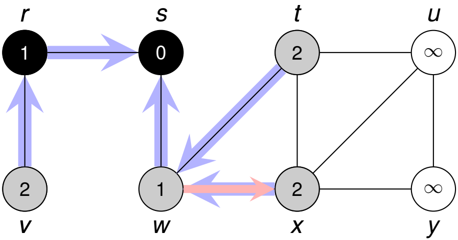
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v t



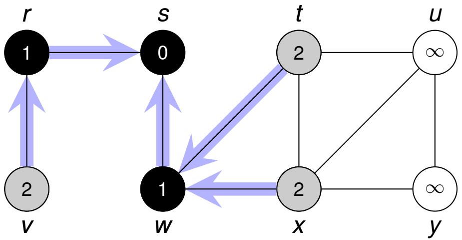
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ v t x



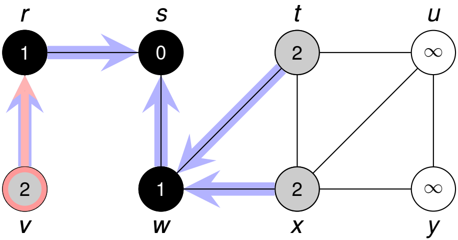
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ v t x



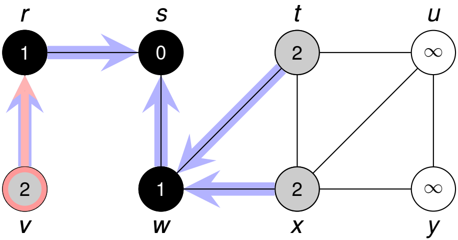
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ t x



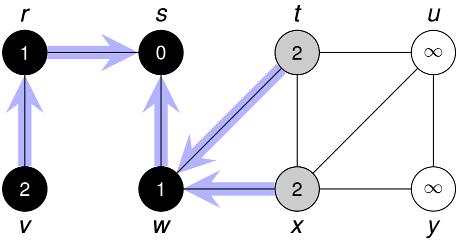
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ t x



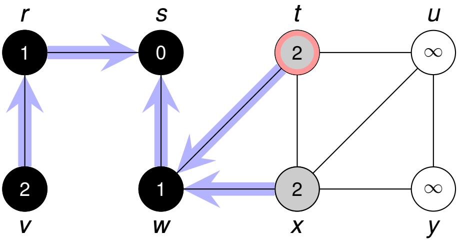
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ t x



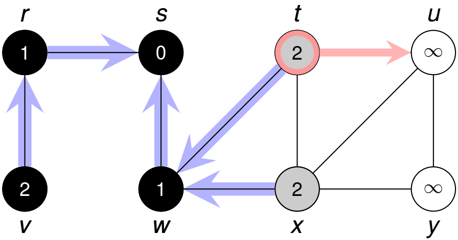
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x



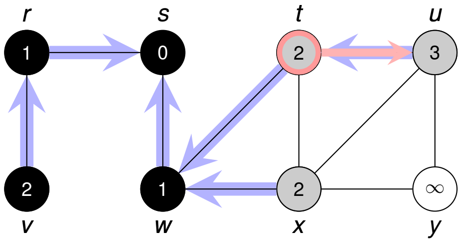
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ x



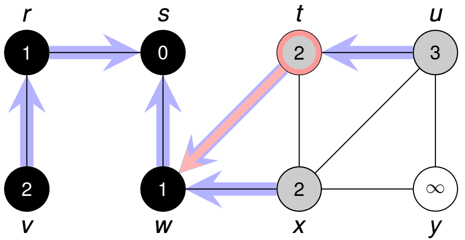
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x u



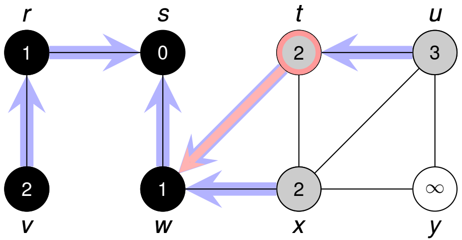
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x u



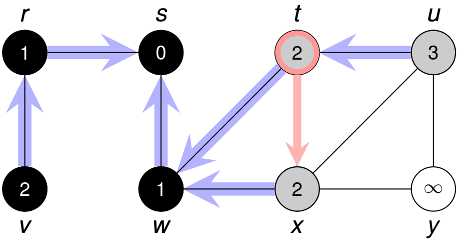
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x u



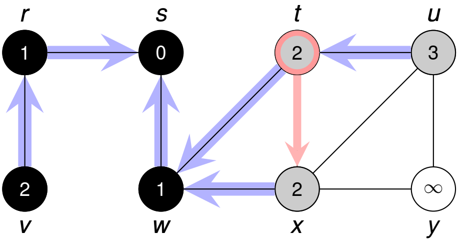
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ x u



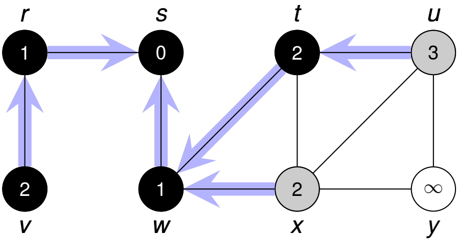
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ x u



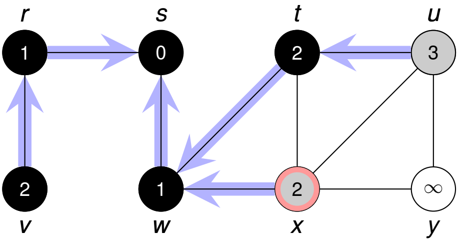
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ x u



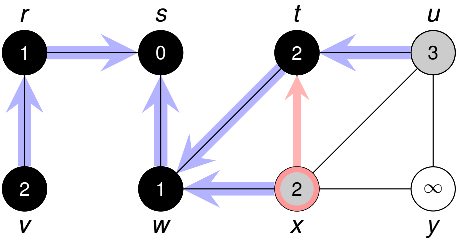
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



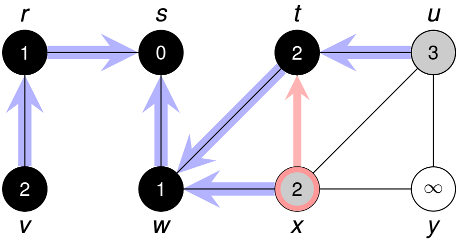
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ *u*



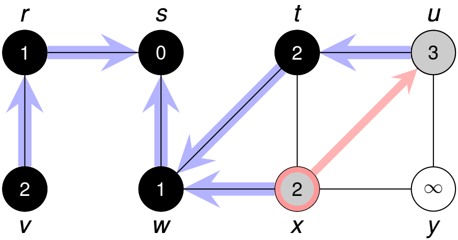
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ *u*



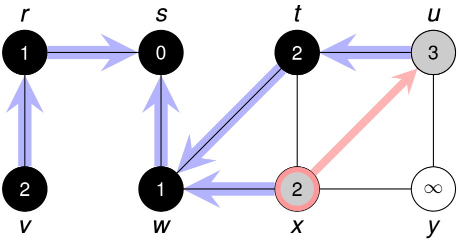
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ *u*



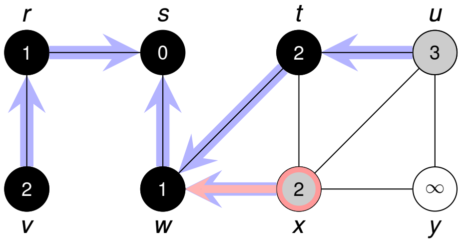
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ $\cancel{x}$ u



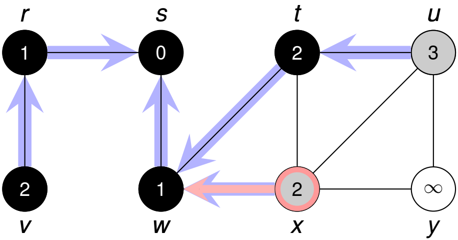
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u



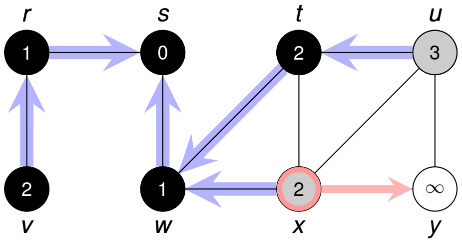
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ $\cancel{x}$ u



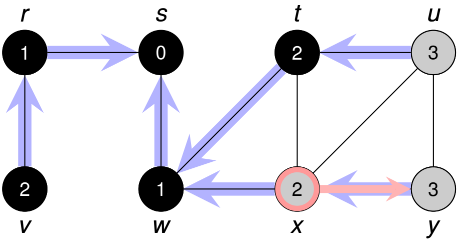
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ $\cancel{x}$ u



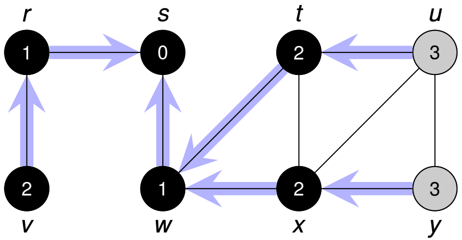
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u y



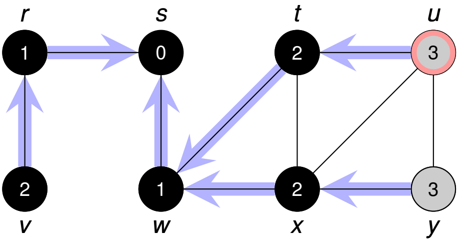
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ u y



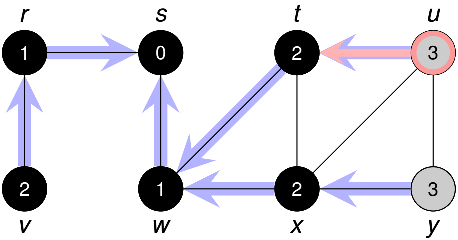
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



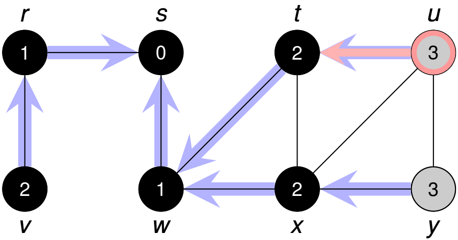
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



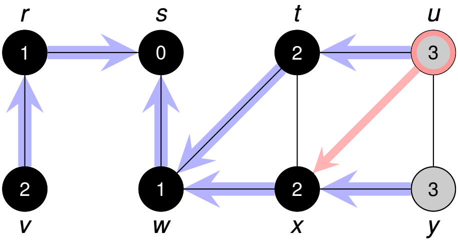
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ ~~*u*~~ *y*



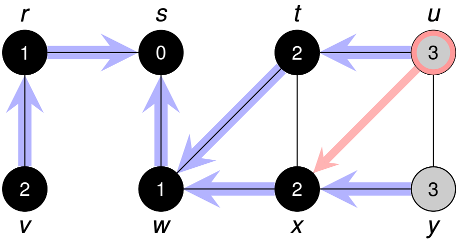
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



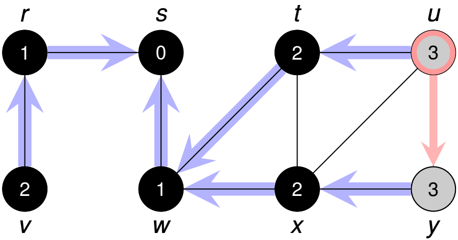
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



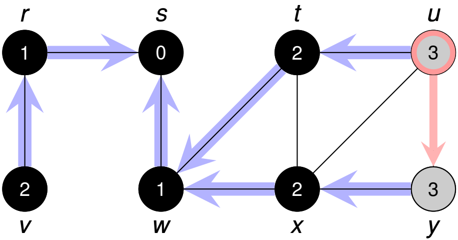
Execution of BFS (Figure 22.3)

Queue: ~~*s*~~ ~~*r*~~ ~~*w*~~ ~~*v*~~ ~~*t*~~ ~~*x*~~ ~~*u*~~ *y*



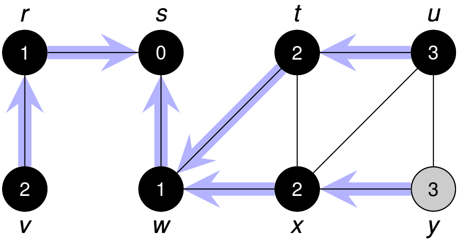
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



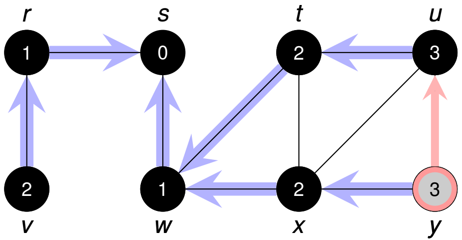
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



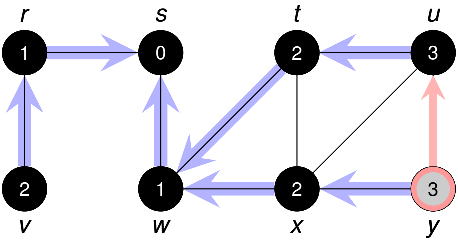
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ y



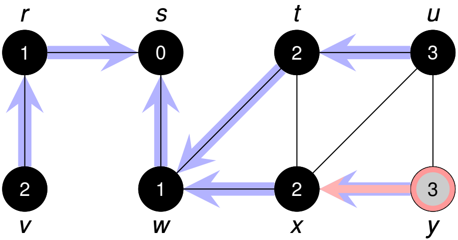
Execution of BFS (Figure 22.3)

Queue: $\cancel{s}$ $\cancel{r}$ $\cancel{w}$ $\cancel{v}$ $\cancel{t}$ $\cancel{x}$ $\cancel{u}$ $\cancel{y}$



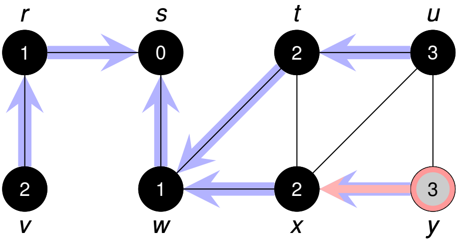
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



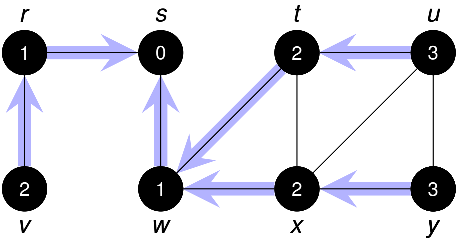
Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



Outline

Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

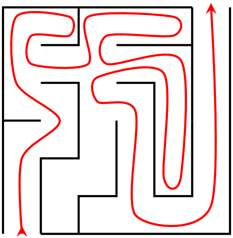
Topological Sort

Depth-First Search: Basic Ideas

Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s

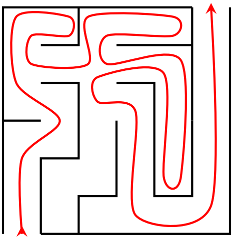
Depth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- As soon as we discover a vertex, explore from it $\rightsquigarrow$ Solving Mazes

Depth-First Search: Basic Ideas



Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex s
- As soon as we discover a vertex, explore from it $\rightsquigarrow$ Solving Mazes
- Two time stamps for every vertex: Discovery Time, Finishing Time

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

   assert(s in G.vertices())

   # Initialize graph
   for v in G.vertices():
      v.predecessor = None
      v.colour = "white"
   dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
   s.colour = "grey"
   s.d = time() # .d = discovery time
   for v in s.adjacent()
       if v.colour = "white"
           v.predecessor = s
           dfsRecurse(G,v)
   s.colour = "black"
   s.f = time() # .f = finish time
```

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

   assert(s in G.vertices())

   # Initialize graph
   for v in G.vertices():
       v.predecessor = None
       v.colour = "white"
   dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
   s.colour = "grey"
   s.d = time() # .d = discovery time
   for v in s.adjacent()
        if v.colour = "white"
            v.predecessor = s
            dfsRecurse(G,v)
   s.colour = "black"
   s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

  assert(s in G.vertices())

  # Initialize graph
  for v in G.vertices():
      v.predecessor = None
      v.colour = "white"
  dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
  s.colour = "grey"
  s.d = time() # .d = discovery time
  for v in s.adjacent()
      if v.colour = "white"
          v.predecessor = s
          dfsRecurse(G,v)
  s.colour = "black"
  s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

    assert(s in G.vertices())

    # Initialize graph
    for v in G.vertices():
        v.predecessor = None
        v.colour = "white"
    dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
    s.colour = "grey"
    s.d = time() # .d = discovery time
    for v in s.adjacent()
        if v.colour = "white"
            v.predecessor = s
            dfsRecurse(G,v)
    s.colour = "black"
    s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

   assert(s in G.vertices())

   # Initialize graph
   for v in G.vertices():
       v.predecessor = None
       v.colour = "white"
   dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
   s.colour = "grey"
   s.d = time() # .d = discovery time
   for v in s.adjacent()
       if v.colour = "white"
           v.predecessor = s
           dfsRecurse(G,v)
   s.colour = "black"
   s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors

Depth-First-Search: Pseudocode

```
def dfs(G,s):
    Run DFS on the given graph G
    starting from the given source s

   assert(s in G.vertices())

   # Initialize graph
   for v in G.vertices():
       v.predecessor = None
       v.colour = "white"
   dfsRecurse(G,s)
```

```
def dfsRecurse(G,s):
   s.colour = "grey"
   s.d = time() # .d = discovery time
   for v in s.adjacent()
       if v.colour = "white"
           v.predecessor = s
           dfsRecurse(G,v)
   s.colour = "black"
   s.f = time() # .f = finish time
```

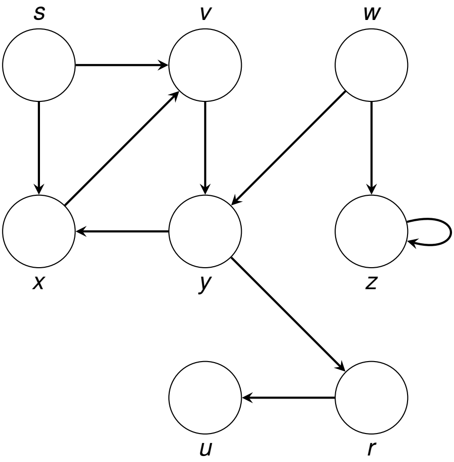
- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

  White = Unvisited

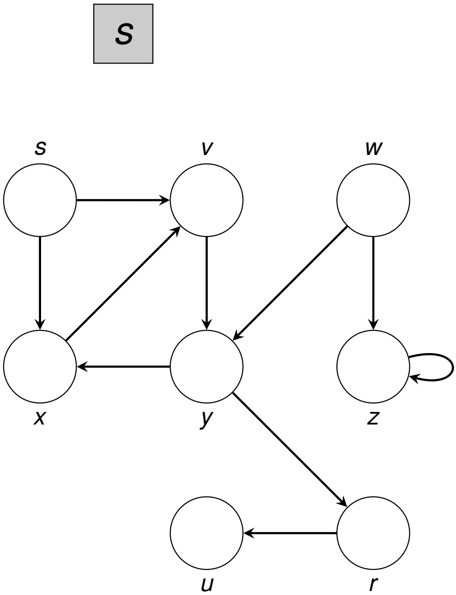
  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime $O(V + E)$

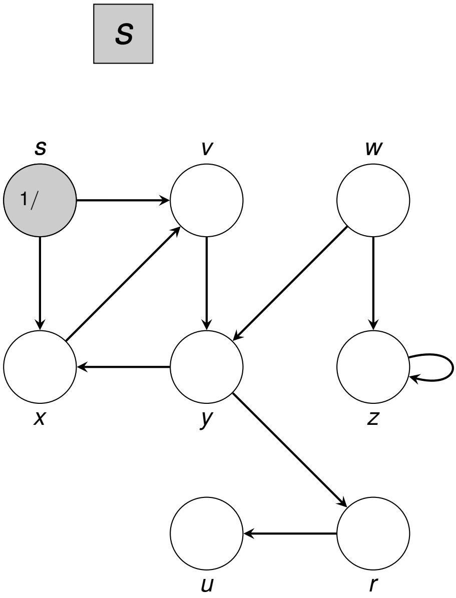
Execution of DFS



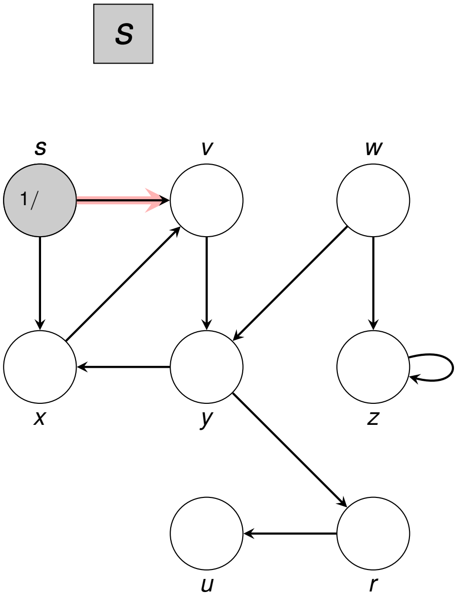
Execution of DFS



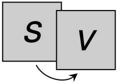
Execution of DFS

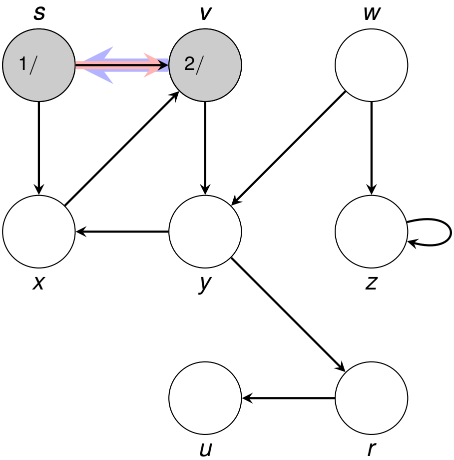


Execution of DFS

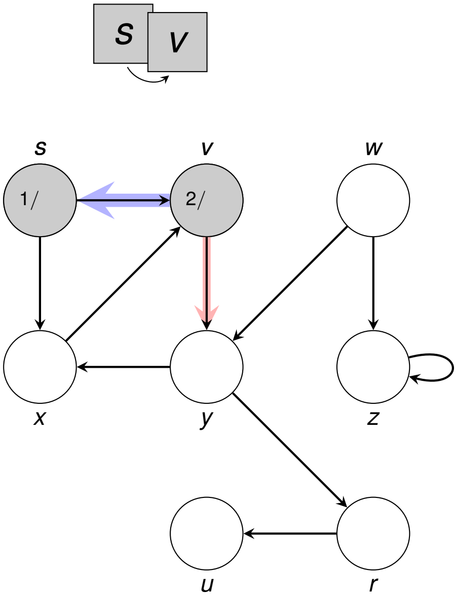


Execution of DFS

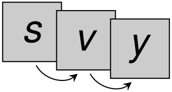


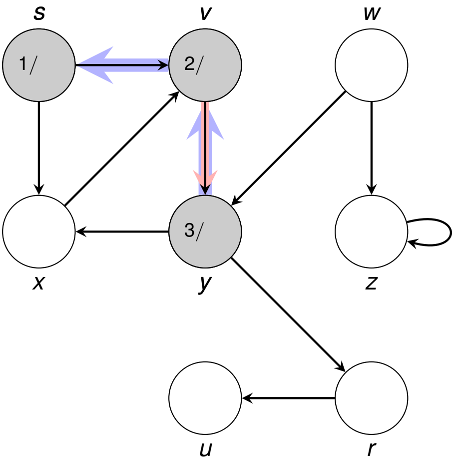


Execution of DFS

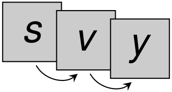


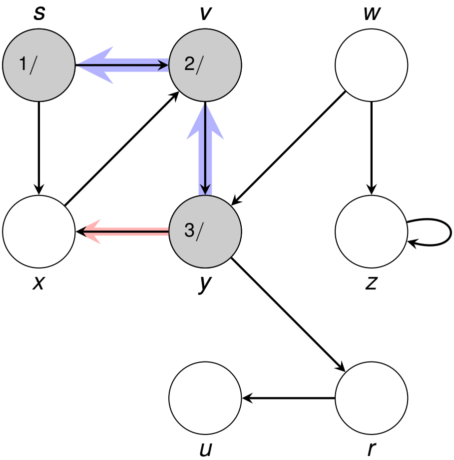
Execution of DFS



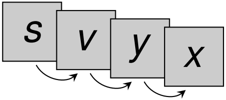


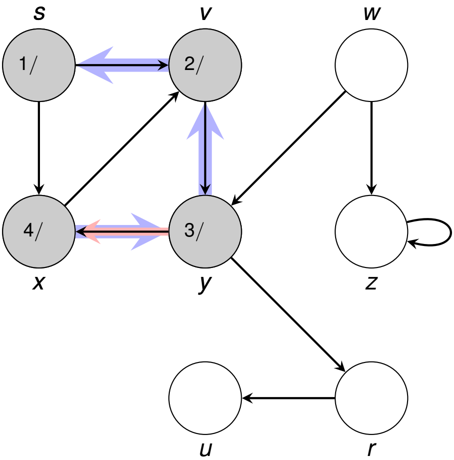
Execution of DFS



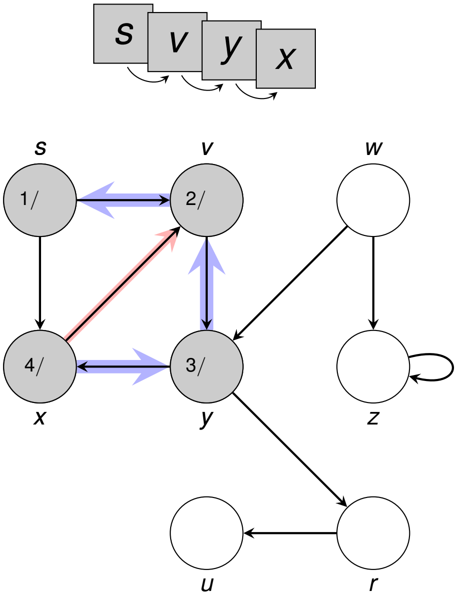


Execution of DFS

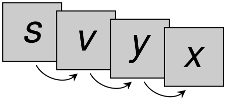


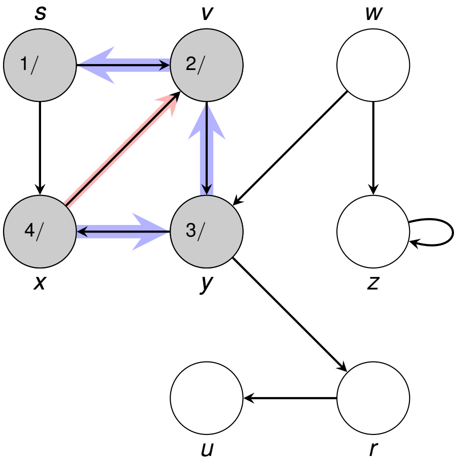


Execution of DFS

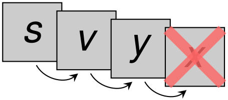


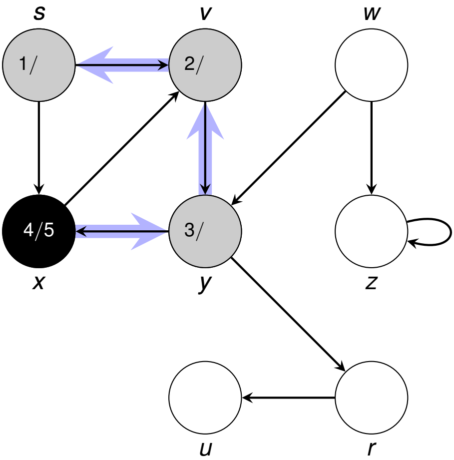
Execution of DFS



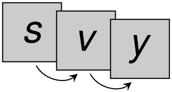


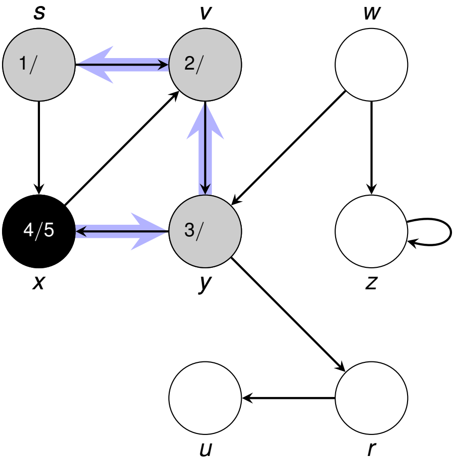
Execution of DFS



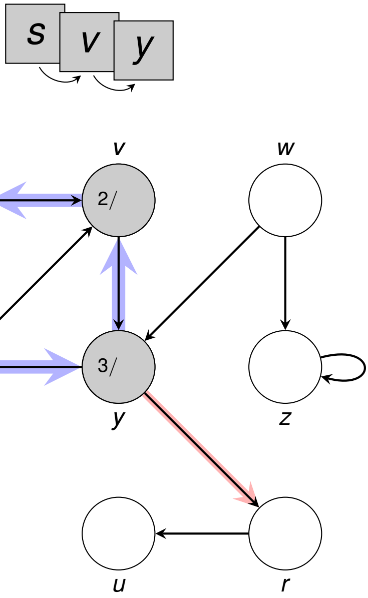


Execution of DFS

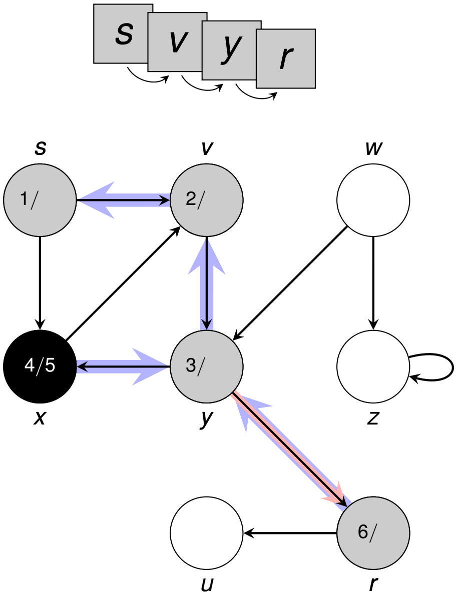




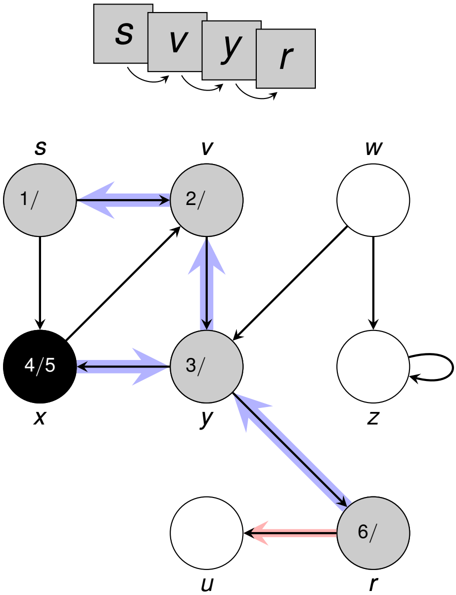
Execution of DFS



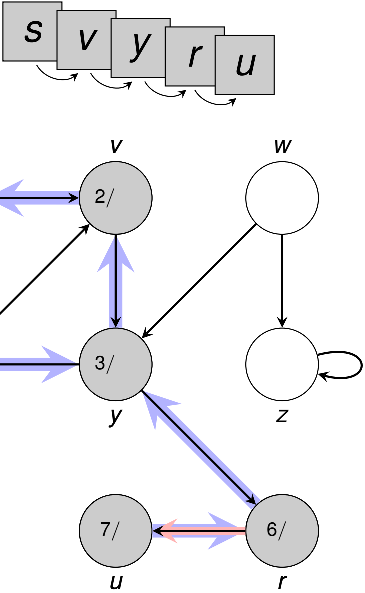
Execution of DFS



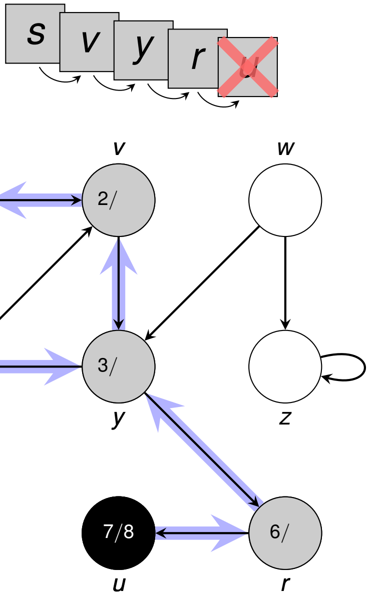
Execution of DFS



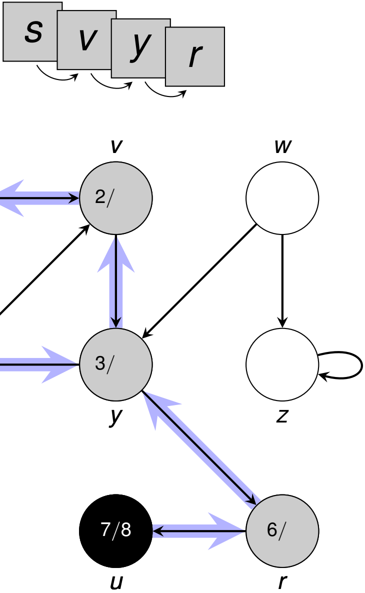
Execution of DFS



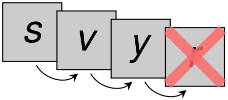
Execution of DFS

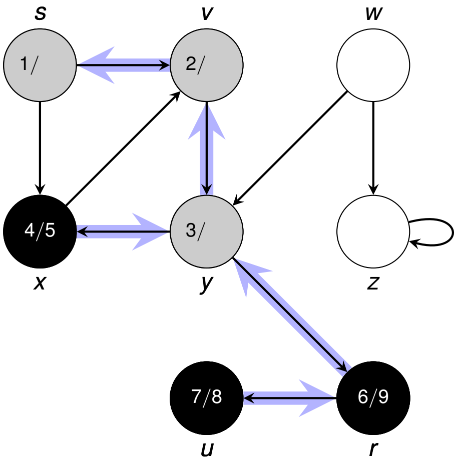


Execution of DFS

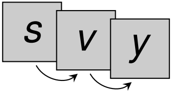


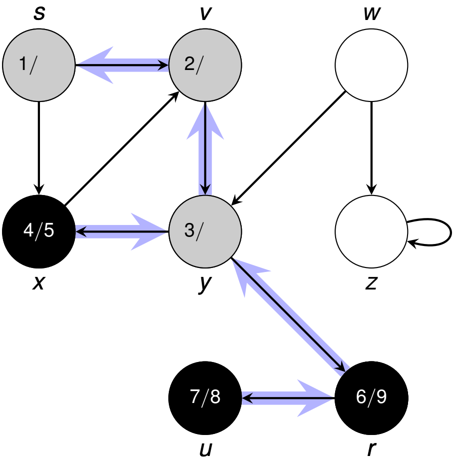
Execution of DFS



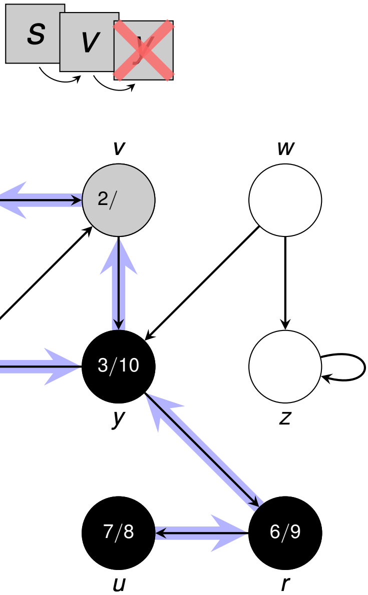


Execution of DFS

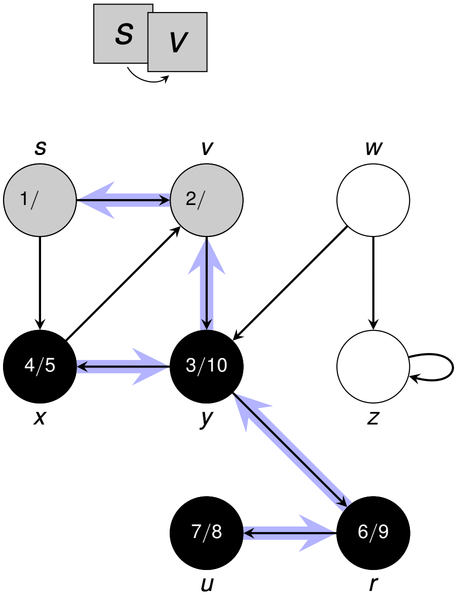




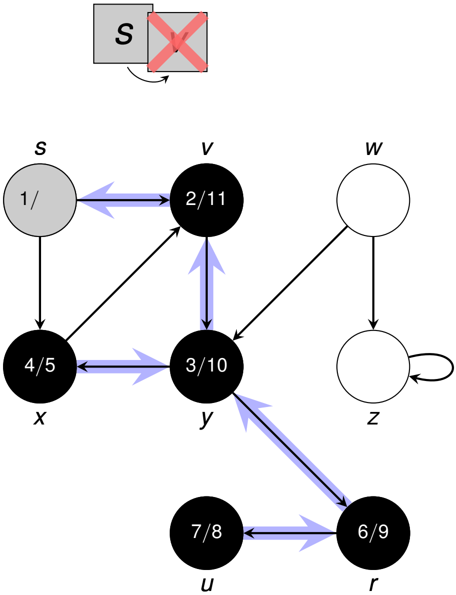
Execution of DFS



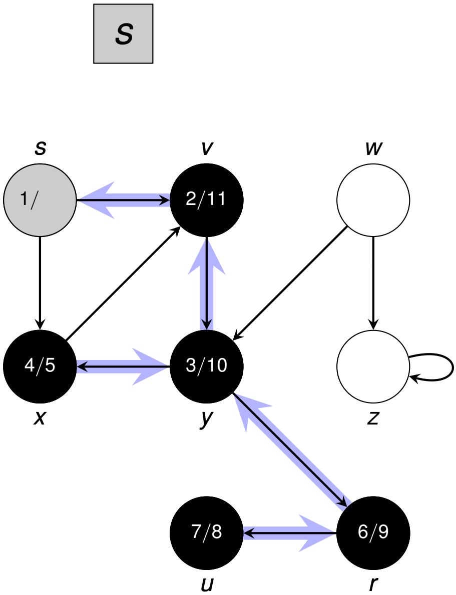
Execution of DFS



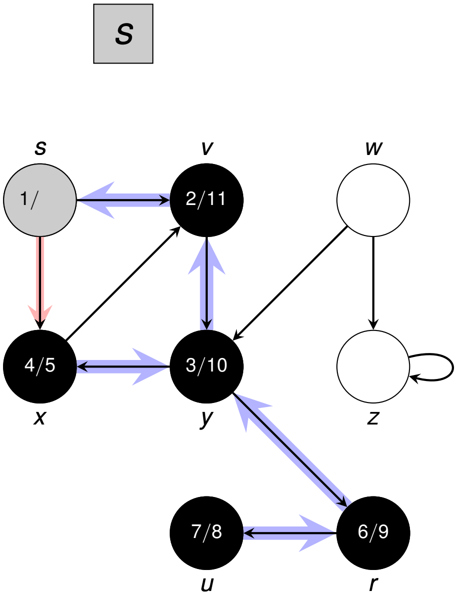
Execution of DFS



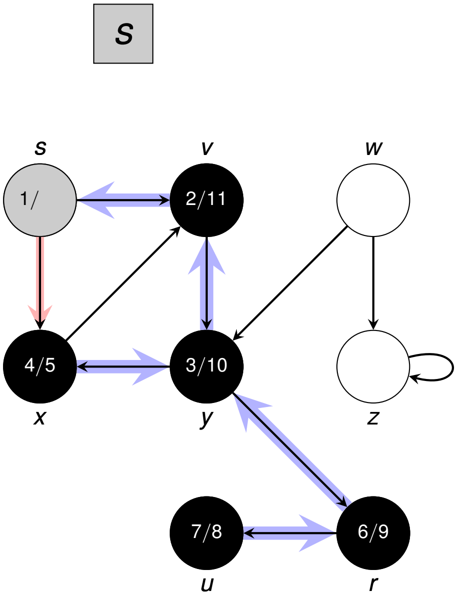
Execution of DFS



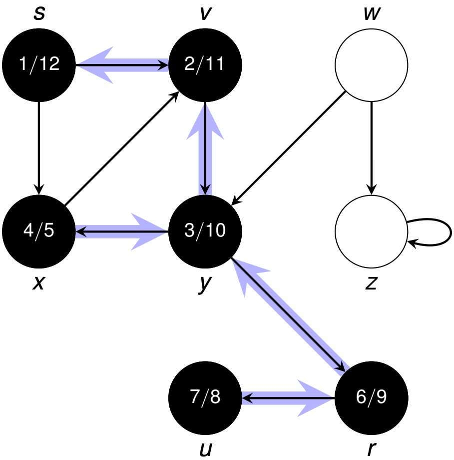
Execution of DFS



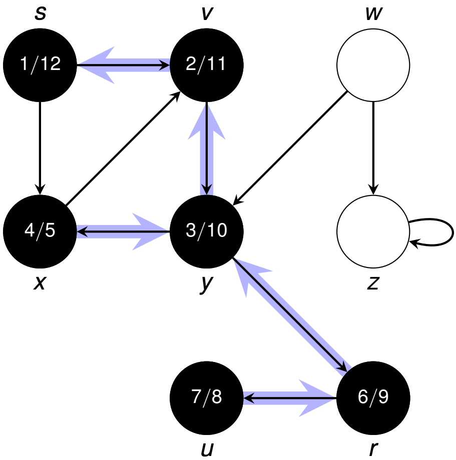
Execution of DFS



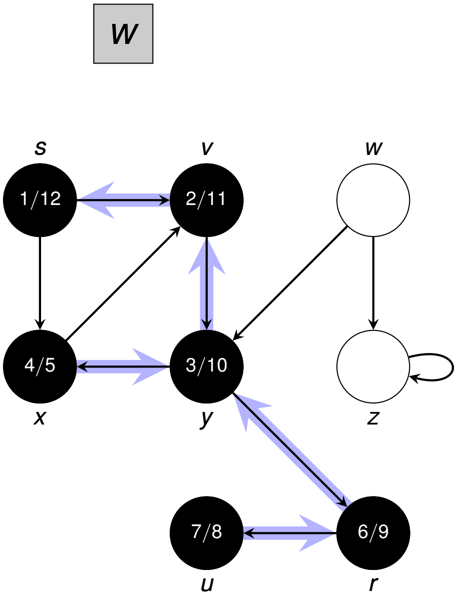
Execution of DFS



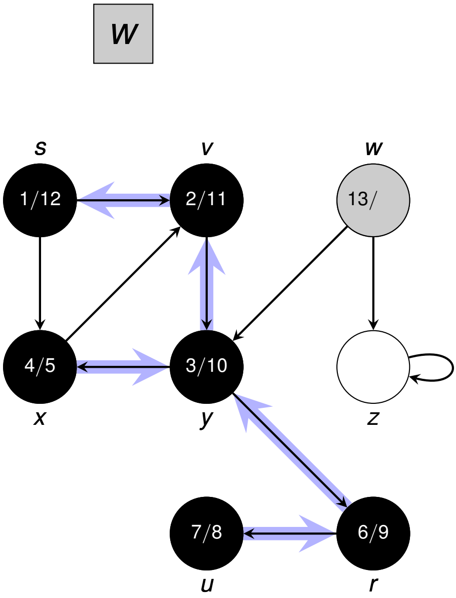
Execution of DFS



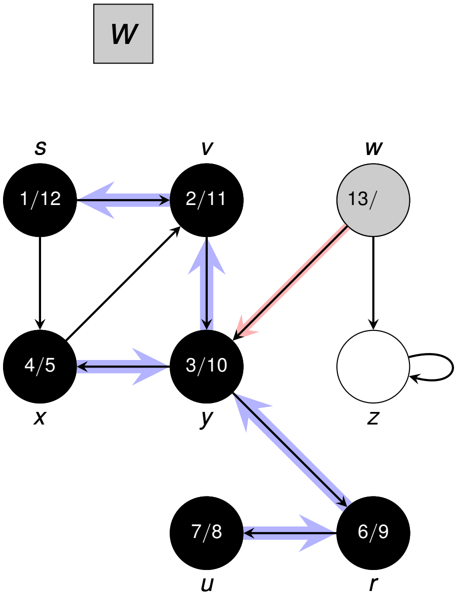
Execution of DFS



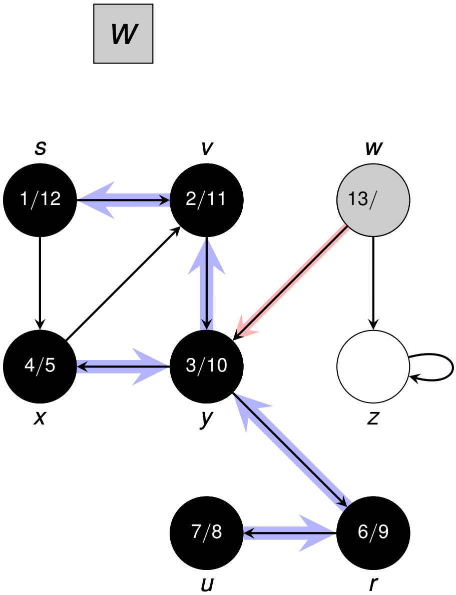
Execution of DFS



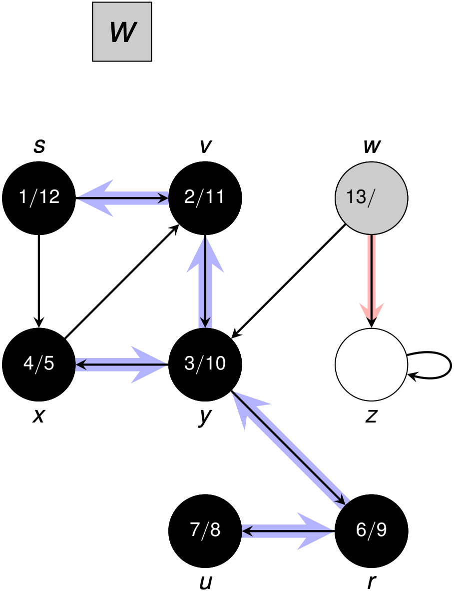
Execution of DFS



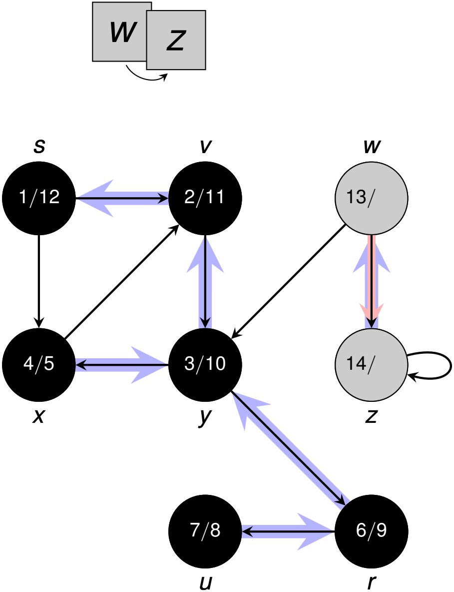
Execution of DFS



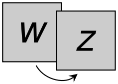
Execution of DFS

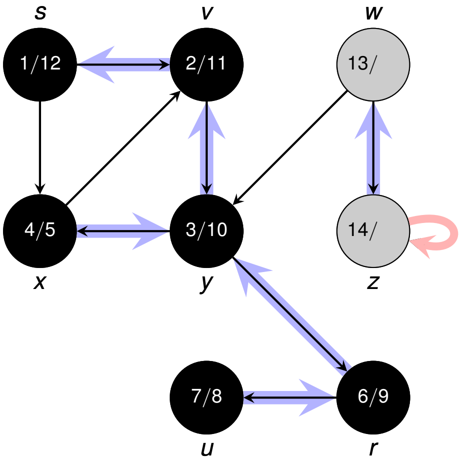


Execution of DFS

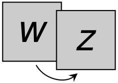


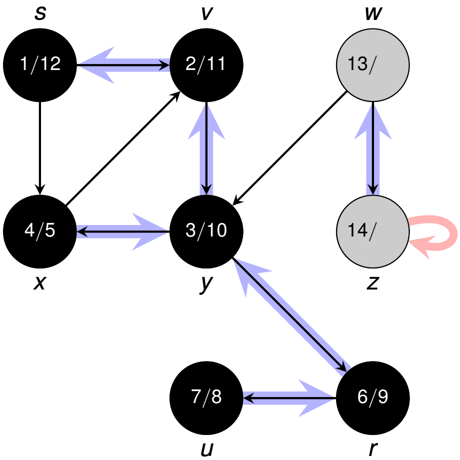
Execution of DFS



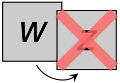


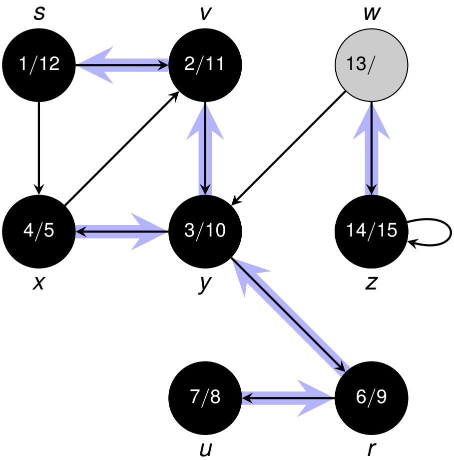
Execution of DFS



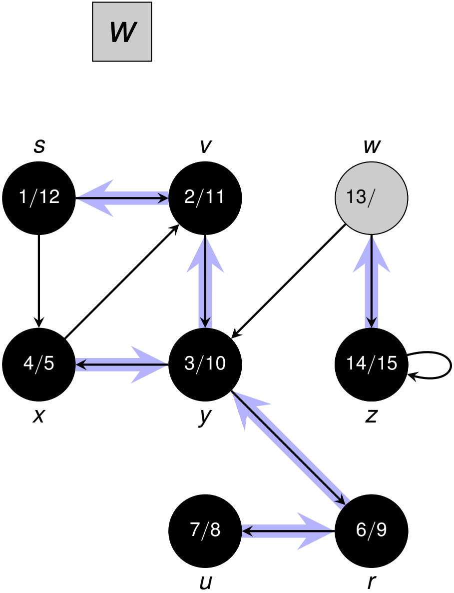


Execution of DFS

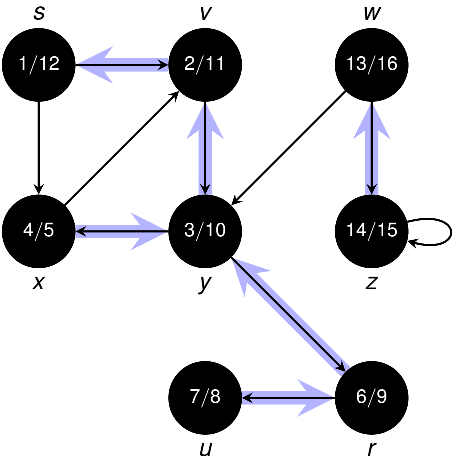




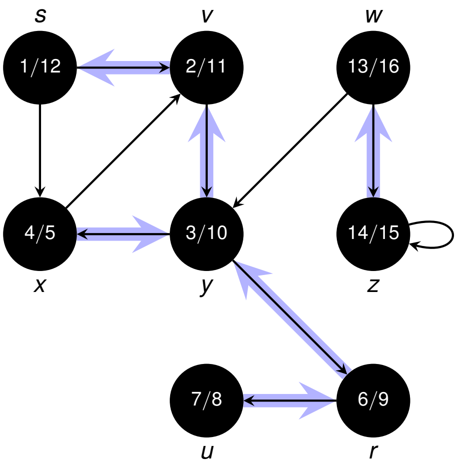
Execution of DFS



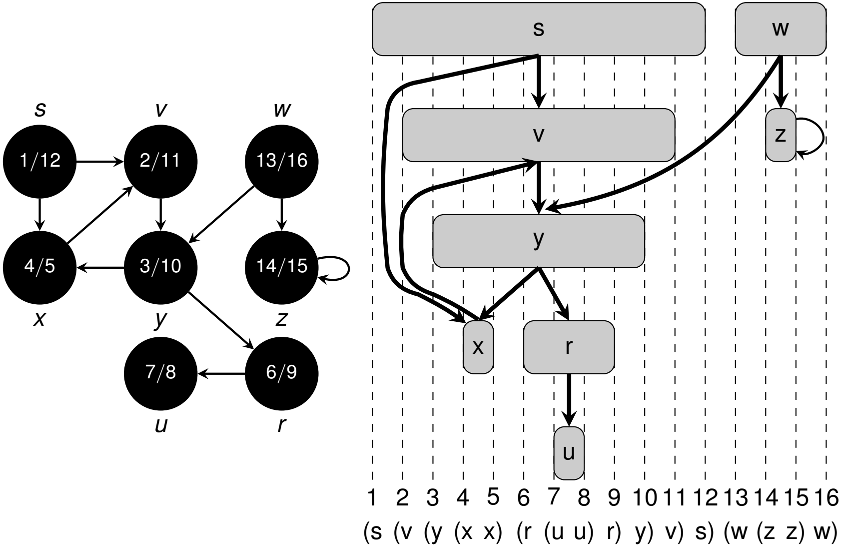
Execution of DFS



Execution of DFS



Paranthesis Theorem (Theorem 22.7)



Outline

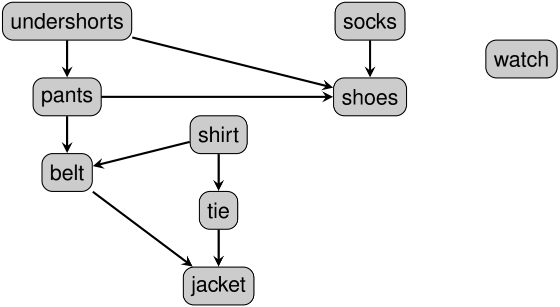
Introduction to Graphs and Graph Searching

Breadth-First Search

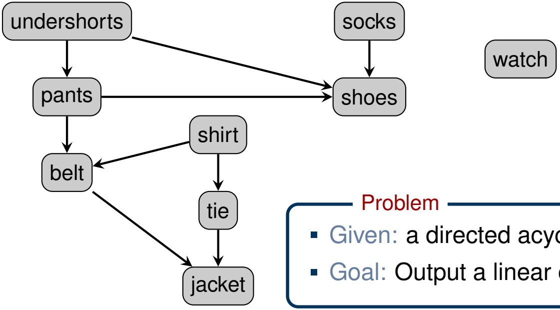
Depth-First Search

Topological Sort

Topological Sort



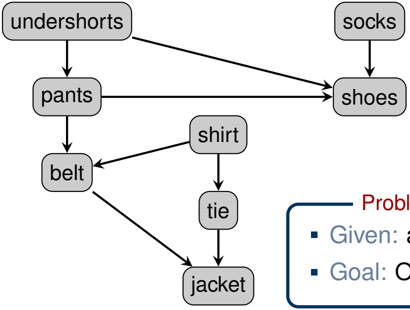
Topological Sort



Problem

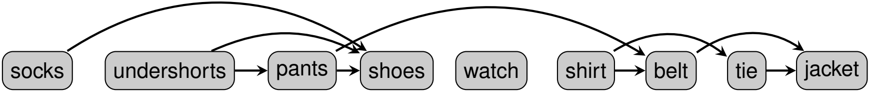
- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices

Topological Sort

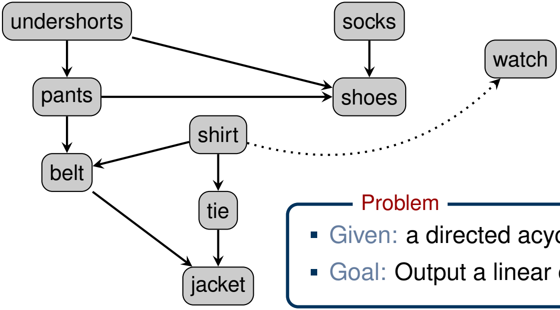


Problem

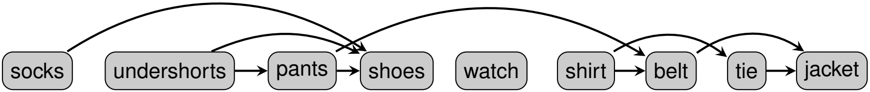
- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices



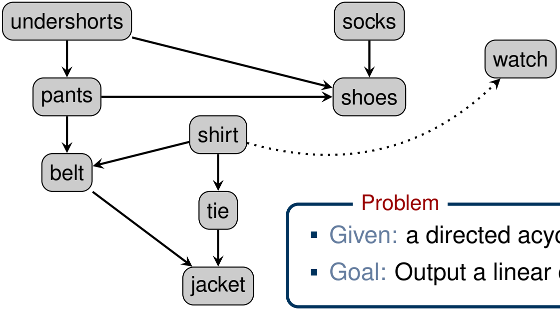
Topological Sort



Problem
- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices

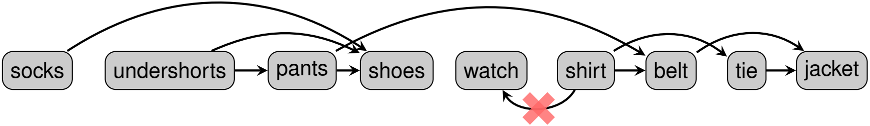


Topological Sort

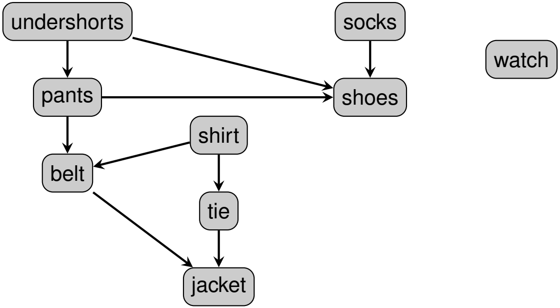


Problem

- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices



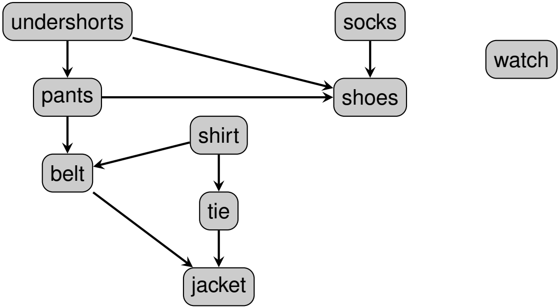
Solving Topological Sort



Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Solving Topological Sort

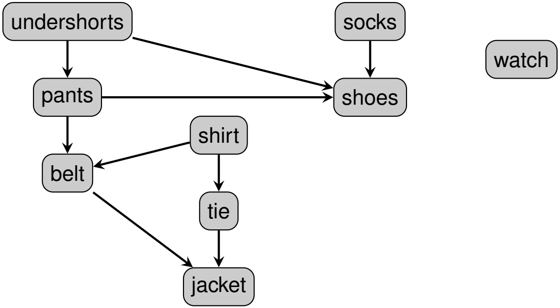


Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Runtime $O(V + E)$

Solving Topological Sort



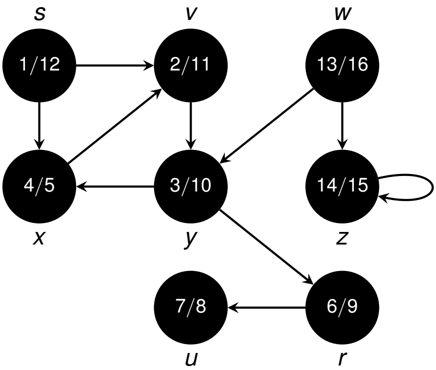
Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

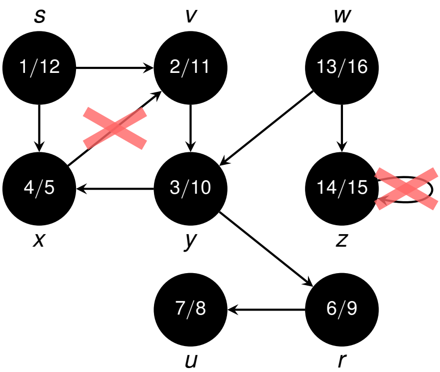
Runtime $O(V+E)$

Don't need to sort the vertices – use DFS directly!

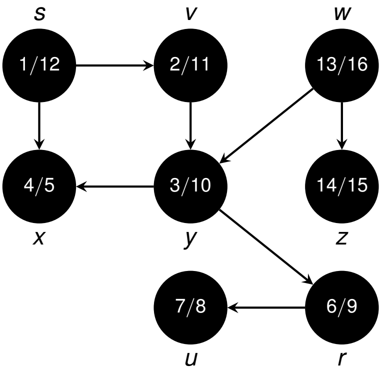
Execution of Knuth's Algorithm



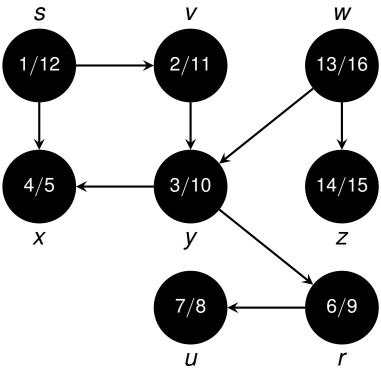
Execution of Knuth's Algorithm



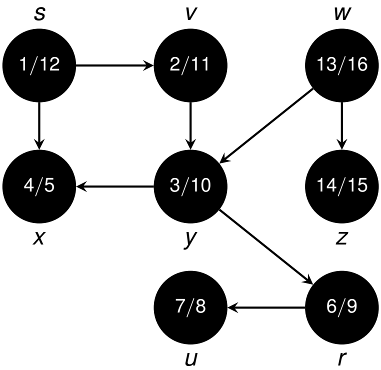
Execution of Knuth's Algorithm



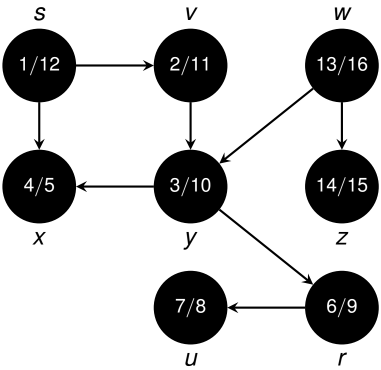
Execution of Knuth's Algorithm

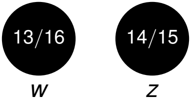


Execution of Knuth's Algorithm

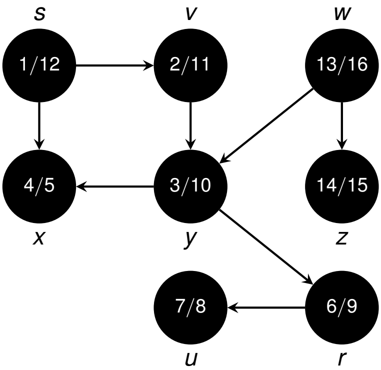


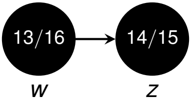
Execution of Knuth's Algorithm



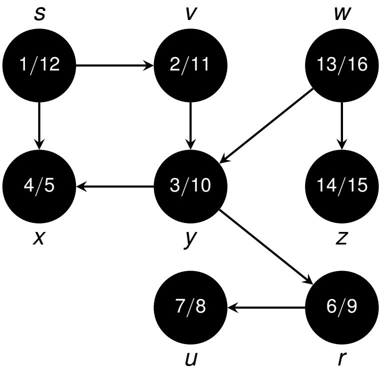


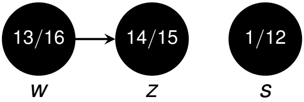
Execution of Knuth's Algorithm



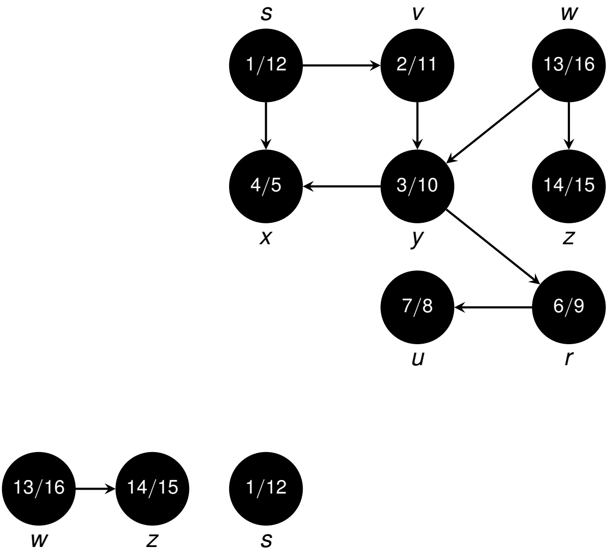


Execution of Knuth's Algorithm

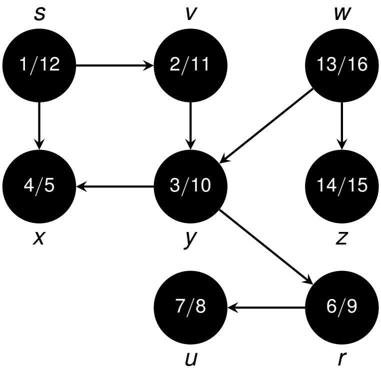




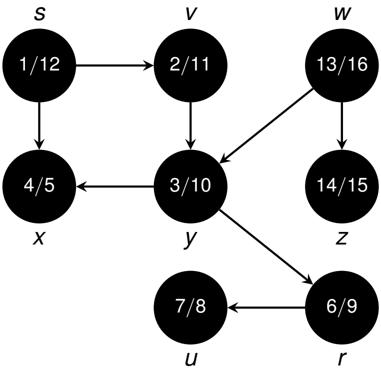
Execution of Knuth's Algorithm



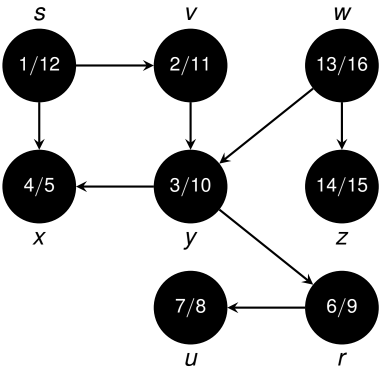
Execution of Knuth's Algorithm

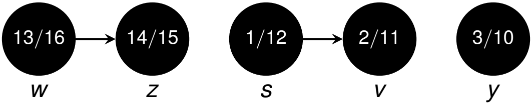


Execution of Knuth's Algorithm

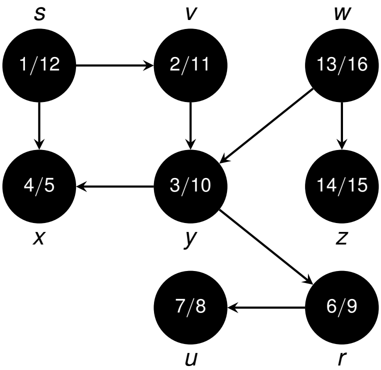


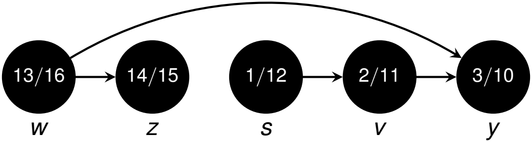
Execution of Knuth's Algorithm



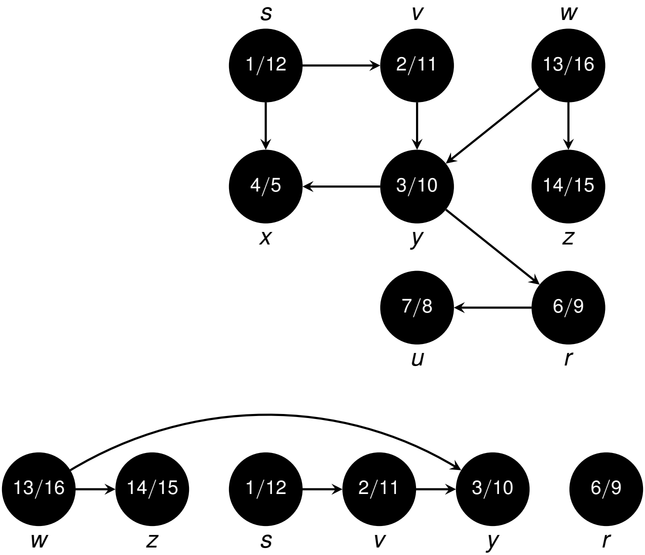


Execution of Knuth's Algorithm

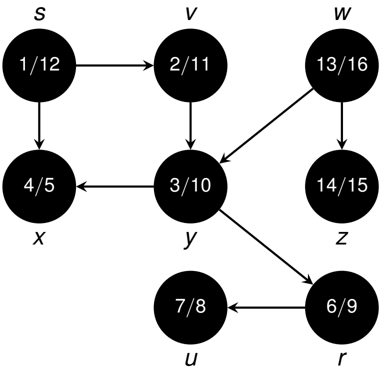


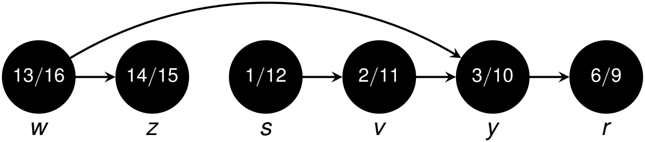


Execution of Knuth's Algorithm

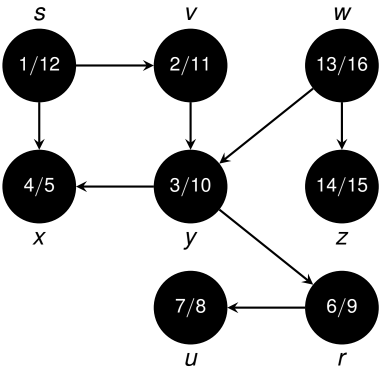


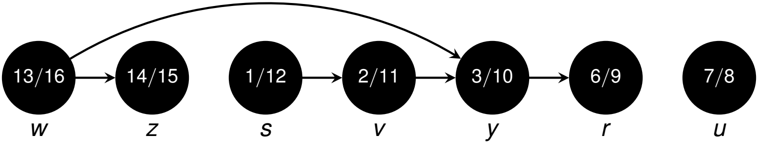
Execution of Knuth's Algorithm



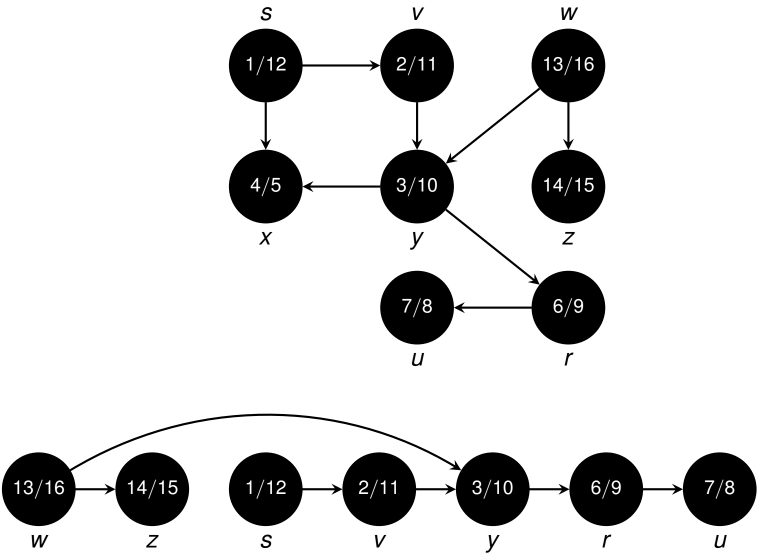


Execution of Knuth's Algorithm

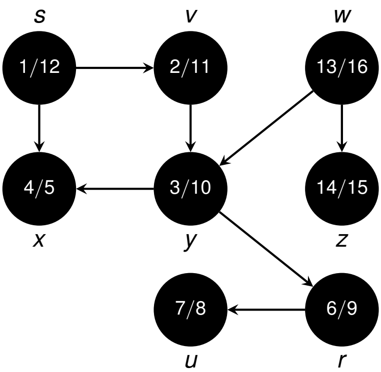


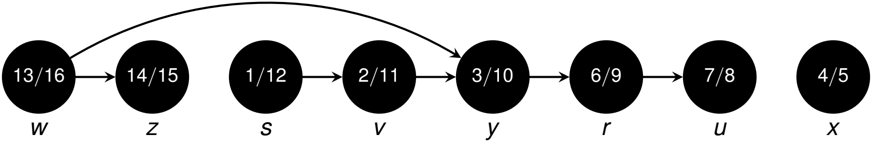


Execution of Knuth's Algorithm

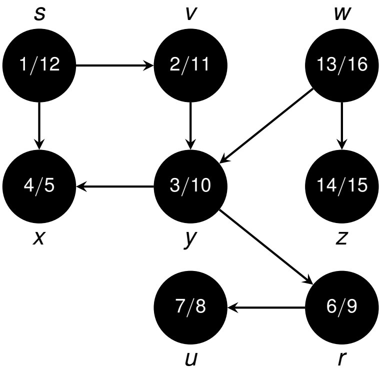


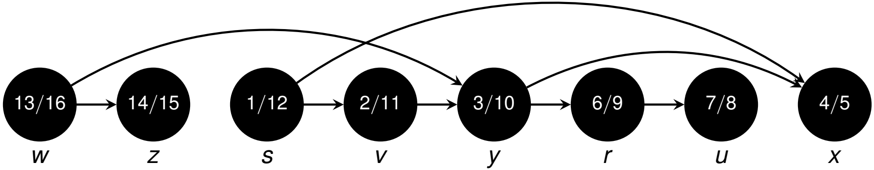
Execution of Knuth's Algorithm



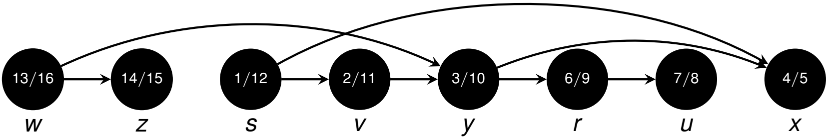


Execution of Knuth's Algorithm





Correctness of Topological Sort using DFS



Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

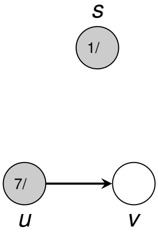
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,



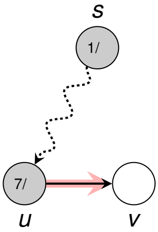
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$



Correctness of Topological Sort using DFS

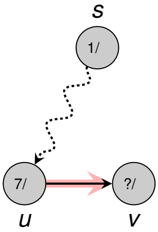
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey,*



Correctness of Topological Sort using DFS

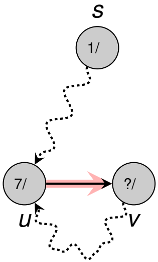
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey,*



Correctness of Topological Sort using DFS

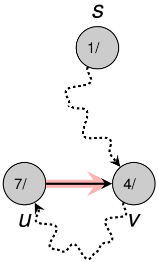
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).



Correctness of Topological Sort using DFS

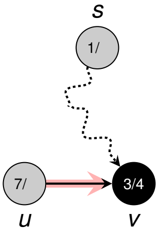
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black,*



Correctness of Topological Sort using DFS

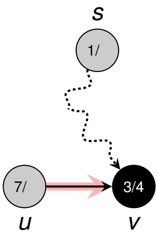
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle* *(can't happen, because G is acyclic!)*.
  2. *If v is black, then $v.f < u.f$.*



Correctness of Topological Sort using DFS

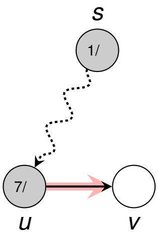
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white,*



Correctness of Topological Sort using DFS

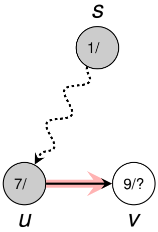
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*



Correctness of Topological Sort using DFS

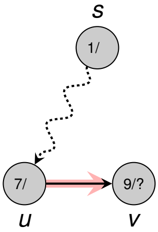
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*



Correctness of Topological Sort using DFS

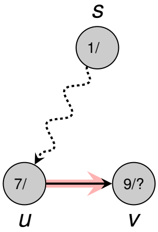
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*



$\Rightarrow$ In all cases $v.f < u.f$, so v appears after u.

Correctness of Topological Sort using DFS

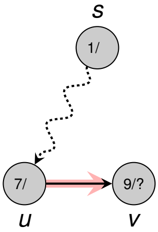
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*

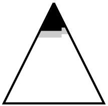


$\Rightarrow$ In all cases $v.f < u.f$, so v appears after u. $\square$

Summary of Graph Searching

Breadth-First-Search

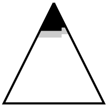
- vertices are processed by a queue
- computes distances and shortest paths
  $\rightsquigarrow$ similar idea used later in Prim's and Dijkstra's algorithm
- Runtime $\mathcal{O}(V + E)$



Summary of Graph Searching

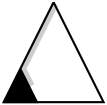
Breadth-First-Search

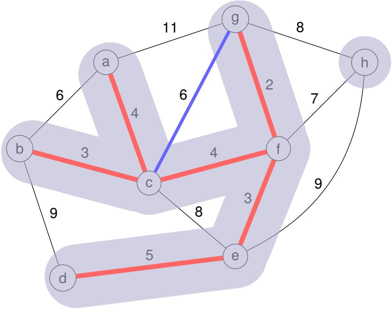
- vertices are processed by a queue
- computes distances and shortest paths
  $\rightsquigarrow$ similar idea used later in Prim's and Dijkstra's algorithm
- Runtime $\mathcal{O}(V+E)$



Depth-First-Search

- vertices are processed by recursive calls ($\approx$ stack)
- discovery and finishing times
- application: Topogical Sorting of DAGs
- Runtime $\mathcal{O}(V+E)$





6.3: Minimum Spanning Tree

Frank Stajano Thomas Sauerwald

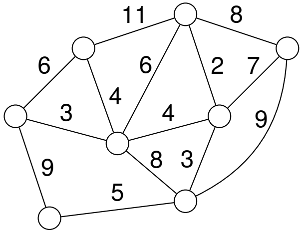
Lent 2016

UNIVERSITY OF CAMBRIDGE

Minimum Spanning Tree Problem

Minimum Spanning Tree Problem

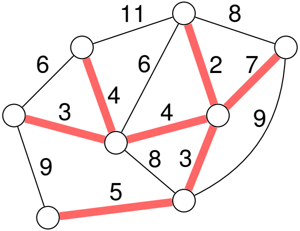
- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights



Minimum Spanning Tree Problem

Minimum Spanning Tree Problem

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

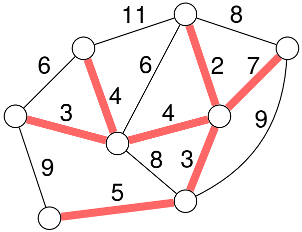


Minimum Spanning Tree Problem

Minimum Spanning Tree Problem

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

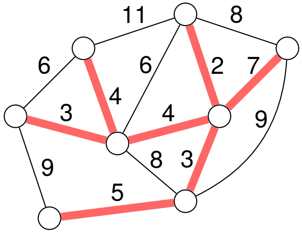
Must be necessarily a tree!



Minimum Spanning Tree Problem

Minimum Spanning Tree Problem

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices



Applications

- Street Networks, Wiring Electronic Components, Laying Pipes
- Weights may represent distances, costs, travel times, capacities, resistance etc.

Generic Algorithm

```
def minimum spanningTree(G)
  A = empty set of edges
  while A does not span all vertices yet:
    add a safe edge to A
```

Generic Algorithm

```
def minimum spanningTree(G)
  A = empty set of edges
  while A does not span all vertices yet:
    add a safe edge to A
```

Definition

An edge of G is safe if by adding the edge to A, the resulting subgraph is still a subset of a minimum spanning tree.

Generic Algorithm

```
def minimum spanningTree(G)
  A = empty set of edges
  while A does not span all vertices yet:
    add a safe edge to A
```

Definition

An edge of G is safe if by adding the edge to A, the resulting subgraph is still a subset of a minimum spanning tree.

How to find a safe edge?

Finding safe edges

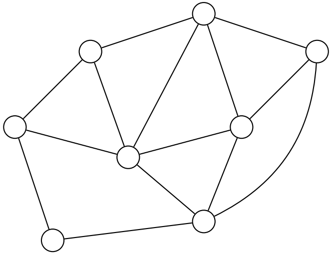
Definitions

- a cut is a partition of V into at least two disjoint sets

Finding safe edges

Definitions

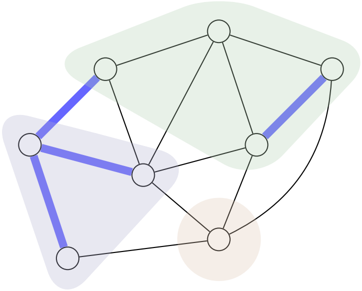
- a cut is a partition of V into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of A goes across the cut



Finding safe edges

Definitions

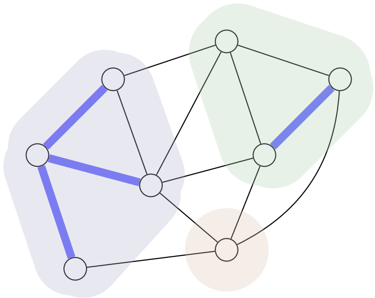
- a cut is a partition of V into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of A goes across the cut



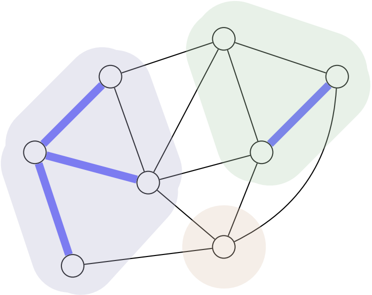
Finding safe edges

Definitions

- a cut is a partition of V into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of A goes across the cut



Finding safe edges



Definitions

- a cut is a partition of V into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of A goes across the cut

Theorem

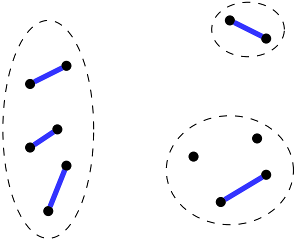
Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:



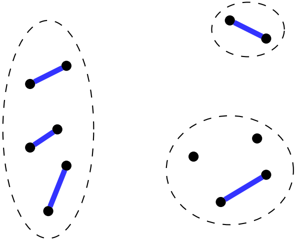
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A



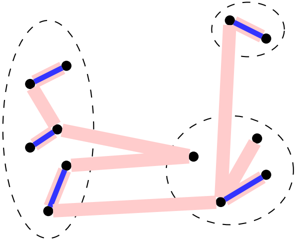
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A



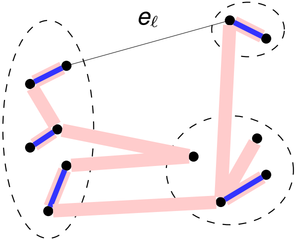
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut



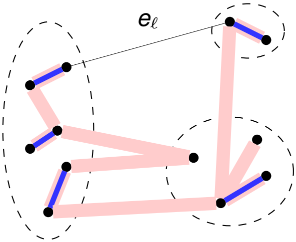
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done



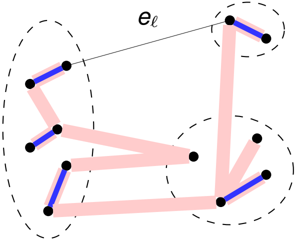
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$,



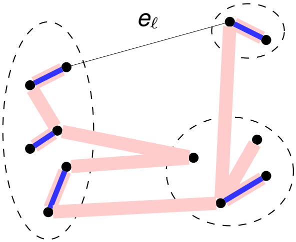
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle



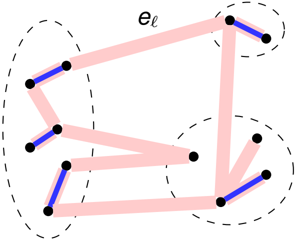
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle



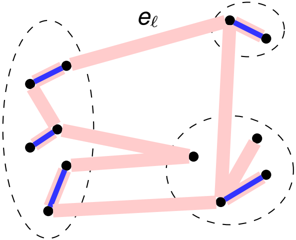
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x



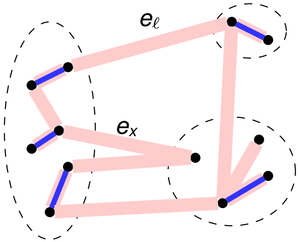
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x



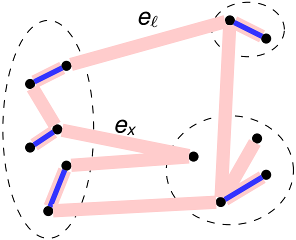
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x
- Consider now the tree $T \cup e_\ell \setminus e_x$:



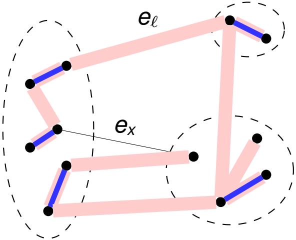
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x
- Consider now the tree $T \cup e_\ell \setminus e_x$:



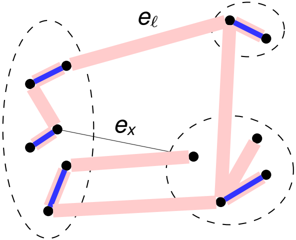
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree



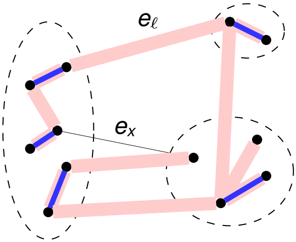
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree
  - If $w(e_\ell) < w(e_x)$, then this spanning tree has smaller cost than T (can't happen!)



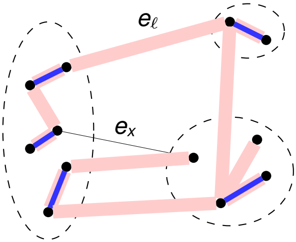
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

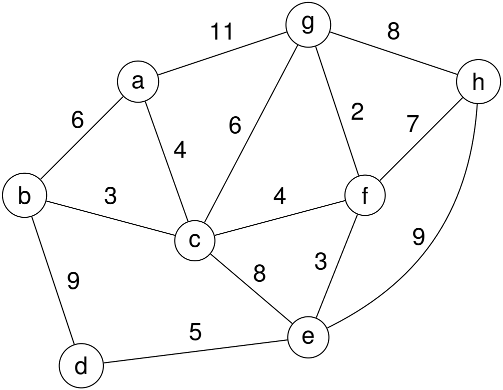
Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree
  - If $w(e_\ell) < w(e_x)$, then this spanning tree has smaller cost than T (can't happen!)
  - If $w(e_\ell) = w(e_x)$, then $T \cup e_\ell \setminus e_x$ is a MST. $\square$



Glimpse at Kruskal's Algorithm

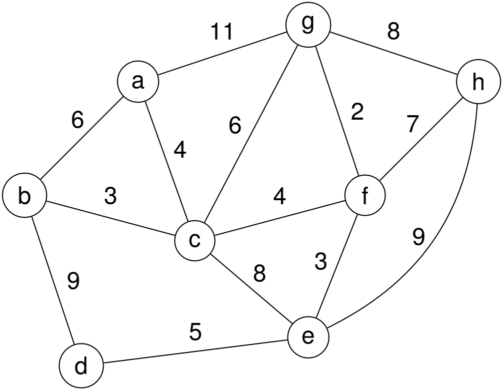
Basic Strategy



Glimpse at Kruskal's Algorithm

Basic Strategy

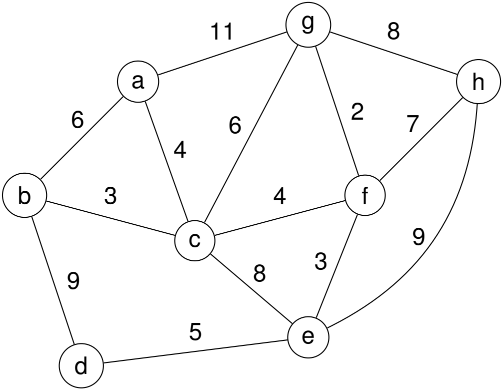
- Let $A \subseteq E$ be a forest, intially empty



Glimpse at Kruskal's Algorithm

Basic Strategy

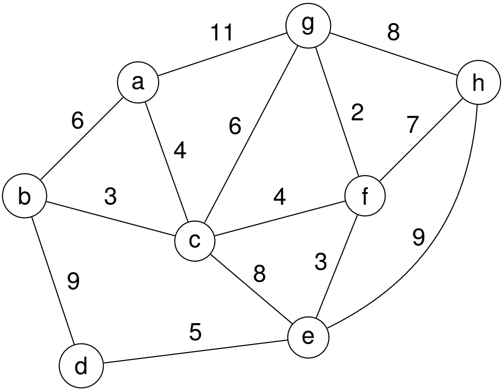
- Let $A \subseteq E$ be a forest, intially empty
- At every step,



Glimpse at Kruskal's Algorithm

Basic Strategy

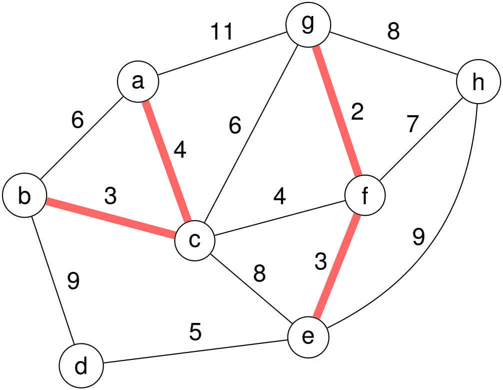
- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:



Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

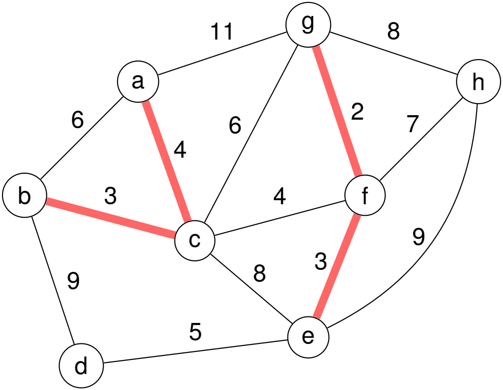


Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle

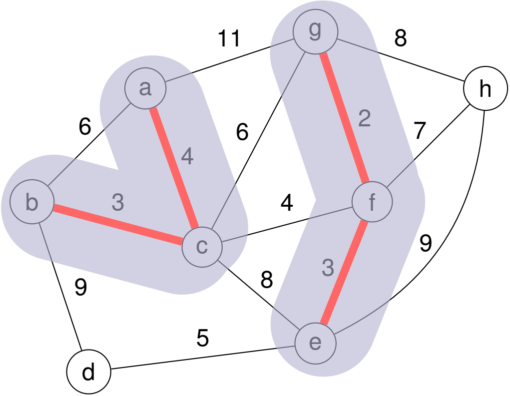


Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle

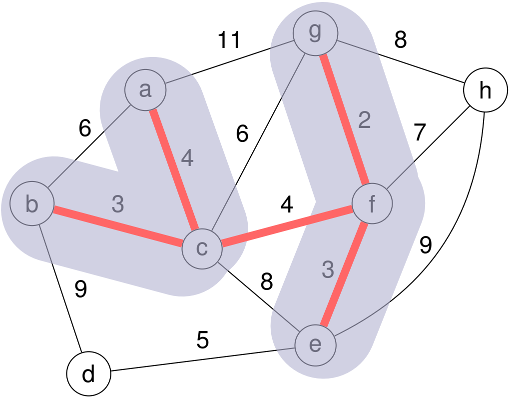


Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle

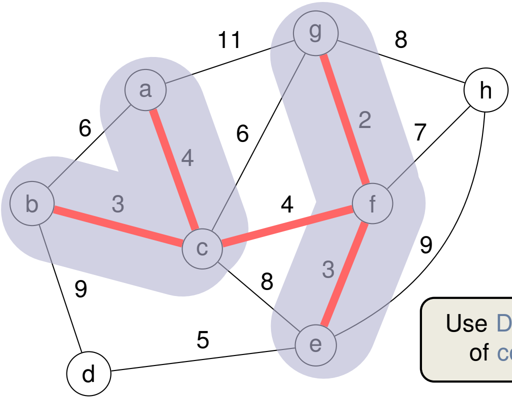


Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle



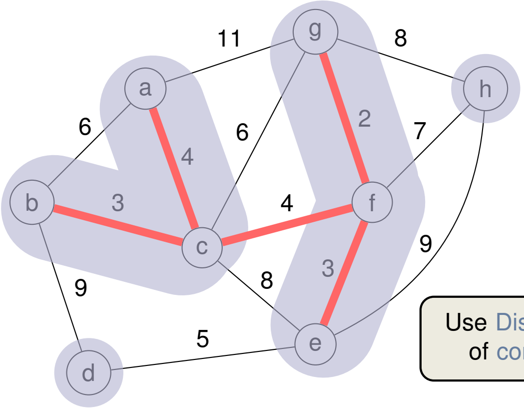
Use Disjoint Sets to keep track of connected components!

Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle



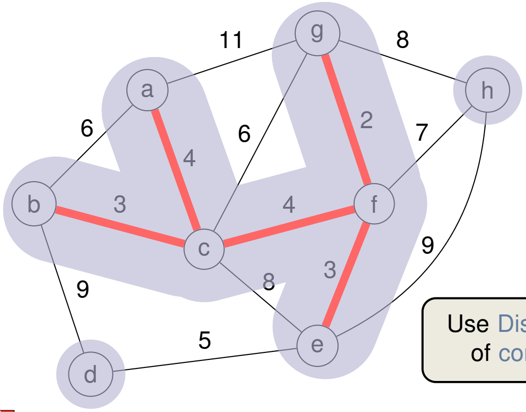
Use Disjoint Sets to keep track of connected components!

Glimpse at Kruskal's Algorithm

Basic Strategy

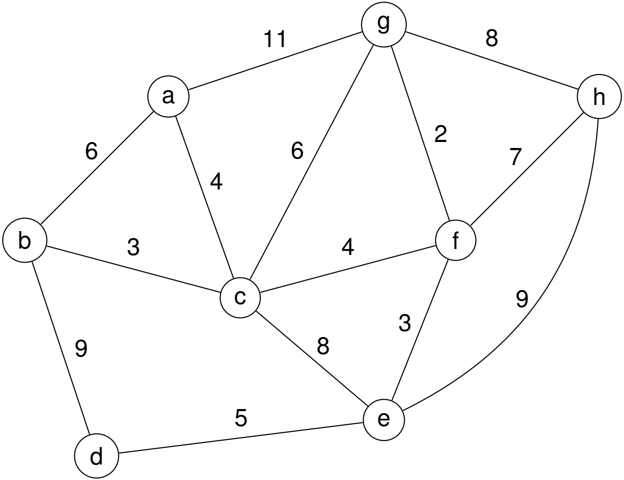
- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle

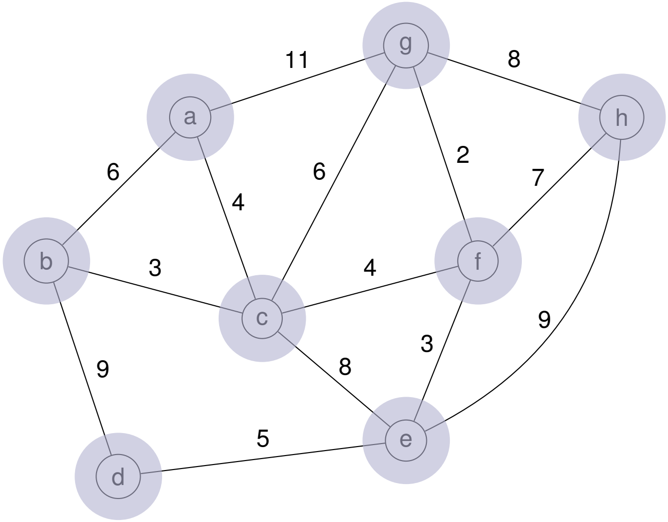


Use Disjoint Sets to keep track of connected components!

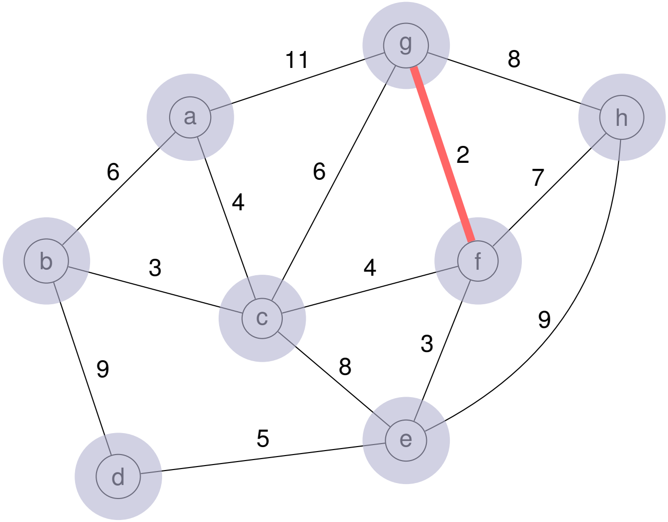
Execution of Kruskal's Algorithm



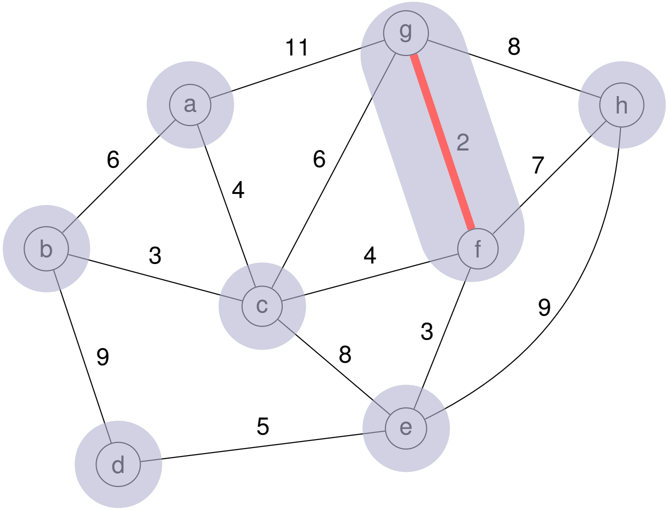
Execution of Kruskal's Algorithm



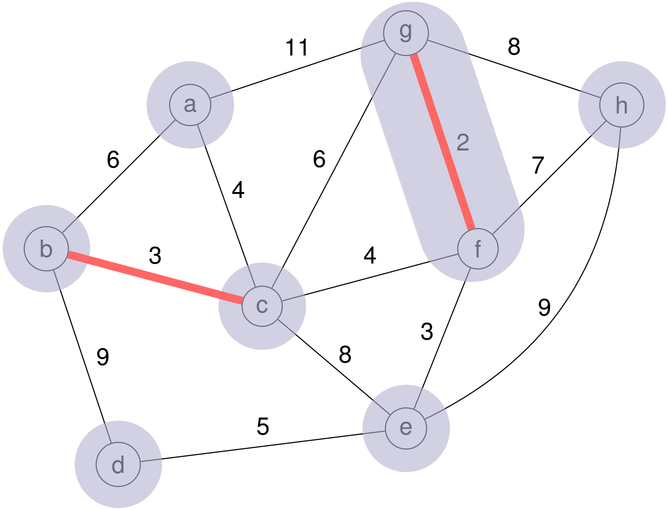
Execution of Kruskal's Algorithm



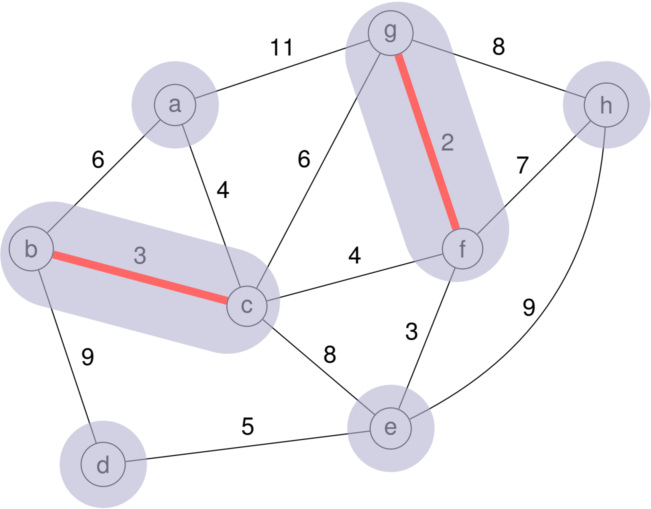
Execution of Kruskal's Algorithm



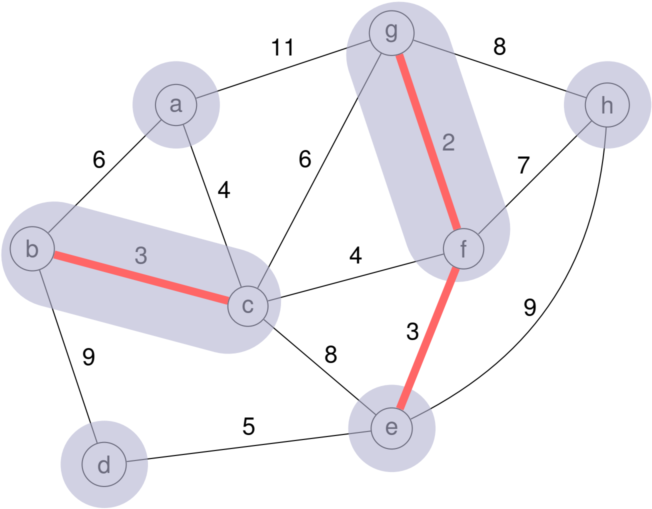
Execution of Kruskal's Algorithm



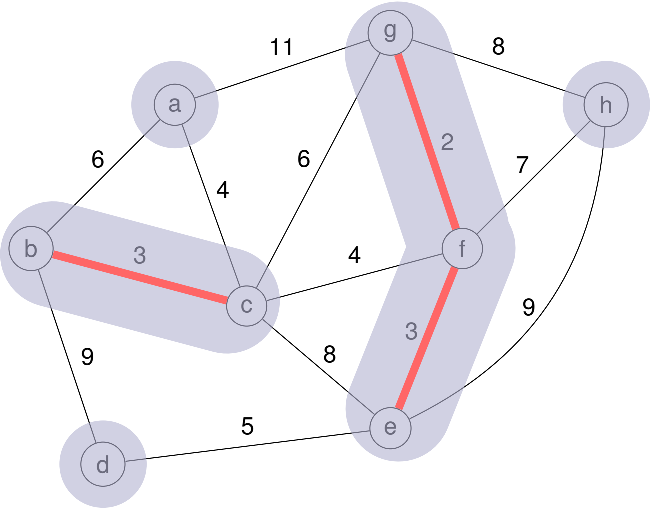
Execution of Kruskal's Algorithm



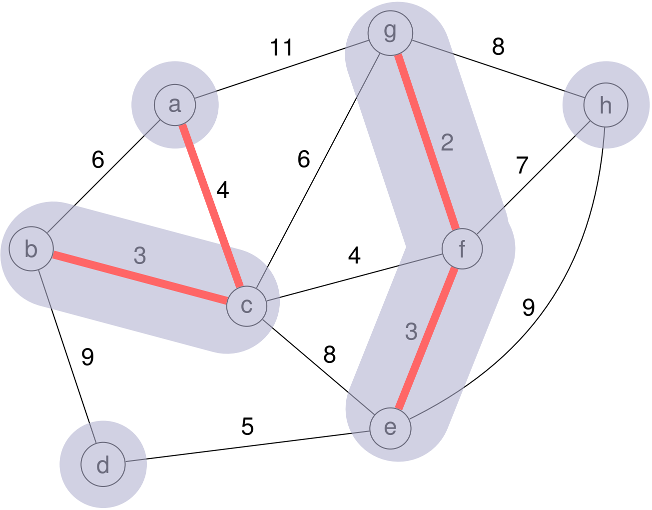
Execution of Kruskal's Algorithm



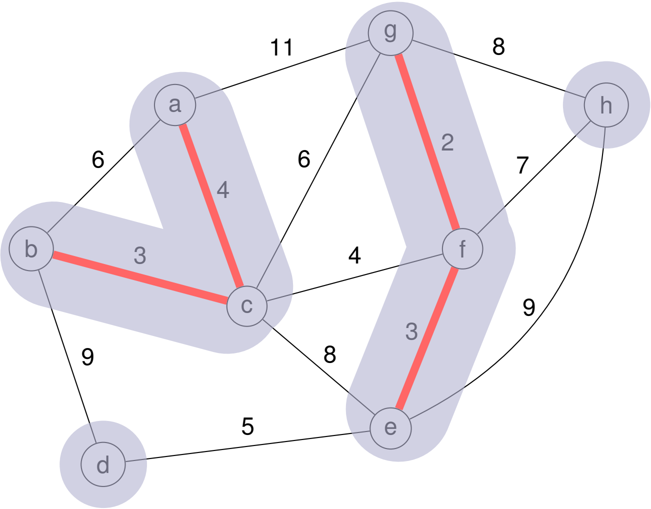
Execution of Kruskal's Algorithm



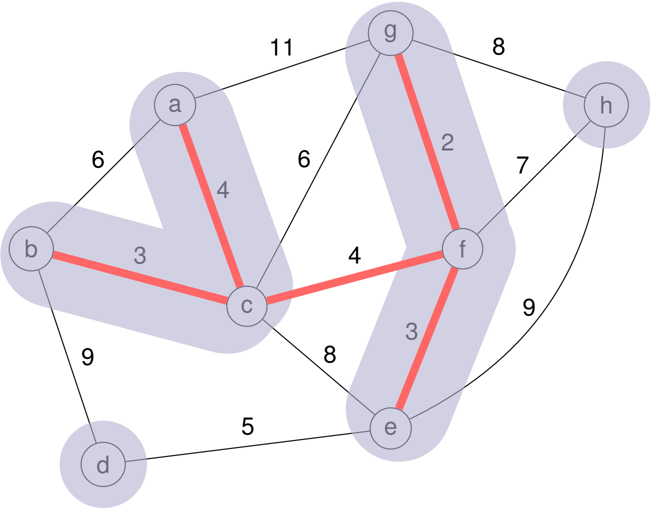
Execution of Kruskal's Algorithm



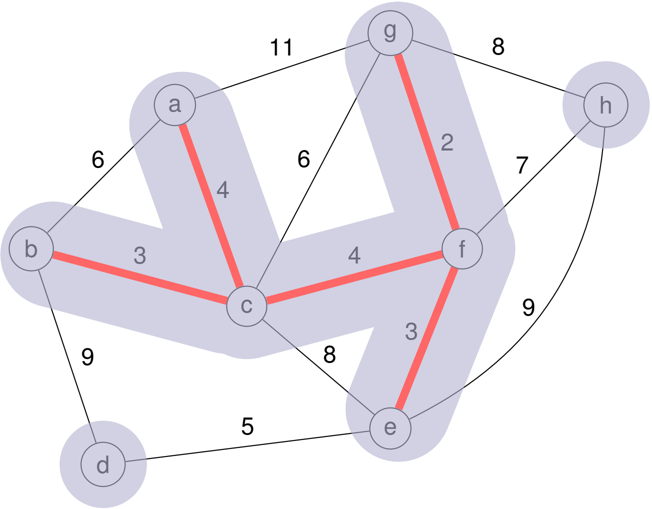
Execution of Kruskal's Algorithm



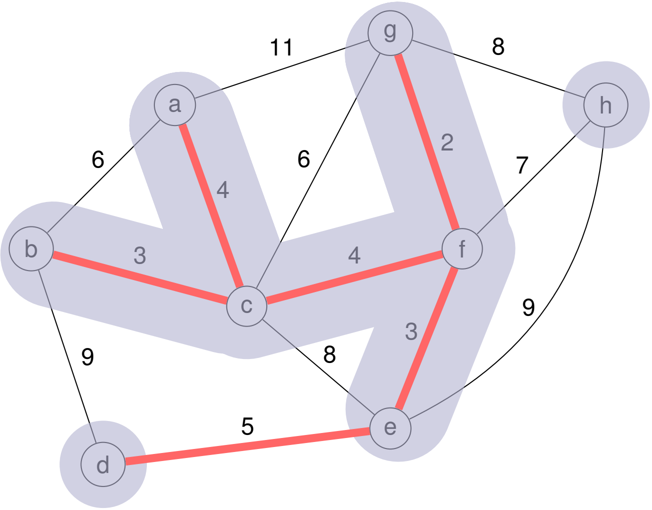
Execution of Kruskal's Algorithm



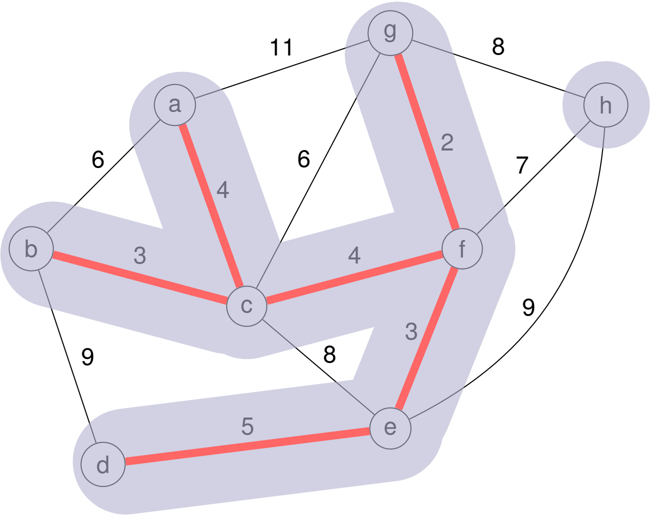
Execution of Kruskal's Algorithm



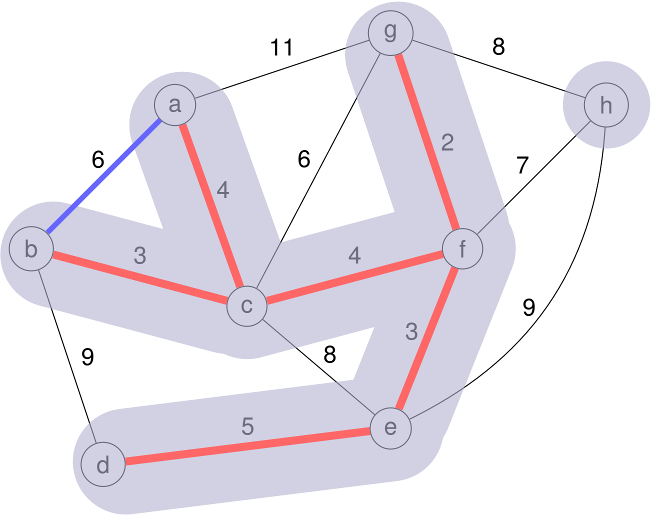
Execution of Kruskal's Algorithm



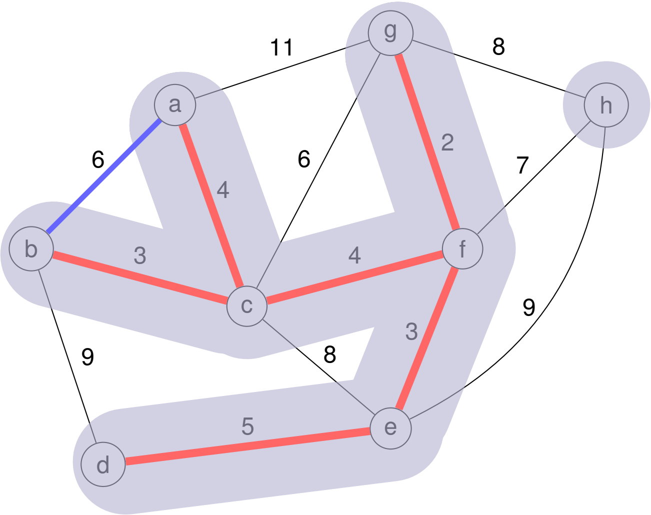
Execution of Kruskal's Algorithm



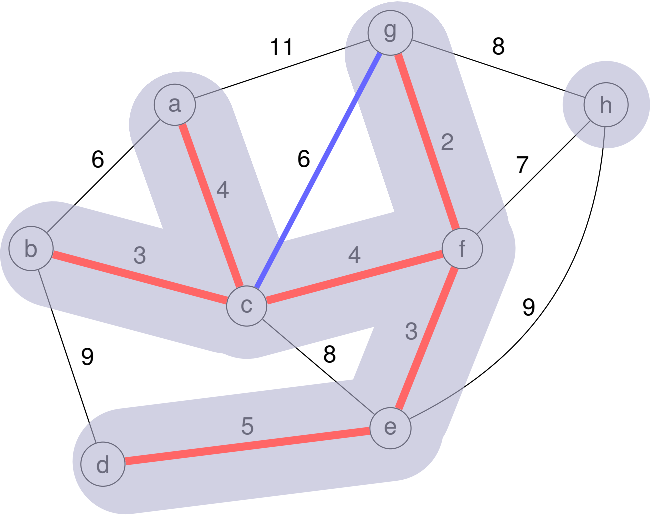
Execution of Kruskal's Algorithm



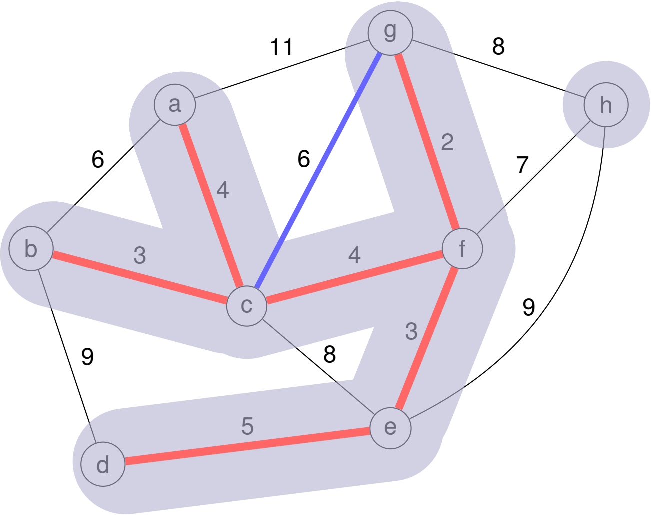
Execution of Kruskal's Algorithm



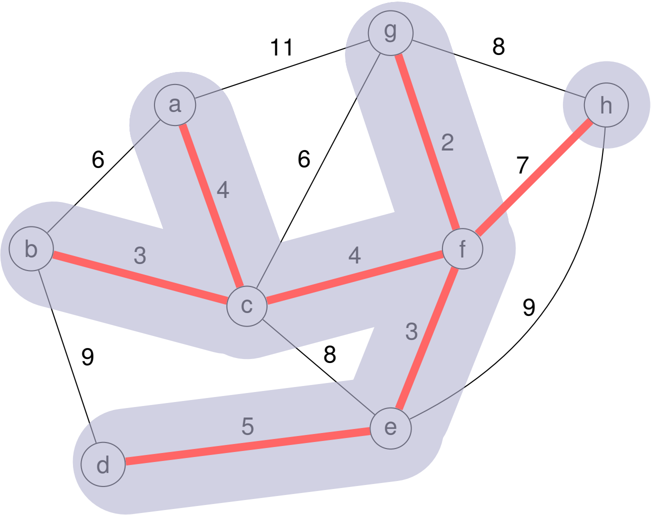
Execution of Kruskal's Algorithm



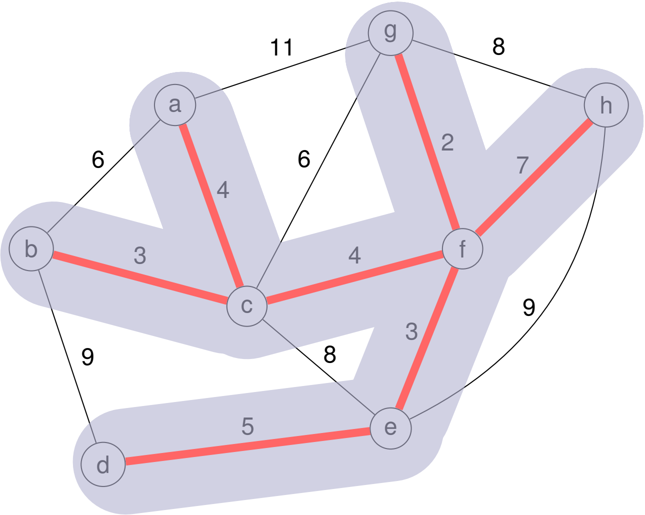
Execution of Kruskal's Algorithm



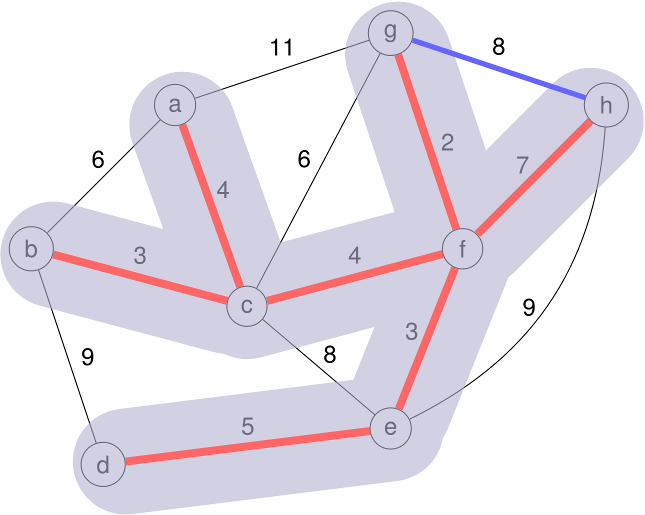
Execution of Kruskal's Algorithm



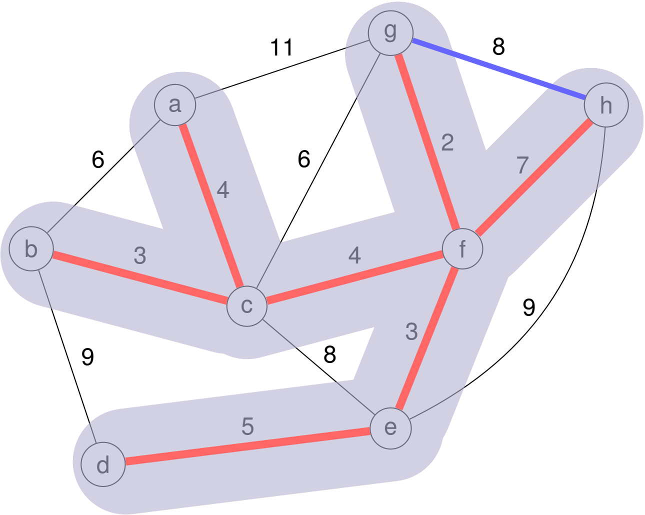
Execution of Kruskal's Algorithm



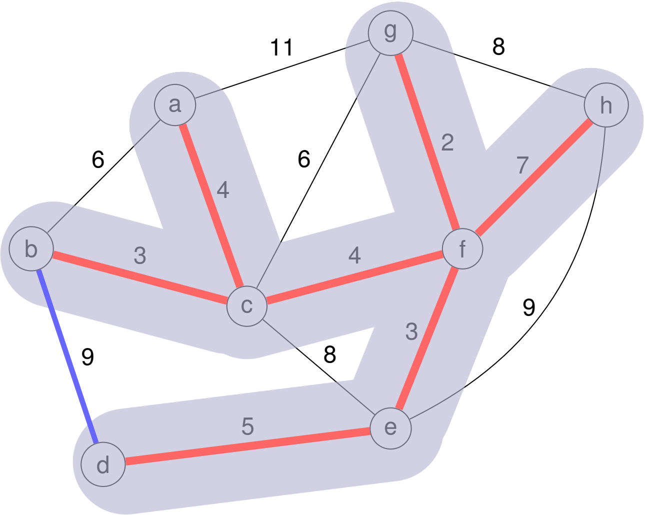
Execution of Kruskal's Algorithm



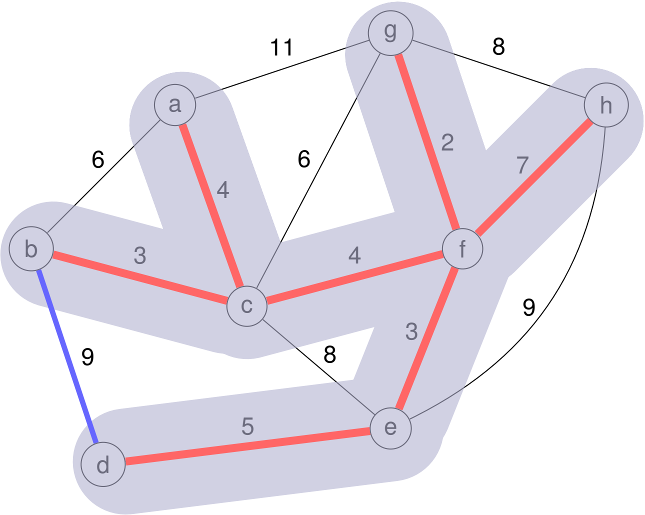
Execution of Kruskal's Algorithm



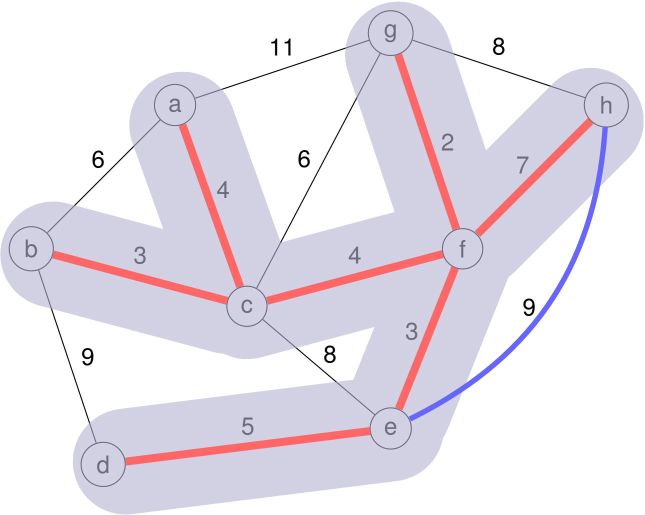
Execution of Kruskal's Algorithm



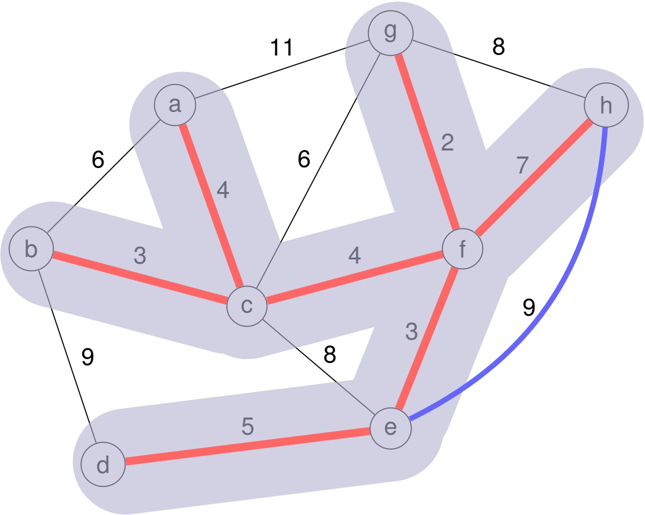
Execution of Kruskal's Algorithm



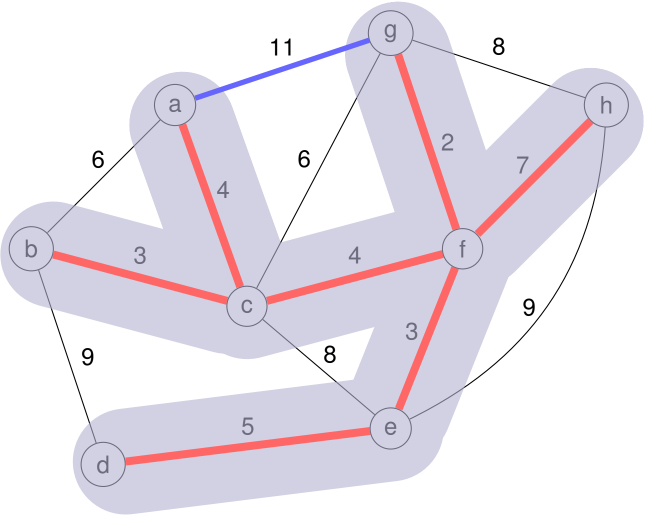
Execution of Kruskal's Algorithm



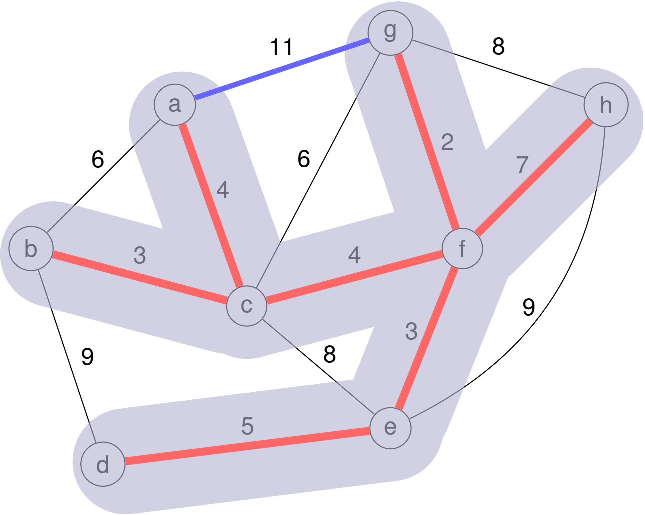
Execution of Kruskal's Algorithm



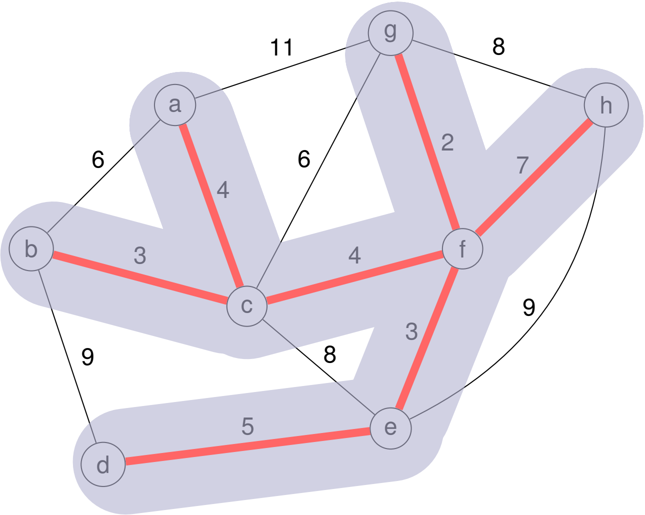
Execution of Kruskal's Algorithm



Execution of Kruskal's Algorithm



Execution of Kruskal's Algorithm



Details of Kruskal’s Algorithm

```
def kruskal(G)
    Apply Kruskal’s algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST; initially empty.
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

Details of Kruskal's Algorithm

```
def kruskal(G)
    Apply Kruskal's algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST; initially empty.
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

Time Complexity

Details of Kruskal's Algorithm

```
def kruskal(G)
    Apply Kruskal's algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST; initially empty.
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

Time Complexity

Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:      Apply Kruskal's algorithm to graph G
2:      Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

Time Complexity

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$

Details of Kruskal’s Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:     startSet = D.findSet(edge.start)
13:     endSet = D.findSet(edge.end)
14:     if startSet != endSet:
15:        A.append(edge)
16:        D.union(startSet,endSet)
17: return A
```

Time Complexity

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$

Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

Time Complexity

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$

Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

Time Complexity

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$

$\Rightarrow$ Overall: $\mathcal{O}(E \log E) = \mathcal{O}(E \log V)$

Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:      Apply Kruskal's algorithm to graph G
2:      Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:     startSet = D.findSet(edge.start)
13:     endSet = D.findSet(edge.end)
14:     if startSet != endSet:
15:        A.append(edge)
16:        D.union(startSet,endSet)
17: return A
```

Time Complexity

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$

$\Rightarrow$ Overall: $\mathcal{O}(E \log E) = \mathcal{O}(E \log V)$

If edges are already sorted, runtime becomes $O(E \cdot \alpha(n))$!

Details of Kruskal's Algorithm

```
def kruskal(G)
    Apply Kruskal's algorithm to graph G
    Return set of edges that form a MST
A = Set() # Set of edges of MST; initially empty.
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)
for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

Correctness

Details of Kruskal's Algorithm

```
def kruskal(G)
    Apply Kruskal's algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST; initially empty.
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

Correctness

- Consider the cut of all connected components (disjoint sets)

Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

Correctness

- Consider the cut of all connected components (disjoint sets)
- L. 14 ensures that we extend A by an edge that goes across the cut

Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

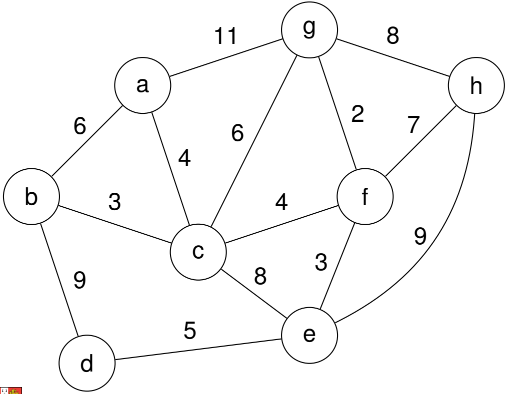
Correctness

- Consider the cut of all connected components (disjoint sets)
- L. 14 ensures that we extend A by an edge that goes across the cut
- This edge is also the lightest edge crossing the cut (otherwise, we would have included a lighter edge before)

Prim's Algorithm

Basic Strategy

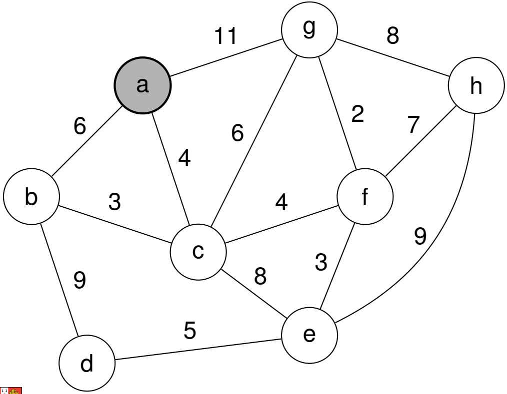
- Start growing a tree from a designated root vertex



Prim's Algorithm

Basic Strategy

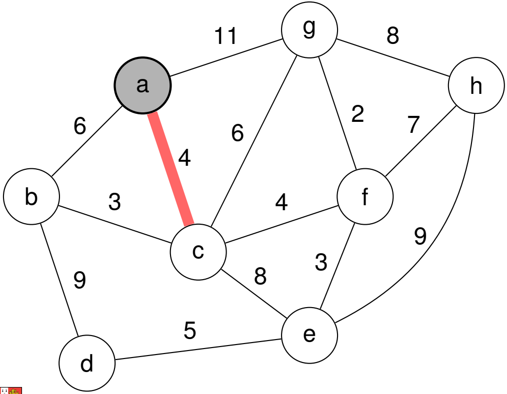
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm

Basic Strategy

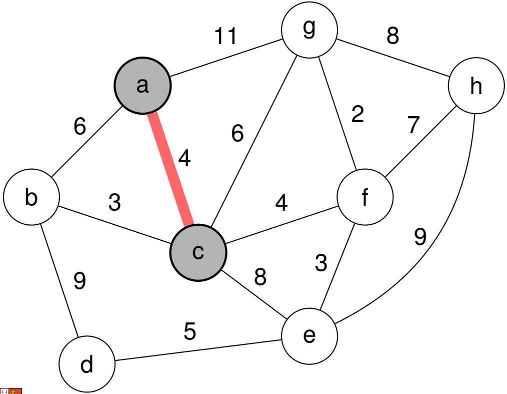
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

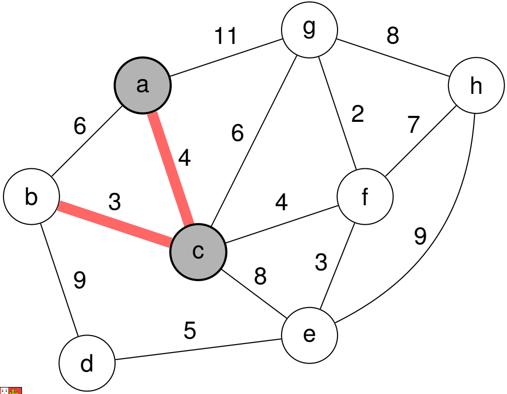
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

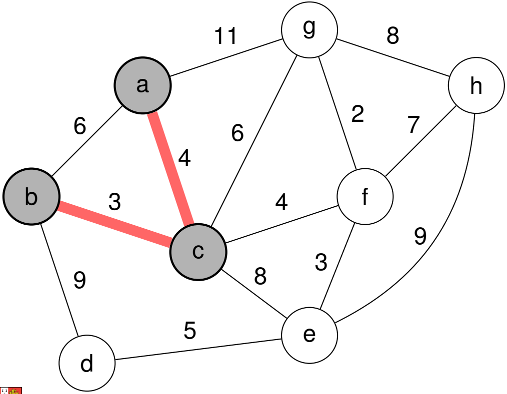
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

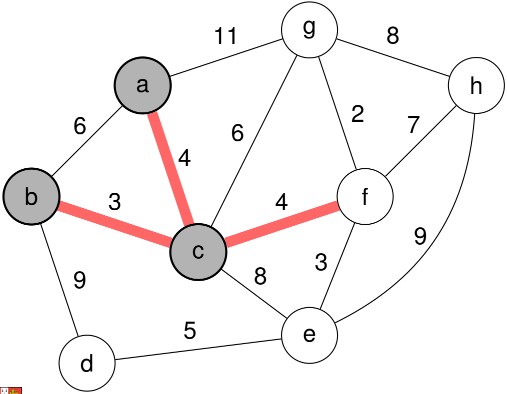
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

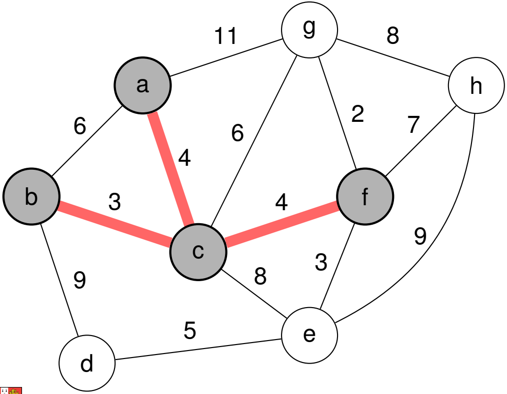
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm

Basic Strategy

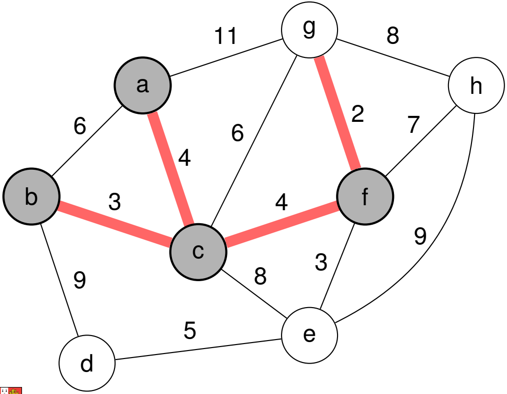
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

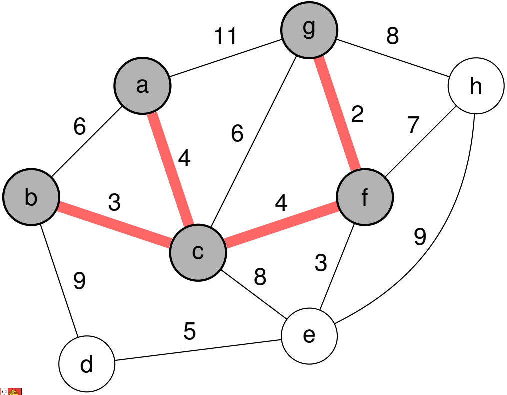
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

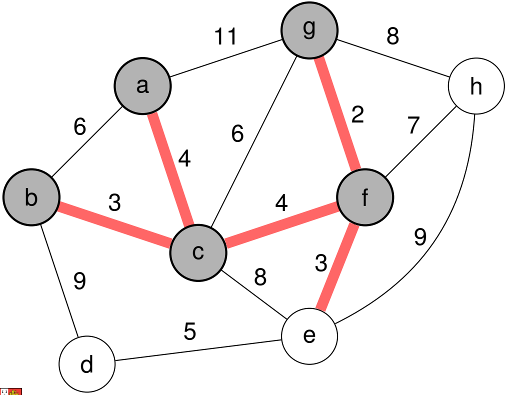
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

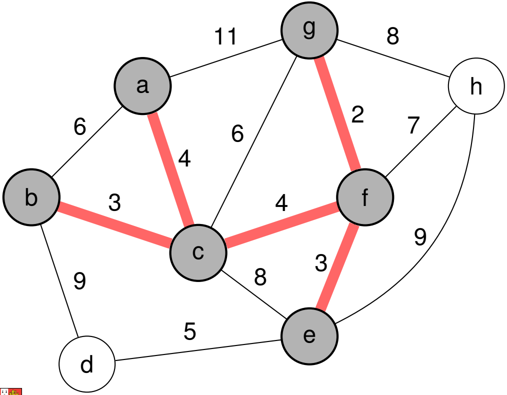
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm

Basic Strategy

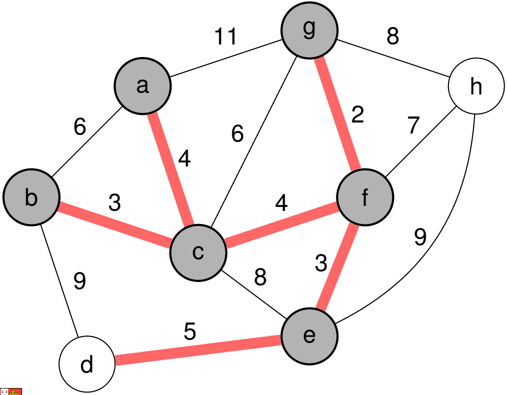
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm

Basic Strategy

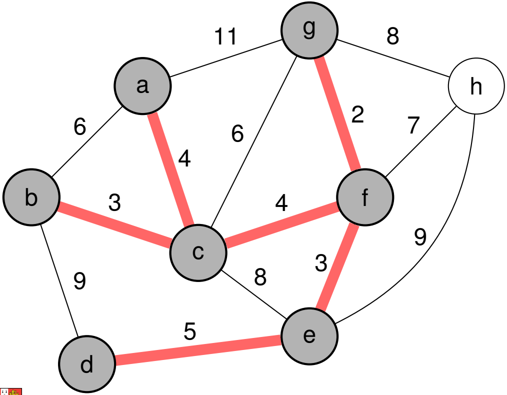
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

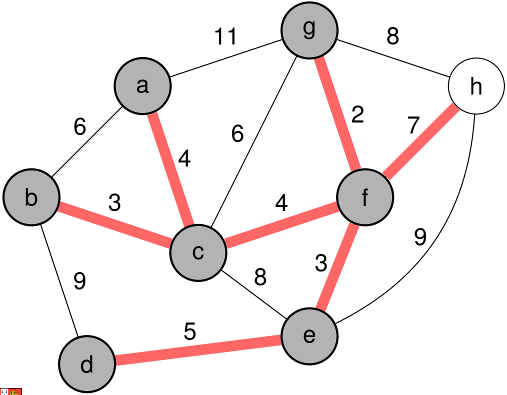
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm

Basic Strategy

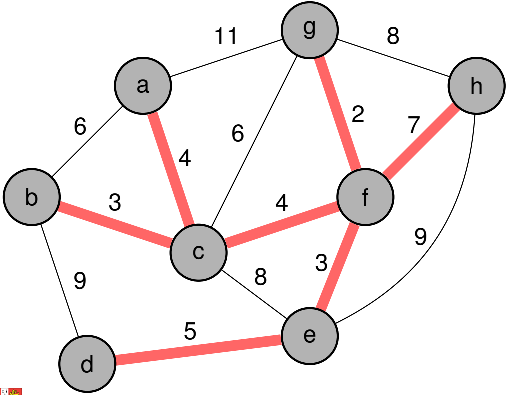
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle

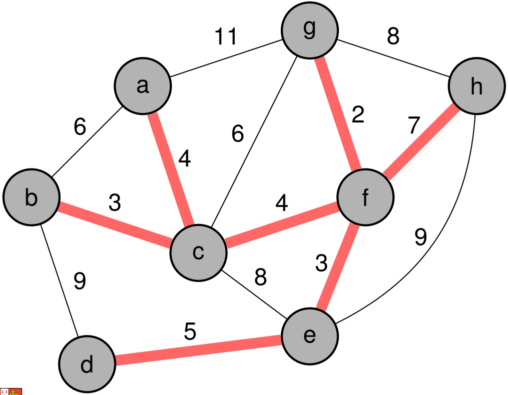


Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Implementation will be based on vertices!

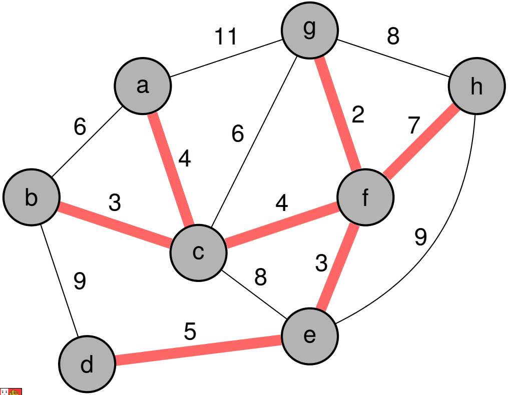


Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Assign every vertex not in A a key which is at all stages equal to the smallest weight of an edge connecting to A



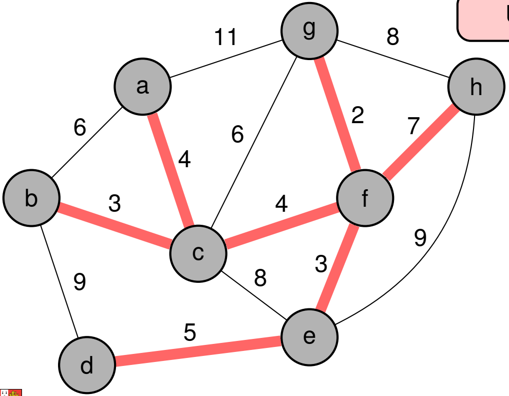
Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Assign every vertex not in A a key which is at all stages equal to the smallest weight of an edge connecting to A

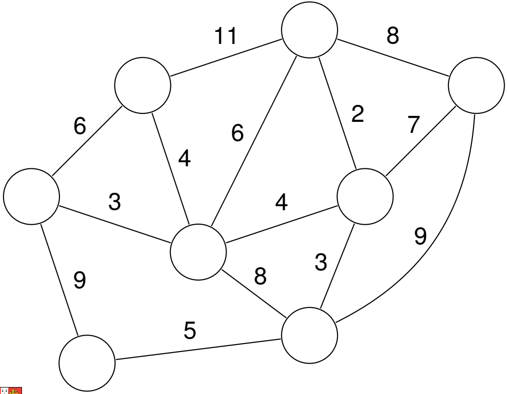
Use a Priority Queue!



Prim's Algorithm

Implementation

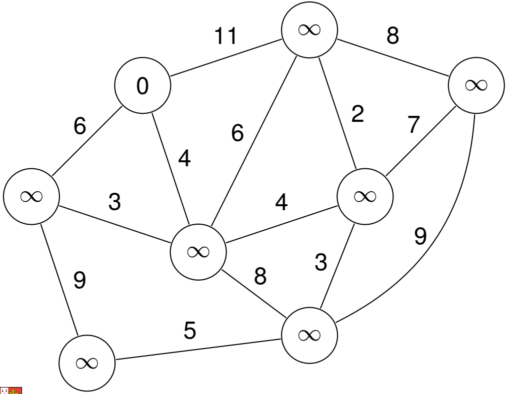
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

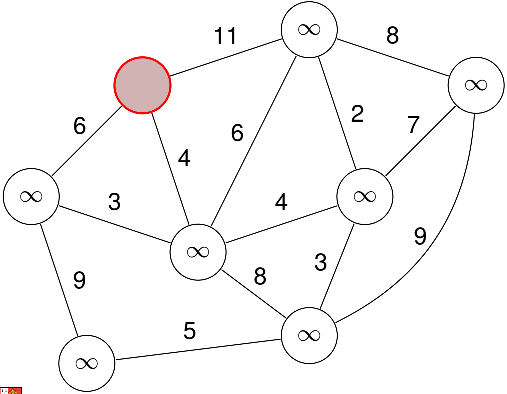
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

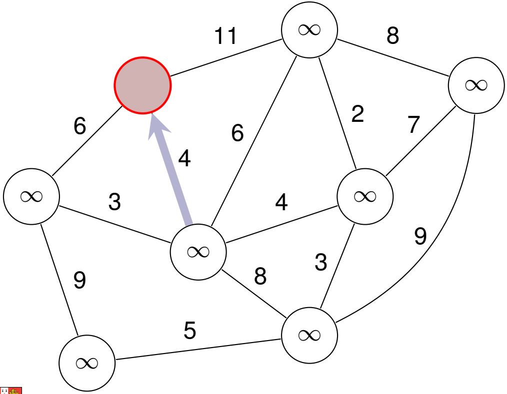
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

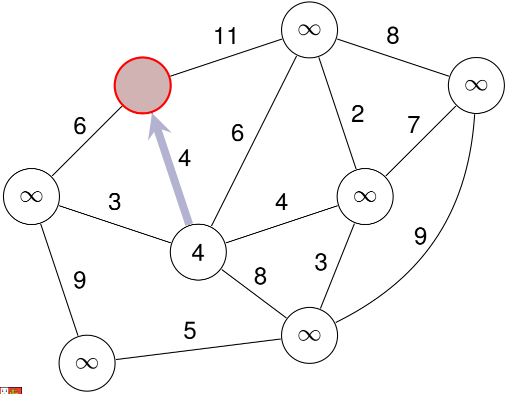
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

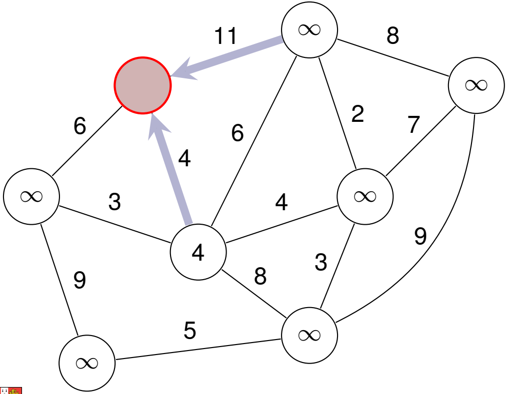
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

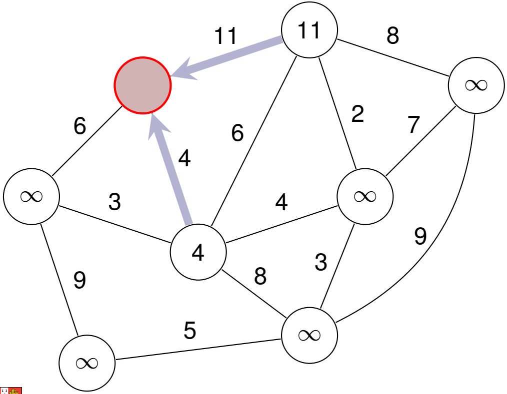
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

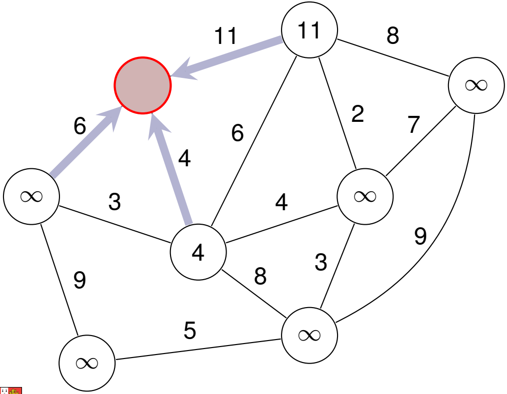
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

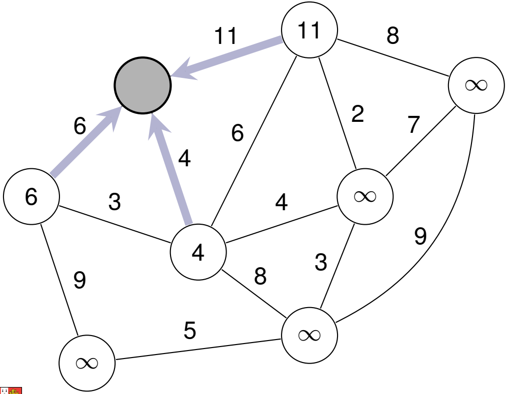
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

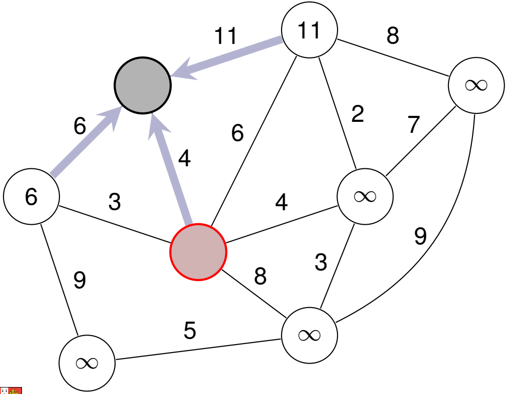
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

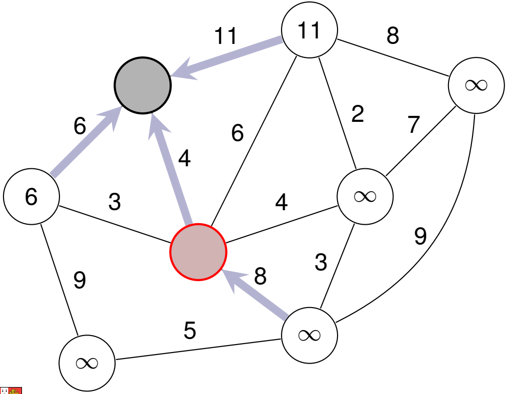
Implementation

- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Implementation

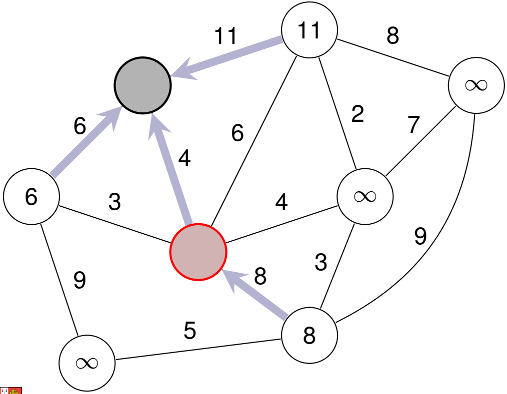
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

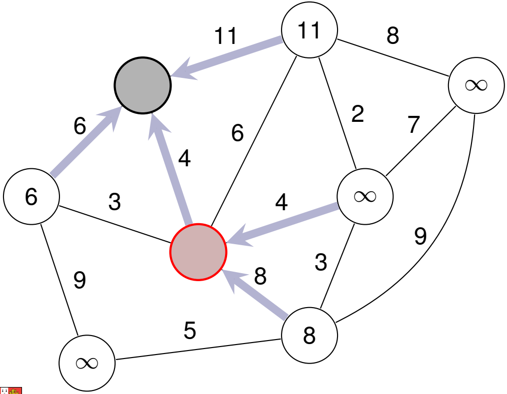
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

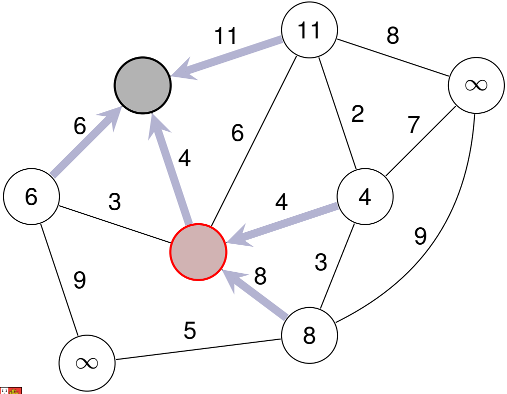
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

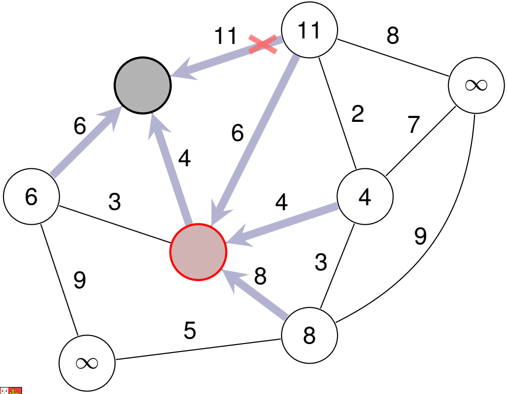
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

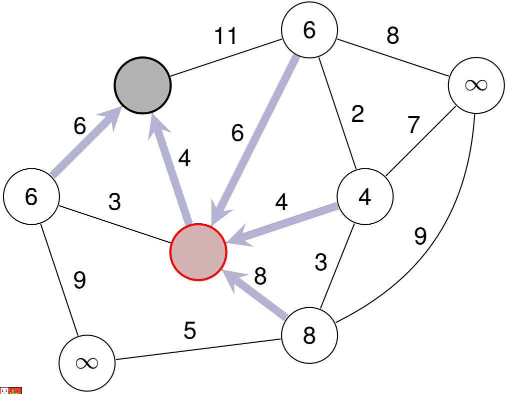
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

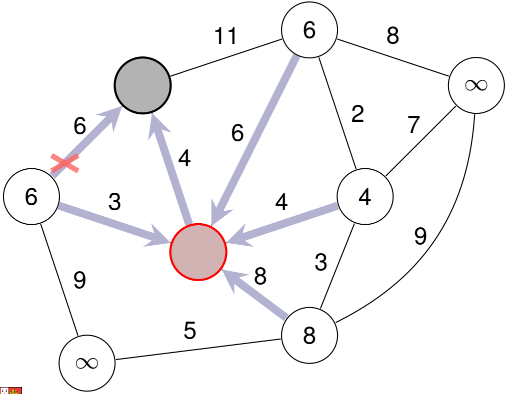
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

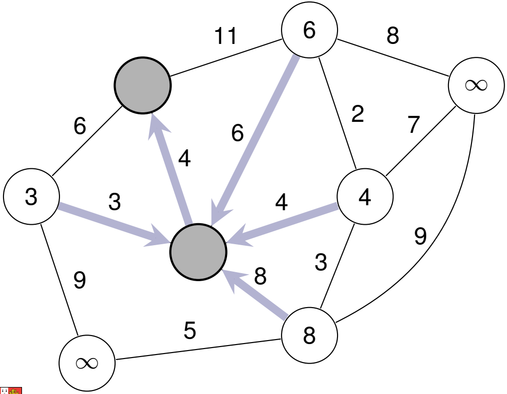
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

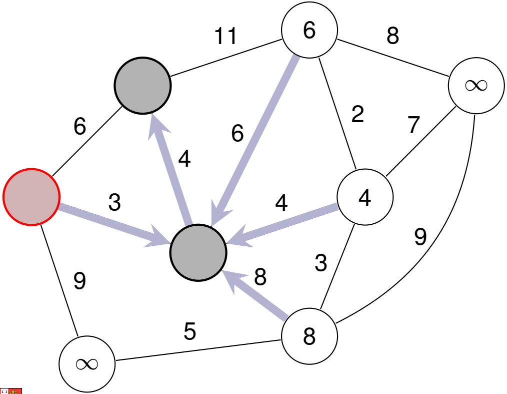
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

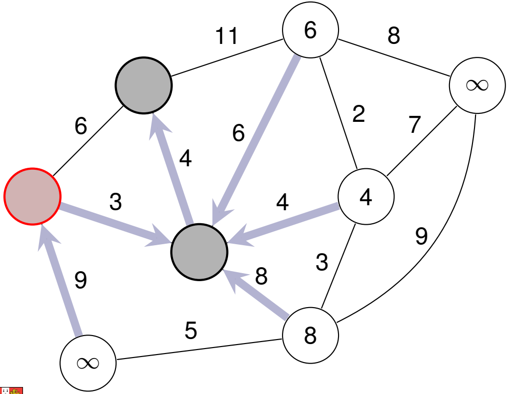
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

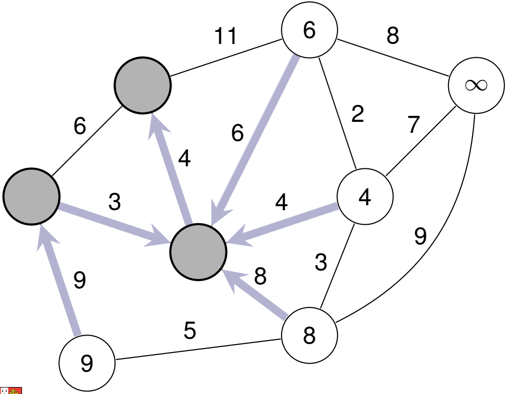
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

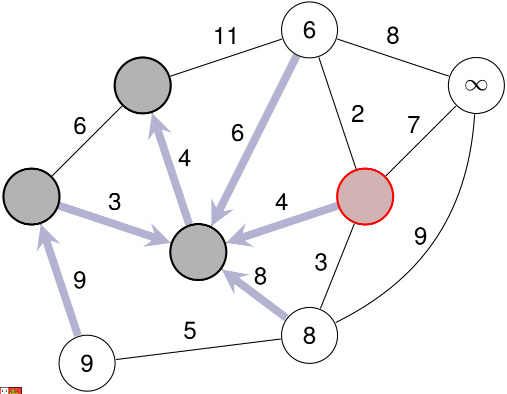
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

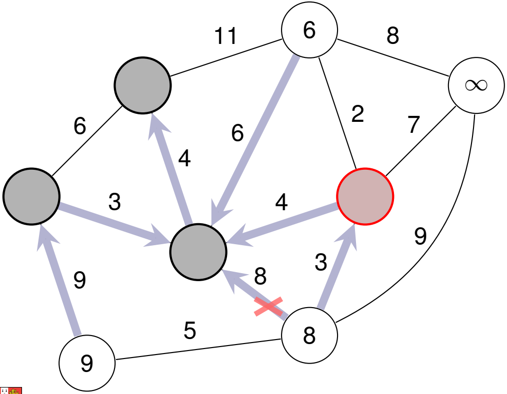
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

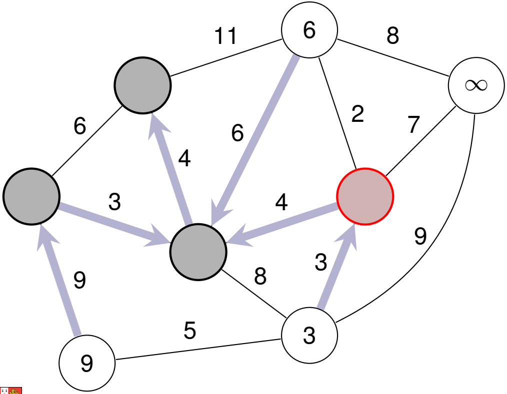
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

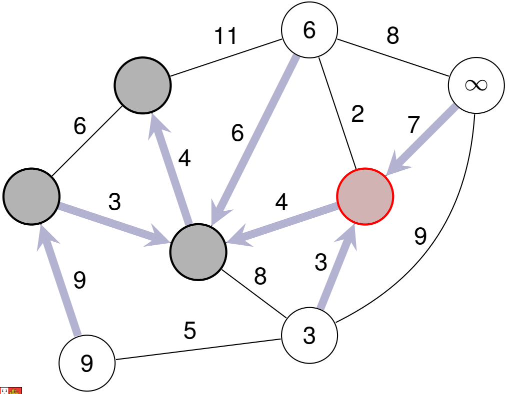
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

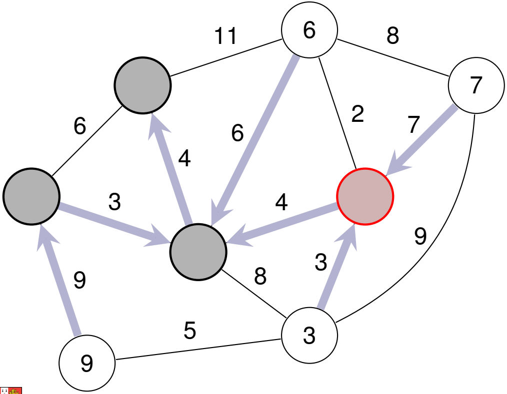
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

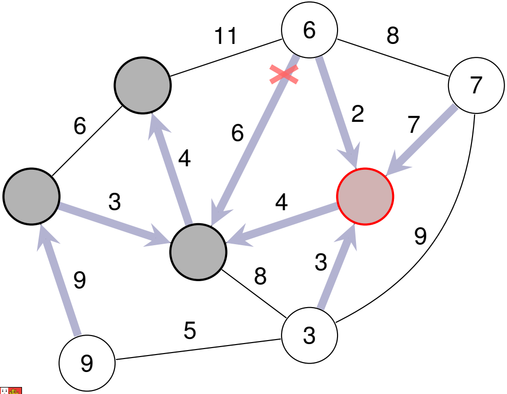
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

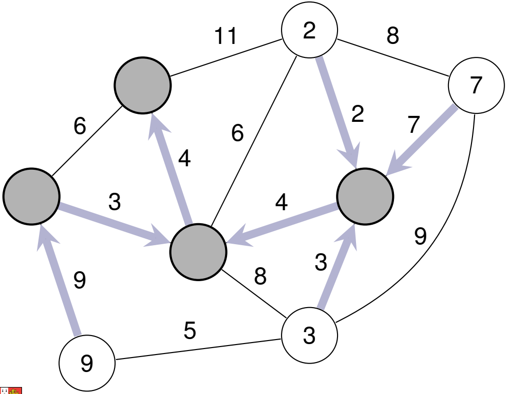
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

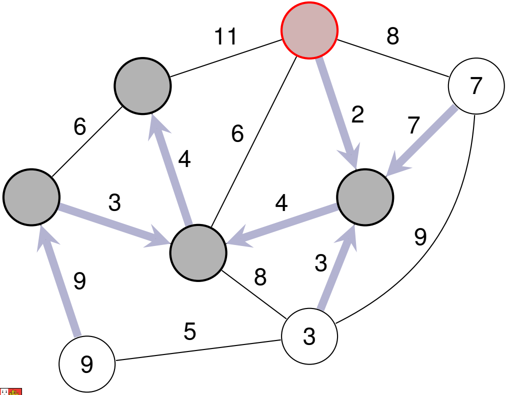
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

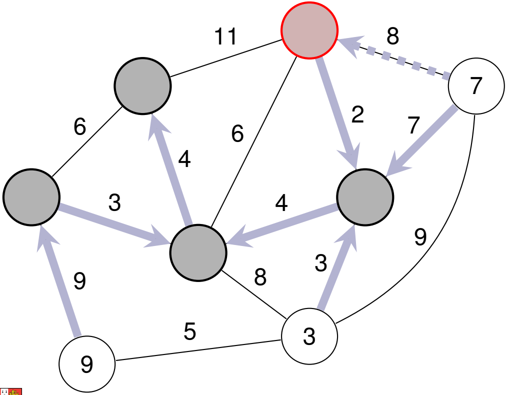
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

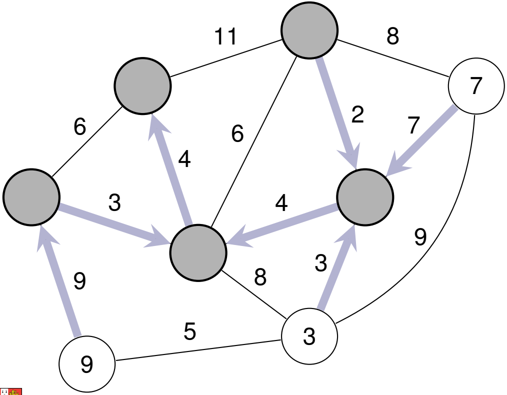
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

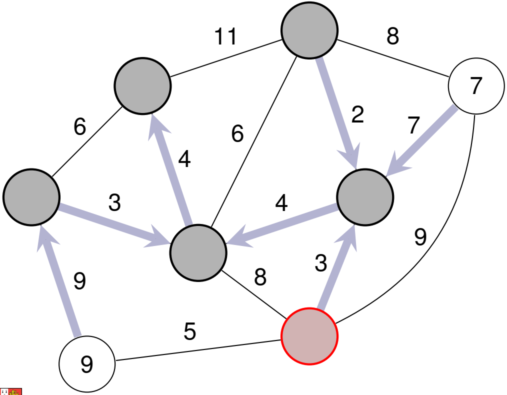
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

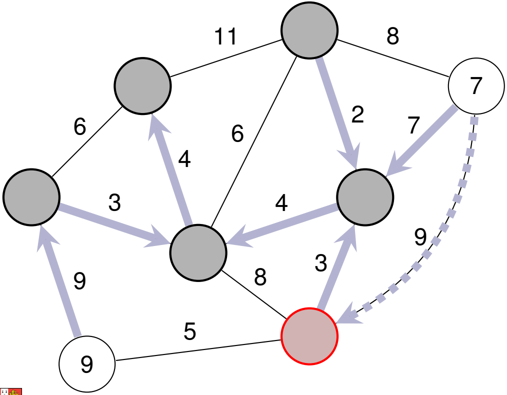
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

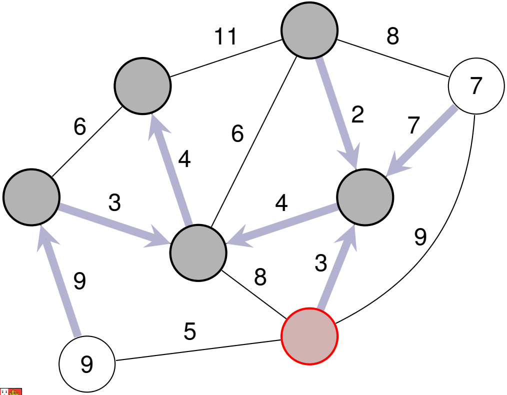
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

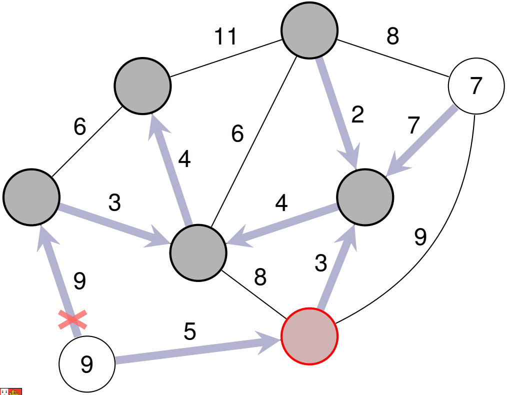
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

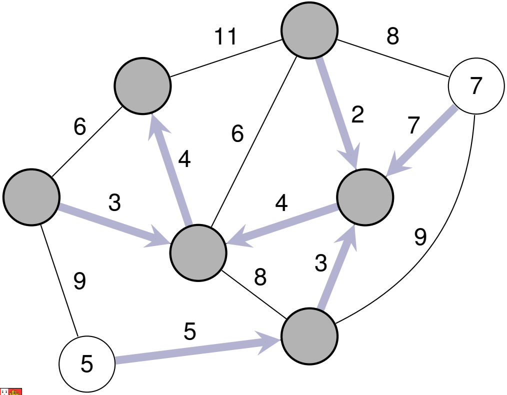
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

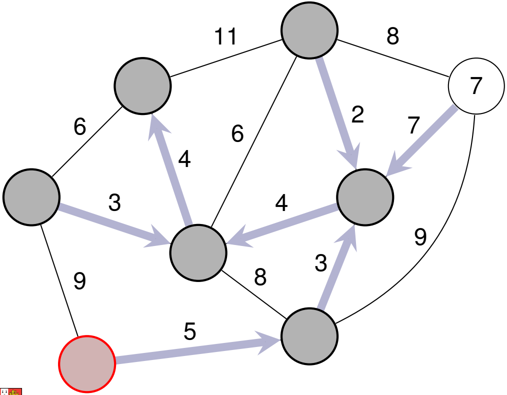
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

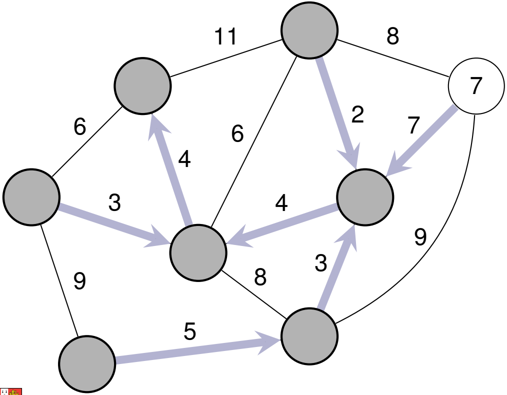
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

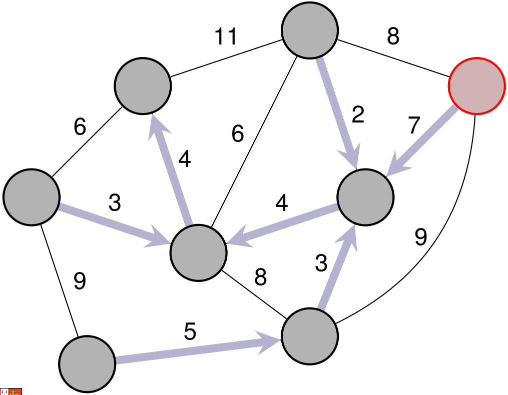
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

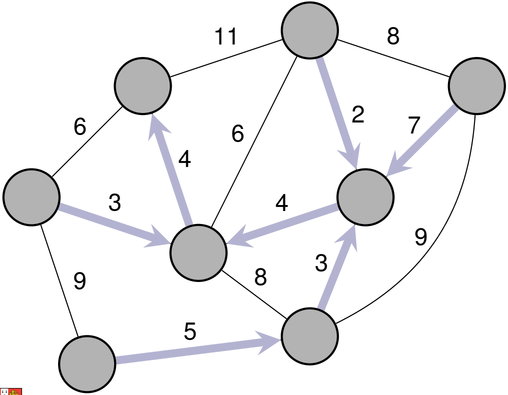
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

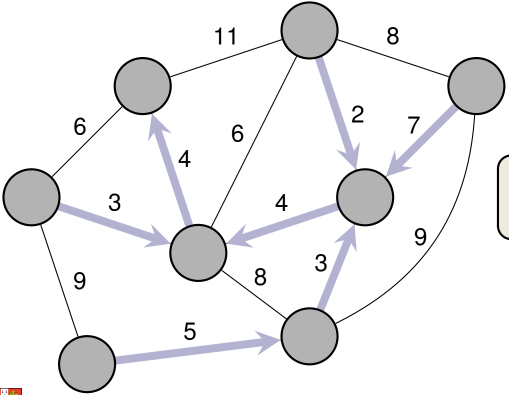
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Prim's Algorithm

Implementation

- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q

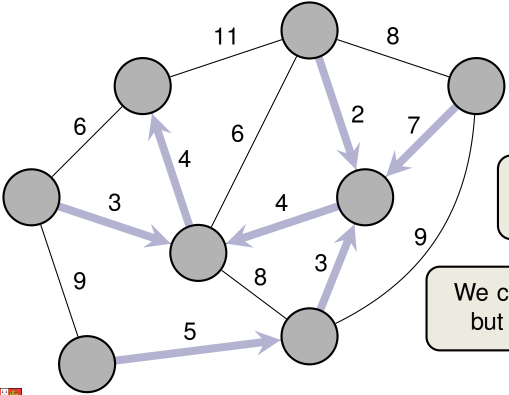


Final MST is given (implicitly) by the pointers!

Prim's Algorithm

Implementation

- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Final MST is given (implicitly) by the pointers!

We computed same MST as Kruskal, but in a completely different order!

Details of Prim's Algorithm

```
def prim(G,r)
    Apply Prim's Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
    Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u,v)
        if Q.hasItem(v) and w < v.key:
            v.predecessor = u
            Q.decreaseKey(item=v, newKey=w)
```

Details of Prim's Algorithm

```
def prim(G,r)
    Apply Prim's Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
    Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u,v)
        if Q.hasItem(v) and w < v.key:
            v.predecessor = u
            Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

Details of Prim's Algorithm

```
def prim(G,r)
    Apply Prim's Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
    Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u,v)
        if Q.hasItem(v) and w < v.key:
            v.predecessor = u
            Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:            v.predecessor = u
20:            Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$,

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$,

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$,

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$,

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:            v.predecessor = u
20:            Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$

  Amortized Cost   Amortized Cost

Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$
  $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$

Details of Prim's Algorithm

```
0:  def prim(G,r)
1:      Apply Prim's Algorithm to graph G and root r
2:      Return result implicitly by modifying G:
3:      MST induced by the .predecessor fields
4:
5:  Q = MinPriorityQueue()
6:  for v in G.vertices():
7:      v.predecessor = None
8:      if v == r:
9:          v.key = 0
10:     else:
11:         v.key = Infinity
12:     Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$
  $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$
- Binary/Binomial Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot \log V)$
  $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E \log V)$

Summary (Kruskal and Prim)

Generic Idea

- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting A

Summary (Kruskal and Prim)

Generic Idea

- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting A

Kruskal's Algorithm

- Gradually transforms a forest into a MST by merging trees
- invokes disjoint set data structure
- Runtime $\mathcal{O}(E \log V)$

Summary (Kruskal and Prim)

Generic Idea

- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting A

Kruskal's Algorithm

- Gradually transforms a forest into a MST by merging trees
- invokes disjoint set data structure
- Runtime $\mathcal{O}(E \log V)$

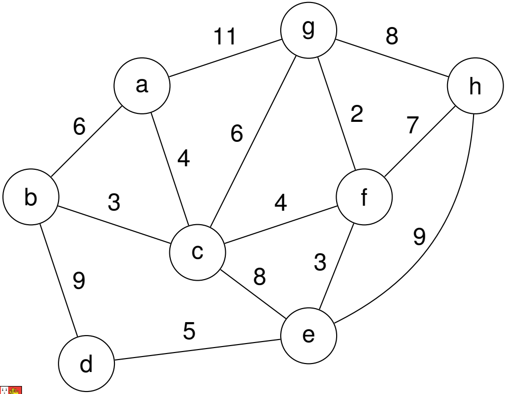
Prim's Algorithm

- Gradually extends a tree into a MST by adding incident edges
- invokes Fibonacci heaps (priority queue)
- Runtime $\mathcal{O}(V \log V + E)$

Outlook: Reverse-Delete Algorithm

Basic Idea

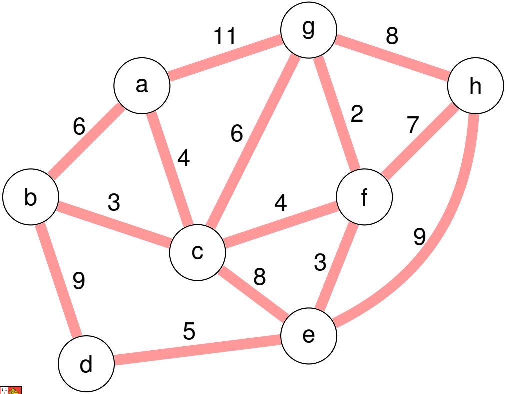
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

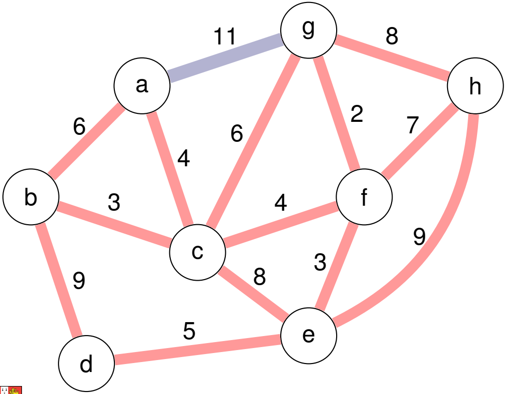
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

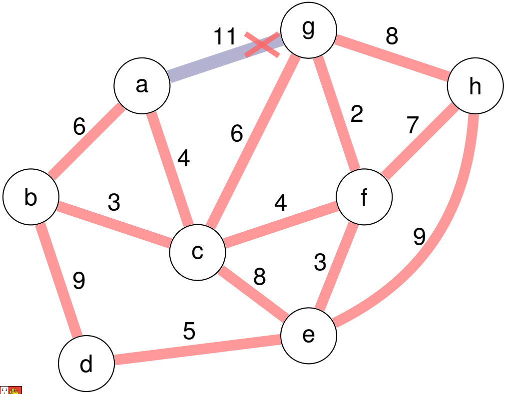
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

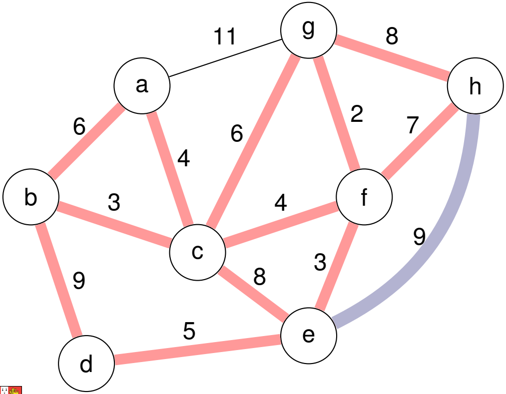
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

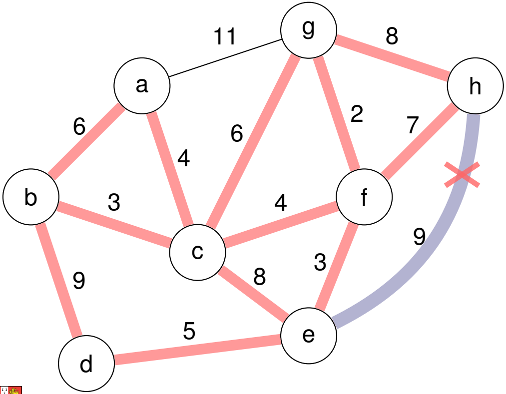
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

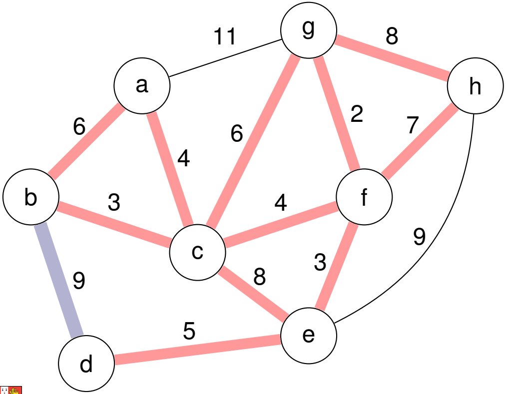
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

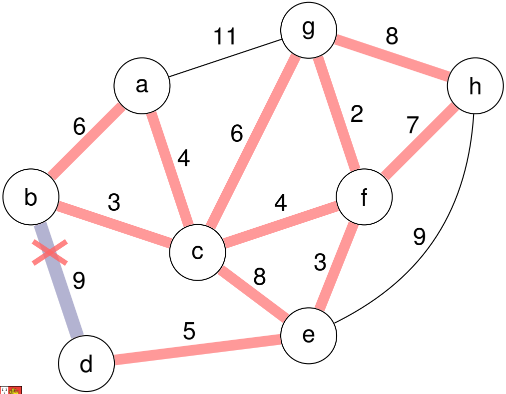
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

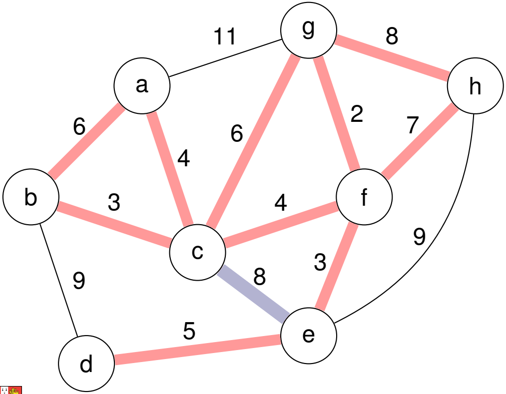
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

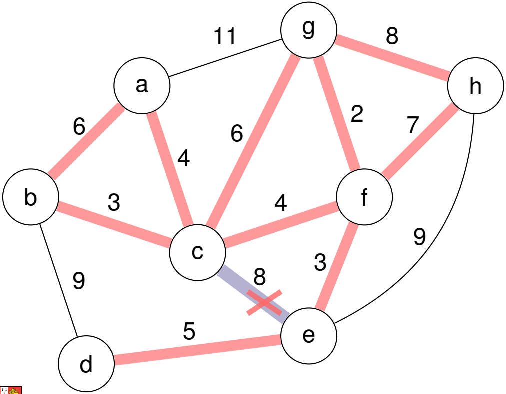
- Let *A* be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from *A* as long as all vertices are connected by *A*



Outlook: Reverse-Delete Algorithm

Basic Idea

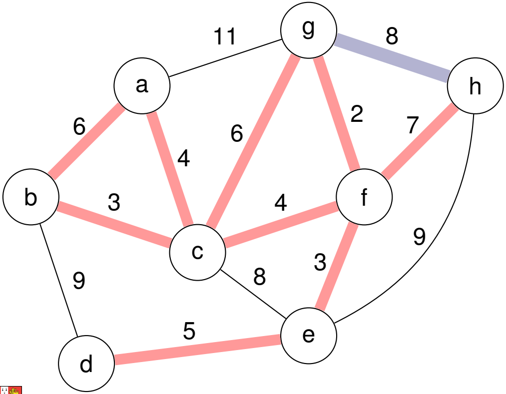
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

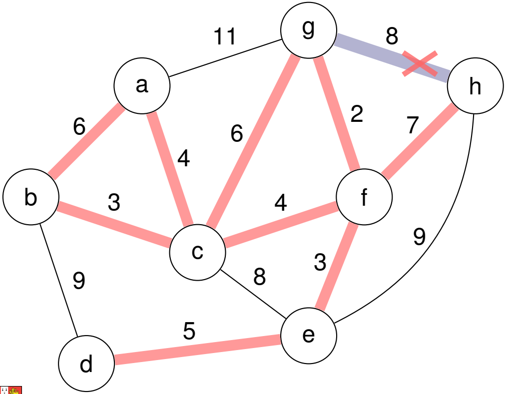
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

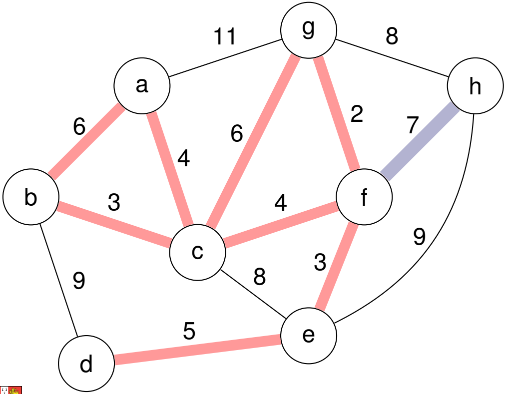
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

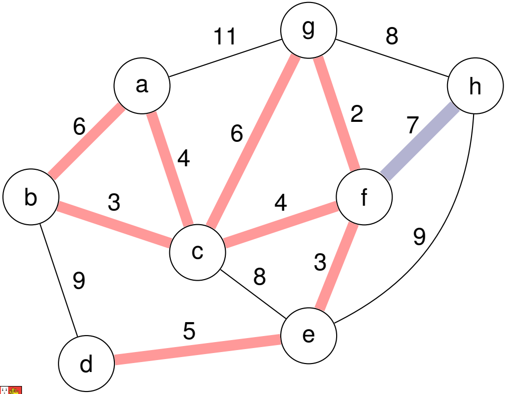
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

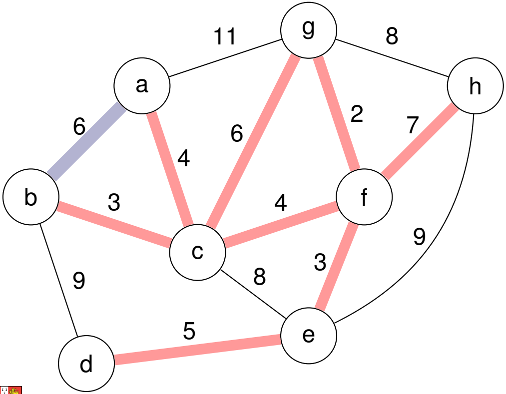
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

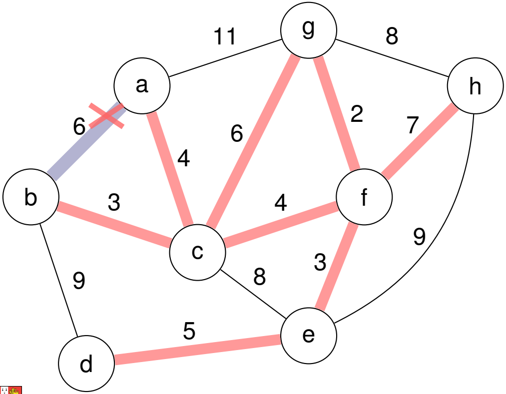
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

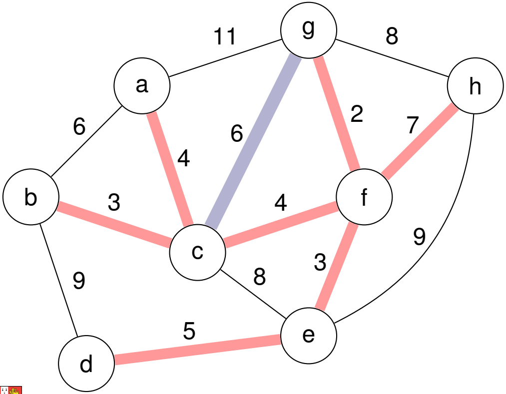
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

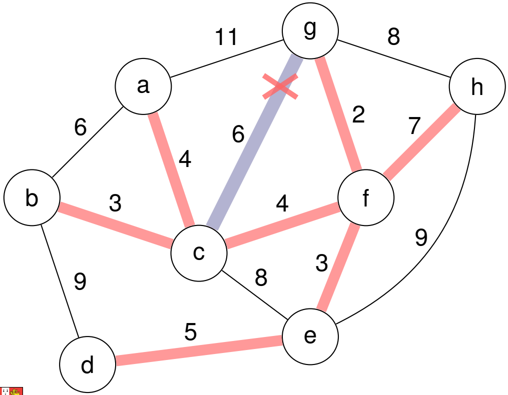
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

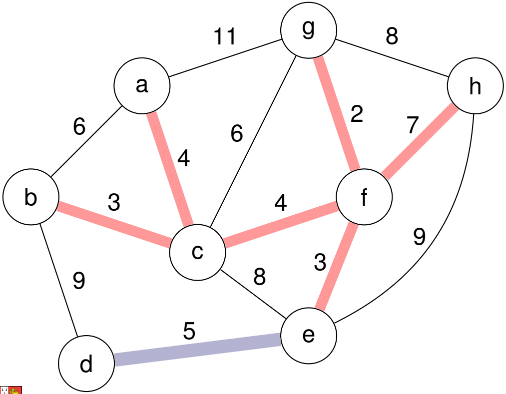
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

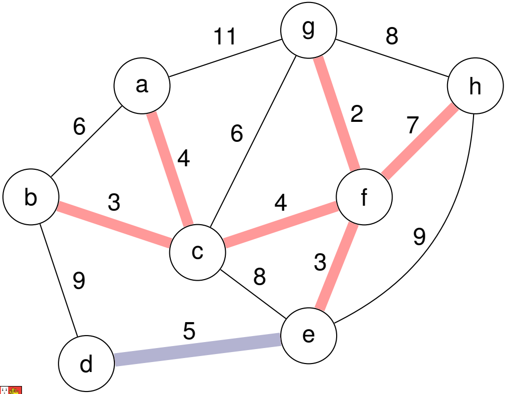
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

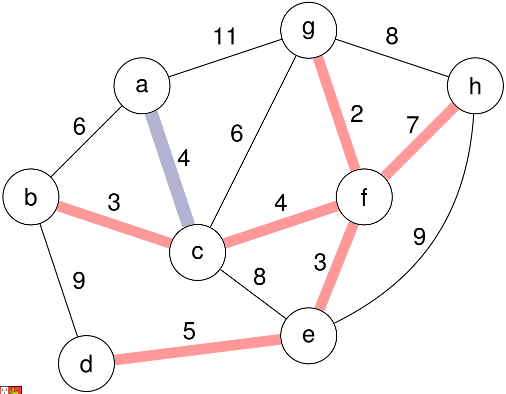
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

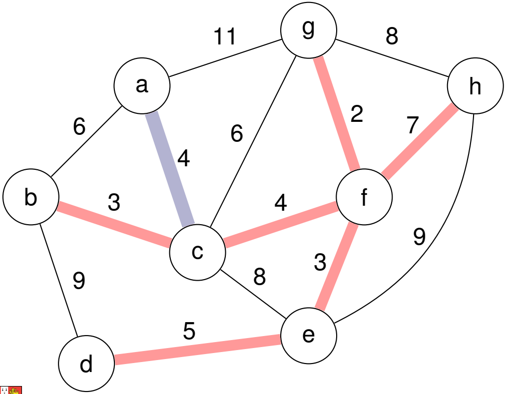
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

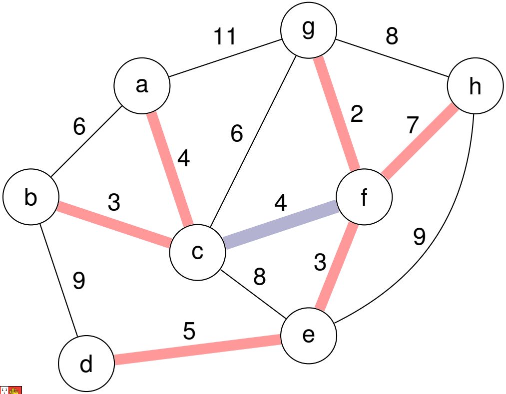
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

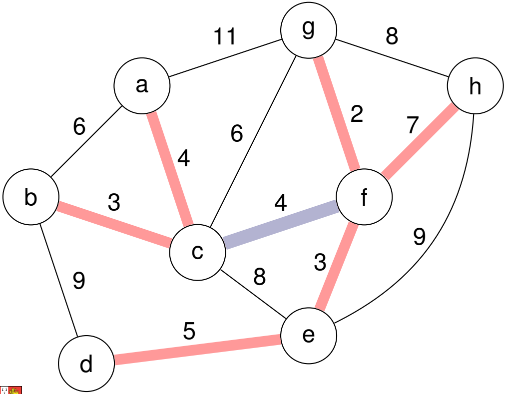
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

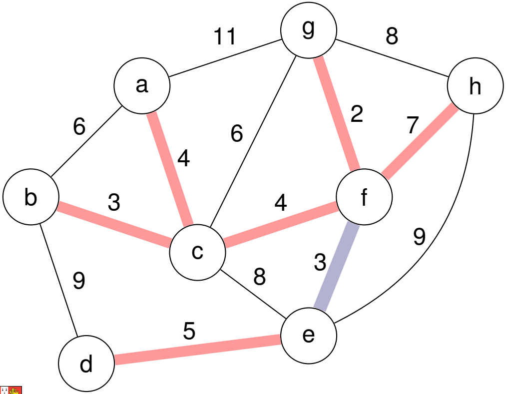
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

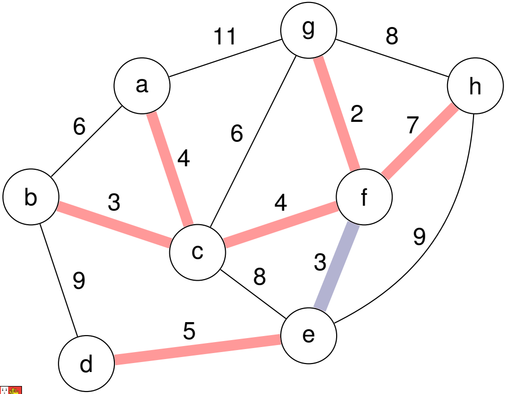
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

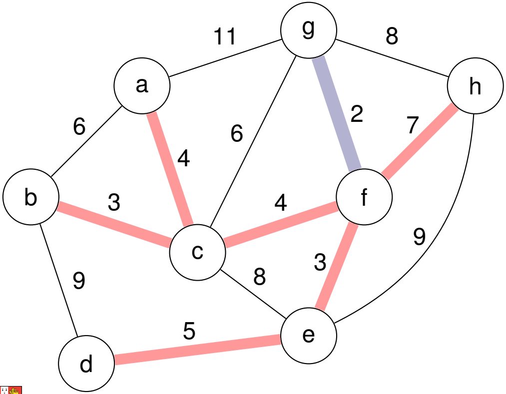
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

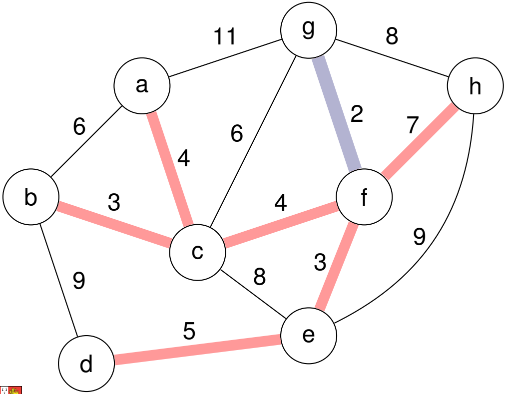
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

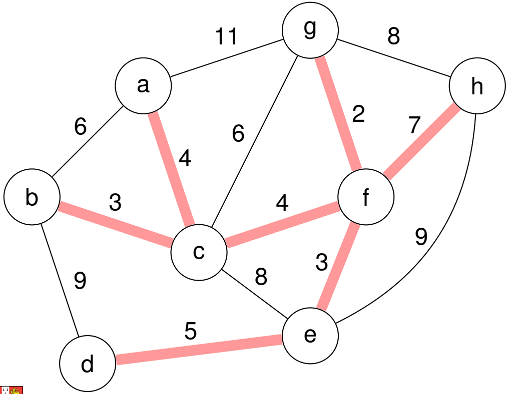
- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A



Outlook: Reverse-Delete Algorithm

Basic Idea

- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A

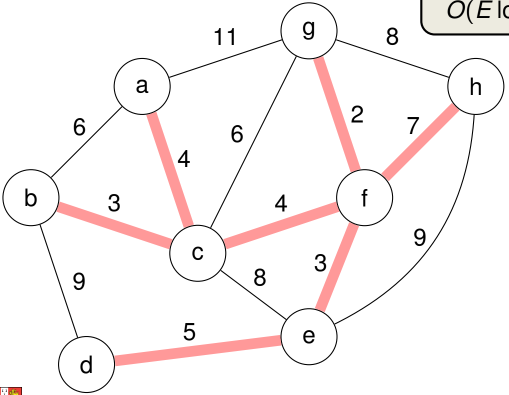


Outlook: Reverse-Delete Algorithm

Basic Idea

- Let A be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from A as long as all vertices are connected by A

Can be implemented in time $O(E \log V (\log\log V)^3)$. [Thorup, 2000]



Current State-of-the-Art

Does a linear-time MST algorithm exist?

Current State-of-the-Art

Does a linear-time MST algorithm exist?

Karger, Klein, Tarjan, JACM'1995

- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

Current State-of-the-Art

Does a linear-time MST algorithm exist?

Karger, Klein, Tarjan, JACM'1995

- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

Chazelle, JACM'2000

- deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$

Current State-of-the-Art

Does a linear-time MST algorithm exist?

Karger, Klein, Tarjan, JACM'1995

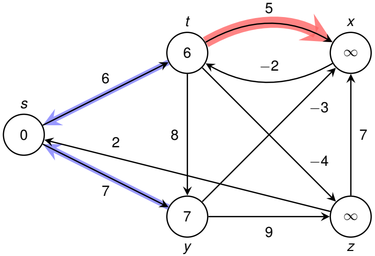
- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

Chazelle, JACM'2000

- deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$

Pettie, Ramachandran, JACM'2002

- deterministic MST algorithm with asymptotically optimal runtime
- however, the runtime itself is not known...



6.4: Single-Source Shortest Paths

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

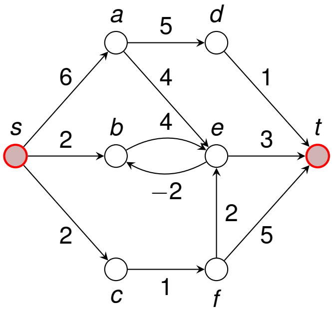
Introduction

Bellman-Ford Algorithm

Shortest Path Problem

Shortest Path Problem

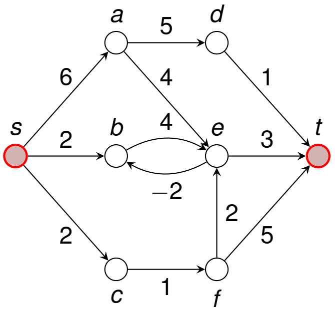
- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$



Shortest Path Problem

Shortest Path Problem

- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G

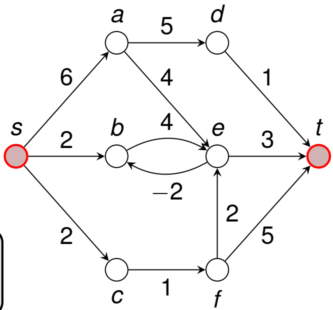


Shortest Path Problem

Shortest Path Problem

- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

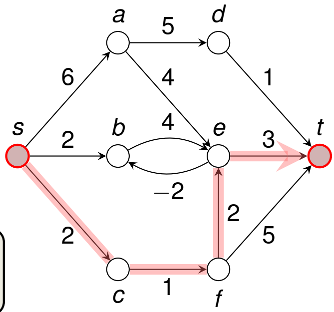


Shortest Path Problem

Shortest Path Problem

- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G

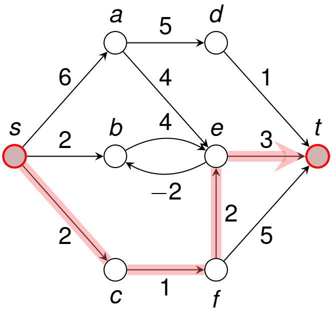
$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.



Shortest Path Problem

Shortest Path Problem

- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G

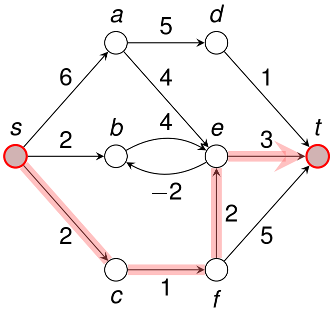


What if G is **unweighted**?

Shortest Path Problem

Shortest Path Problem

- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G



What if G is **unweighted**?

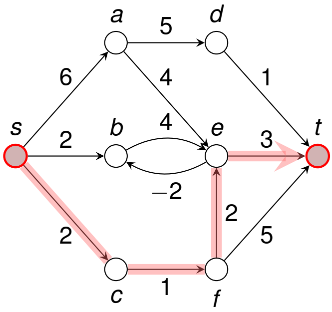
Two possible answers are:

1. Run BFS (computes shortest paths in unweighted graphs)
2. Set the weight of all edges to 1

Shortest Path Problem

Shortest Path Problem

- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G



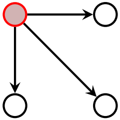
Applications

- Car Navigation, Internet Routing, Arbitrage in Concurrency Exchange

Variants of Shortest Path Problems

Single-source shortest-paths problem (SSSP)

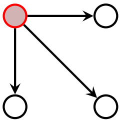
- Bellman-Ford Algorithm
- Dijsktra Algorithm



Variants of Shortest Path Problems

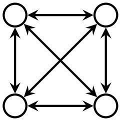
Single-source shortest-paths problem (SSSP)

- Bellman-Ford Algorithm
- Dijsktra Algorithm

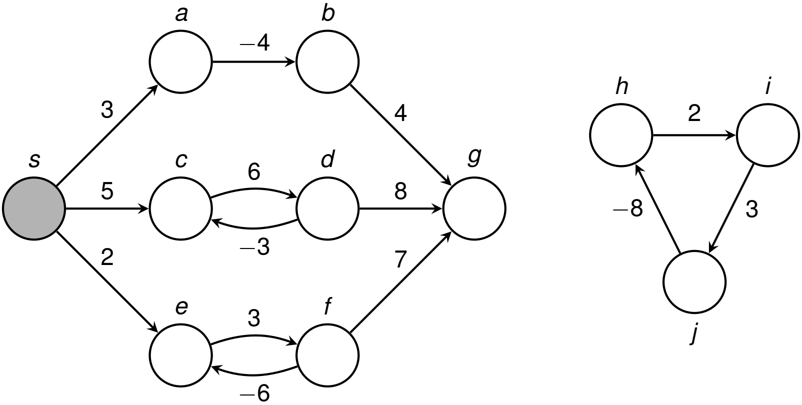


All-pairs shortest-paths problem (APSP)

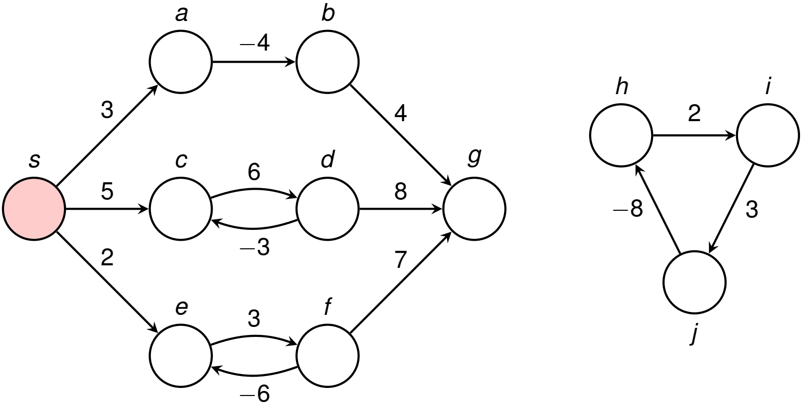
- Shortest Paths via Matrix Multiplication
- Johnson's Algorithm



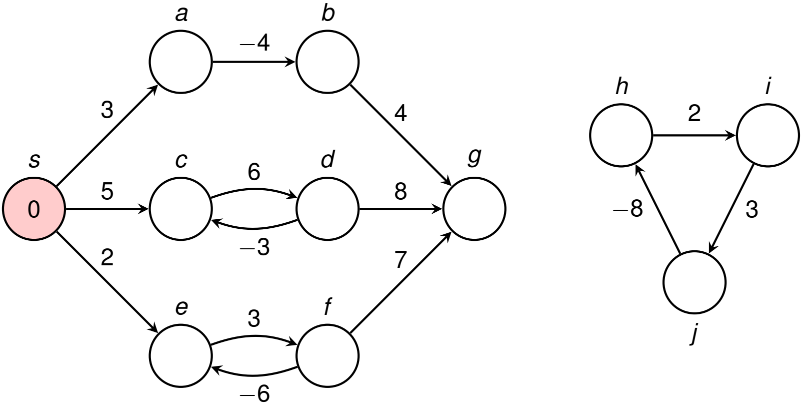
Distances and Negative-Weight Cycles (Figure 24.1)



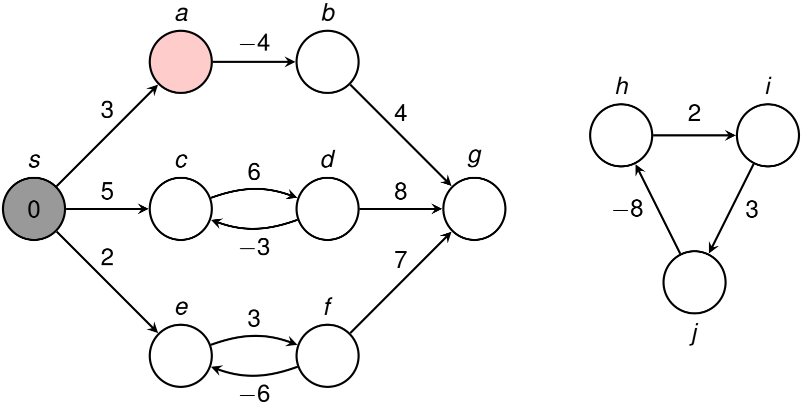
Distances and Negative-Weight Cycles (Figure 24.1)



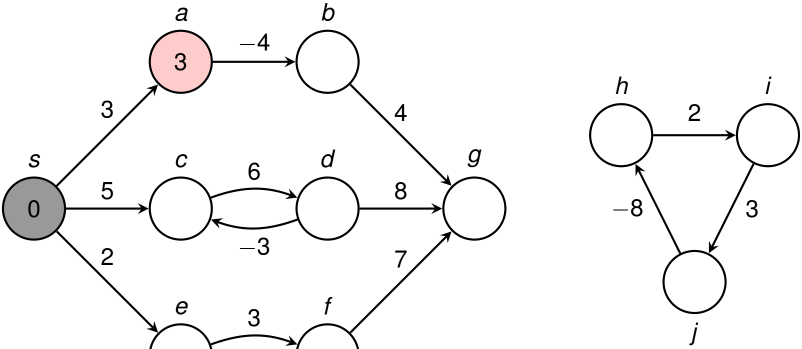
Distances and Negative-Weight Cycles (Figure 24.1)



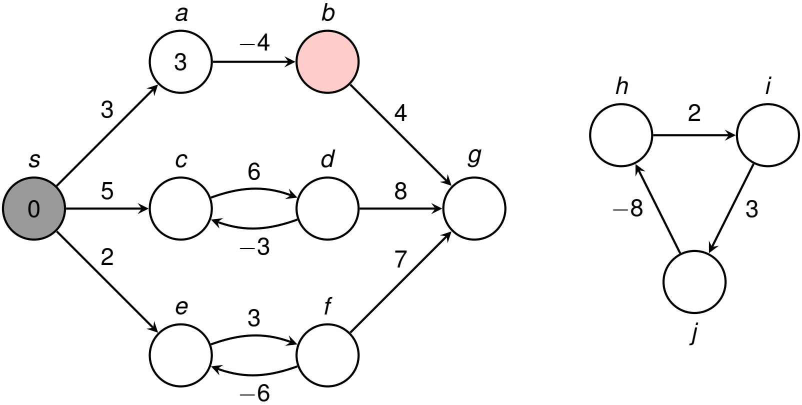
Distances and Negative-Weight Cycles (Figure 24.1)



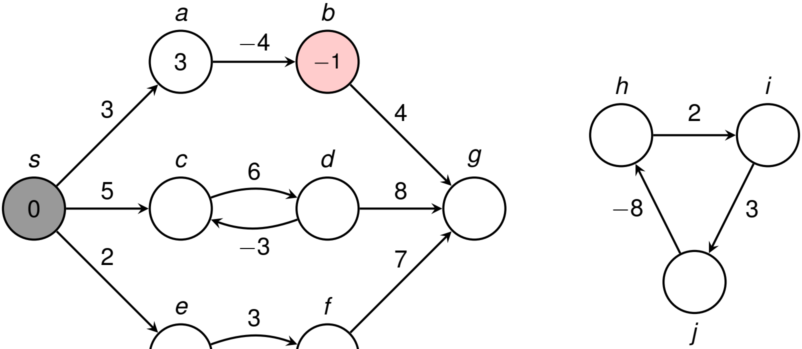
Distances and Negative-Weight Cycles (Figure 24.1)



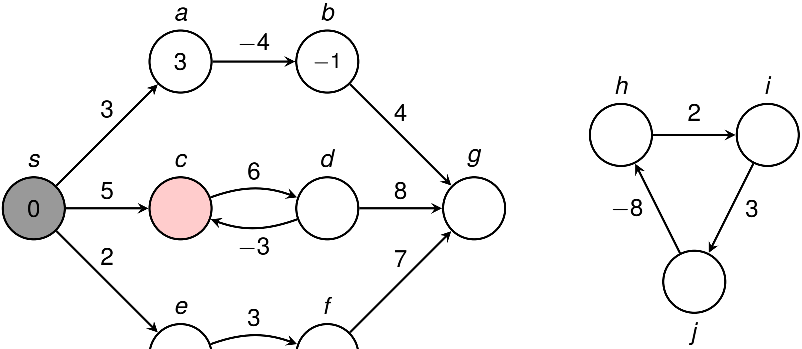
Distances and Negative-Weight Cycles (Figure 24.1)



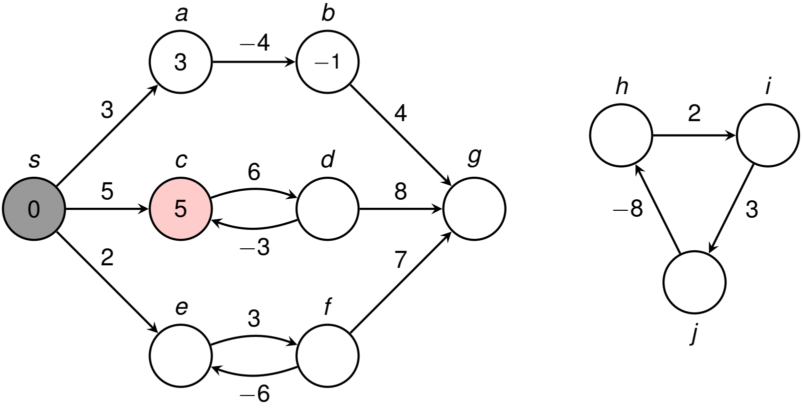
Distances and Negative-Weight Cycles (Figure 24.1)



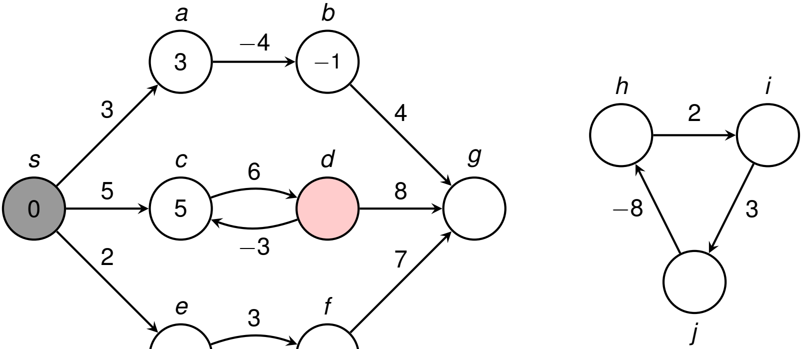
Distances and Negative-Weight Cycles (Figure 24.1)



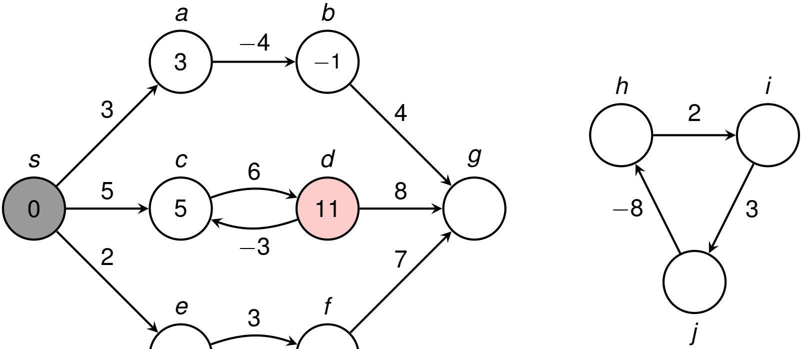
Distances and Negative-Weight Cycles (Figure 24.1)



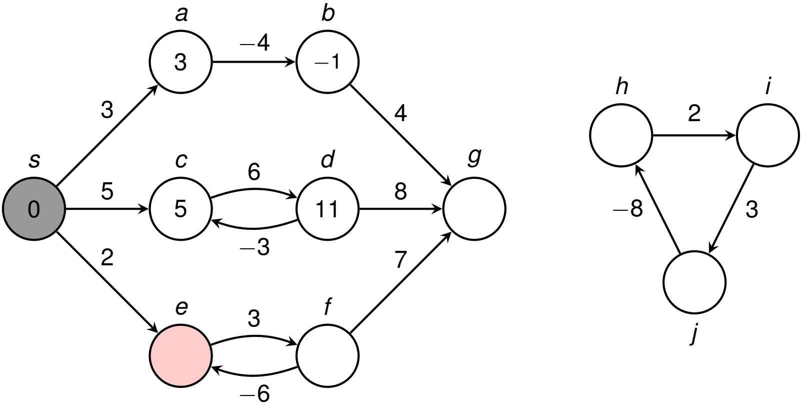
Distances and Negative-Weight Cycles (Figure 24.1)



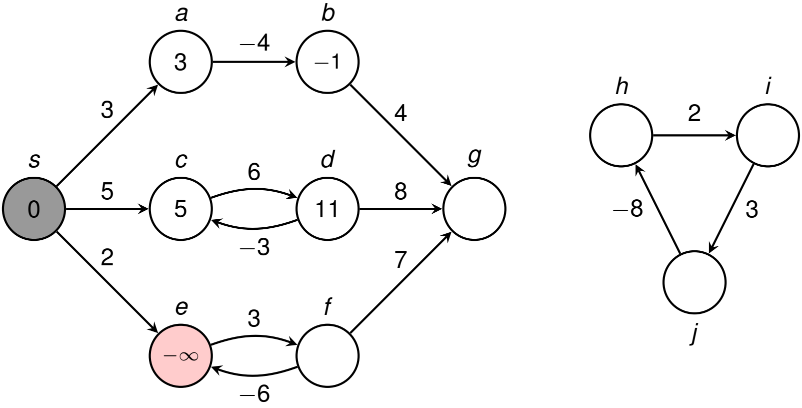
Distances and Negative-Weight Cycles (Figure 24.1)



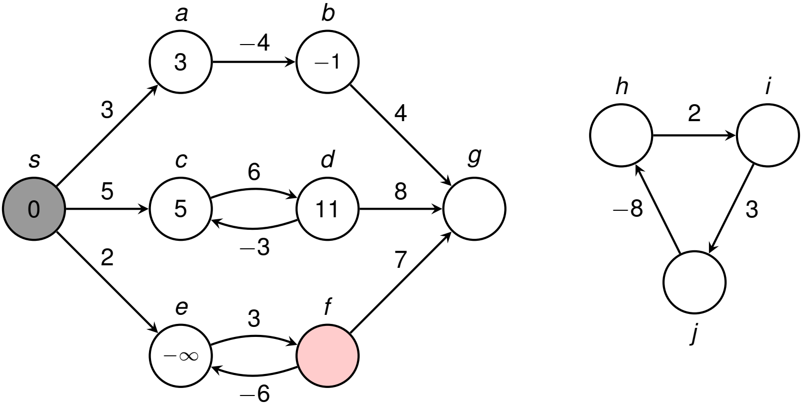
Distances and Negative-Weight Cycles (Figure 24.1)



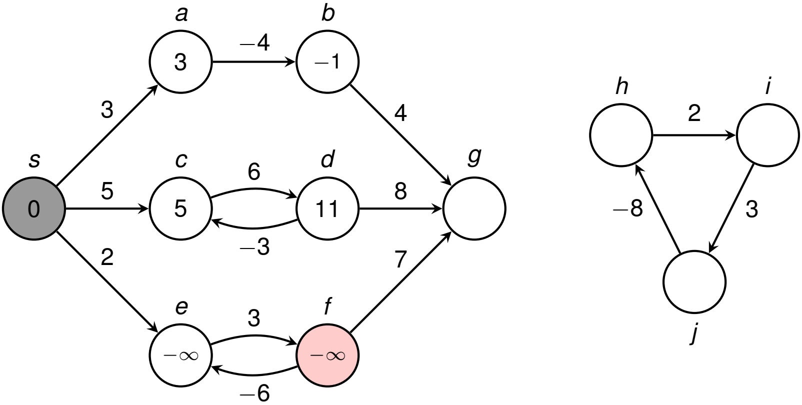
Distances and Negative-Weight Cycles (Figure 24.1)



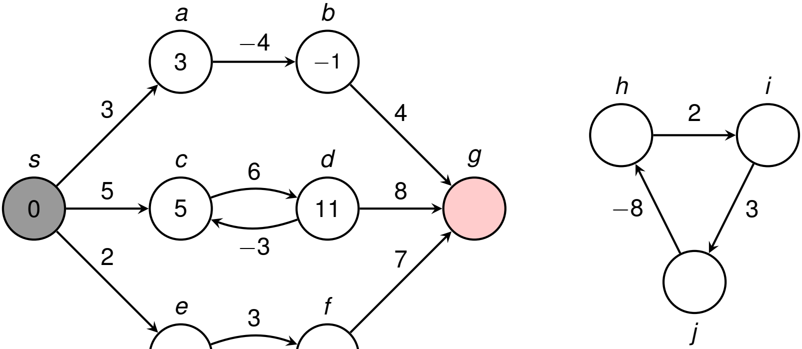
Distances and Negative-Weight Cycles (Figure 24.1)



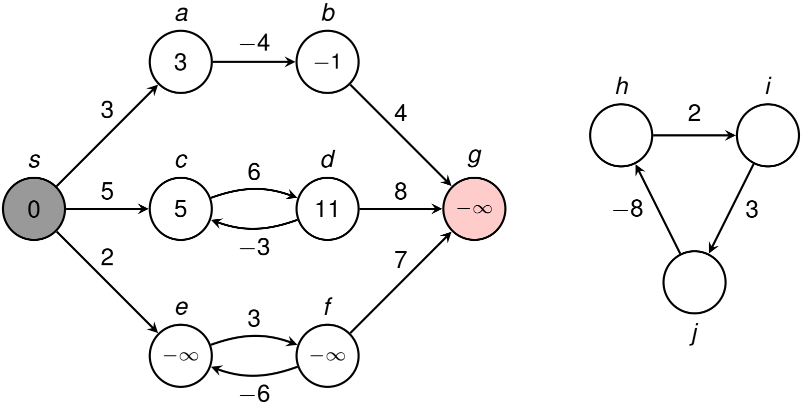
Distances and Negative-Weight Cycles (Figure 24.1)



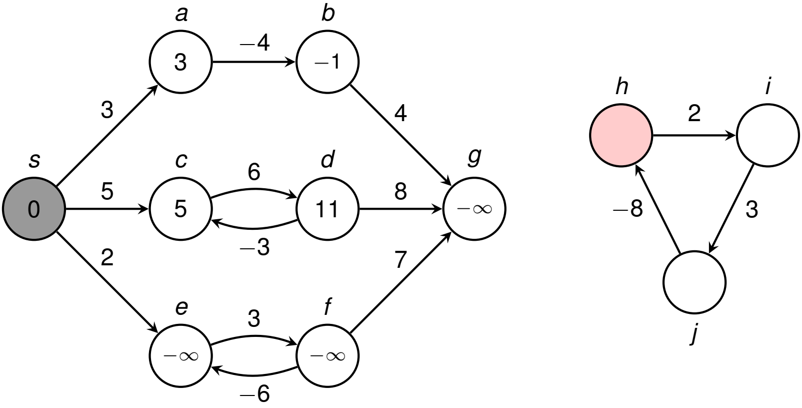
Distances and Negative-Weight Cycles (Figure 24.1)



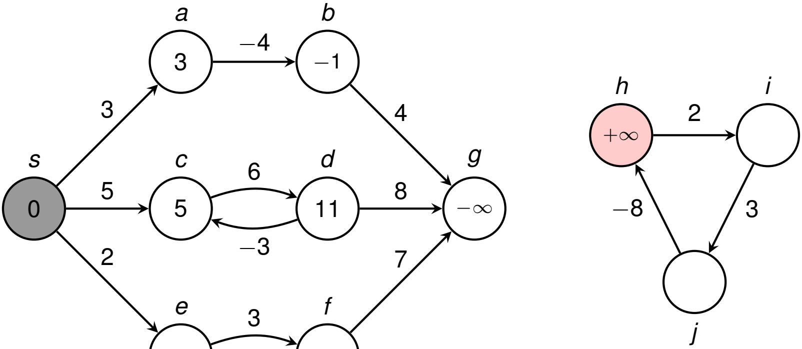
Distances and Negative-Weight Cycles (Figure 24.1)



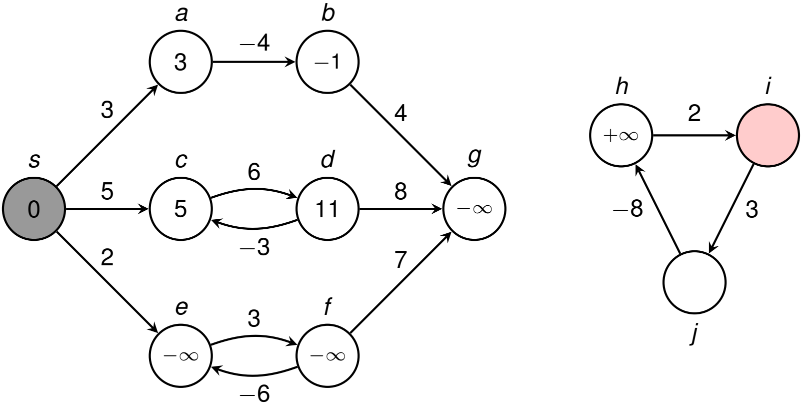
Distances and Negative-Weight Cycles (Figure 24.1)



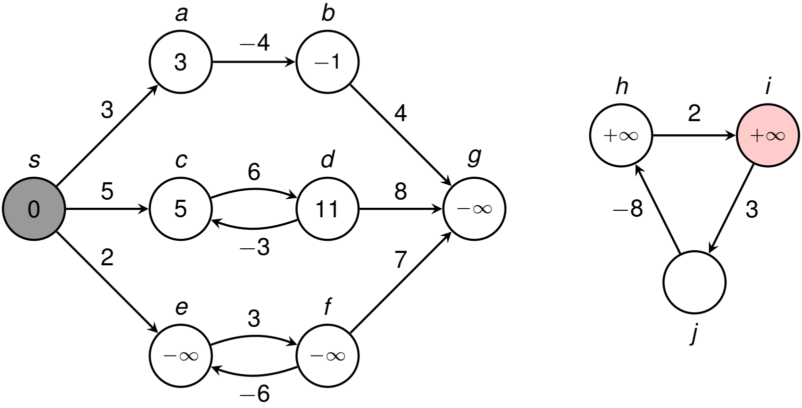
Distances and Negative-Weight Cycles (Figure 24.1)



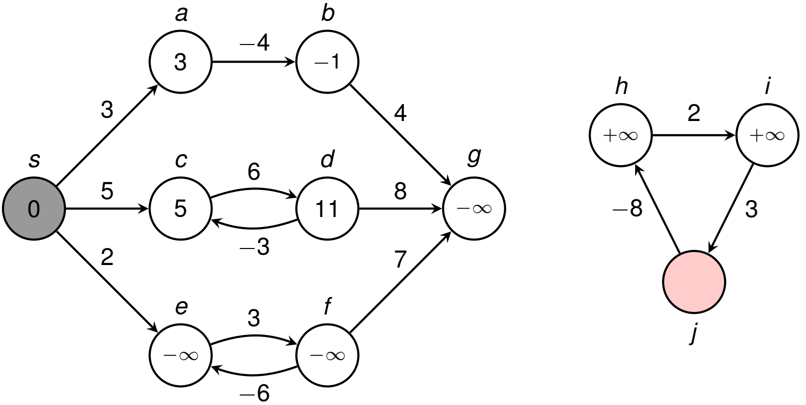
Distances and Negative-Weight Cycles (Figure 24.1)



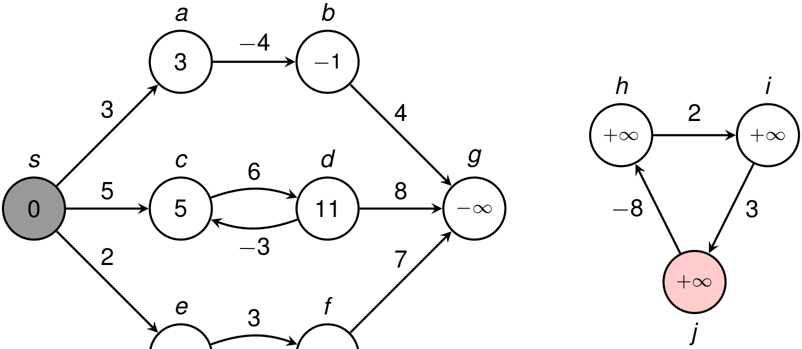
Distances and Negative-Weight Cycles (Figure 24.1)



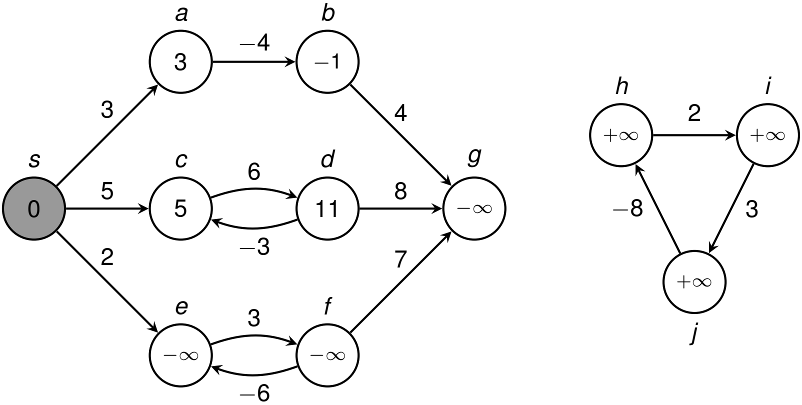
Distances and Negative-Weight Cycles (Figure 24.1)



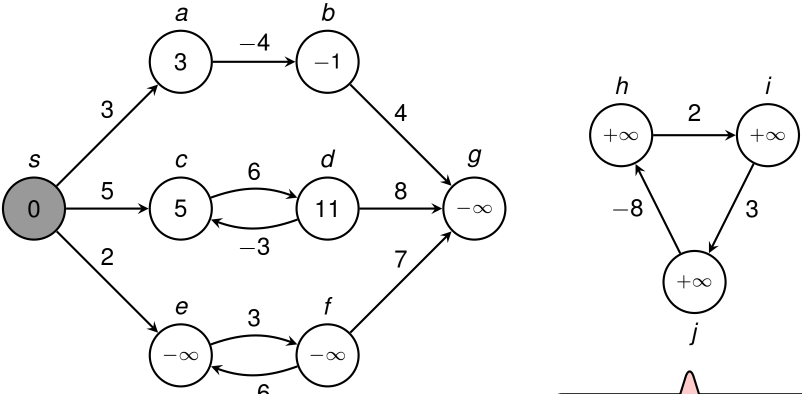
Distances and Negative-Weight Cycles (Figure 24.1)



Distances and Negative-Weight Cycles (Figure 24.1)



Distances and Negative-Weight Cycles (Figure 24.1)



Negative-Weight Cycle (reachable from *s*)

Negative-Weight Cycle (not reachable from *s*)

Outline

Introduction

Bellman-Ford Algorithm

Relaxing Edges

Definition

Fix the source vertex $s \in V$

- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

Relaxing Edges

Definition

Fix the source vertex $s \in V$

- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$

Relaxing Edges

Definition

Fix the source vertex $s \in V$

- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing Edges

Definition

Fix the source vertex $s \in V$

- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

Relaxing Edges

Definition

Fix the source vertex $s \in V$

- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$

Relaxing Edges

Definition

Fix the source vertex $s \in V$

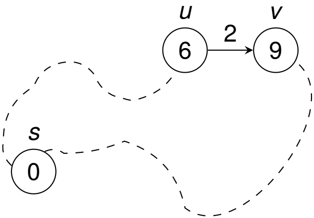
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



Relaxing Edges

Definition

Fix the source vertex $s \in V$

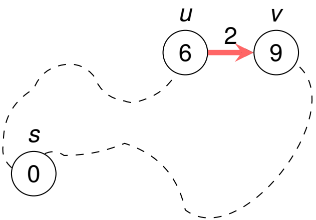
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



Relaxing Edges

Definition

Fix the source vertex $s \in V$

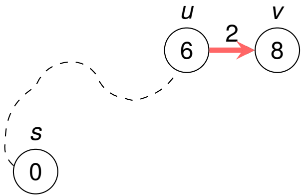
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



Relaxing Edges

Definition

Fix the source vertex $s \in V$

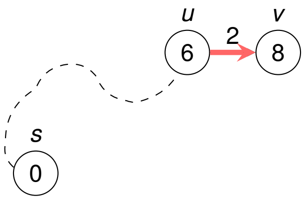
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



After relaxing (u, v), regardless of whether we found a shortcut: $v.d \leq u.d + w(u, v)$

Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

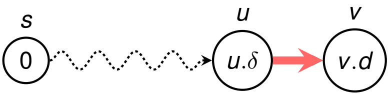
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

Upper-bound Property (Lemma 24.11)

- We always have $v.d \geq v.\delta$ for all $v \in V$, and once $v.d$ achieves the value $v.\delta$, it never changes.

Convergence Property (Lemma 24.14)

- If $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v, and if $u.d = u.\delta$ prior to relaxing edge (u, v), then $v.d = v.\delta$ at all times afterward.



$v.d$

Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

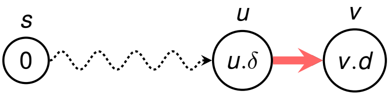
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

Upper-bound Property (Lemma 24.11)

- We always have $v.d \geq v.\delta$ for all $v \in V$, and once $v.d$ achieves the value $v.\delta$, it never changes.

Convergence Property (Lemma 24.14)

- If $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v, and if $u.d = u.\delta$ prior to relaxing edge (u, v), then $v.d = v.\delta$ at all times afterward.



$$v.d \leq u.d + w(u, v)$$

Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

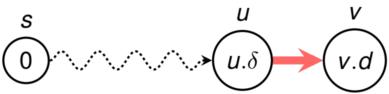
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

Upper-bound Property (Lemma 24.11)

- We always have $v.d \geq v.\delta$ for all $v \in V$, and once $v.d$ achieves the value $v.\delta$, it never changes.

Convergence Property (Lemma 24.14)

- If $s \rightsquigarrow u \to v$ is a shortest path from s to v, and if $u.d = u.\delta$ prior to relaxing edge (u, v), then $v.d = v.\delta$ at all times afterward.



$$\begin{aligned} v.d &\leq u.d + w(u, v) \\ &= u.\delta + w(u, v) \end{aligned}$$

Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

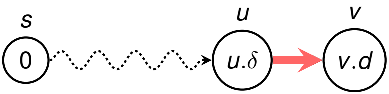
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

Upper-bound Property (Lemma 24.11)

- We always have $v.d \geq v.\delta$ for all $v \in V$, and once $v.d$ achieves the value $v.\delta$, it never changes.

Convergence Property (Lemma 24.14)

- If $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v, and if $u.d = u.\delta$ prior to relaxing edge (u, v), then $v.d = v.\delta$ at all times afterward.



$$
\begin{aligned}
v.d &\leq u.d + w(u, v) \\
&= u.\delta + w(u, v) \\
&= v.\delta
\end{aligned}
$$

Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

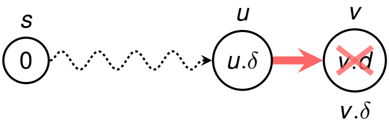
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

Upper-bound Property (Lemma 24.11)

- We always have $v.d \geq v.\delta$ for all $v \in V$, and once $v.d$ achieves the value $v.\delta$, it never changes.

Convergence Property (Lemma 24.14)

- If $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v, and if $u.d = u.\delta$ prior to relaxing edge (u, v), then $v.d = v.\delta$ at all times afterward.



$$\begin{aligned} v.d &\leq u.d + w(u, v) \\ &= u.\delta + w(u, v) \\ &= v.\delta \end{aligned}$$

Since $v.d \geq v.\delta$, we have $v.d = v.\delta$. □

Path-Relaxation Property

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Path-Relaxation Property

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Proof:

- By induction on i, $0 \leq i \leq k$:
  After the ith edge of p is relaxed, we have $v_i.d = v_i.\delta$.

Path-Relaxation Property

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Proof:

- By induction on i, $0 \leq i \leq k$:
  After the ith edge of p is relaxed, we have $v_i.d = v_i.\delta$.
- For $i = 0$, by the initialization $s.d = s.\delta = 0$.
  Upper-bound Property $\Rightarrow$ the value of $s.d$ never changes after that.

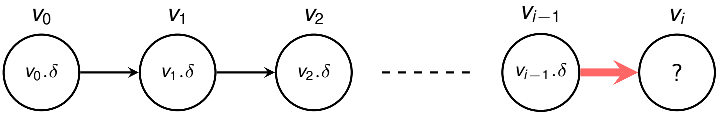
Path-Relaxation Property

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Proof:

- By induction on i, $0 \leq i \leq k$:
  After the ith edge of p is relaxed, we have $v_i.d = v_i.\delta$.
- For $i = 0$, by the initialization $s.d = s.\delta = 0$.
  Upper-bound Property $\Rightarrow$ the value of $s.d$ never changes after that.
- Inductive Step ($i - 1 \to i$): Assume $v_{i-1}.d = v_{i-1}.\delta$ and relax (v_{i-1}, v_i).



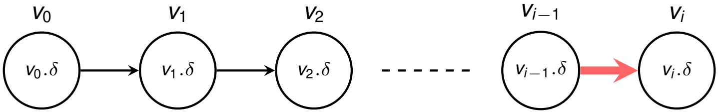
Path-Relaxation Property

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Proof:

- By induction on i, $0 \leq i \leq k$:
  After the ith edge of p is relaxed, we have $v_i.d = v_i.\delta$.
- For $i = 0$, by the initialization $s.d = s.\delta = 0$.
  Upper-bound Property $\Rightarrow$ the value of $s.d$ never changes after that.
- Inductive Step ($i - 1 \to i$): Assume $v_{i-1}.d = v_{i-1}.\delta$ and relax (v_{i-1}, v_i).
  Convergence Property $\Rightarrow$ $v_i.d = v_i.\delta$ (now and at all later steps) $\square$



Path-Relaxation Property

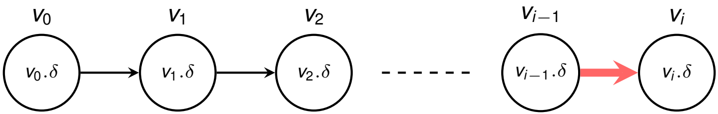
"Propagation": By relaxing proper edges, set of vertices with $v.\delta = v.d$ gets larger

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Proof:

- By induction on i, $0 \leq i \leq k$:
  After the ith edge of p is relaxed, we have $v_i.d = v_i.\delta$.
- For $i = 0$, by the initialization $s.d = s.\delta = 0$.
  Upper-bound Property $\Rightarrow$ the value of $s.d$ never changes after that.
- Inductive Step ($i - 1 \to i$): Assume $v_{i-1}.d = v_{i-1}.\delta$ and relax (v_{i-1}, v_i).
  Convergence Property $\Rightarrow$ $v_i.d = v_i.\delta$ (now and at all later steps) □



The Bellman-Ford Algorithm

```
 BELLMAN-FORD(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0

repeat |V|-1 times
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
      if e.start.d + e.weight.d < e.end.d:
         e.end.d = e.start.d + e.weight
         e.end.predecessor = e.start

for e in G.edges()
   if e.start.d + e.weight.d < e.end.d:
      return FALSE
return TRUE
```

The Bellman-Ford Algorithm

```
BELLMAN-FORD(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0
repeat |V|-1 times
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
      if e.start.d + e.weight.d < e.end.d:
         e.end.d = e.start.d + e.weight
         e.end.predecessor = e.start
for e in G.edges()
   if e.start.d + e.weight.d < e.end.d:
      return FALSE
return TRUE
```

Time Complexity

The Bellman-Ford Algorithm

```
 BELLMAN-FORD(G,w,s)
0: assert(s in G.vertices())
1: for v in G.vertices()
2:    v.predecessor = None
3:    v.d = Infinity
4: s.d = 0
5:
6: repeat |V|-1 times
7:    for e in G.edges()
8:    Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
9:       if e.start.d + e.weight.d < e.end.d:
10:          e.end.d = e.start.d + e.weight
11:          e.end.predecessor = e.start
12:
13: for e in G.edges()
14:    if e.start.d + e.weight.d < e.end.d:
15:       return FALSE
16: return TRUE
```

Time Complexity

- A single call of line 9-11 costs $\mathcal{O}(1)$

The Bellman-Ford Algorithm

```
 BELLMAN-FORD(G,w,s)
0: assert(s in G.vertices())
1: for v in G.vertices()
2:    v.predecessor = None
3:    v.d = Infinity
4: s.d = 0
5:
6: repeat |V|-1 times
7:    for e in G.edges()
8:    Relax edge e=(u,v): Check if u.d + w(u,v) < v.d
9:       if e.start.d + e.weight.d < e.end.d:
10:         e.end.d = e.start.d + e.weight
11:         e.end.predecessor = e.start
12:
13: for e in G.edges()
14:    if e.start.d + e.weight.d < e.end.d:
15:       return FALSE
16: return TRUE
```

Time Complexity

- A single call of line 9-11 costs $\mathcal{O}(1)$
- In each pass every edge is relaxed $\Rightarrow \mathcal{O}(E)$ time per pass

The Bellman-Ford Algorithm

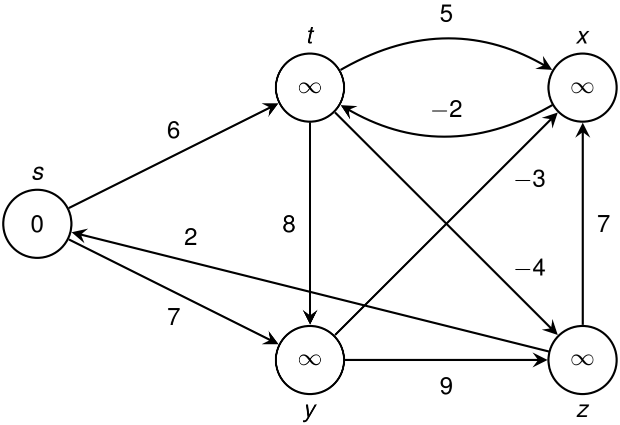
```
 BELLMAN-FORD(G,w,s)
0: assert(s in G.vertices())
1: for v in G.vertices()
2:    v.predecessor = None
3:    v.d = Infinity
4: s.d = 0
5:
6: repeat |V|-1 times
7:    for e in G.edges()
8:    Relax edge e=(u,v): Check if u.d + w(u,v) < v.d
9:       if e.start.d + e.weight.d < e.end.d:
10:         e.end.d = e.start.d + e.weight
11:         e.end.predecessor = e.start
12:
13: for e in G.edges()
14:    if e.start.d + e.weight.d < e.end.d:
15:       return FALSE
16: return TRUE
```

Time Complexity

- A single call of line 9-11 costs $\mathcal{O}(1)$
- In each pass every edge is relaxed $\Rightarrow \mathcal{O}(E)$ time per pass
- Overall $(V-1)+1 = V$ passes $\Rightarrow \mathcal{O}(V \cdot E)$ time

Execution of Bellman-Ford (Figure 24.4)

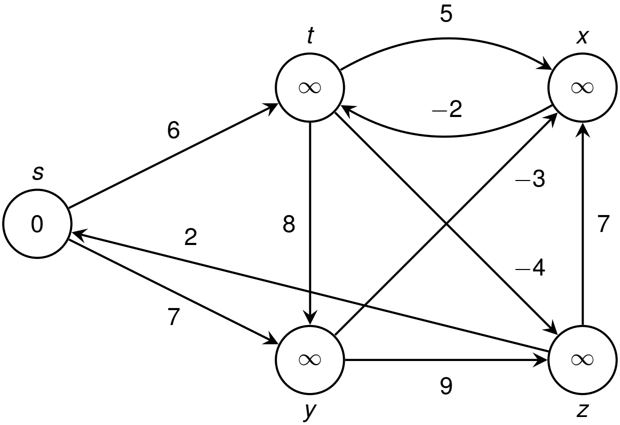
Pass: 1



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

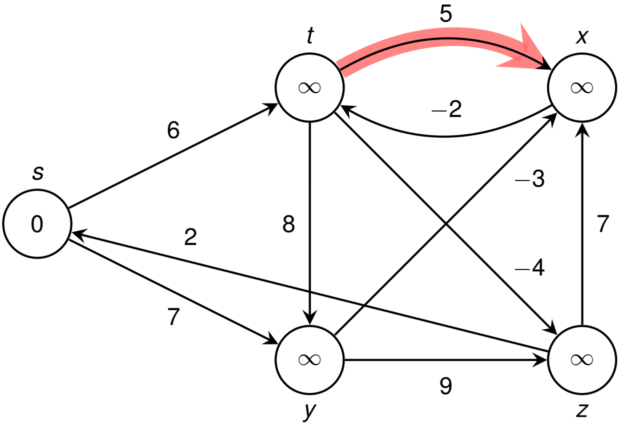
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

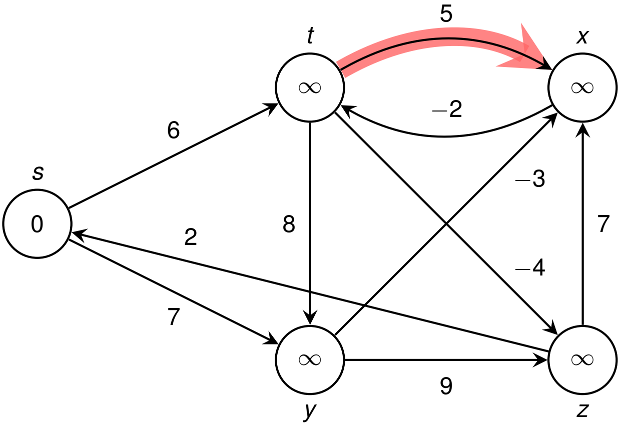
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

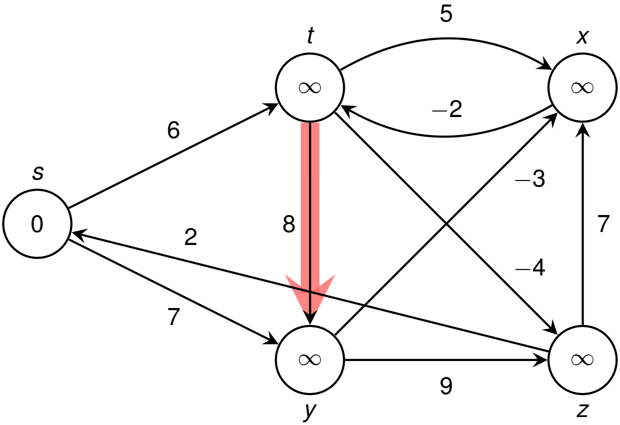
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

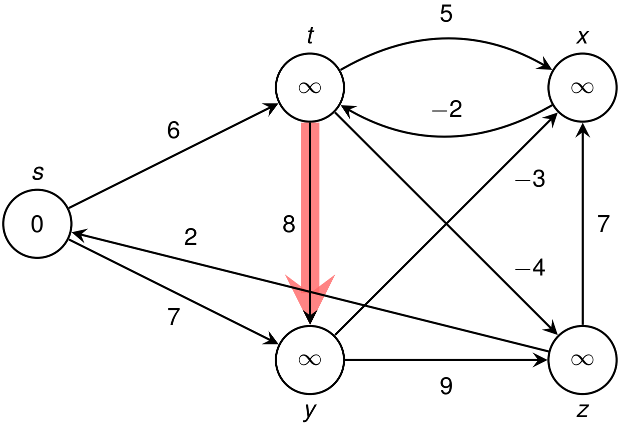
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

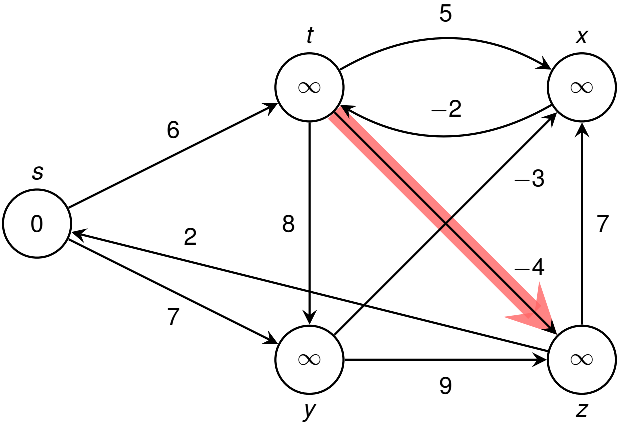
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

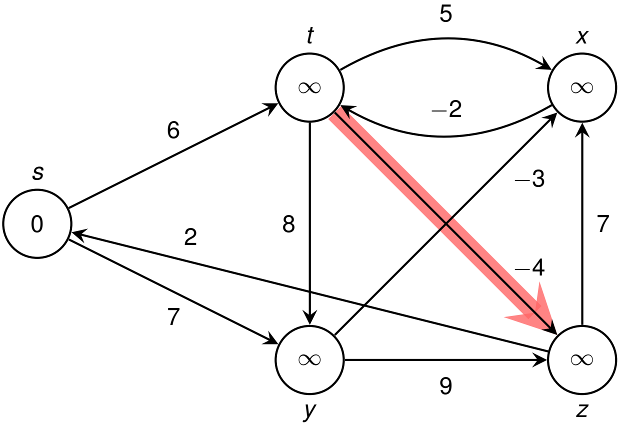
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

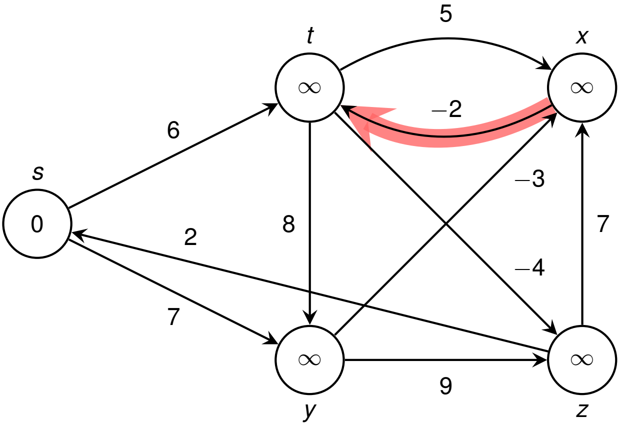
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

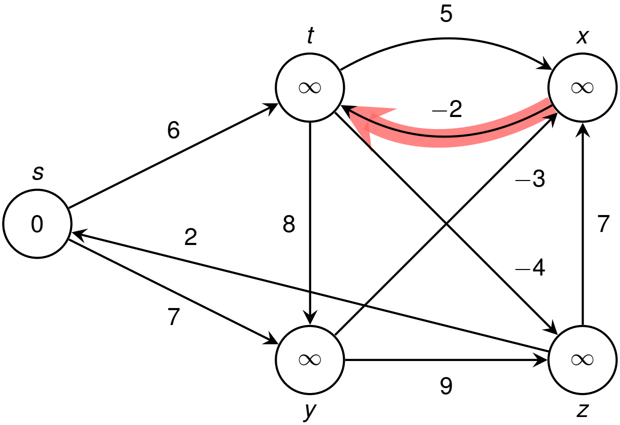
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

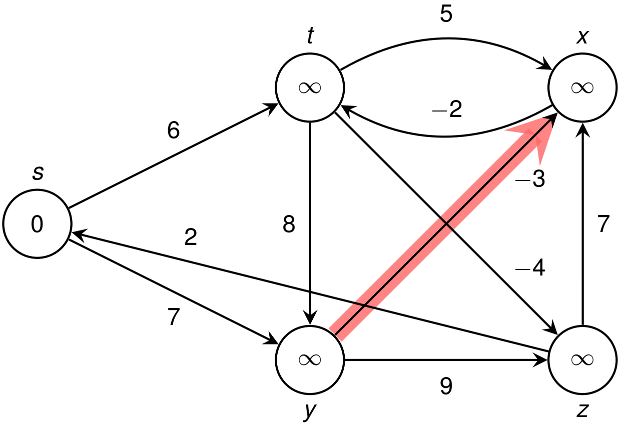
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

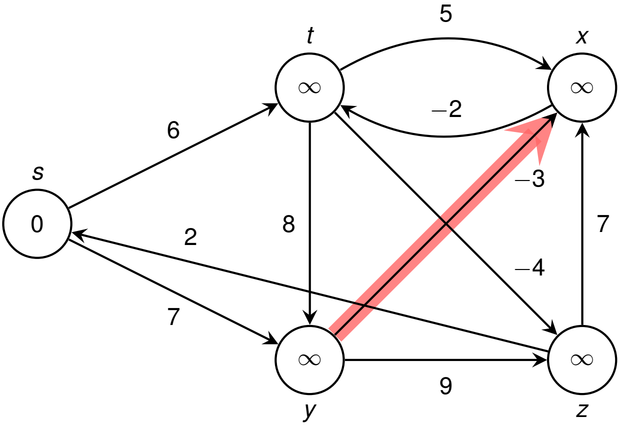
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

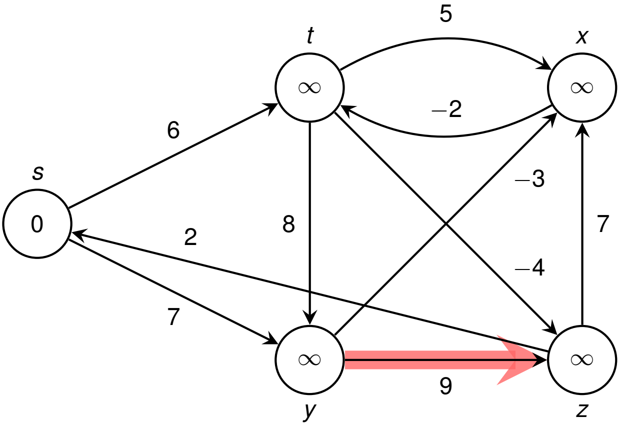
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

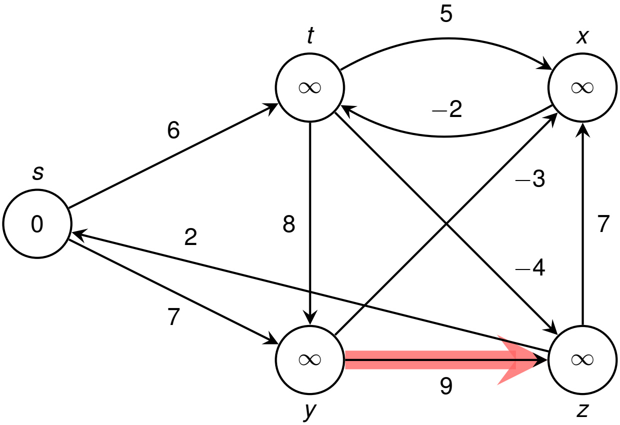
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

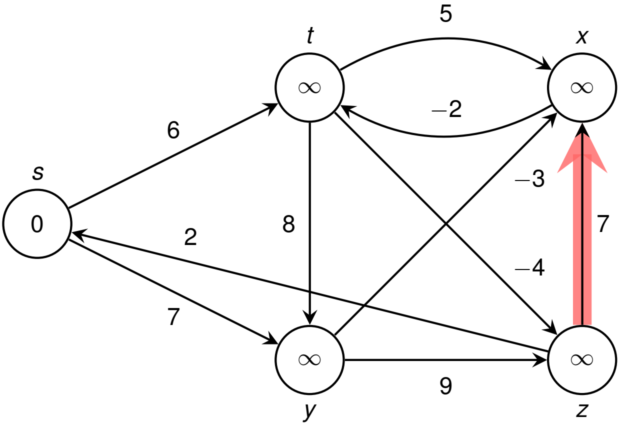
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

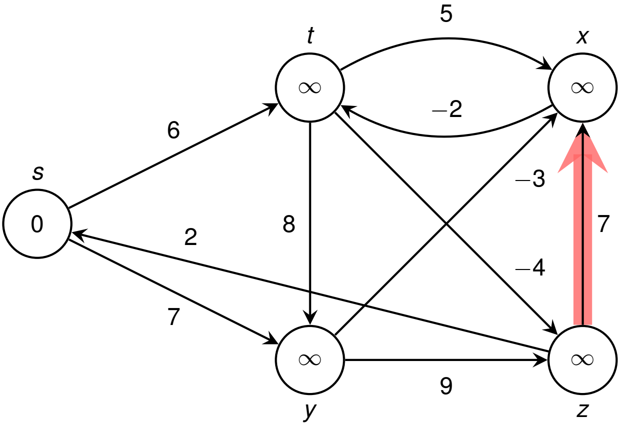
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

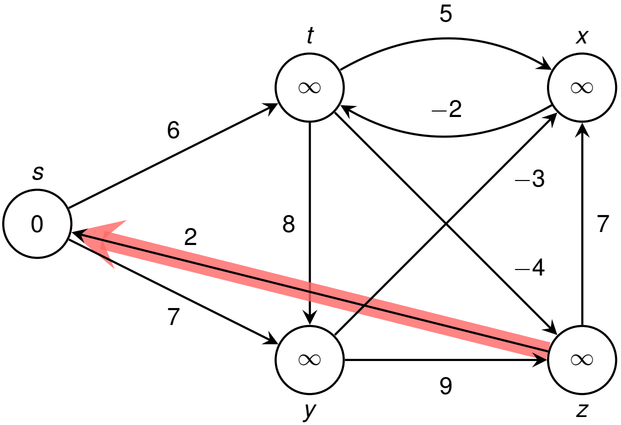
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

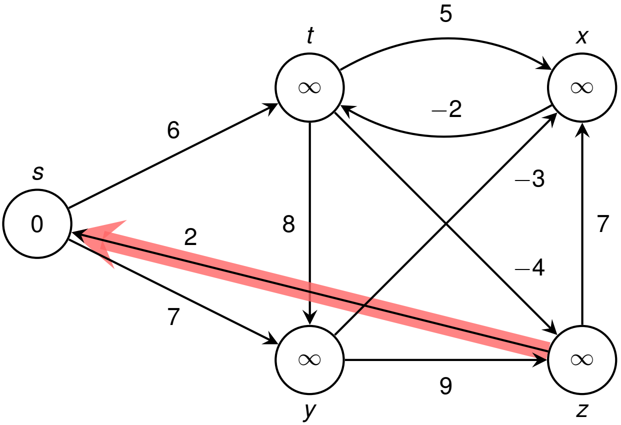
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

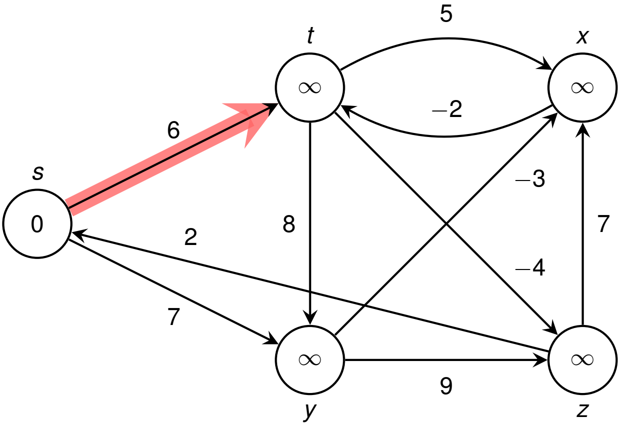
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

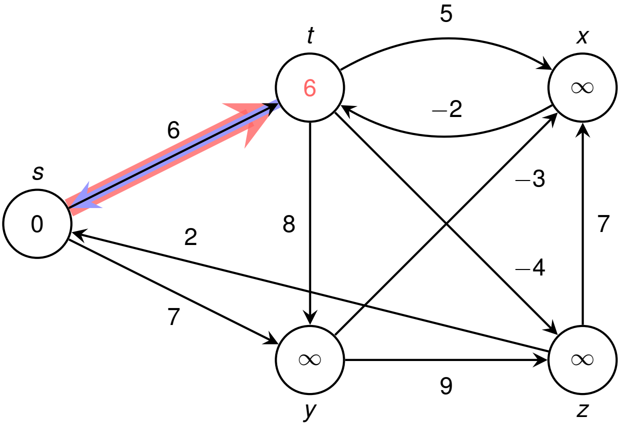
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

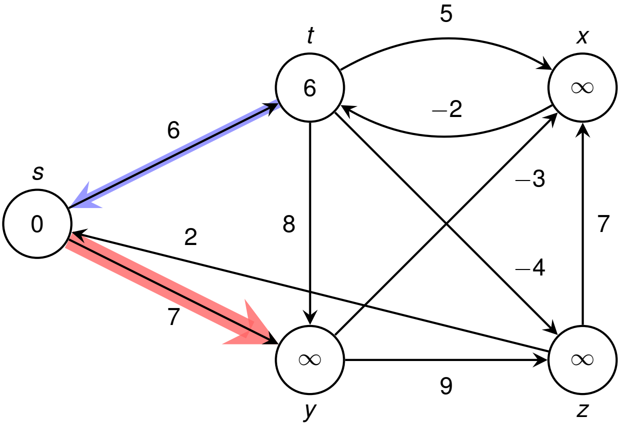
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

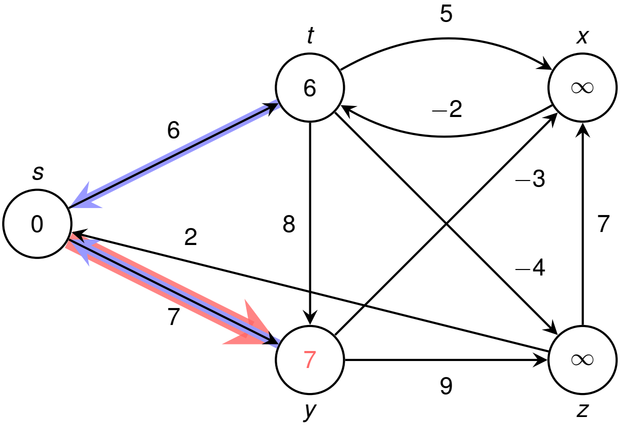
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 1

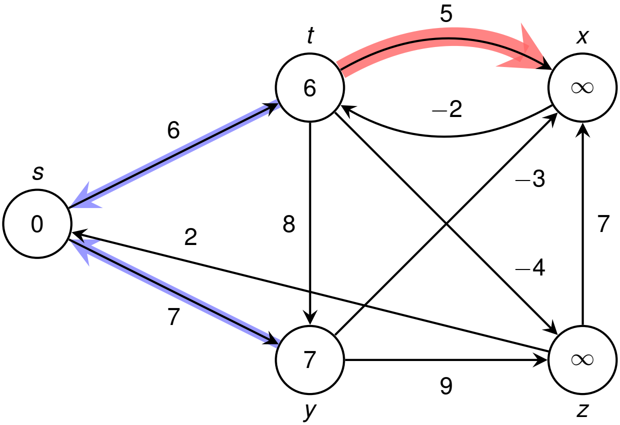
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

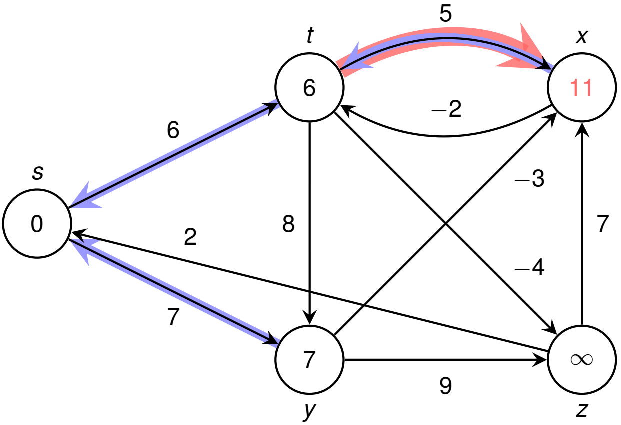
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

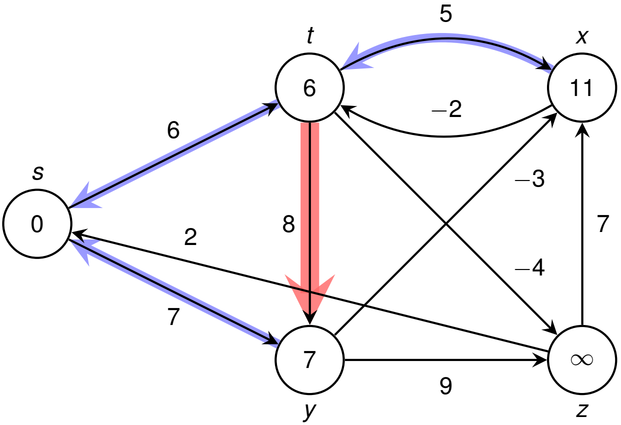
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

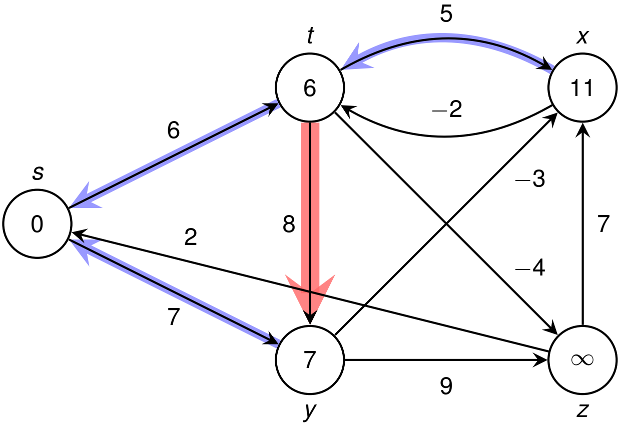
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

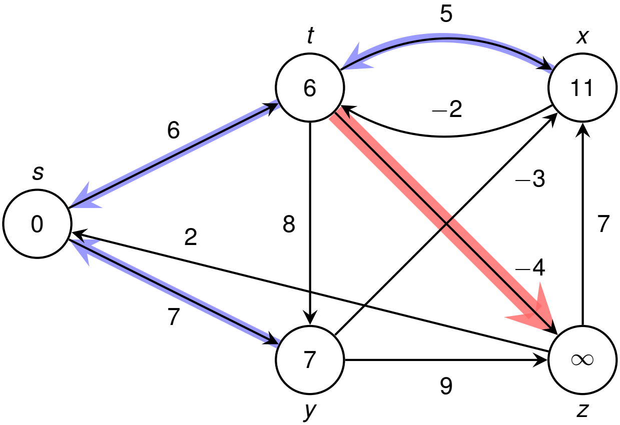
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

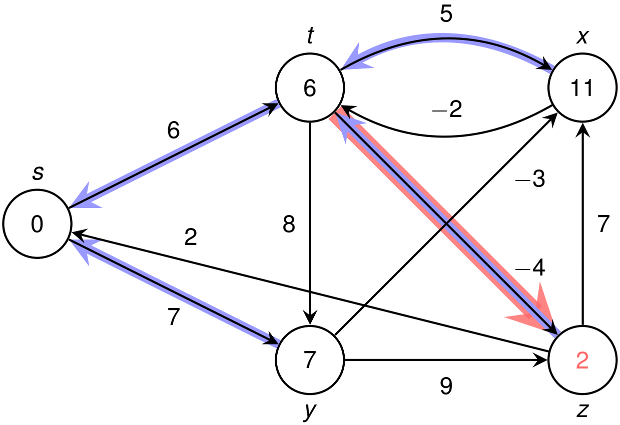
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

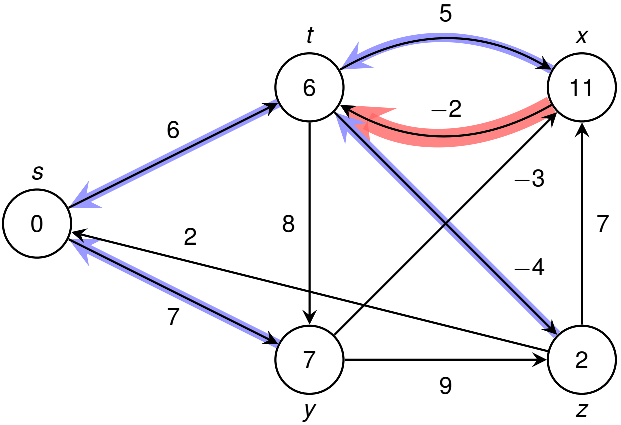
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

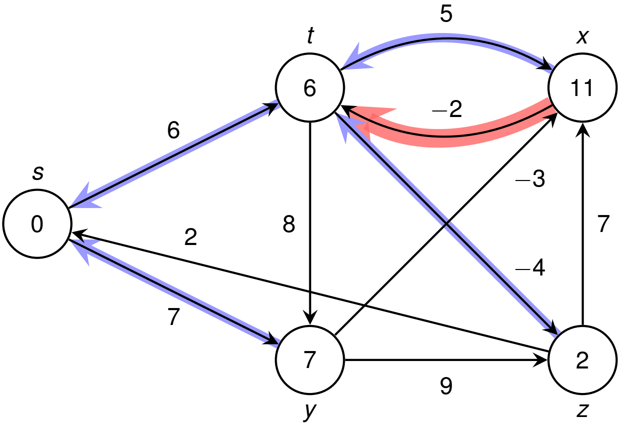
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

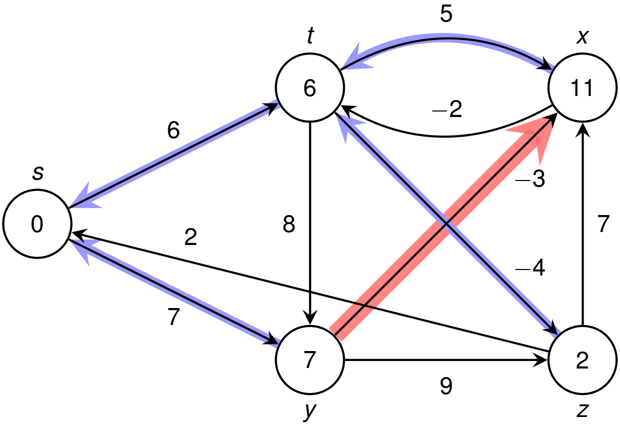
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

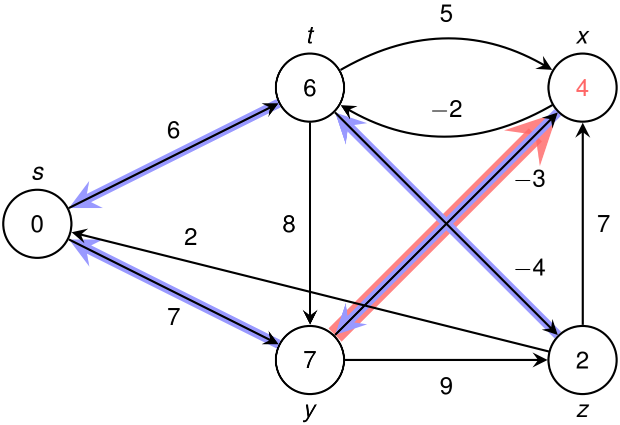
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

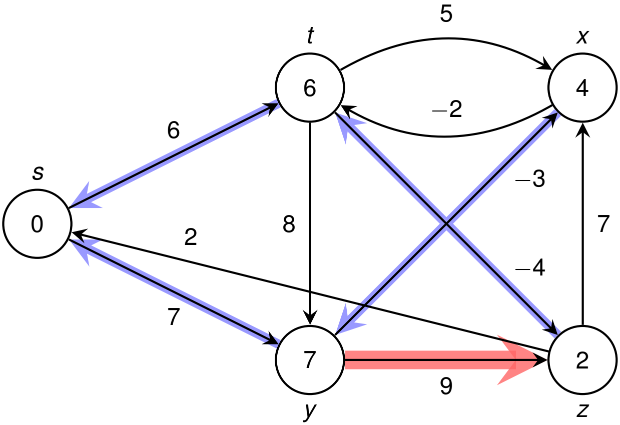
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

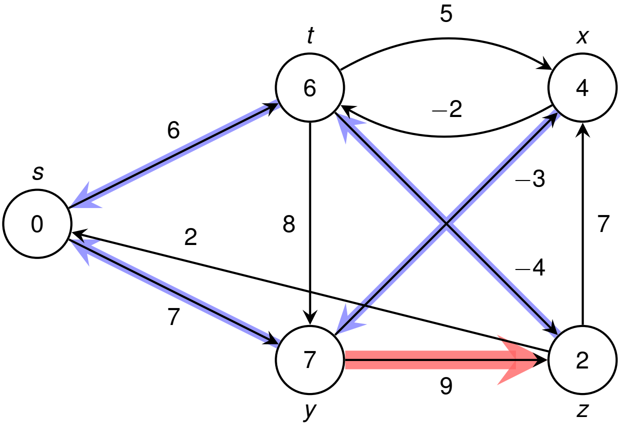
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

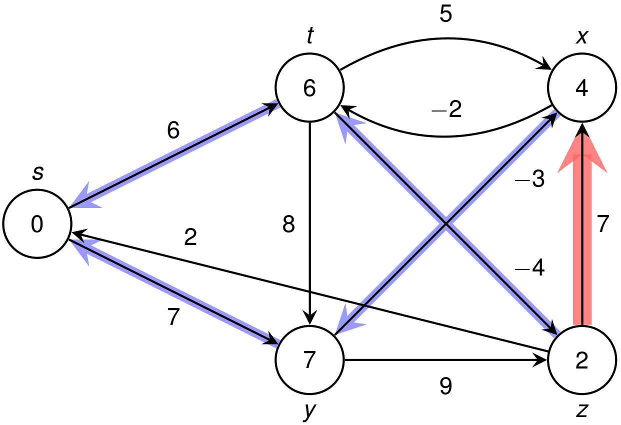
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

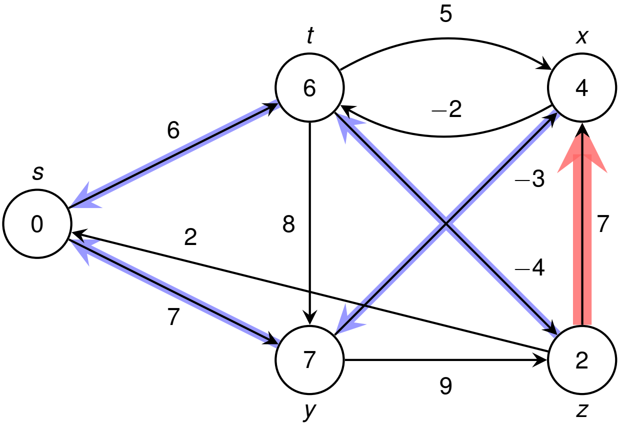
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

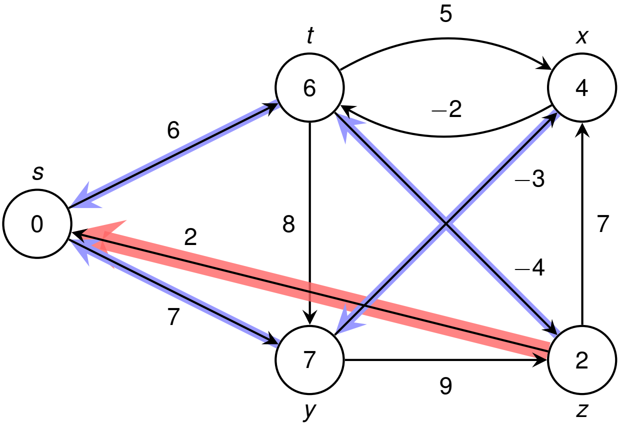
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

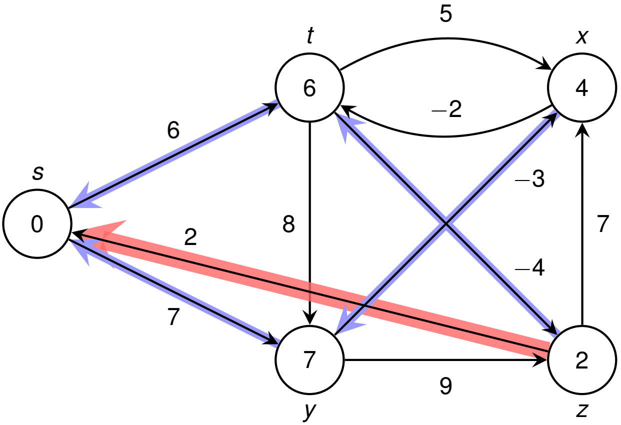
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

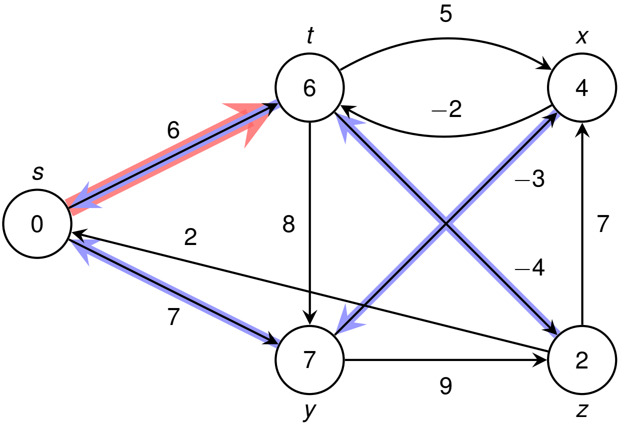
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

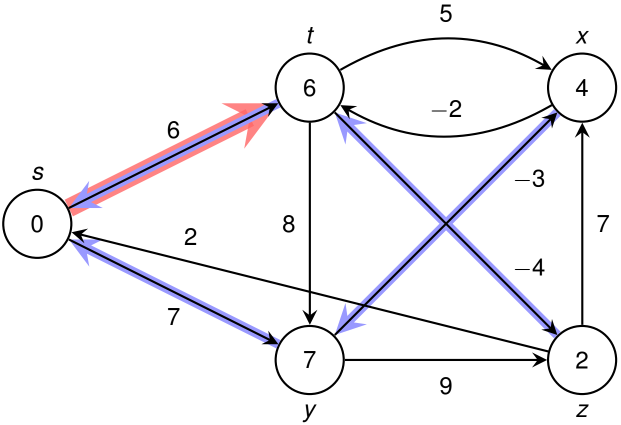
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

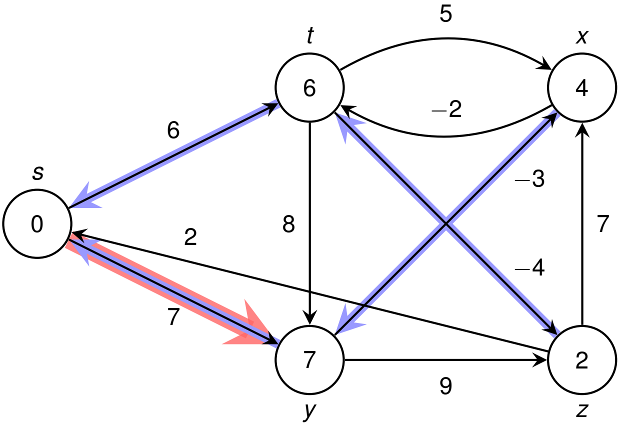
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

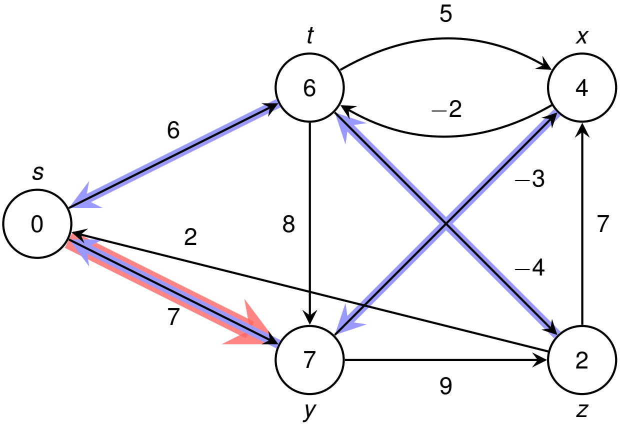
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 2

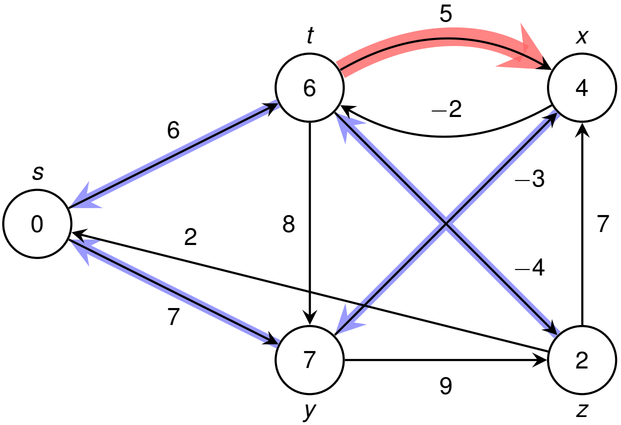
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

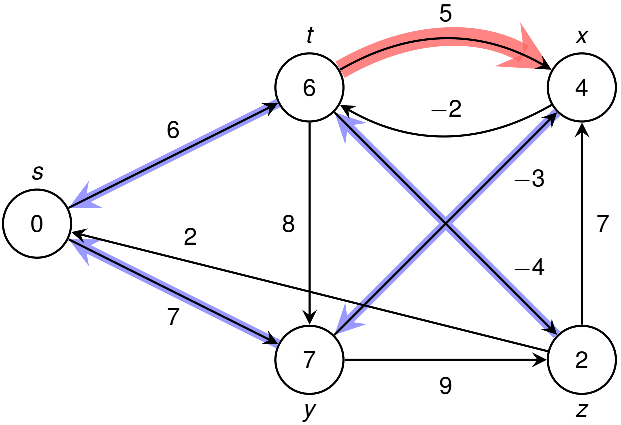
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

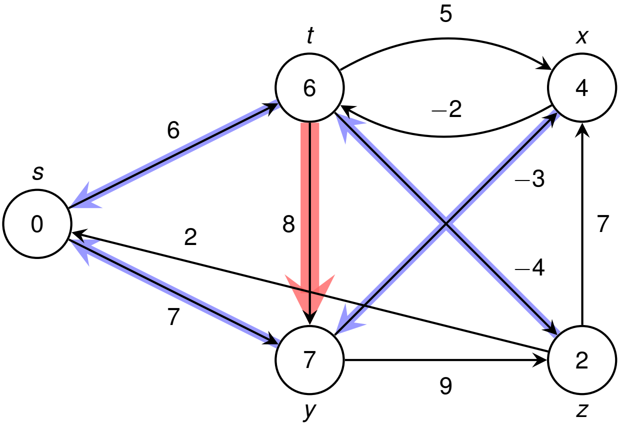
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

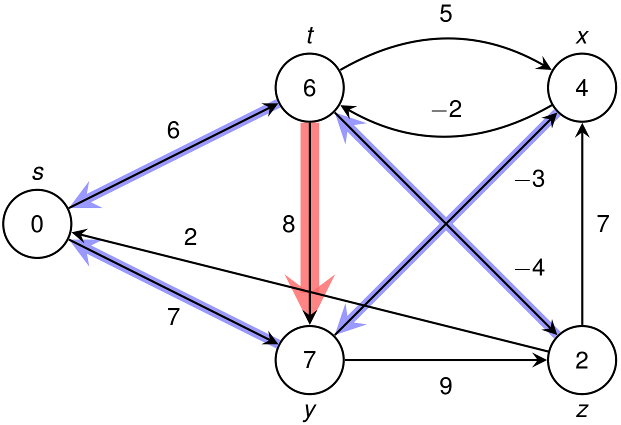
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

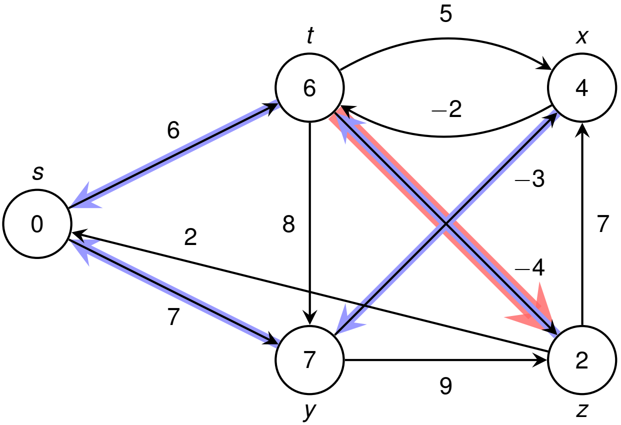
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

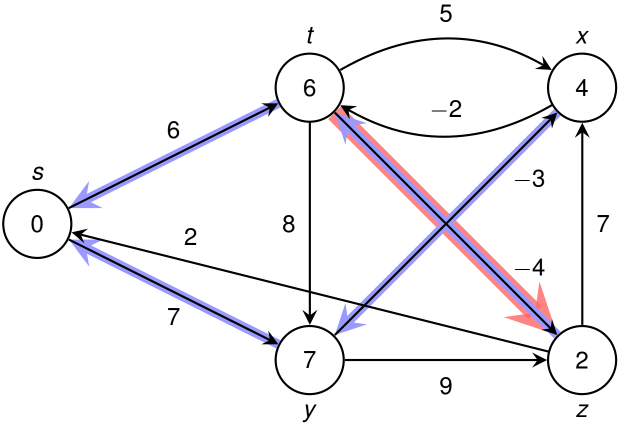
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

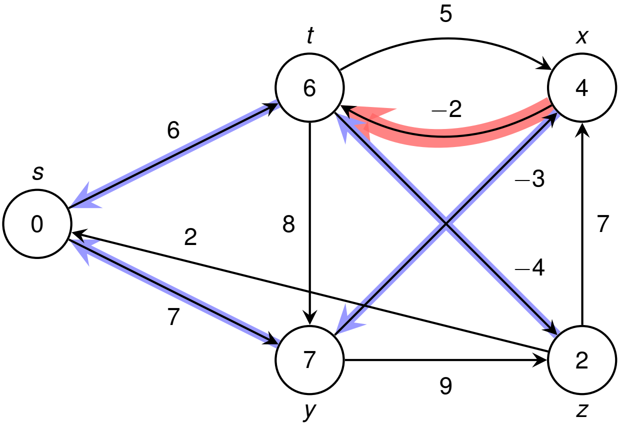
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

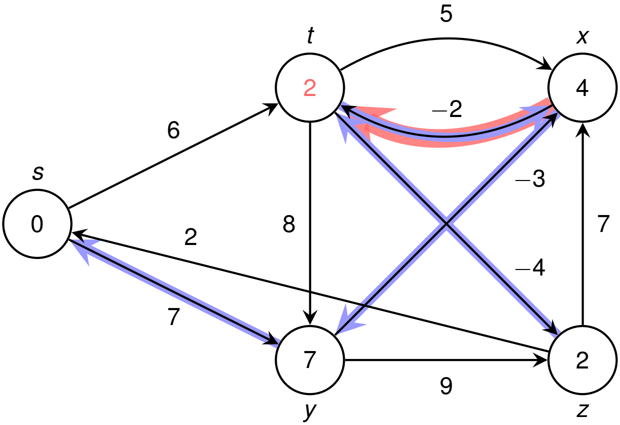
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

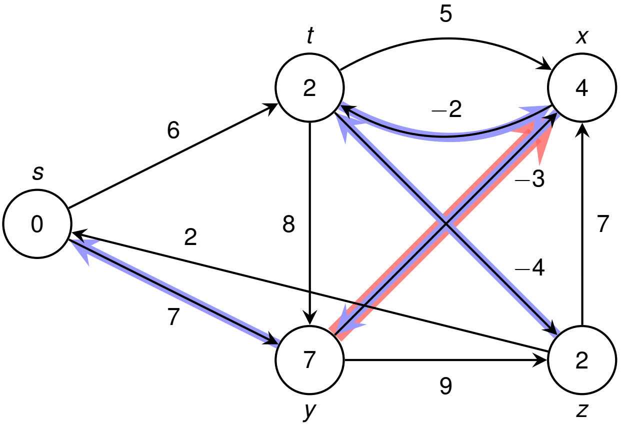
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

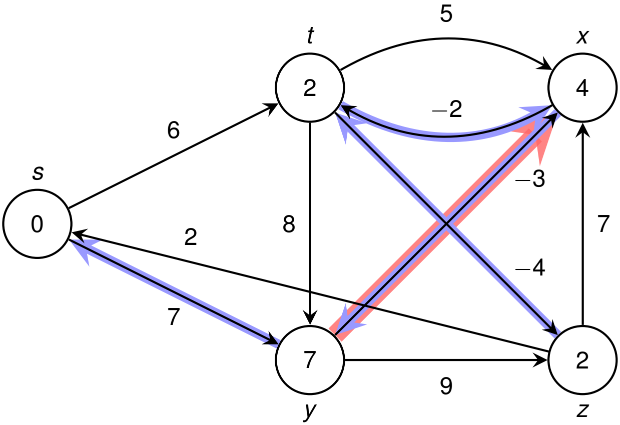
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

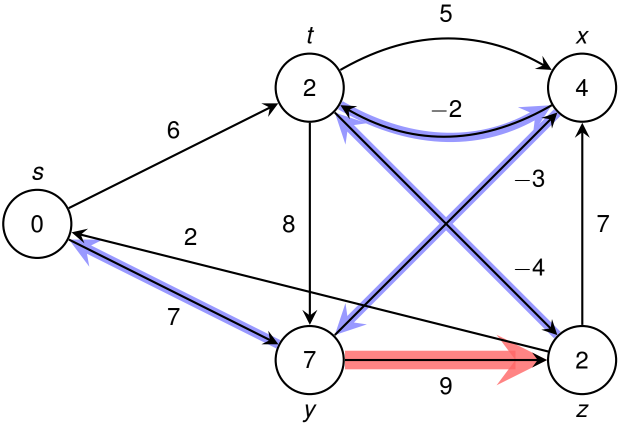
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

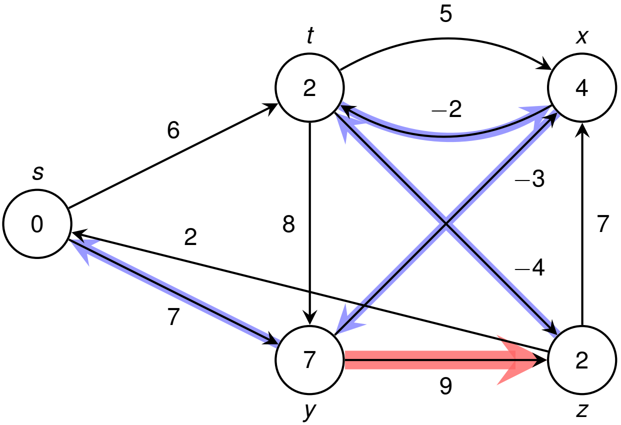
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

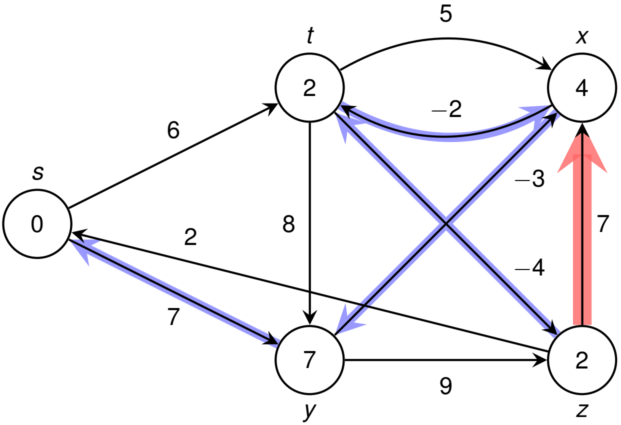
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

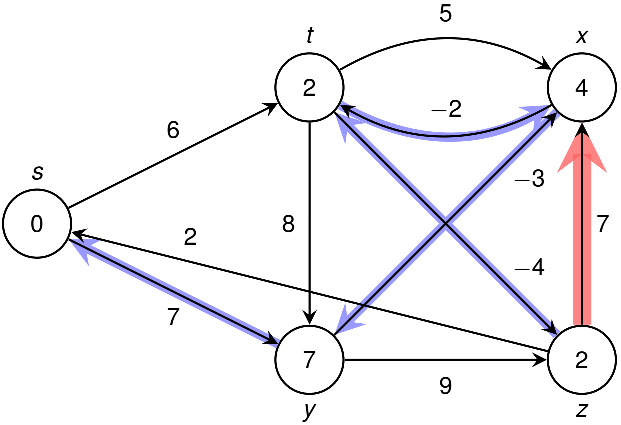
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

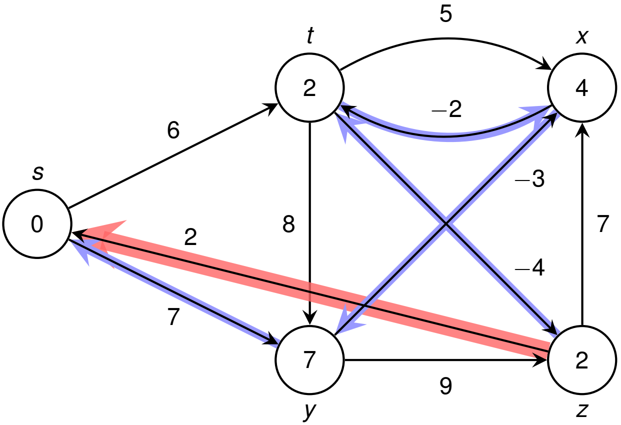
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

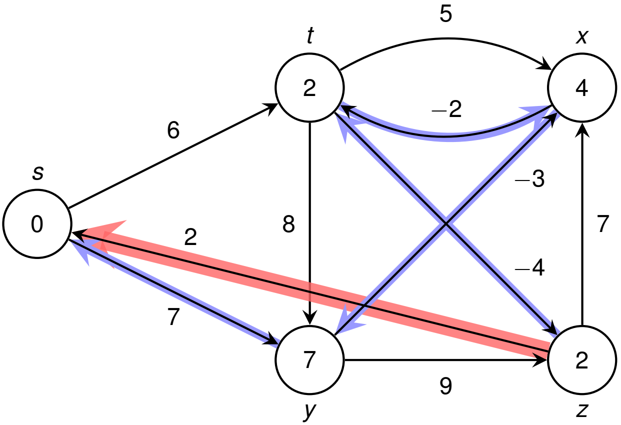
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

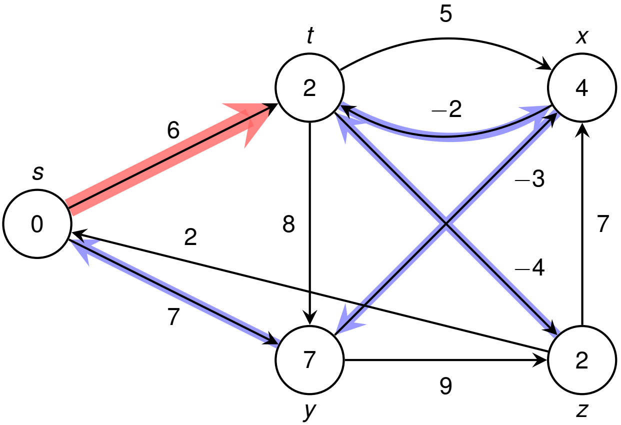
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

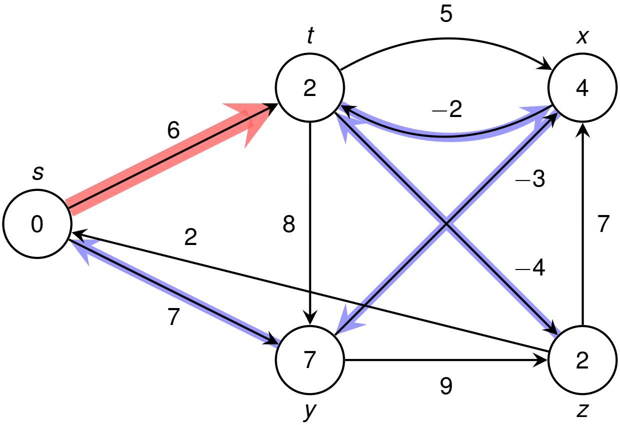
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

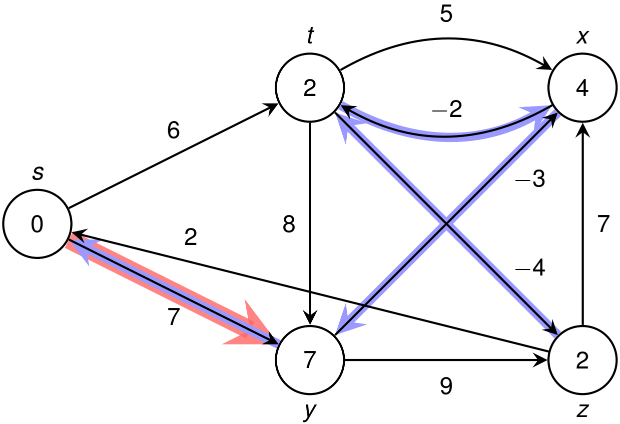
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

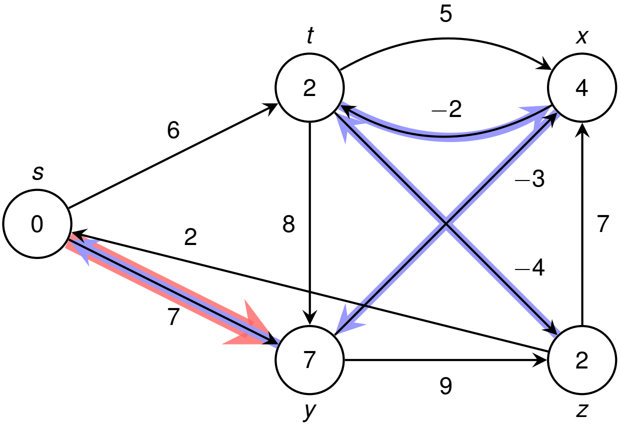
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 3

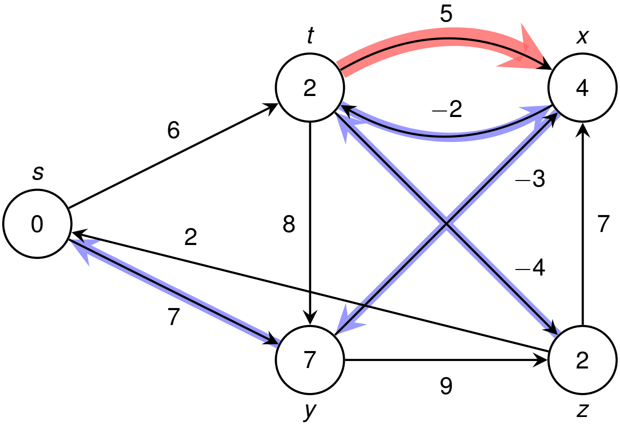
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

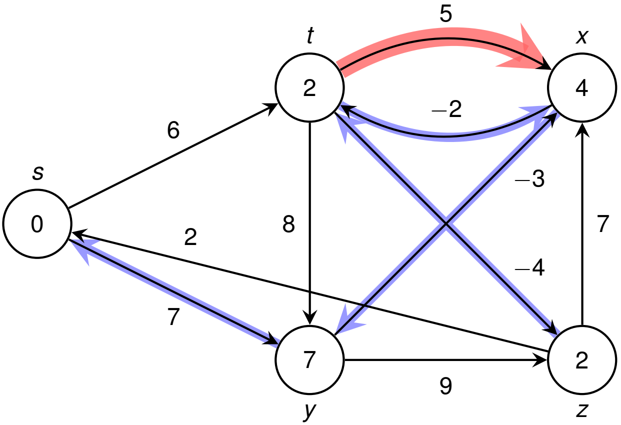
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

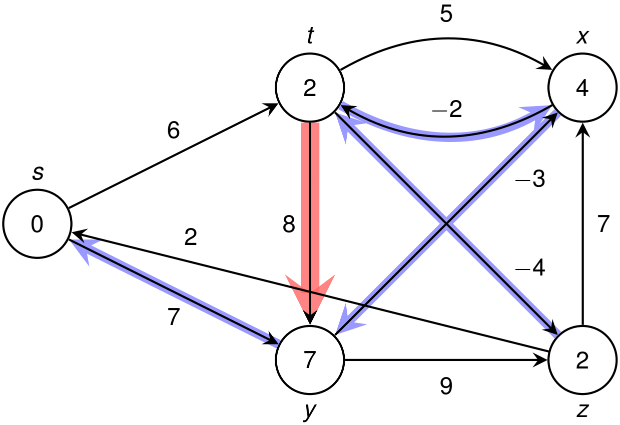
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

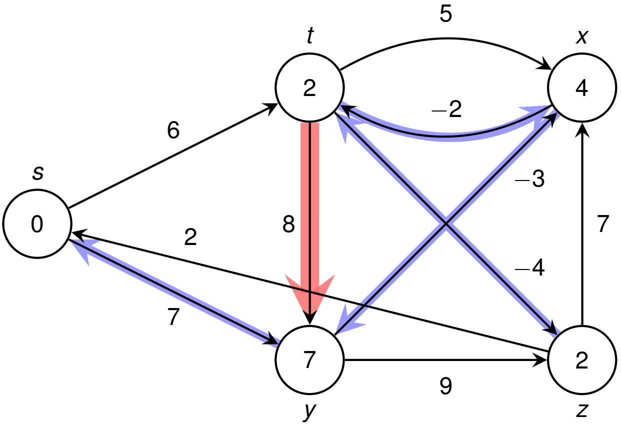
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

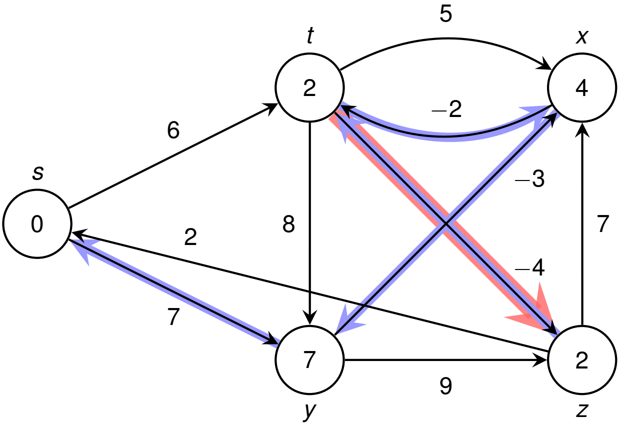
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

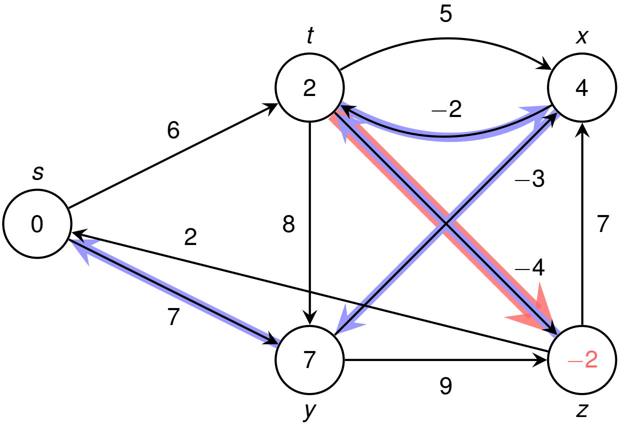
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

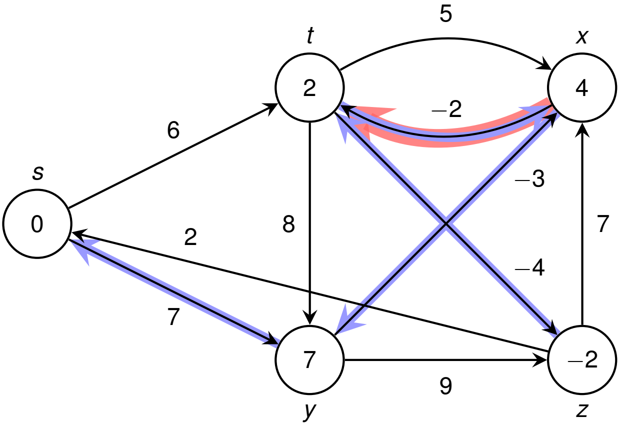
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

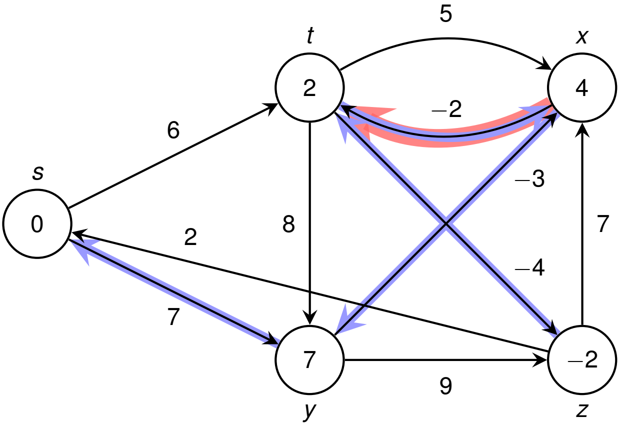
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

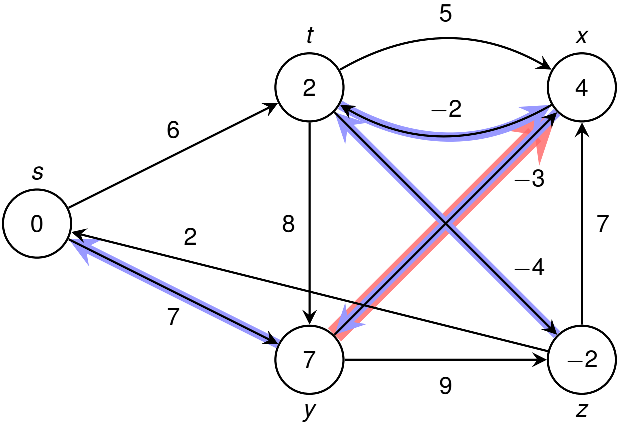
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

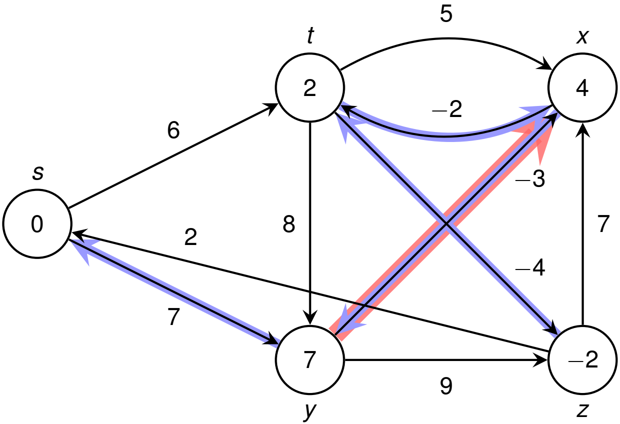
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

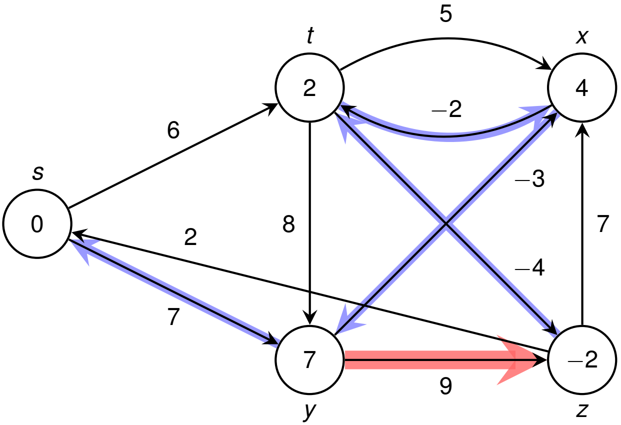
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

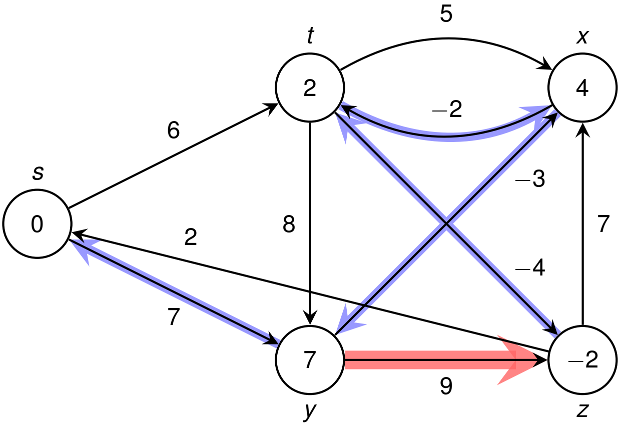
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

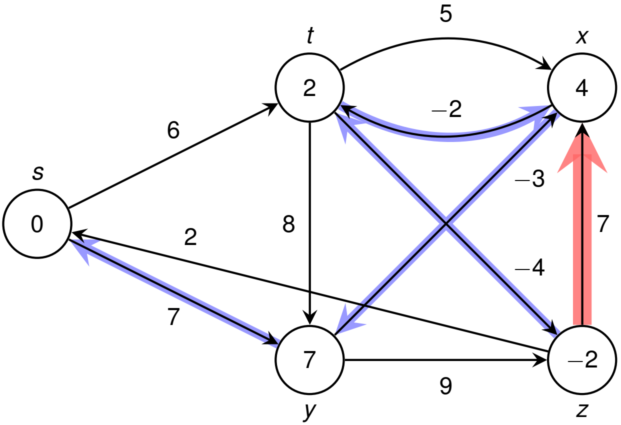
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

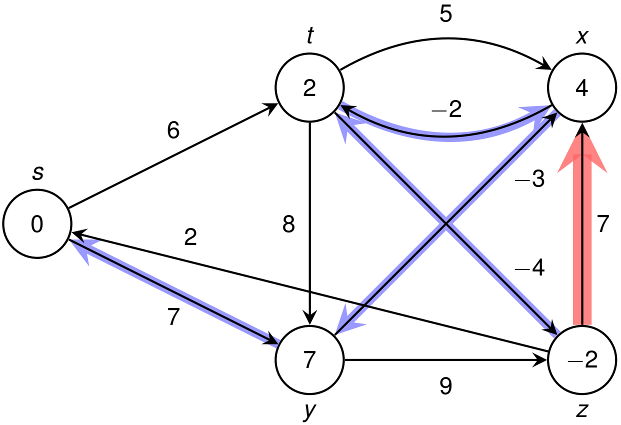
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

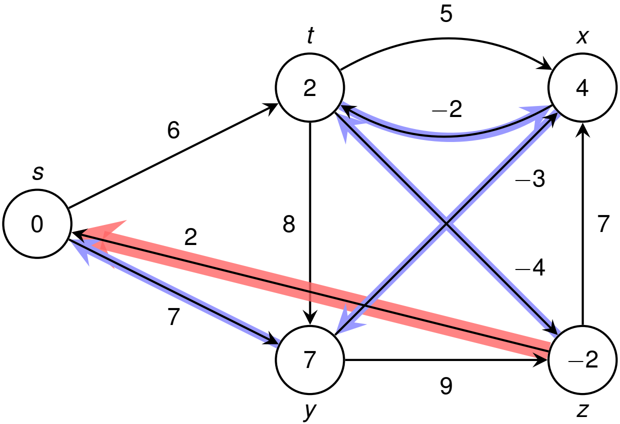
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

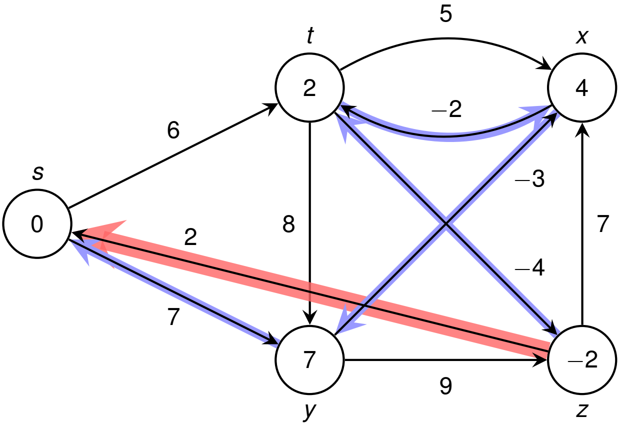
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

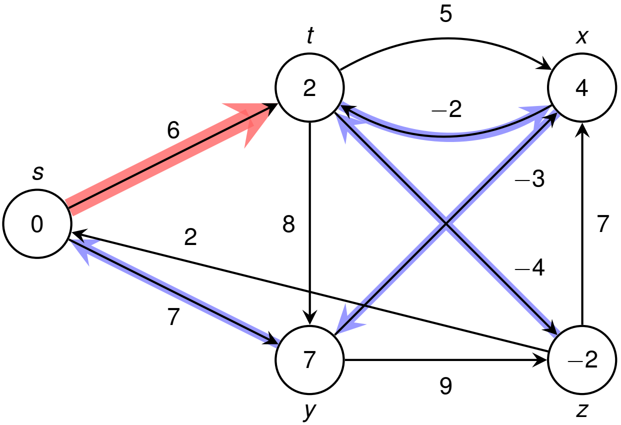
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

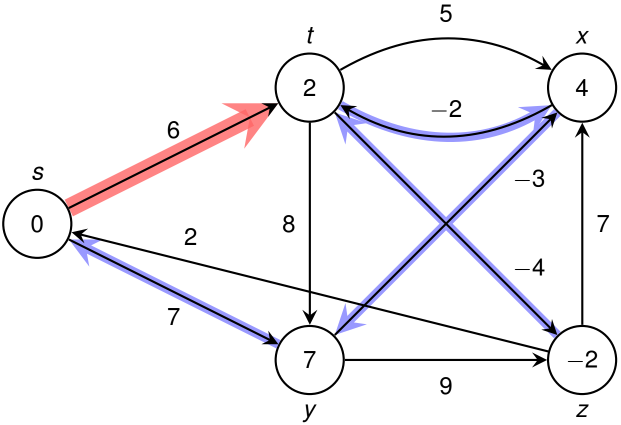
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

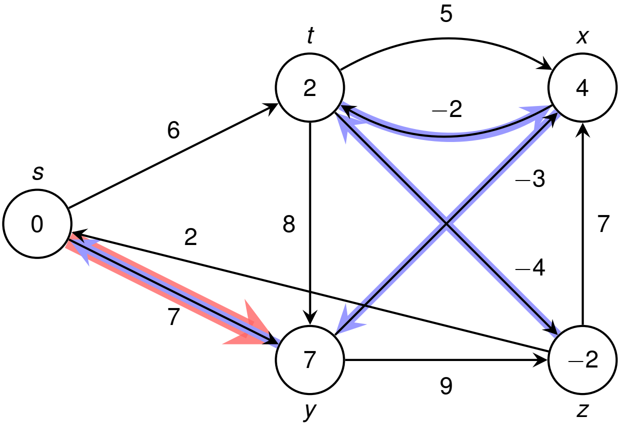
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

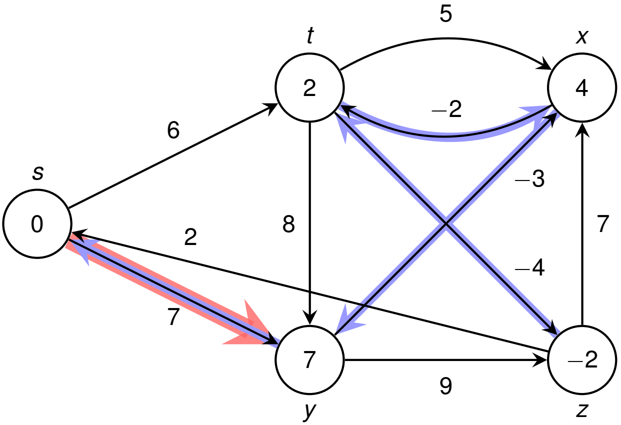
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

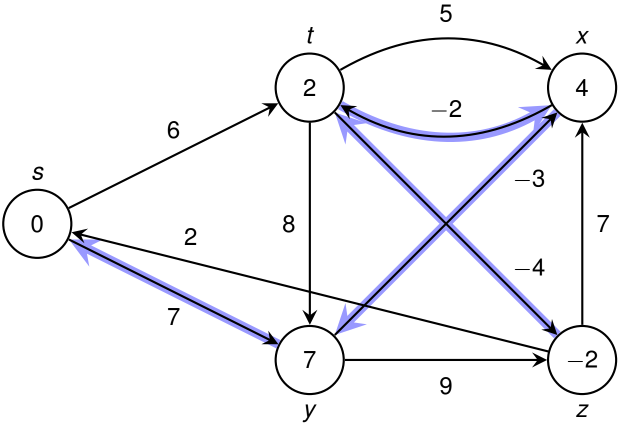
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4)

Pass: 4

Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V| - 1$ passes, $v.d = v.\delta$

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V|-1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V|-1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V|-1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V|-1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V| - 1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges
- Let $(u, v) \in E$ be any edge. After $|V| - 1$ passes:

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V| - 1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges
- Let $(u, v) \in E$ be any edge. After $|V| - 1$ passes:

$$v.d = v.\delta$$

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V| - 1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges
- Let $(u, v) \in E$ be any edge. After $|V| - 1$ passes:

$$v.d = v.\delta \leq u.\delta + w(u, v)$$

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V| - 1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges
- Let $(u, v) \in E$ be any edge. After $|V| - 1$ passes:

$$v.d = v.\delta \leq u.\delta + w(u, v) = u.d + w(u, v)$$

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V| - 1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V| - 1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges
- Let $(u, v) \in E$ be any edge. After $|V| - 1$ passes:

$$v.d = v.\delta \leq u.\delta + w(u, v) = u.d + w(u, v) \qquad \square$$

Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V|-1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ (that are reachable) and Bellman-Ford returns TRUE.

Proof that $v.d = v.\delta$

- Let v be a vertex reachable from s
- Let $p = (v_0 = s, v_1, \ldots, v_k = v)$ be a shortest path from s to v
- p is simple, hence $k \leq |V| - 1$
- Path-Relaxation Property $\Rightarrow$ after $|V|-1$ passes, $v.d = v.\delta$

Proof that Bellman-Ford returns TRUE

- Need to prove: $v.d \leq u.d + w(u, v)$ for all edges
- Let $(u, v) \in E$ be any edge. After $|V| - 1$ passes:

$$v.d = v.\delta \leq u.\delta + w(u, v) = u.d + w(u, v) \qquad \square$$

Triangle inequality (holds even if $w(u, v) < 0$!)

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Proof:

- Let $c = (v_0, v_1, \ldots, v_k = v_0)$ be a negative-weight cycle reachable from s

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Proof:

- Let $c = (v_0, v_1, \ldots, v_k = v_0)$ be a negative-weight cycle reachable from s
- If Bellman-Ford returns TRUE, then for every $1 \leq i < k$,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Proof:

- Let $c = (v_0, v_1, \ldots, v_k = v_0)$ be a negative-weight cycle reachable from s
- If Bellman-Ford returns TRUE, then for every $1 \leq i < k$,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

$$\Rightarrow \quad \sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Proof:

- Let $c = (v_0, v_1, \ldots, v_k = v_0)$ be a negative-weight cycle reachable from s
- If Bellman-Ford returns TRUE, then for every $1 \leq i < k$,

$$
\begin{aligned}
&& v_i.d &\leq v_{i-1}.d + w(v_{i-1}, v_i) \\
\Rightarrow && \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
\Rightarrow && 0 &\leq \sum_{i=1}^{k} w(v_{i-1}, v_i)
\end{aligned}
$$

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Proof:

- Let $c = (v_0, v_1, \ldots, v_k = v_0)$ be a negative-weight cycle reachable from s
- If Bellman-Ford returns TRUE, then for every $1 \leq i < k$,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

$$\Rightarrow \quad \sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\Rightarrow \qquad 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

This cancellation is only valid if all $.d$-values are finite!

Bellman-Ford Algorithm: Correctness (2/2)

Theorem 24.3

If G contains a negative-weight cycle reachable from s, then Bellman-Ford returns FALSE.

Proof:

- Let $c = (v_0, v_1, \ldots, v_k = v_0)$ be a negative-weight cycle reachable from s
- If Bellman-Ford returns TRUE, then for every $1 \leq i < k$,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

$$\Rightarrow \quad \sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\Rightarrow \quad 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

This cancellation is only valid if all $.d$-values are finite!

- This contradicts the assumption that c is a negative-weight cycle! $\square$

The Bellman-Ford Algorithm

```
BELLMAN-FORD(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0

repeat |V|-1 times
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
      if e.start.d + e.weight.d < e.end.d:
         e.end.d = e.start.d + e.weight
         e.end.predecessor = e.start

for e in G.edges()
   if e.start.d + e.weight.d < e.end.d:
      return FALSE
return TRUE
```

The Bellman-Ford Algorithm

```
BELLMAN-FORD(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0

repeat |V|-1 times
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
      if e.start.d + e.weight.d < e.end.d:
         e.end.d = e.start.d + e.weight
         e.end.predecessor = e.start

for e in G.edges()
   if e.start.d + e.weight.d < e.end.d:
      return FALSE
return TRUE
```

Can we terminate earlier if there is a pass that keeps all *.d* variables?

The Bellman-Ford Algorithm

```
BELLMAN-FORD(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0

repeat |V|-1 times
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
      if e.start.d + e.weight.d < e.end.d:
         e.end.d = e.start.d + e.weight
         e.end.predecessor = e.start

for e in G.edges()
   if e.start.d + e.weight.d < e.end.d:
      return FALSE
return TRUE
```

Can we terminate earlier if there is a pass that keeps all *.d* variables?

Yes, because if pass i keeps all *.d* variables, then so does pass $i+1$.

The Bellman-Ford Algorithm (modified)

```
 BELLMAN-FORD-NEW(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0

repeat |V| times
   flag = 0
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
       if e.start.d + e.weight.d < e.end.d:
          e.end.d = e.start.d + e.weight
          e.end.predecessor = e.start
          flag = 1
   if flag = 0 return TRUE

return FALSE
```

Can we terminate earlier if there is a pass that keeps all *.d* variables?

Yes, because if pass i keeps all *.d* variables, then so does pass $i+1$.

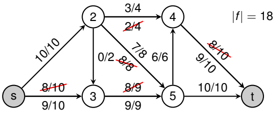
The Bellman-Ford Algorithm (modified)

```
BELLMAN-FORD-NEW(G,w,s)
assert(s in G.vertices())
for v in G.vertices()
   v.predecessor = None
   v.d = Infinity
s.d = 0

repeat |V| times
   flag = 0
   for e in G.edges()
   Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
       if e.start.d + e.weight.d < e.end.d:
          e.end.d = e.start.d + e.weight
          e.end.predecessor = e.start
          flag = 1
   if flag = 0 return TRUE

return FALSE
```

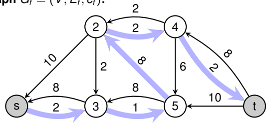
Can we terminate earlier if there is a pass that keeps all *.d* variables?

Yes, because if pass i keeps all *.d* variables, then so does pass $i+1$.

Graph $G = (V, E, c)$:



Residual Graph $G_f = (V, E_f, c_f)$:



6.6: Maximum flow

Frank Stajano

Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

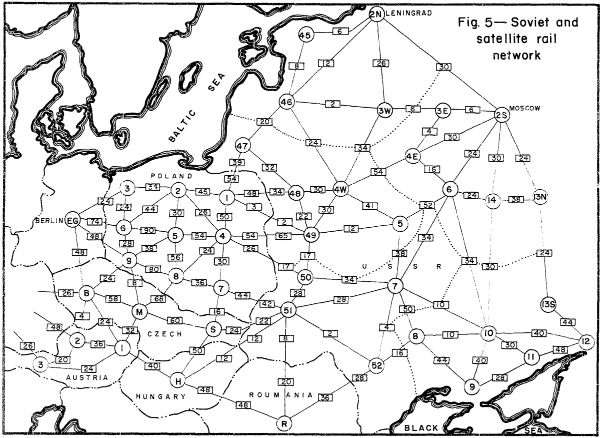
Introduction

Ford-Fulkerson

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

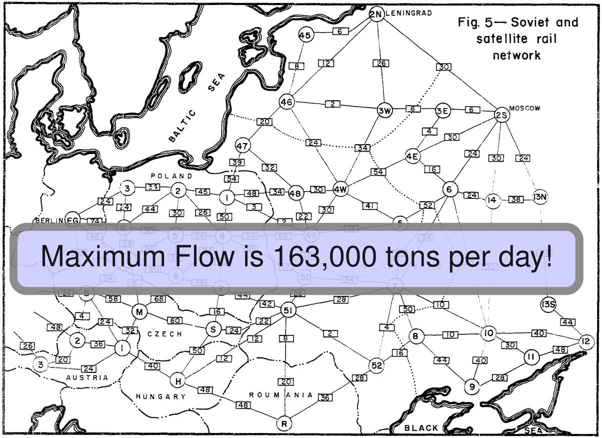
History of the Maximum Flow Problem [Harris, Ross (1955)]



Legend: —·· — International boundary ·········· Regional boundaries of the USSR (they are included as a matter of general information)

(7) Operating divisions. Those located in Russia are believed to be accurately located. Some Russian divisions (2, 3, 4 and 13) are located in two regions and are so indicated. Divisions shown in the satellites are indicated according to the authors' best judgment, since intelligence reports are unavailable. Train capacities in Russia are for 1000-net-ton trains or their equivalent. Train capacities in Poland are for 666 net tons (or the equivalent). Train capacities in all other satellites are for 400 net tons (or the equivalent) except in East Germany. In East Germany, train capacities are those of entering lines. The numbers shown in boxes are total interdivisional capacities.

History of the Maximum Flow Problem [Harris, Ross (1955)]



Maximum Flow is 163,000 tons per day!

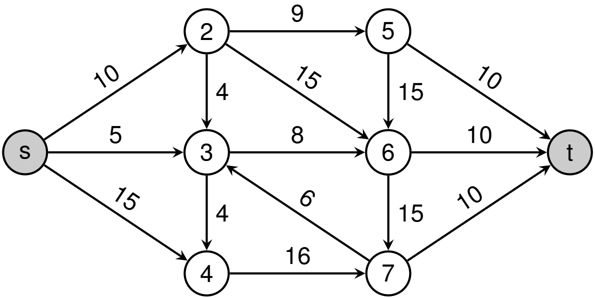
Legend: ——·—— International boundary ·········· Regional boundaries of the USSR (they are included as a matter of general information)

(7) Operating divisions. Those located in Russia are believed to be accurately located. Some Russian divisions (2, 3, 4 and 13) are located in two regions and are so indicated. Divisions shown in the satellites are indicated according to the authors' best judgment, since intelligence reports are unavailable. Train capacities in Russia are for 1000-net-ton trains or their equivalent. Train capacities in Poland are for 666 net tons (or the equivalent). Train capacities in all other satellites are for 400 net tons (or the equivalent) except in East Germany. In East Germany, train capacities are those of entering lines. The numbers shown in boxes are total interdivisional capacities.

Flow Network

Flow Network

- Abstraction for material (one commodity!) flowing through the edges
- $G = (V, E)$ directed graph without parallel edges
- distinguished nodes: source s and sink t
- every edge e has a capacity $c(e)$

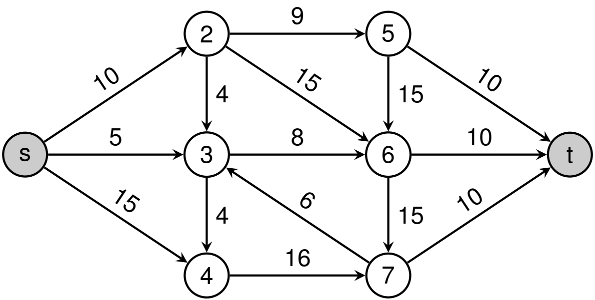


Flow Network

Flow Network

- Abstraction for material (one commodity!) flowing through the edges
- $G = (V, E)$ directed graph without parallel edges
- distinguished nodes: source s and sink t
- every edge e has a capacity $c(e)$

Capacity function $c : V \times V \to \mathbb{R}^+$



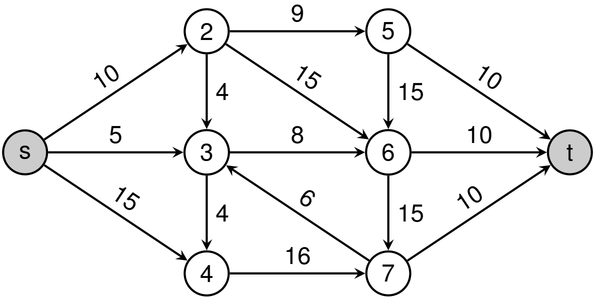
Flow Network

Flow Network

- Abstraction for material (one commodity!) flowing through the edges
- $G = (V, E)$ directed graph without parallel edges
- distinguished nodes: source s and sink t
- every edge e has a capacity $c(e)$

Capacity function $c : V \times V \to \mathbb{R}^+$

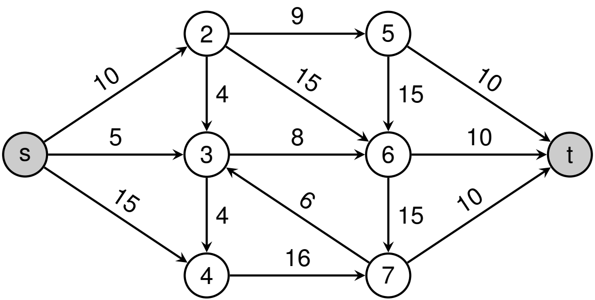
$c(u, v) = 0 \Leftrightarrow (u, v) \notin E$



Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

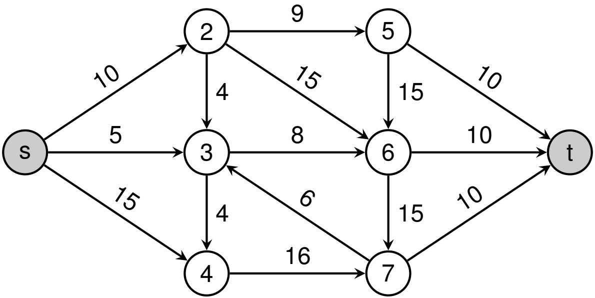


Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$

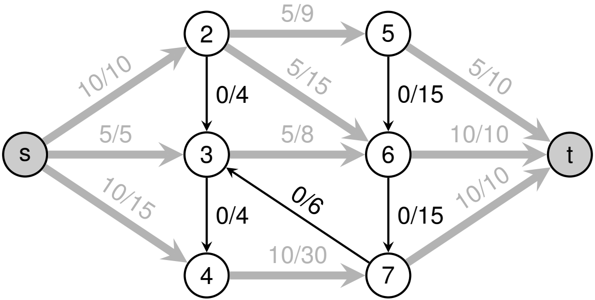


Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \le c(u, v)$

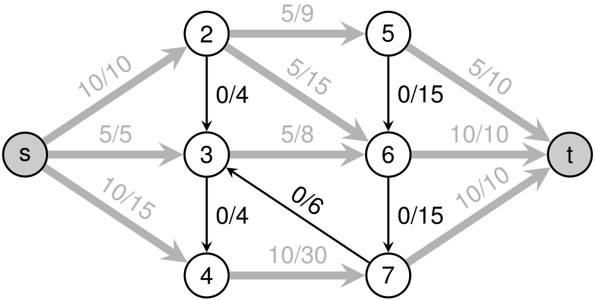


Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$

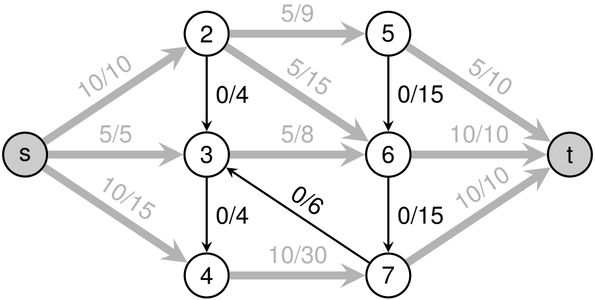


Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$



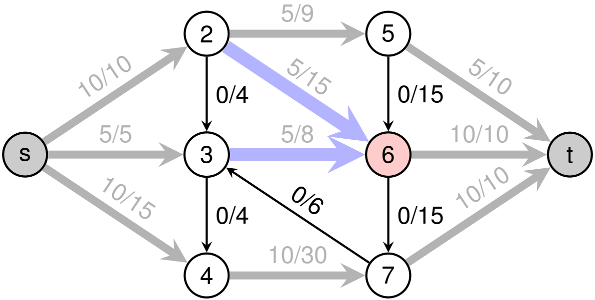
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

Flow Conservation



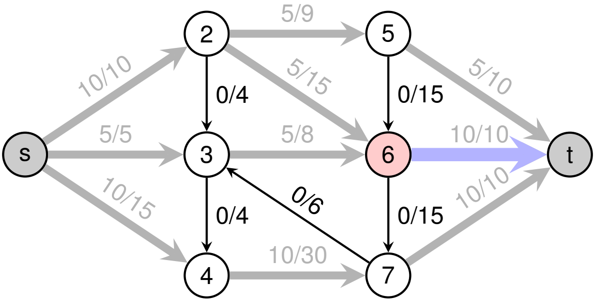
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

Flow Conservation



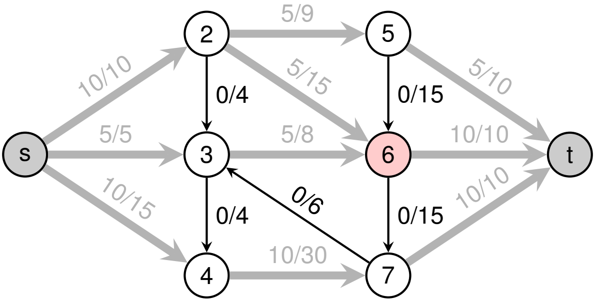
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

Flow Conservation



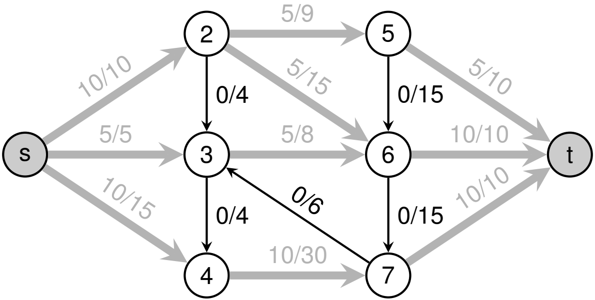
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

Flow Conservation



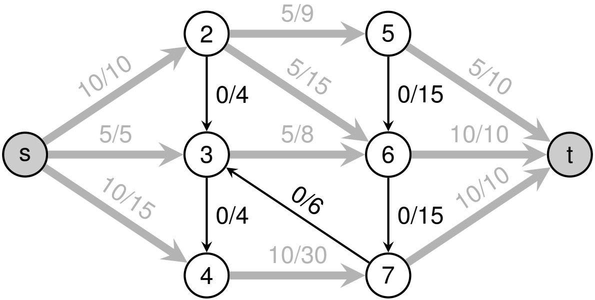
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$



Flow Network

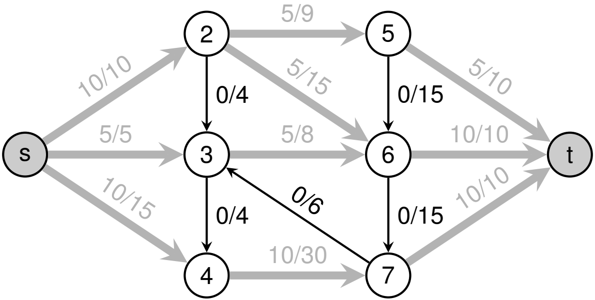
Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u,v) \leq c(u,v)$
- For every $u, v \in V$, $f(u,v) = -f(v,u)$
- For every $u \in V \setminus \{s,t\}$, $\sum_{v \in V} f(u,v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s,v)$

$\sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)$



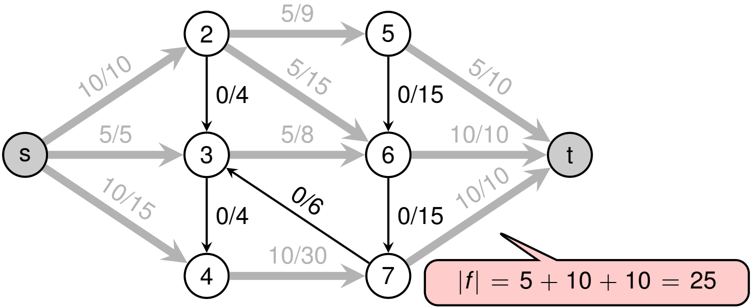
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$



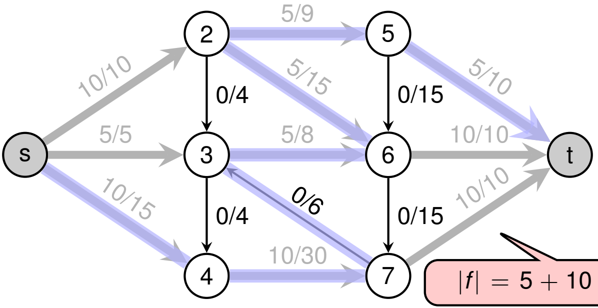
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$



$|f| = 5 + 10 + 10 = 25$

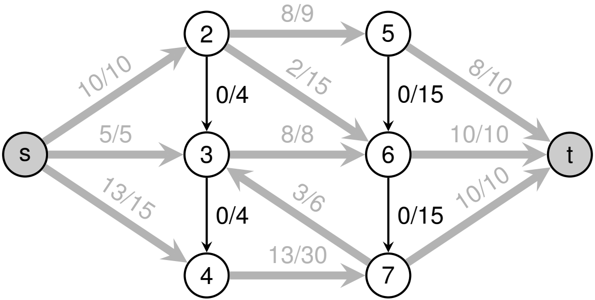
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$



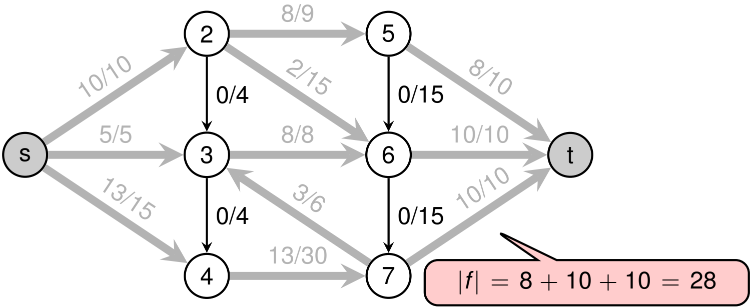
Flow Network

Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$



Flow Network

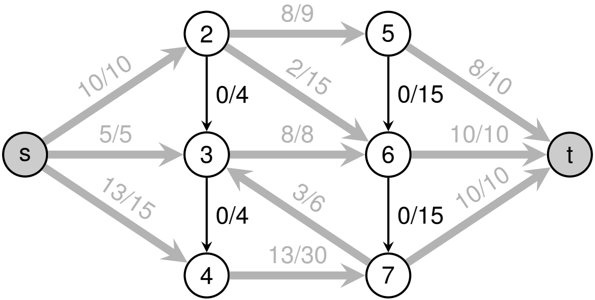
Flow

A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u, v) \leq c(u, v)$
- For every $u, v \in V$, $f(u, v) = -f(v, u)$
- For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$

The value of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$

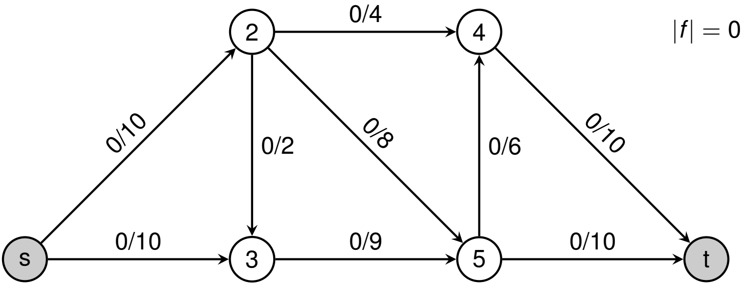
How to find a Maximum Flow?



A First Attempt

Greedy Algorithm

- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p

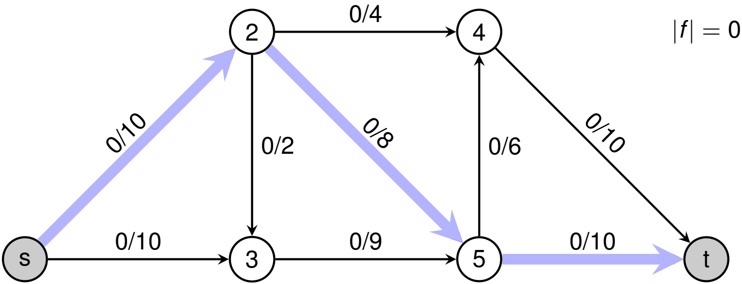


$|f| = 0$

A First Attempt

Greedy Algorithm

- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p

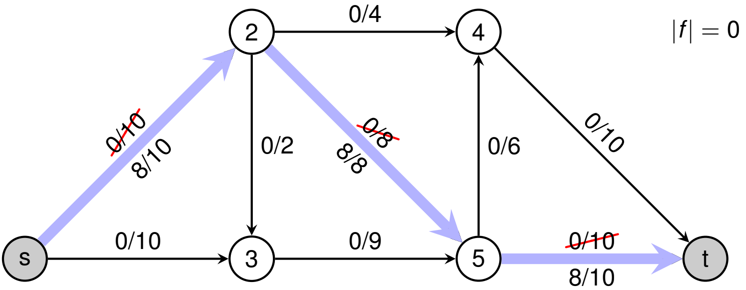


$|f| = 0$

A First Attempt

Greedy Algorithm

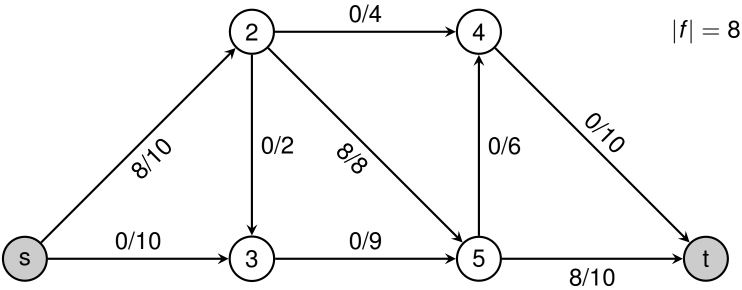
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p

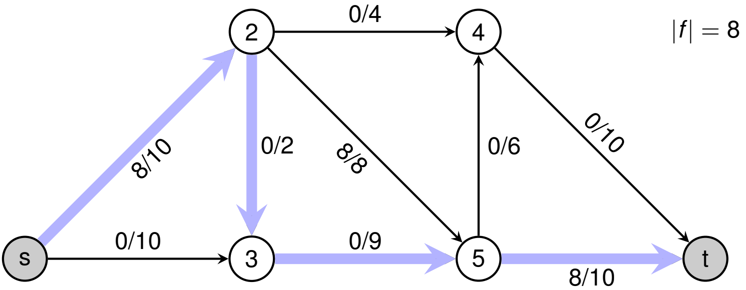


$|f| = 8$

A First Attempt

Greedy Algorithm

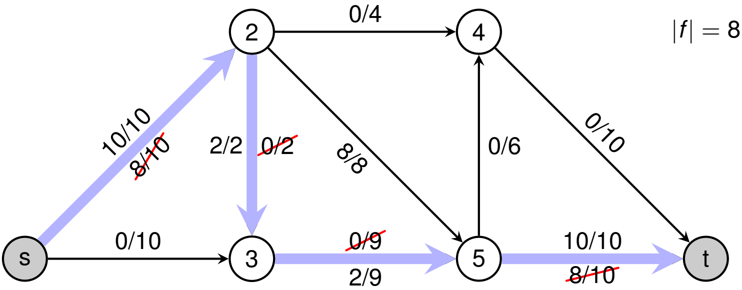
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p

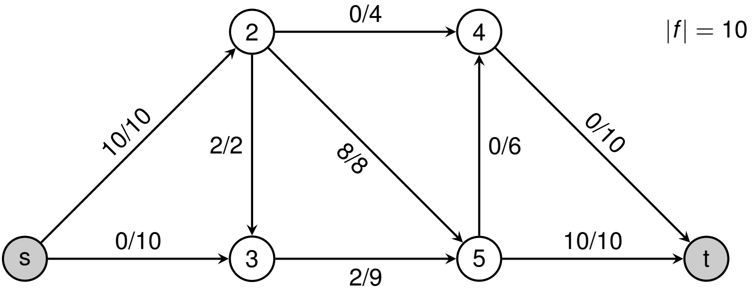


$|f| = 8$

A First Attempt

Greedy Algorithm

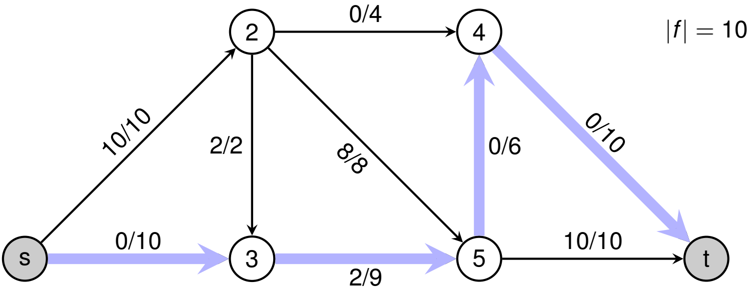
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

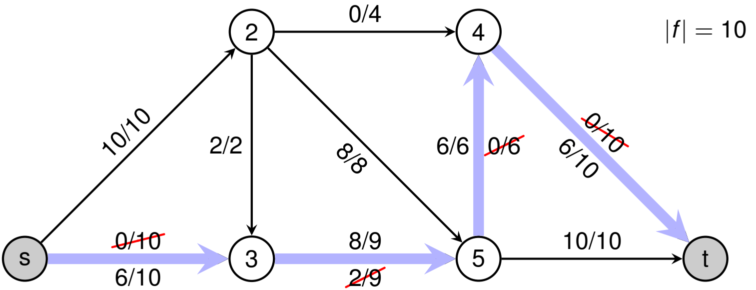
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

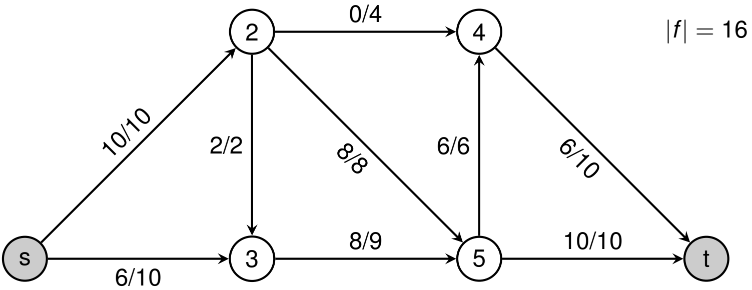
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

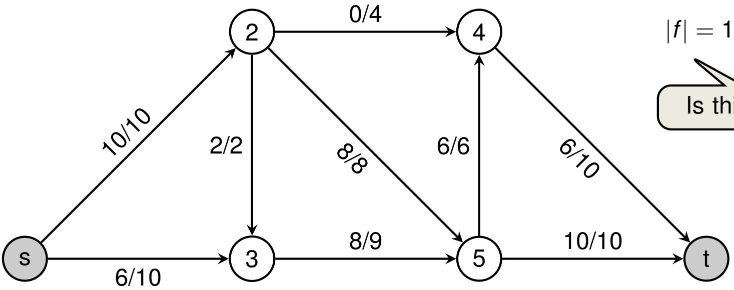
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

- Start with $f(u,v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s,t)-path p where each edge $e = (u,v)$ has $f(u,v) < c(u,v)$
  - Augment flow along p



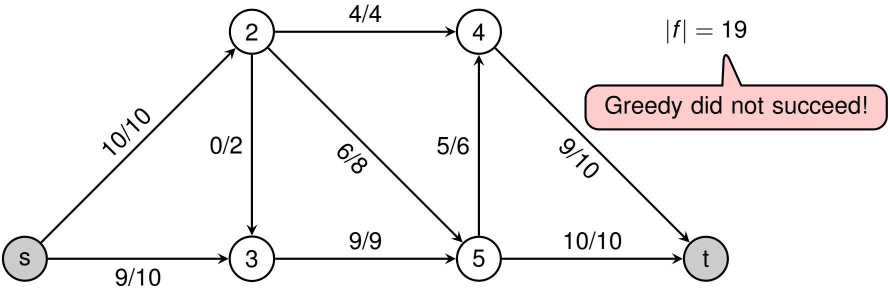
$|f| = 16$

Is this optimal?

A First Attempt

Greedy Algorithm

- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



Outline

Introduction

Ford-Fulkerson

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

Residual Graph

Original Edge

Edge $e = (u, v) \in E$

- flow $f(u, v)$ and capacity $c(u, v)$

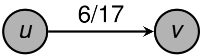
Residual Graph

Original Edge

Edge $e = (u, v) \in E$

- flow $f(u, v)$ and capacity $c(u, v)$

Graph G:



Residual Graph

Original Edge

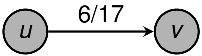
Edge $e = (u, v) \in E$

- flow $f(u, v)$ and capacity $c(u, v)$

Residual Capacity

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E\,, \\ f(v, u) & \text{if } (v, u) \in E\,, \\ 0 & \text{otherwise.} \end{cases}$$

Graph G:



Residual Graph

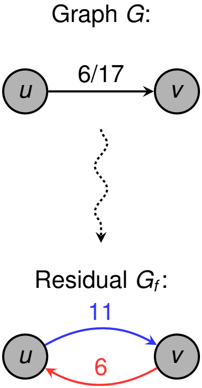
Original Edge

Edge $e = (u, v) \in E$

- flow $f(u, v)$ and capacity $c(u, v)$

Residual Capacity

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E\,, \\ f(v, u) & \text{if } (v, u) \in E\,, \\ 0 & \text{otherwise.} \end{cases}$$



Residual Graph

Original Edge

Edge $e = (u, v) \in E$

- flow $f(u, v)$ and capacity $c(u, v)$

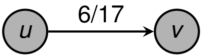
Residual Capacity

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E\,, \\ f(v, u) & \text{if } (v, u) \in E\,, \\ 0 & \text{otherwise.} \end{cases}$$

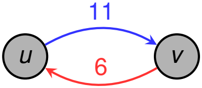
Residual Graph

- $G_f = (V, E_f, c_f)$, $E_f := \{(u, v) \colon c_f(u, v) > 0\}$

Graph G:



Residual G_f:



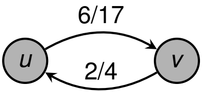
Residual Graph with anti-parallel edges

Original Edge

Edge $e = (u, v) \in E$ (& possibly $e' = (v, u) \in E$)

- flow $f(u, v)$ and capacity $c(u, v)$

Graph G:



Residual Graph with anti-parallel edges

Original Edge

Edge $e = (u, v) \in E$ (& possibly $e' = (v, u) \in E$)

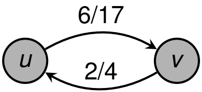
- flow $f(u, v)$ and capacity $c(u, v)$

Residual Capacity

For every pair $(u, v) \in V \times V$,

$$c_f(u, v) = c(u, v) - f(u, v).$$

Graph G:



Residual Graph with anti-parallel edges

Original Edge

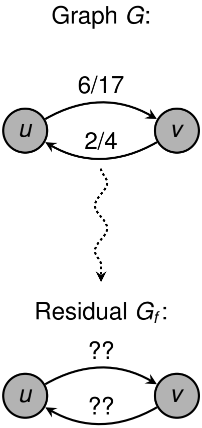
Edge $e = (u, v) \in E$ (& possibly $e' = (v, u) \in E$)

- flow $f(u, v)$ and capacity $c(u, v)$

Residual Capacity

For every pair $(u, v) \in V \times V$,

$$c_f(u, v) = c(u, v) - f(u, v).$$



Residual Graph with anti-parallel edges

Original Edge

Edge $e = (u, v) \in E$ (& possibly $e' = (v, u) \in E$)

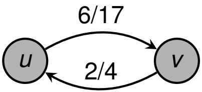
- flow $f(u, v)$ and capacity $c(u, v)$

Residual Capacity

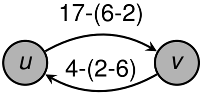
For every pair $(u, v) \in V \times V$,

$$c_f(u, v) = c(u, v) - f(u, v).$$

Graph G:



Residual G_f:



Residual Graph with anti-parallel edges

Original Edge

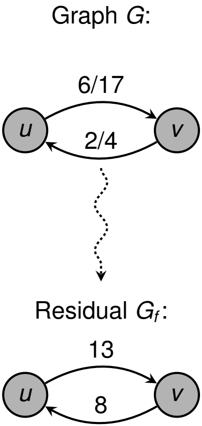
Edge $e = (u, v) \in E$ (& possibly $e' = (v, u) \in E$)

- flow $f(u, v)$ and capacity $c(u, v)$

Residual Capacity

For every pair $(u, v) \in V \times V$,

$$c_f(u, v) = c(u, v) - f(u, v).$$



Residual Graph with anti-parallel edges

Original Edge

Edge $e = (u, v) \in E$ (& possibly $e' = (v, u) \in E$)

- flow $f(u, v)$ and capacity $c(u, v)$

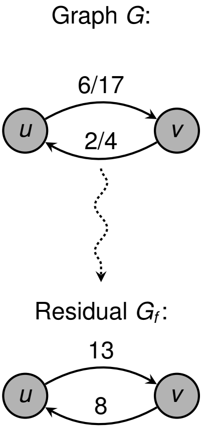
Residual Capacity

For every pair $(u, v) \in V \times V$,

$$c_f(u, v) = c(u, v) - f(u, v).$$

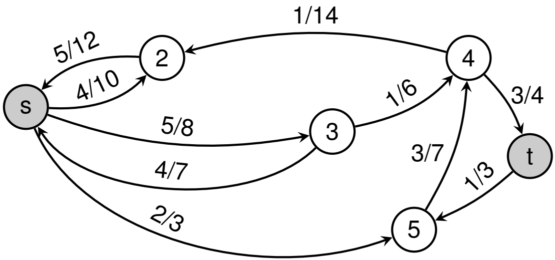
Residual Graph

- $G_f = (V, E_f, c_f)$, $E_f := \{(u, v) : c_f(u, v) > 0\}$

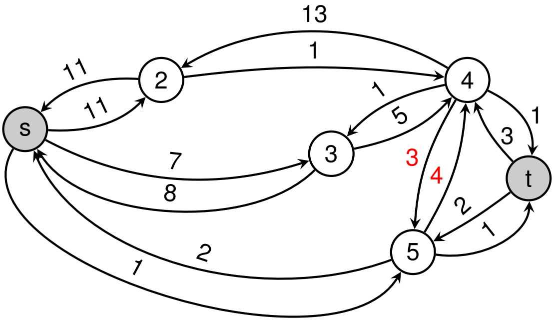


Example of a Residual Graph (Handout)

Flow network G

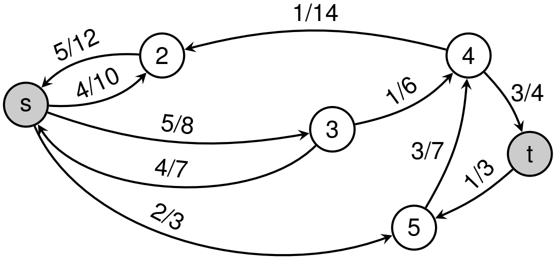


Residual Graph G_f



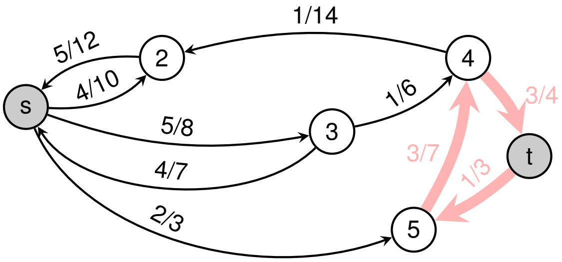
Example of a Residual Graph (Handout)

Flow network G



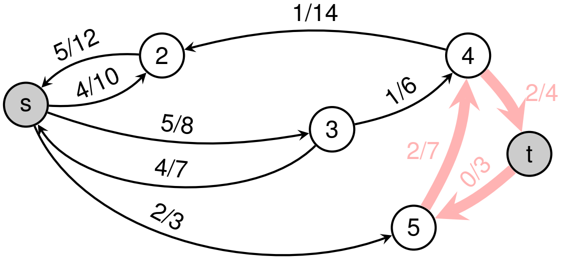
Example of a Residual Graph (Handout)

Flow network G



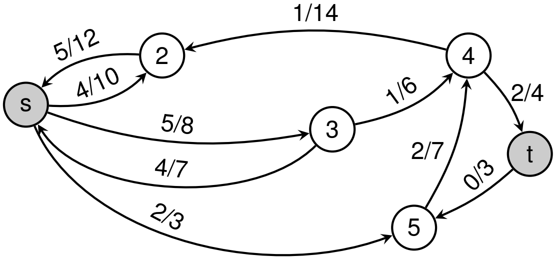
Example of a Residual Graph (Handout)

Flow network G



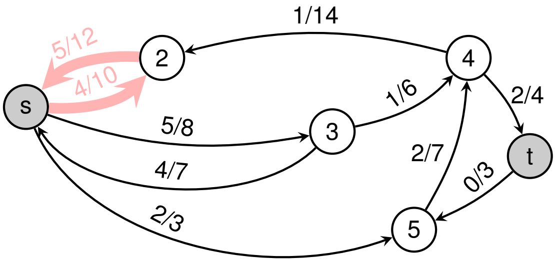
Example of a Residual Graph (Handout)

Flow network G



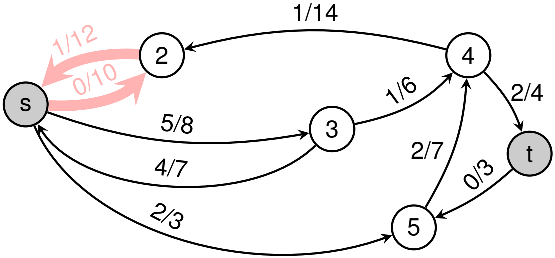
Example of a Residual Graph (Handout)

Flow network G



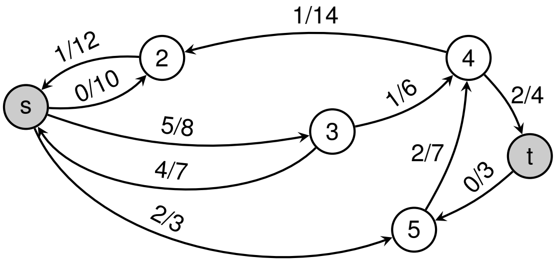
Example of a Residual Graph (Handout)

Flow network G



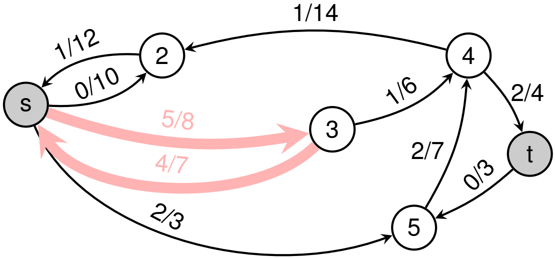
Example of a Residual Graph (Handout)

Flow network G



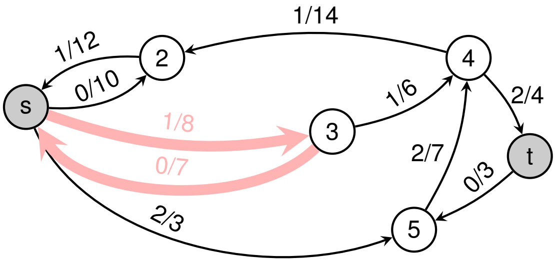
Example of a Residual Graph (Handout)

Flow network G



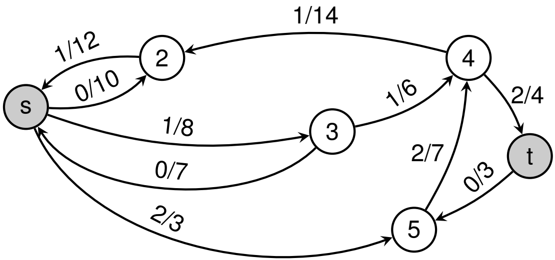
Example of a Residual Graph (Handout)

Flow network G



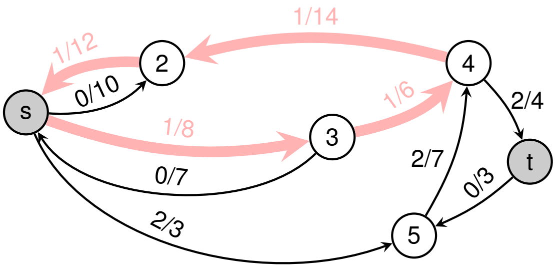
Example of a Residual Graph (Handout)

Flow network G



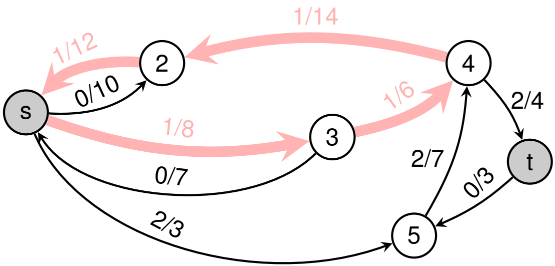
Example of a Residual Graph (Handout)

Flow network G



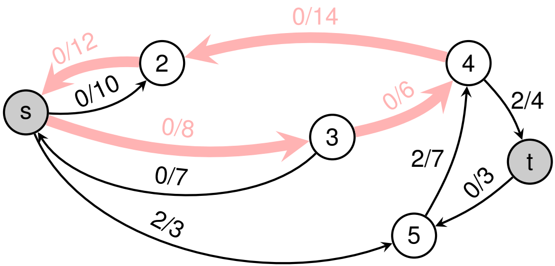
Example of a Residual Graph (Handout)

Flow network G



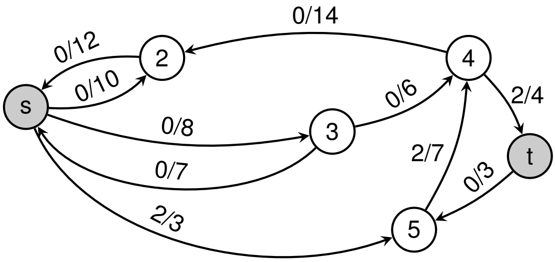
Example of a Residual Graph (Handout)

Flow network G



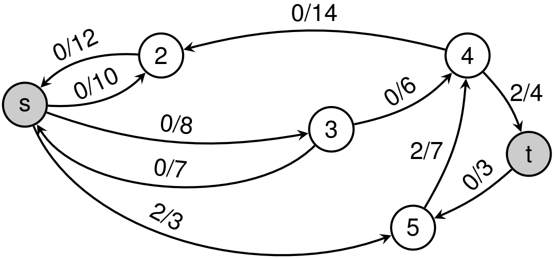
Example of a Residual Graph (Handout)

Flow network G



Example of a Residual Graph (Handout)

Flow network G



By successively eliminating cycles we can simplify and reduce the "transportation" cost of a flow.

The Ford-Fulkerson Method (“Enhanced Greedy”)

```
0: def fordFulkerson(G)
1:    initialize flow to 0 on all edges
2:    while an augmenting path in G_f can be found:
3:       push as much extra flow as possible through it
```

The Ford-Fulkerson Method ("Enhanced Greedy")

```
def fordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Augmenting path: Path from source to sink in G_f

The Ford-Fulkerson Method (“Enhanced Greedy”)

```
def fordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

If f' is a flow in G_f and f a flow in G, then $f + f'$ is a flow in G

The Ford-Fulkerson Method (“Enhanced Greedy”)

```
0: def fordFulkerson(G)
1:     initialize flow to 0 on all edges
2:     while an augmenting path in G_f can be found:
3:         push as much extra flow as possible through it
```

Questions:

- How to find an augmenting path?
- Does this method terminate?
- If it terminates, how good is the solution?

The Ford-Fulkerson Method ("Enhanced Greedy")

```
def fordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

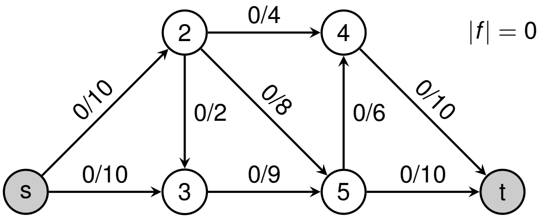
Using BFS or DFS, we can find an augmenting path in $O(V + E)$ time.

Questions:

- How to find an augmenting path?
- Does this method terminate?
- If it terminates, how good is the solution?

Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

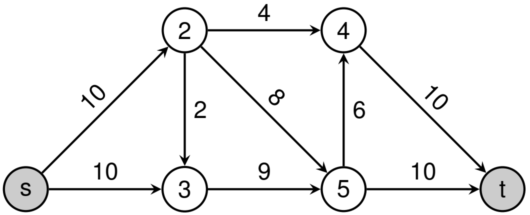
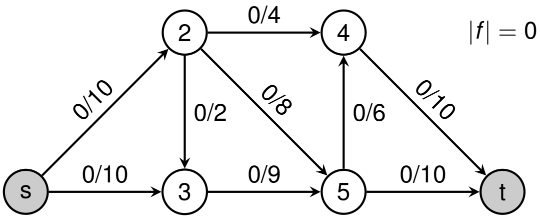


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

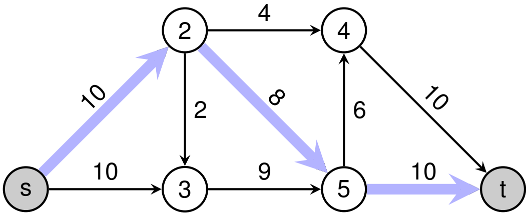
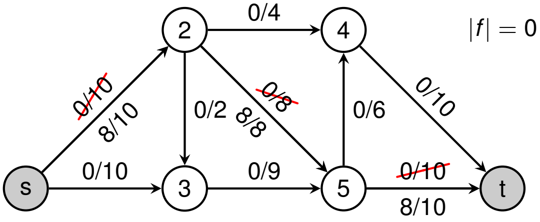


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

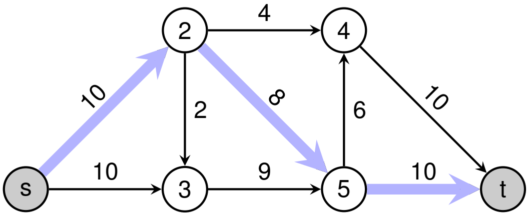
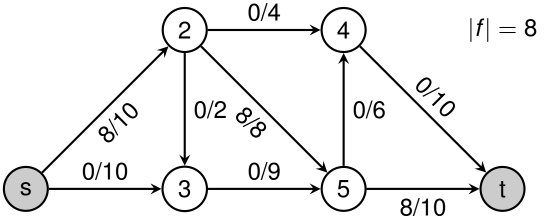


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

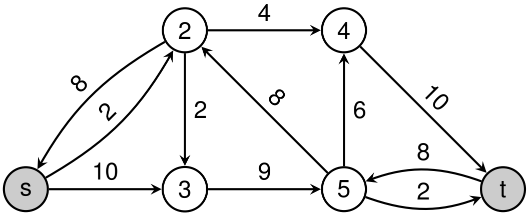
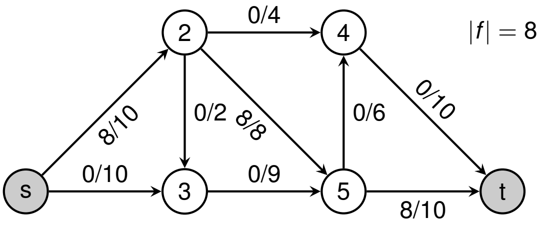


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

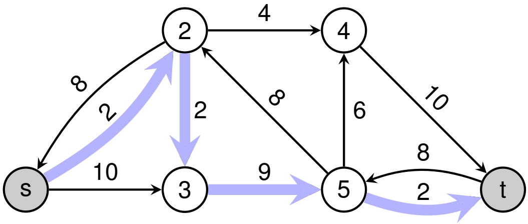
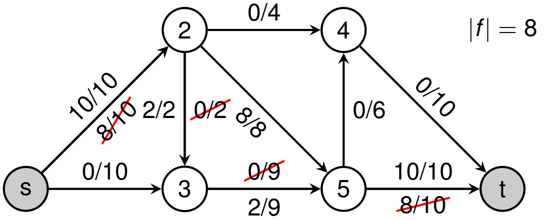


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

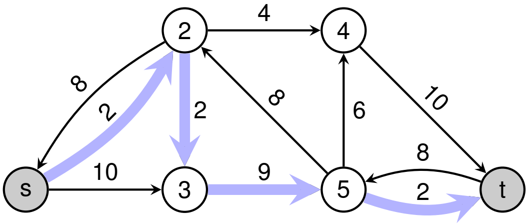
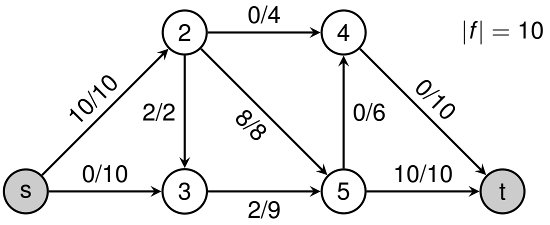


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

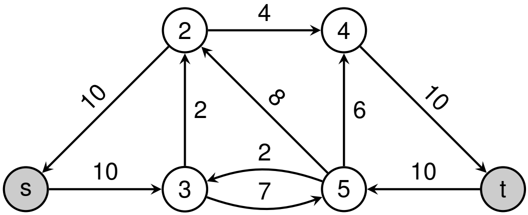
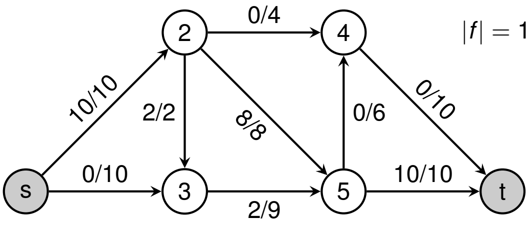


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



$|f| = 10$

Residual Graph $G_f = (V, E_f, c_f)$**:**

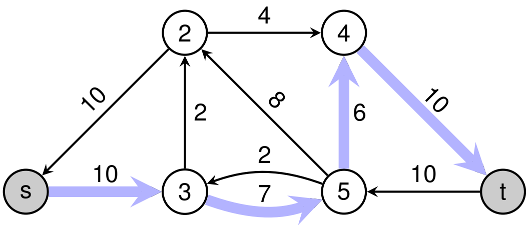
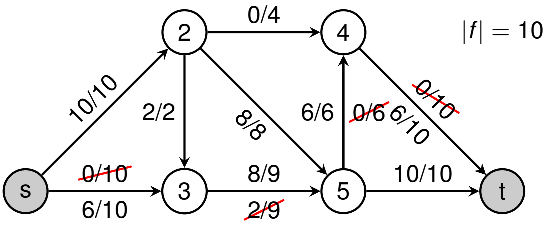


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

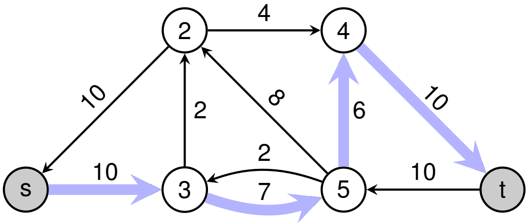
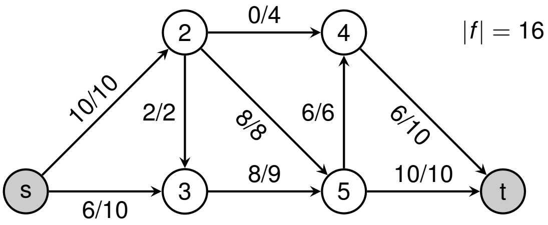


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

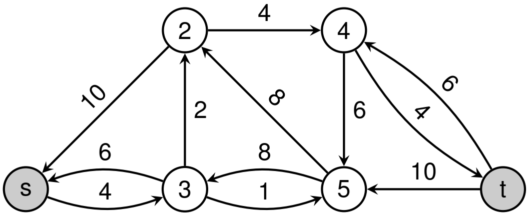
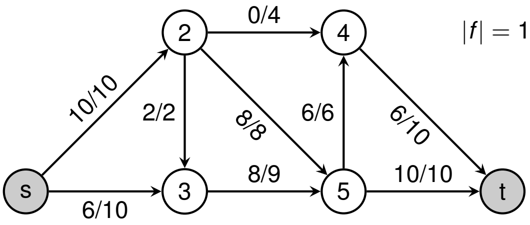


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



$|f| = 16$

Residual Graph $G_f = (V, E_f, c_f)$**:**

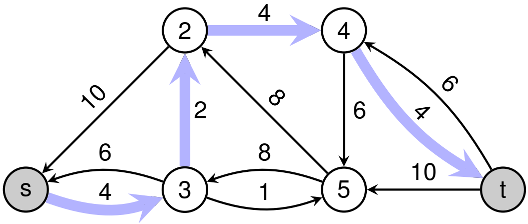
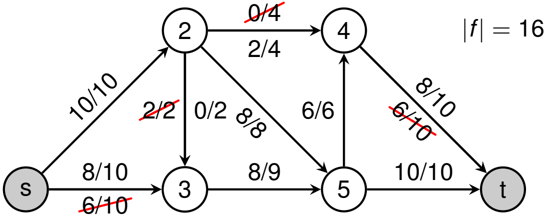


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

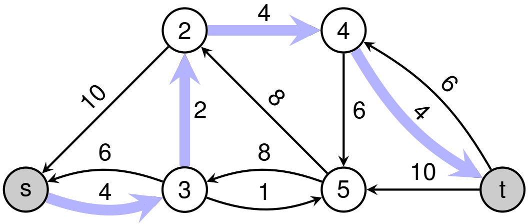
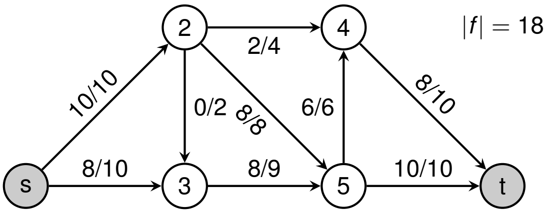


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

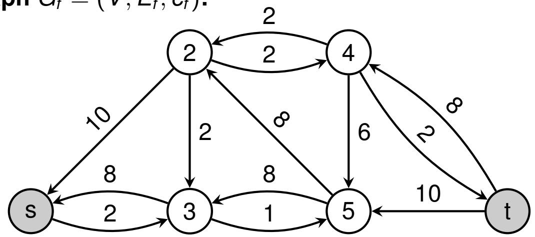
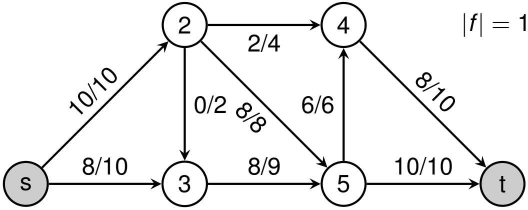


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



$|f| = 18$

Residual Graph $G_f = (V, E_f, c_f)$**:**

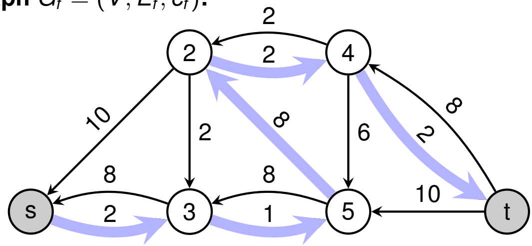
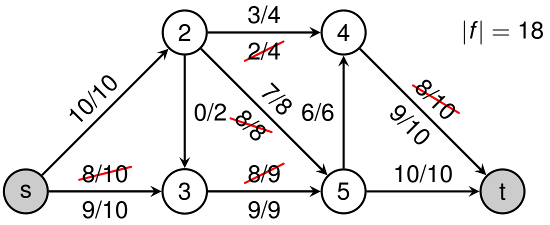


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

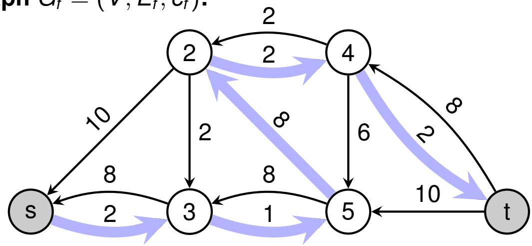
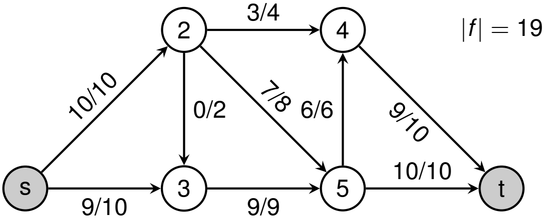


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

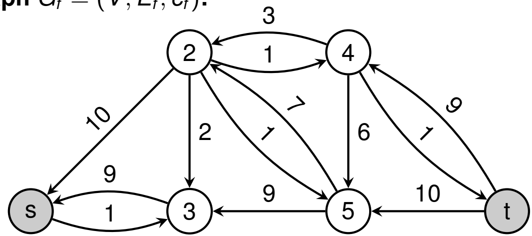
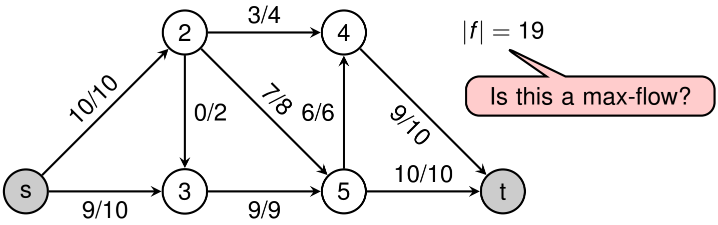


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

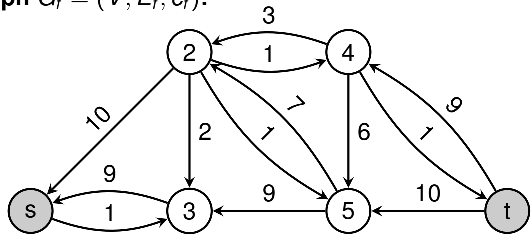
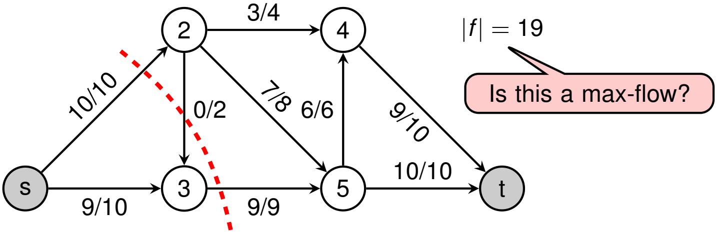


Illustration of the Ford-Fulkerson Method

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

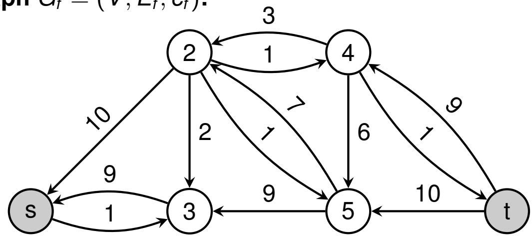
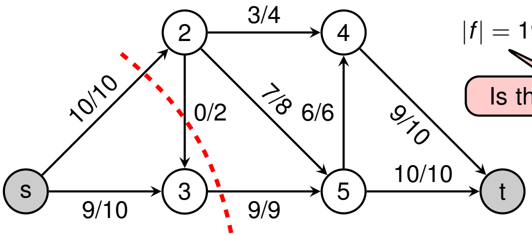


Illustration of the Ford-Fulkerson Method

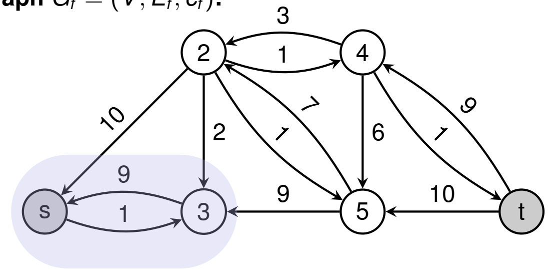
Graph $G = (V, E, c)$**:**



$|f| = 19$

Is this a max-flow?

Residual Graph $G_f = (V, E_f, c_f)$**:**



Outline

Introduction

Ford-Fulkerson

A Glimpse at the Max-Flow Min-Cut Theorem

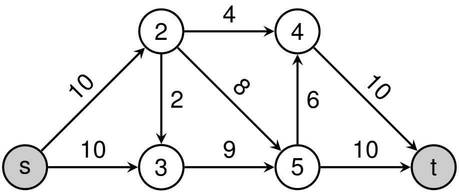
Analysis of Ford-Fulkerson

From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

Graph $G = (V, E, c)$**:**

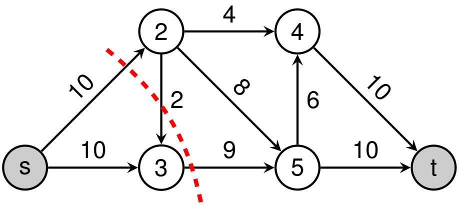


From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

Graph $G = (V, E, c)$**:**



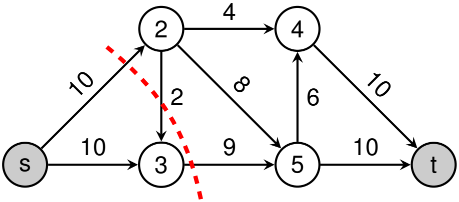
From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of capacities of the edges from S to T:

$$\mathrm{c}(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{(u,v) \in E(S,T)} c(u, v)$$

Graph $G = (V, E, c)$**:**



$\mathrm{c}(\{s, 3\}, \{2, 4, 5, t\}) =$

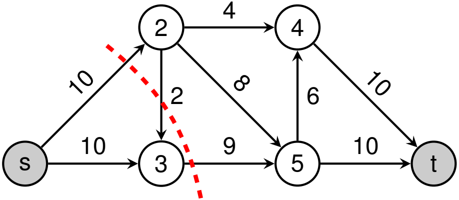
From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of capacities of the edges from S to T:

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{(u,v) \in E(S,T)} c(u, v)$$

Graph $G = (V, E, c)$**:**



$c(\{s, 3\}, \{2, 4, 5, t\}) = 10 + 9 = 19$

From Flows to Cuts

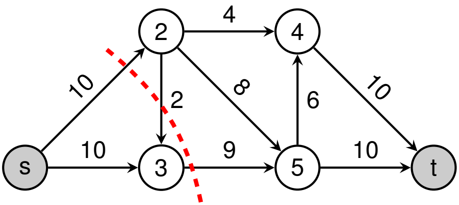
Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of capacities of the edges from S to T:

$$\mathrm{c}(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{(u,v) \in E(S,T)} c(u, v)$$

- A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Graph $G = (V, E, c)$**:**



$$\mathrm{c}(\{s, 3\}, \{2, 4, 5, t\}) = 10 + 9 = 19$$

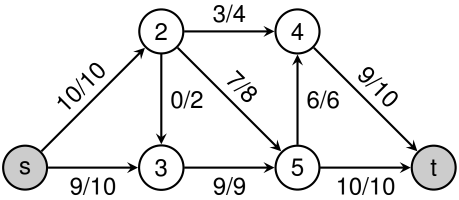
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



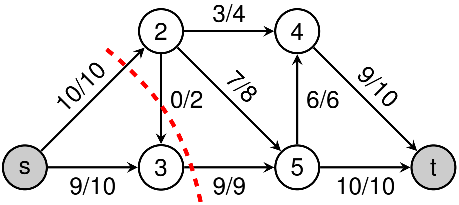
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



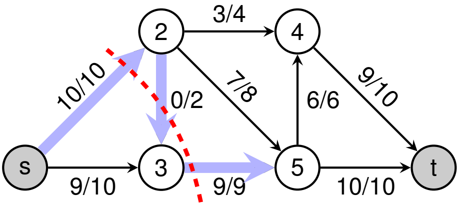
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



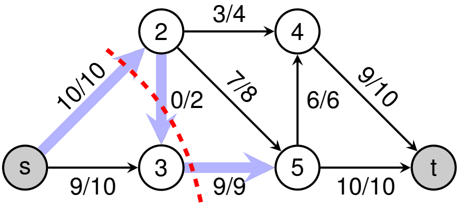
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



$10 - 0 + 9 = 19$

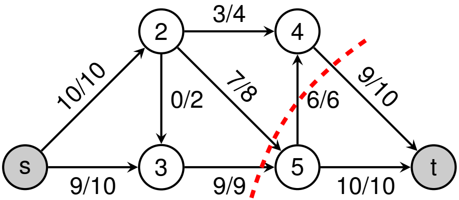
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S, T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



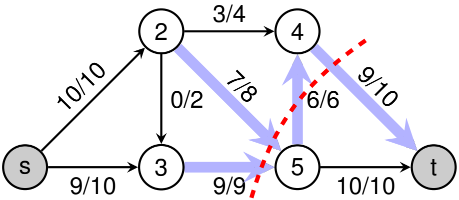
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$: $|f| = 19$



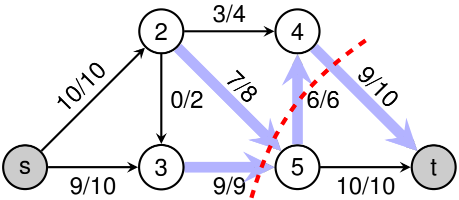
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



$9 + 7 - 6 + 9 = 19$

Outline

Introduction

Ford-Fulkerson

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Analysis of Ford-Fulkerson

```
0: def FordFulkerson(G)
1:     initialize flow to 0 on all edges
2:     while an augmenting path in G_f can be found:
3:         push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Flow before iteration integral
& capacities in G_f are integral
$\Rightarrow$ Flow after iteration integeral

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Theorem

For integral capacities $c(u, v)$, Ford-Fulkerson terminates after $C := \max_{u,v} c(u, v)$ iterations and returns the maximum flow.

Analysis of Ford-Fulkerson

```
0: def FordFulkerson(G)
1:     initialize flow to 0 on all edges
2:     while an augmenting path in G_f can be found:
3:         push as much extra flow as possible through it
```

Lemma

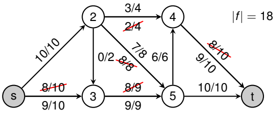
If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Theorem

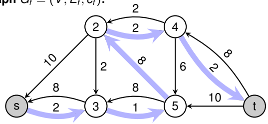
For integral capacities $c(u, v)$, Ford-Fulkerson terminates after $C := \max_{u,v} c(u, v)$ iterations and returns the maximum flow.

(proof omitted here, see CLRS3)

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**



6.6: Maximum flow

Frank Stajano

Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

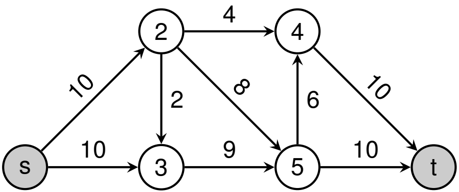
Matchings in Bipartite Graphs

From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

Graph $G = (V, E, c)$**:**

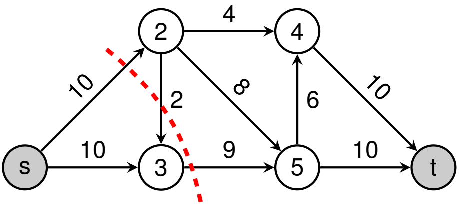


From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

Graph $G = (V, E, c)$**:**



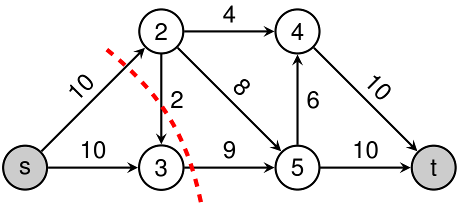
From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of capacities of the edges from S to T:

$$\mathrm{c}(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{(u,v) \in E(S,T)} c(u, v)$$

Graph $G = (V, E, c)$**:**



$\mathrm{c}(\{s, 3\}, \{2, 4, 5, t\}) =$

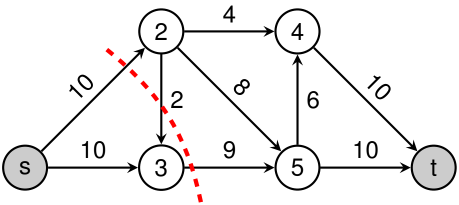
From Flows to Cuts

Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of capacities of the edges from S to T:

$$\mathrm{c}(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{(u,v) \in E(S,T)} c(u, v)$$

Graph $G = (V, E, c)$**:**



$\mathrm{c}(\{s, 3\}, \{2, 4, 5, t\}) = 10 + 9 = 19$

From Flows to Cuts

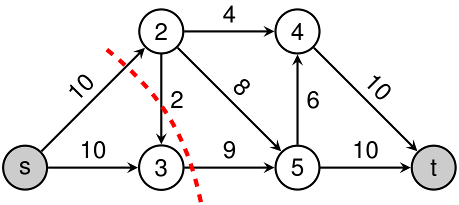
Cut

- A cut (S, T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of capacities of the edges from S to T:

$$\mathrm{c}(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{(u,v) \in E(S,T)} c(u, v)$$

- A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Graph $G = (V, E, c)$**:**



$\mathrm{c}(\{s, 3\}, \{2, 4, 5, t\}) = 10 + 9 = 19$

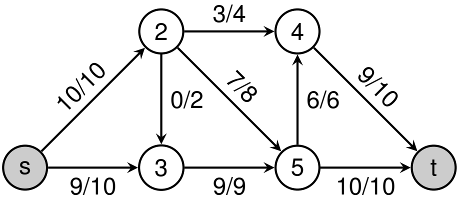
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



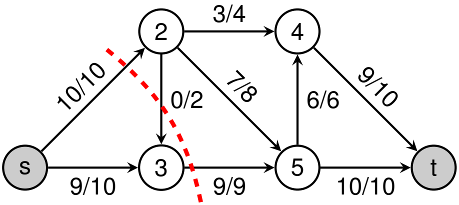
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$

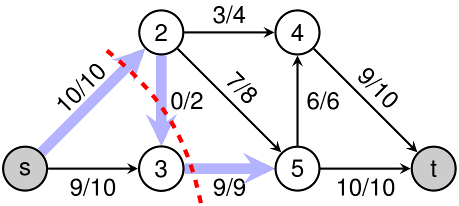


Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



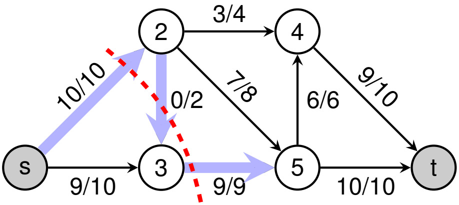
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$: $|f| = 19$



$10 - 0 + 9 = 19$

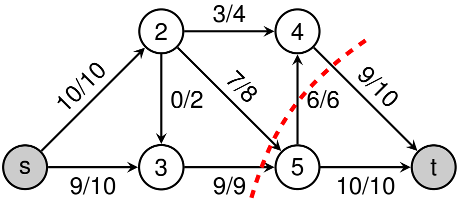
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$



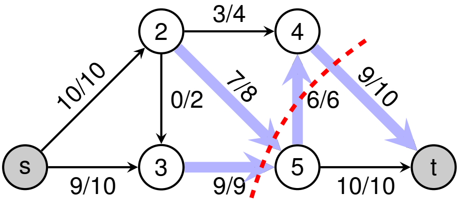
From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Graph $G = (V, E, c)$**:** $|f| = 19$

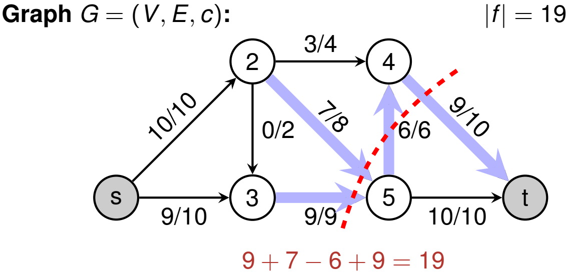


From Flows to Cuts

Theorem (Max-Flow Min-Cut Theorem)

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$



Extra: Proof of the Max-Flow Min-Cut Theorem (Easy Direction)

1. For every $u, v \in V$, $f(u, v) \leq c(u, v)$,
2. For every $u, v \in V$, $f(u, v) = -f(v, u)$,
3. For every $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$.

- Let f be any flow and (S, T) be any cut:

$$\begin{aligned} |f| &= \sum_{v \in V} f(s, v) \\ &\stackrel{(3)}{=} \sum_{u \in S} \sum_{v \in V} f(u, v) \\ &= \sum_{u \in S} \sum_{v \in S} f(u, v) + \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\stackrel{(2)}{=} \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\stackrel{(1)}{\leq} \sum_{u \in S} \sum_{v \in T} c(u, v) \\ &= c(S, T). \end{aligned}$$

Flow-Value-Lemma:
For any cut (S, T),

$$|f| = \sum_{u \in S} \sum_{v \in T} f(u, v).$$

- Since this holds for any pair of flow and cut, it follows that

$$\max_f |f| \leq \min_{(S,T)} c(S, T) \qquad \square$$

Outline

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

Matchings in Bipartite Graphs

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Analysis of Ford-Fulkerson

```
0: def FordFulkerson(G)
1:     initialize flow to 0 on all edges
2:     while an augmenting path in G_f can be found:
3:         push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Flow before iteration integral
& capacities in G_f are integral
$\Rightarrow$ Flow after iteration integral

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
        push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

Theorem

For integral capacities $c(u, v)$, Ford-Fulkerson terminates after $C := \max_{u,v} c(u, v)$ iterations and returns the maximum flow.

Analysis of Ford-Fulkerson

```
0: def FordFulkerson(G)
1:     initialize flow to 0 on all edges
2:     while an augmenting path in G_f can be found:
3:         push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

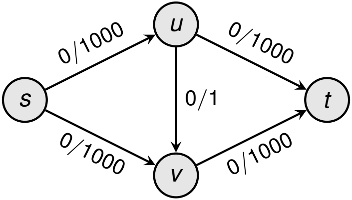
Theorem

For integral capacities $c(u, v)$, Ford-Fulkerson terminates after $C := \max_{u,v} c(u, v)$ iterations and returns the maximum flow.

(proof omitted here, see CLRS3)

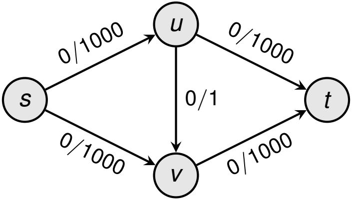
Slow Convergence of Ford-Fulkerson (Figure 26.7)

Slow Convergence of Ford-Fulkerson (Figure 26.7)

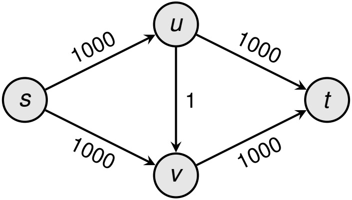


G

Slow Convergence of Ford-Fulkerson (Figure 26.7)

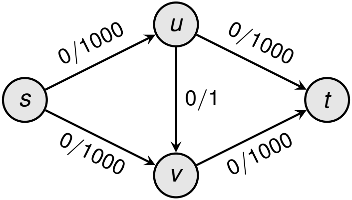


G

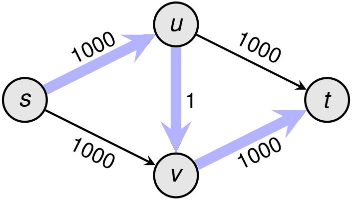


G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)

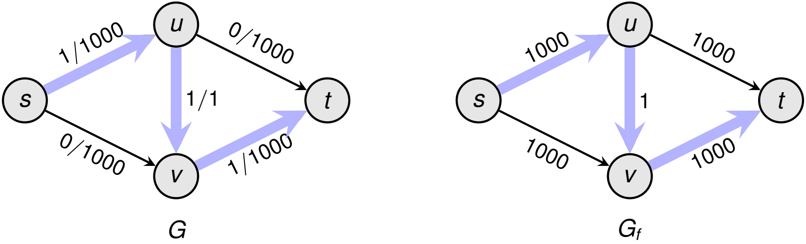


G

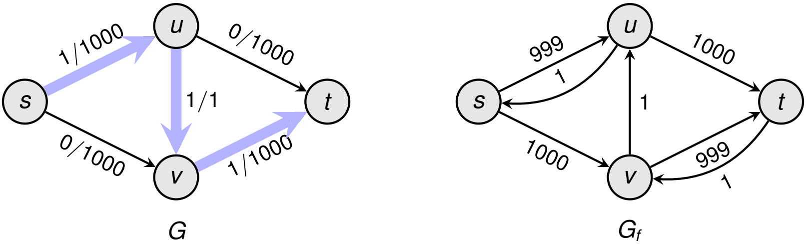


G_f

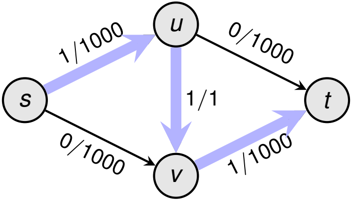
Slow Convergence of Ford-Fulkerson (Figure 26.7)



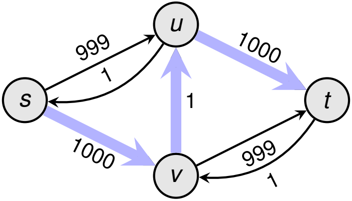
Slow Convergence of Ford-Fulkerson (Figure 26.7)



Slow Convergence of Ford-Fulkerson (Figure 26.7)

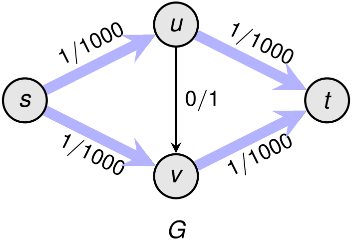


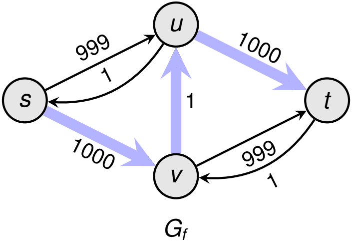
G



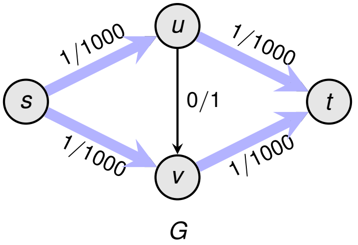
G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)

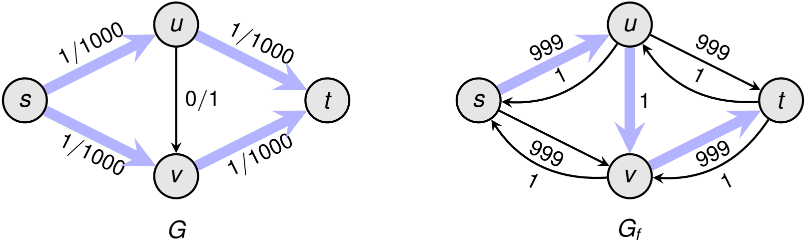




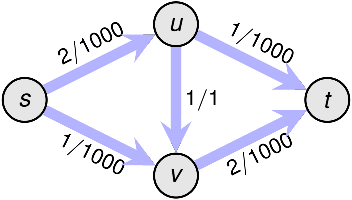
Slow Convergence of Ford-Fulkerson (Figure 26.7)



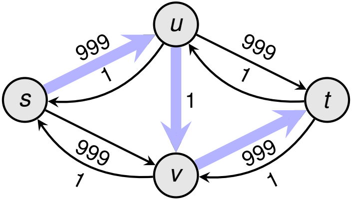
Slow Convergence of Ford-Fulkerson (Figure 26.7)



Slow Convergence of Ford-Fulkerson (Figure 26.7)

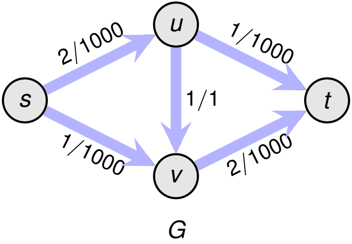


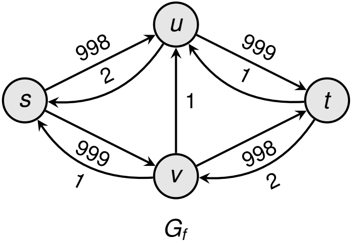
G



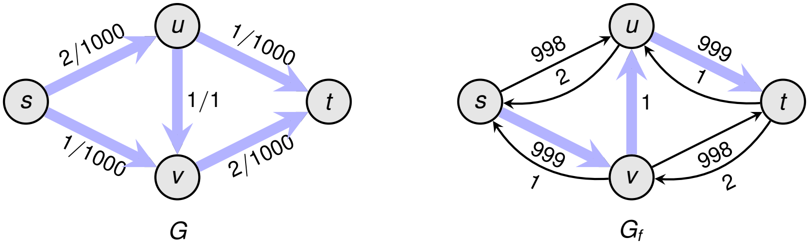
G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)

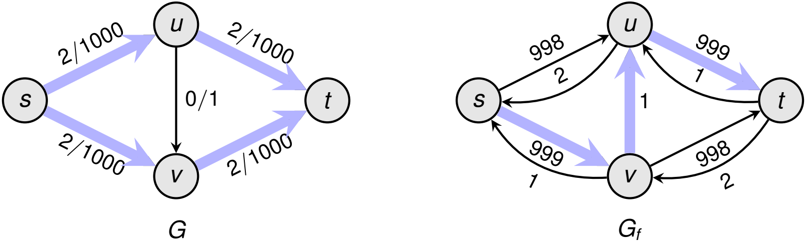




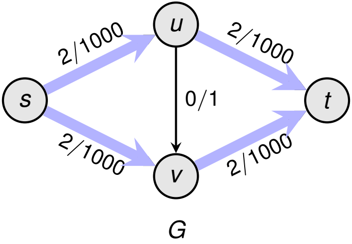
Slow Convergence of Ford-Fulkerson (Figure 26.7)

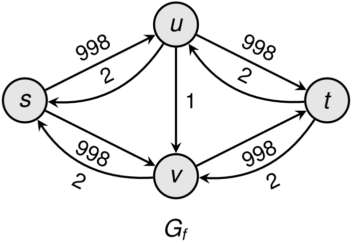


Slow Convergence of Ford-Fulkerson (Figure 26.7)

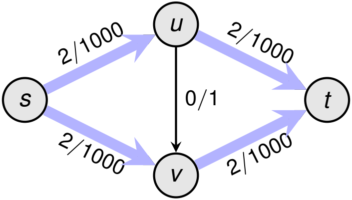


Slow Convergence of Ford-Fulkerson (Figure 26.7)

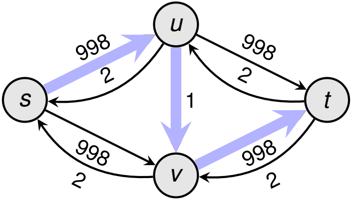




Slow Convergence of Ford-Fulkerson (Figure 26.7)

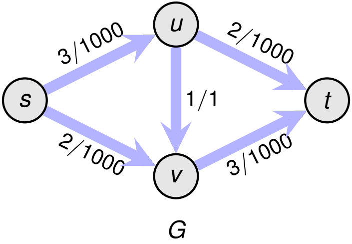


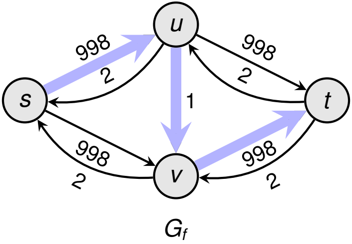
G



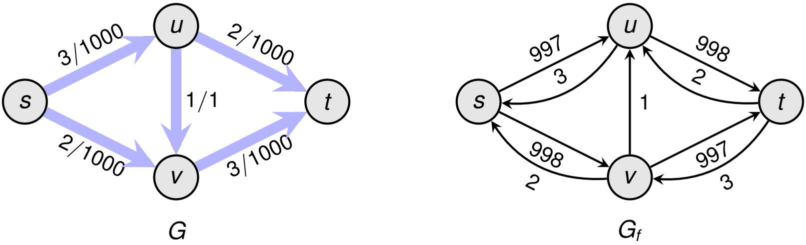
G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)

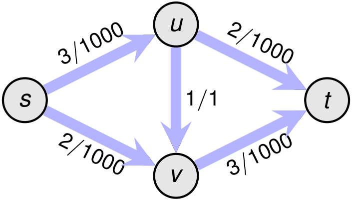




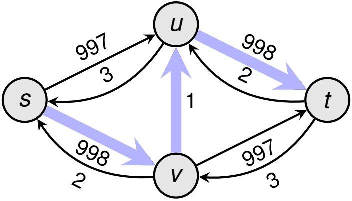
Slow Convergence of Ford-Fulkerson (Figure 26.7)



Slow Convergence of Ford-Fulkerson (Figure 26.7)

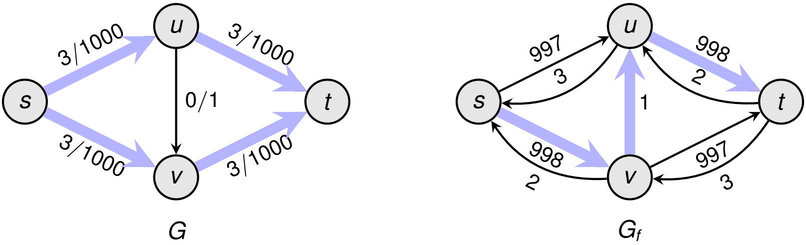


G

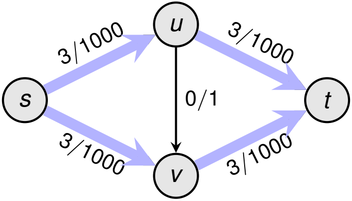


G_f

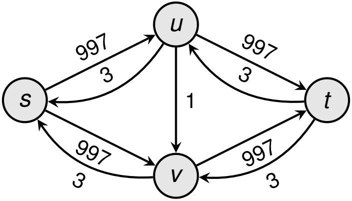
Slow Convergence of Ford-Fulkerson (Figure 26.7)



Slow Convergence of Ford-Fulkerson (Figure 26.7)

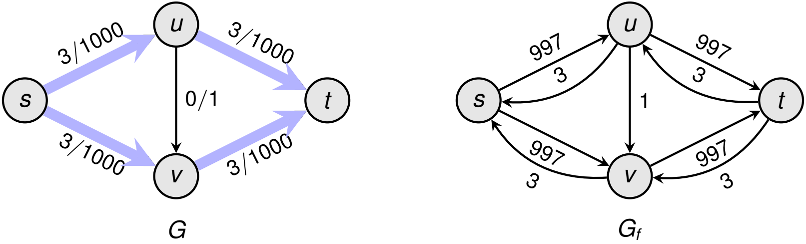


G



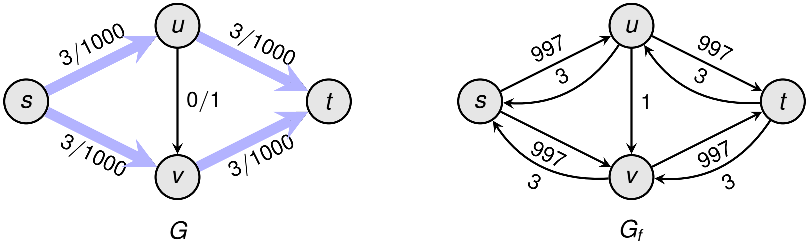
G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)



Number of iterations is $C := \max_{u,v} c(u, v)$!

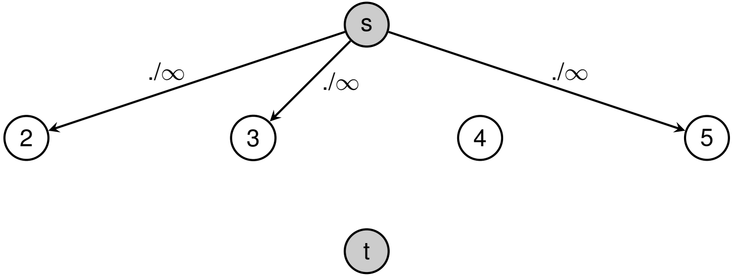
Slow Convergence of Ford-Fulkerson (Figure 26.7)



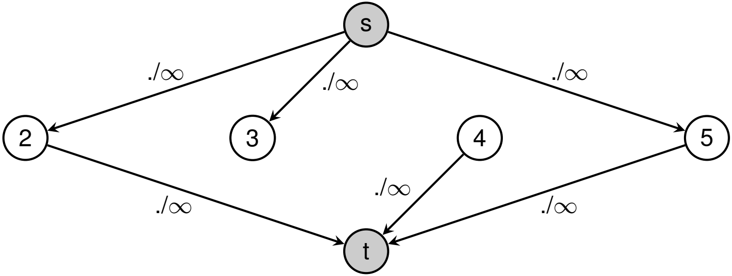
Number of iterations is $C := \max_{u,v} c(u,v)$!

For irrational capacities, Ford-Fulkerson may even fail to terminate!

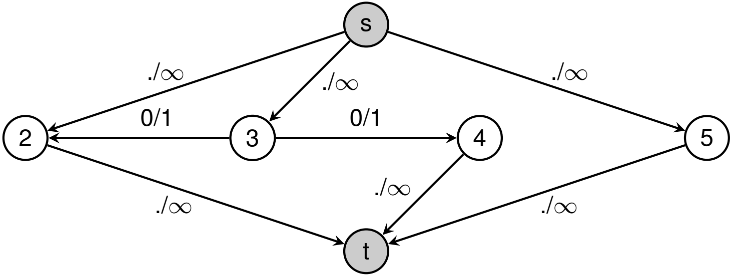
Non-Termination of Ford-Fulkerson for Irrational Capacities



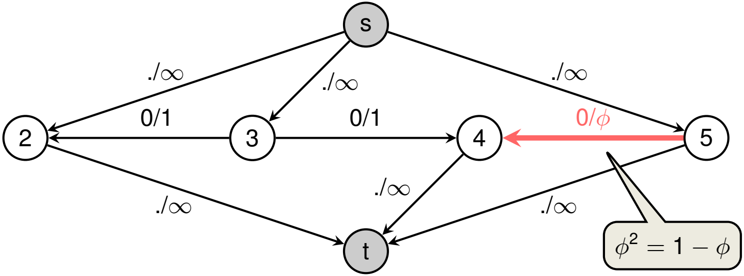
Non-Termination of Ford-Fulkerson for Irrational Capacities



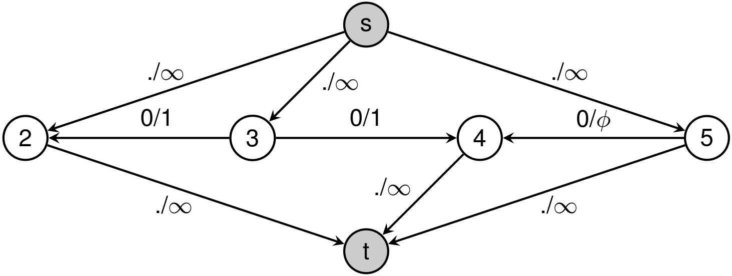
Non-Termination of Ford-Fulkerson for Irrational Capacities



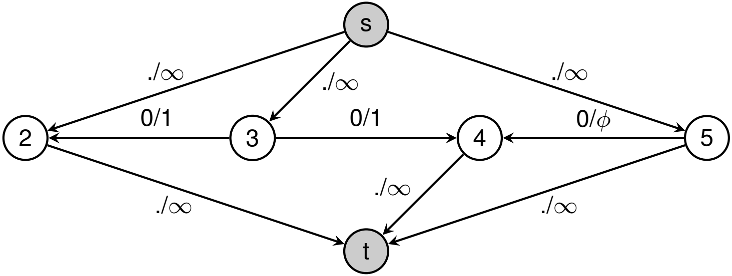
Non-Termination of Ford-Fulkerson for Irrational Capacities

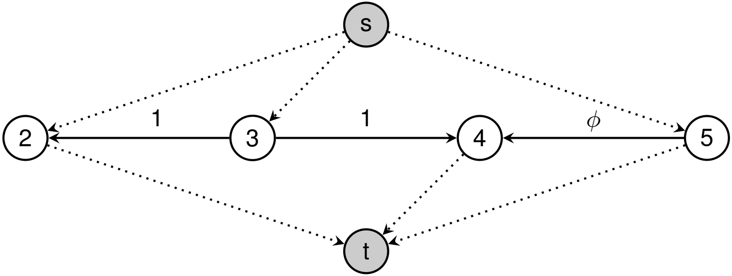


Non-Termination of Ford-Fulkerson for Irrational Capacities

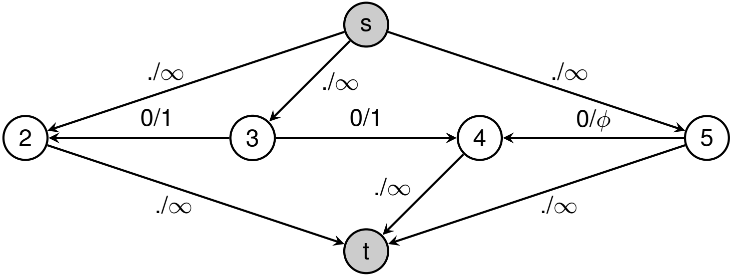


Non-Termination of Ford-Fulkerson for Irrational Capacities

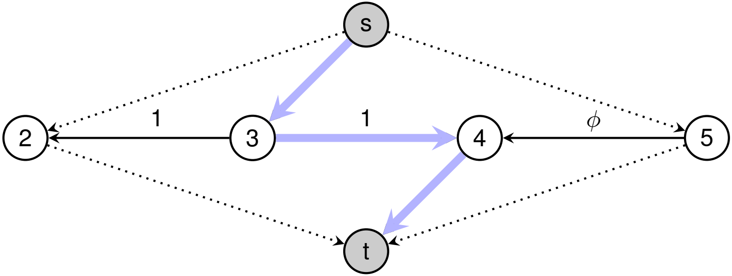




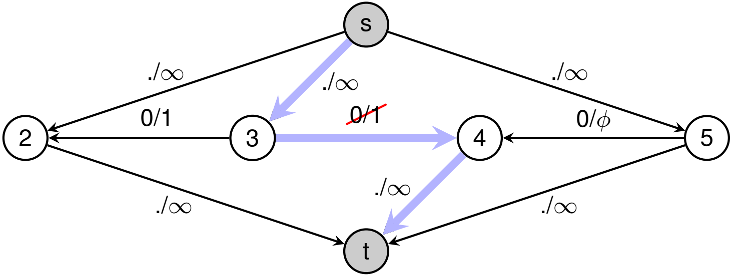
Non-Termination of Ford-Fulkerson for Irrational Capacities



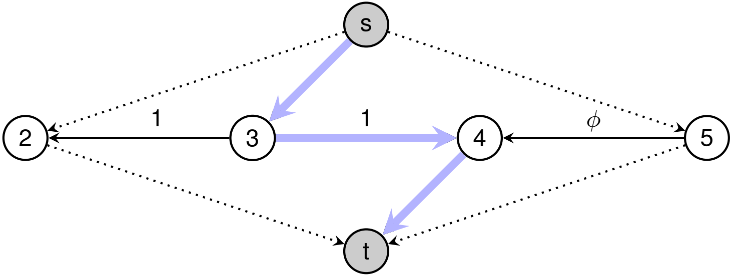
Iteration: 1, $|f| = 0$



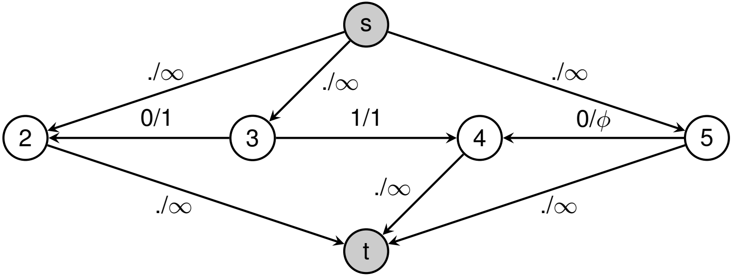
Non-Termination of Ford-Fulkerson for Irrational Capacities



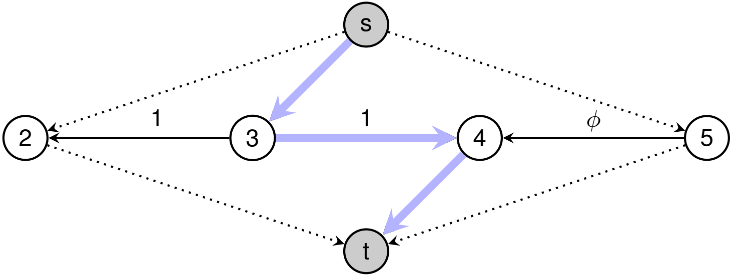
Iteration: 1, $|f| = 0$



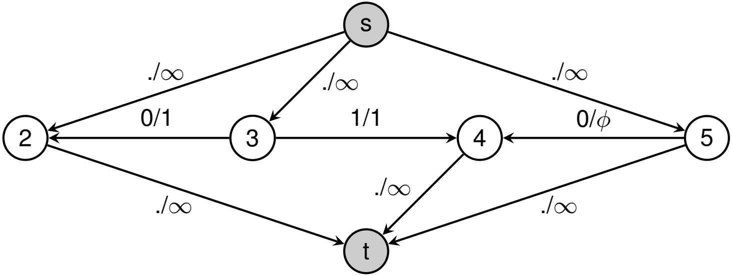
Non-Termination of Ford-Fulkerson for Irrational Capacities



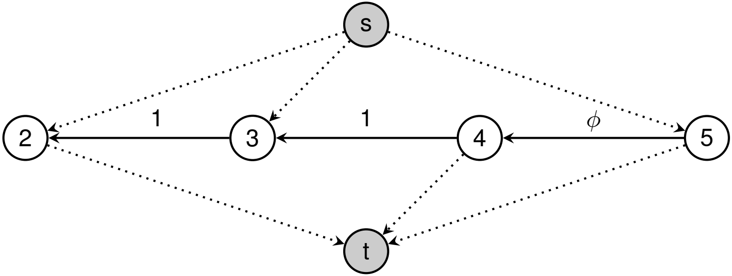
Iteration: 1, $|f| = 1$



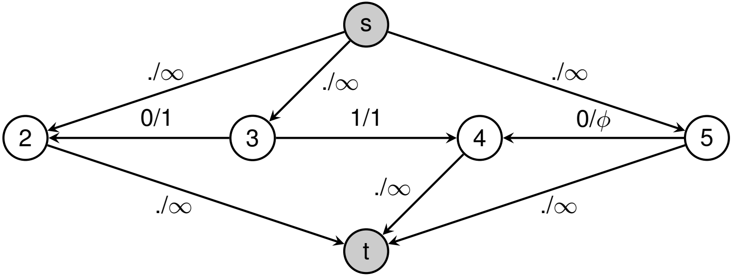
Non-Termination of Ford-Fulkerson for Irrational Capacities



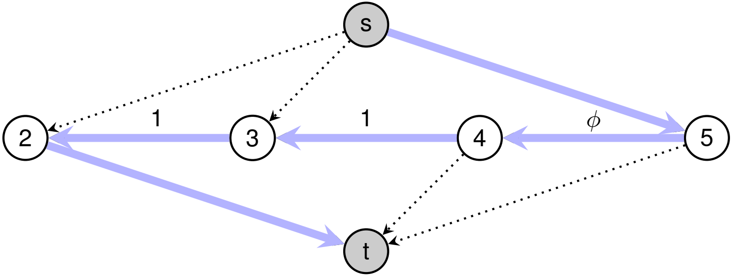
Iteration: 1, $|f| = 1$



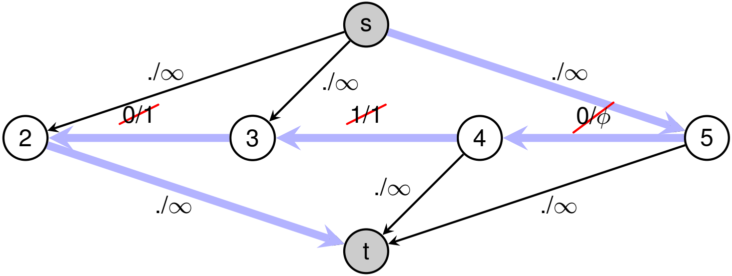
Non-Termination of Ford-Fulkerson for Irrational Capacities



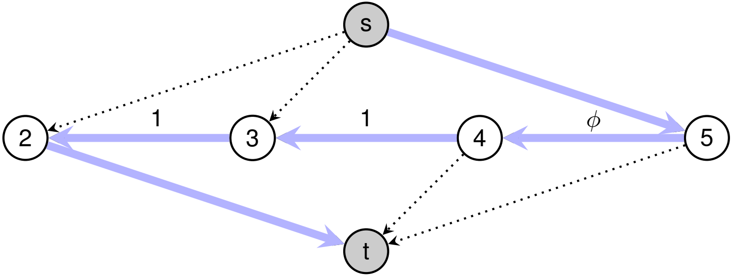
Iteration: 2, $|f| = 1$



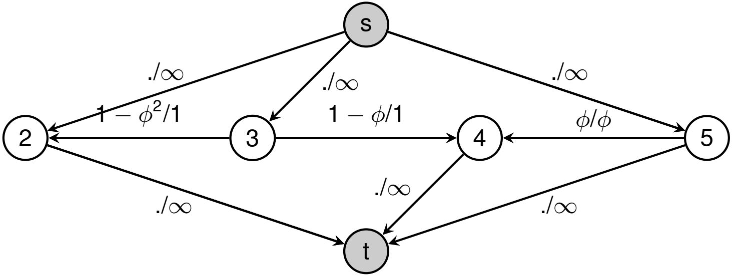
Non-Termination of Ford-Fulkerson for Irrational Capacities



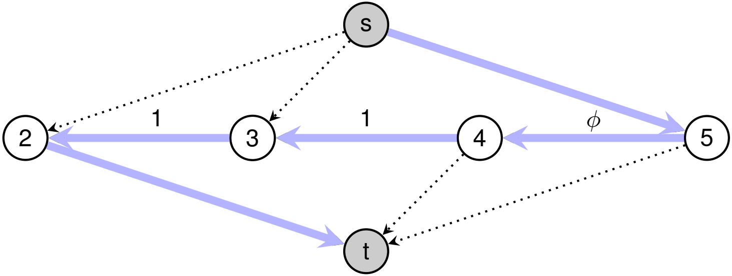
Iteration: 2, $|f| = 1$



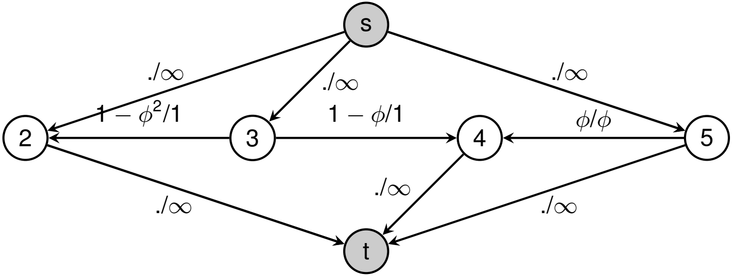
Non-Termination of Ford-Fulkerson for Irrational Capacities



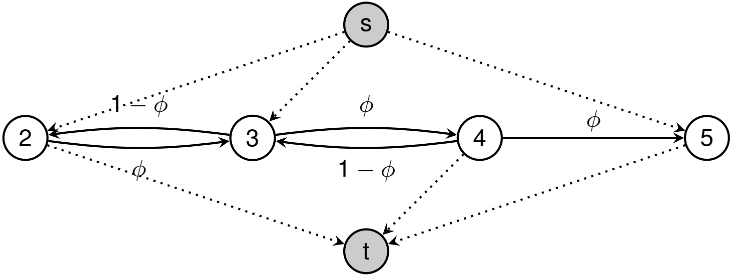
Iteration: 2, $|f| = 1 + \phi$



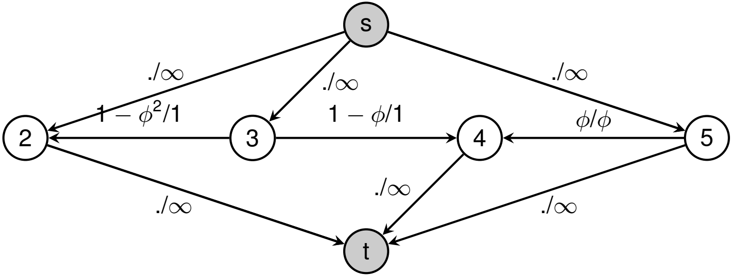
Non-Termination of Ford-Fulkerson for Irrational Capacities



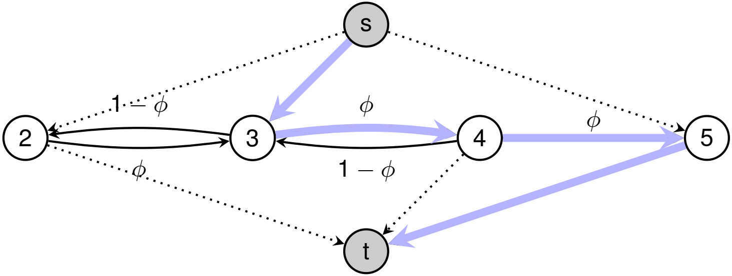
Iteration: 2, $|f| = 1 + \phi$



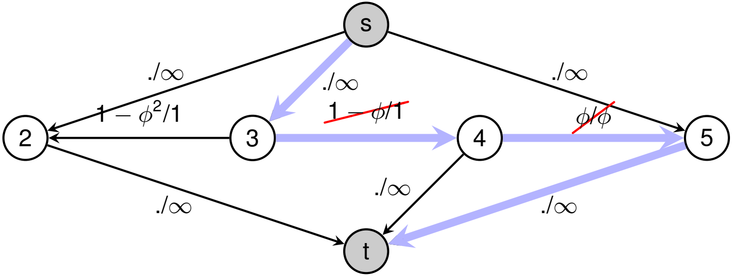
Non-Termination of Ford-Fulkerson for Irrational Capacities



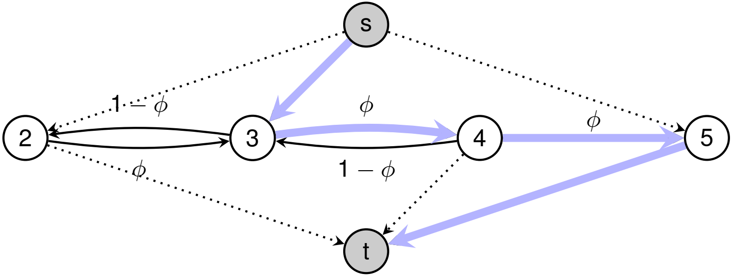
Iteration: 3, $|f| = 1 + \phi$



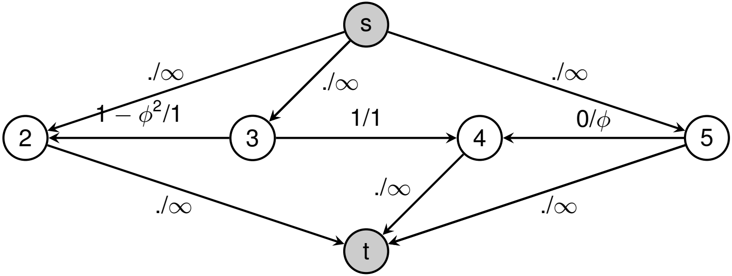
Non-Termination of Ford-Fulkerson for Irrational Capacities



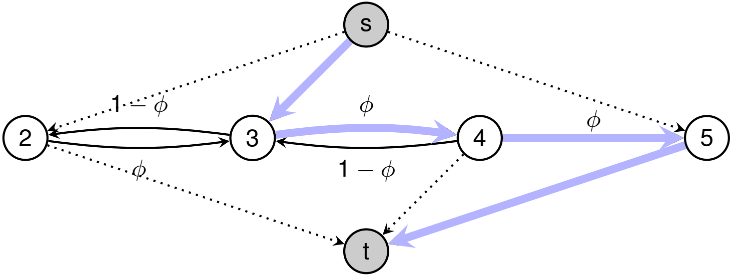
Iteration: 3, $|f| = 1 + \phi$



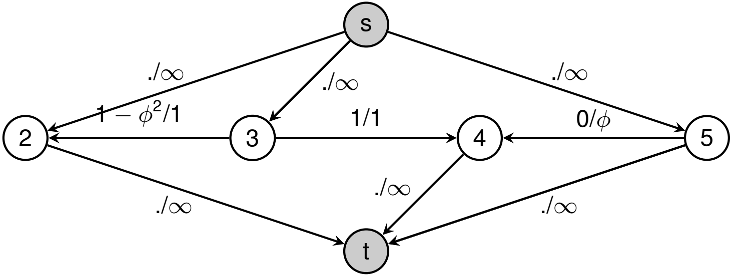
Non-Termination of Ford-Fulkerson for Irrational Capacities



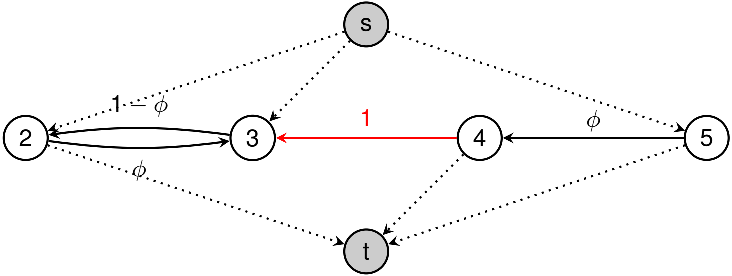
Iteration: 3, $|f| = 1 + 2 \cdot \phi$



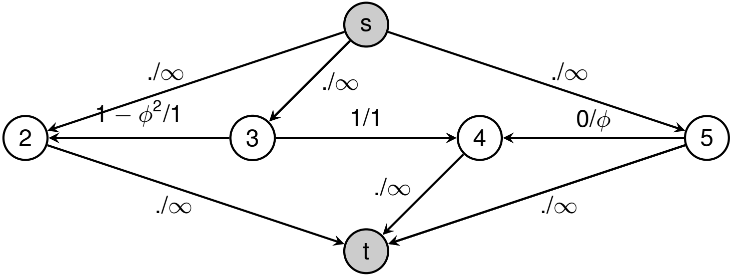
Non-Termination of Ford-Fulkerson for Irrational Capacities



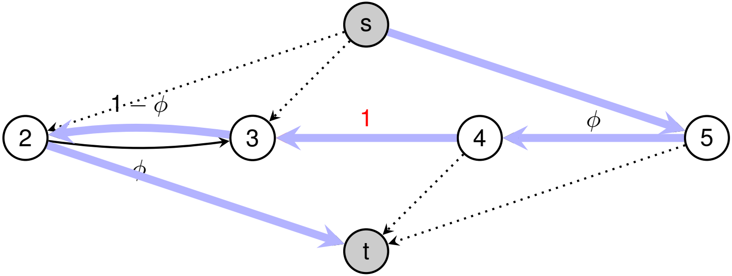
Iteration: 3, $|f| = 1 + 2 \cdot \phi$



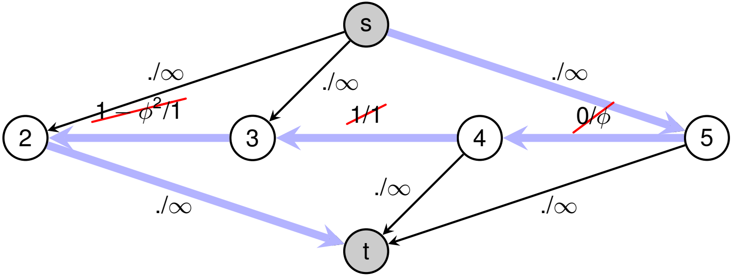
Non-Termination of Ford-Fulkerson for Irrational Capacities



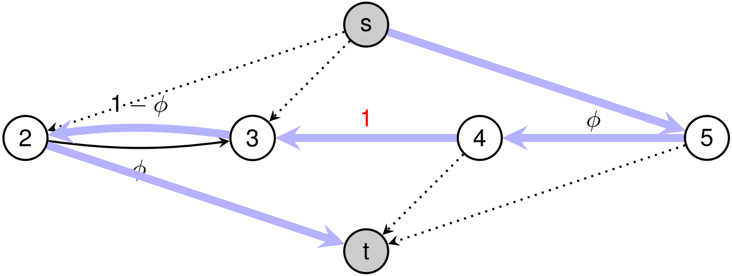
Iteration: 4, $|f| = 1 + 2 \cdot \phi$



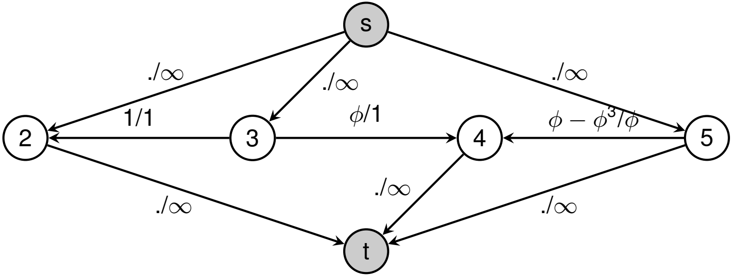
Non-Termination of Ford-Fulkerson for Irrational Capacities



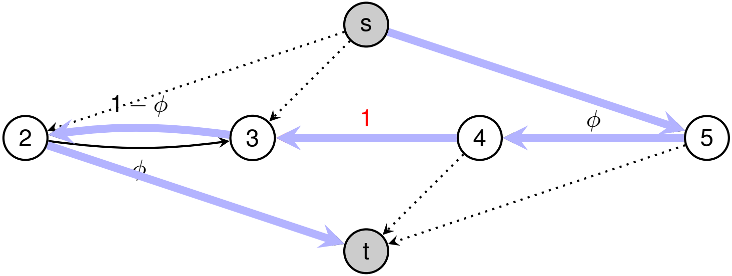
Iteration: 4, $|f| = 1 + 2 \cdot \phi$



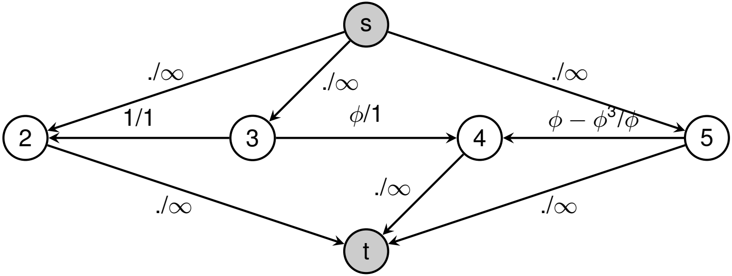
Non-Termination of Ford-Fulkerson for Irrational Capacities



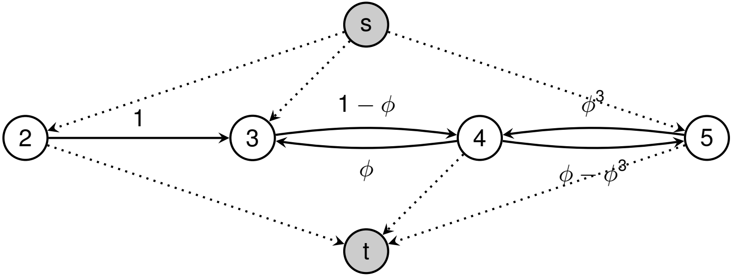
Iteration: 4, $|f| = 1 + 2 \cdot \phi + \phi^2$



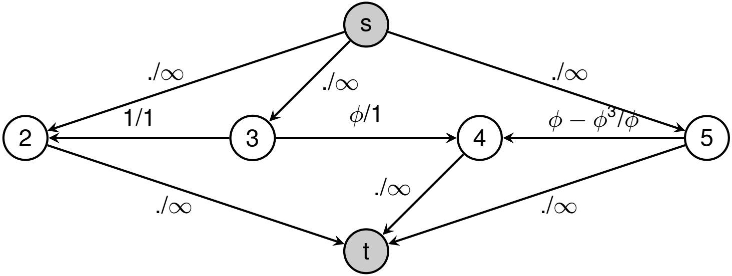
Non-Termination of Ford-Fulkerson for Irrational Capacities



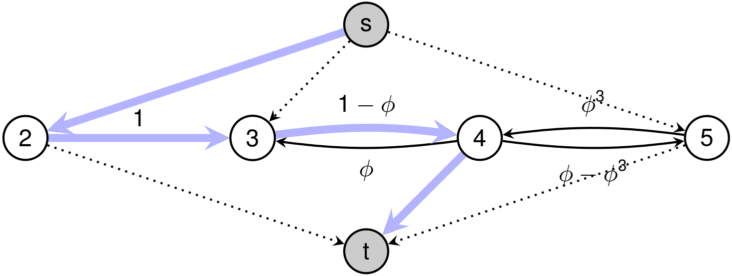
Iteration: 4, $|f| = 1 + 2 \cdot \phi + \phi^2$



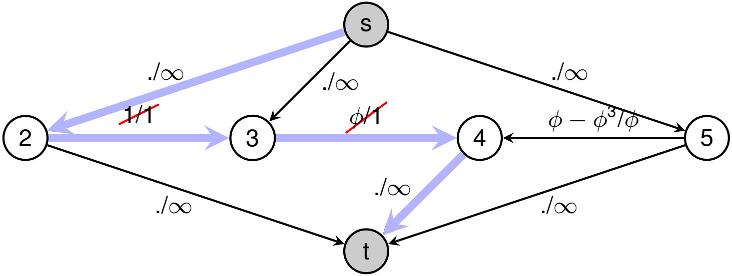
Non-Termination of Ford-Fulkerson for Irrational Capacities



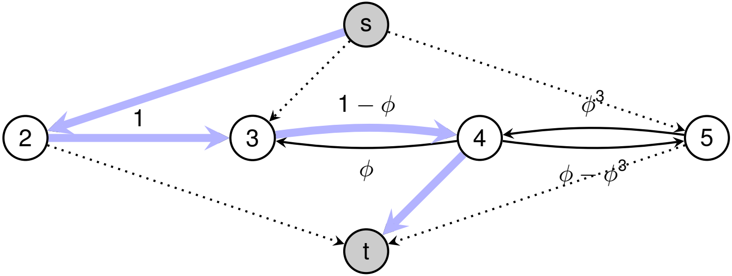
Iteration: 5, $|f| = 1 + 2 \cdot \phi + \phi^2$



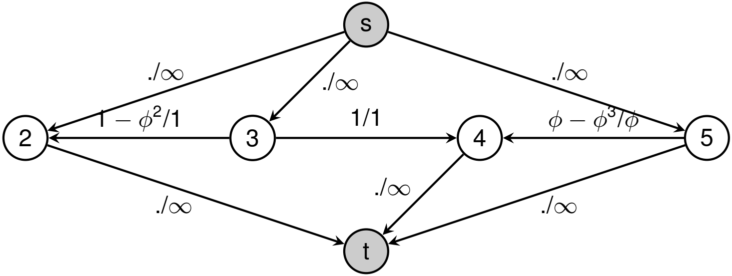
Non-Termination of Ford-Fulkerson for Irrational Capacities



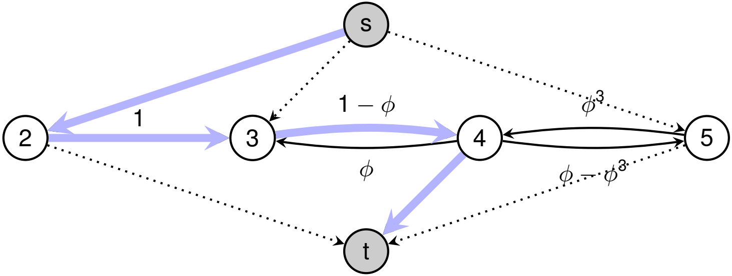
Iteration: 5, $|f| = 1 + 2 \cdot \phi + \phi^2$



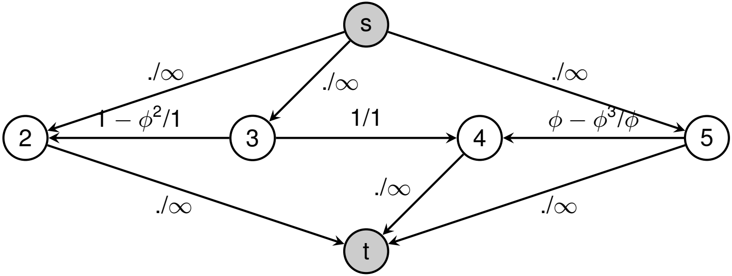
Non-Termination of Ford-Fulkerson for Irrational Capacities



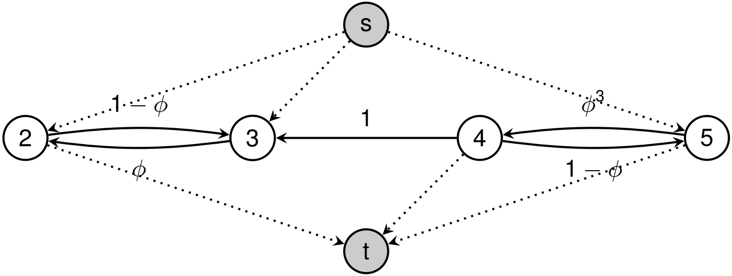
Iteration: 5, $|f| = 1 + 2 \cdot \phi + 2 \cdot \phi^2$



Non-Termination of Ford-Fulkerson for Irrational Capacities

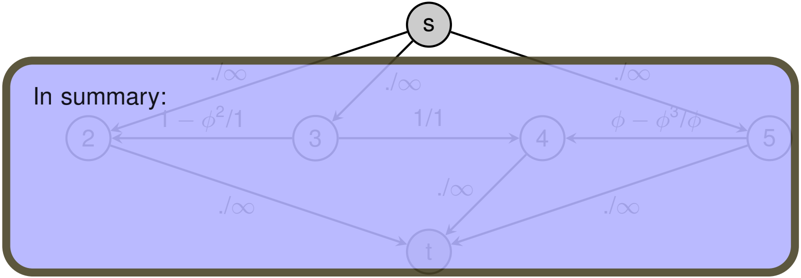


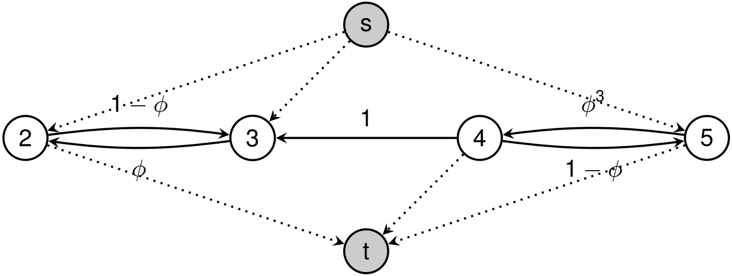
Iteration: 5, $|f| = 1 + 2 \cdot \phi + 2 \cdot \phi^2$



Non-Termination of Ford-Fulkerson for Irrational Capacities

In summary:

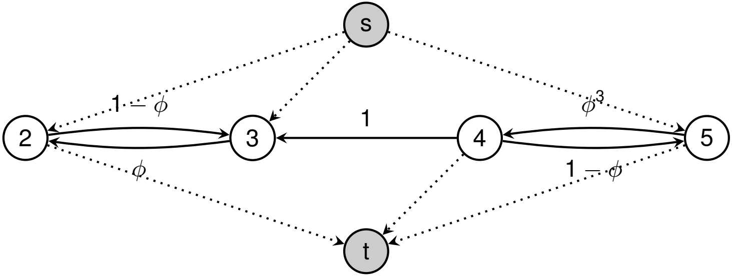




Non-Termination of Ford-Fulkerson for Irrational Capacities

In summary:

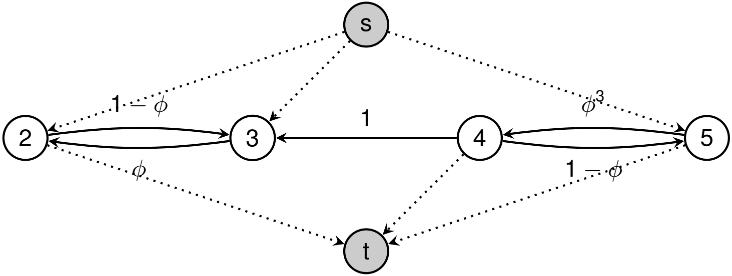
- After iteration 1: $\xleftarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$



Non-Termination of Ford-Fulkerson for Irrational Capacities

In summary:

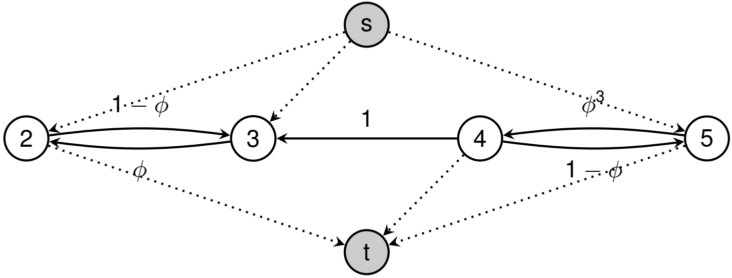
- After iteration 1: $\overset{0}{\longleftarrow}, \overset{1}{\longrightarrow}, \overset{0}{\longleftarrow}, |f| = 1$
- After iteration 5: $\overset{1-\phi^2}{\longleftarrow}, \overset{1}{\longrightarrow}, \overset{\phi-\phi^3}{\longleftarrow}, |f| = 1 + 2\phi + 2\phi^2$



Non-Termination of Ford-Fulkerson for Irrational Capacities

In summary:

- After iteration 1: $\xleftarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$
- After iteration 5: $\xleftarrow{1-\phi^2}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^3}$, $|f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\xleftarrow{1-\phi^4}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^5}$, $|f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

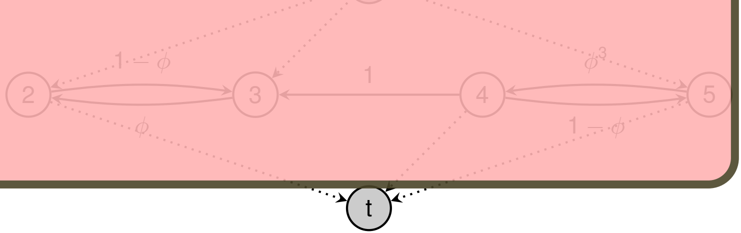


Non-Termination of Ford-Fulkerson for Irrational Capacities

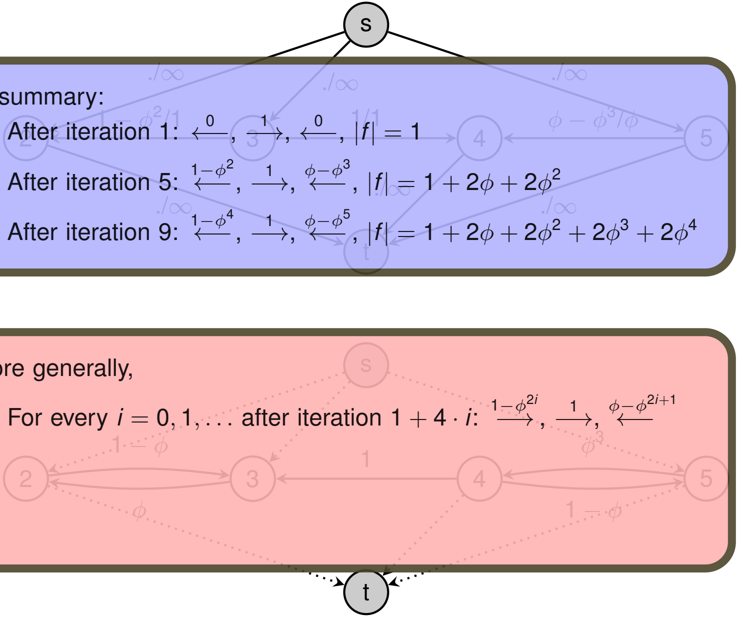
In summary:

- After iteration 1: $\xleftarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$
- After iteration 5: $\xleftarrow{1-\phi^2}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^3}$, $|f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\xleftarrow{1-\phi^4}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^5}$, $|f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

More generally,



Non-Termination of Ford-Fulkerson for Irrational Capacities



In summary:

- After iteration 1: $\xleftarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$
- After iteration 5: $\xleftarrow{1-\phi^2}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^3}$, $|f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\xleftarrow{1-\phi^4}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^5}$, $|f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

More generally,

- For every $i = 0, 1, \ldots$ after iteration $1 + 4 \cdot i$: $\xrightarrow{1-\phi^{2i}}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^{2i+1}}$

Non-Termination of Ford-Fulkerson for Irrational Capacities

In summary:

- After iteration 1: $\xleftarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$
- After iteration 5: $\xleftarrow{1-\phi^2}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^3}$, $|f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\xleftarrow{1-\phi^4}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^5}$, $|f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

More generally,

- For every $i = 0, 1, \ldots$ after iteration $1 + 4 \cdot i$: $\xrightarrow{1-\phi^{2i}}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^{2i+1}}$
- **Ford-Fulkerson does not terminate!**

Non-Termination of Ford-Fulkerson for Irrational Capacities

In summary:

- After iteration 1: $\overset{0}{\longleftarrow}, \overset{1}{\longrightarrow}, \overset{0}{\longleftarrow}, |f| = 1$
- After iteration 5: $\overset{1-\phi^2}{\longleftarrow}, \overset{1}{\longrightarrow}, \overset{\phi-\phi^3}{\longleftarrow}, |f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\overset{1-\phi^4}{\longleftarrow}, \overset{1}{\longrightarrow}, \overset{\phi-\phi^5}{\longleftarrow}, |f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

More generally,

- For every $i = 0, 1, \ldots$ after iteration $1 + 4 \cdot i$: $\overset{1-\phi^{2i}}{\longrightarrow}, \overset{1}{\longrightarrow}, \overset{\phi-\phi^{2i+1}}{\longleftarrow}$
- **Ford-Fulkerson does not terminate!**
- $|f| = 1 + 2\sum_{i=1}^{\infty} \varphi^i \approx 4.23607 < 5$

Non-Termination of Ford-Fulkerson for Irrational Capacities

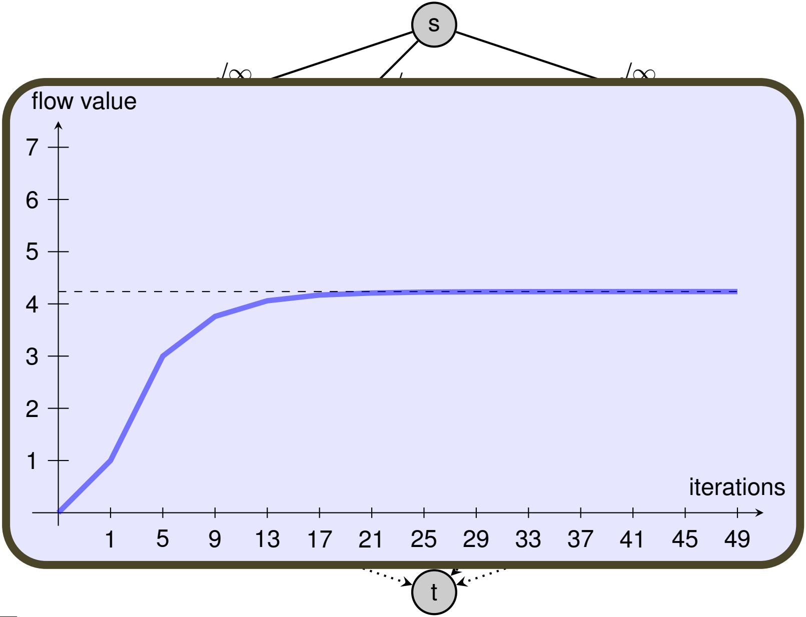
In summary:

- After iteration 1: $\xleftarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$
- After iteration 5: $\xleftarrow{1-\phi^2}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^3}$, $|f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\xleftarrow{1-\phi^4}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^5}$, $|f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

More generally,

- For every $i = 0, 1, \ldots$ after iteration $1 + 4 \cdot i$: $\xrightarrow{1-\phi^{2i}}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^{2i+1}}$
- **Ford-Fulkerson does not terminate!**
- $|f| = 1 + 2\sum_{i=1}^{\infty} \varphi^i \approx 4.23607 < 5$
- **It does not even converge to a maximum flow!**

Non-Termination of Ford-Fulkerson for Irrational Capacities



Summary and Outlook

Ford-Fulkerson Method

- works only for integral (rational) capacities
- Runtime: $O(E \cdot |f^*|) = O(E \cdot V \cdot C)$

Summary and Outlook

Ford-Fulkerson Method

- works only for integral (rational) capacities
- Runtime: $O(E \cdot |f^*|) = O(E \cdot V \cdot C)$

Capacity-Scaling Algorithm

Summary and Outlook

Ford-Fulkerson Method

- works only for integral (rational) capacities
- Runtime: $O(E \cdot |f^*|) = O(E \cdot V \cdot C)$

Capacity-Scaling Algorithm

- Idea: Find an augmenting path with high capacity
- Consider subgraph of G_f consisting of edges (u, v) with $c_f(u, v) > \Delta$
- scaling parameter Δ, which is initially $2^{\lceil \log_2 C \rceil}$ and 1 after termination
- Runtime: $O(E^2 \cdot \log C)$

Summary and Outlook

Ford-Fulkerson Method

- works only for integral (rational) capacities
- Runtime: $O(E \cdot |f^*|) = O(E \cdot V \cdot C)$

Capacity-Scaling Algorithm

- Idea: Find an augmenting path with high capacity
- Consider subgraph of G_f consisting of edges (u, v) with $c_f(u, v) > \Delta$
- scaling parameter Δ, which is initially $2^{\lceil \log_2 C \rceil}$ and 1 after termination
- Runtime: $O(E^2 \cdot \log C)$

Edmonds-Karp Algorithm

- Idea: Find the shortest augmenting path in G_f
- Runtime: $O(E^2 \cdot V)$

Outline

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

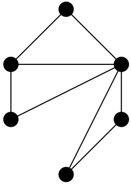
Matchings in Bipartite Graphs

Application: Maximum-Bipartite-Matching Problem

Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.

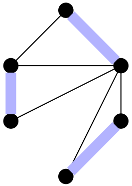
Application: Maximum-Bipartite-Matching Problem



Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.

Application: Maximum-Bipartite-Matching Problem



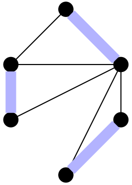
Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.

Application: Maximum-Bipartite-Matching Problem

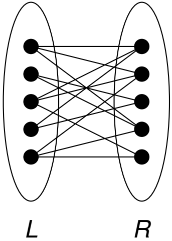
Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.



Bipartite Graph

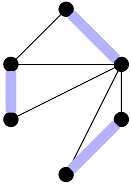
A graph G is bipartite if V can be partitioned into L and R so that all edges go between L and R.



Application: Maximum-Bipartite-Matching Problem

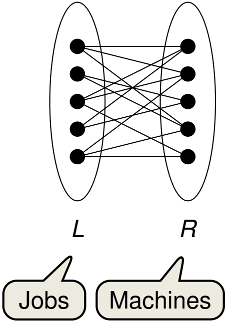
Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.

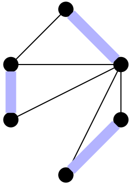


Bipartite Graph

A graph G is bipartite if V can be partitioned into L and R so that all edges go between L and R.



Application: Maximum-Bipartite-Matching Problem



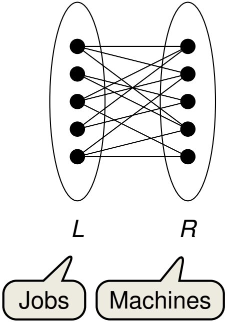
Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.

Bipartite Graph

A graph G is bipartite if V can be partitioned into L and R so that all edges go between L and R.

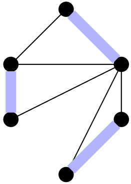
Given a bipartite graph $G = (L \cup R, E)$, find a matching of maximum cardinality.



Application: Maximum-Bipartite-Matching Problem

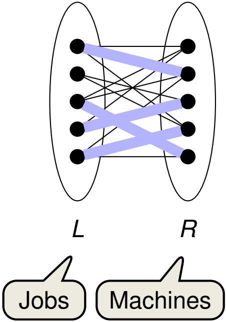
Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.



Bipartite Graph

A graph G is bipartite if V can be partitioned into L and R so that all edges go between L and R.

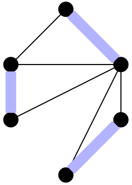


Given a bipartite graph $G = (L \cup R, E)$, find a matching of maximum cardinality.

Application: Maximum-Bipartite-Matching Problem

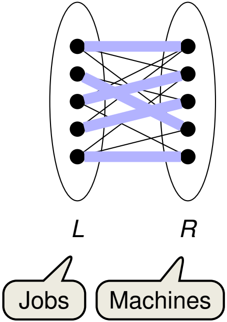
Matching

A matching is a subset $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident to v.



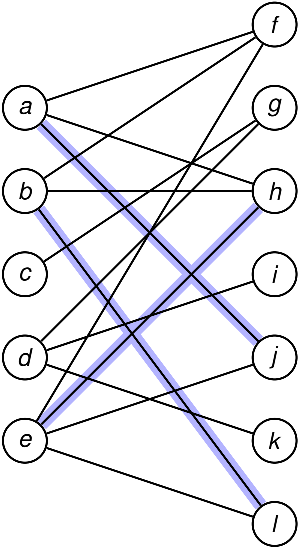
Bipartite Graph

A graph G is bipartite if V can be partitioned into L and R so that all edges go between L and R.

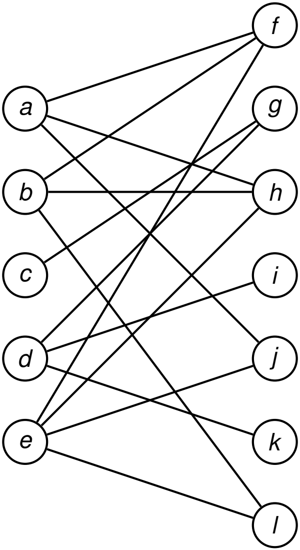


Given a bipartite graph $G = (L \cup R, E)$, find a matching of maximum cardinality.

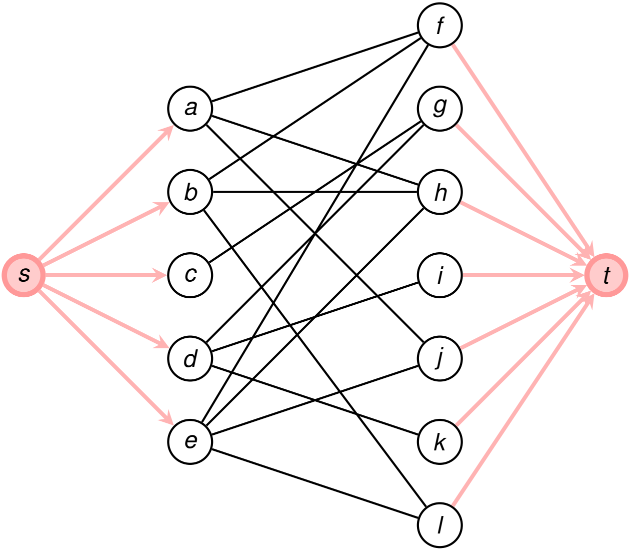
Matchings in Bipartite Graphs via Maximum Flows



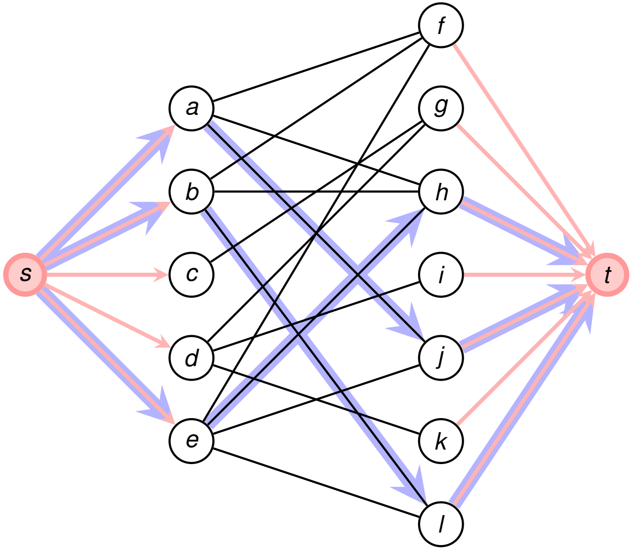
Matchings in Bipartite Graphs via Maximum Flows



Matchings in Bipartite Graphs via Maximum Flows



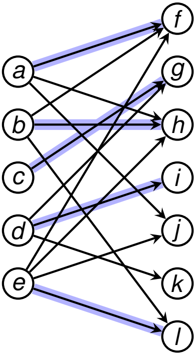
Matchings in Bipartite Graphs via Maximum Flows



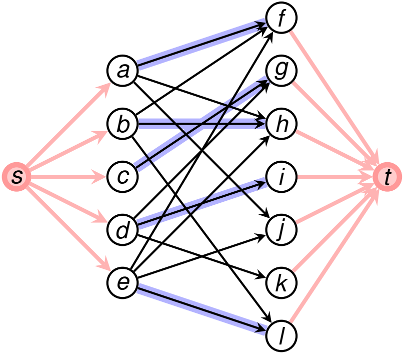
Correspondence between Maximum Matchings and Max Flow

Theorem (Corollary 26.11)

The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in the corresponding flow network $\widetilde{G}$.



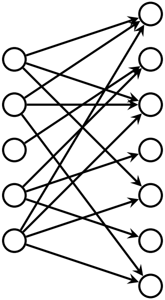
Graph G



Graph $\widetilde{G}$

From Matching to Flow

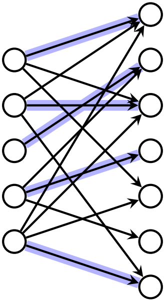
- Given a maximum matching of cardinality k



Graph G

From Matching to Flow

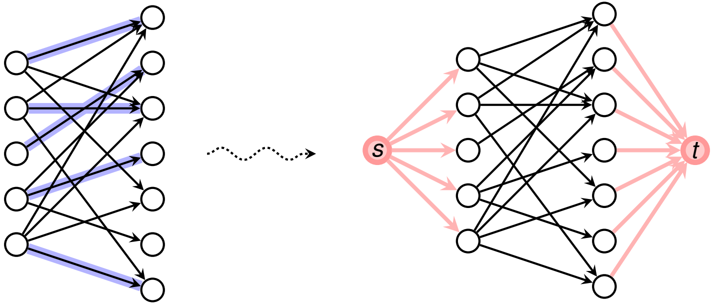
- Given a maximum matching of cardinality k



Graph G

From Matching to Flow

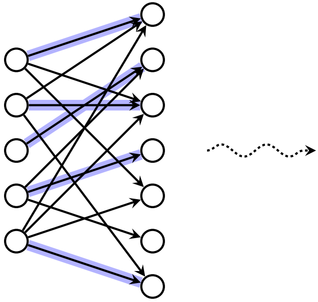
- Given a maximum matching of cardinality k
- Consider flow f that sends one unit along each each of k paths



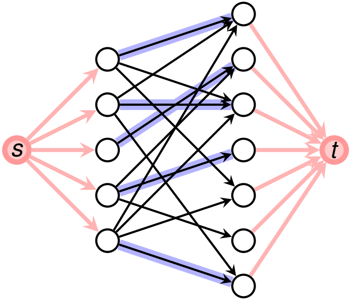
Graph G Graph $\widetilde{G}$

From Matching to Flow

- Given a maximum matching of cardinality k
- Consider flow f that sends one unit along each each of k paths



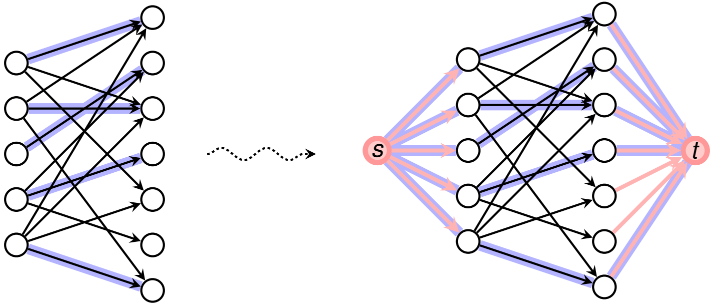
Graph G



Graph $\widetilde{G}$

From Matching to Flow

- Given a maximum matching of cardinality k
- Consider flow f that sends one unit along each each of k paths

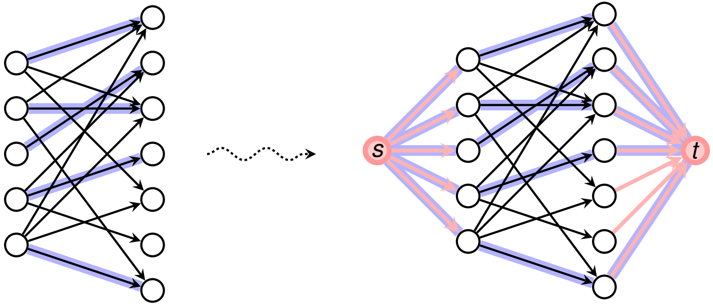


Graph G Graph $\widetilde{G}$

From Matching to Flow

- Given a maximum matching of cardinality k
- Consider flow f that sends one unit along each each of k paths

$\Rightarrow$ f is a flow and has value k



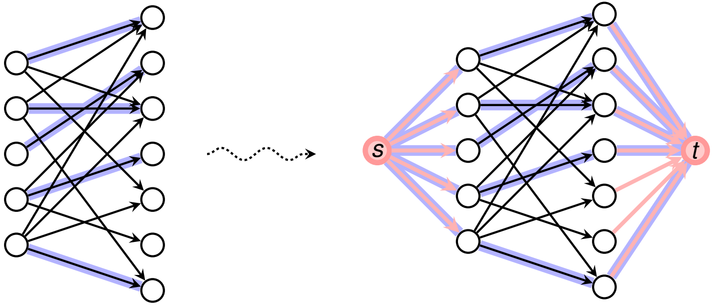
Graph G Graph $\widetilde{G}$

From Matching to Flow

- Given a maximum matching of cardinality k
- Consider flow f that sends one unit along each each of k paths

$\Rightarrow$ f is a flow and has value k

max cardinality matching $\leq$ value of maxflow



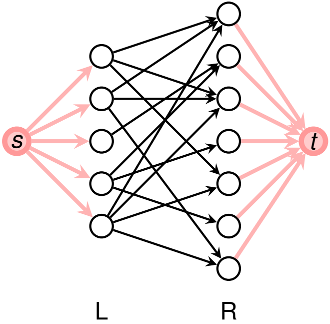
Graph G Graph $\widetilde{G}$

From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k

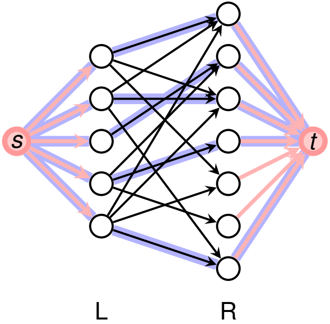
From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k



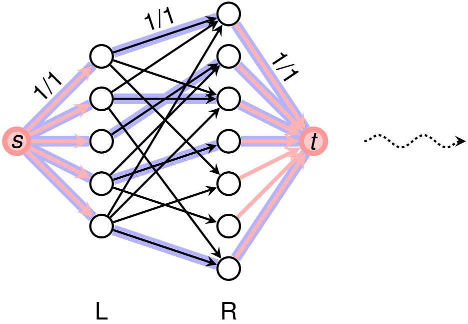
From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k



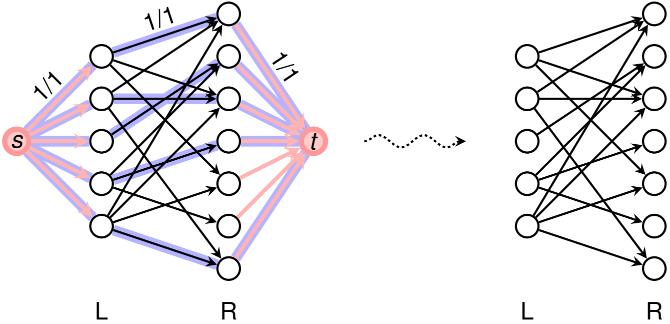
From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral



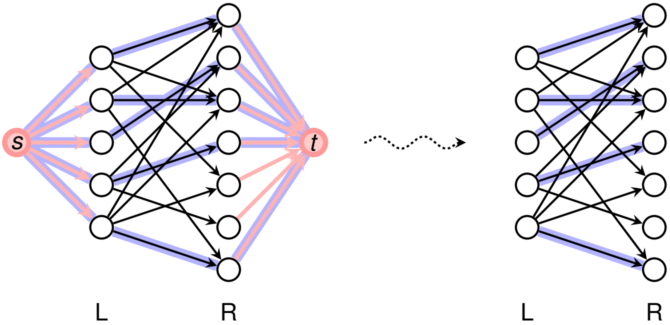
From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one



From Flow to Matching

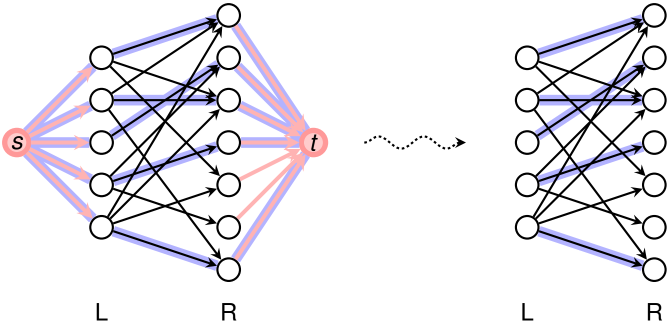
- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u, v) \in \{0, 1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

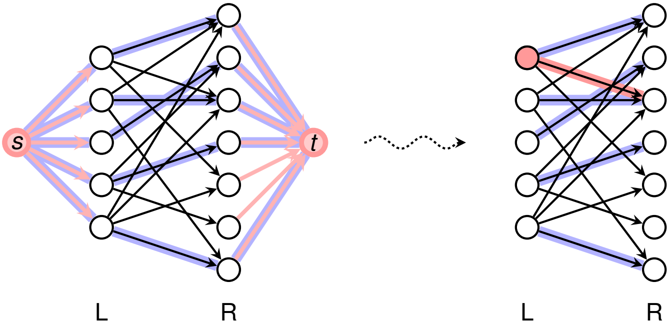
a) Flow Conservation



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u, v) \in \{0, 1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

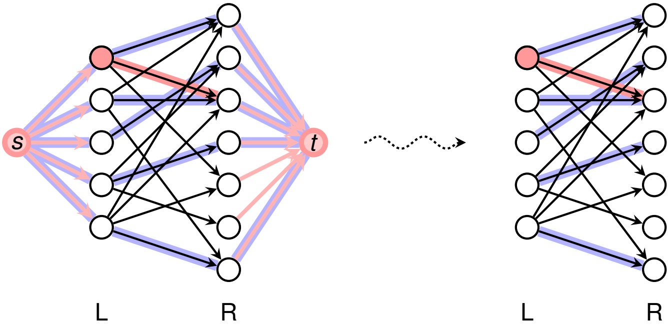
a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

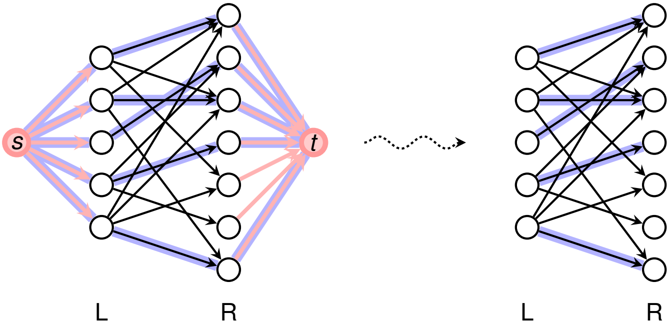
a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

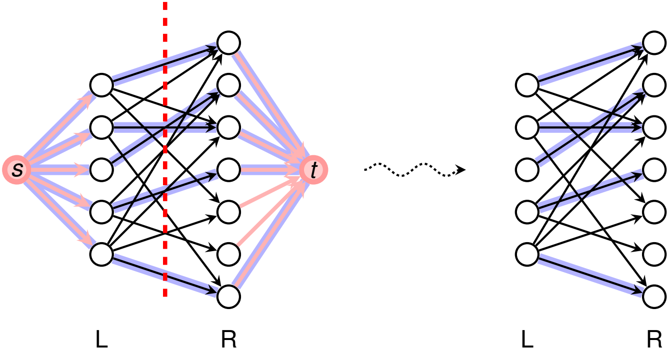
a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

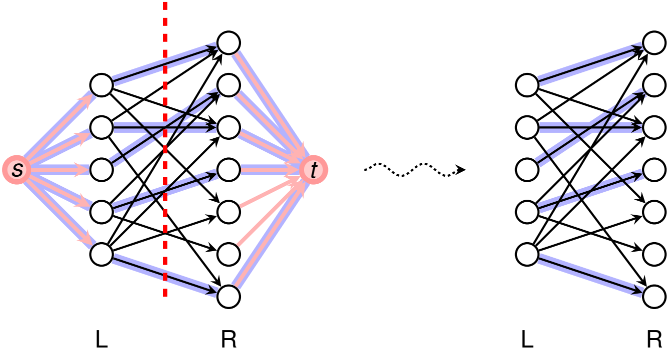
a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit
c) Cut $(L \cup \{s\}, R \cup \{t\})$



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

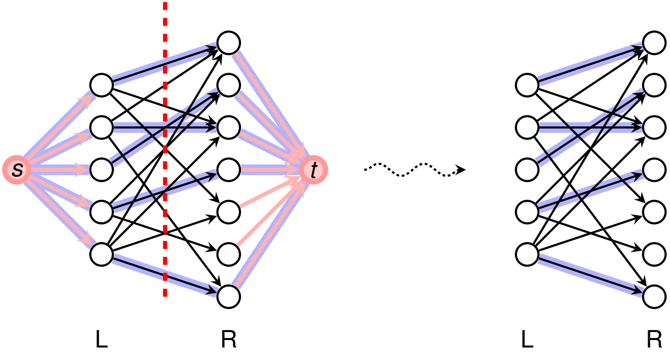
a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit
c) Cut $(L \cup \{s\}, R \cup \{t\}) \Rightarrow$ net flow is k



From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit
c) Cut $(L \cup \{s\}, R \cup \{t\}) \Rightarrow$ net flow is $k \Rightarrow M'$ has k edges

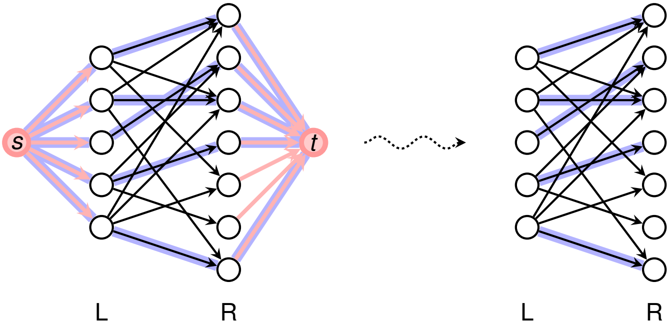


From Flow to Matching

- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit
c) Cut $(L \cup \{s\}, R \cup \{t\}) \Rightarrow$ net flow is $k \Rightarrow M'$ has k edges

$\Rightarrow$ By a) & b), M' is a matching and by c), M' has cardinality k



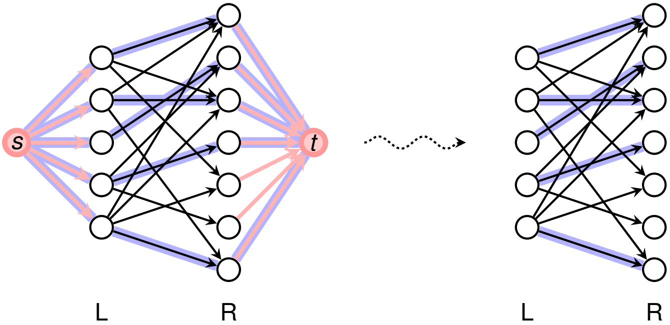
From Flow to Matching

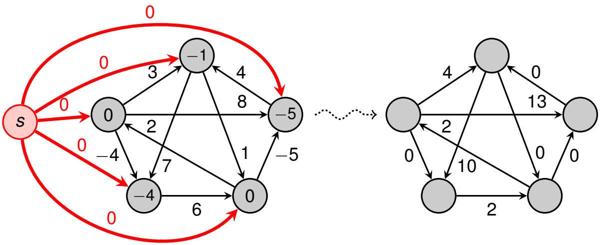
- Let f be a maximum flow in $\widetilde{G}$ of value k
- Integrality Theorem $\Rightarrow f(u,v) \in \{0,1\}$ and k integral
- Let M' be all edges from L to R which carry a flow of one

a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit
c) Cut $(L \cup \{s\}, R \cup \{t\}) \Rightarrow$ net flow is $k \Rightarrow M'$ has k edges

$\Rightarrow$ By a) & b), M' is a matching and by c), M' has cardinality k

value of maxflow $\leq$ max cardinality matching





6.5: All-Pairs Shortest Paths

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

All-Pairs Shortest Path

APSP via Matrix Multiplication

Johnson's Algorithm

Formalising the Problem

All-Pairs Shortest Path Problem

- Given: directed graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, with edge weights represented by a matrix W:

$$w_{i,j} = \begin{cases} \text{weight of edge } (i,j) & \text{for an edge } (i,j) \in E, \\ \infty & \text{if there is no edge from } i \text{ to } j, \\ 0 & \text{if } i = j. \end{cases}$$

Formalising the Problem

All-Pairs Shortest Path Problem

- Given: directed graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, with edge weights represented by a matrix W:

$$w_{i,j} = \begin{cases} \text{weight of edge } (i,j) & \text{for an edge } (i,j) \in E, \\ \infty & \text{if there is no edge from } i \text{ to } j, \\ 0 & \text{if } i = j. \end{cases}$$

- Goal: Obtain a matrix of shortest path weights L, that is

$$\ell_{i,j} = \begin{cases} \text{weight of a shortest path from } i \text{ to } j, & \text{if } j \text{ is reachable from } i \\ \infty & \text{otherwise.} \end{cases}$$

Formalising the Problem

All-Pairs Shortest Path Problem

- Given: directed graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, with edge weights represented by a matrix W:

$$w_{i,j} = \begin{cases} \text{weight of edge } (i,j) & \text{for an edge } (i,j) \in E, \\ \infty & \text{if there is no edge from } i \text{ to } j, \\ 0 & \text{if } i = j. \end{cases}$$

- Goal: Obtain a matrix of shortest path weights L, that is

$$\ell_{i,j} = \begin{cases} \text{weight of a shortest path from } i \text{ to } j, & \text{if } j \text{ is reachable from } i \\ \infty & \text{otherwise.} \end{cases}$$

Here we will only compute the weight of the shortest path without keeping track of the edges of the path!

Outline

All-Pairs Shortest Path

APSP via Matrix Multiplication

Johnson's Algorithm

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

$$\ell_{i,j}^{(2)} = \min\left(\ell_{i,j}^{(1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(1)} + w_{k,j}\right)$$

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

$$\ell_{i,j}^{(2)} = \min\left(\ell_{i,j}^{(1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(1)} + w_{k,j}\right)$$

$$\ell_{i,j}^{(m)} =$$

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

$$\ell_{i,j}^{(2)} = \min\left(\ell_{i,j}^{(1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(1)} + w_{k,j}\right)$$

$$\boxed{\ell_{i,j}^{(m)} = \min\left(\ell_{i,j}^{(m-1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(m-1)} + w_{k,j}\right)}$$

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

$$\ell_{i,j}^{(2)} = \min\left(\ell_{i,j}^{(1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(1)} + w_{k,j}\right)$$

Recall that $w_{j,j} = 0$!

$$\ell_{i,j}^{(m)} = \min\left(\ell_{i,j}^{(m-1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(m-1)} + w_{k,j}\right)$$

A Recursive Approach

Basic Idea

- Any shortest path from i to j of length $k \geq 2$ is the concatenation of a shortest path of length $k-1$ and an edge

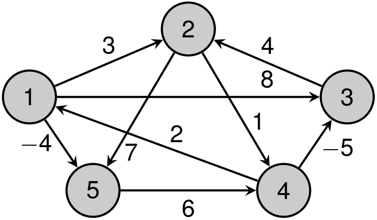
- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

$$\ell_{i,j}^{(2)} = \min\left(\ell_{i,j}^{(1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(1)} + w_{k,j}\right)$$

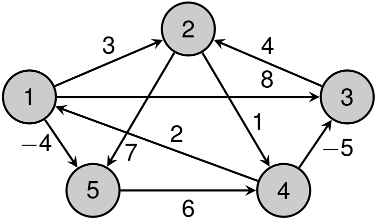
Recall that $w_{j,j} = 0$!

$$\ell_{i,j}^{(m)} = \min\left(\ell_{i,j}^{(m-1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(m-1)} + w_{k,j}\right) = \min_{1 \leq k \leq n}\left(\ell_{i,k}^{(m-1)} + w_{k,j}\right)$$

Example of Shortest Path via Matrix Multiplication (Figure 25.1)

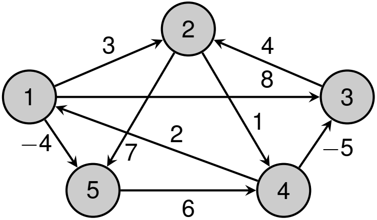


Example of Shortest Path via Matrix Multiplication (Figure 25.1)



$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

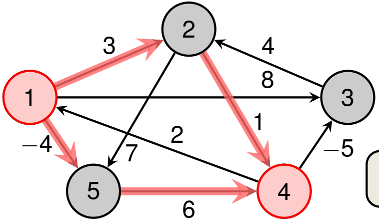
Example of Shortest Path via Matrix Multiplication (Figure 25.1)



$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{?} & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

Example of Shortest Path via Matrix Multiplication (Figure 25.1)

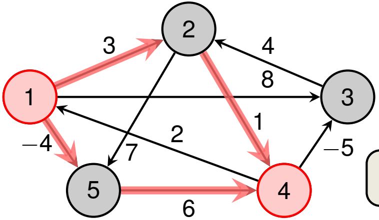


$$\ell_{1,4}^{(2)} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & ? & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

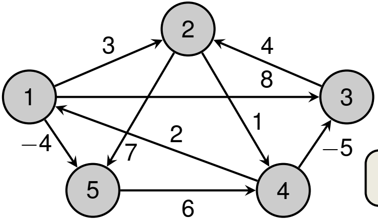
Example of Shortest Path via Matrix Multiplication (Figure 25.1)



$$\ell^{(2)}_{1,4} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, \; -4+6\}$$

$$L^{(1)} = W = \left(\begin{array}{ccc|c|c} 0 & 3 & 8 & \infty & -4 \\ \hline \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{array}\right) \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

Example of Shortest Path via Matrix Multiplication (Figure 25.1)



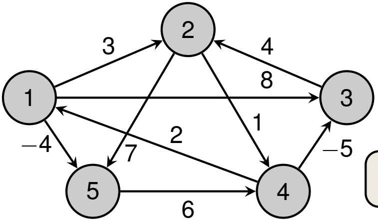
$$\ell_{1,4}^{(2)} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{2} & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Example of Shortest Path via Matrix Multiplication (Figure 25.1)

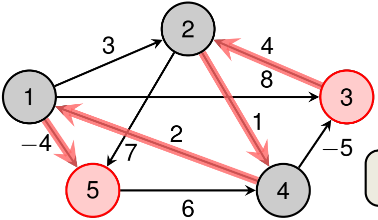


$$\ell_{1,4}^{(2)} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{2} & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & \boxed{?} \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Example of Shortest Path via Matrix Multiplication (Figure 25.1)



$$\ell^{(2)}_{1,4} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

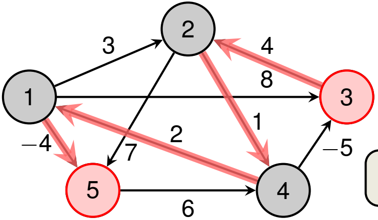
$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{2} & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & \boxed{?} \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\ell^{(4)}_{3,5} = \min\{7-4, 4+7, 0+\infty, 5+\infty, 11+0\}$$

Example of Shortest Path via Matrix Multiplication (Figure 25.1)



$$\ell^{(2)}_{1,4} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{2} & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \left(\begin{array}{ccccc} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ \hline 7 & 4 & 0 & 5 & 11 \\ \hline 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{array}\right) \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & \boxed{3} \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\ell^{(4)}_{3,5} = \min\{7-4, 4+7, 0+\infty, 5+\infty, 11+0\}$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n-1 = |V| - 1$ edges (assuming absence of negative-weight cycles)

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n-1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n-1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n - 1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n - 1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

- The correspondence is as follows:

$$\min \Leftrightarrow \sum$$

$$+ \Leftrightarrow \times$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n-1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

- The correspondence is as follows:

$$\min \Leftrightarrow \sum$$

$$+ \Leftrightarrow \times$$

$$\infty \Leftrightarrow ?$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n-1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

- The correspondence is as follows:

$$\begin{aligned} \min &\Leftrightarrow \sum \\ + &\Leftrightarrow \times \\ \infty &\Leftrightarrow 0 \end{aligned}$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1\leq k\leq n}\left(\ell_{i,k}^{(m-1)} + w_{k,j}\right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n-1 = |V|-1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1\leq k\leq n}\left(\ell_{i,k}^{(m-1)} + w_{k,j}\right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1\leq k\leq n}\left(\ell_{i,k}^{(m-1)} \times w_{k,j}\right)$$

- The correspondence is as follows:

$$\begin{array}{ccc} \min & \Leftrightarrow & \sum \\ + & \Leftrightarrow & \times \\ \infty & \Leftrightarrow & 0 \\ 0 & \Leftrightarrow & ? \end{array}$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n - 1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

- The correspondence is as follows:

$$\begin{array}{ccc} \min & \Leftrightarrow & \sum \\ + & \Leftrightarrow & \times \\ \infty & \Leftrightarrow & 0 \\ 0 & \Leftrightarrow & 1 \end{array}$$

Computing $L^{(m)}$

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L$, since every shortest path uses at most $n - 1 = |V| - 1$ edges (assuming absence of negative-weight cycles)
- Computing $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

$L^{(m)}$ can be computed in $\mathcal{O}(n^3)$

- The correspondence is as follows:

$$\begin{array}{ccc} \min & \Leftrightarrow & \sum \\ + & \Leftrightarrow & \times \\ \infty & \Leftrightarrow & 0 \\ 0 & \Leftrightarrow & 1 \end{array}$$

Computing $L^{(n-1)}$ efficiently

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

- For, say, $n = 738$, we subsequently compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \dots, L^{(737)} = L$$

Computing $L^{(n-1)}$ efficiently

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

Takes $\mathcal{O}(n \cdot n^3) = \mathcal{O}(n^4)$

- For, say, $n = 738$, we subsequently compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \ldots, L^{(737)} = L$$

Computing $L^{(n-1)}$ efficiently

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

Takes $\mathcal{O}(n \cdot n^3) = \mathcal{O}(n^4)$

- For, say, $n = 738$, we subsequently compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \ldots, L^{(737)} = L$$

- Since we don't need the intermediate matrices, a more efficient way is

$$L^{(1)}, L^{(2)}, L^{(4)}, \ldots, L^{(512)}, L^{(1024)} = L$$

Computing $L^{(n-1)}$ efficiently

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

Takes $\mathcal{O}(n \cdot n^3) = \mathcal{O}(n^4)$

- For, say, $n = 738$, we subsequently compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \ldots, L^{(737)} = L$$

- Since we don't need the intermediate matrices, a more efficient way is

$$L^{(1)}, L^{(2)}, L^{(4)}, \ldots, L^{(512)}, L^{(1024)} = L$$

Takes $\mathcal{O}(\log n \cdot n^3)$.

Computing $L^{(n-1)}$ efficiently

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

Takes $\mathcal{O}(n \cdot n^3) = \mathcal{O}(n^4)$

- For, say, $n = 738$, we subsequently compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \ldots, L^{(737)} = L$$

- Since we don't need the intermediate matrices, a more efficient way is

$$L^{(1)}, L^{(2)}, L^{(4)}, \ldots, L^{(512)}, L^{(1024)} = L$$

We need $L^{(4)} = L^{(2)} \cdot L^{(2)} = L^{(3)} \cdot L^{(1)}$! (see Ex. 25.1-4)

Takes $\mathcal{O}(\log n \cdot n^3)$.

Outline

All-Pairs Shortest Path

APSP via Matrix Multiplication

Johnson's Algorithm

Johnson's Algorithm

Overview

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles
- one pass of Bellman-Ford and $|V|$ passes of Dijkstra

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles
- one pass of Bellman-Ford and $|V|$ passes of Dijkstra
- after Bellman-Ford, edges are reweighted s.t.

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles
- one pass of Bellman-Ford and $|V|$ passes of Dijkstra
- after Bellman-Ford, edges are reweighted s.t.
  - all edge weights are non-negative

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles
- one pass of Bellman-Ford and $|V|$ passes of Dijkstra
- after Bellman-Ford, edges are reweighted s.t.
  - all edge weights are non-negative
  - shortest paths are maintained

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles
- one pass of Bellman-Ford and $|V|$ passes of Dijkstra
- after Bellman-Ford, edges are reweighted s.t.
  - all edge weights are non-negative
  - shortest paths are maintained

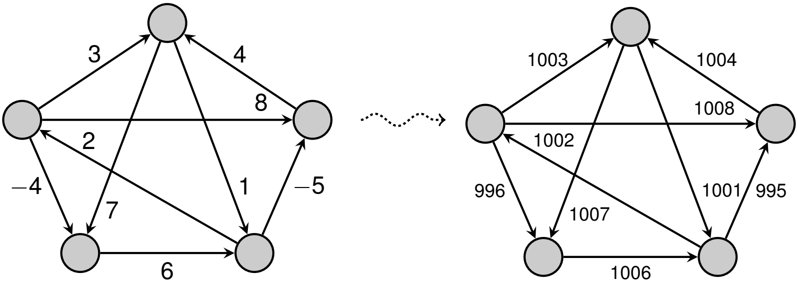
Adding a constant to every edge doesn't work!

Johnson's Algorithm

Overview

- allow negative-weight edges and negative-weight cycles
- one pass of Bellman-Ford and $|V|$ passes of Dijkstra
- after Bellman-Ford, edges are reweighted s.t.
  - all edge weights are non-negative
  - shortest paths are maintained

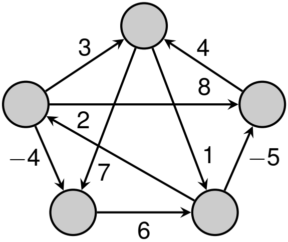
Adding a constant to every edge doesn't work!



How Johnson's Algorithm works

Johnson's Algorithm

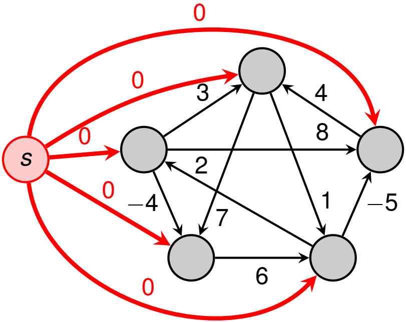
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0



How Johnson's Algorithm works

Johnson's Algorithm

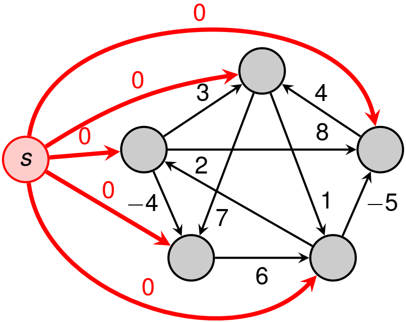
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0



How Johnson's Algorithm works

Johnson's Algorithm

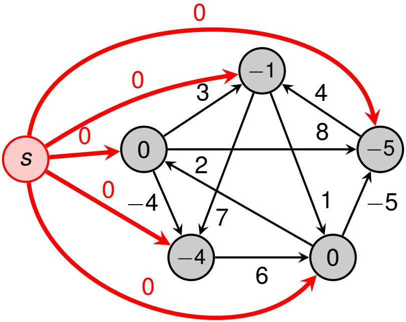
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s



How Johnson's Algorithm works

Johnson's Algorithm

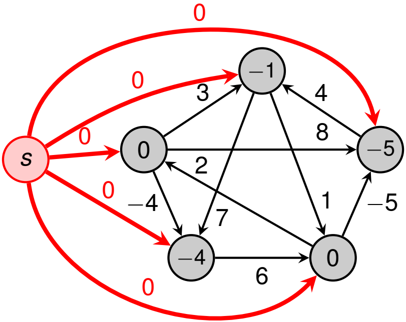
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s



How Johnson's Algorithm works

Johnson's Algorithm

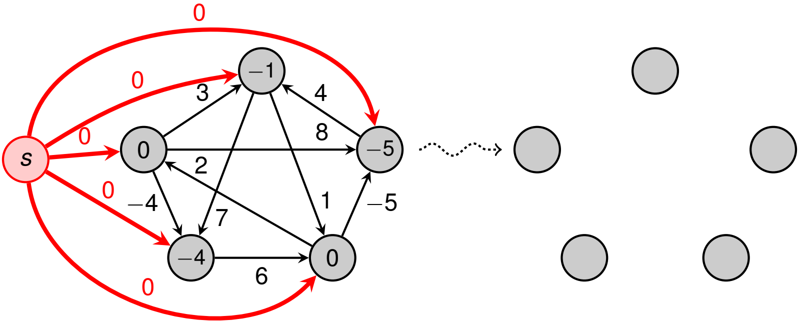
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort



How Johnson's Algorithm works

Johnson's Algorithm

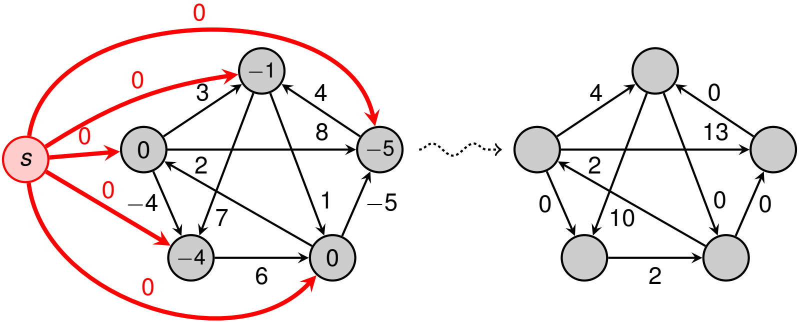
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges



How Johnson's Algorithm works

Johnson's Algorithm

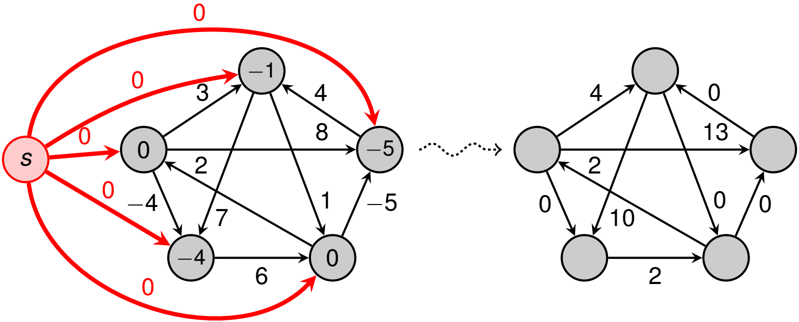
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges



How Johnson's Algorithm works

Johnson's Algorithm

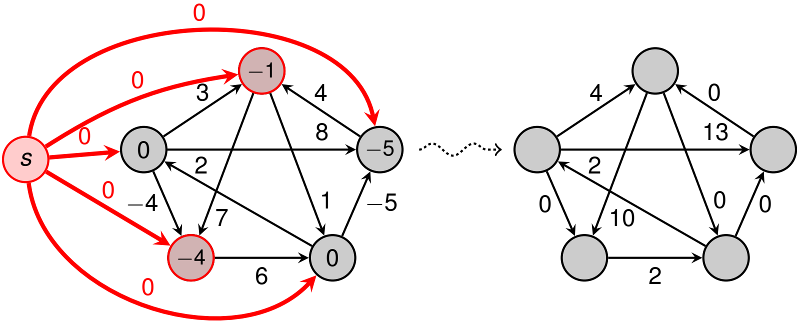
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$



How Johnson's Algorithm works

Johnson's Algorithm

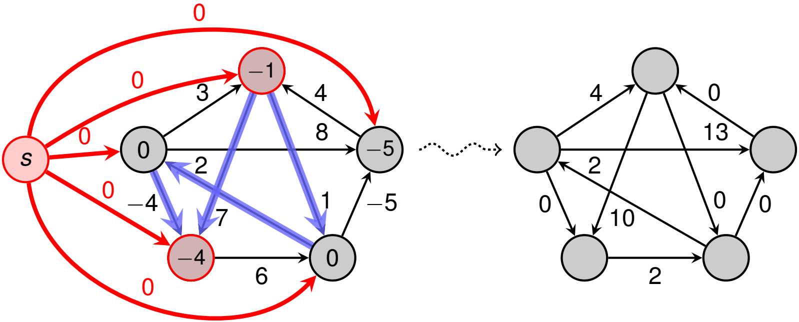
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$



How Johnson's Algorithm works

Johnson's Algorithm

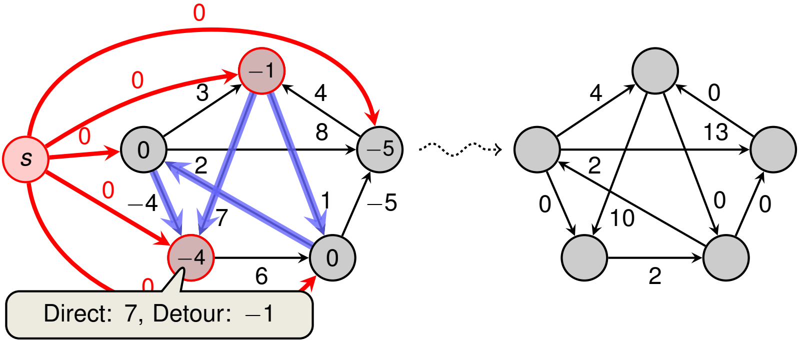
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$



How Johnson's Algorithm works

Johnson's Algorithm

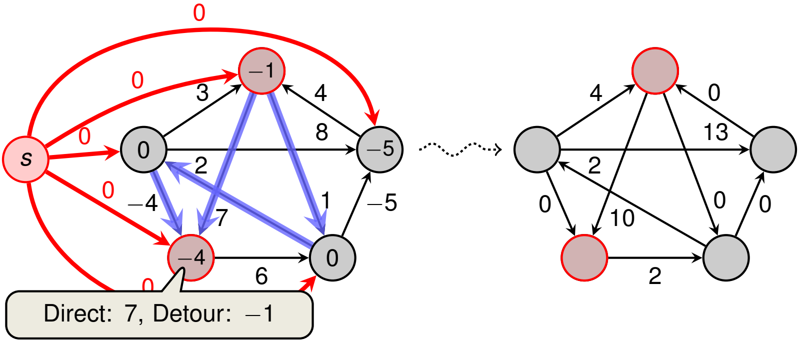
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$



How Johnson's Algorithm works

Johnson's Algorithm

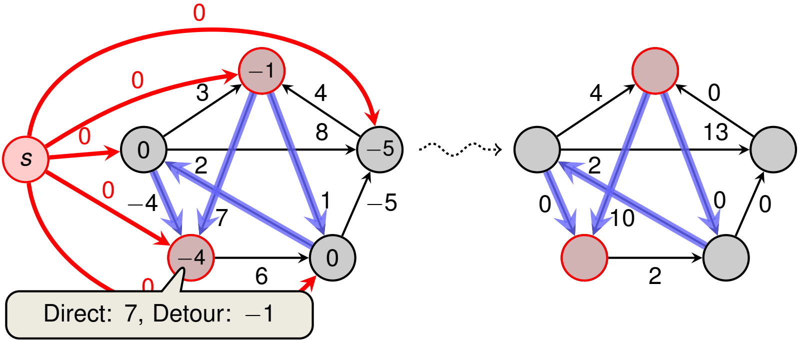
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$



How Johnson's Algorithm works

Johnson's Algorithm

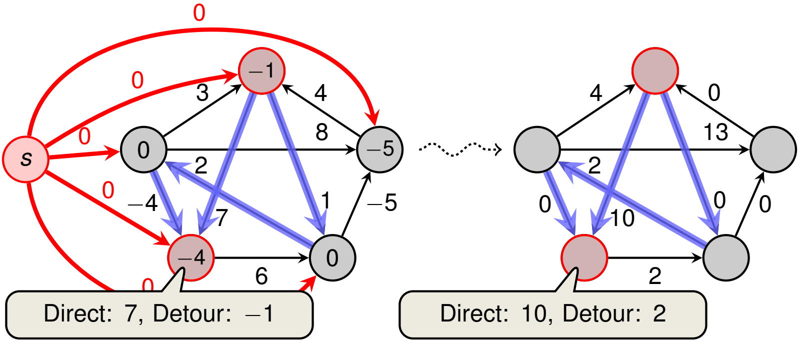
1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$



How Johnson's Algorithm works

Johnson's Algorithm

1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$

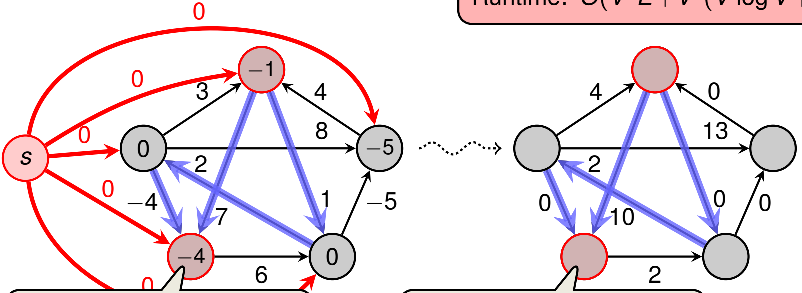


How Johnson's Algorithm works

Johnson's Algorithm

1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
2. Run Bellman-Ford on this augmented graph with source s
   - If there are negative weight cycles, abort
   - Otherwise:
     1) Reweight every edge (u, v) by $\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$
     2) Remove vertex s and its incident edges
3. For every vertex $v \in V$, run Dijkstra on $(G, E, \widetilde{w})$

Runtime: $O(V \cdot E + V \cdot (V \log V + E))$



Direct: 7, Detour: −1

Direct: 10, Detour: 2

Correctness of Johnson's Algorithm

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Correctness of Johnson's Algorithm

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 1.

Correctness of Johnson's Algorithm

$$\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 1.

Let $u.\delta$ and $v.\delta$ be the distances from the fake source s

Correctness of Johnson's Algorithm

$$\widetilde{w}(u,v) = w(u,v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 1.

Let $u.\delta$ and $v.\delta$ be the distances from the fake source s

$$u.\delta + w(u,v) \geq v.\delta \qquad \text{(triangle inequality)}$$

Correctness of Johnson's Algorithm

$$\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 1.

Let $u.\delta$ and $v.\delta$ be the distances from the fake source s

$$u.\delta + w(u, v) \geq v.\delta \qquad \text{(triangle inequality)}$$

$$\Rightarrow \qquad \widetilde{w}(u, v) + u.\delta + w(u, v) \geq w(u, v) + u.\delta - v.\delta + v.\delta$$

Correctness of Johnson's Algorithm

$$\widetilde{w}(u,v) = w(u,v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 1.

Let $u.\delta$ and $v.\delta$ be the distances from the fake source s

$$\begin{aligned} u.\delta + w(u,v) &\geq v.\delta && \text{(triangle inequality)} \\ \Rightarrow \quad \widetilde{w}(u,v) + u.\delta + w(u,v) &\geq w(u,v) + u.\delta - v.\delta + v.\delta \\ \Rightarrow \quad \widetilde{w}(u,v) &\geq 0 \end{aligned}$$

□

Correctness of Johnson's Algorithm

$$\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Correctness of Johnson's Algorithm

$$\widetilde{w}(u,v) = w(u,v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Let $p = (v_0, v_1, \ldots, v_k)$ be any path

Correctness of Johnson's Algorithm

$$\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Let $p = (v_0, v_1, \dots, v_k)$ be any path

- In the original graph, the weight is $\sum_{i=1}^{k} w(v_{i-1}, v_i) = w(p)$.

Correctness of Johnson's Algorithm

$$\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Let $p = (v_0, v_1, \ldots, v_k)$ be any path

- In the original graph, the weight is $\sum_{i=1}^{k} w(v_{i-1}, v_i) = w(p)$.
- In the reweighted graph, the weight is

$$\sum_{i=1}^{k} \widetilde{w}(v_{i-1}, v_i)$$

Correctness of Johnson's Algorithm

$$\widetilde{w}(u,v) = w(u,v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Let $p = (v_0, v_1, \ldots, v_k)$ be any path

- In the original graph, the weight is $\sum_{i=1}^{k} w(v_{i-1}, v_i) = w(p)$.
- In the reweighted graph, the weight is

$$\sum_{i=1}^{k} \widetilde{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + v_{i-1}.\delta - v_i.\delta\right)$$

Correctness of Johnson's Algorithm

$$\widetilde{w}(u, v) = w(u, v) + u.\delta - v.\delta$$

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Let $p = (v_0, v_1, \ldots, v_k)$ be any path

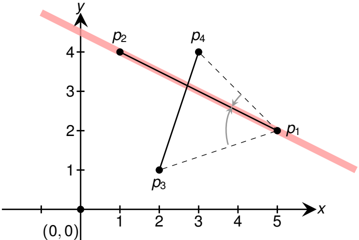
- In the original graph, the weight is $\sum_{i=1}^{k} w(v_{i-1}, v_i) = w(p)$.
- In the reweighted graph, the weight is

$$\sum_{i=1}^{k} \widetilde{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + v_{i-1}.\delta - v_i.\delta\right) = w(p) + v_0.\delta - v_k.\delta \qquad \square$$

Comparison of all Shortest-Path Algorithms

Algorithm	SSSP		APSP		negative weights
	sparse	dense	sparse	dense	
Bellman-Ford	V^2	V^3	V^3	V^4	✓
Dijkstra	$V \log V$	V^2	$V^2 \log V$	V^3	X
Matrix Mult.	–	–	$V^3 \log V$	$V^3 \log V$	(✓)
Johnson	–	–	$V^2 \log V$	V^3	✓

can handle negative weight edges,
but not negative weight cycles



$$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$$

$$(p_4 - p_1) \times (p_2 - p_1) = (-2, 2) \times (-4, 2) = 4$$

7: Geometric Algorithms

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Introduction and Line Intersection

Convex Hull

Glimpse at (More) Advanced Algorithms

Introduction

Computational Geometry

- Branch that studies algorithms for geometric problems

Introduction

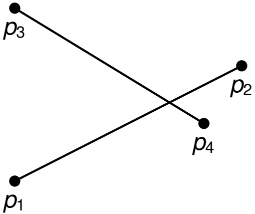
Computational Geometry

- Branch that studies algorithms for geometric problems
- typically, input is a set of points, line segments etc.

Introduction

Computational Geometry

- Branch that studies algorithms for geometric problems
- typically, input is a set of points, line segments etc.



Do these lines intersect?

Introduction

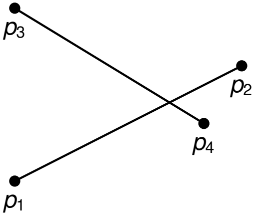
Computational Geometry

- Branch that studies algorithms for geometric problems
- typically, input is a set of points, line segments etc.

Applications

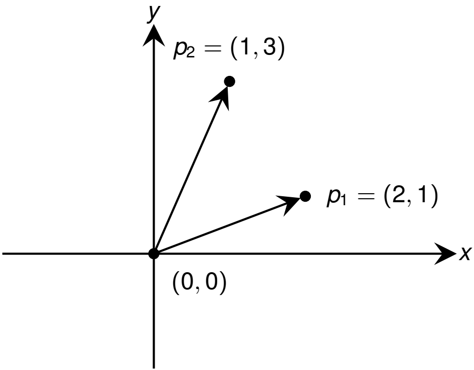
- computer graphics
- computer vision
- textile layout
- VLSI design

⋮

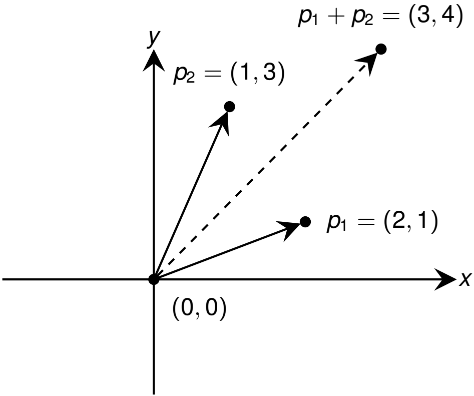


Do these lines intersect?

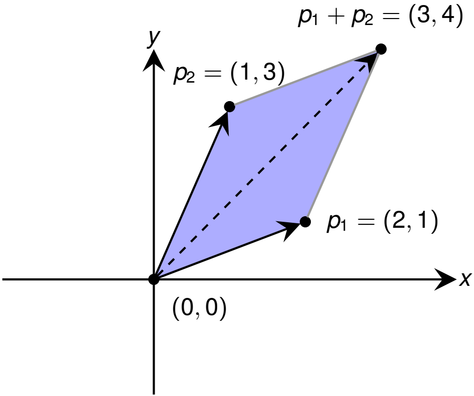
Cross Product (Area)



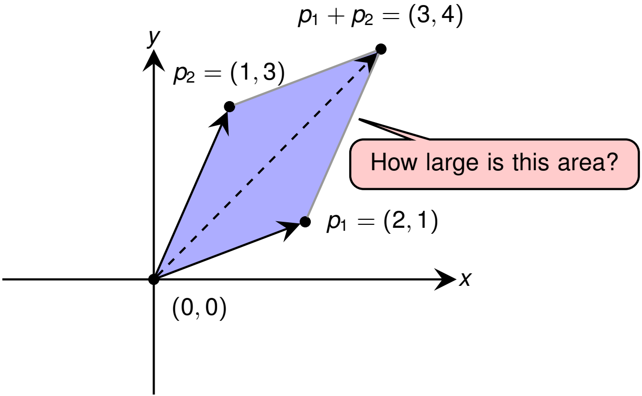
Cross Product (Area)



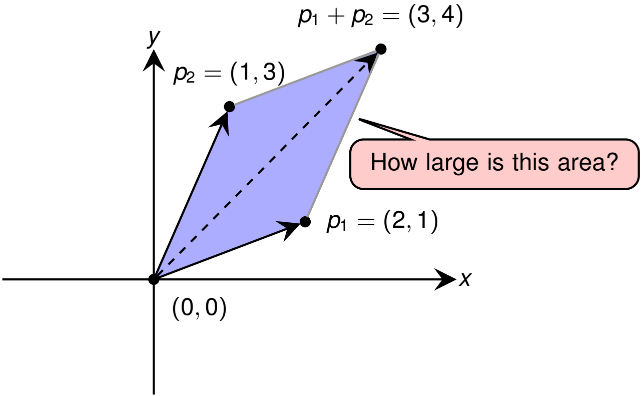
Cross Product (Area)



Cross Product (Area)

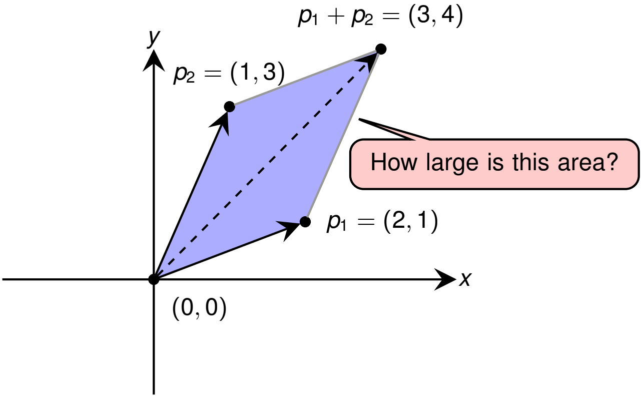


Cross Product (Area)



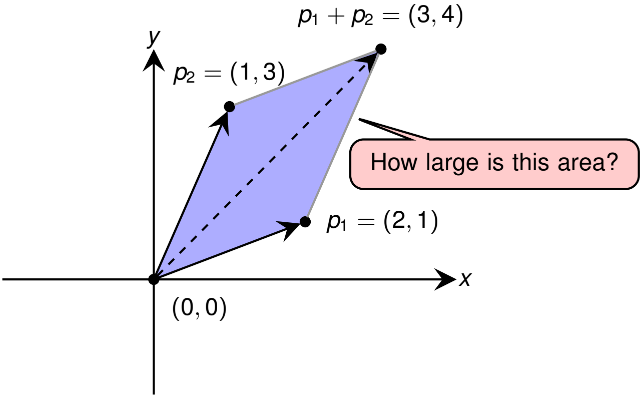
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$

Cross Product (Area)



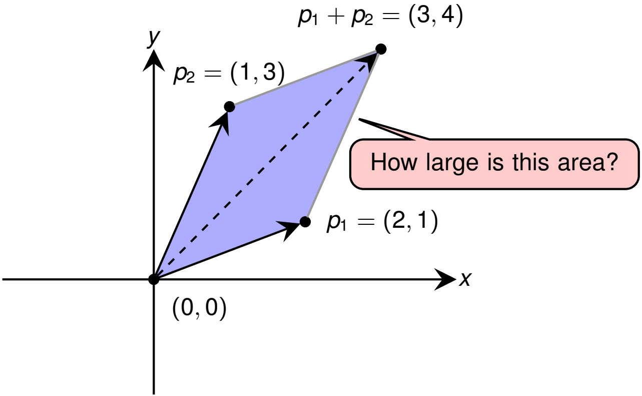
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1$$

Cross Product (Area)



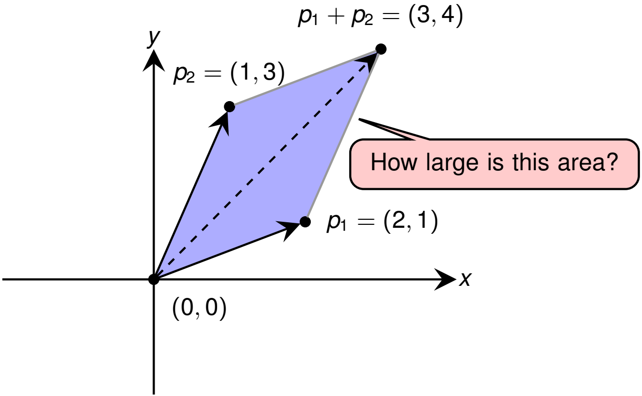
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1$$

Cross Product (Area)



$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

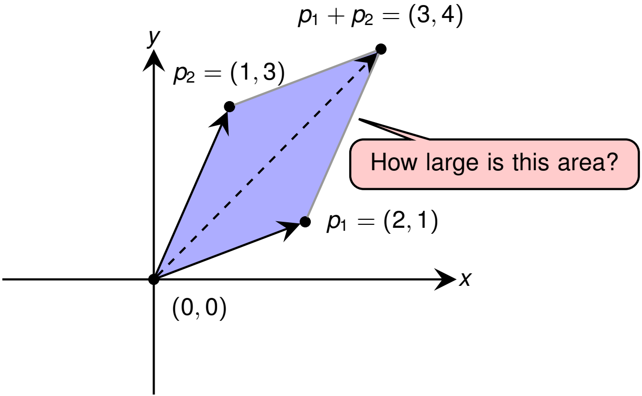
Cross Product (Area)



$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

$p_2 \times p_1$

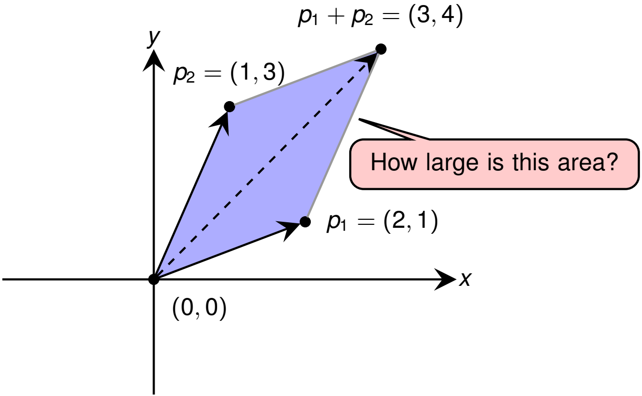
Cross Product (Area)



$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

$$p_2 \times p_1 = y_1 x_2 - y_2 x_1$$

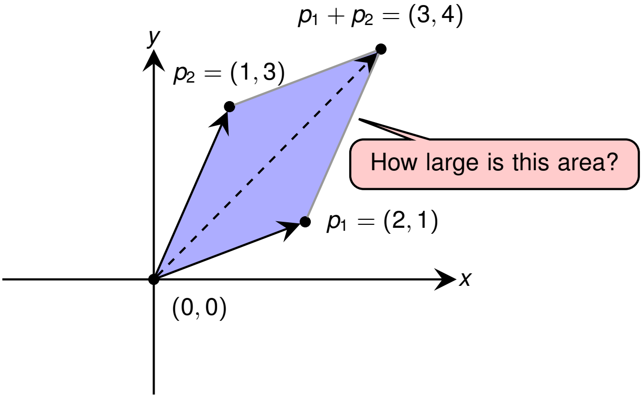
Cross Product (Area)



$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

$$p_2 \times p_1 = y_1 x_2 - y_2 x_1 = -(p_1 \times p_2)$$

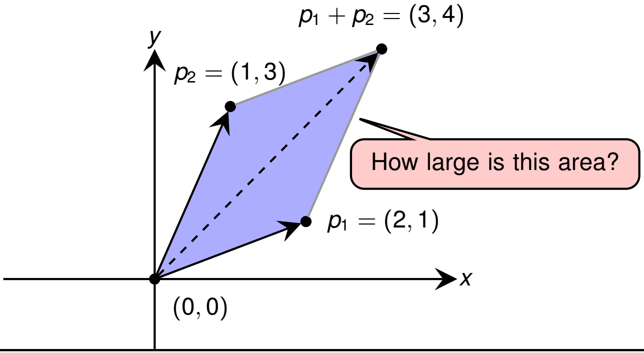
Cross Product (Area)



$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

$$p_2 \times p_1 = y_1 x_2 - y_2 x_1 = -(p_1 \times p_2) = -5$$

Cross Product (Area)



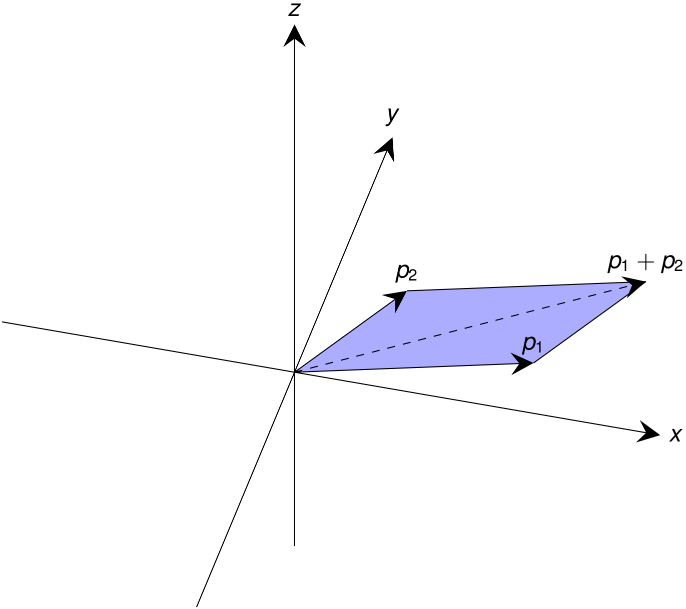
Alternatively, one could take the dot-product (but not used here):

$$p_1 \cdot p_2 = \|p_1\| \cdot \|p_2\| \cdot \cos(\phi).$$

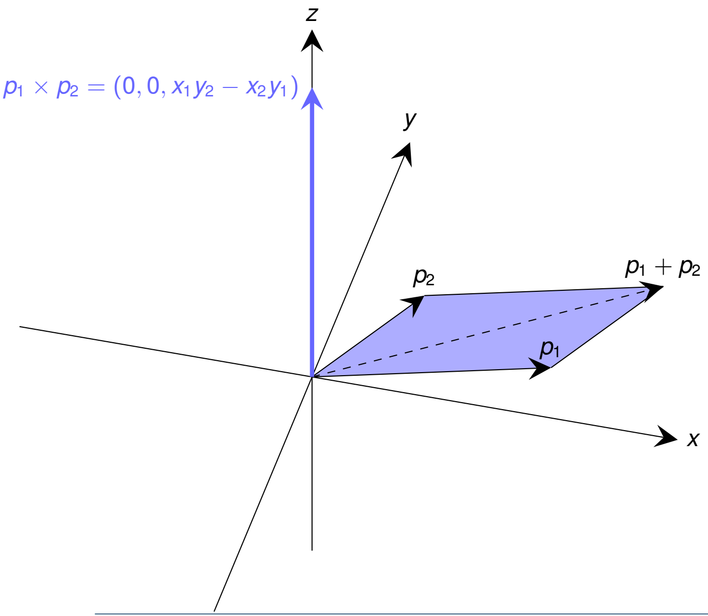
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

$$p_2 \times p_1 = y_1 x_2 - y_2 x_1 = -(p_1 \times p_2) = -5$$

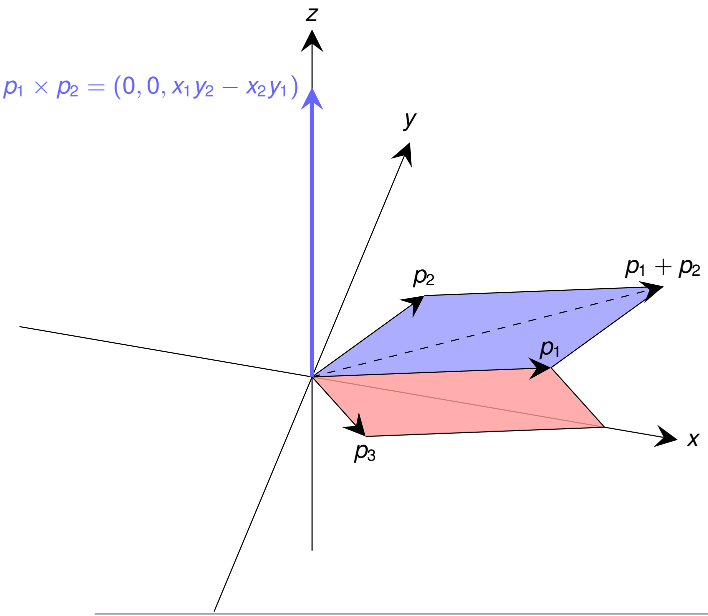
Cross Product in 3D



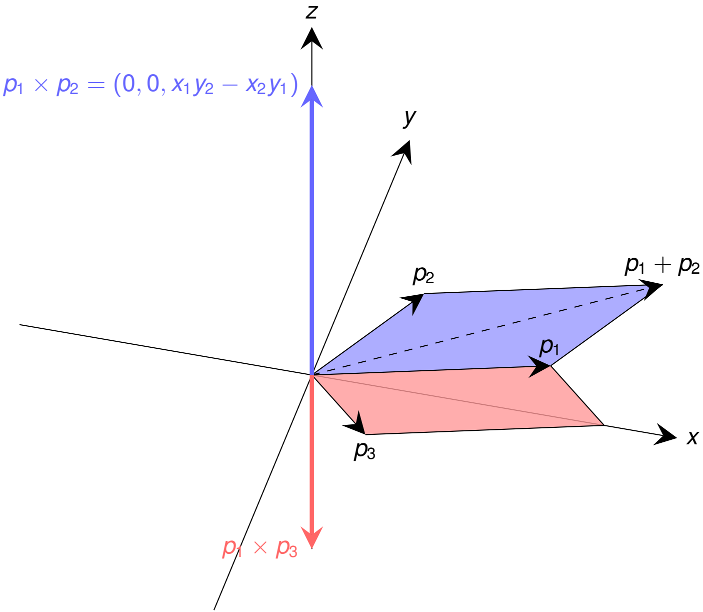
Cross Product in 3D



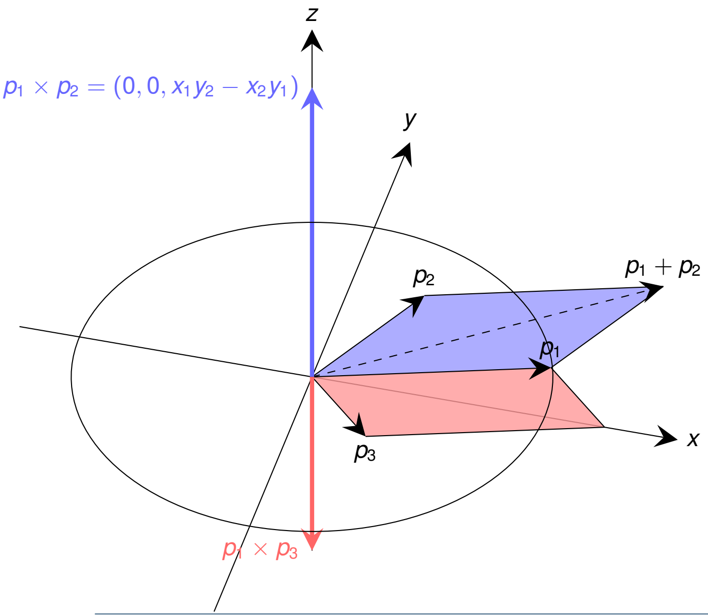
Cross Product in 3D



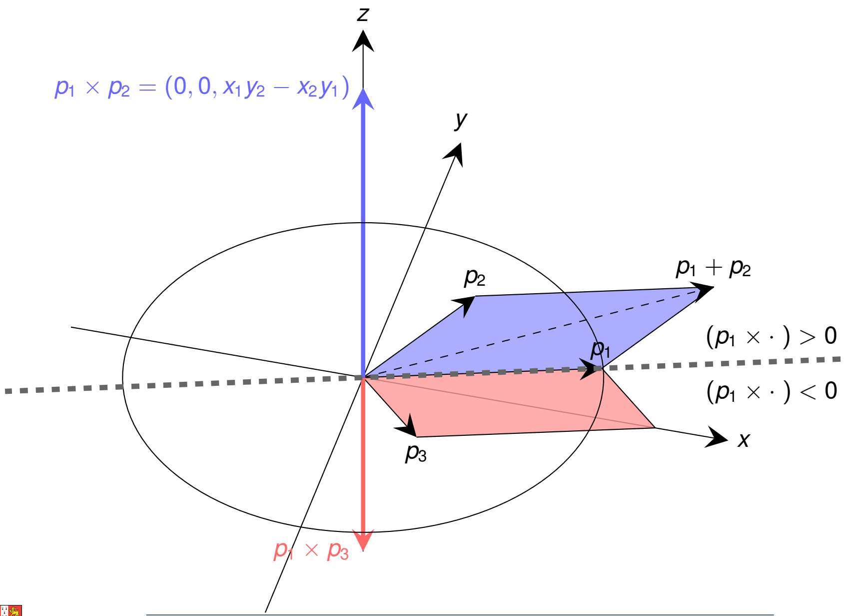
Cross Product in 3D



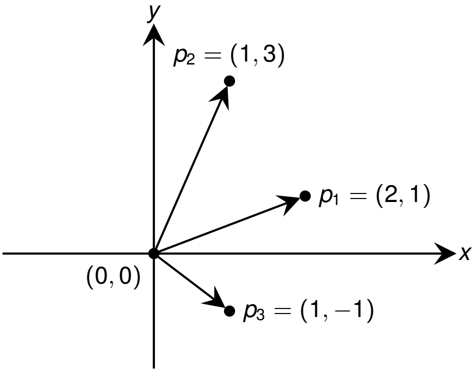
Cross Product in 3D



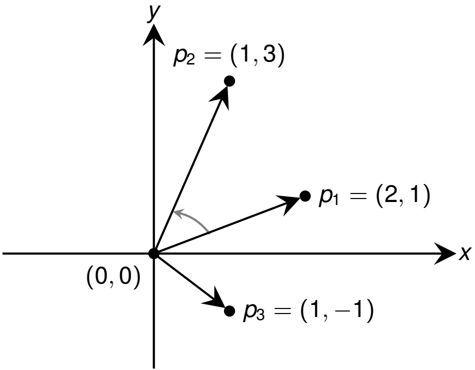
Cross Product in 3D



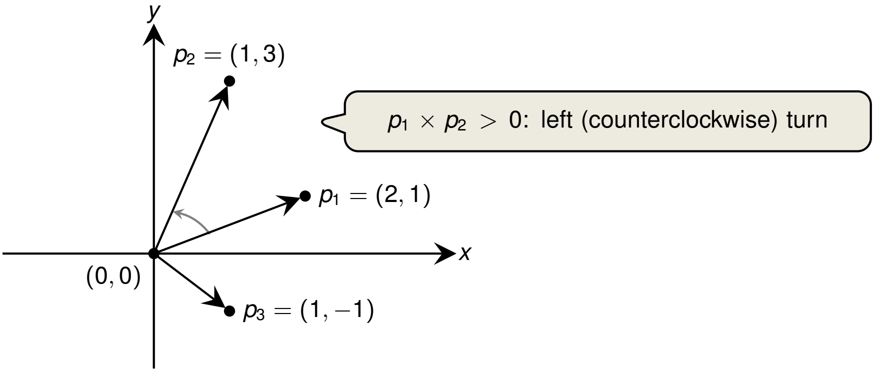
Using Cross product to determine Turns



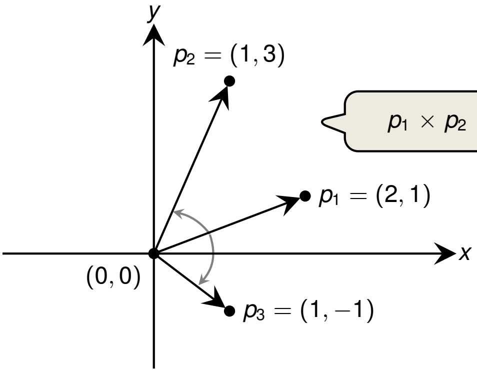
Using Cross product to determine Turns



Using Cross product to determine Turns

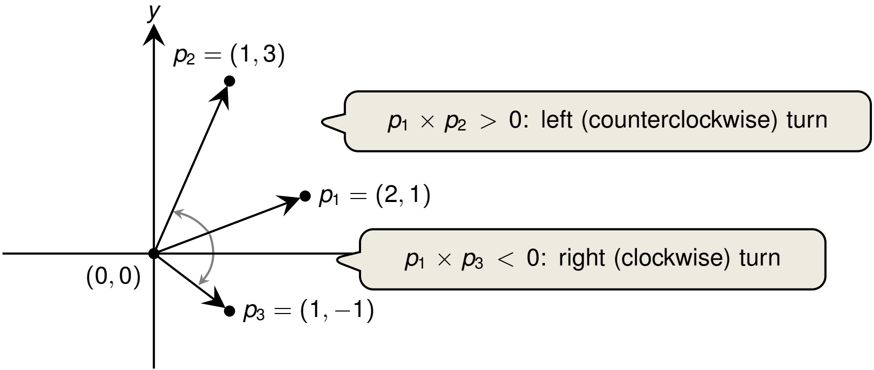


Using Cross product to determine Turns

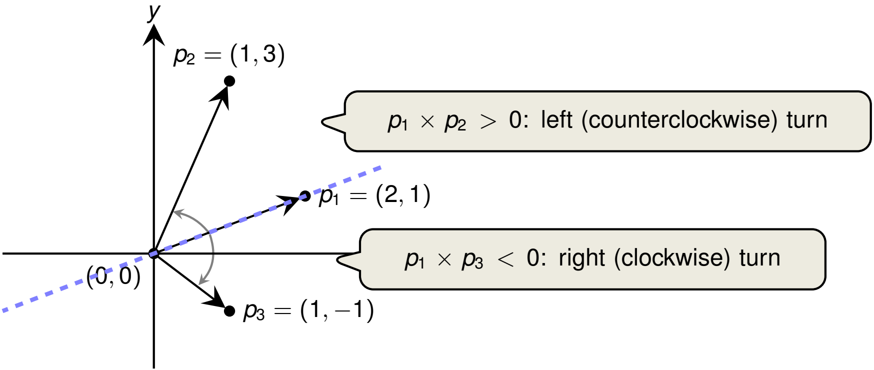


$p_1 \times p_2 > 0$: left (counterclockwise) turn

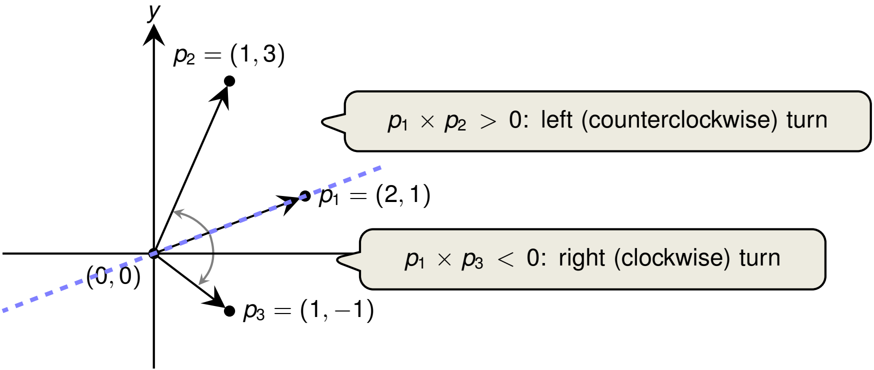
Using Cross product to determine Turns



Using Cross product to determine Turns

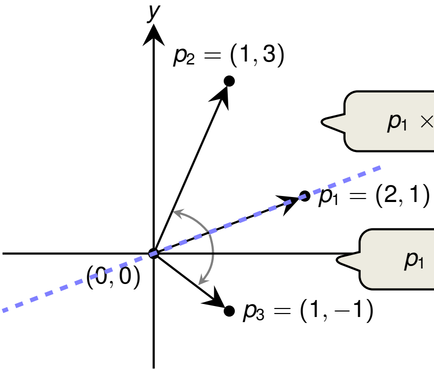


Using Cross product to determine Turns



Sign of cross product determines turn!

Using Cross product to determine Turns



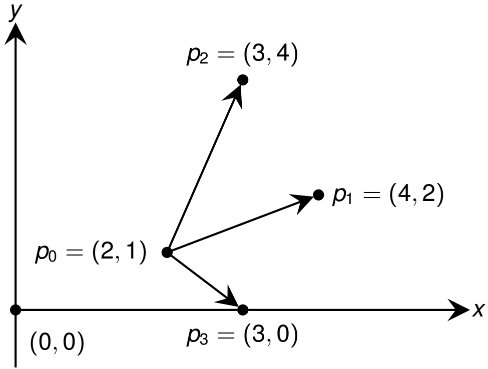
$p_1 \times p_2 > 0$: left (counterclockwise) turn

$p_1 \times p_3 < 0$: right (clockwise) turn

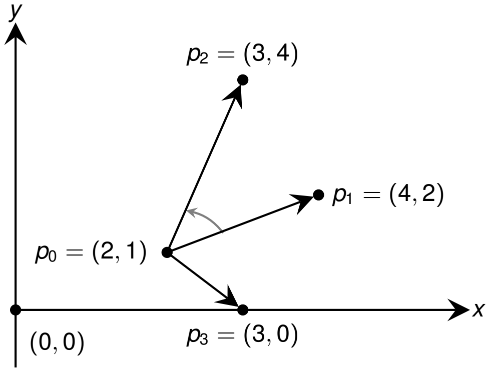
Sign of cross product determines turn!

Cross product equals zero iff vectors are colinear

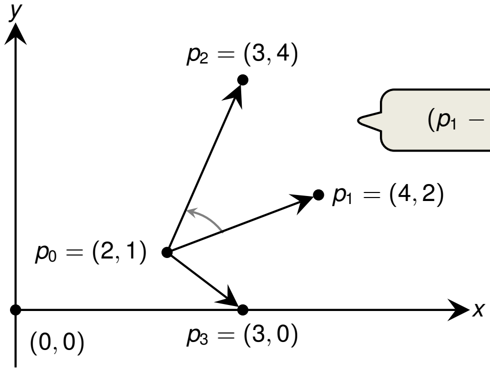
Using Cross product to determine Turns (origin shifted)



Using Cross product to determine Turns (origin shifted)

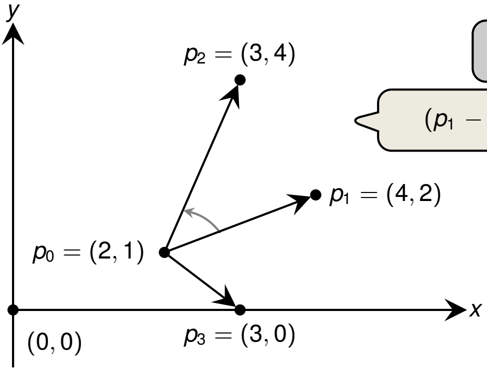


Using Cross product to determine Turns (origin shifted)



$(p_1 - p_0) \times (p_2 - p_0) > 0$: left turn

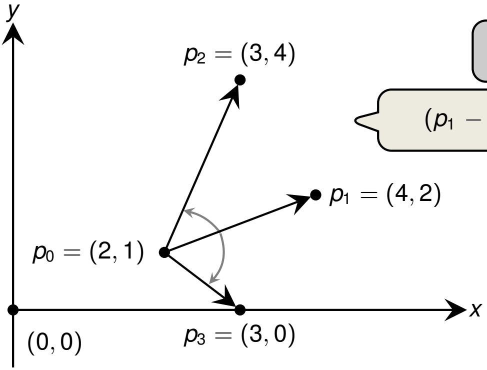
Using Cross product to determine Turns (origin shifted)



$(2,1) \times (1,3) = 5$

$(p_1 - p_0) \times (p_2 - p_0) > 0$: left turn

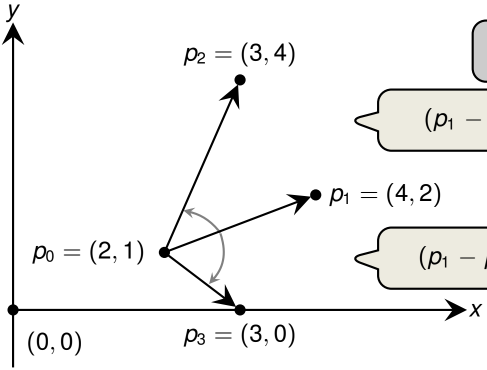
Using Cross product to determine Turns (origin shifted)



$(2,1) \times (1,3) = 5$

$(p_1 - p_0) \times (p_2 - p_0) > 0$: left turn

Using Cross product to determine Turns (origin shifted)

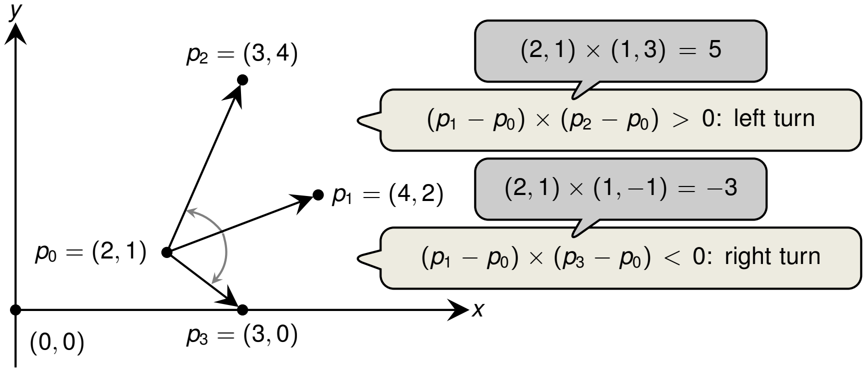


$(2,1) \times (1,3) = 5$

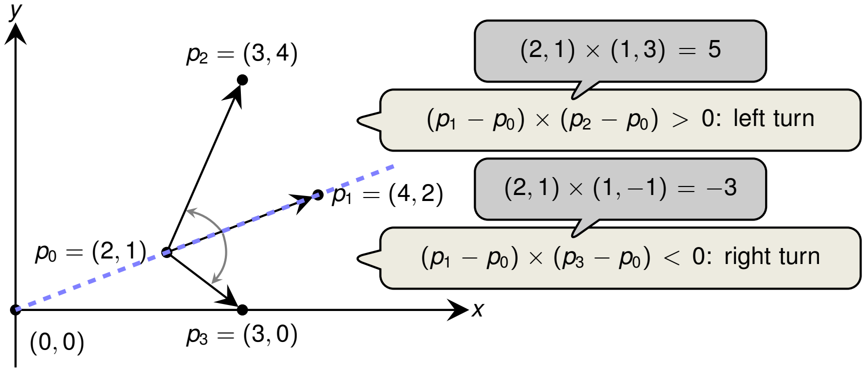
$(p_1 - p_0) \times (p_2 - p_0) > 0$: left turn

$(p_1 - p_0) \times (p_3 - p_0) < 0$: right turn

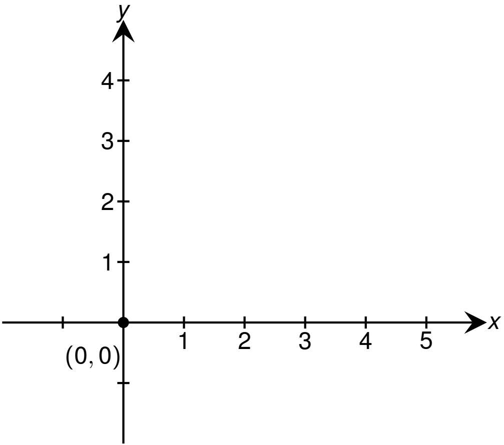
Using Cross product to determine Turns (origin shifted)



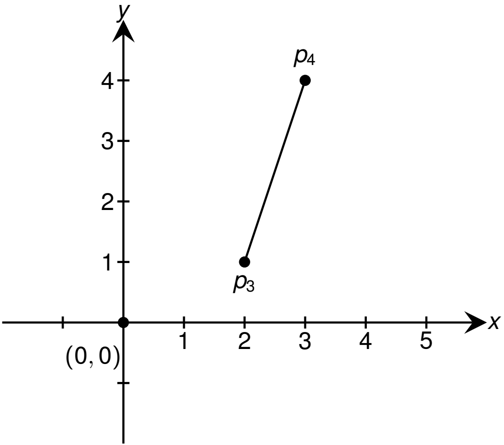
Using Cross product to determine Turns (origin shifted)



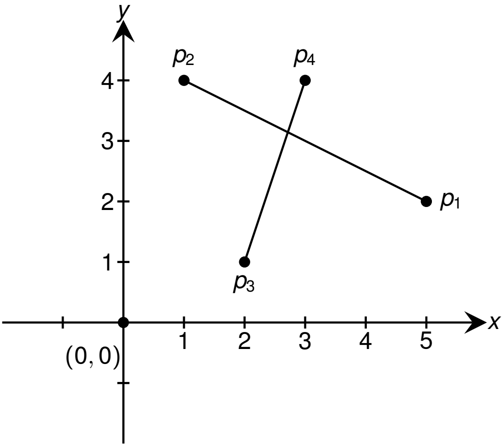
Solving Line Intersection



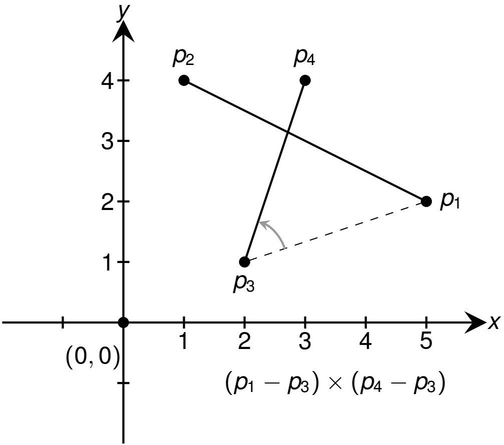
Solving Line Intersection



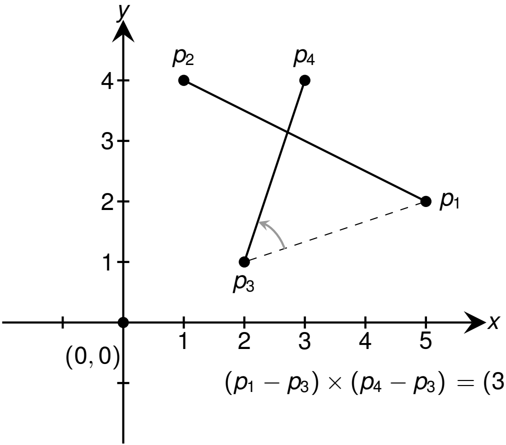
Solving Line Intersection



Solving Line Intersection

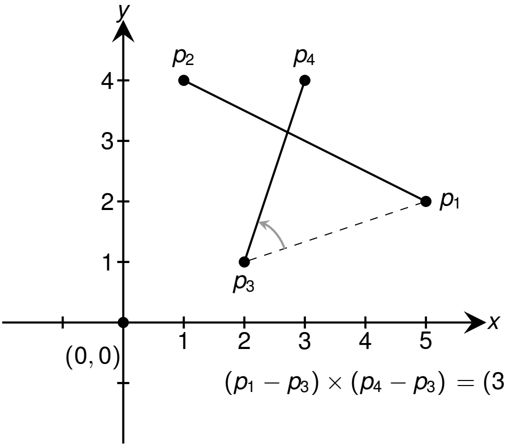


Solving Line Intersection



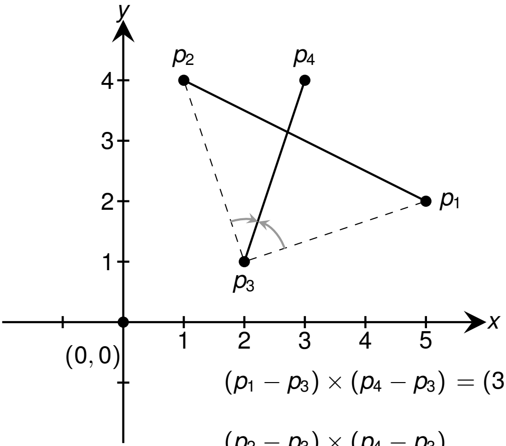
$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3)$$

Solving Line Intersection



$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

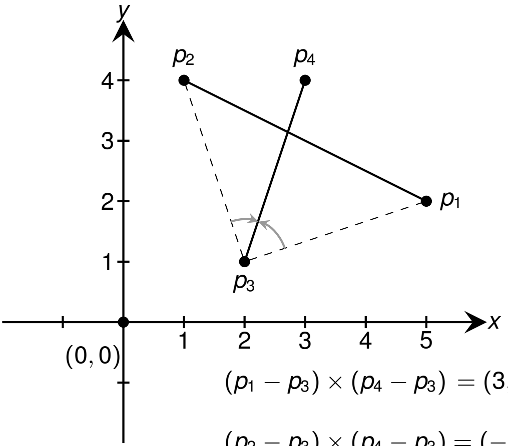
Solving Line Intersection



$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

$$(p_2 - p_3) \times (p_4 - p_3)$$

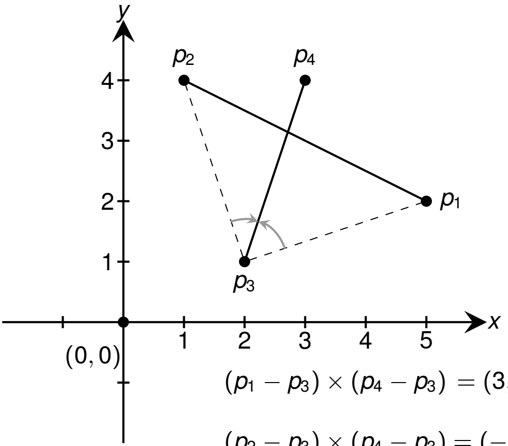
Solving Line Intersection



$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

$$(p_2 - p_3) \times (p_4 - p_3) = (-1, 3) \times (1, 3)$$

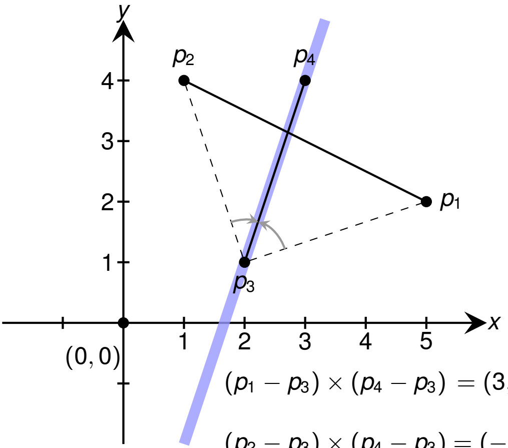
Solving Line Intersection



$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

$$(p_2 - p_3) \times (p_4 - p_3) = (-1, 3) \times (1, 3) = -6$$

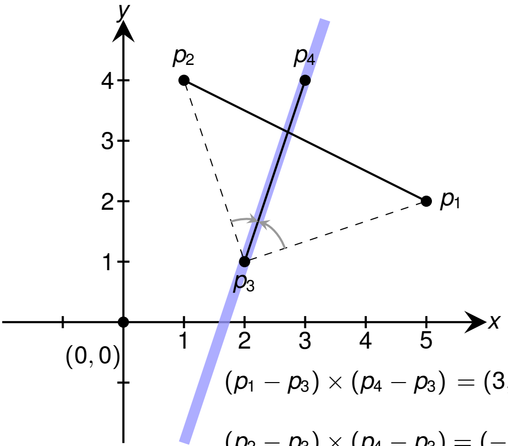
Solving Line Intersection



$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

$$(p_2 - p_3) \times (p_4 - p_3) = (-1, 3) \times (1, 3) = -6$$

Solving Line Intersection

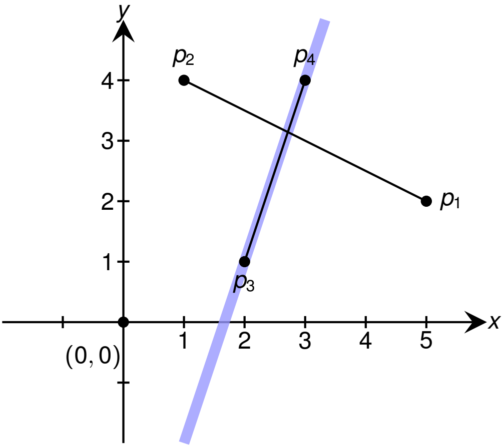


$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

$$(p_2 - p_3) \times (p_4 - p_3) = (-1, 3) \times (1, 3) = -6$$

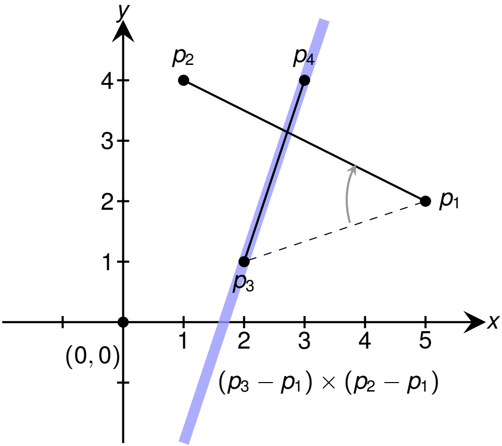
Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection



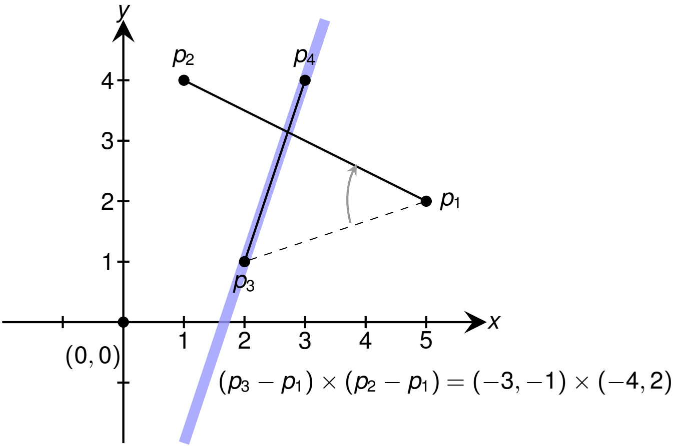
Opposite signs $\Rightarrow$ $\overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection



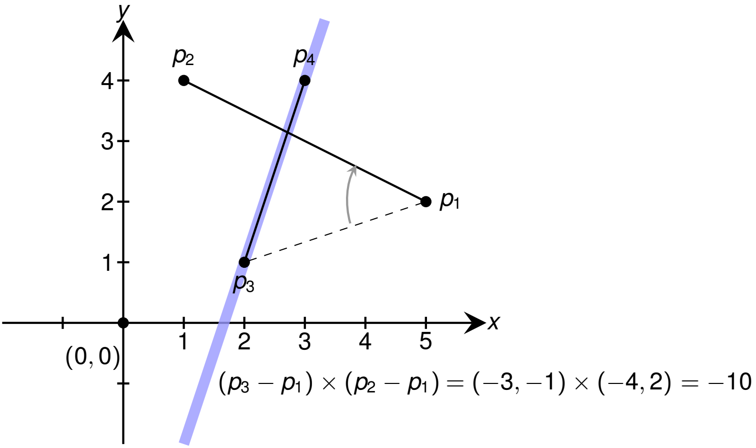
Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection



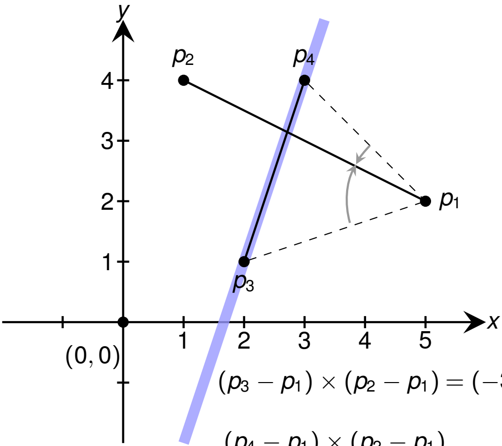
Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection



Opposite signs $\Rightarrow$ $\overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection

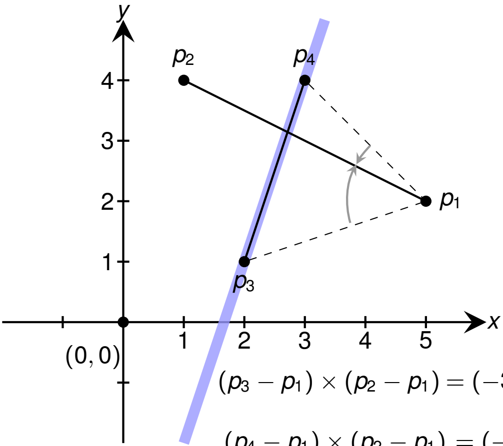


$$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$$

$$(p_4 - p_1) \times (p_2 - p_1)$$

Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection

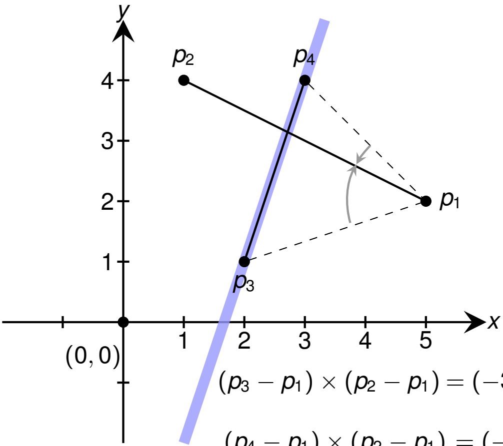


$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$

$(p_4 - p_1) \times (p_2 - p_1) = (-2, 2) \times (-4, 2)$

Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection

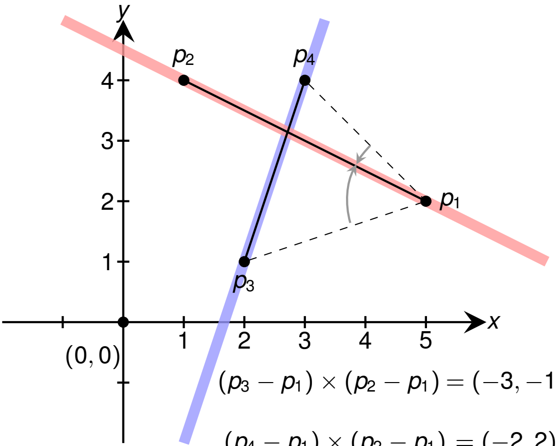


$$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$$

$$(p_4 - p_1) \times (p_2 - p_1) = (-2, 2) \times (-4, 2) = 4$$

Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection



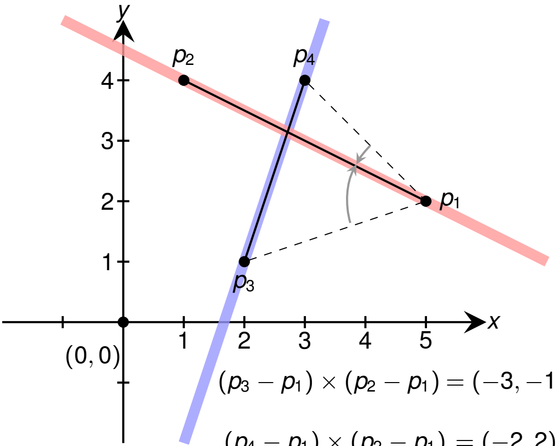
$$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$$

$$(p_4 - p_1) \times (p_2 - p_1) = (-2, 2) \times (-4, 2) = 4$$

Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow \overline{p_3 p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection



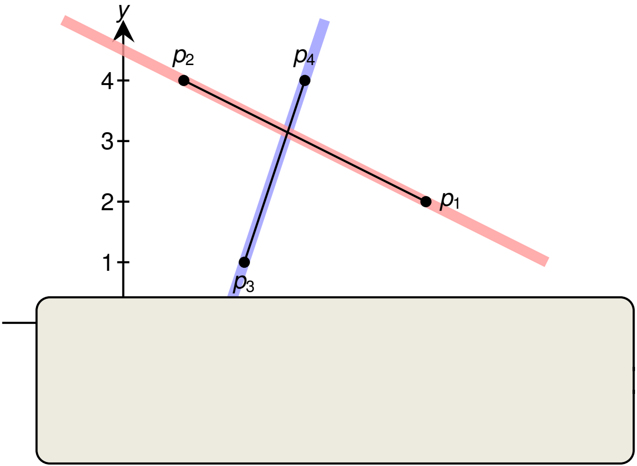
$$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$$

$$(p_4 - p_1) \times (p_2 - p_1) = (-2, 2) \times (-4, 2) = 4$$

Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow \overline{p_3 p_4}$ crosses (infinite) line through p_1 and p_2

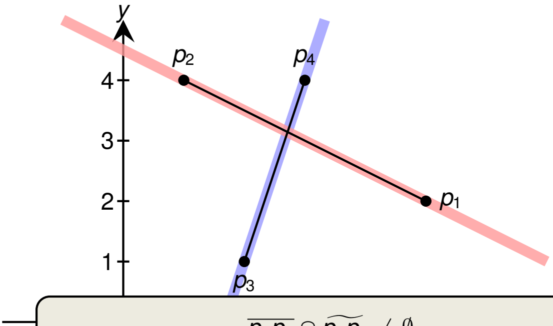
Solving Line Intersection



Opposite signs ⇒ $\overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs ⇒ $\overline{p_3 p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection

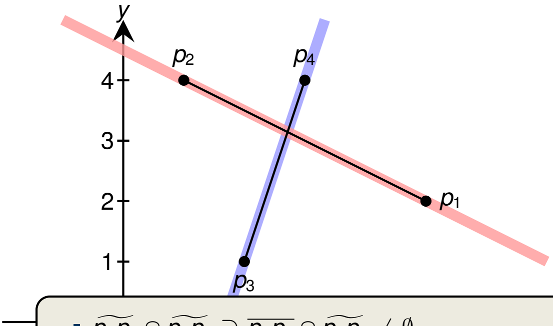


$$\overline{p_1p_2} \cap \widetilde{p_3p_4} \neq \emptyset$$
$$\widetilde{p_1p_2} \cap \overline{p_3p_4} \neq \emptyset$$

Opposite signs $\Rightarrow$ $\overline{p_1p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow$ $\overline{p_3p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection

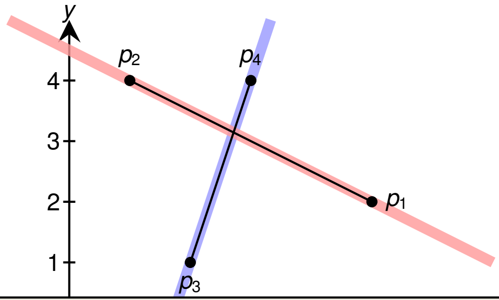


- $\widetilde{p_1p_2} \cap \widetilde{p_3p_4} \supseteq \overline{p_1p_2} \cap \widetilde{p_3p_4} \neq \emptyset$
- $\widetilde{p_1p_2} \cap \widetilde{p_3p_4} \supseteq \widetilde{p_1p_2} \cap \overline{p_3p_4} \neq \emptyset$

Opposite signs $\Rightarrow$ $\overline{p_1p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow$ $\overline{p_3p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection

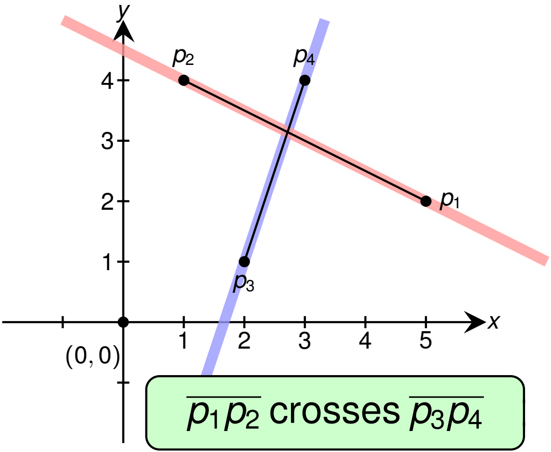


- $\widetilde{p_1p_2} \cap \widetilde{p_3p_4} \supseteq \overline{p_1p_2} \cap \widetilde{p_3p_4} \neq \emptyset$
- $\widetilde{p_1p_2} \cap \widetilde{p_3p_4} \supseteq \widetilde{p_1p_2} \cap \overline{p_3p_4} \neq \emptyset$
- Since $\widetilde{p_1p_2} \cap \widetilde{p_3p_4}$ consists of (at most) one point $\Rightarrow \overline{p_1p_2} \cap \overline{p_3p_4} \neq \emptyset$

Opposite signs $\Rightarrow \overline{p_1p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow \overline{p_3p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection

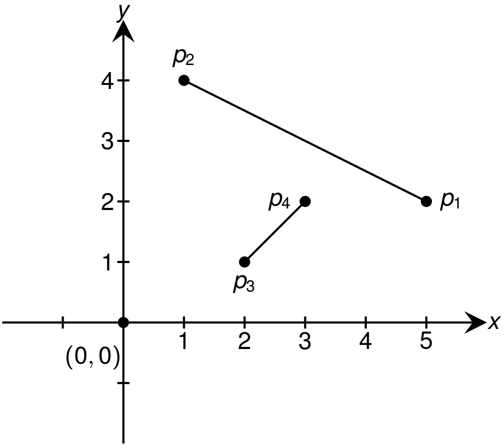


$\overline{p_1 p_2}$ crosses $\overline{p_3 p_4}$

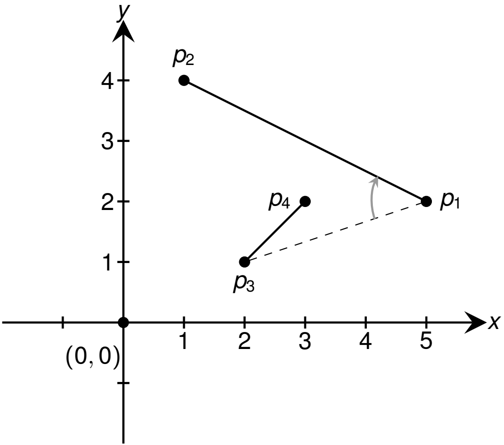
Opposite signs $\Rightarrow$ $\overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow$ $\overline{p_3 p_4}$ crosses (infinite) line through p_1 and p_2

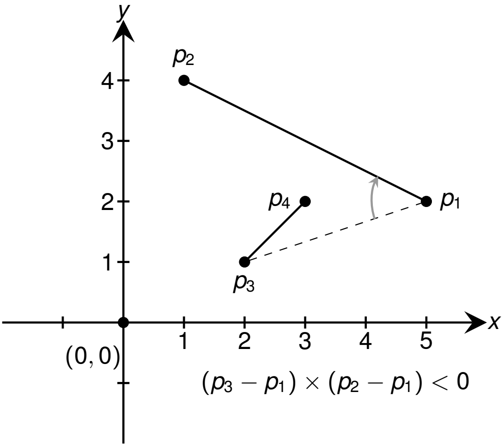
Solving Line Intersection



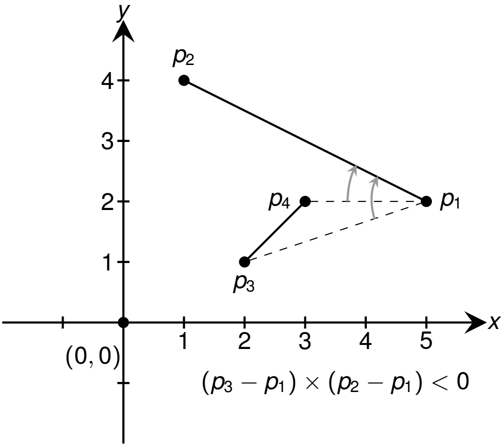
Solving Line Intersection



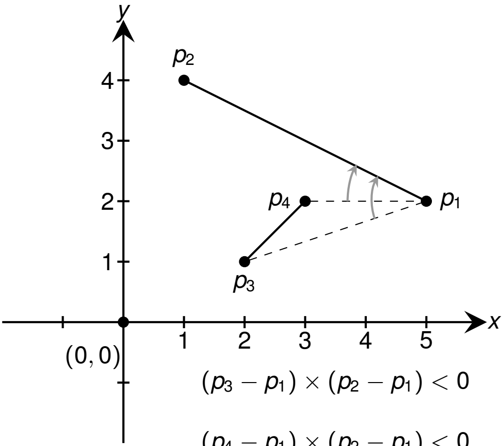
Solving Line Intersection



Solving Line Intersection



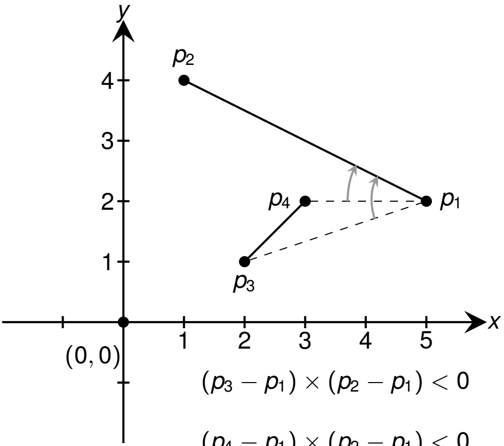
Solving Line Intersection



$$(p_3 - p_1) \times (p_2 - p_1) < 0$$

$$(p_4 - p_1) \times (p_2 - p_1) < 0$$

Solving Line Intersection

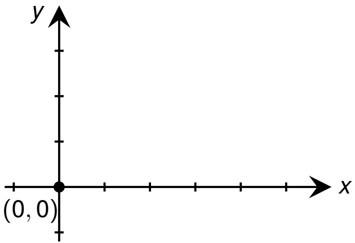


$$(p_3 - p_1) \times (p_2 - p_1) < 0$$

$$(p_4 - p_1) \times (p_2 - p_1) < 0$$

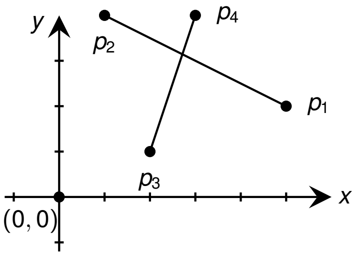
$\overline{p_1 p_2}$ does **not** cross $\overline{p_3 p_4}$

Solving Line Intersection



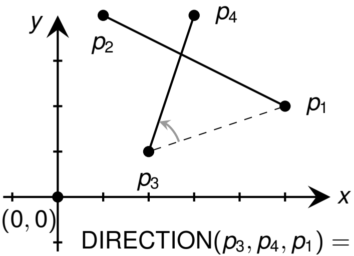
0: $\text{DIRECTION}(p_i, p_j, p_k)$
1: return $(p_k - p_i) \times (p_j - p_i)$

Solving Line Intersection



DIRECTION(p_i, p_j, p_k)
    return $(p_k - p_i) \times (p_j - p_i)$

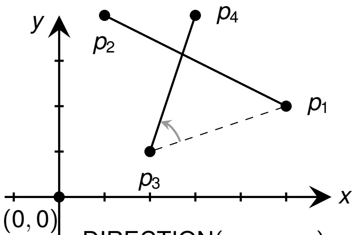
Solving Line Intersection



$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

DIRECTION(p_i, p_j, p_k)
    return $(p_k - p_i) \times (p_j - p_i)$

Solving Line Intersection

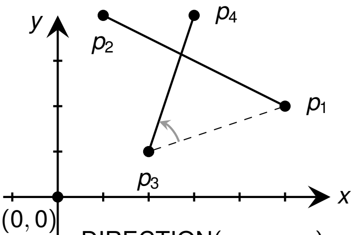


$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

```
DIRECTION(p_i, p_j, p_k)
    return (p_k − p_i) × (p_j − p_i)
```

```
SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)
    d_1 = DIRECTION(p_3, p_4, p_1)
    d_2 = DIRECTION(p_3, p_4, p_2)
    d_3 = DIRECTION(p_1, p_2, p_3)
    d_4 = DIRECTION(p_1, p_2, p_4)
    If d_1 · d_2 < 0 and d_3 · d_4 < 0 return TRUE
        ··· (handle all degenerate cases)
```

Solving Line Intersection

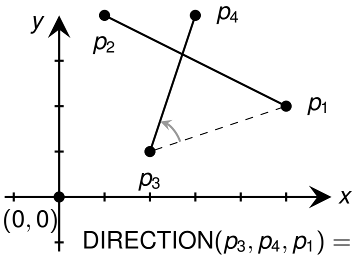


$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

```
DIRECTION(p_i, p_j, p_k)
    return (p_k − p_i) × (p_j − p_i)
```

```
SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)
    d_1 = DIRECTION(p_3, p_4, p_1)
    d_2 = DIRECTION(p_3, p_4, p_2)
    d_3 = DIRECTION(p_1, p_2, p_3)
    d_4 = DIRECTION(p_1, p_2, p_4)
    If d_1 · d_2 < 0 and d_3 · d_4 < 0 return TRUE
        ··· (handle all degenerate cases)
```

Solving Line Intersection



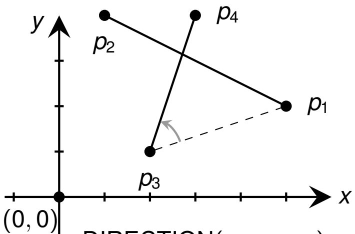
$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

```
DIRECTION(p_i, p_j, p_k)
    return (p_k − p_i) × (p_j − p_i)
```

```
SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)
    d_1 = DIRECTION(p_3, p_4, p_1)
    d_2 = DIRECTION(p_3, p_4, p_2)
    d_3 = DIRECTION(p_1, p_2, p_3)
    d_4 = DIRECTION(p_1, p_2, p_4)
    If d_1 · d_2 < 0 and d_3 · d_4 < 0 return TRUE
        · · · (handle all degenerate cases)
```

In total 4 satisfying conditions!

Solving Line Intersection



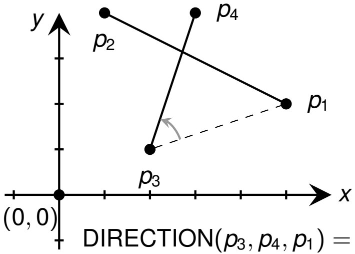
$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

```
DIRECTION(p_i, p_j, p_k)
    return (p_k − p_i) × (p_j − p_i)
```

```
SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)
    d_1 = DIRECTION(p_3, p_4, p_1)
    d_2 = DIRECTION(p_3, p_4, p_2)
    d_3 = DIRECTION(p_1, p_2, p_3)
    d_4 = DIRECTION(p_1, p_2, p_4)
    If d_1 · d_2 < 0 and d_3 · d_4 < 0 return TRUE
        ··· (handle all degenerate cases)
```

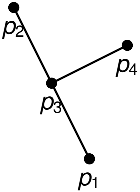
Lines could touch or be colinear

Solving Line Intersection



$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

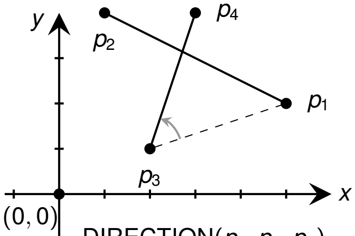
```
DIRECTION(p_i, p_j, p_k)
    return (p_k − p_i) × (p_j − p_i)
```



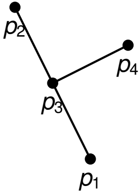

```
SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)
    d_1 = DIRECTION(p_3, p_4, p_1)
    d_2 = DIRECTION(p_3, p_4, p_2)
    d_3 = DIRECTION(p_1, p_2, p_3)
    d_4 = DIRECTION(p_1, p_2, p_4)
    If d_1 · d_2 < 0 and d_3 · d_4 < 0 return TRUE
        · · · (handle all degenerate cases)
```

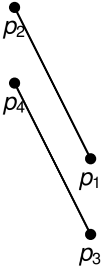
Lines could touch or be colinear

Solving Line Intersection



$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$





DIRECTION(p_i, p_j, p_k)
    return $(p_k - p_i) \times (p_j - p_i)$

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4)
    $d_1 =$ DIRECTION(p_3, p_4, p_1)
    $d_2 =$ DIRECTION(p_3, p_4, p_2)
    $d_3 =$ DIRECTION(p_1, p_2, p_3)
    $d_4 =$ DIRECTION(p_1, p_2, p_4)
    If $d_1 \cdot d_2 < 0$ and $d_3 \cdot d_4 < 0$ return TRUE
        $\cdots$ (handle all degenerate cases)

Lines could touch or be colinear

Outline

Introduction and Line Intersection

Convex Hull

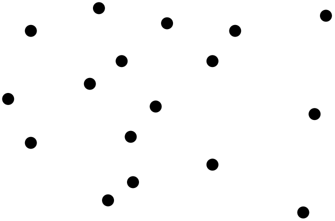
Glimpse at (More) Advanced Algorithms

Convex Hull

Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

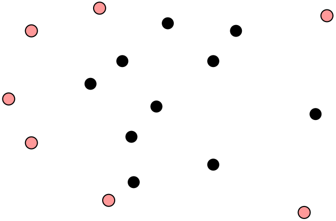
Convex Hull



Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

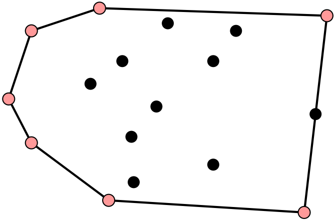
Convex Hull



Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

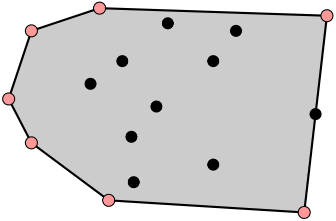
Convex Hull



Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

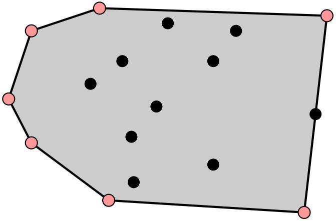
Convex Hull



Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

Convex Hull

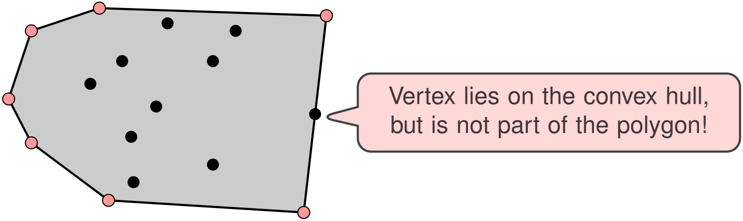


Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

Smallest perimeter fence enclosing the points

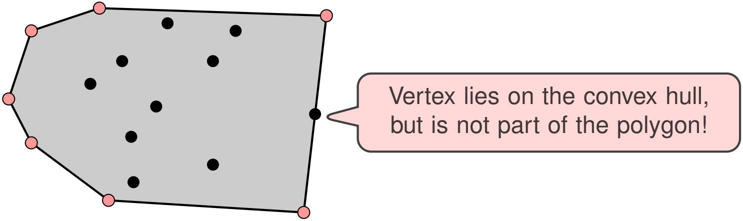
Convex Hull



Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

Convex Hull

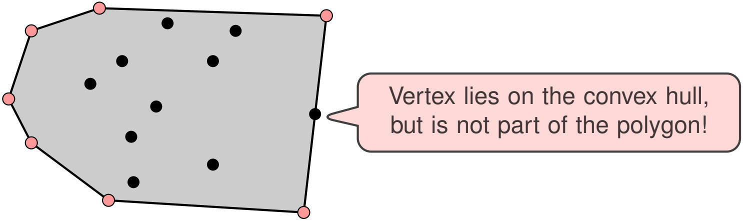


Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

Convex Hull Problem

Convex Hull



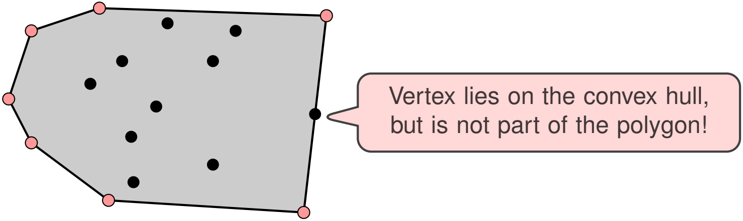
Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

Convex Hull Problem

- Input: set of points Q in the Euclidean space

Convex Hull



Definition

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

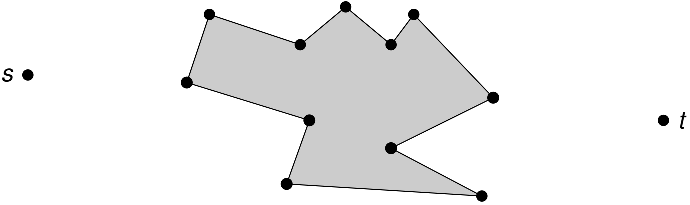
Convex Hull Problem

- Input: set of points Q in the Euclidean space
- Output: return points of the convex hull in counterclockwise order

Application of Convex Hull

Robot Motion Planning

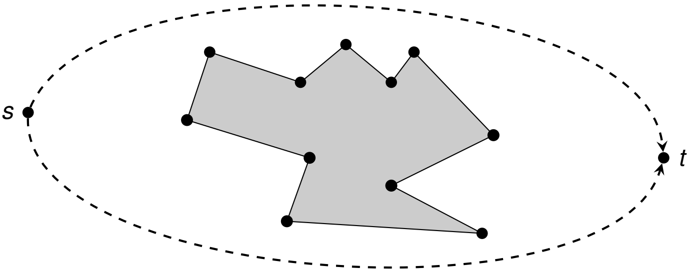
Find shortest path from s to t which avoids a polygonal obstacle.



Application of Convex Hull

Robot Motion Planning

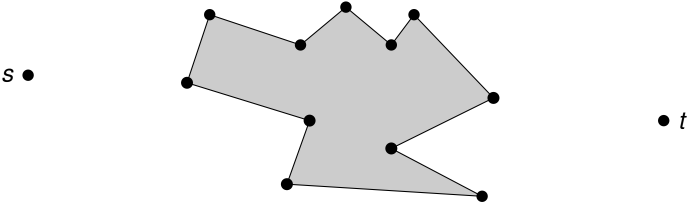
Find shortest path from s to t which avoids a polygonal obstacle.



Application of Convex Hull

Robot Motion Planning

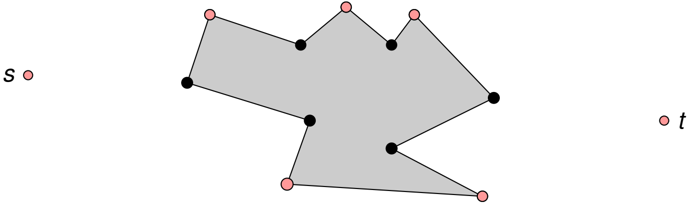
Find shortest path from s to t which avoids a polygonal obstacle.



Application of Convex Hull

Robot Motion Planning

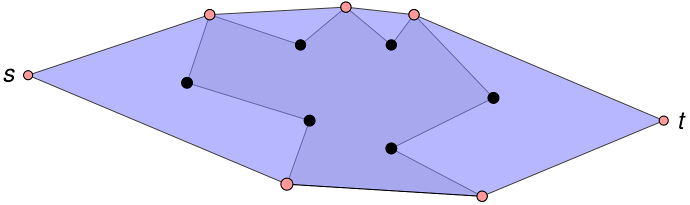
Find shortest path from s to t which avoids a polygonal obstacle.



Application of Convex Hull

Robot Motion Planning

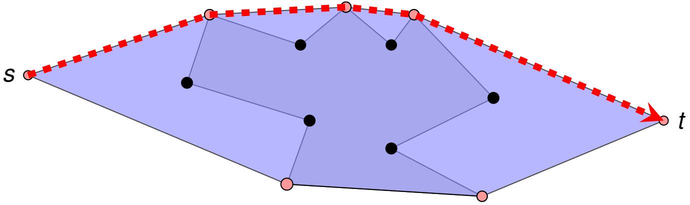
Find shortest path from s to t which avoids a polygonal obstacle.



Application of Convex Hull

Robot Motion Planning

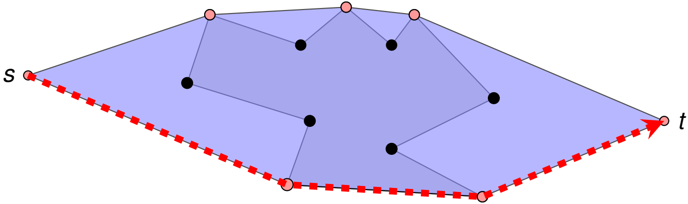
Find shortest path from s to t which avoids a polygonal obstacle.



Application of Convex Hull

Robot Motion Planning

Find shortest path from s to t which avoids a polygonal obstacle.

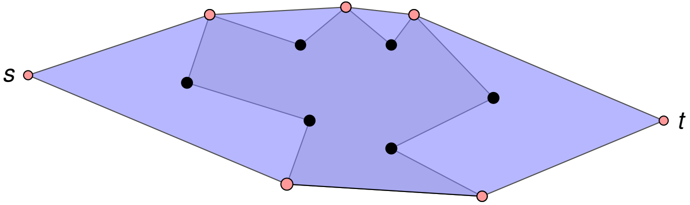


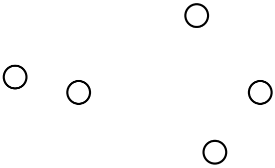
Application of Convex Hull

Robot Motion Planning

Find shortest path from s to t which avoids a polygonal obstacle.

can be solved by computing the Convex hull!

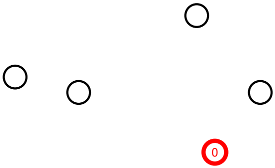




Basic Idea

- Start with the point with smallest y-coordinate

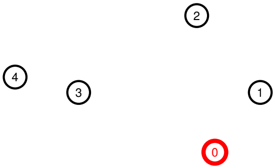
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate

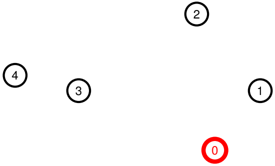
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle

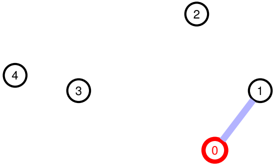
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull

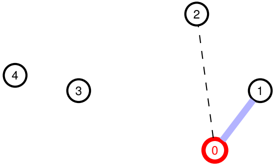
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull

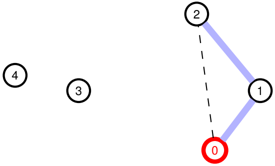
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine

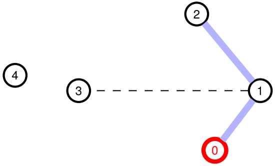
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine

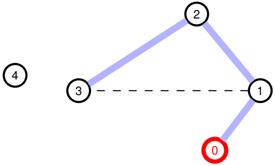
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine

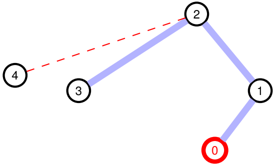
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓

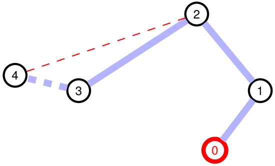
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise,

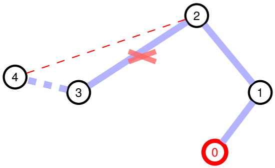
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

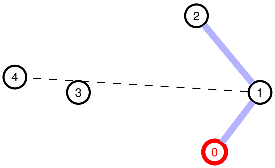
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

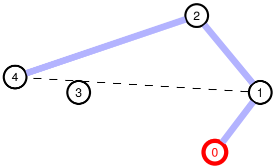
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

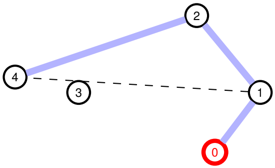
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

Graham's Scan

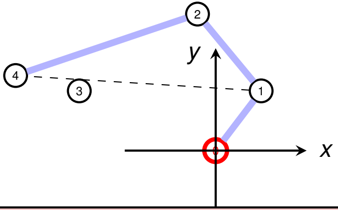


Efficient Sorting by comparing (not computing!) polar angles

Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

Graham's Scan

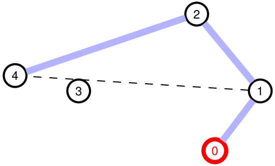


Efficient Sorting by comparing (not computing!) polar angles

Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

Graham's Scan

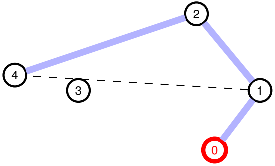


Efficient Sorting by comparing (not computing!) polar angles

Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

Graham's Scan



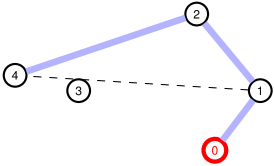
Use Cross Product!

Efficient Sorting by comparing (not computing!) polar angles

Basic Idea

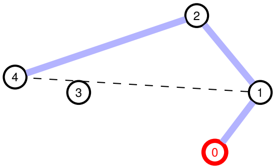
- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

Graham's Scan



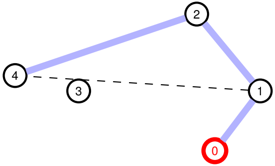

```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

Graham's Scan




```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

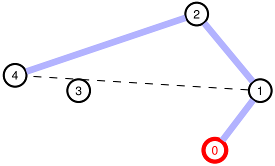
Graham's Scan




```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

Takes $O(n \log n)$ time

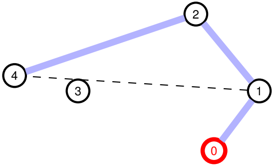
Graham's Scan




```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

Takes $O(n \log n)$ time

Graham's Scan

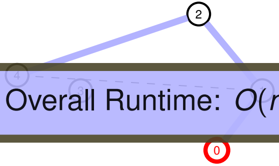



```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

Takes $O(n \log n)$ time

Takes $O(n)$ time, since every point is part of a PUSH or POP at most once.

Graham's Scan



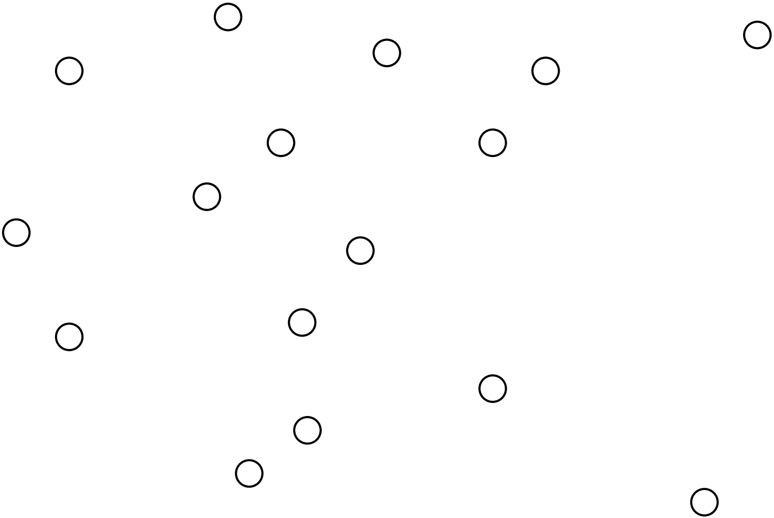
Overall Runtime: $O(n \log n)$

```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

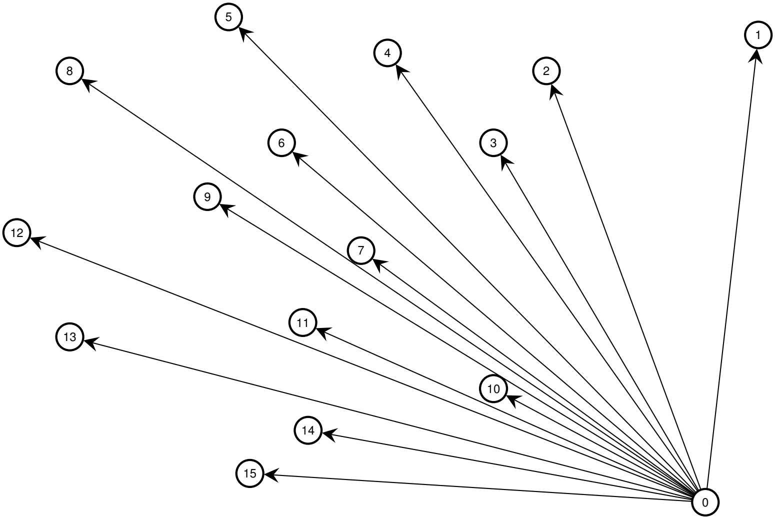
Takes $O(n \log n)$ time

Takes $O(n)$ time, since every point is part of a PUSH or POP at most once.

Execution of Graham's Scan

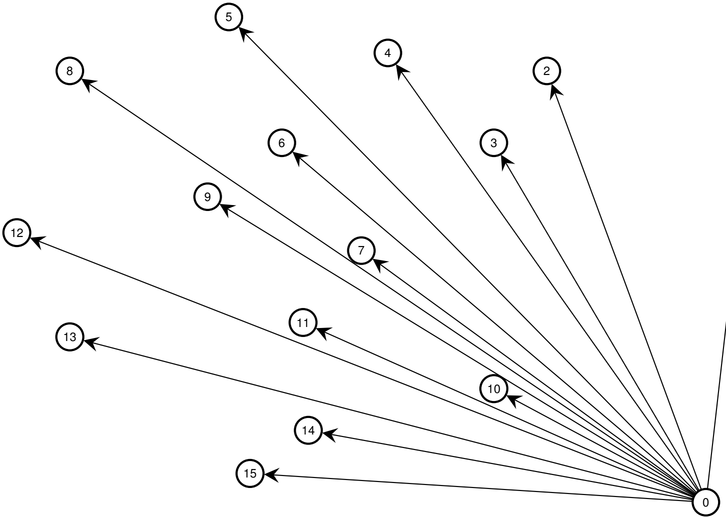


Execution of Graham's Scan



Execution of Graham's Scan

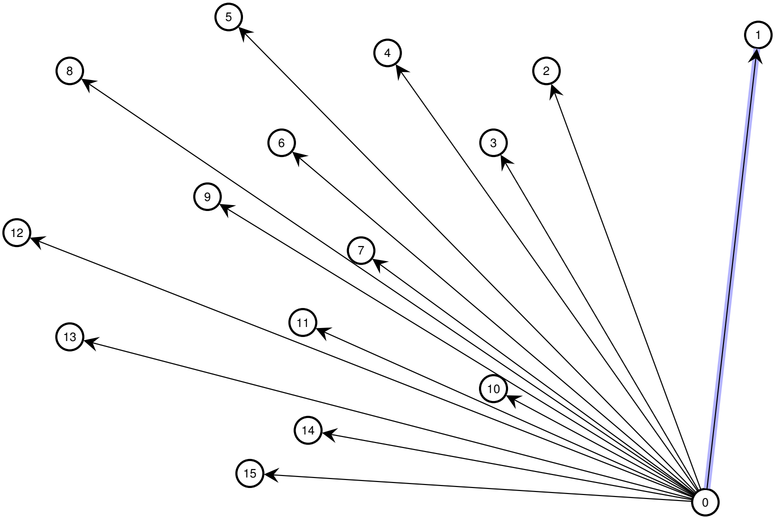
$i = 0$ $\boxed{0}$



Execution of Graham's Scan

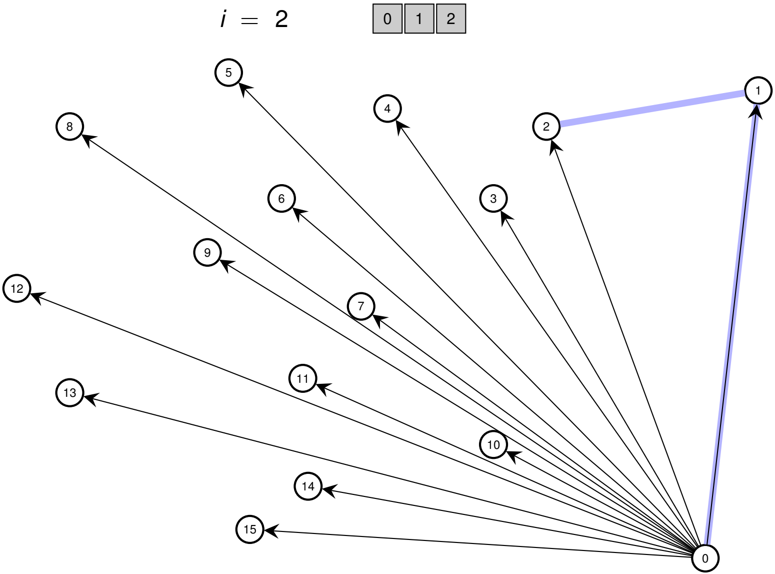
$i = 1$

0	1



Execution of Graham's Scan

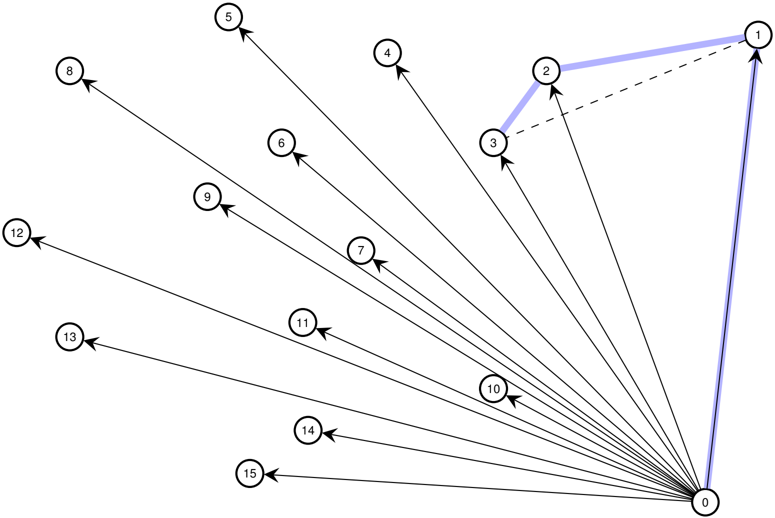
$i = 2$



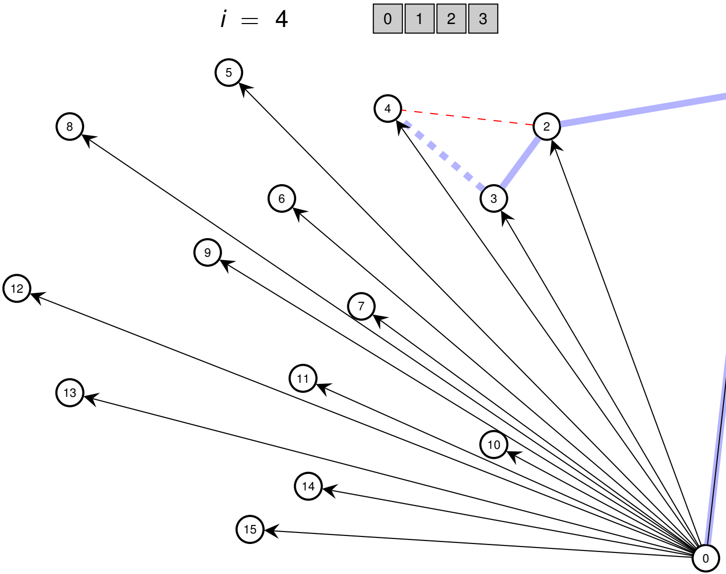
Execution of Graham's Scan

$i = 3$

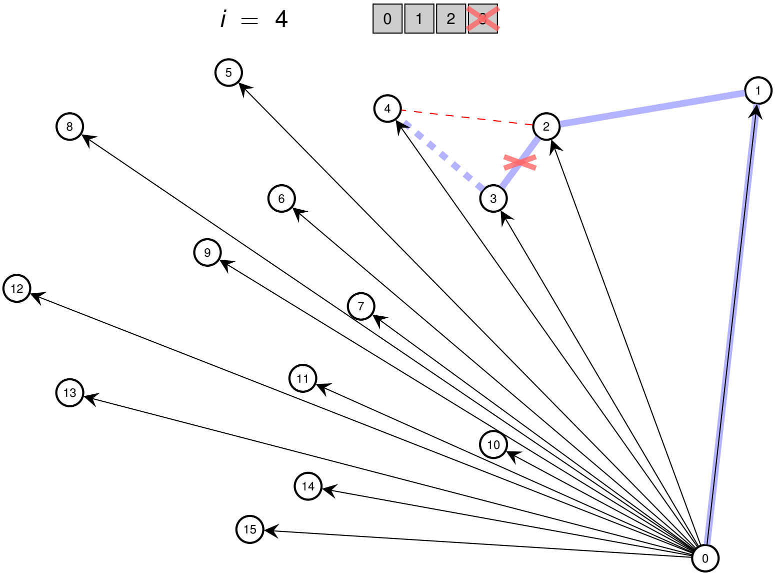
0	1	2	3



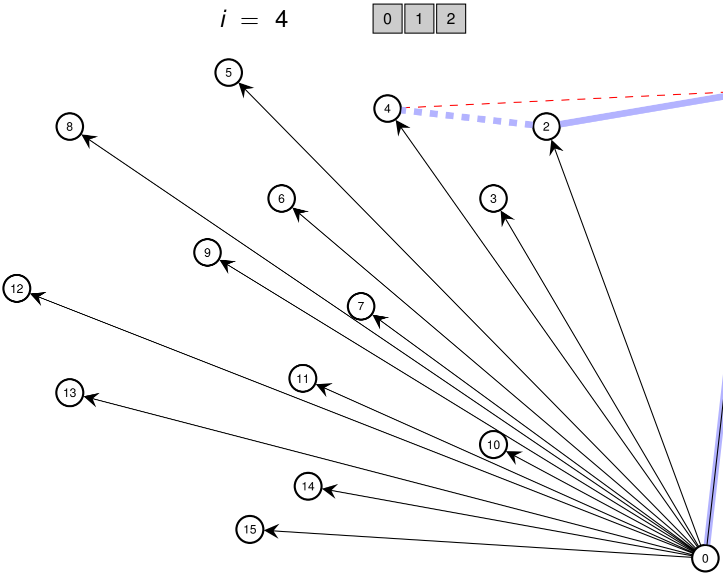
Execution of Graham's Scan



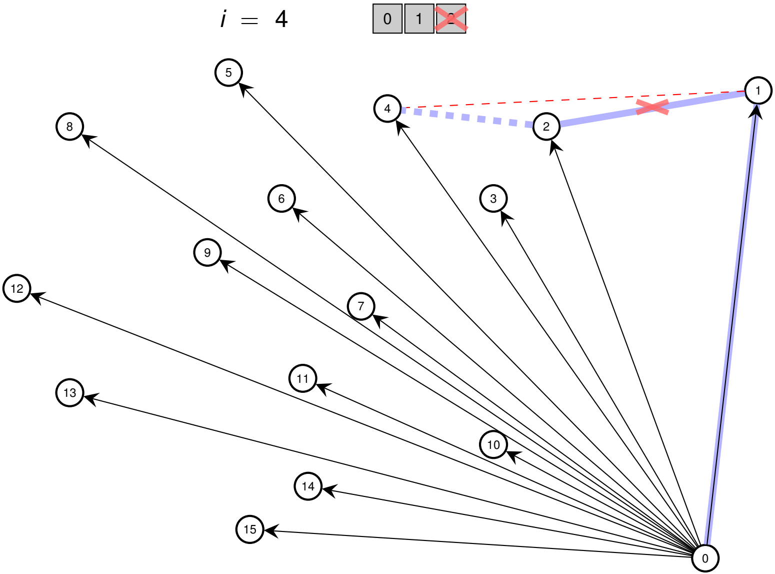
Execution of Graham's Scan



Execution of Graham's Scan



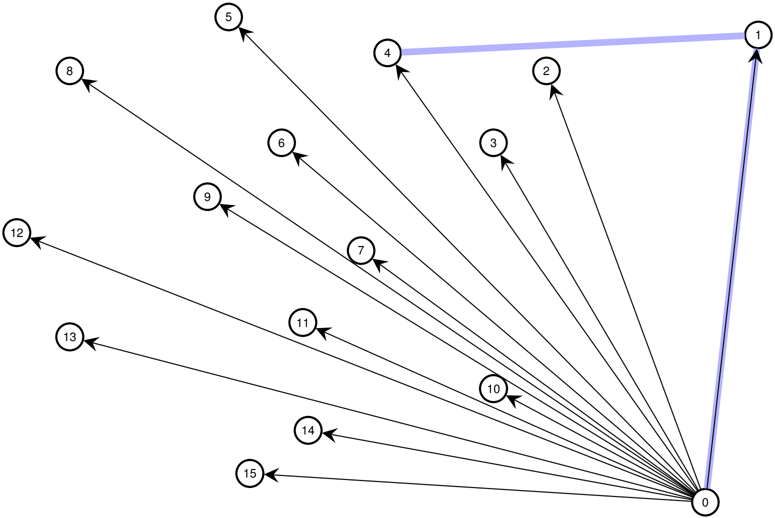
Execution of Graham's Scan



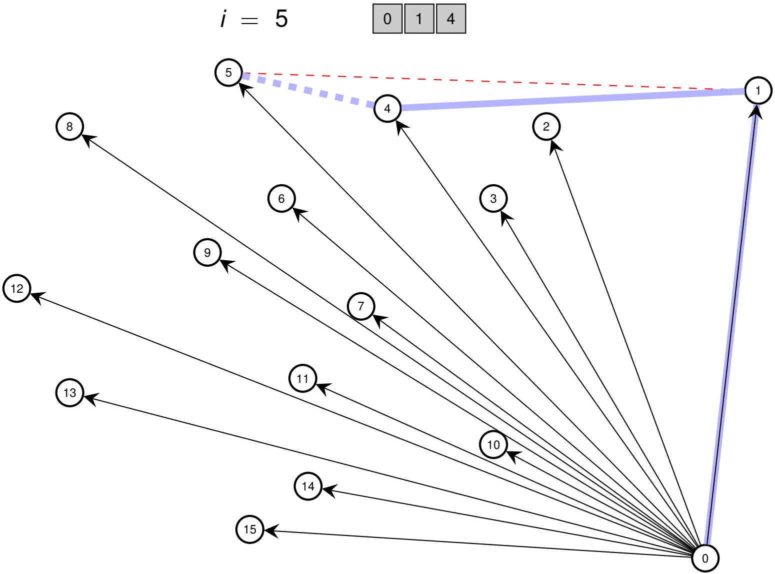
Execution of Graham's Scan

$i = 4$

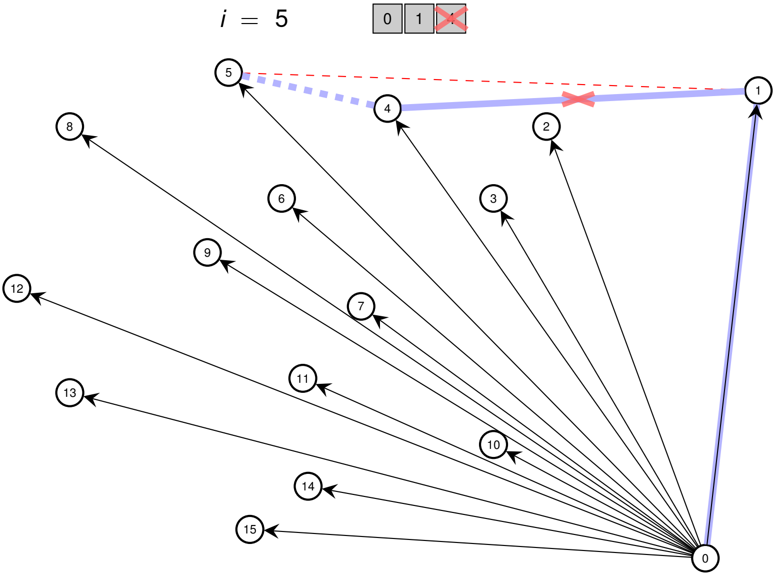
0	1	4



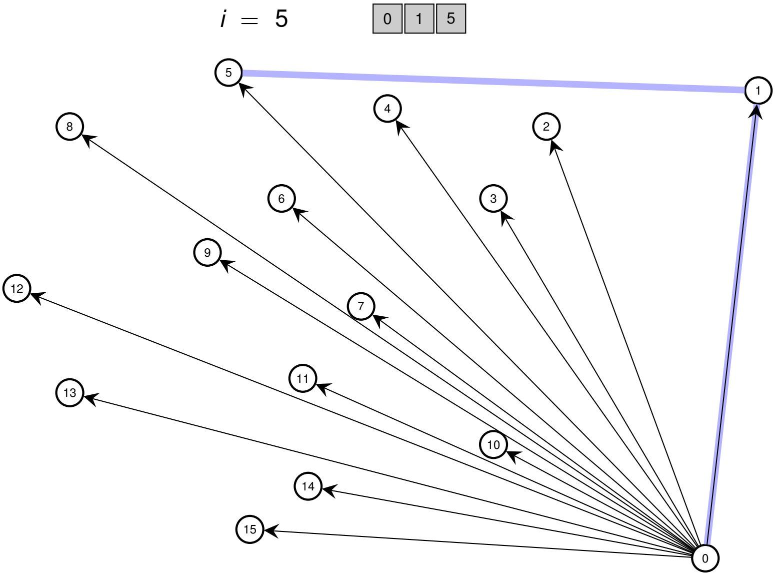
Execution of Graham's Scan



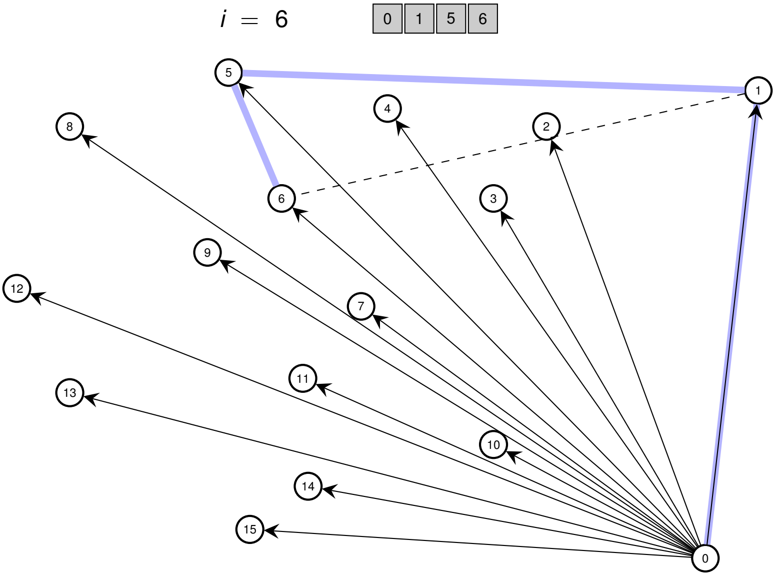
Execution of Graham's Scan



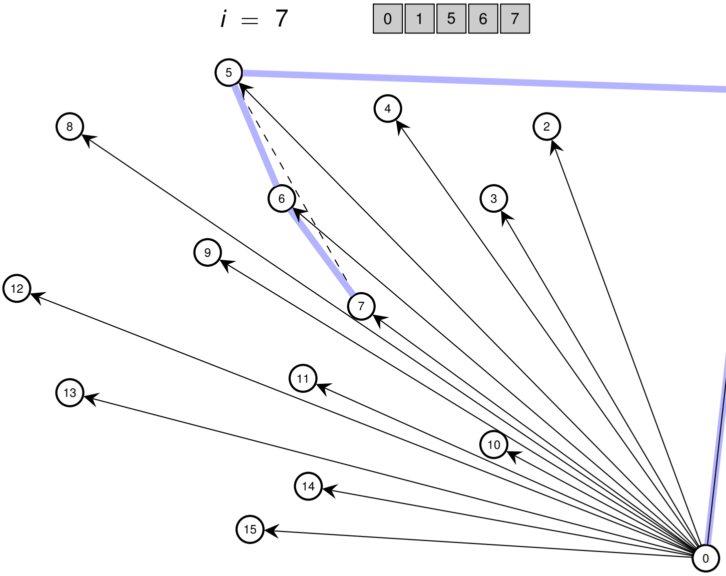
Execution of Graham’s Scan



Execution of Graham's Scan



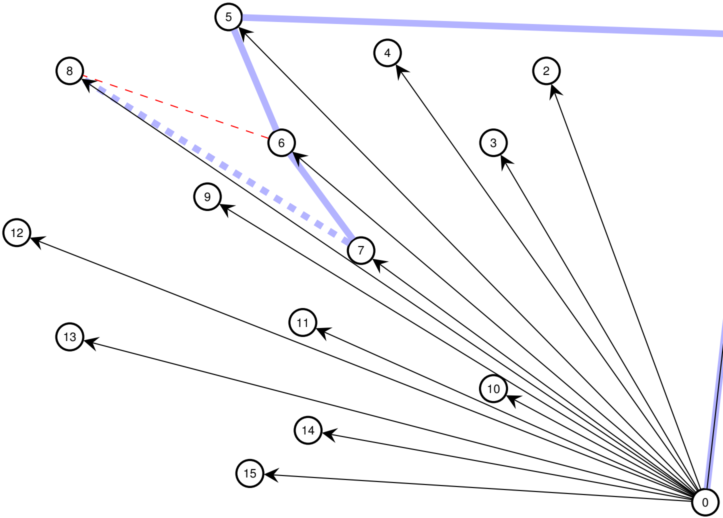
Execution of Graham's Scan



Execution of Graham's Scan

$i = 8$

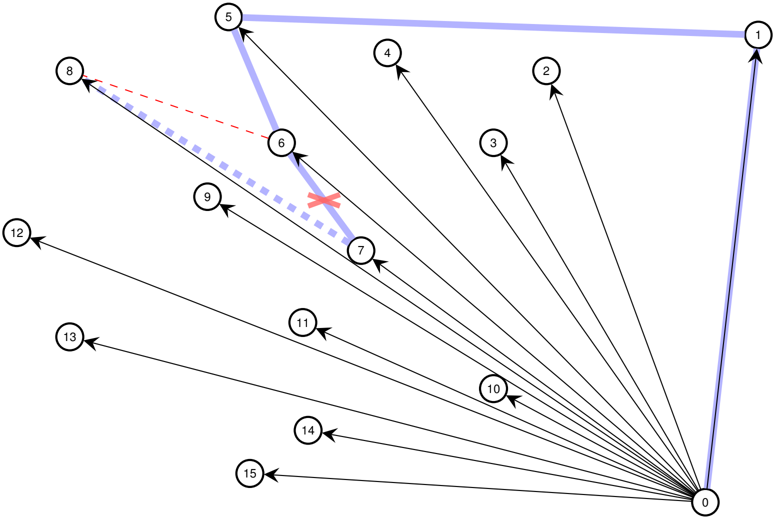
0	1	5	6	7



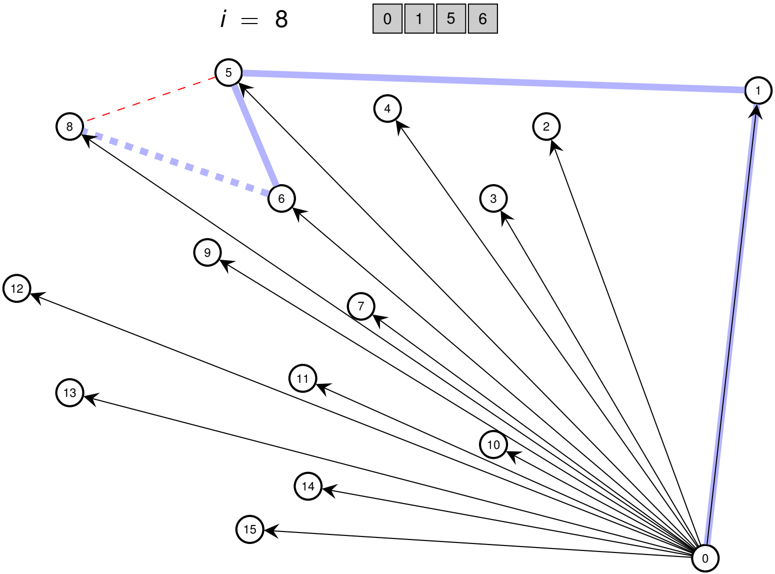
Execution of Graham's Scan

$i = 8$

0	1	5	6	~~7~~



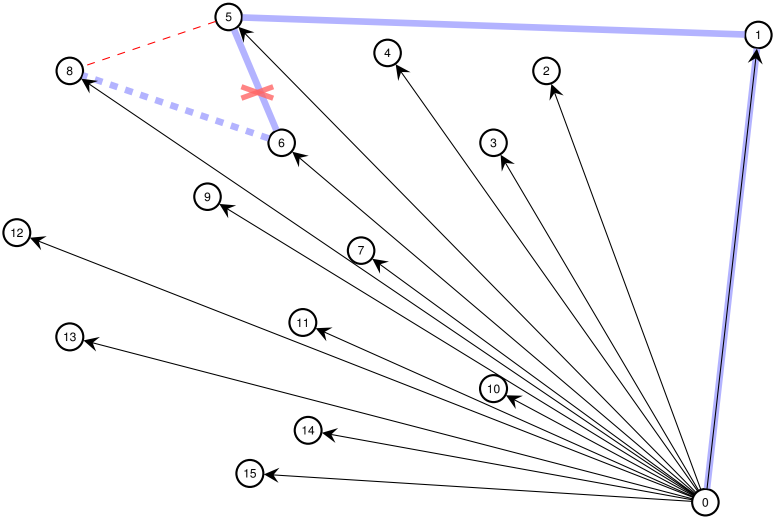
Execution of Graham's Scan



Execution of Graham's Scan

$i = 8$

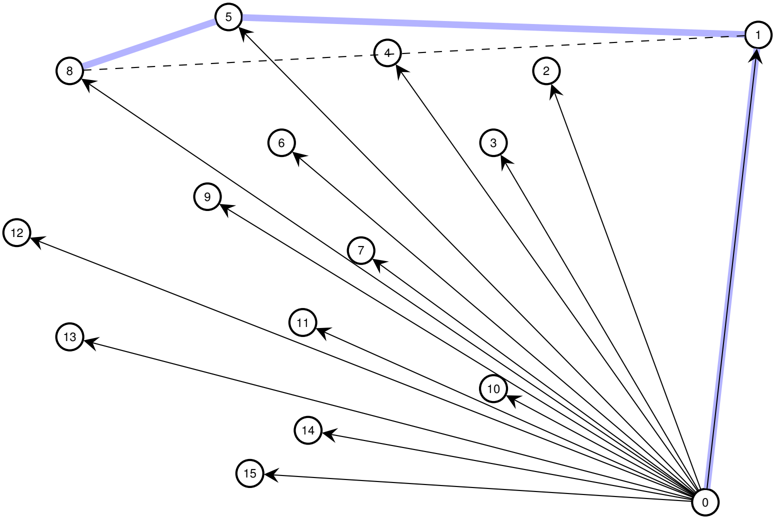
0	1	5	~~6~~



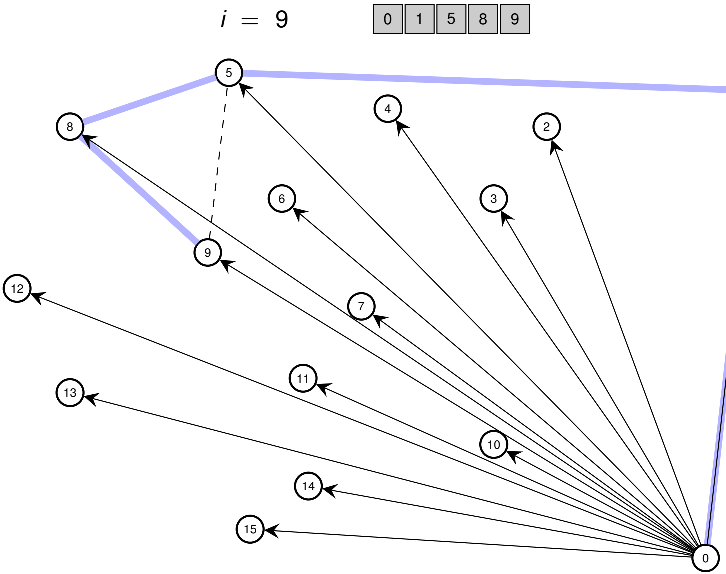
Execution of Graham's Scan

$i = 8$

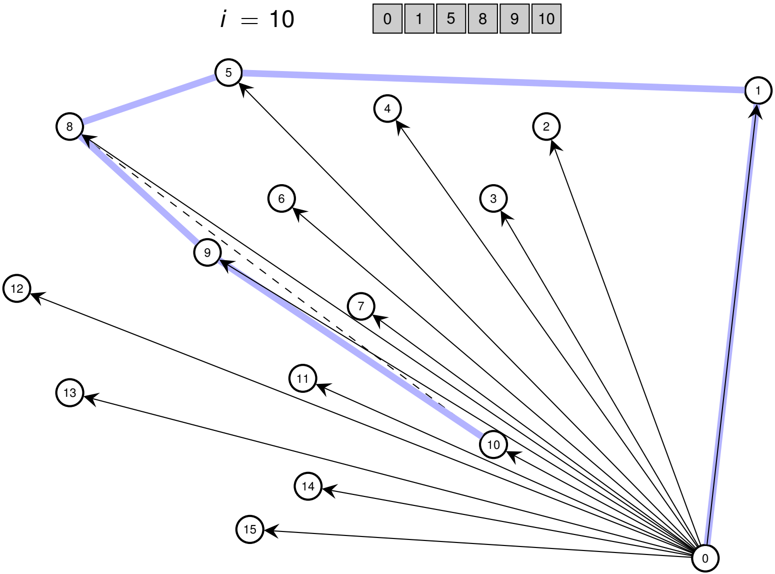
0	1	5	8



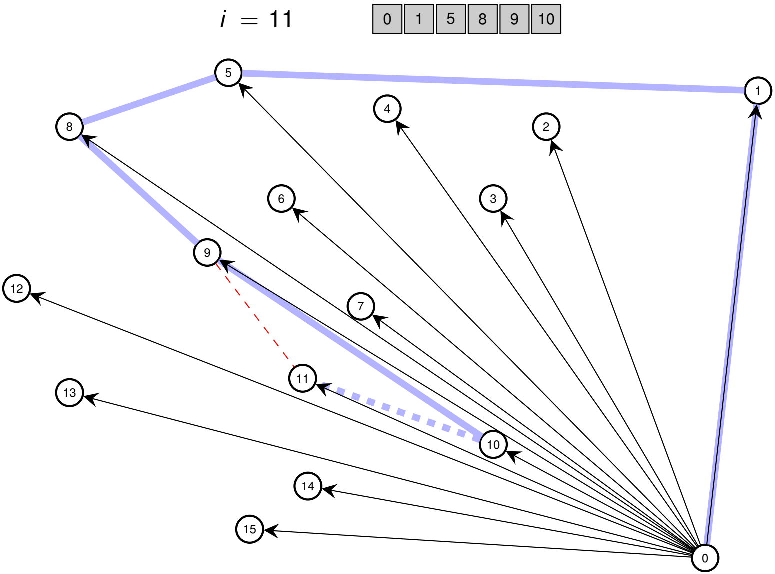
Execution of Graham's Scan



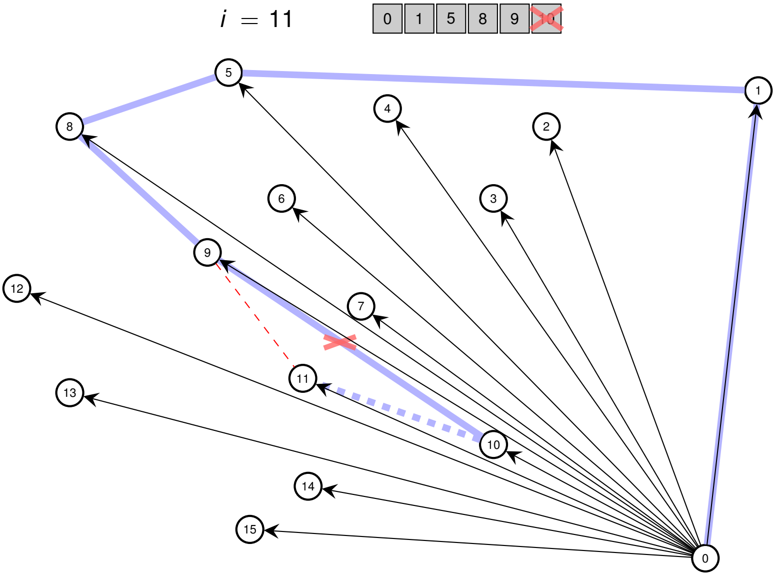
Execution of Graham's Scan



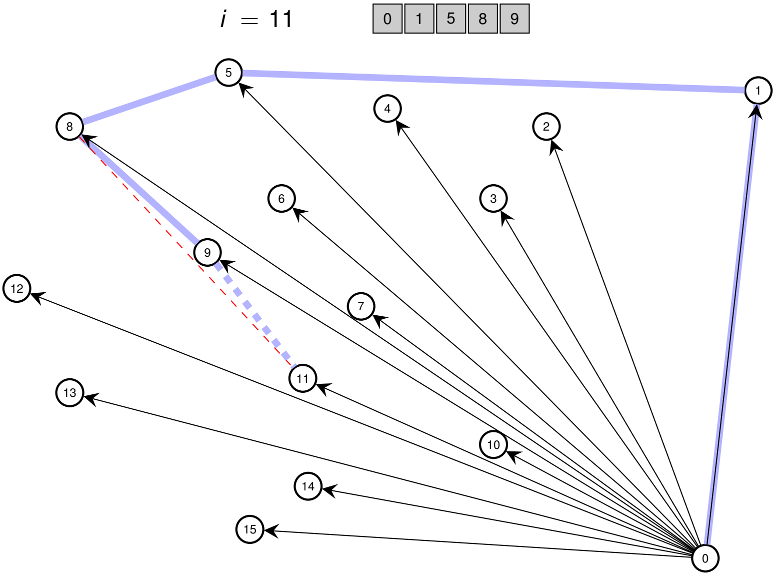
Execution of Graham's Scan



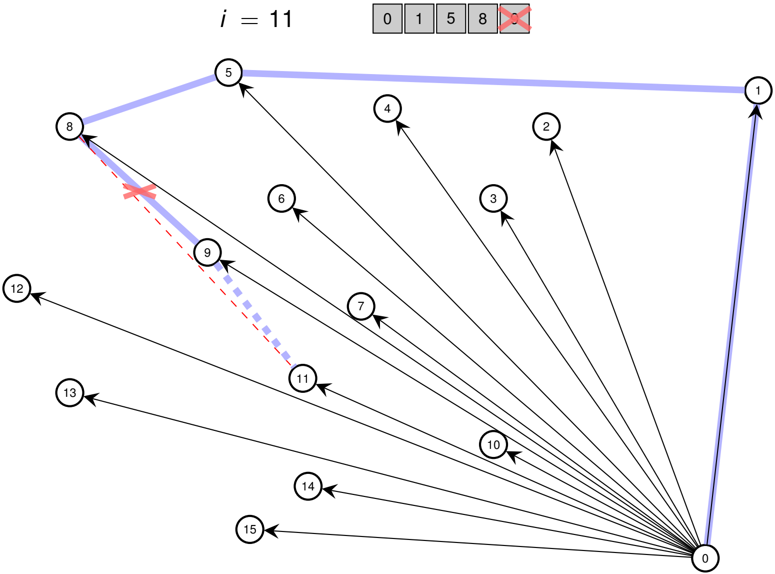
Execution of Graham's Scan



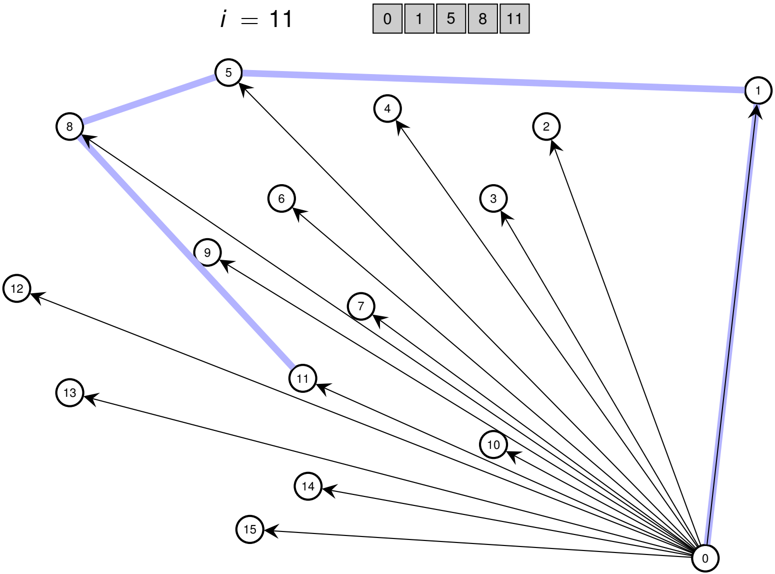
Execution of Graham's Scan



Execution of Graham’s Scan



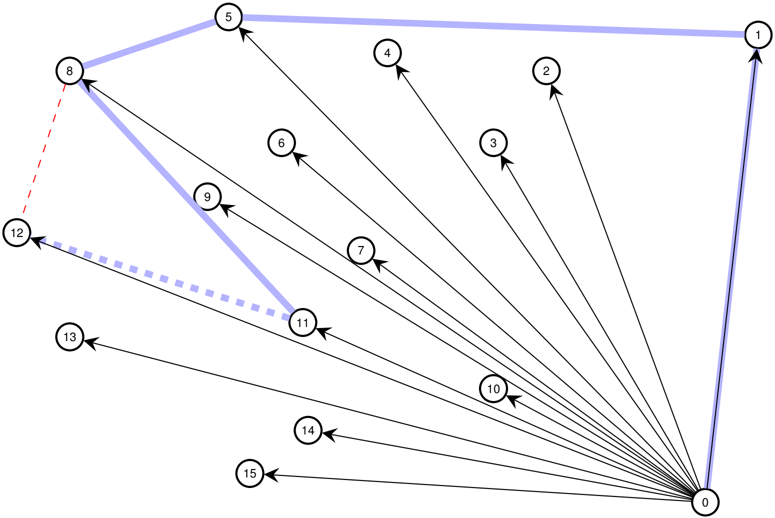
Execution of Graham's Scan



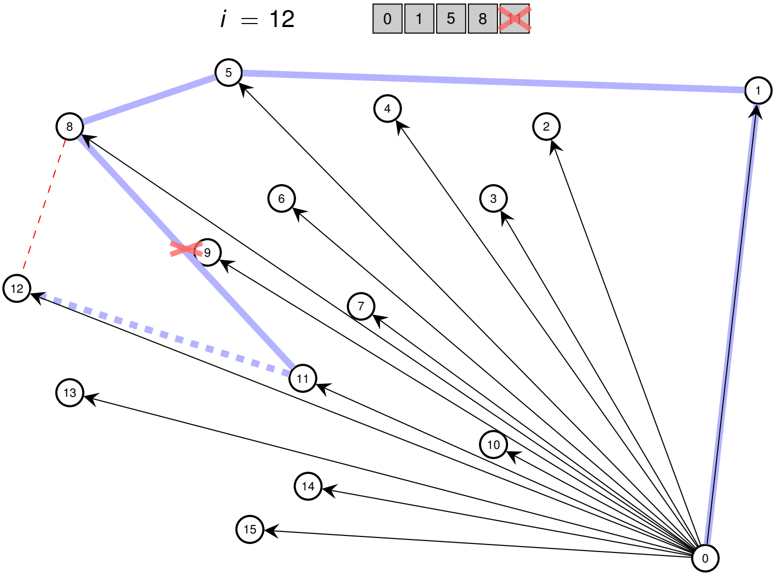
Execution of Graham's Scan

$i = 12$

0	1	5	8	11



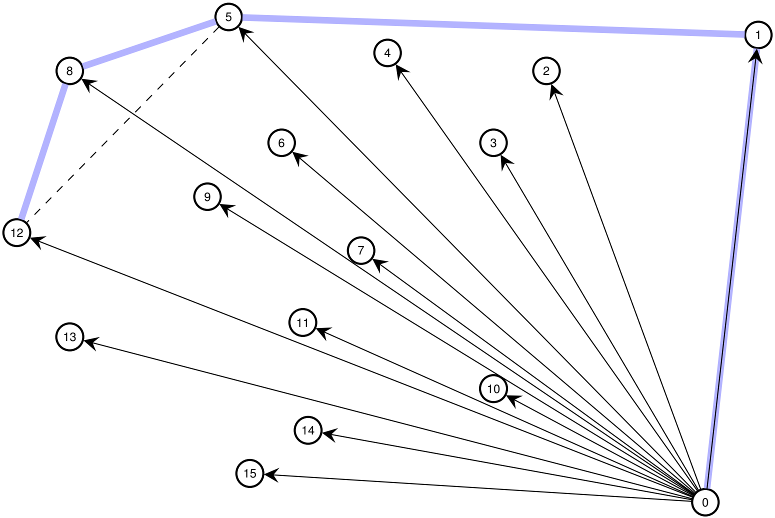
Execution of Graham’s Scan



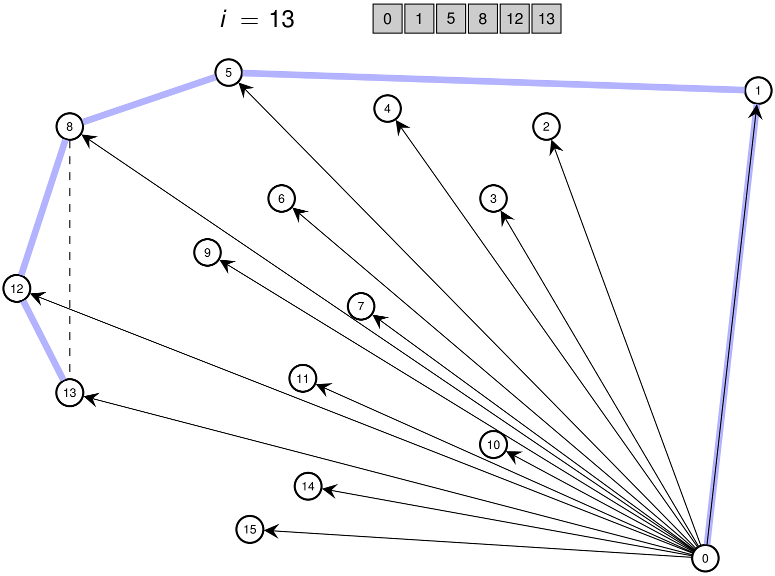
Execution of Graham's Scan

$i = 12$

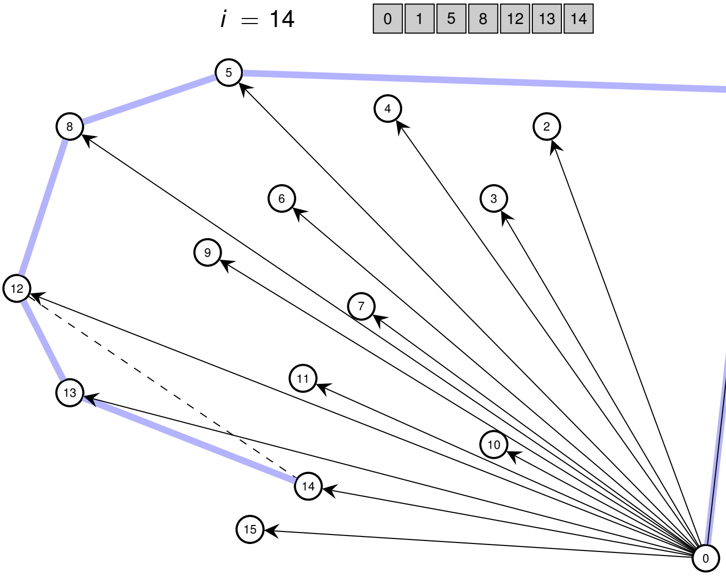
0	1	5	8	12



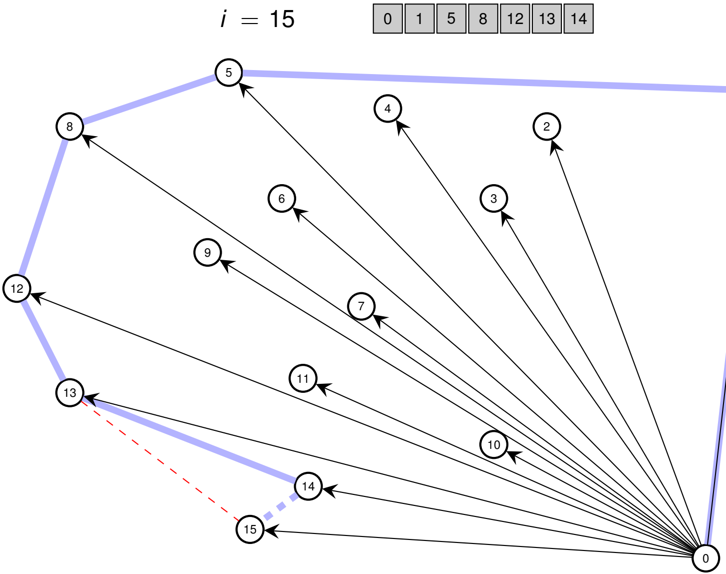
Execution of Graham's Scan



Execution of Graham's Scan



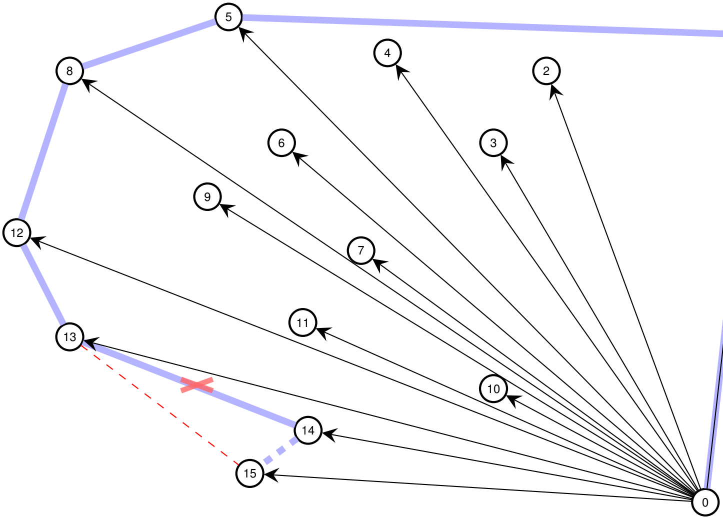
Execution of Graham's Scan



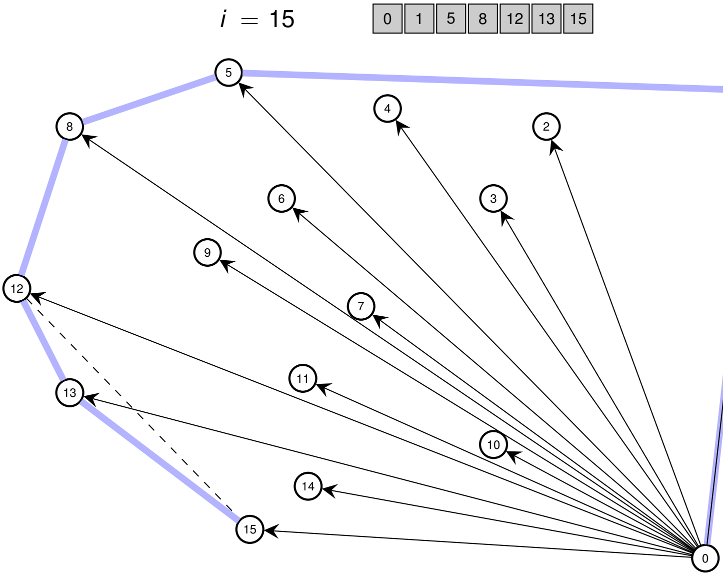
Execution of Graham's Scan

$i = 15$

0	1	5	8	12	13	~~14~~



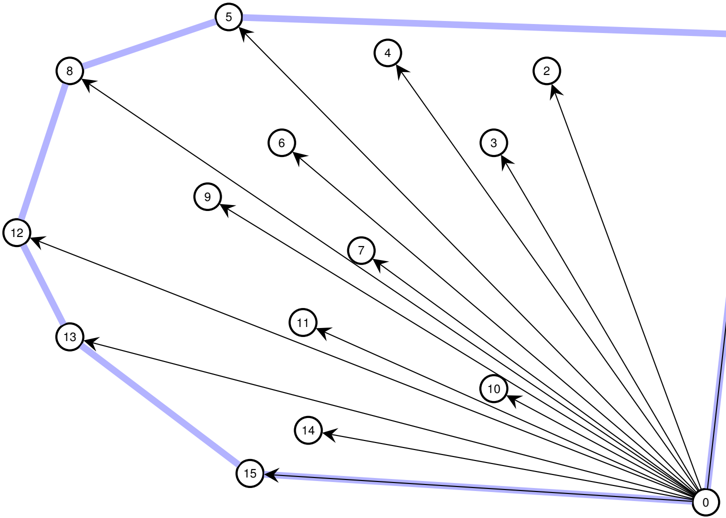
Execution of Graham's Scan



Execution of Graham's Scan

$i = 15$ ✓

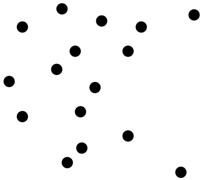
0	1	5	8	12	13	15



Jarvis' March (Gift wrapping)

Intuition

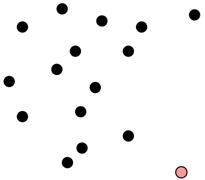
- Wrapping taut paper around the points



Jarvis' March (Gift wrapping)

Intuition

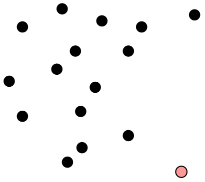
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point



Jarvis' March (Gift wrapping)

Intuition

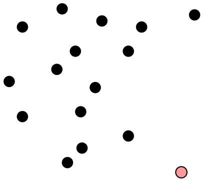
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point



Jarvis' March (Gift wrapping)

Intuition

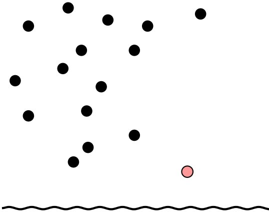
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point



Jarvis' March (Gift wrapping)

Intuition

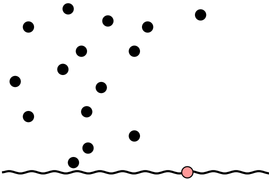
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point



Jarvis' March (Gift wrapping)

Intuition

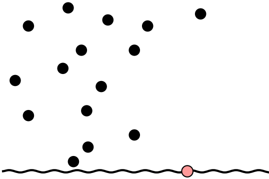
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point



Jarvis' March (Gift wrapping)

Intuition

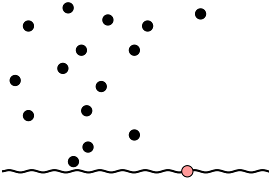
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

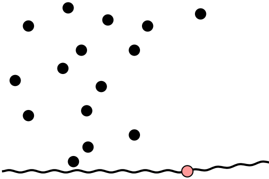
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

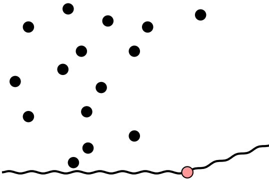
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

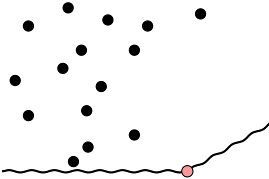
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

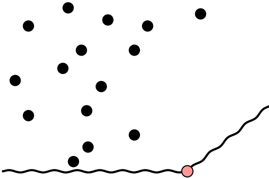
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

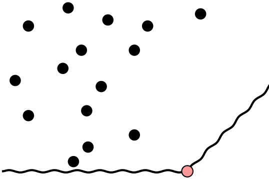
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis’ March (Gift wrapping)

Intuition

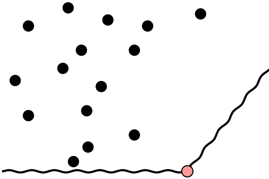
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

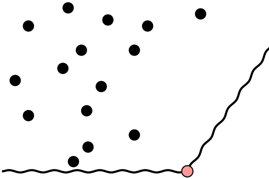
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

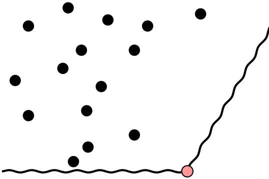
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

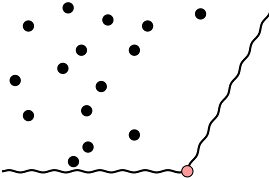
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

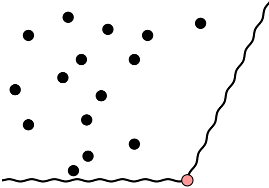
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

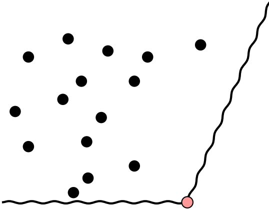
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

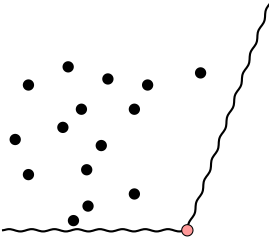
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis’ March (Gift wrapping)

Intuition

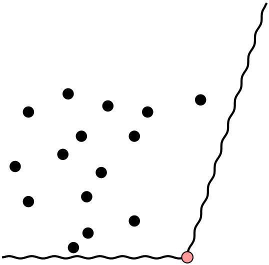
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

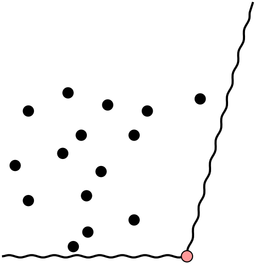
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

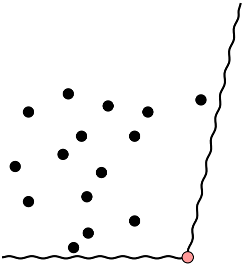
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

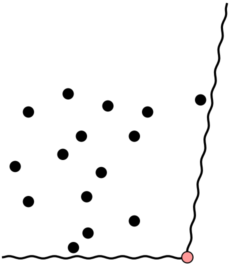
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

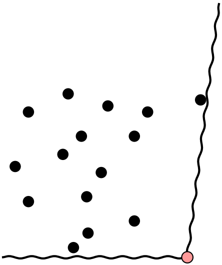
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

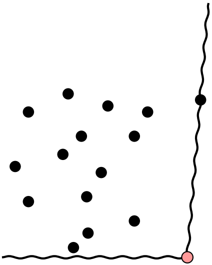
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

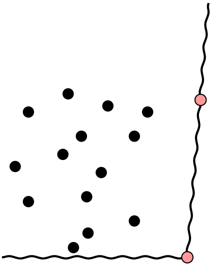
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point



Jarvis' March (Gift wrapping)

Intuition

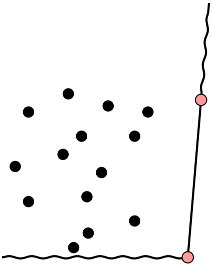
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2



Jarvis' March (Gift wrapping)

Intuition

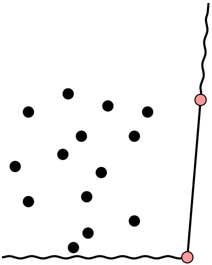
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2



Jarvis' March (Gift wrapping)

Intuition

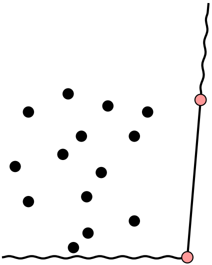
- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2



Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2



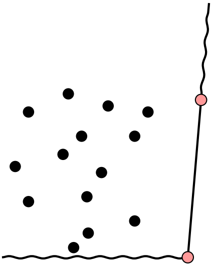
Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2

Algorithm

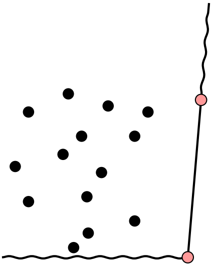
1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0



Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2



Algorithm

1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k

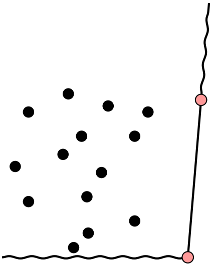
Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2

Algorithm

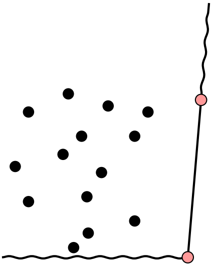
1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k
4. Next point the one with smallest angle w.r.t. p_k



Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2



Algorithm

1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k
4. Next point the one with smallest angle w.r.t. p_k

Here, we rotate the coordinate system by 180!

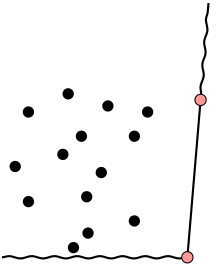
Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2

Algorithm

1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k
4. Next point the one with smallest angle w.r.t. p_k
5. Continue until p_0 is reached



Jarvis' March (Gift wrapping)

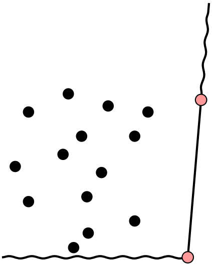
Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2

Algorithm

1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k
4. Next point the one with smallest angle w.r.t. p_k
5. Continue until p_0 is reached

Runtime: $O(n \cdot h)$, where h is no. points on convex hull.



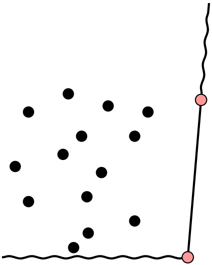
Jarvis' March (Gift wrapping)

Intuition

- Wrapping taut paper around the points
  1. Tape end of paper at lowest point
  2. Pull paper to the right until it touches a point
  3. Tape paper and go to 2

Algorithm

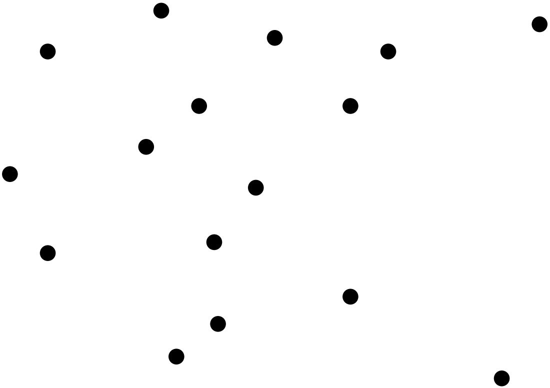
1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k
4. Next point the one with smallest angle w.r.t. p_k
5. Continue until p_0 is reached



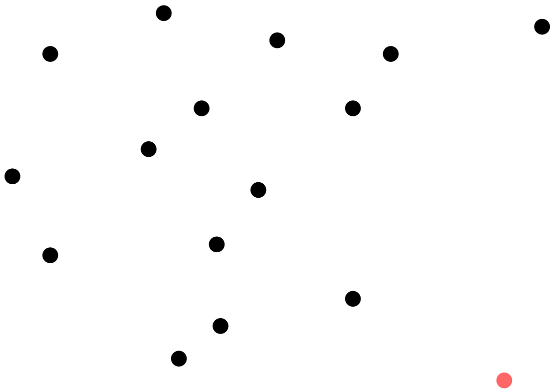
Runtime: $O(n \cdot h)$, where h is no. points on convex hull.

Output sensitive algorithm!

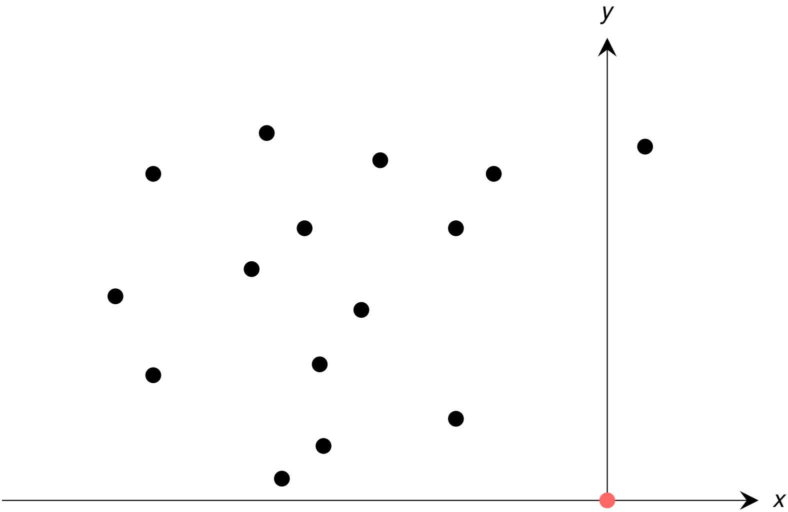
Execution of Jarvis' March



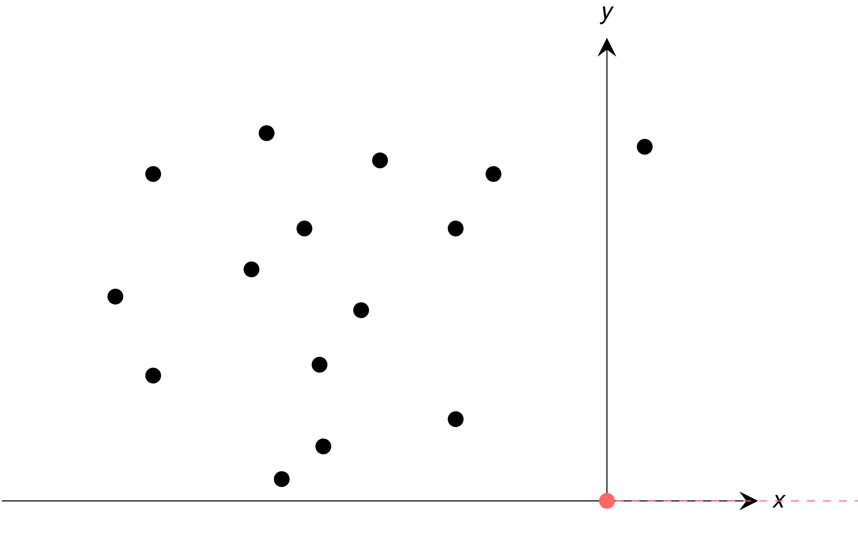
Execution of Jarvis' March



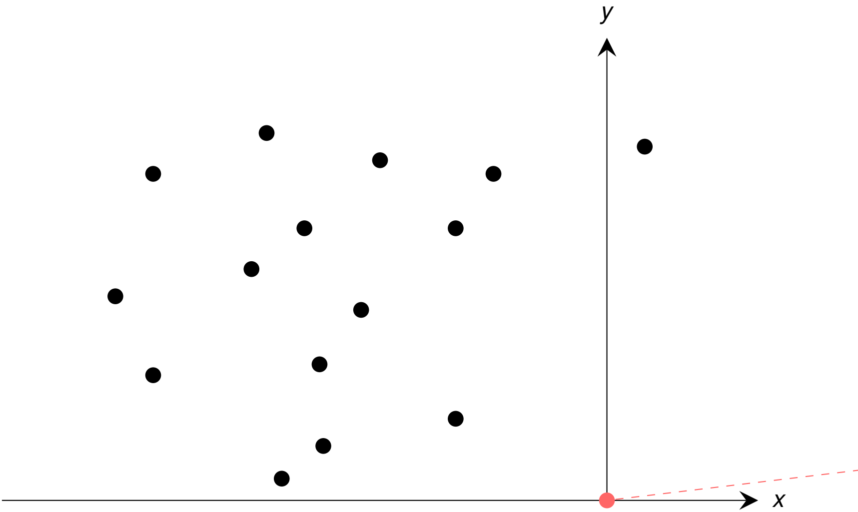
Execution of Jarvis' March



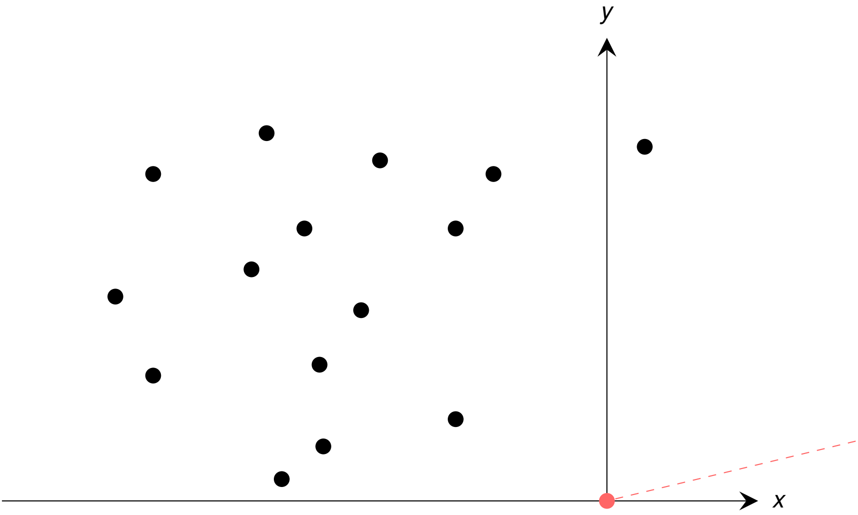
Execution of Jarvis' March



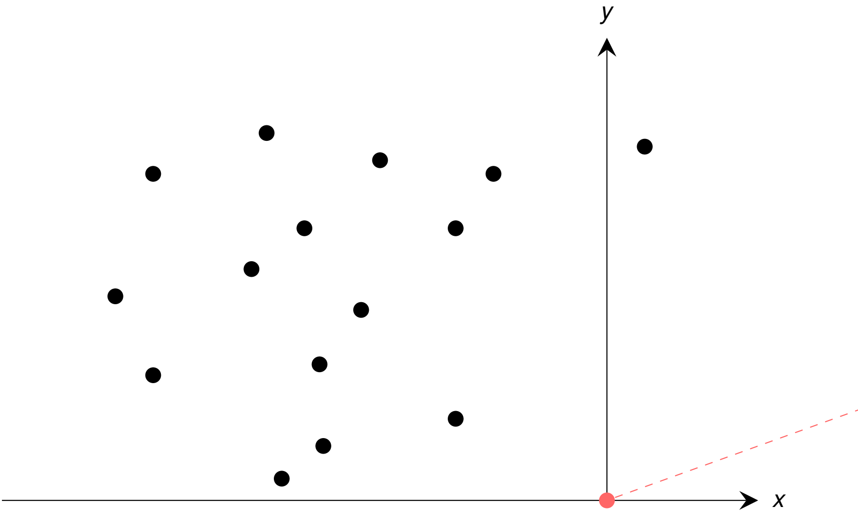
Execution of Jarvis' March



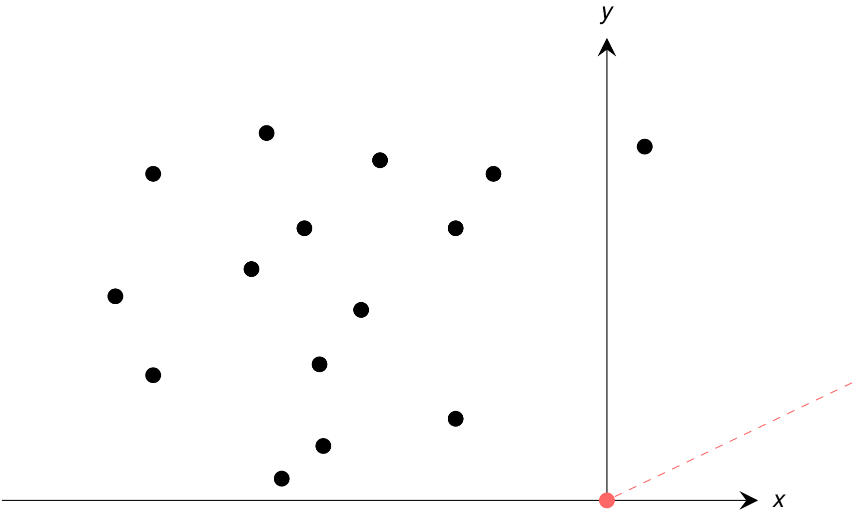
Execution of Jarvis' March



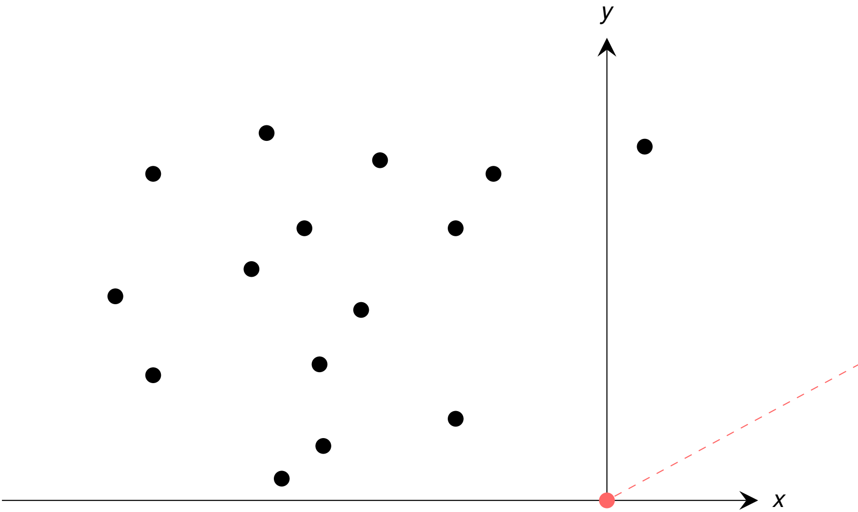
Execution of Jarvis' March



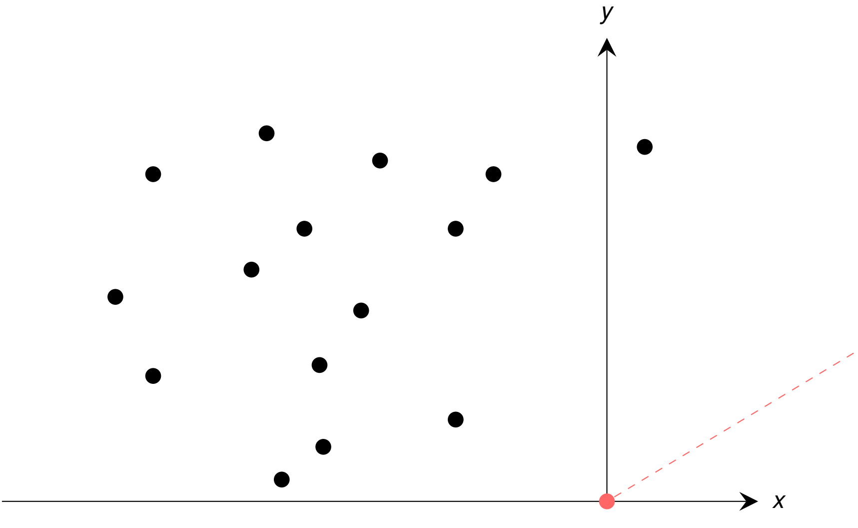
Execution of Jarvis' March



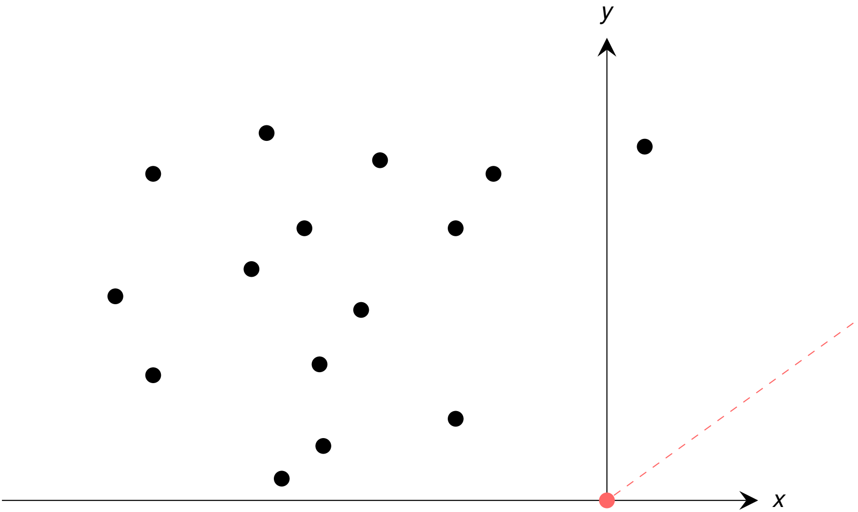
Execution of Jarvis' March



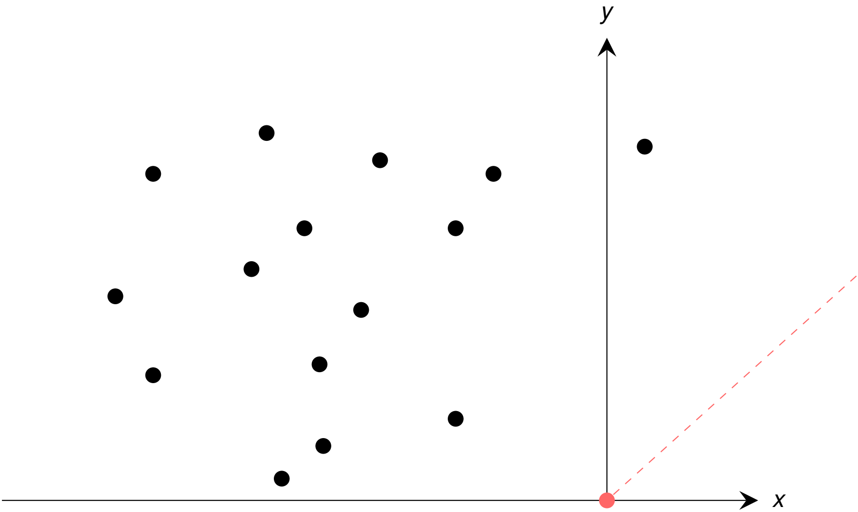
Execution of Jarvis' March



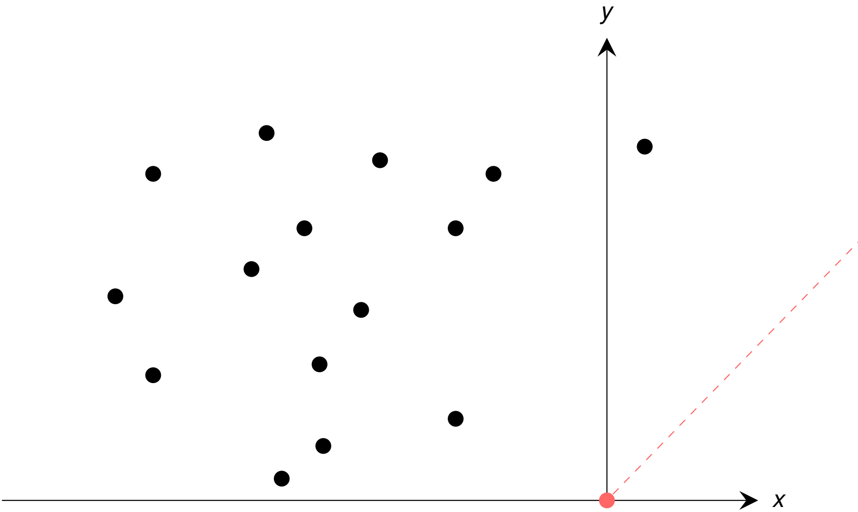
Execution of Jarvis' March



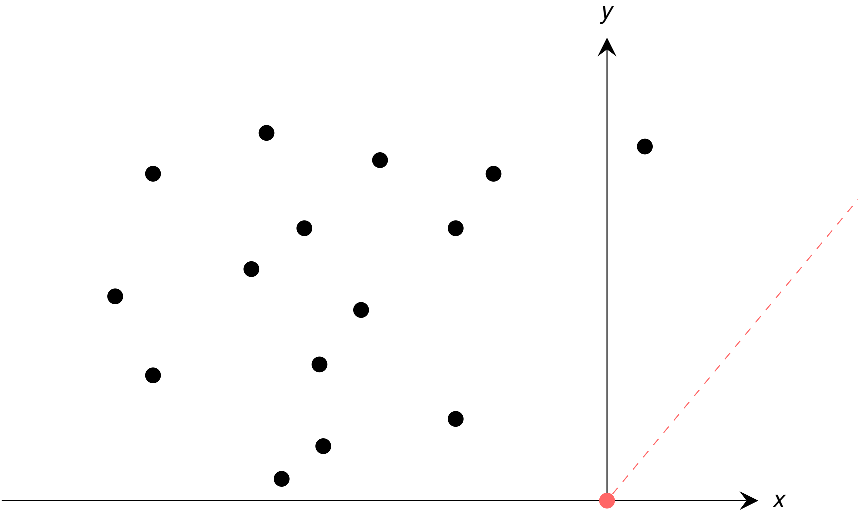
Execution of Jarvis' March



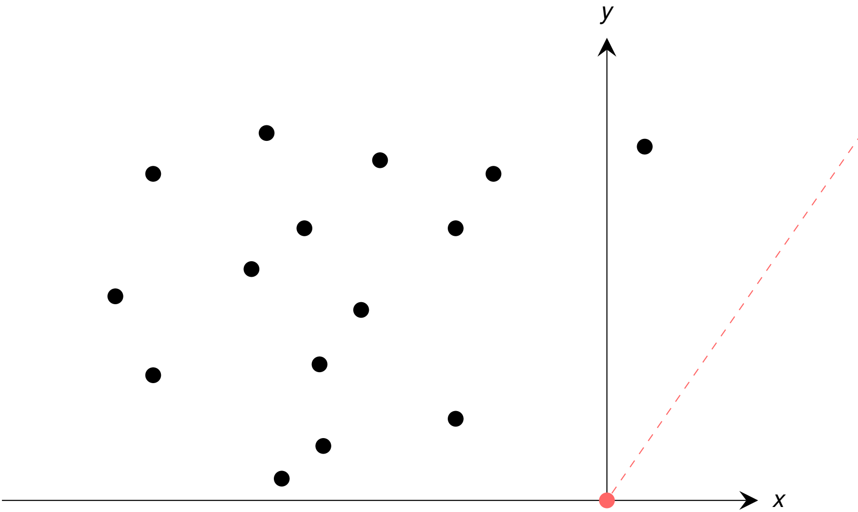
Execution of Jarvis' March



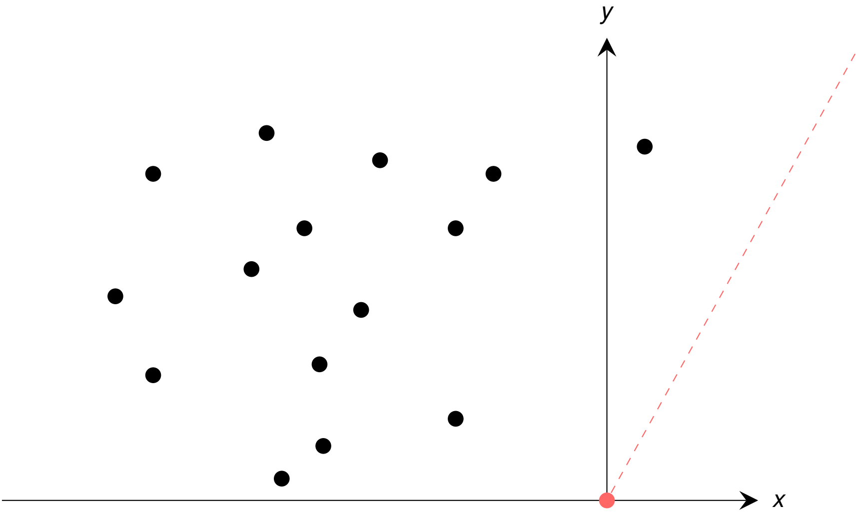
Execution of Jarvis' March



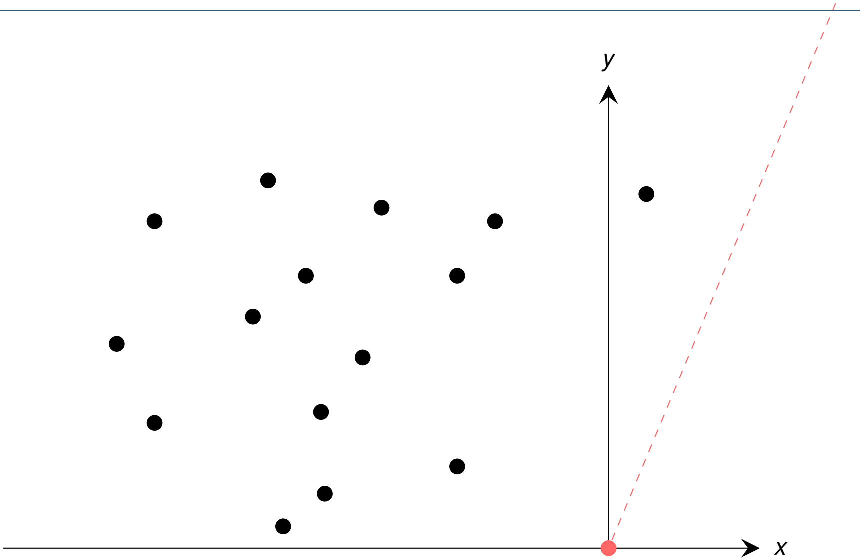
Execution of Jarvis' March



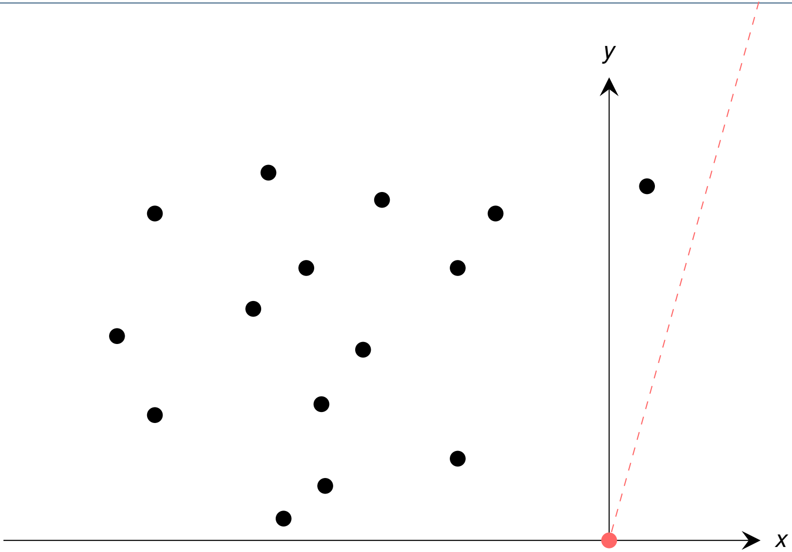
Execution of Jarvis' March



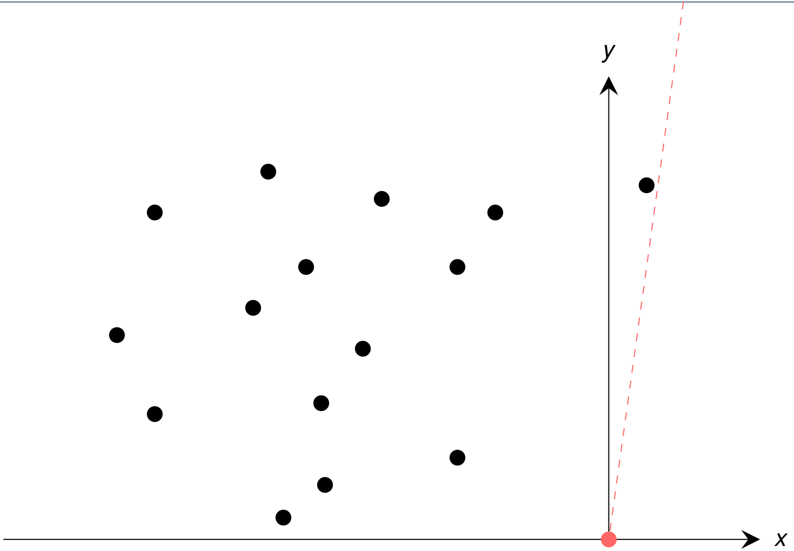
Execution of Jarvis' March



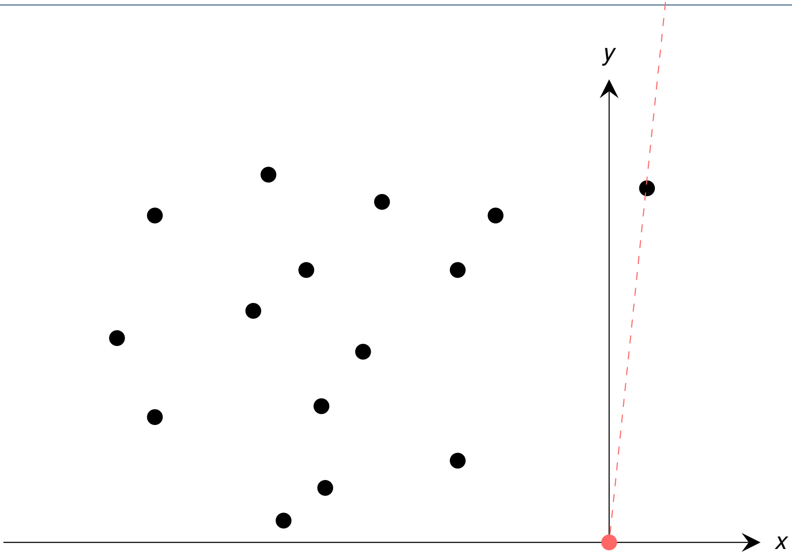
Execution of Jarvis' March



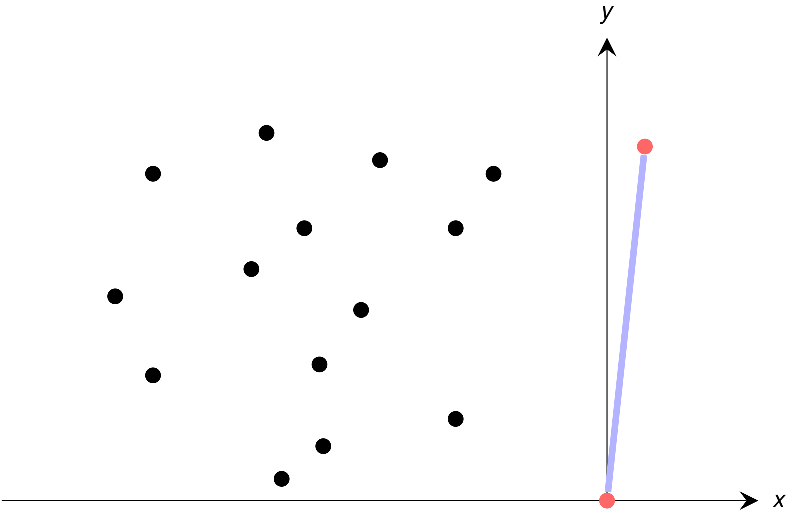
Execution of Jarvis' March



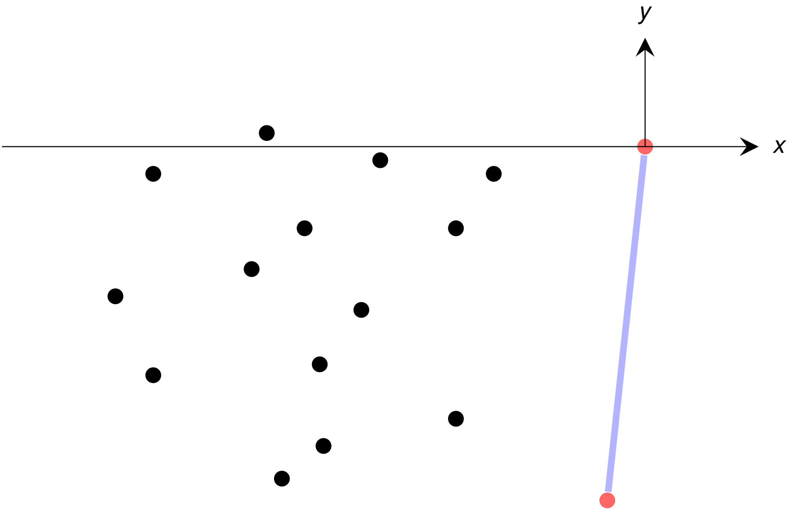
Execution of Jarvis' March



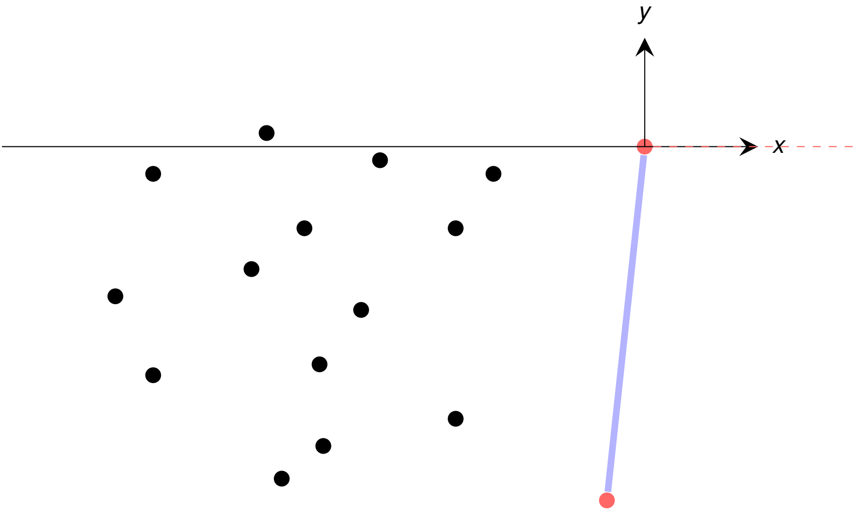
Execution of Jarvis' March



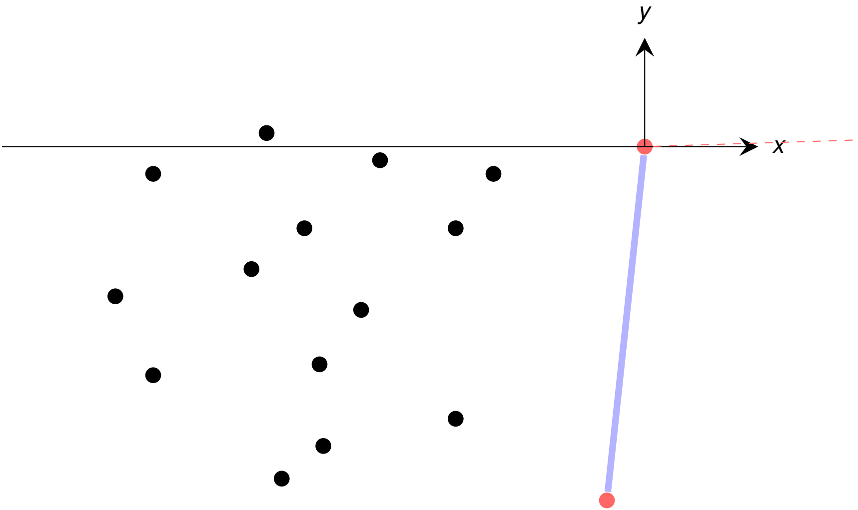
Execution of Jarvis' March



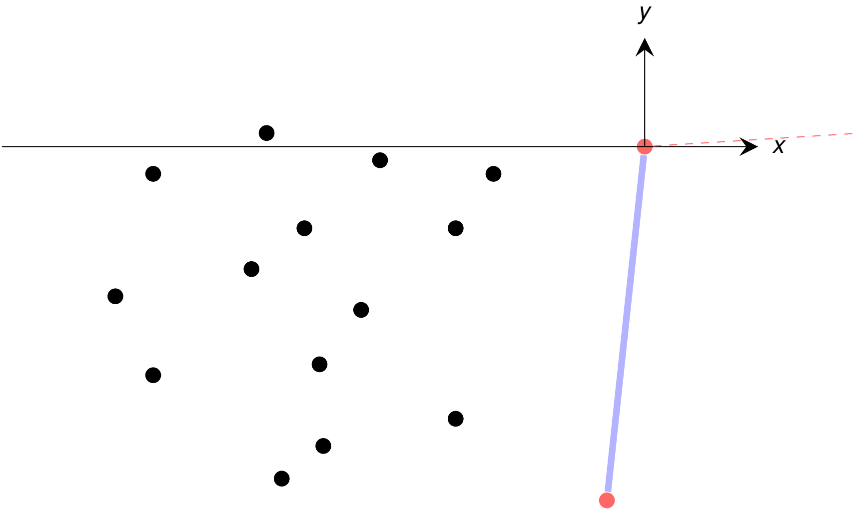
Execution of Jarvis' March



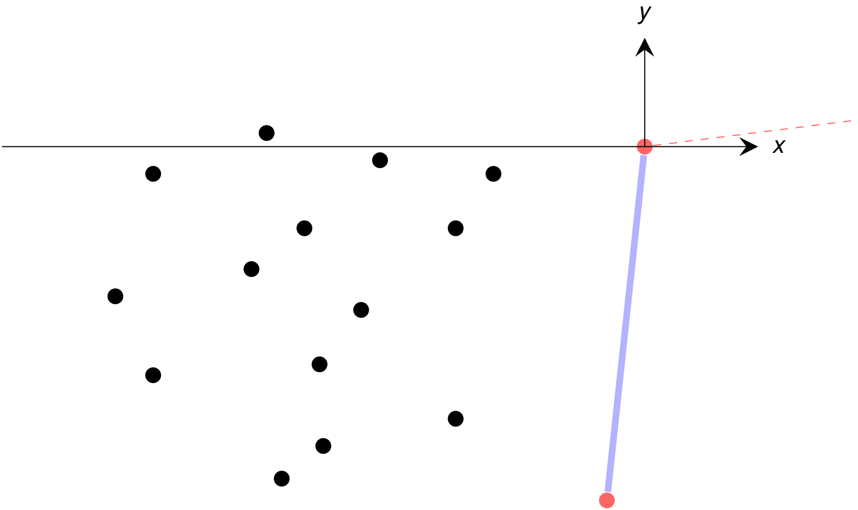
Execution of Jarvis' March



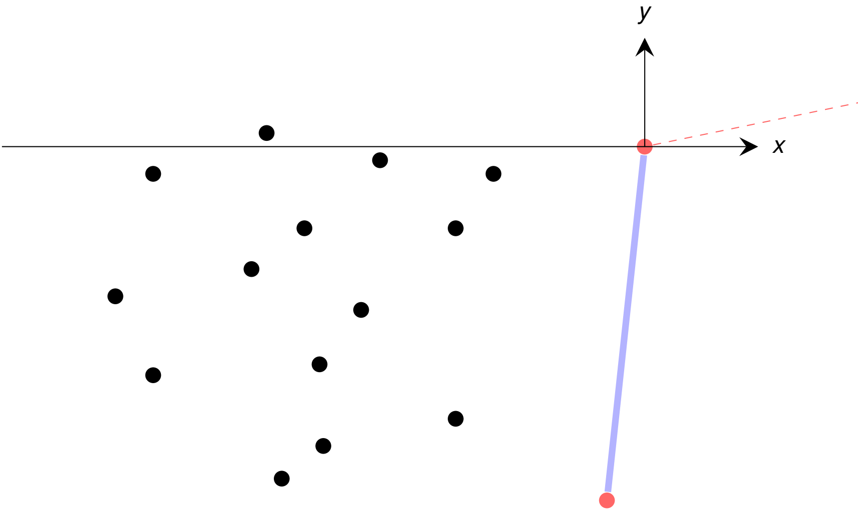
Execution of Jarvis' March



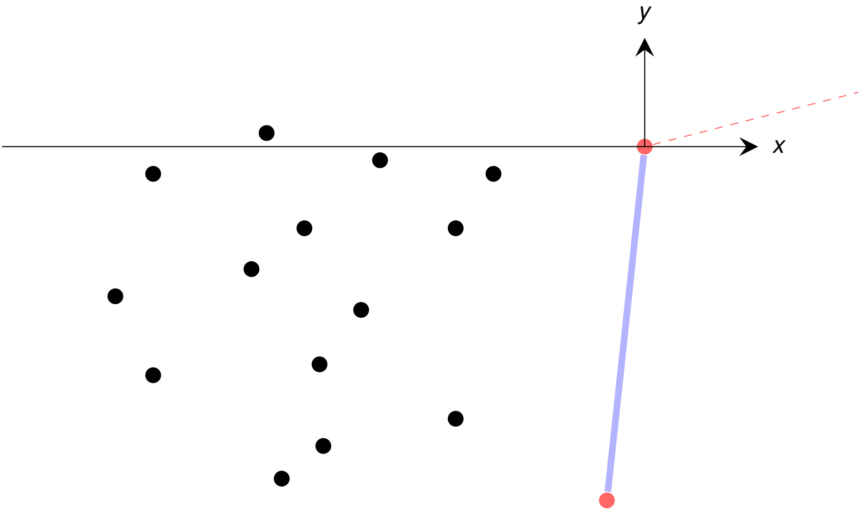
Execution of Jarvis' March



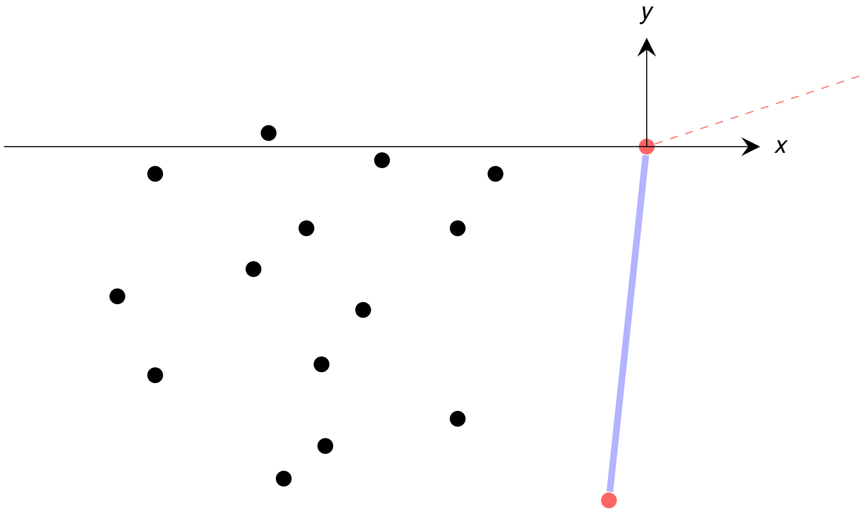
Execution of Jarvis' March



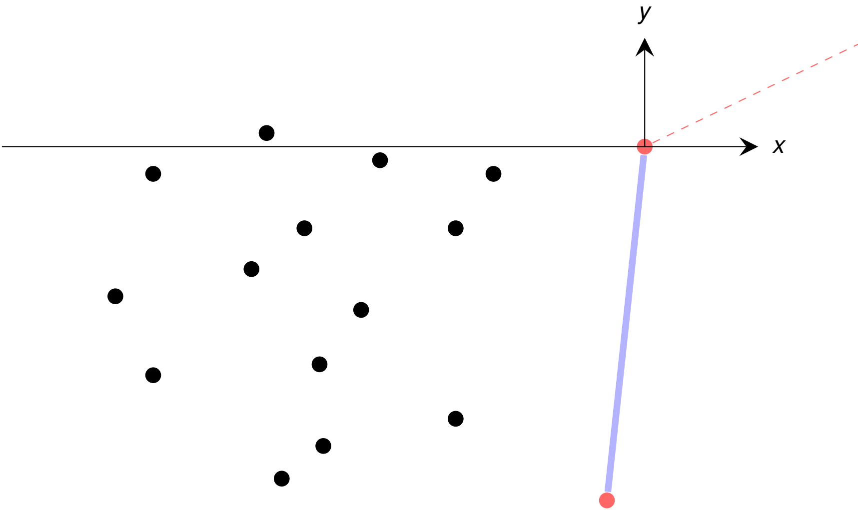
Execution of Jarvis' March



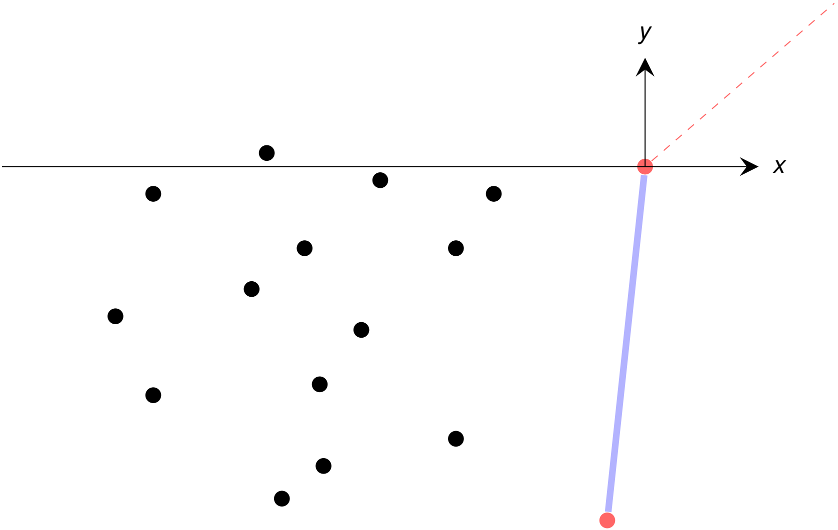
Execution of Jarvis' March



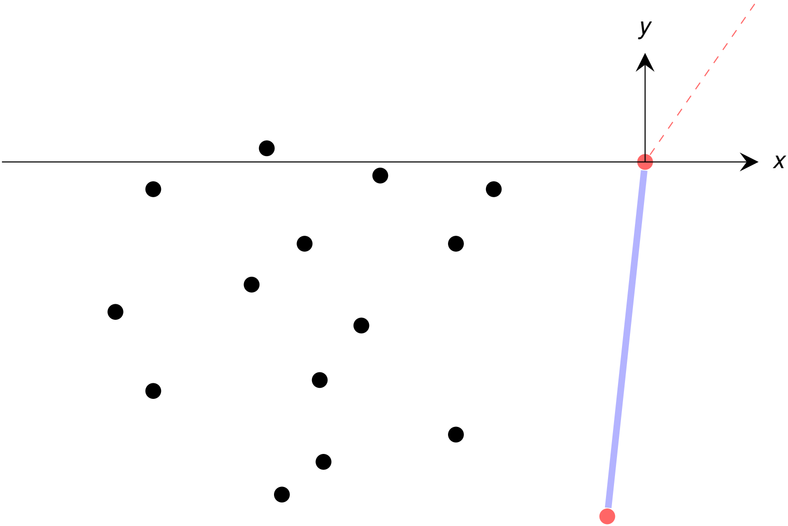
Execution of Jarvis' March



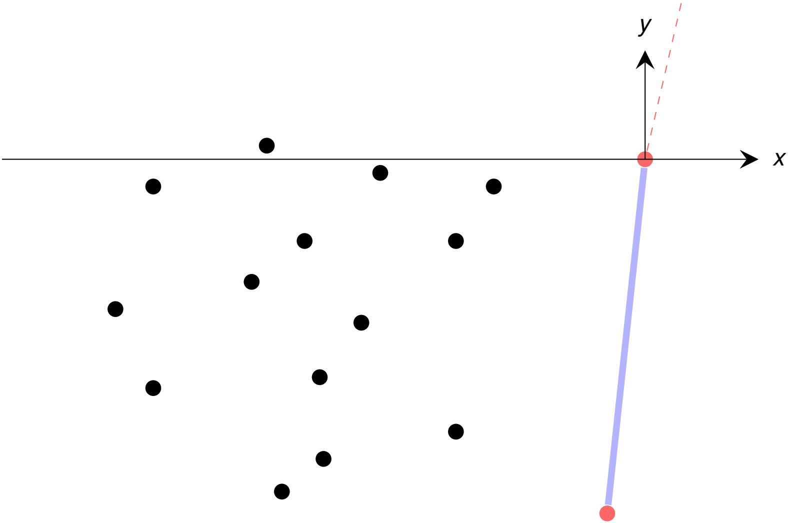
Execution of Jarvis' March



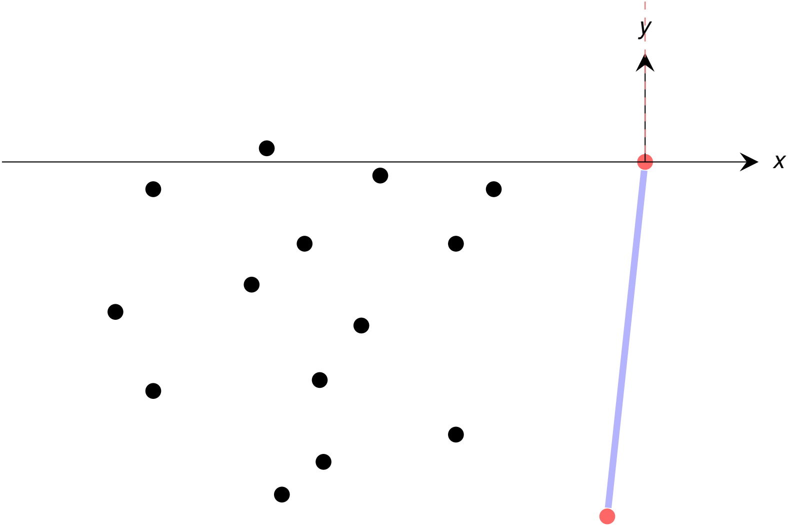
Execution of Jarvis' March



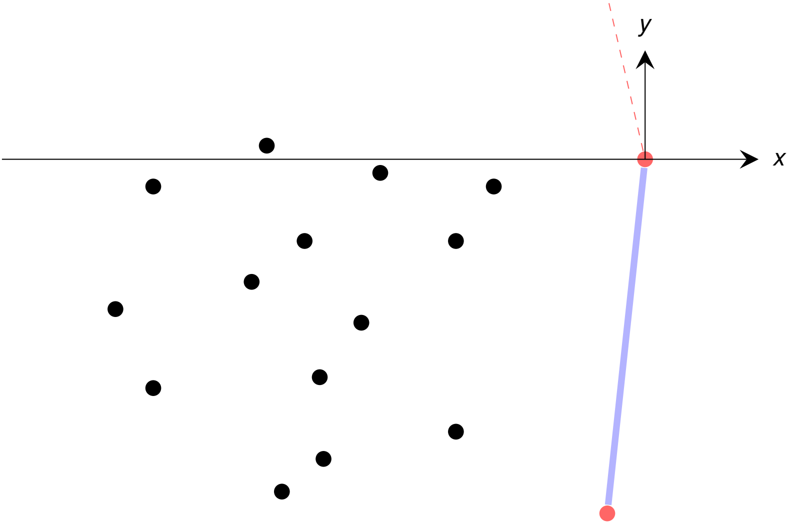
Execution of Jarvis' March



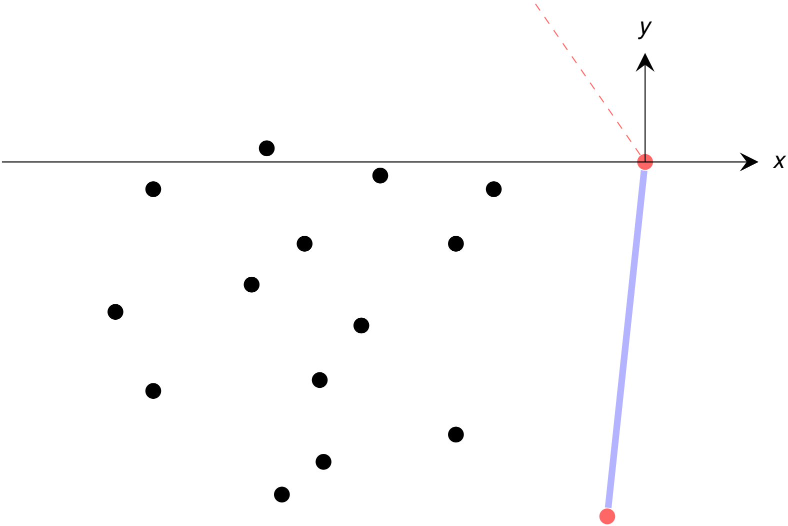
Execution of Jarvis' March



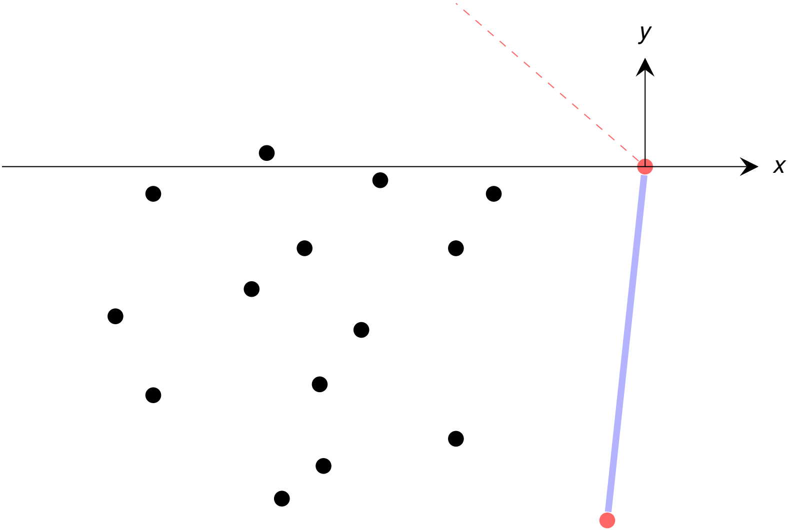
Execution of Jarvis' March



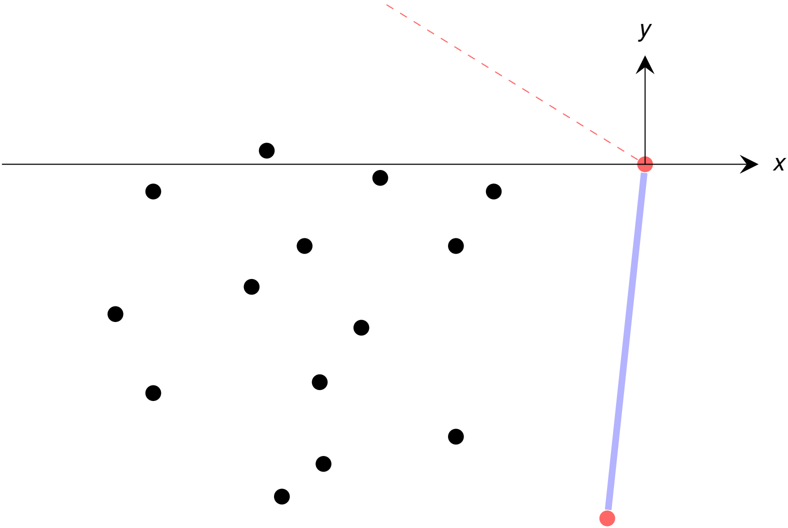
Execution of Jarvis' March



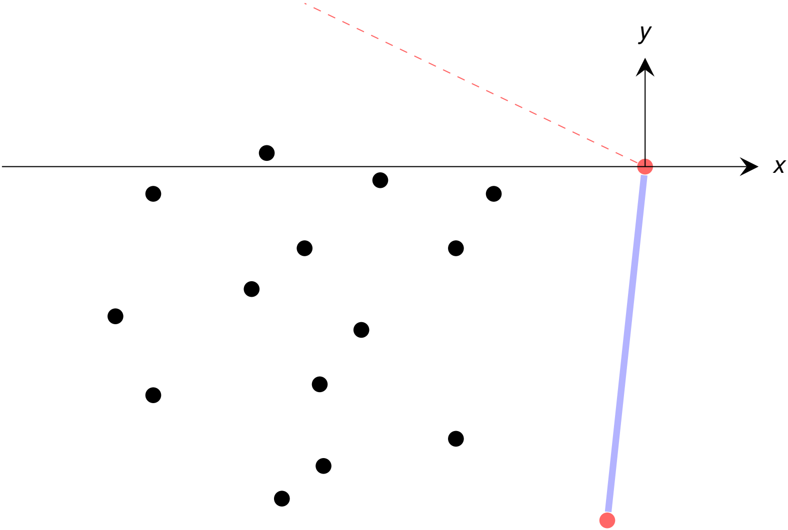
Execution of Jarvis' March



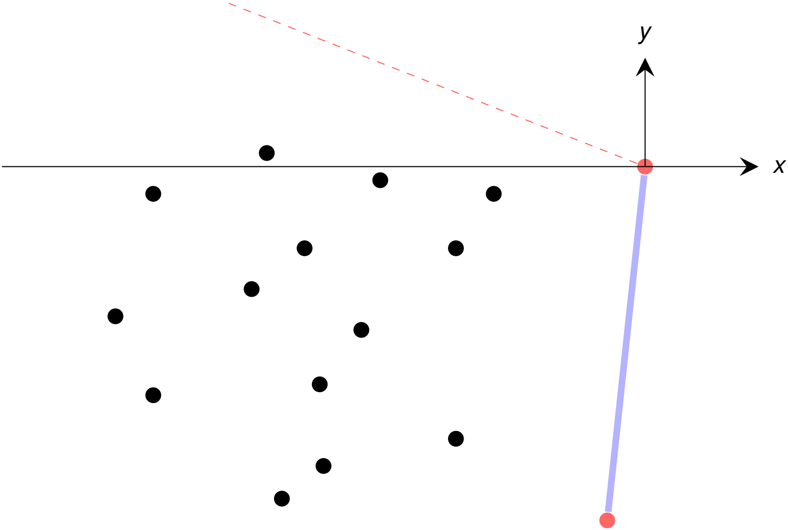
Execution of Jarvis' March



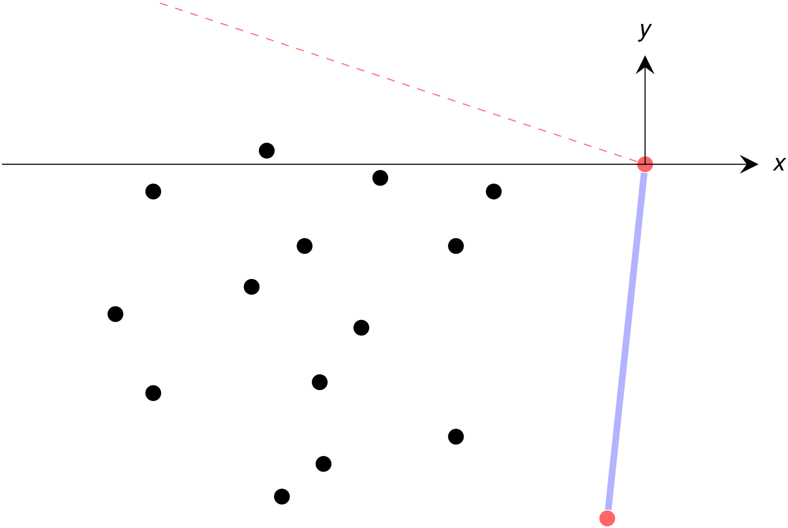
Execution of Jarvis' March



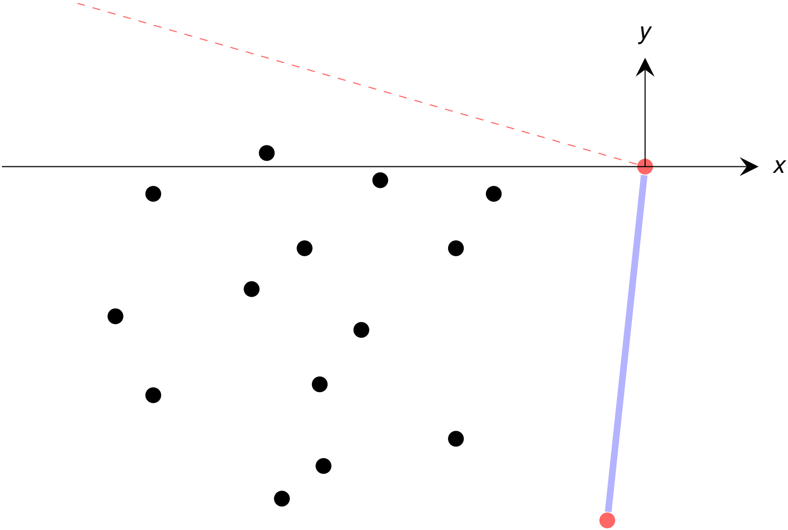
Execution of Jarvis' March



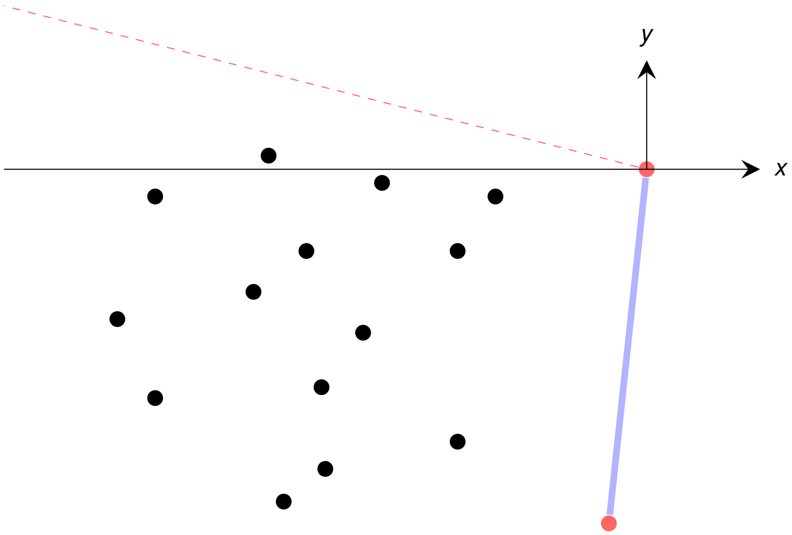
Execution of Jarvis' March



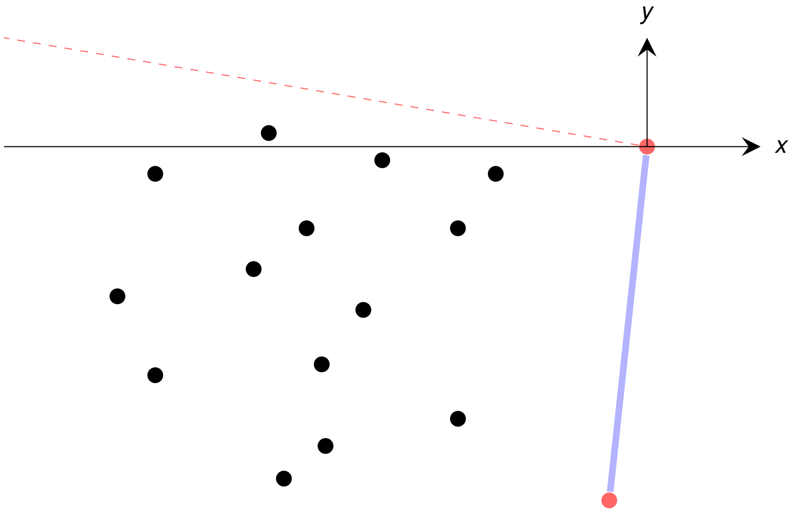
Execution of Jarvis' March



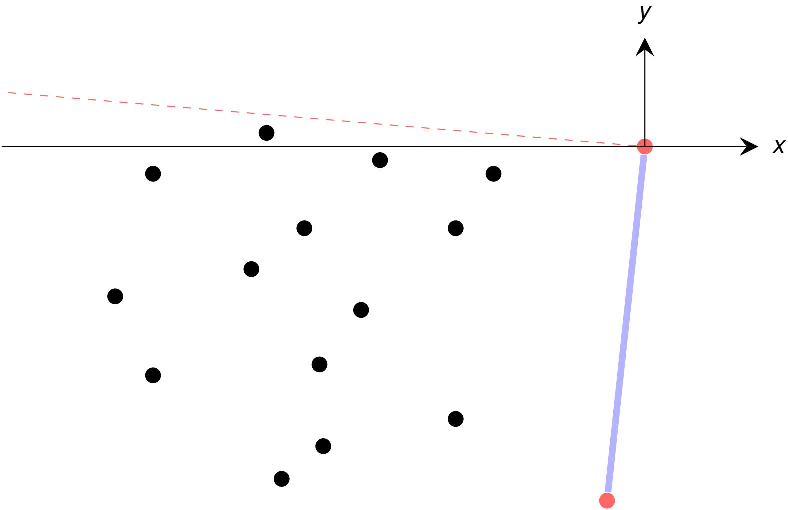
Execution of Jarvis' March



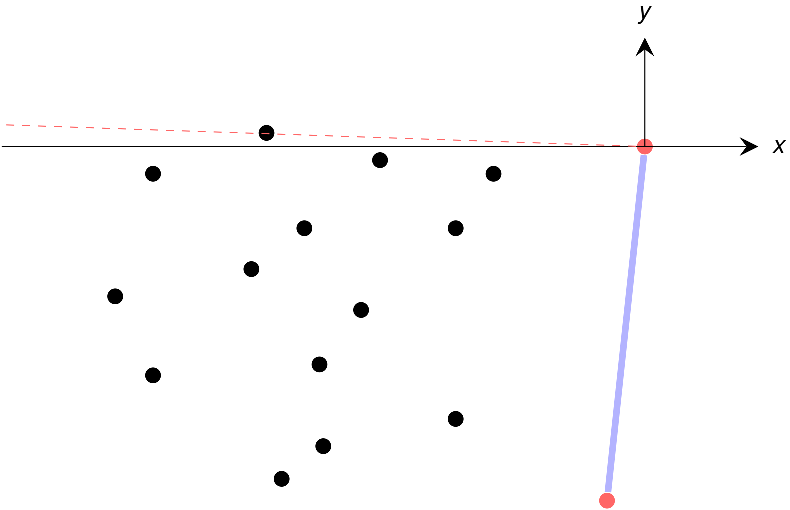
Execution of Jarvis' March



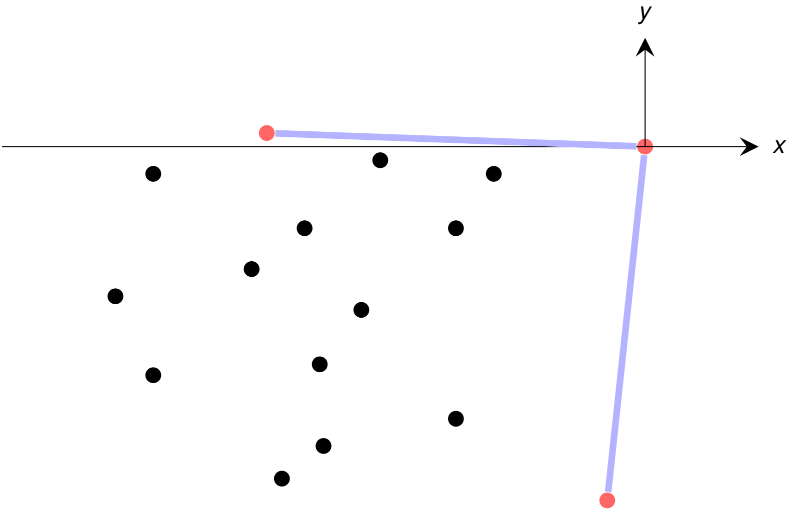
Execution of Jarvis' March



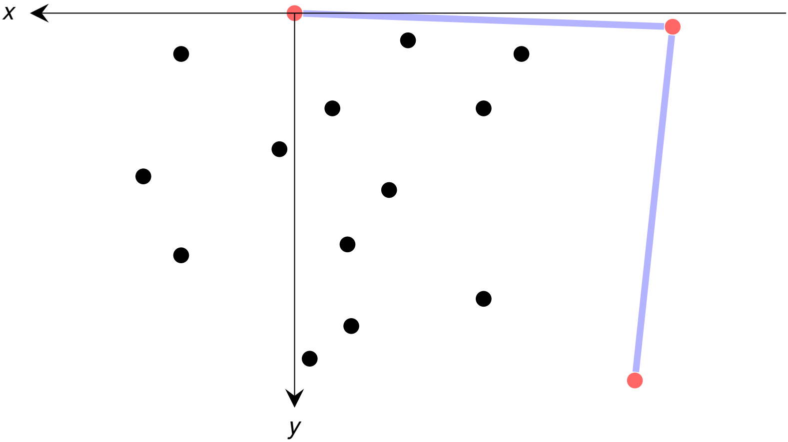
Execution of Jarvis' March



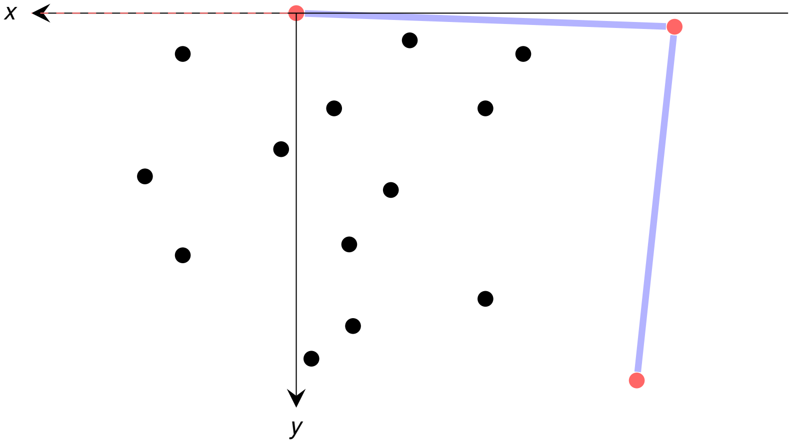
Execution of Jarvis' March



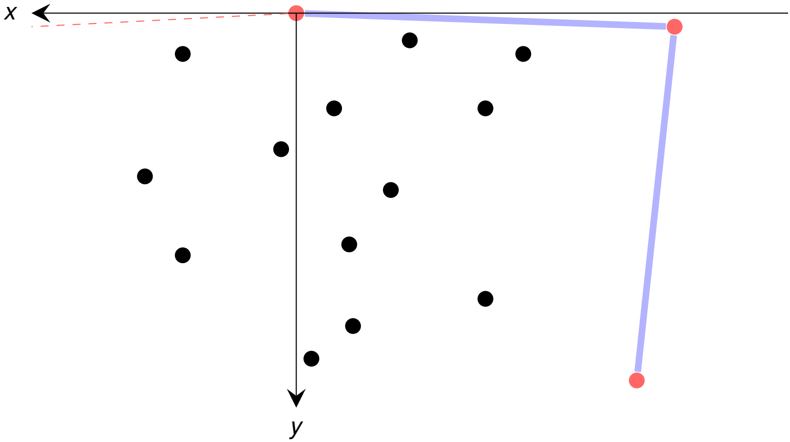
Execution of Jarvis' March



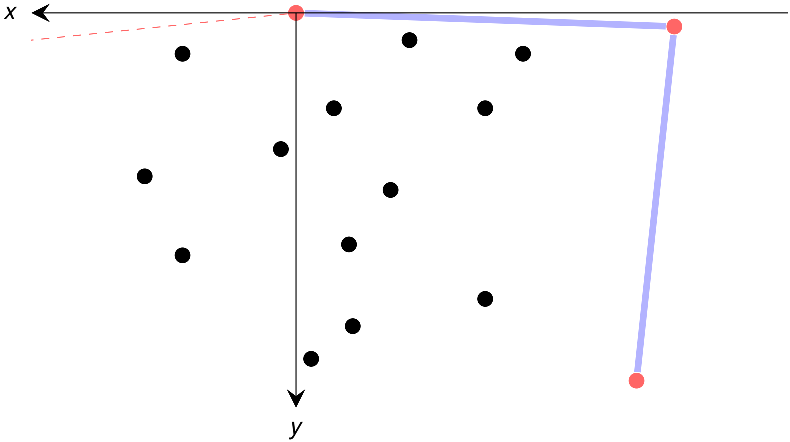
Execution of Jarvis' March



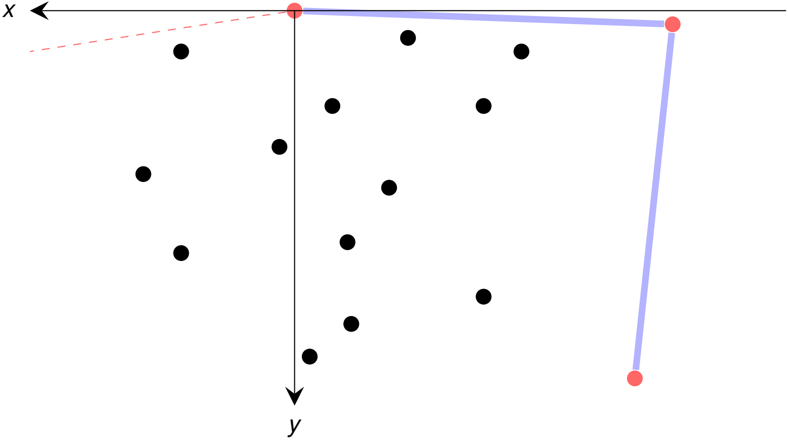
Execution of Jarvis' March



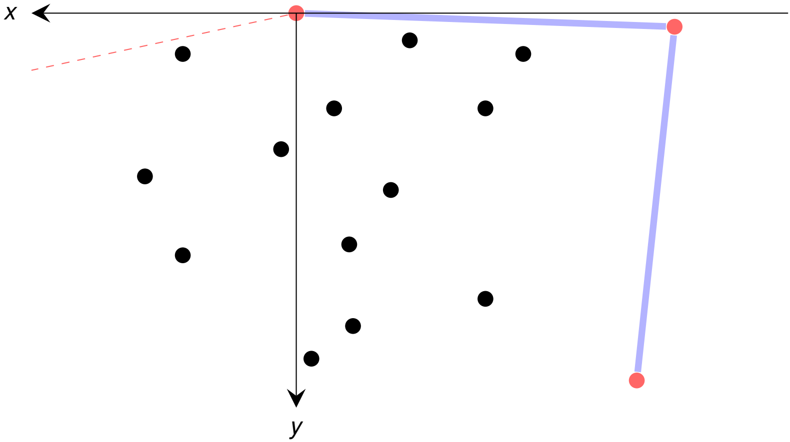
Execution of Jarvis' March



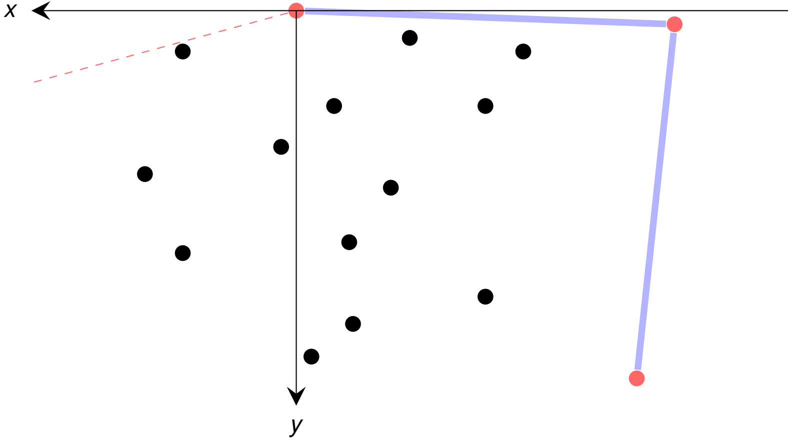
Execution of Jarvis' March



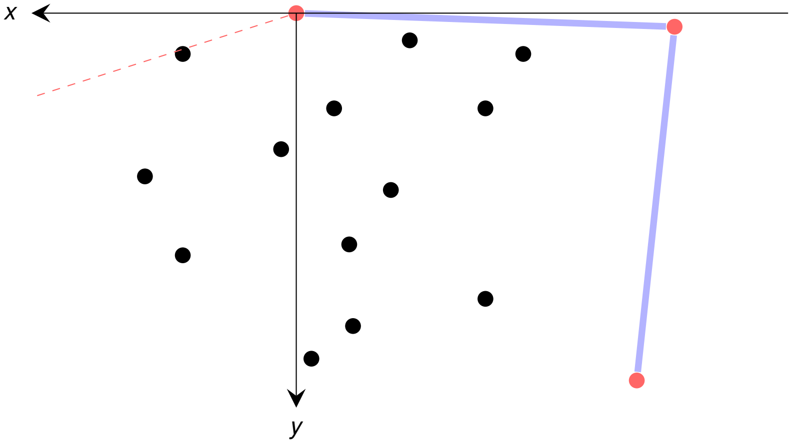
Execution of Jarvis' March



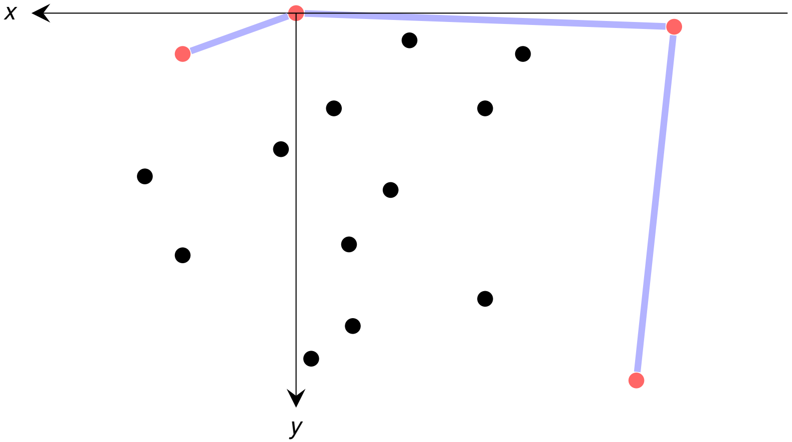
Execution of Jarvis' March



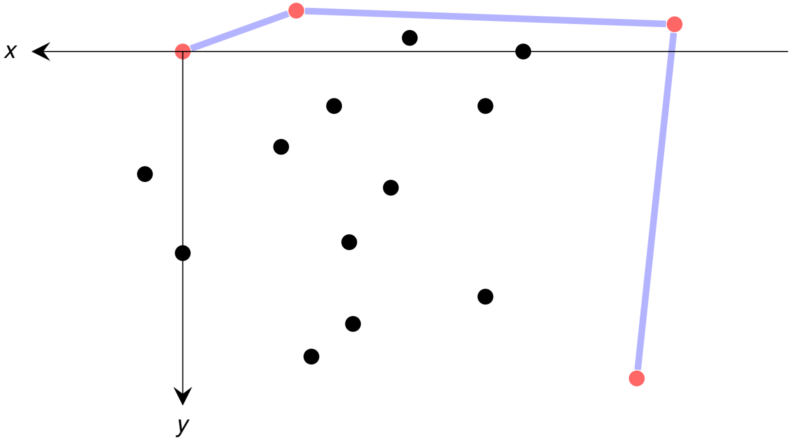
Execution of Jarvis' March



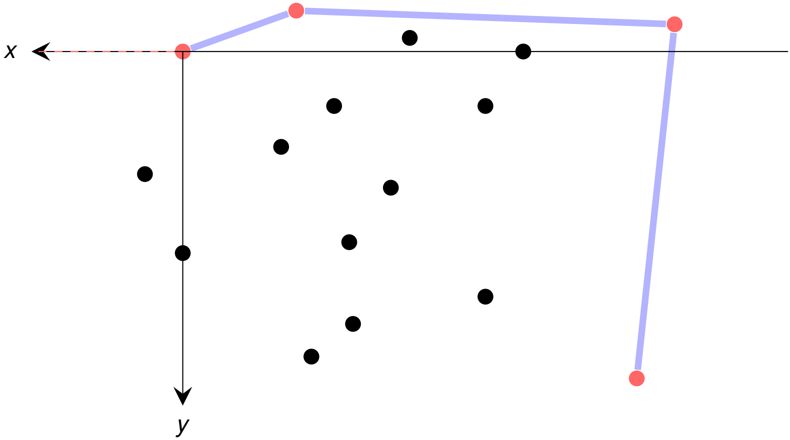
Execution of Jarvis' March



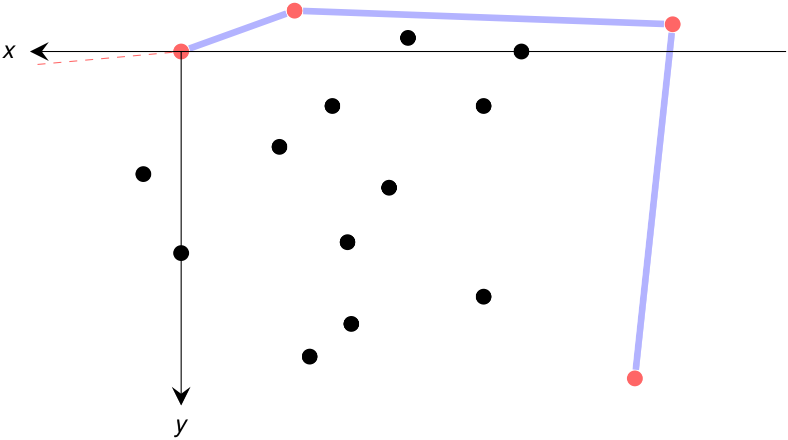
Execution of Jarvis' March



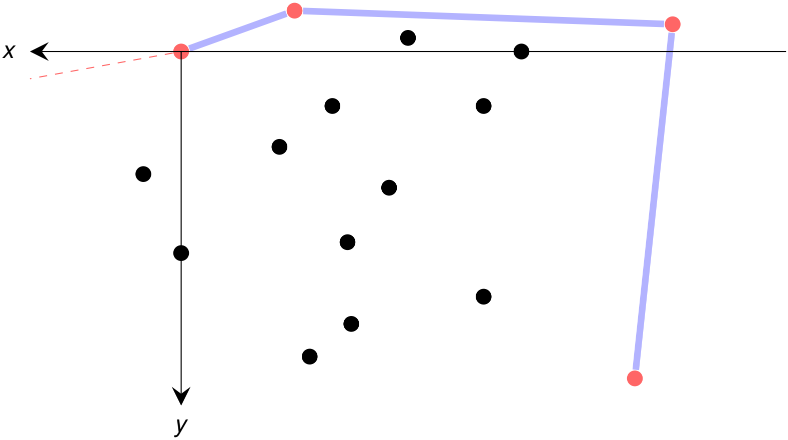
Execution of Jarvis' March



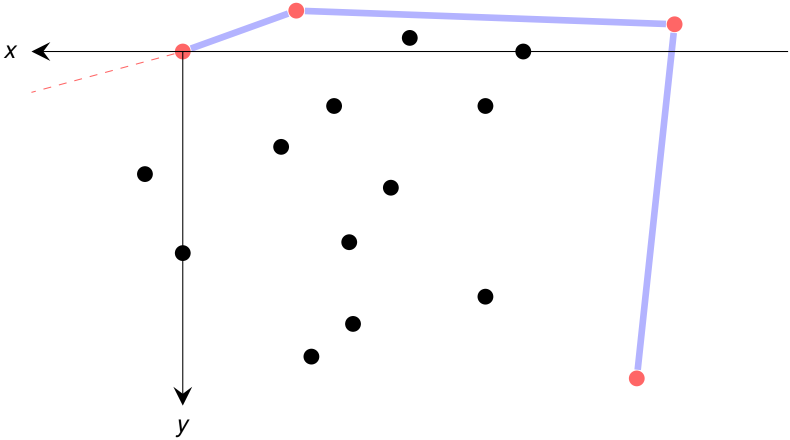
Execution of Jarvis' March



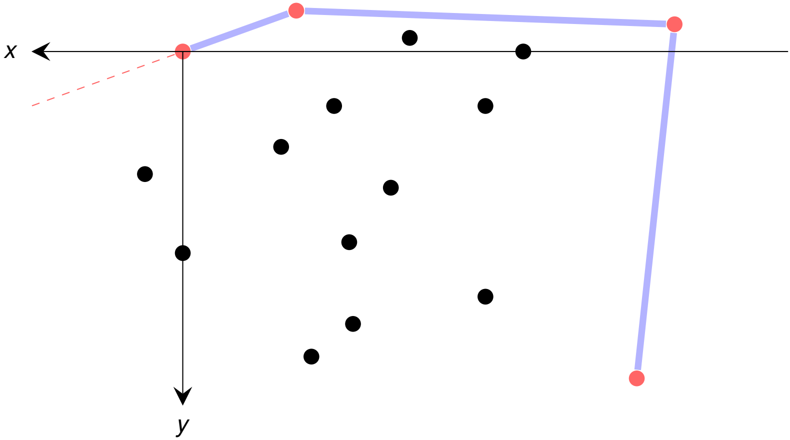
Execution of Jarvis' March



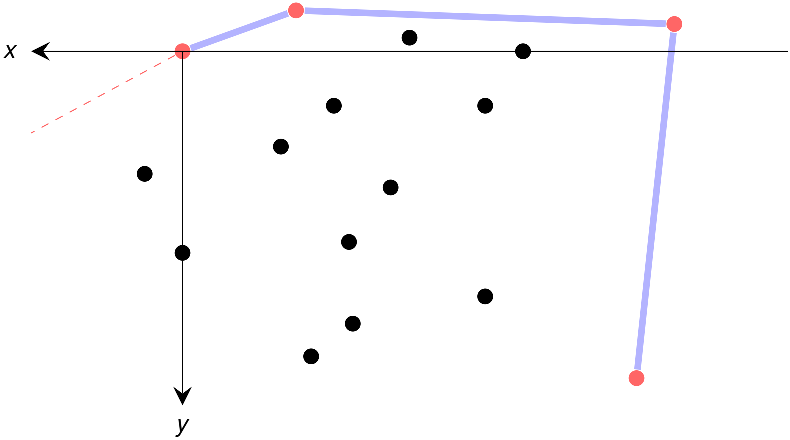
Execution of Jarvis' March



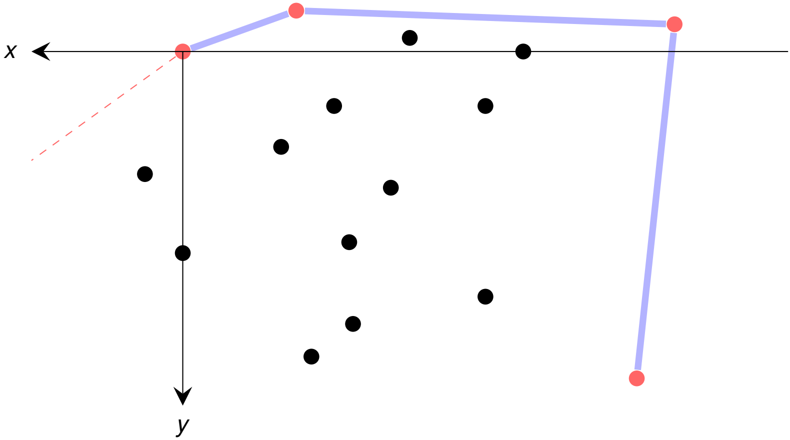
Execution of Jarvis' March



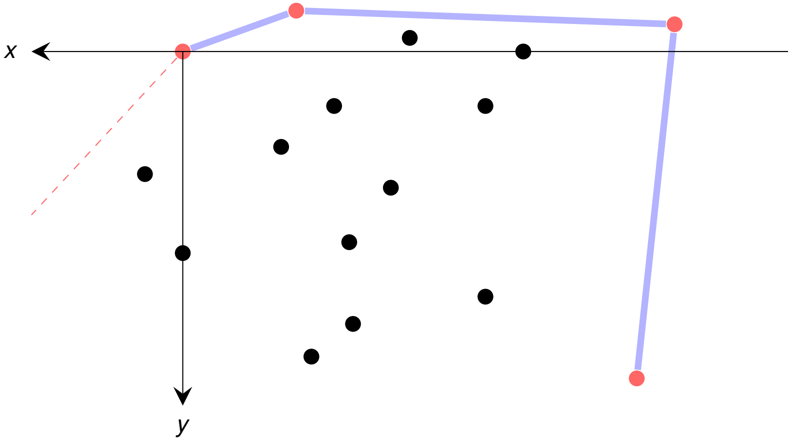
Execution of Jarvis' March



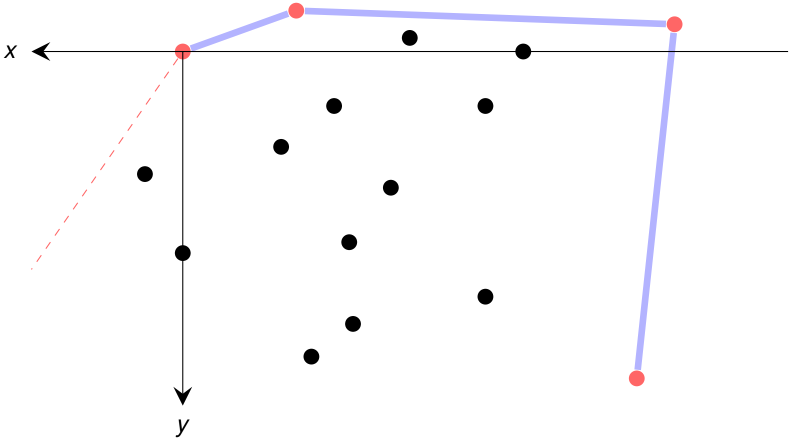
Execution of Jarvis' March



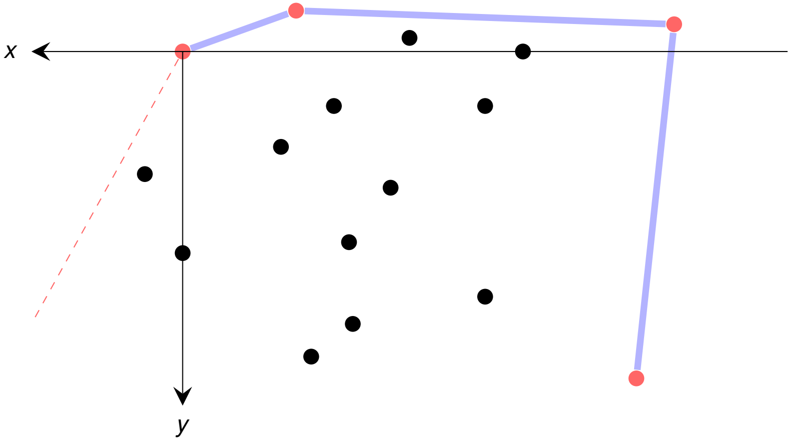
Execution of Jarvis' March



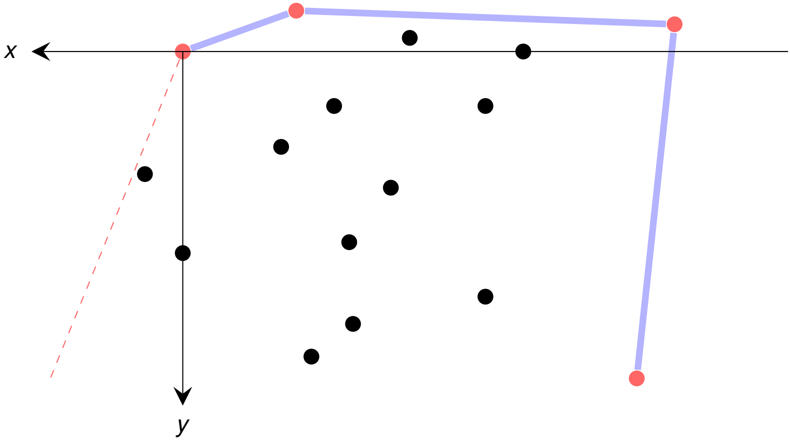
Execution of Jarvis' March



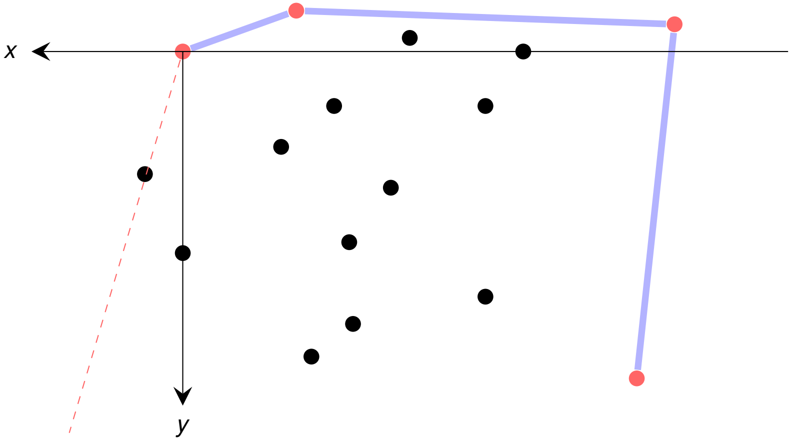
Execution of Jarvis' March



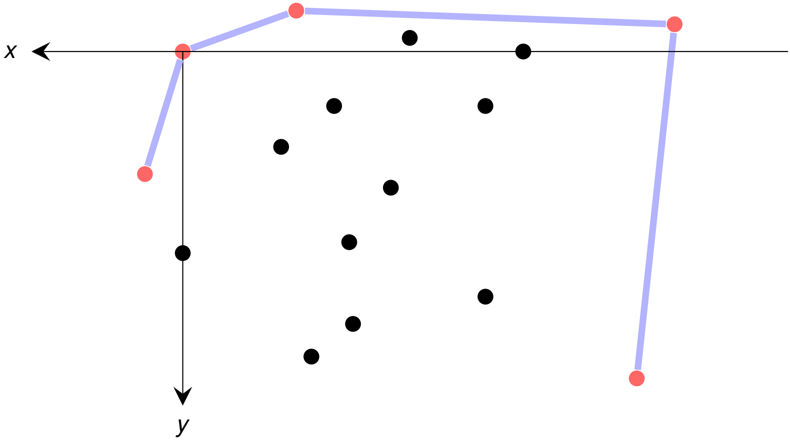
Execution of Jarvis' March



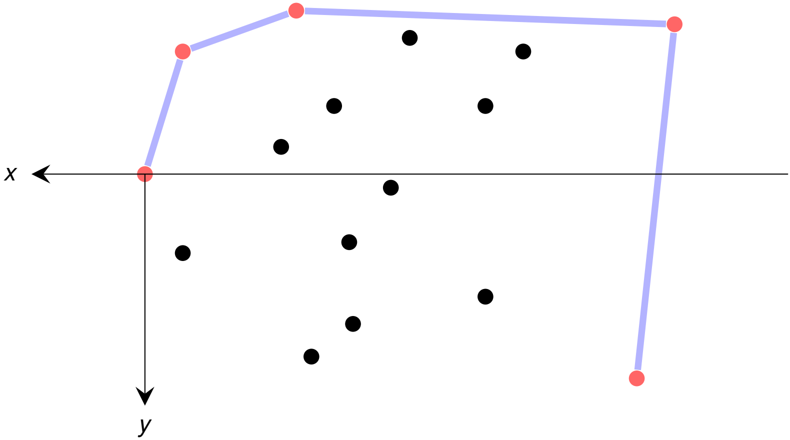
Execution of Jarvis' March



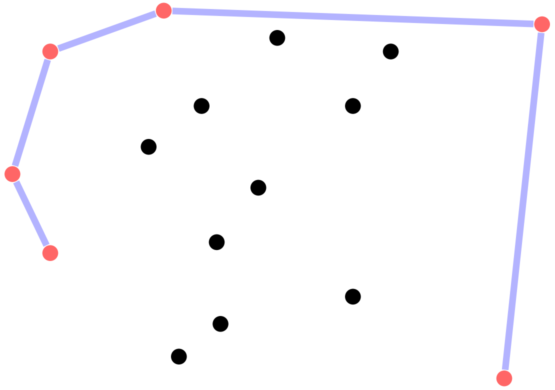
Execution of Jarvis' March



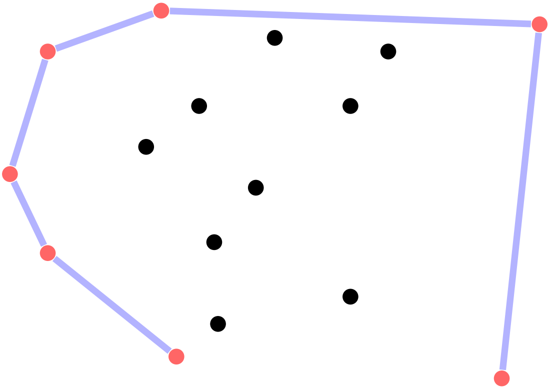
Execution of Jarvis' March



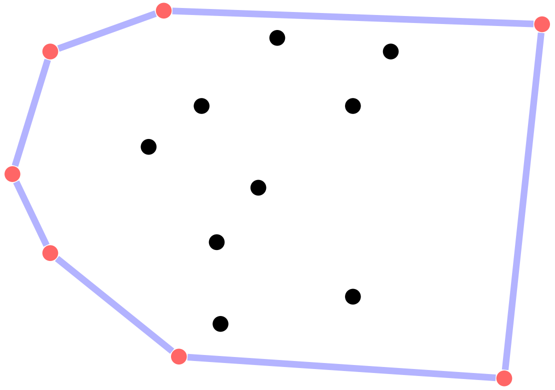
Execution of Jarvis' March



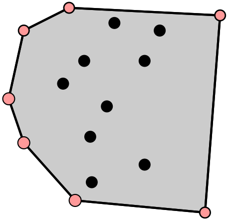
Execution of Jarvis' March



Execution of Jarvis' March

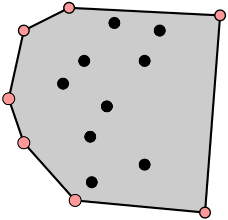


Computing Convex Hull: Summary



Computing Convex Hull: Summary

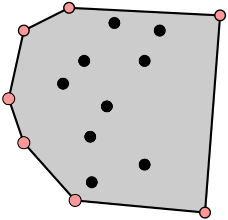
Graham's Scan



Computing Convex Hull: Summary

Graham's Scan

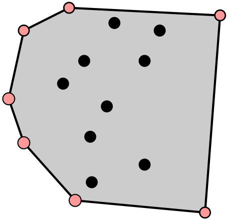
- natural backtracking algorithm



Computing Convex Hull: Summary

Graham's Scan

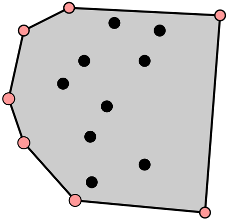
- natural backtracking algorithm
- cross-product avoids computing polar angles



Computing Convex Hull: Summary

Graham's Scan

- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

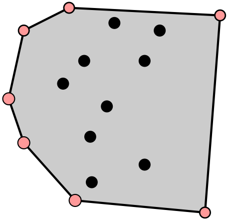


Computing Convex Hull: Summary

Graham's Scan

- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March



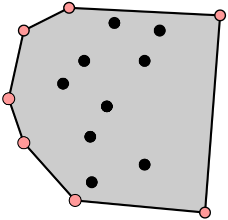
Computing Convex Hull: Summary

Graham's Scan

- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift



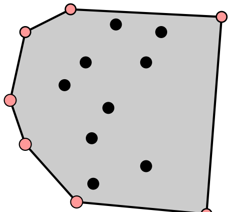
Computing Convex Hull: Summary

Graham's Scan

- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift
- Runtime $O(nh) \rightsquigarrow$ output-sensitive



Computing Convex Hull: Summary

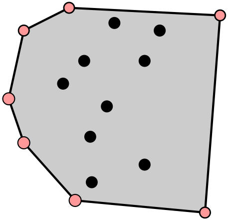
Graham's Scan

- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift
- Runtime $O(nh) \rightsquigarrow$ output-sensitive

Improves Graham's scan only if $h = O(\log n)$



Computing Convex Hull: Summary

Graham's Scan

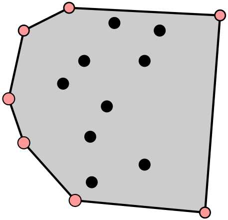
- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift
- Runtime $O(nh) \rightsquigarrow$ output-sensitive

Improves Graham's scan only if $h = O(\log n)$

There exists an algorithm with $O(n \log h)$ runtime!



Computing Convex Hull: Summary

Graham's Scan

- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

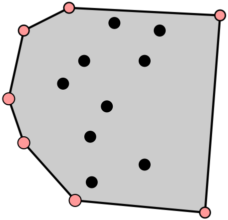
Jarvis' March

- proceeds like wrapping a gift
- Runtime $O(nh) \rightsquigarrow$ output-sensitive

Improves Graham's scan only if $h = O(\log n)$

There exists an algorithm with $O(n \log h)$ runtime!

Lessons Learned



Computing Convex Hull: Summary

Graham's Scan

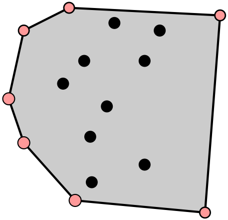
- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift
- Runtime $O(nh) \rightsquigarrow$ output-sensitive

Improves Graham's scan only if $h = O(\log n)$

There exists an algorithm with $O(n \log h)$ runtime!



Lessons Learned

- cross product very powerful tool
  (avoids trigonometry and divison!)

Computing Convex Hull: Summary

Graham's Scan

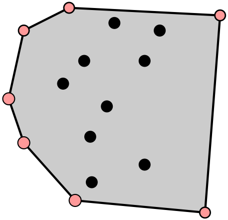
- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift
- Runtime $O(nh) \rightsquigarrow$ output-sensitive

Improves Graham's scan only if $h = O(\log n)$

There exists an algorithm with $O(n \log h)$ runtime!



Lessons Learned

- cross product very powerful tool
  (avoids trigonometry and divison!)
- take care of degenerate cases

Outline

Introduction and Line Intersection

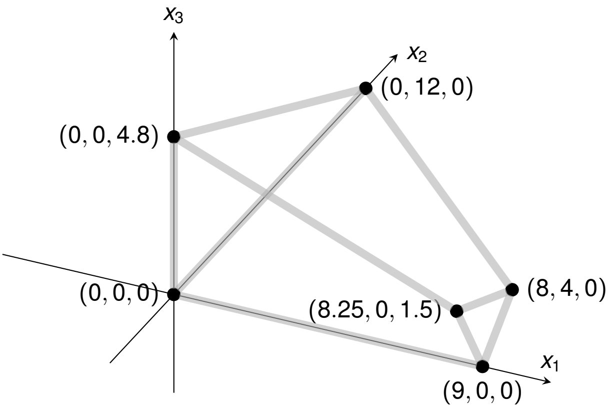
Convex Hull

Glimpse at (More) Advanced Algorithms

$$
\begin{array}{llllllll}
\text{maximize} & 3x_1 & + & x_2 & + & 2x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\
& 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\
& 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\
& & & x_1, x_2, x_3 & & & \geq & 0
\end{array}
$$

Go to End

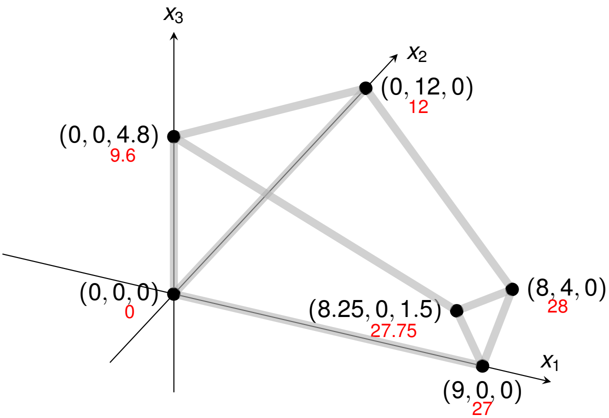
Linear Programming and Simplex



$$
\begin{array}{lrcrcrcr}
\text{maximize} & 3x_1 & + & x_2 & + & 2x_3 & & \\
\text{subject to} & & & & & & & \\
 & x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\
 & 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\
 & 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\
 & & & x_1, x_2, x_3 & & & \geq & 0
\end{array}
$$

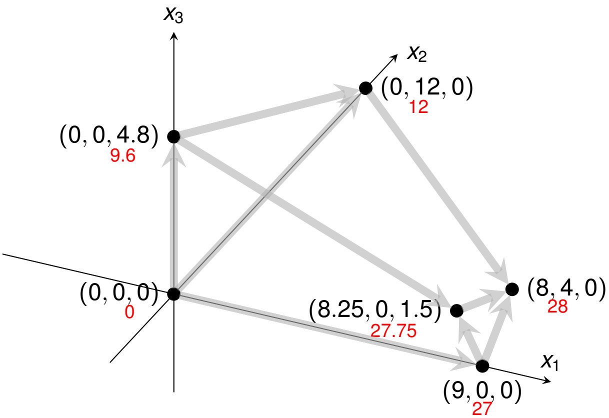
Go to End

Linear Programming and Simplex



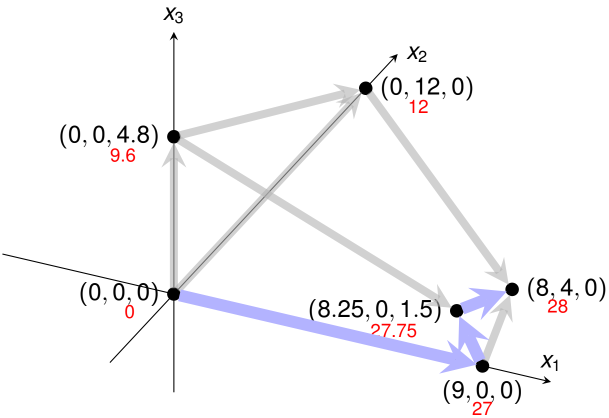
$$
\begin{array}{llllllll}
\text{maximize} & 3x_1 & + & x_2 & + & 2x_3 & & \\
\text{subject to} & & & & & & & \\
 & x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\
 & 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\
 & 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\
 & & & x_1, x_2, x_3 & & & \geq & 0
\end{array}
$$

Go to End



$$
\begin{array}{llllllll}
\text{maximize} & 3x_1 & + & x_2 & + & 2x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\
& 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\
& 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\
& & & x_1, x_2, x_3 & & & \geq & 0
\end{array}
$$

Go to End



maximize $3x_1 + x_2 + 2x_3$

subject to

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\ 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\ 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\ & & x_1, x_2, x_3 & & & \geq & 0 \end{array}$$

Go to End

The Original Article (1954)

SOLUTION OF A LARGE-SCALE TRAVELING-SALESMAN PROBLEM*

G. DANTZIG, R. FULKERSON, AND S. JOHNSON

The Rand Corporation, Santa Monica, California

(Received August 9, 1954)

It is shown that a certain tour of 49 cities, one in each of the 48 states and Washington, D. C., has the shortest road distance.

THE TRAVELING-SALESMAN PROBLEM might be described as follows: Find the shortest route (tour) for a salesman starting from a given city, visiting each of a specified group of cities, and then returning to the original point of departure. More generally, given an n by n symmetric matrix $D=(d_{IJ})$, where d_{IJ} represents the 'distance' from I to J, arrange the points in a cyclic order in such a way that the sum of the d_{IJ} between consecutive points is minimal. Since there are only a finite number of possibilities (at most $\frac{1}{2}(n-1)!$) to consider, the problem is to devise a method of picking out the optimal arrangement which is reasonably efficient for fairly large values of n. Although algorithms have been devised for problems of similar nature, e.g., the optimal assignment problem,[3,7,8] little is known about the traveling-salesman problem. We do not claim that this note alters the situation very much; what we shall do is outline a way of approaching the problem that sometimes, at least, enables one to find an optimal path and prove it so. In particular, it will be shown that a certain arrangement of 49 cities, one in each of the 48 states and Washington, D. C., is best, the d_{IJ} used representing road distances as taken from an atlas.

Travelling Salesman Problem: The 42 (49) Cities

1. Manchester, N. H.
2. Montpelier, Vt.
3. Detroit, Mich.
4. Cleveland, Ohio
5. Charleston, W. Va.
6. Louisville, Ky.
7. Indianapolis, Ind.
8. Chicago, Ill.
9. Milwaukee, Wis.
10. Minneapolis, Minn.
11. Pierre, S. D.
12. Bismarck, N. D.
13. Helena, Mont.
14. Seattle, Wash.
15. Portland, Ore.
16. Boise, Idaho
17. Salt Lake City, Utah
18. Carson City, Nev.
19. Los Angeles, Calif.
20. Phoenix, Ariz.
21. Santa Fe, N. M.
22. Denver, Colo.
23. Cheyenne, Wyo.
24. Omaha, Neb.
25. Des Moines, Iowa
26. Kansas City, Mo.
27. Topeka, Kans.
28. Oklahoma City, Okla.
29. Dallas, Tex.
30. Little Rock, Ark.
31. Memphis, Tenn.
32. Jackson, Miss.
33. New Orleans, La.
34. Birmingham, Ala.
35. Atlanta, Ga.
36. Jacksonville, Fla.
37. Columbia, S. C.
38. Raleigh, N. C.
39. Richmond, Va.
40. Washington, D. C.
41. Boston, Mass.
42. Portland, Me.

A. Baltimore, Md.
B. Wilmington, Del.
C. Philadelphia, Penn.
D. Newark, N. J.
E. New York, N. Y.
F. Hartford, Conn.
G. Providence, R. I.

Road Distances

TABLE I

ROAD DISTANCES BETWEEN CITIES IN ADJUSTED UNITS

The figures in the table are mileages between the two specified numbered cities, less 11, divided by 17, and rounded to the nearest integer.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
2	8																																								
3	39	45																																							
4	37	47	9																																						
5	50	49	21	15																																					
6	61	62	21	20	17																																				
7	58	60	16	17	18	6																																			
8	59	60	15	20	26	17	10																																		
9	62	66	20	25	31	22	15	5																																	
10	81	81	40	44	50	41	35	24	20																																
11	103	107	62	67	72	63	57	46	41	23																															
12	108	117	66	71	77	68	61	51	46	26	11																														
13	145	149	104	108	114	106	99	88	84	63	49	40																													
14	181	185	140	144	150	142	135	124	120	99	85	76	35																												
15	187	191	146	150	156	142	137	130	125	105	90	81	41	10																											
16	161	170	120	124	130	115	110	104	105	90	72	64	34	31	27																										
17	142	146	101	104	111	97	91	85	86	75	51	59	29	53	48	21																									
18	174	178	133	138	143	129	123	117	118	107	83	84	54	46	35	26	31																								
19	185	186	142	143	140	130	126	124	128	118	93	101	72	69	58	58	43	26																							
20	164	165	120	123	124	106	106	105	110	104	86	97	71	93	82	62	42	45	22																						
21	137	139	94	96	94	80	78	77	84	77	56	64	65	90	87	58	36	68	50	30																					
22	117	122	77	80	83	68	62	60	61	50	34	42	49	82	77	60	30	62	70	49	21																				
23	114	118	73	78	84	69	63	57	59	48	28	36	43	77	72	45	27	59	69	55	27	5																			
24	85	89	44	48	53	41	34	28	29	22	23	35	69	105	102	74	56	88	99	81	54	32	29																		
25	77	80	36	40	46	34	27	19	21	14	29	40	77	114	111	84	64	96	107	87	60	40	37	8																	
26	87	89	44	46	46	30	28	29	32	27	36	47	78	116	112	84	66	98	95	75	47	36	39	12	11																
27	91	93	48	50	48	34	32	33	36	30	34	45	77	115	110	83	63	97	91	72	44	32	36	9	15	3															
28	105	106	62	63	64	47	46	49	54	48	46	59	85	119	115	88	66	98	79	59	31	36	42	28	33	21	20														
29	111	113	69	71	66	51	53	56	61	57	59	71	96	130	126	98	75	98	85	62	38	47	53	39	42	29	30	12													
30	91	92	50	51	46	30	34	38	43	49	60	71	103	141	136	109	90	115	99	81	53	61	62	36	34	24	28	20	20												
31	83	85	42	43	38	22	26	32	36	51	63	75	106	142	140	112	93	126	108	88	60	64	66	39	36	27	31	28	28	8											
32	89	91	55	55	50	34	39	44	49	63	76	87	120	155	150	123	100	123	109	86	62	71	78	52	49	39	44	35	24	15	12										
33	95	97	64	63	56	42	49	56	60	75	86	97	126	160	155	128	104	128	113	90	67	76	82	62	59	49	53	40	29	25	23	11									
34	74	81	44	43	35	23	30	39	44	62	78	89	121	159	155	127	108	136	124	101	75	79	81	54	50	42	46	43	39	23	14	14	21								
35	67	69	42	41	31	25	32	41	46	64	83	90	130	164	160	133	114	146	134	111	85	84	86	59	52	47	51	53	49	32	24	24	30	9							
36	74	76	61	60	42	44	51	60	66	83	102	110	147	185	179	155	133	159	146	122	98	105	107	79	71	66	70	70	60	48	40	36	33	25	18						
37	57	59	46	41	25	30	36	47	52	71	93	98	136	172	172	148	126	158	147	124	121	97	99	71	65	59	63	67	62	46	38	37	43	23	13	17					
38	45	46	41	34	20	34	38	48	53	73	96	99	137	176	178	151	131	163	159	135	108	102	103	73	67	64	69	75	72	54	46	49	54	34	24	29	12				
39	35	37	35	26	18	34	36	46	51	70	93	97	134	171	176	151	129	161	163	139	118	102	101	71	65	65	70	84	78	58	50	56	62	41	32	38	21	9			
40	29	33	30	21	18	35	33	40	45	65	87	91	117	166	171	144	125	157	156	139	113	95	97	67	60	62	67	79	82	62	53	59	66	45	38	45	27	15	6		
41	3	11	41	37	47	57	55	58	63	83	105	109	147	186	188	164	144	176	182	161	134	119	116	86	78	84	88	101	108	88	80	86	92	71	64	71	54	41	32	25	
42	5	12	55	41	53	64	61	61	66	84	111	113	150	186	192	166	147	180	188	167	140	124	119	90	87	90	94	107	114	77	86	92	98	80	74	77	60	48	38	32	6

The (Unique) Optimal Tour (699 Units $\approx$ 12,345 miles)

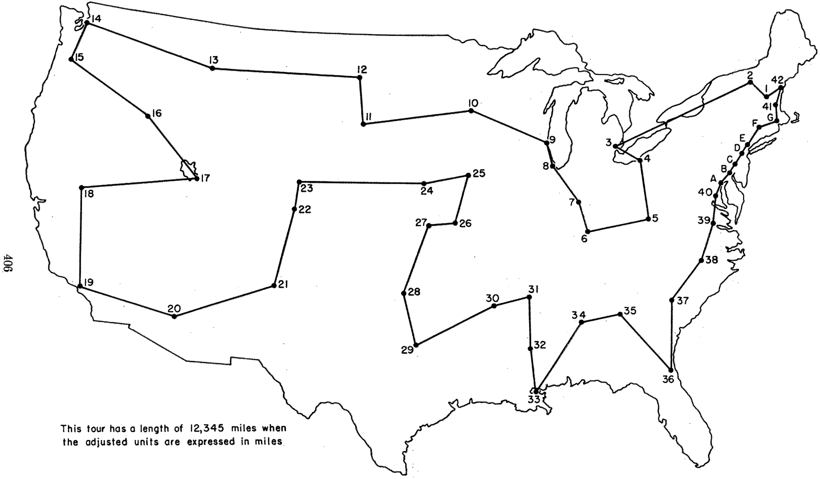
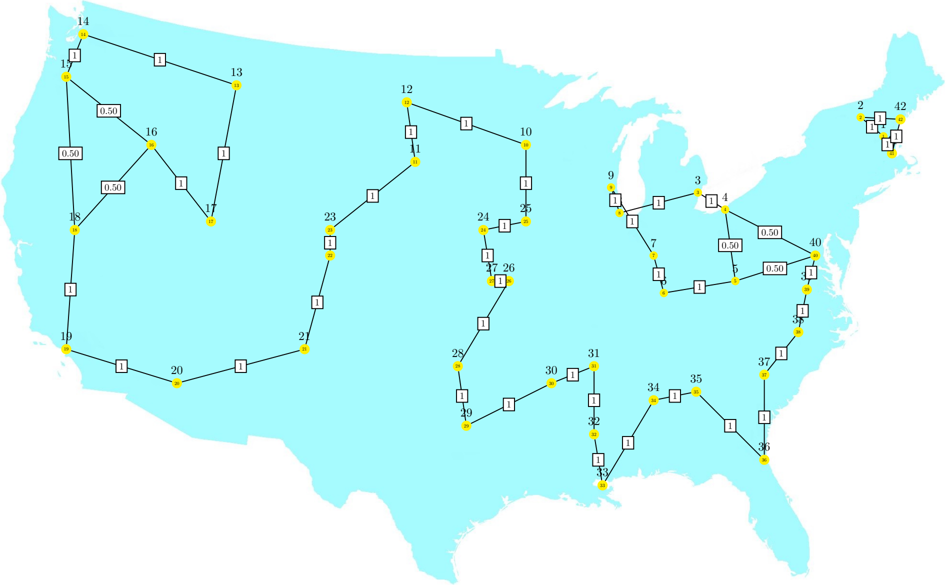
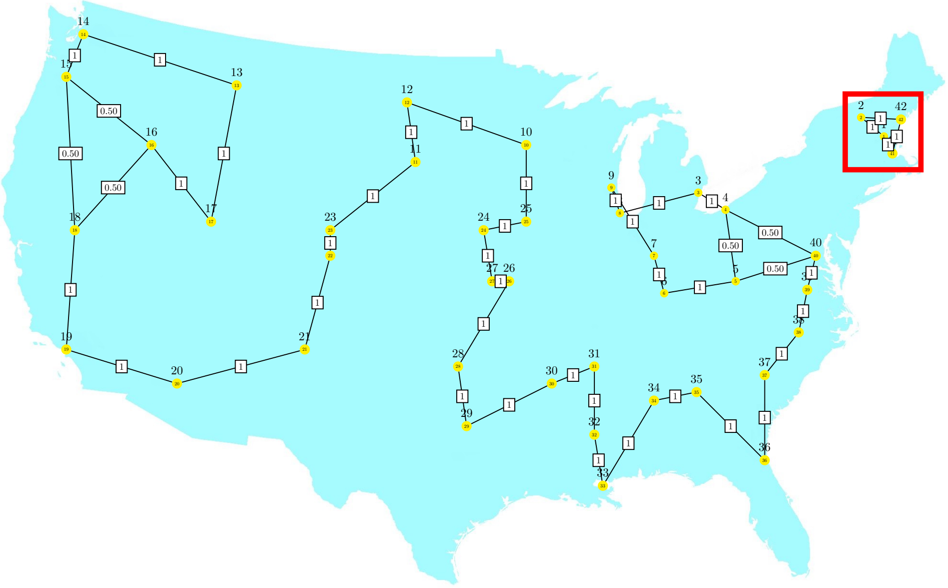


FIG. 16. The optimal tour of 49 cities.

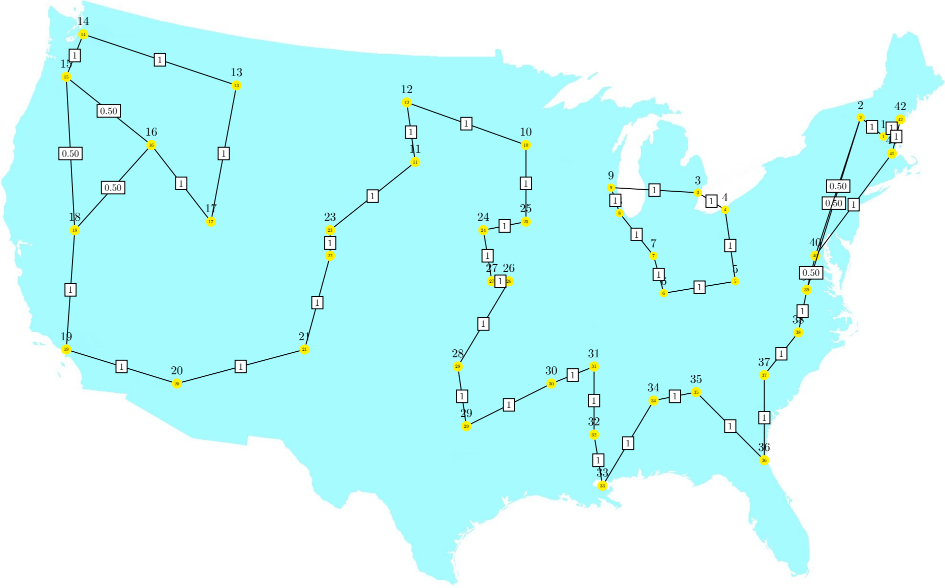
Iteration 1: Objective 641



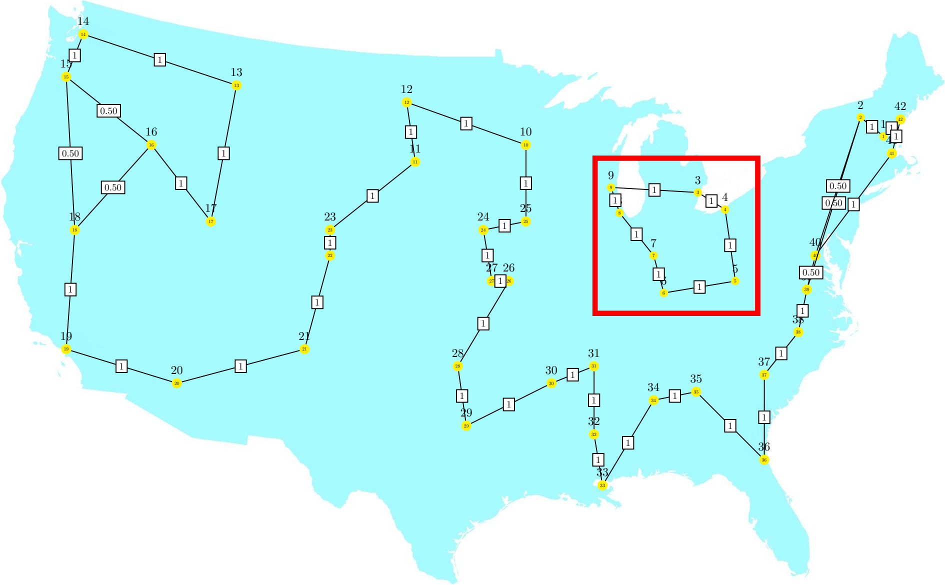
Iteration 1: Objective 641, Eliminate Subtour $1, 2, 41, 42$



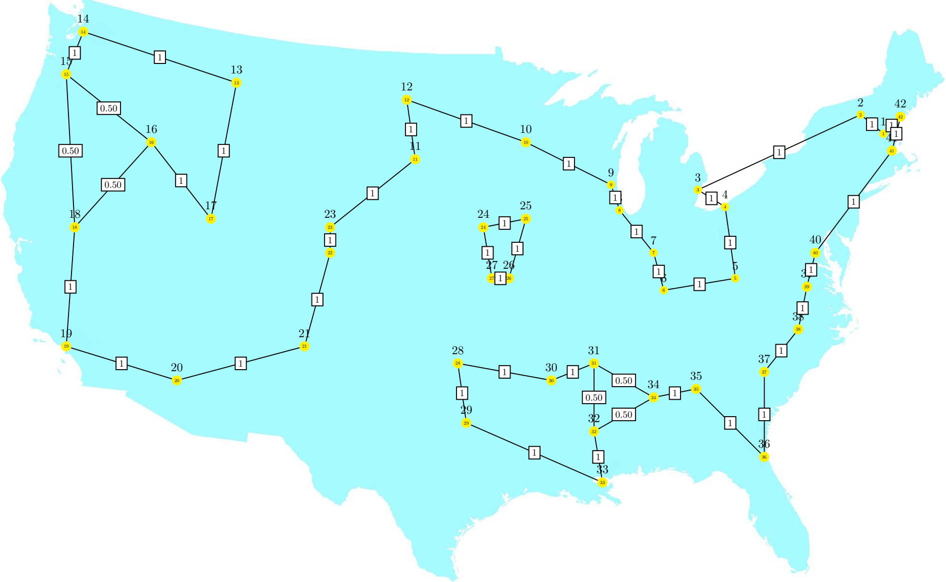
Iteration 2: Objective 676



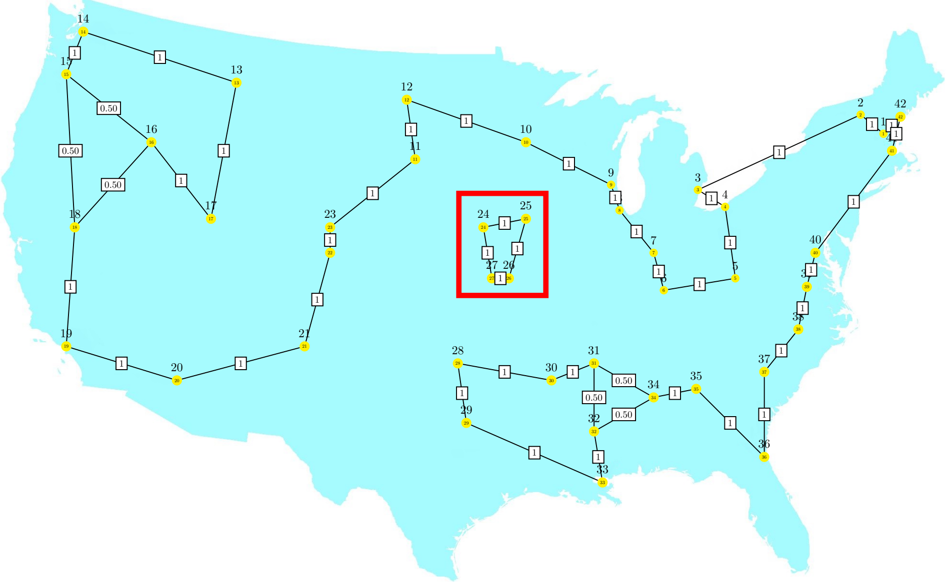
Iteration 2: Objective 676, Eliminate Subtour $3-9$



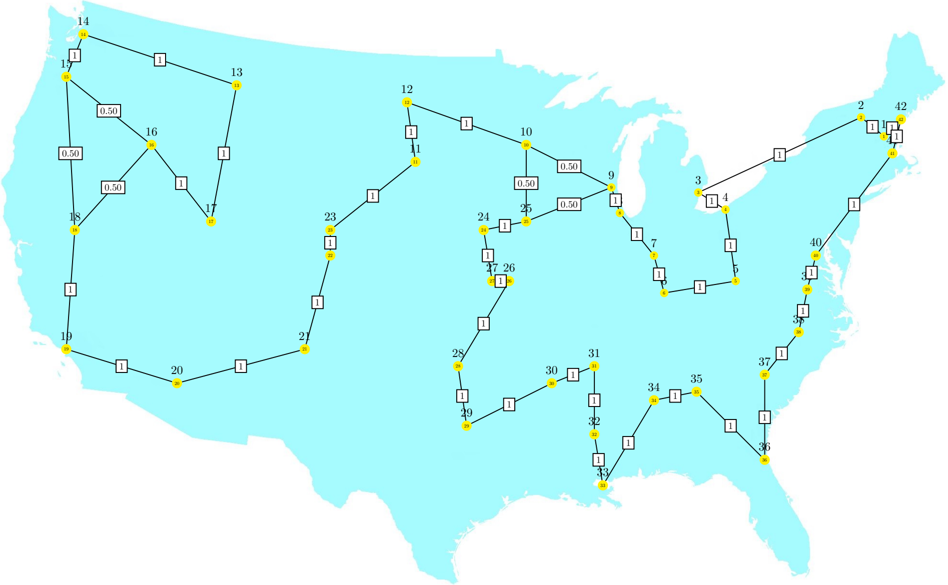
Iteration 3: Objective 681



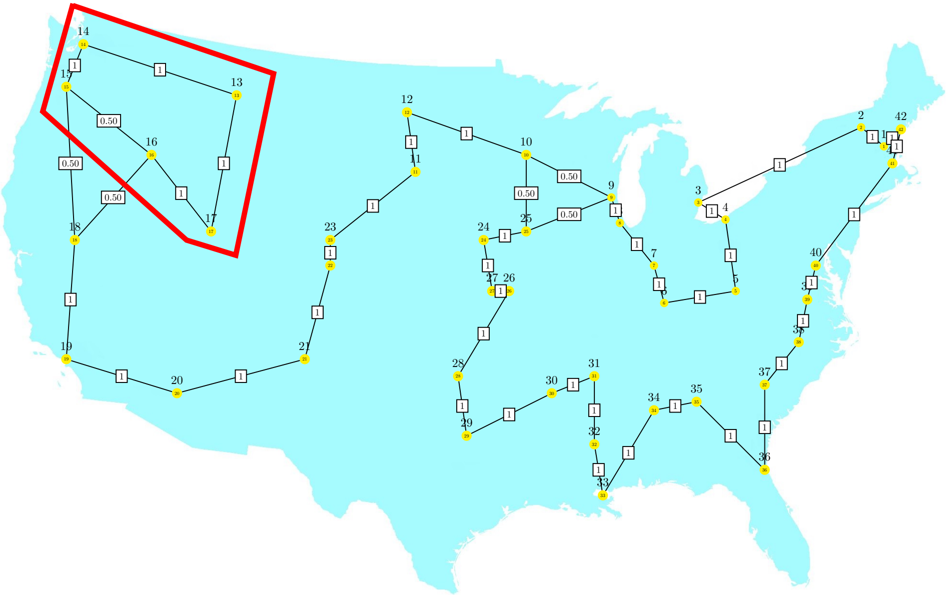
Iteration 3: Objective 681, Eliminate Subtour $24, 25, 26, 27$



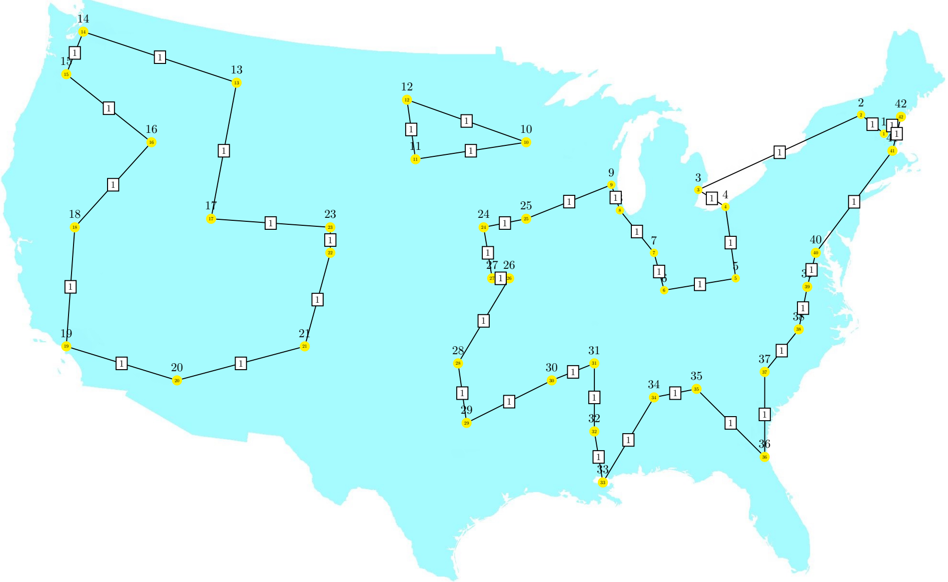
Iteration 4: Objective 682.5



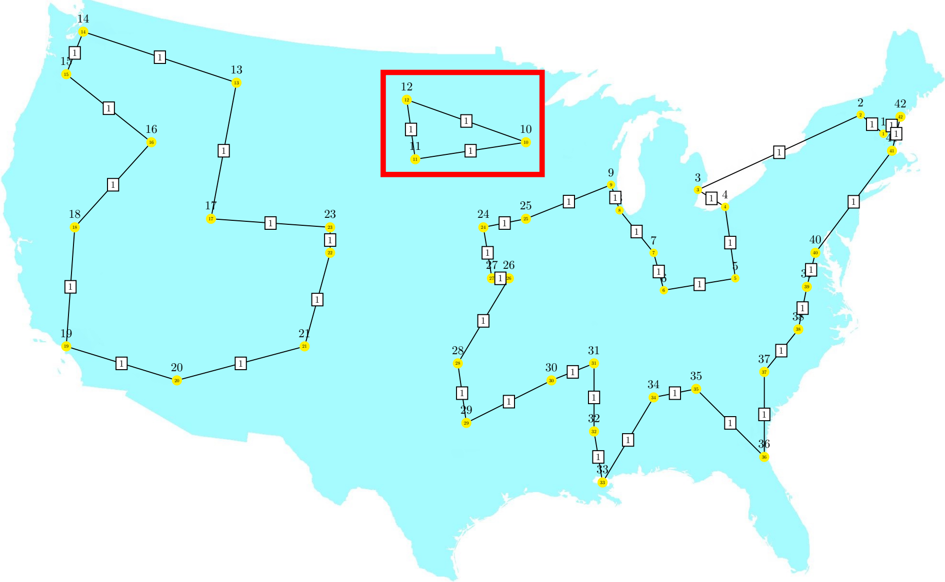
Iteration 4: Objective 682.5, Eliminate Small Cut by $13 - 17$



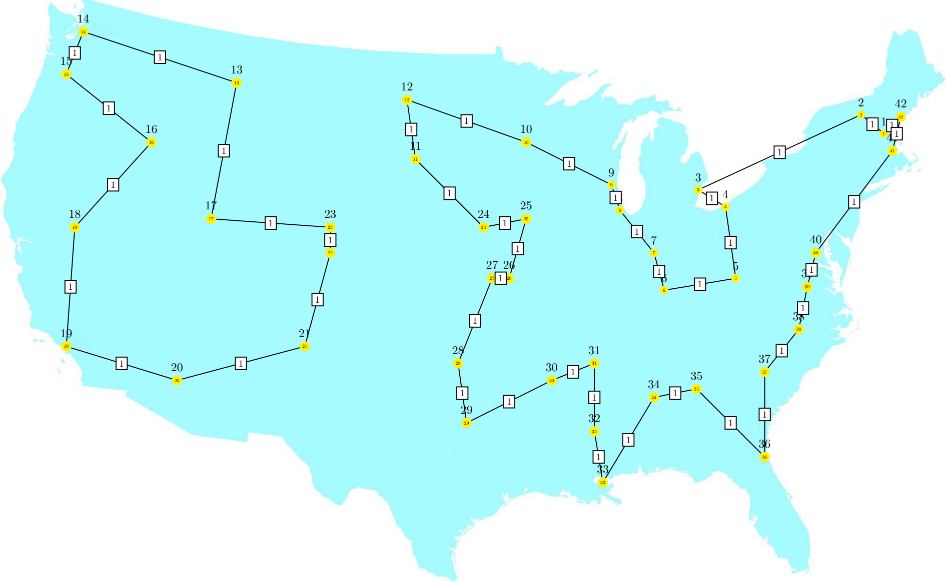
Iteration 5: Objective 686



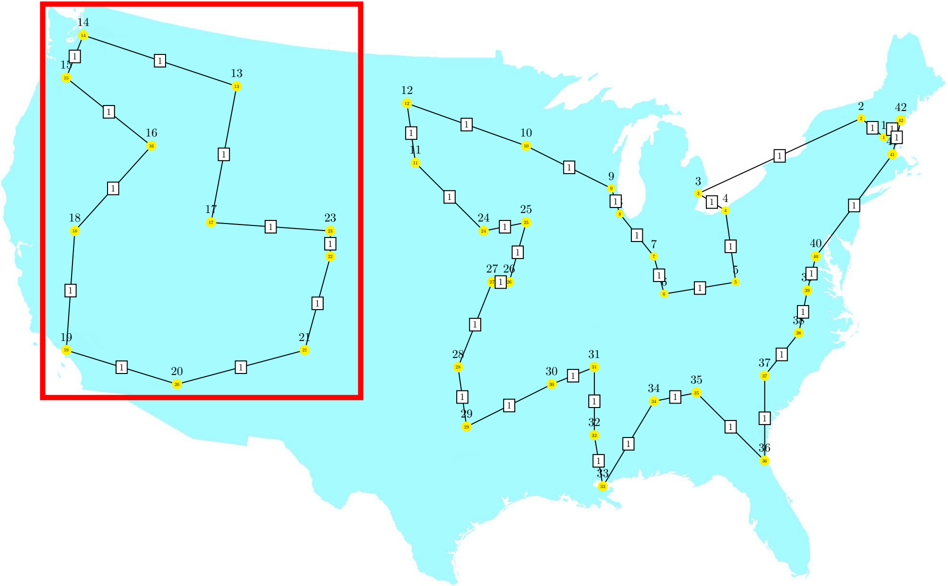
Iteration 5: Objective 686, Eliminate Subtour $10, 11, 12$



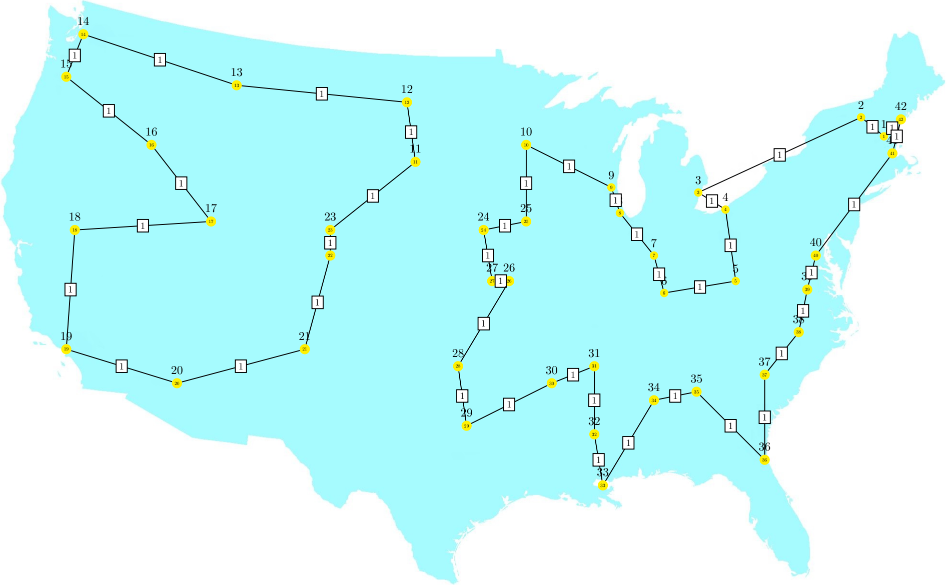
Iteration 6: Objective 686



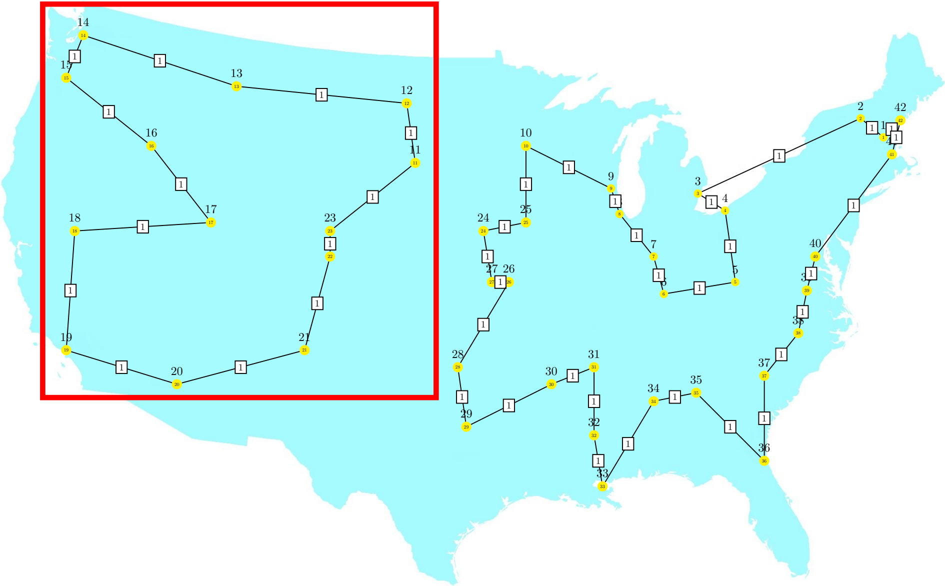
Iteration 6: Objective 686, Eliminate Subtour $13 - 23$



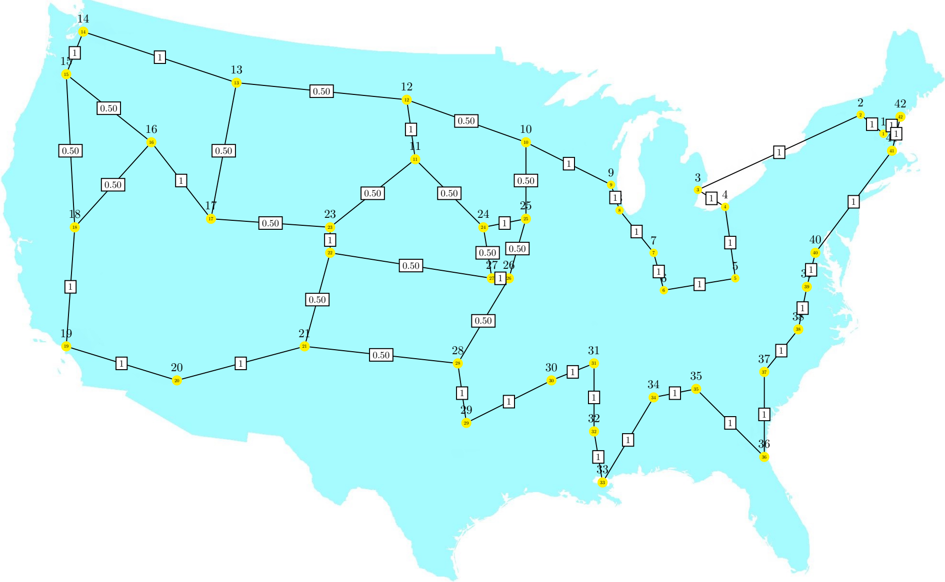
Iteration 7: Objective 688



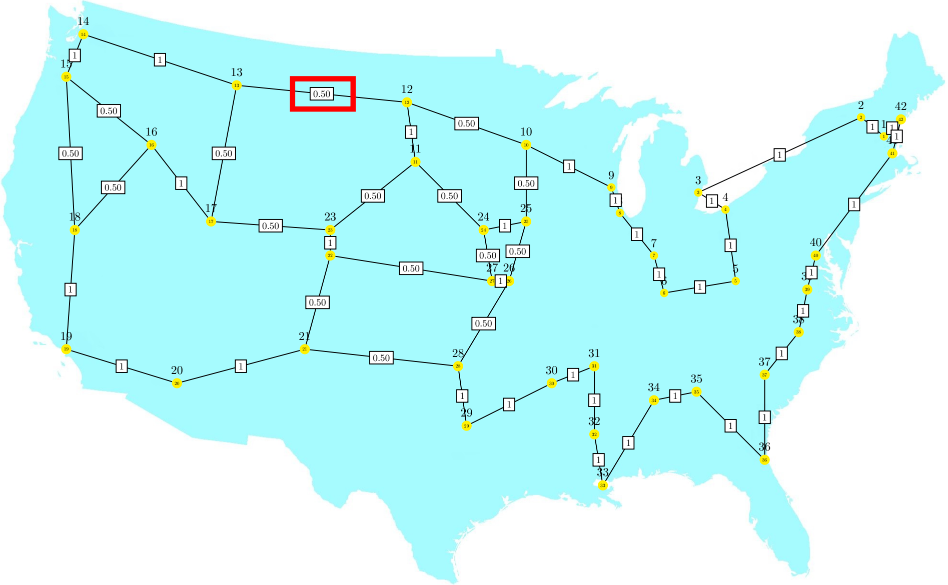
Iteration 7: Objective 688, Eliminate Subtour $11 - 23$



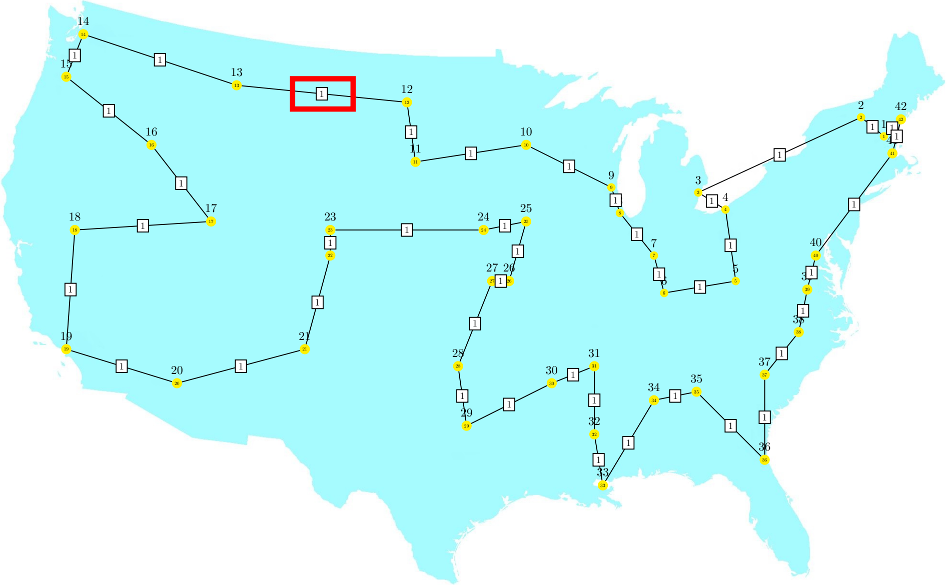
Iteration 8: Objective 697



Iteration 8: Objective 697, Branch on $x(13, 12)$



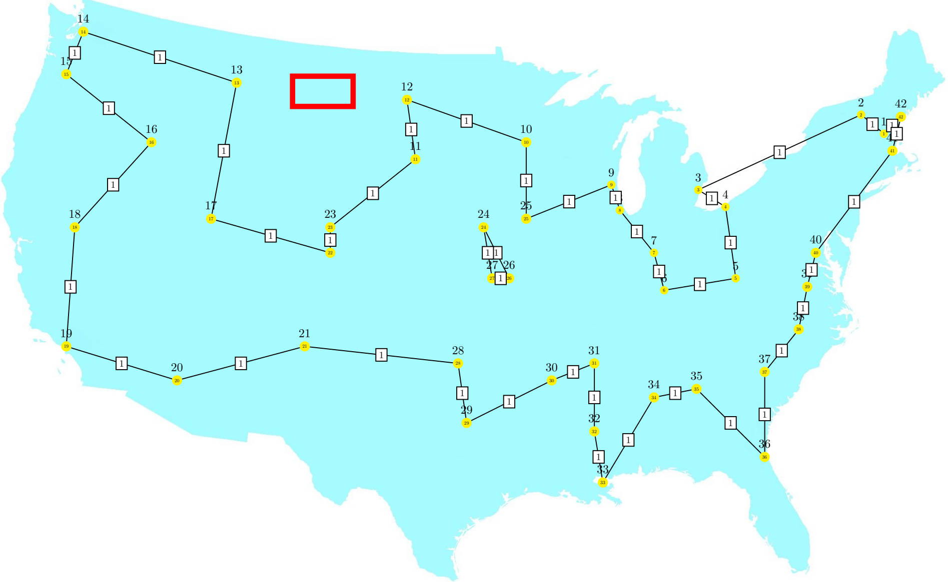
Iteration 9, Branch a $x(13, 12) = 1$: Objective 699 (Valid Tour)



```
Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 12.6.1.0
  with Simplex, Mixed Integer & Barrier Optimizers
5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
Copyright IBM Corp. 1988, 2014.  All Rights Reserved.

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> read tsp.lp
Problem 'tsp.lp' read.
Read time = 0.00 sec. (0.06 ticks)
CPLEX> primopt
Tried aggregator 1 time.
LP Presolve eliminated 1 rows and 1 columns.
Reduced LP has 49 rows, 860 columns, and 2483 nonzeros.
Presolve time = 0.00 sec. (0.36 ticks)

Iteration log . . .
Iteration:     1   Infeasibility =            33.999999
Iteration:    26   Objective     =          1510.000000
Iteration:    90   Objective     =           923.000000
Iteration:   155   Objective     =           711.000000

Primal simplex - Optimal:  Objective =  6.9900000000e+02
Solution time =    0.00 sec.  Iterations = 168 (25)
Deterministic time = 1.16 ticks  (288.86 ticks/sec)

CPLEX> 
```

```
CPLEX> display solution variables -
Variable Name           Solution Value
x_2_1                         1.000000
x_42_1                        1.000000
x_3_2                         1.000000
x_4_3                         1.000000
x_5_4                         1.000000
x_6_5                         1.000000
x_7_6                         1.000000
x_8_7                         1.000000
x_9_8                         1.000000
x_10_9                        1.000000
x_11_10                       1.000000
x_12_11                       1.000000
x_13_12                       1.000000
x_14_13                       1.000000
x_15_14                       1.000000
x_16_15                       1.000000
x_17_16                       1.000000
x_18_17                       1.000000
x_19_18                       1.000000
x_20_19                       1.000000
x_21_20                       1.000000
x_22_21                       1.000000
x_23_22                       1.000000
x_24_23                       1.000000
x_25_24                       1.000000
x_26_25                       1.000000
x_27_26                       1.000000
x_28_27                       1.000000
x_29_28                       1.000000
x_30_29                       1.000000
x_31_30                       1.000000
x_32_31                       1.000000
x_33_32                       1.000000
x_34_33                       1.000000
x_35_34                       1.000000
x_36_35                       1.000000
x_37_36                       1.000000
x_38_37                       1.000000
x_39_38                       1.000000
x_40_39                       1.000000
x_41_40                       1.000000
x_42_41                       1.000000
All other variables in the range 1-861 are 0.
```

Iteration 10, Branch b $x(13, 12) = 0$: Objective 701



The End

Thank you for attending this course &
Best wishes for the rest of your Tripos!

- Don't forget to visit the online feedback page!
- Please send comments on the slides to:
  `tms41@cam.ac.uk`