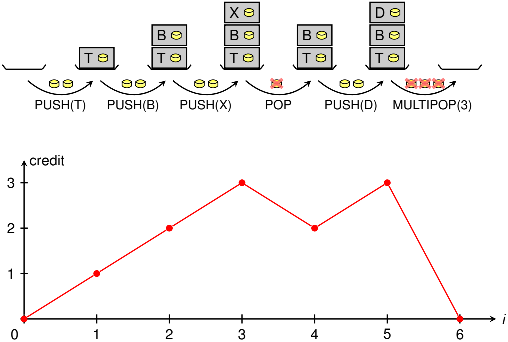


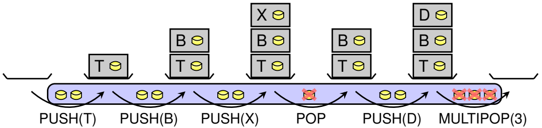
5.1: Amortized Analysis

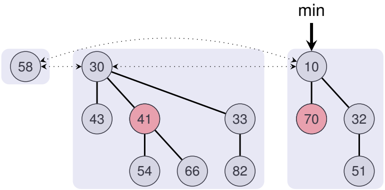
Frank Stajano Thomas Sauerwald

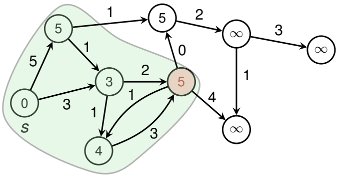
Lent 2016

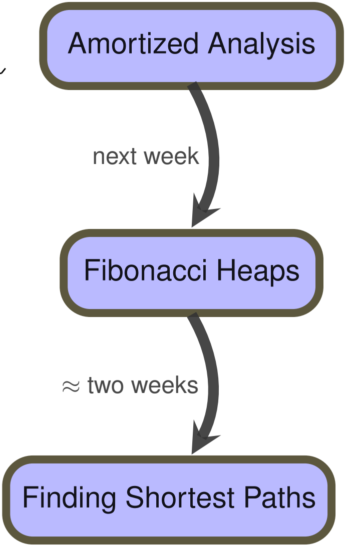
UNIVERSITY OF CAMBRIDGE

Use of Amortized Analysis









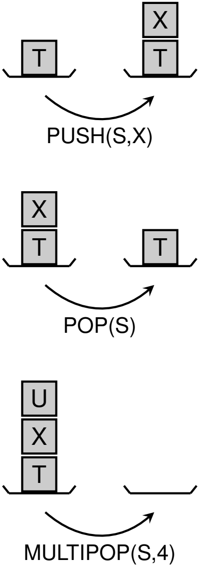
Motivating Example: Stack

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
```

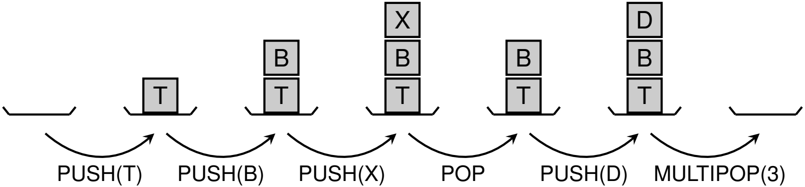


What is the largest possible cost of a sequence of n stack operations (starting from an empty stack)?

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



A new Analysis Tool: Amortized Analysis

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

Amortized Analysis

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

This is **not** average case analysis!

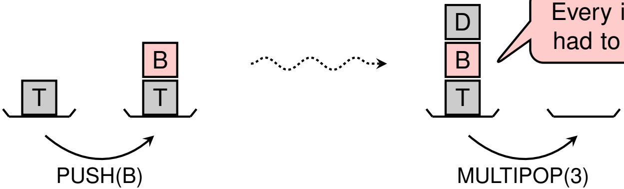
Aggregate Analysis

- 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

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!

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

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

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

Second Technique: Potential Method

Potential Method

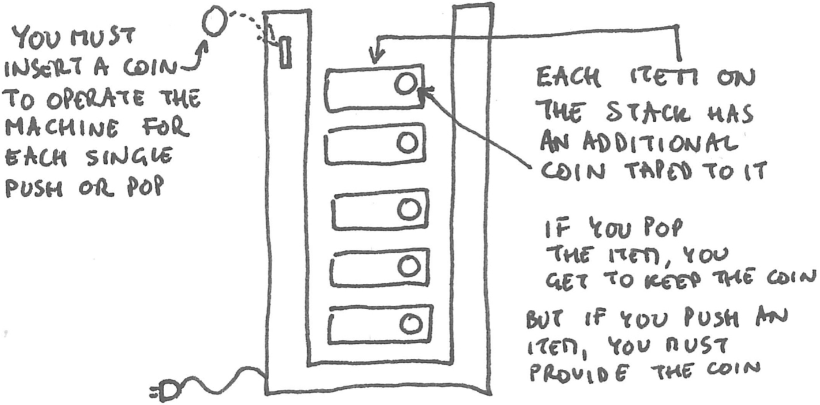
- allow different amortized costs

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

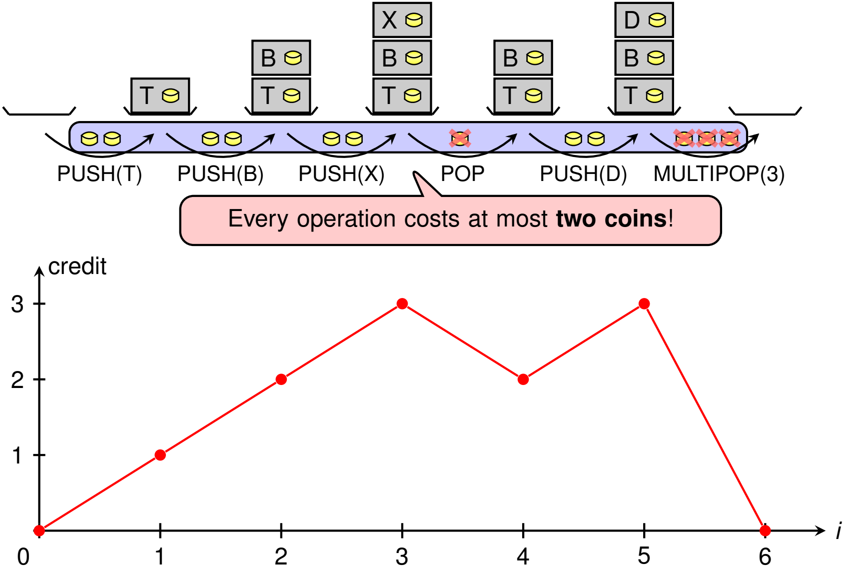
Potential of a data structure can be also thought of as

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

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



Stack and Coins



Potential Method in Detail

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

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

Function that maps states of the data structure to some value

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

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

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

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

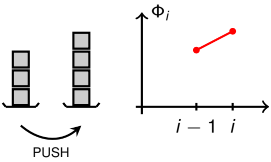
If $\Phi_n \geq 0$ for all n, sum of amortized costs is an upper bound for the sum of actual costs!

Stack: Analysis via Potential Method

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: $\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



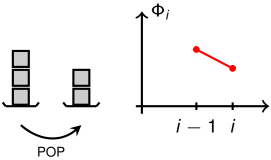
Amortized Cost $\leq 2 \Rightarrow T(n) \leq 2n$

POP

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

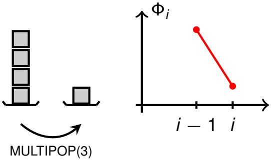
$n/2$ PUSH, $n/2$ POP $\Rightarrow T(n) \leq n$

Stack is non-empty!



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 $2^k - 1$
- only operation: **`INC`**
  - increases the counter by one
  - total cost: ~~$\leq k$~~
    number of flips (smallest index of a zero)

```
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

↓ 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!)**

Incrementing a Binary Counter ($k = 8$)

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

- 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 $\frac{T(n)}{n} \leq 2$.

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

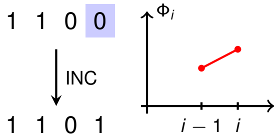
Binary Counter: Analysis via Potential Function

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

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

Increment without Carry-Over

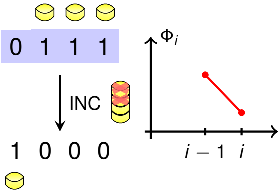
- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c_i} = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



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

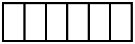
Amortized Analysis

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

E.g. by bounding the number of expensive operations

Aggregate Analysis

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

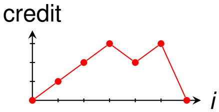
$T(n)$ 

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

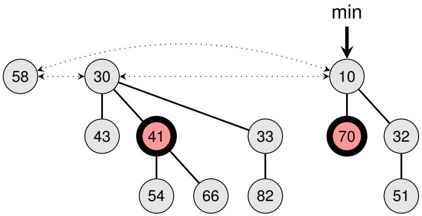
$T(n)$ 



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	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 **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

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

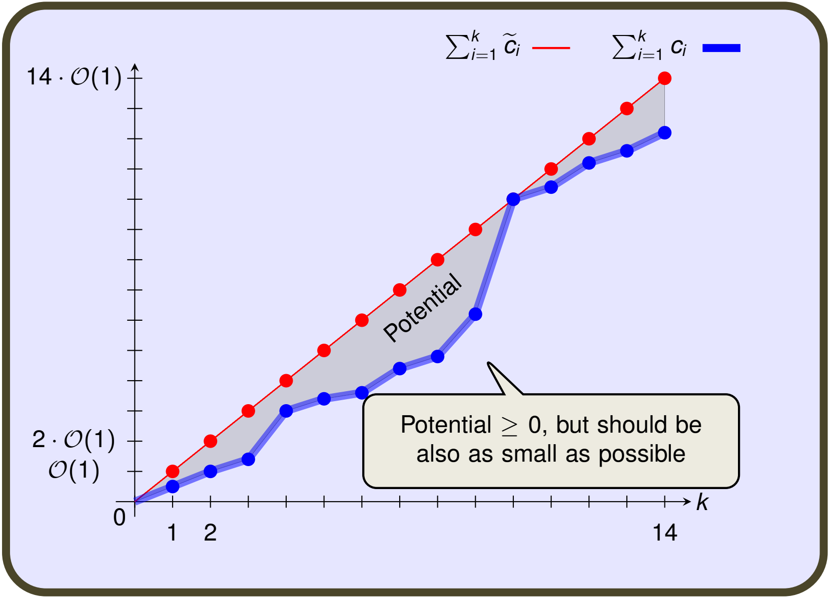
$\Rightarrow \sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$

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

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

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

Actual vs. Amortized Cost



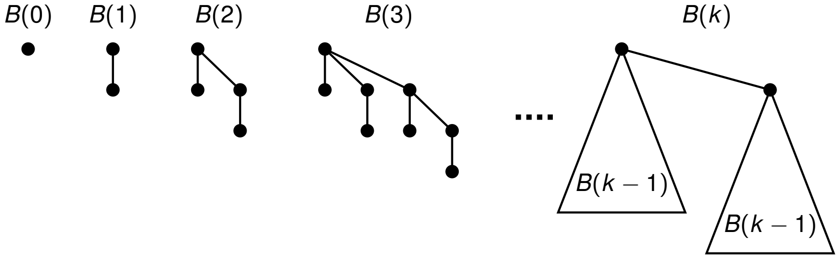
Outline

Structure

Operations

Reminder: Binomial Heaps

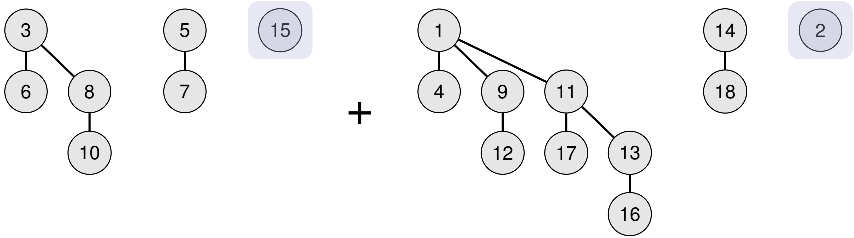
Binomial Trees

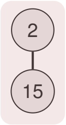


Binomial Heaps

- 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

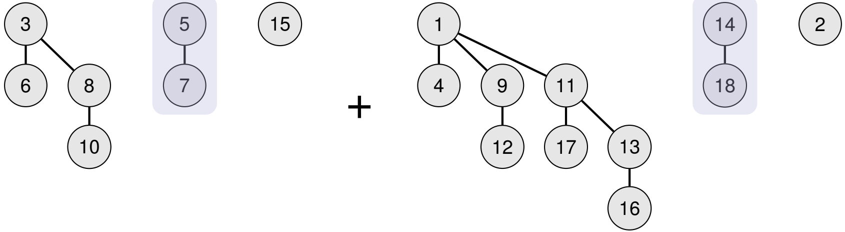
Merging two Binomial Heaps (1/7)

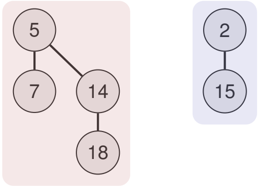




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

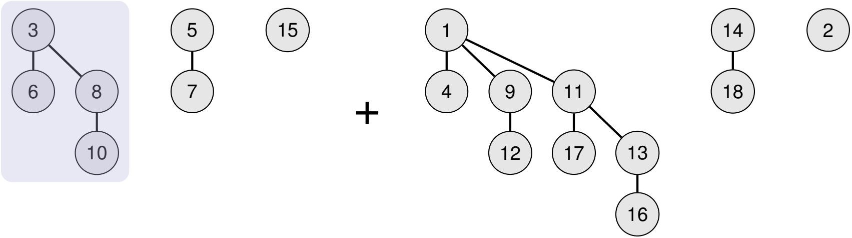
Merging two Binomial Heaps (2/7)

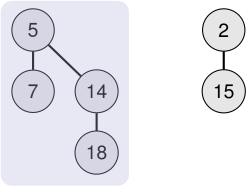




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

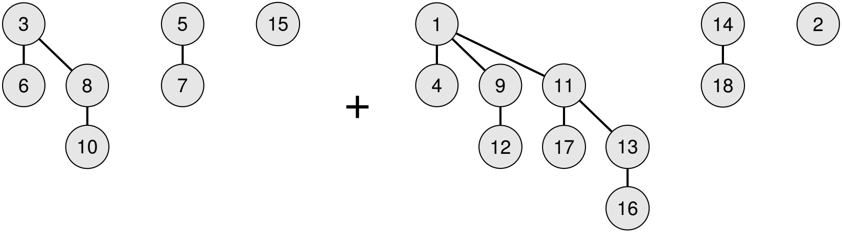
Merging two Binomial Heaps (3/7)

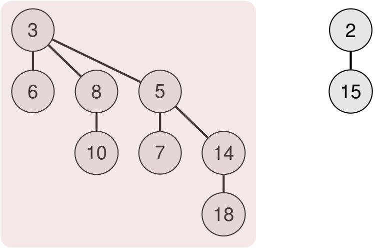




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

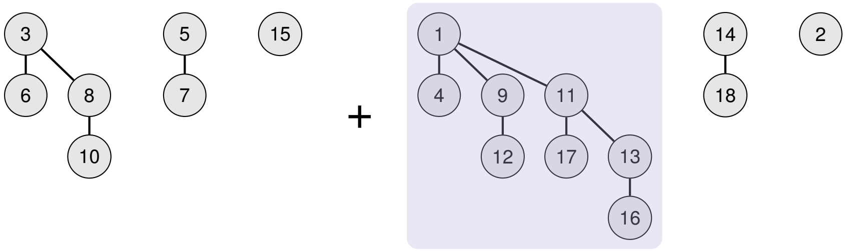
Merging two Binomial Heaps (4/7)

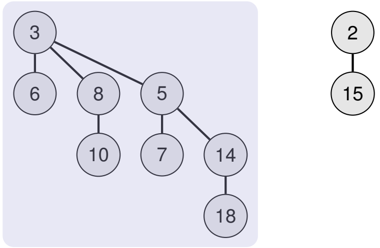




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

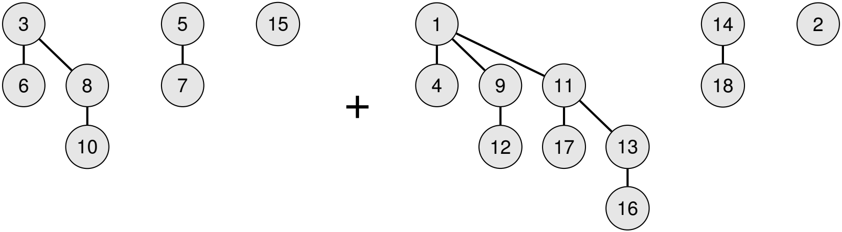
Merging two Binomial Heaps (5/7)

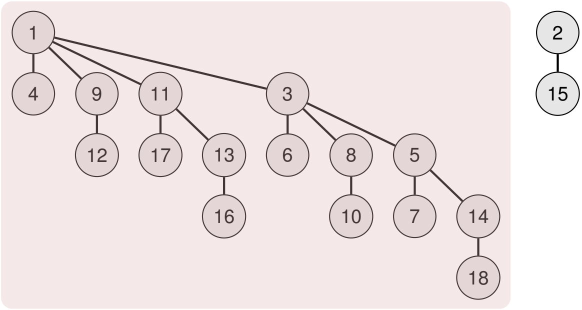




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

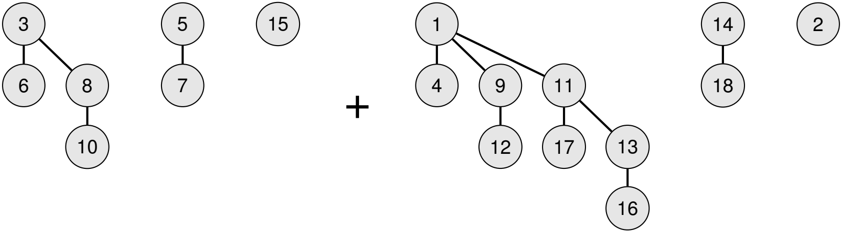
Merging two Binomial Heaps (6/7)

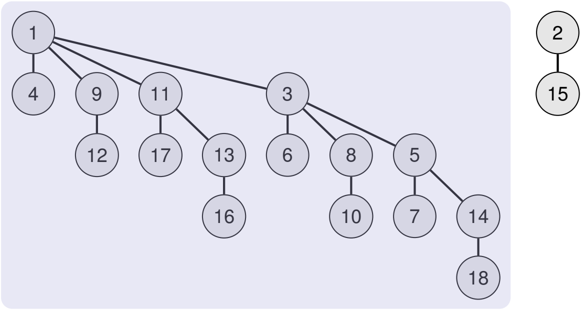




0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

Merging two Binomial Heaps (7/7)



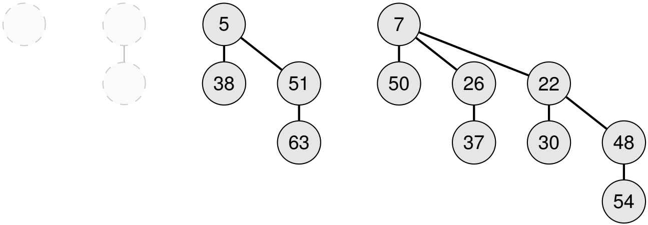


0	0	1	1	1	= 7
0	1	0	1	1	= 11
1	1	1	1		
1	0	0	1	0	= 18

Binomial Heap vs. Fibonacci Heap: Structure

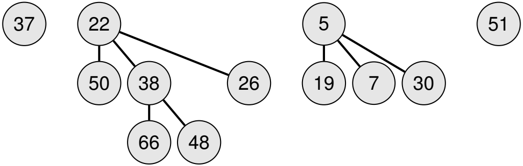
Binomial Heap:

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



Fibonacci Heap:

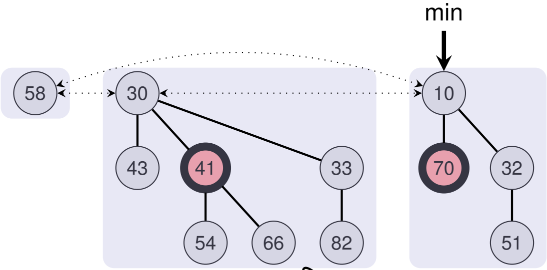
- forest of MIN-HEAPs
- lazily defer tidying up; do it on-the-fly when search for the MIN



Structure of Fibonacci Heaps

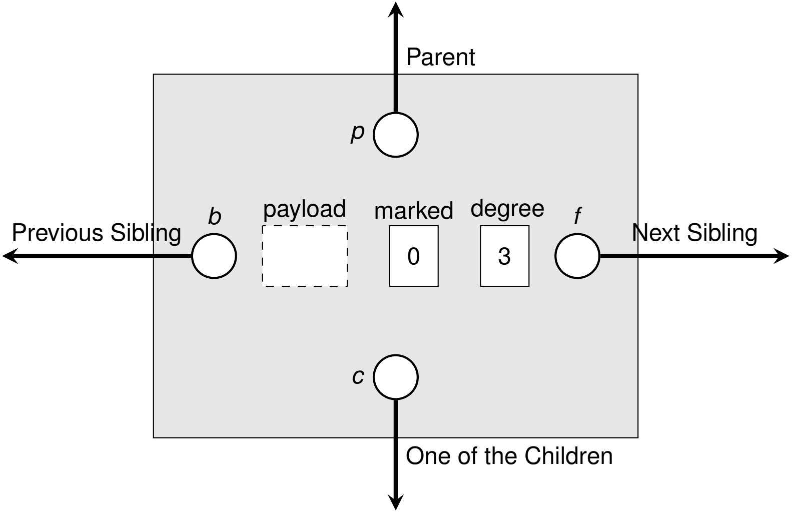
Fibonacci Heap

- Forest of MIN-HEAPs
- 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

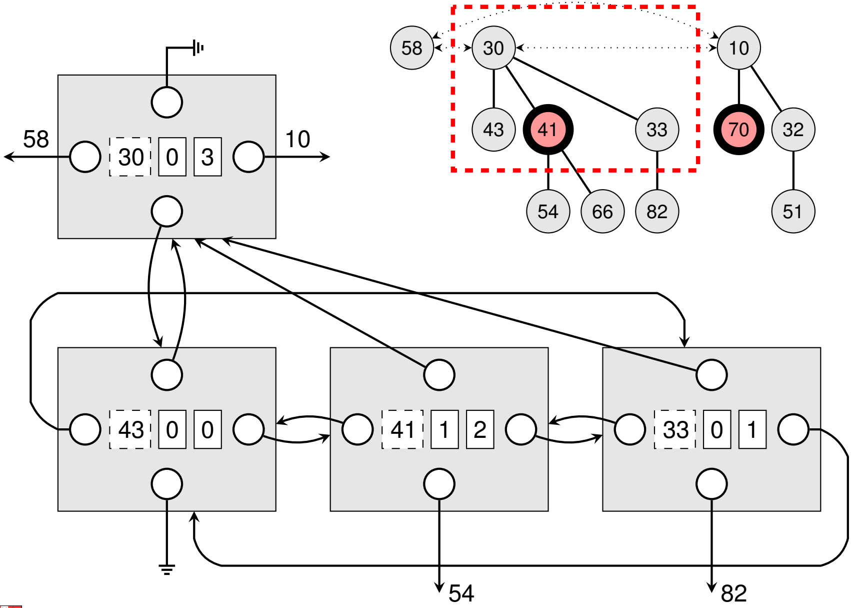


How do we implement a Fibonacci Heap?

A single Node



Magnifying a Four-Node Portion



Outline

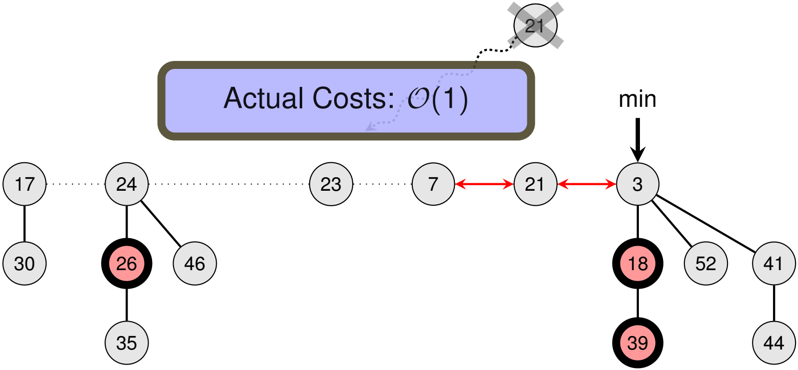
Structure

Operations

Fibonacci Heap: INSERT

INSERT

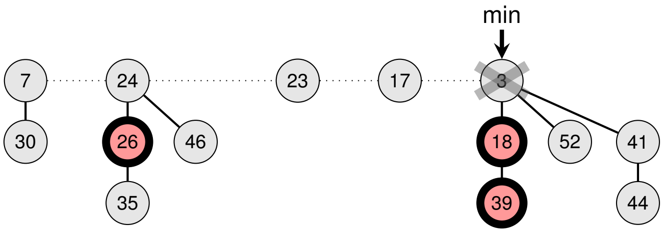
- Create a singleton tree
- Add to root list and update min-pointer (if necessary)



Fibonacci Heap: EXTRACT-MIN (1/11)

EXTRACT-MIN

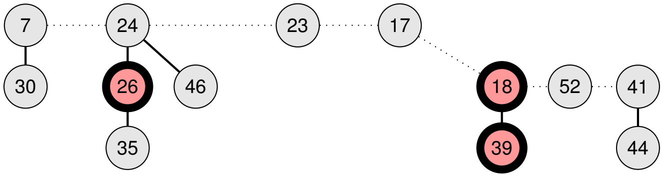
- Delete min



Fibonacci Heap: EXTRACT-MIN (2/11)

EXTRACT-MIN

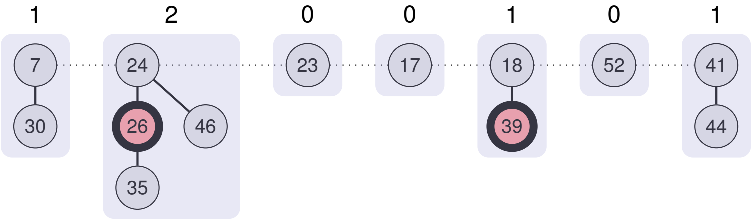
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: EXTRACT-MIN (3/11)

EXTRACT-MIN

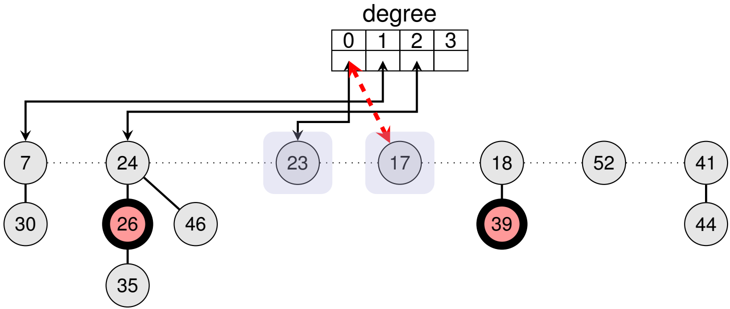
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN (4/11)

EXTRACT-MIN

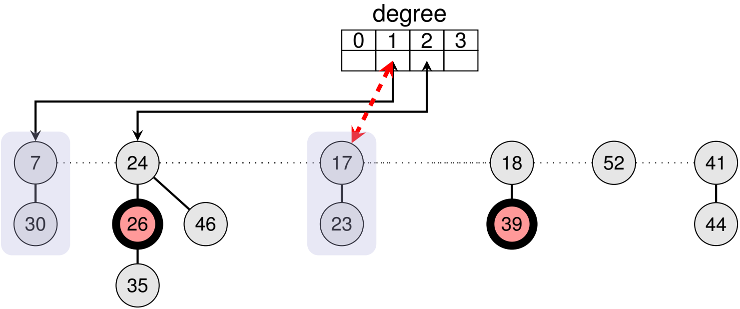
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min (5/11)

Extract-Min

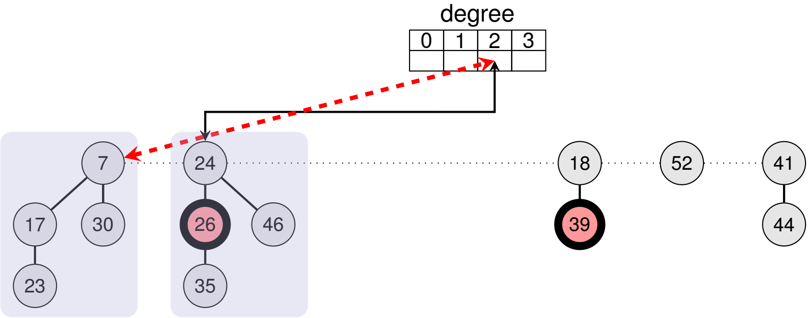
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN (6/11)

EXTRACT-MIN

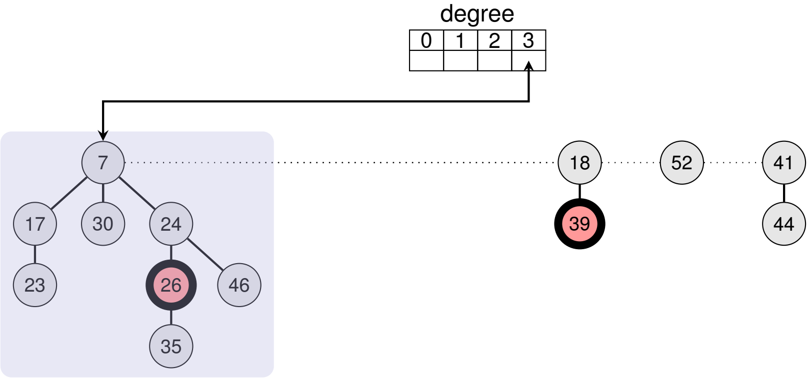
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN (7/11)

EXTRACT-MIN

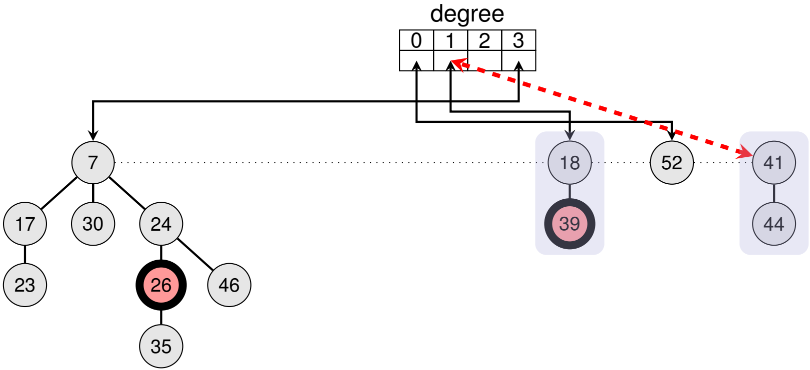
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN (8/11)

EXTRACT-MIN

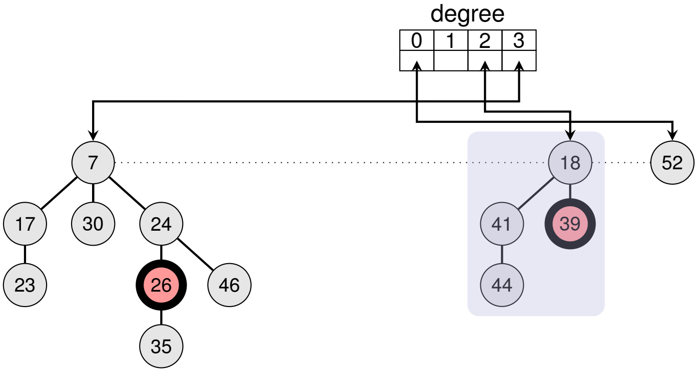
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN (9/11)

EXTRACT-MIN

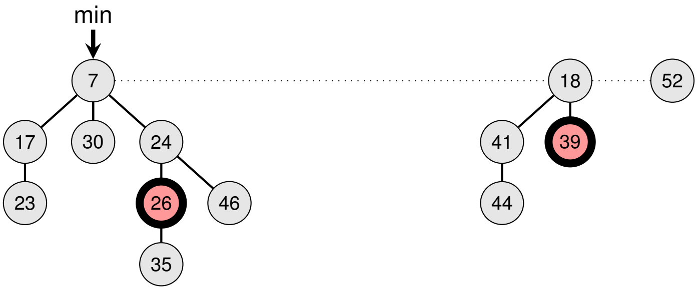
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓



Fibonacci Heap: EXTRACT-MIN (10/11)

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 (11/11)

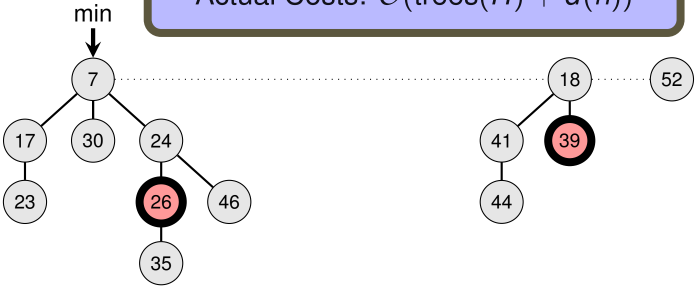
EXTRACT-MIN

- 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

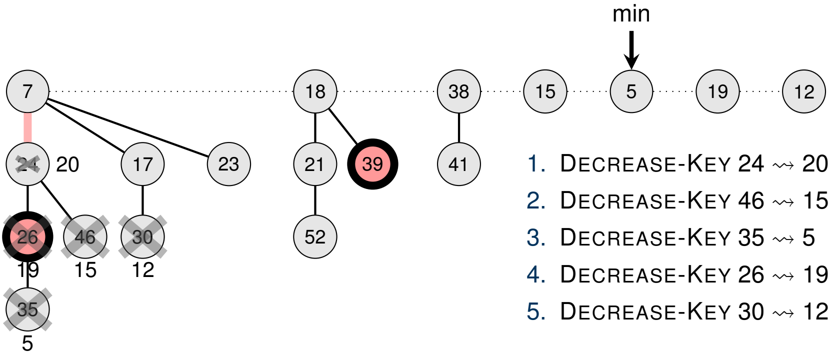
Actual Costs: $\mathcal{O}(\text{trees}(H) + d(n))$



Fibonacci Heap: DECREASE-KEY (First Try) (1/3)

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).

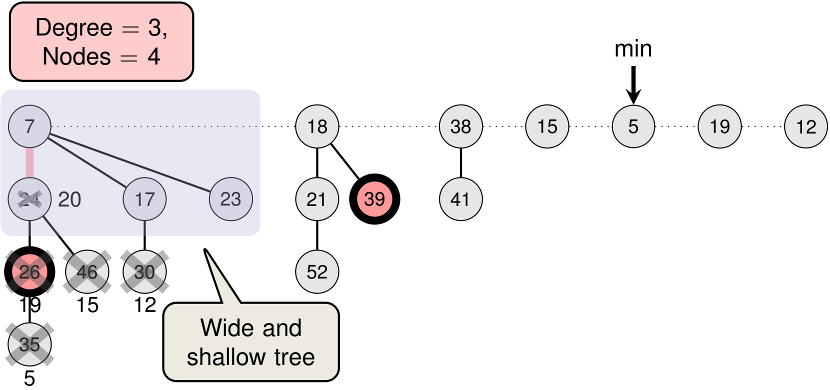


1. DECREASE-KEY 24 ⇝ 20
2. DECREASE-KEY 46 ⇝ 15
3. DECREASE-KEY 35 ⇝ 5
4. DECREASE-KEY 26 ⇝ 19
5. DECREASE-KEY 30 ⇝ 12

Fibonacci Heap: DECREASE-KEY (First Try) (2/3)

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).

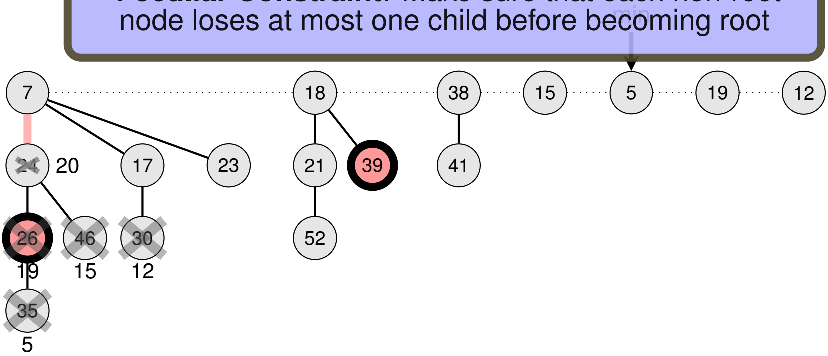


Fibonacci Heap: DECREASE-KEY (First Try) (3/3)

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

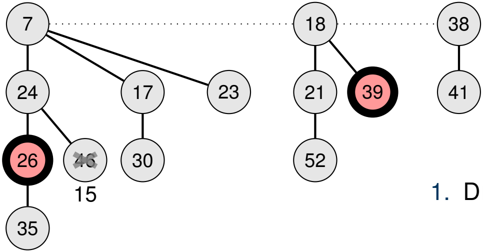


Fibonacci Heap: DECREASE-KEY (1/7)

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

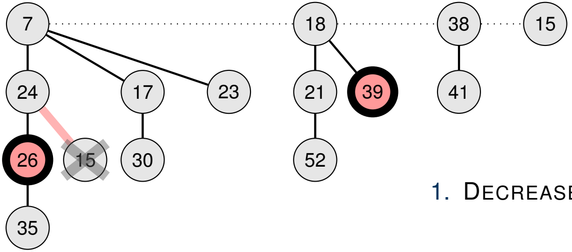
Fibonacci Heap: DECREASE-KEY (2/7)

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 ✓

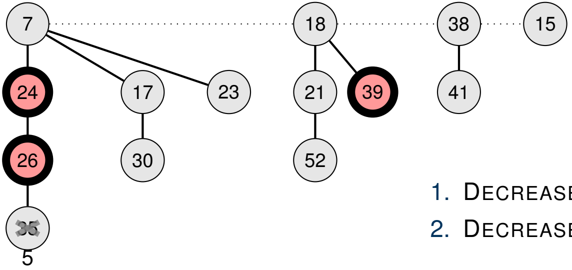
Fibonacci Heap: DECREASE-KEY (3/7)

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

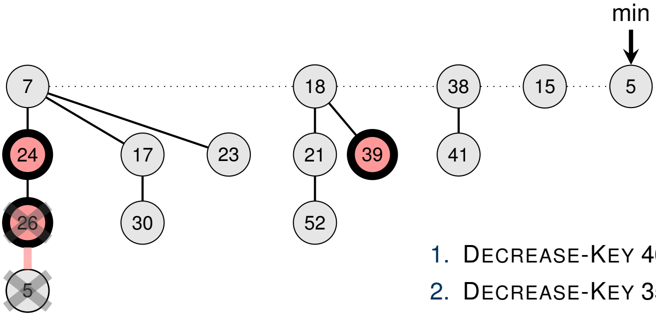
Fibonacci Heap: DECREASE-KEY (4/7)

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

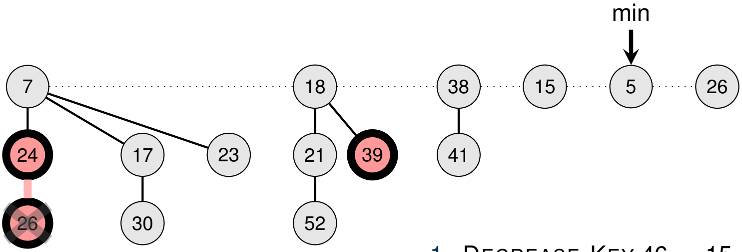
Fibonacci Heap: DECREASE-KEY (5/7)

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

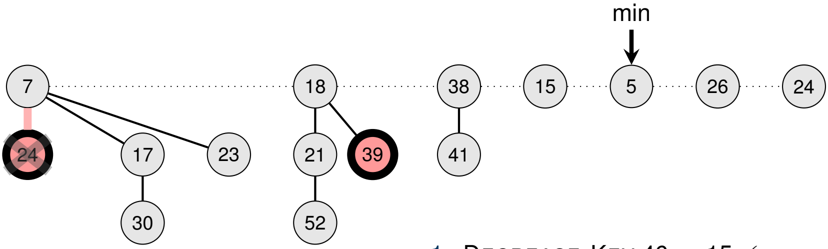
Fibonacci Heap: DECREASE-KEY (6/7)

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 (7/7)

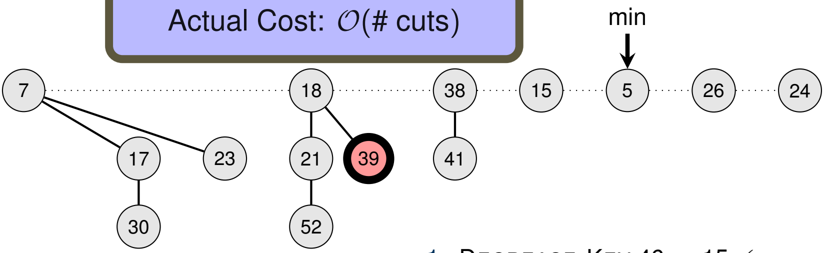
DECREASE-KEY of node x

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

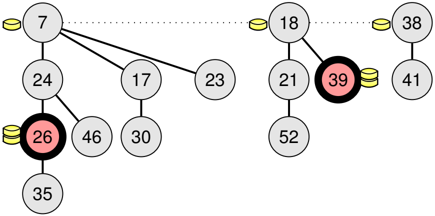
$\Rightarrow$ Cut tree rooted at x, unmark x, meld into root list and:

- 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

Glimpse at the Analysis

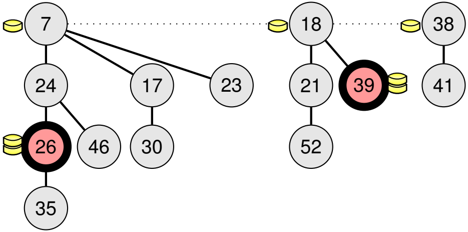
Amortized Analysis

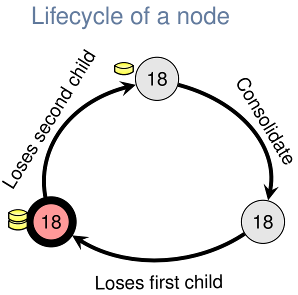
Bounding the Maximum Degree

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 $\mathcal{O}(1)$?

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





Outline

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.

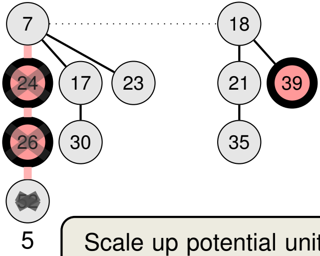
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

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

Change in Potential

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

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



Scale up potential units

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

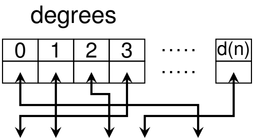
- Extract-Min: $\mathcal{O}(\text{trees}(H) + d(n))$

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

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

Glimpse at the Analysis

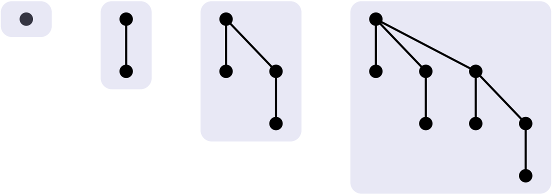
Amortized Analysis

Bounding the Maximum Degree

Bounding the Maximum Degree

Binomial Heap

Every tree is a binomial tree $\Rightarrow d(n) \leq \log_2 n$.



$d = 3, n = 2^3$

Fibonacci Heap

Not all trees are binomial trees, but still $d(n) \leq \log_\varphi n$, where $\varphi \approx 1.62$.

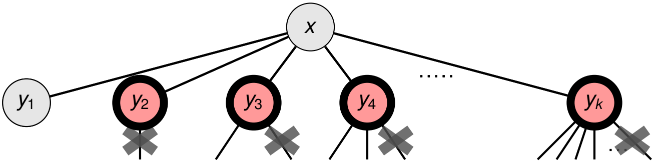
Lower Bounding Degrees of Children

We will prove a stronger statement:
Any tree with degree k contains at least φ^k nodes.

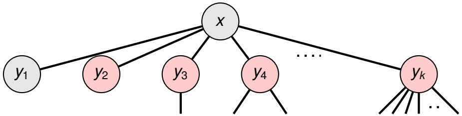
$$d(n) \leq \log_\varphi n$$

- 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



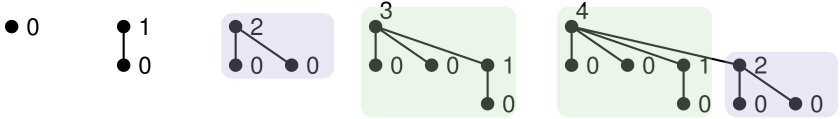
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

Let $N(k)$ be the minimum possible number of nodes of a subtree rooted at a node of degree k.

$$N(k) \ = \ F(k+2)?$$

$N(0) = 1$ $N(1) = 2$ $N(2) = 3$ $N(3) = 5$ $N(4) = 8 = 5 + 3$

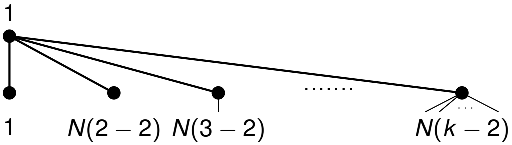


From Minimum Subtree Sizes to Fibonacci Numbers

$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

$$N(k) \;=\; F(k+2)?$$

$N(k) =$



$$\begin{aligned} N(k) &= 1 + 1 + N(2-2) + N(3-2) + \cdots + N(k-2) \\ &= 1 + 1 + \sum_{\ell=0}^{k-2} N(\ell) \\ &= 1 + 1 + \sum_{\ell=0}^{k-3} N(\ell) + N(k-2) \\ &= N(k-1) + N(k-2) \\ &= F(k+1) + F(k) = F(k+2) \qquad \square \end{aligned}$$

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 $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$.

$\varphi^2 = \varphi + 1$

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 ~~$\mathcal{O}(d(n))$~~ $\mathcal{O}(\log n)$
- DECREASE-KEY amortized cost $\mathcal{O}(1)$

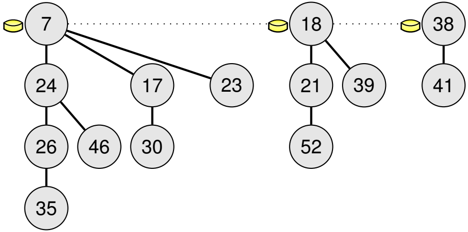
$$n \geq N(k) = F(k+2) \geq \varphi^k$$

$$\Rightarrow \qquad \log_\varphi n \geq k$$

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

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	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

DELETE = DECREASE-KEY + EXTRACT-MIN

EXTRACT-MIN = MIN + DELETE

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

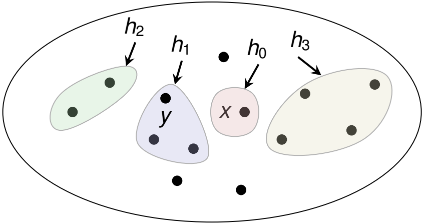
$\Rightarrow$ less efficient than the original Fibonacci heap

$\Rightarrow$ 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	-	$\mathcal{O}(1)$

all this requires key values to be in a universe of size u!



5.3: Disjoint Sets

Frank Stajano Thomas Sauerwald

Lent 2016

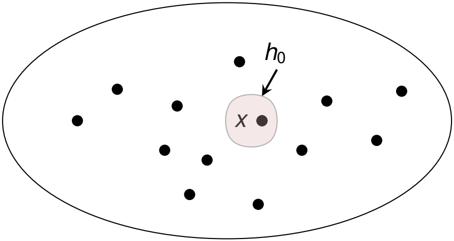
UNIVERSITY OF CAMBRIDGE

Disjoint Sets (aka Union Find) (1/5)

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)`**

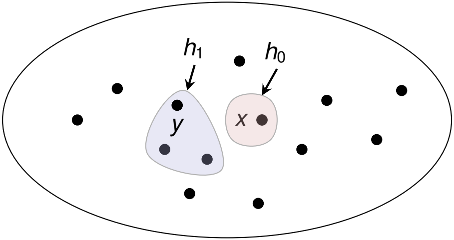


Disjoint Sets (aka Union Find) (2/5)

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)`

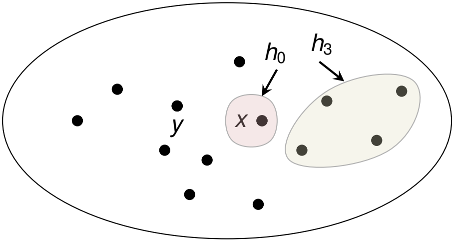


Disjoint Sets (aka Union Find) (3/5)

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**`)`**

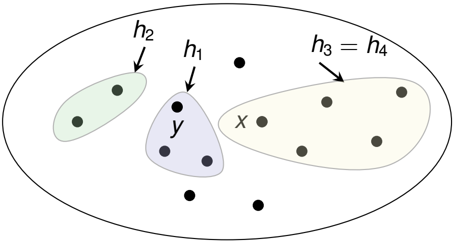


Disjoint Sets (aka Union Find) (4/5)

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`)`

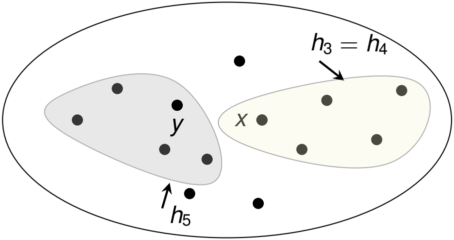


Disjoint Sets (aka Union Find) (5/5)

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)

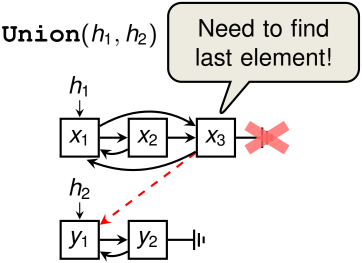


First Attempt: List Implementation (1/2)

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time



First Attempt: List Implementation (2/2)

UNION-Operation

- Add extra pointer to the last element in each list

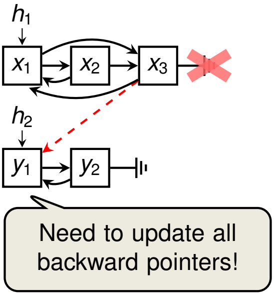
⇒ ~~UNION takes constant time~~

FINDSET-Operation

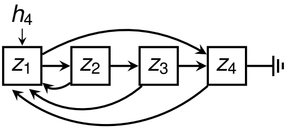
- Add backward pointer to the list head from everywhere

⇒ FINDSET takes constant time

Union(h_1, h_2)



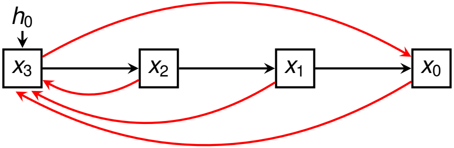
FindSet(z_3)



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



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
- Append shorter list to the longer list (breaking ties arbitrarily)

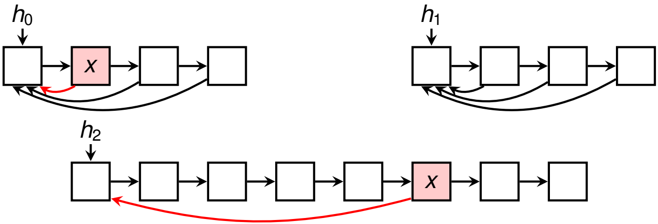
can be done easily without significant overhead

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.

Can we improve on this further?

Proof:

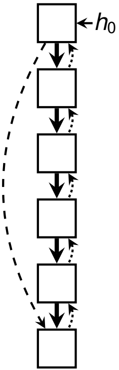
- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

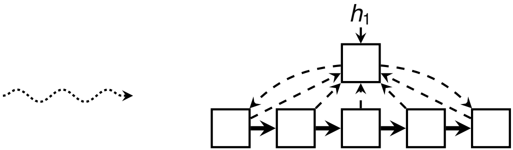
$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

□

How to Improve?





Basic Idea: Update Backward Pointers only during FIND-SET

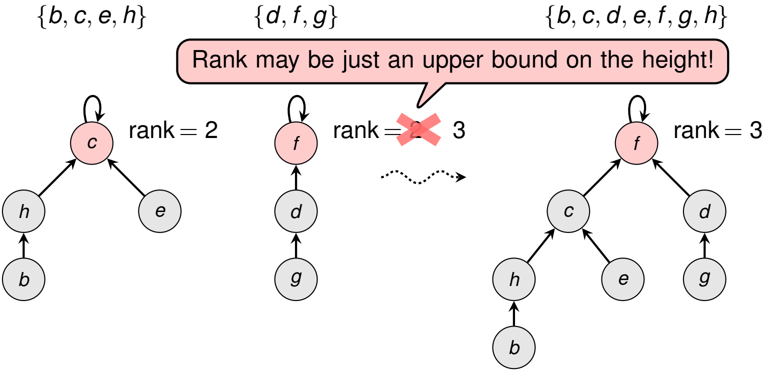
Doubly-Linked List

- 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



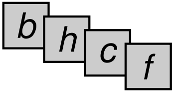
Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

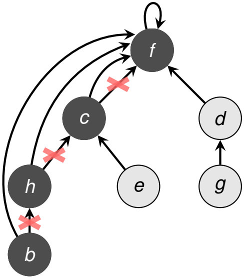
Append tree of smaller height $\rightsquigarrow$ Union by Rank

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

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

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

$\log^*(n)$, **the iterated logarithm**, satifies $\alpha(n) \leq \log^*(n)$, but still $\log^*(10^{80}) = 5$.

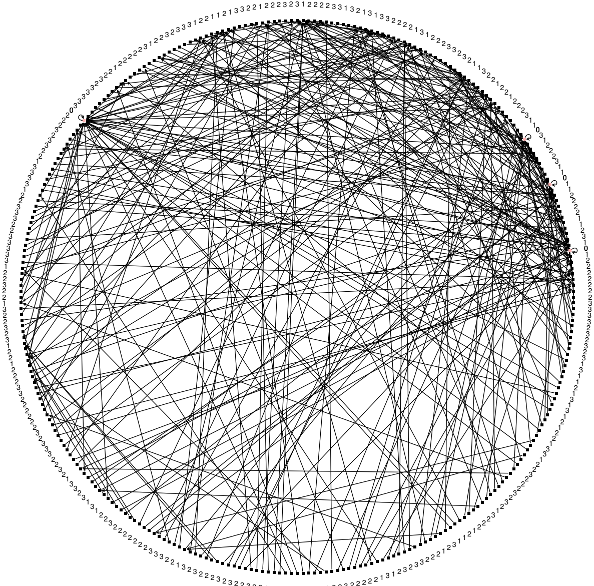
More than the number of atoms in the universe!

Simulating the Effects of Union by Rank and Path Compression

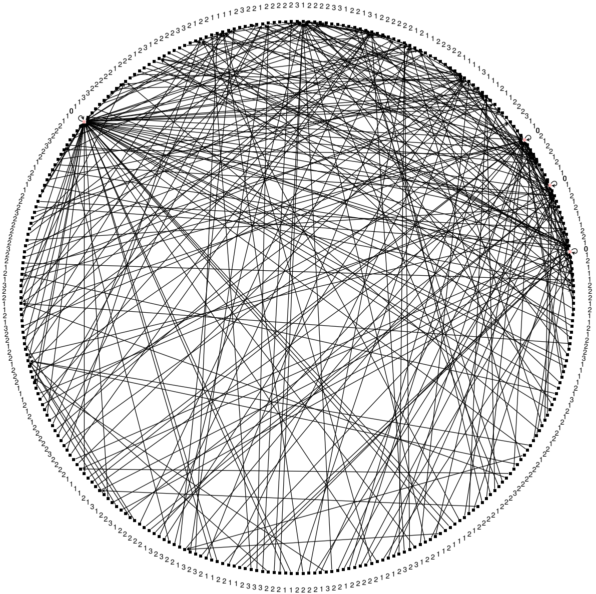
Experimental Setup

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 UNION(FINDSET(i), FINDSET(r))
3. Perform $j \in \{0, 100, 200, 300, 600, 900\}$ many additional FINDSET(r), where $1 \leq r \leq 300$ is random

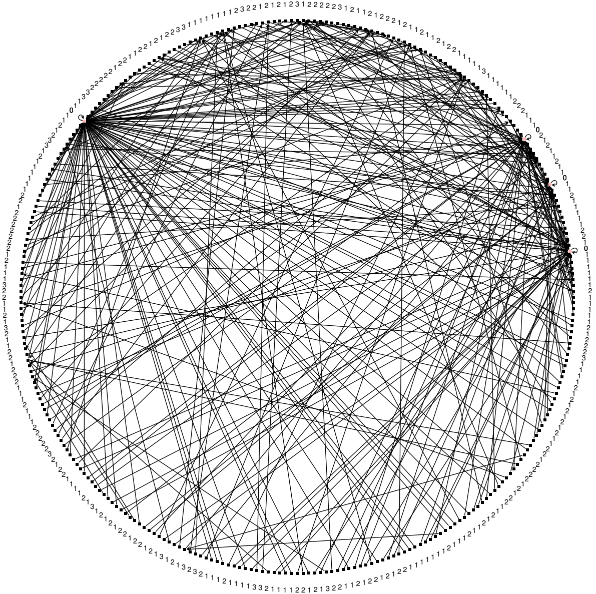
Union by Rank without Path Compression



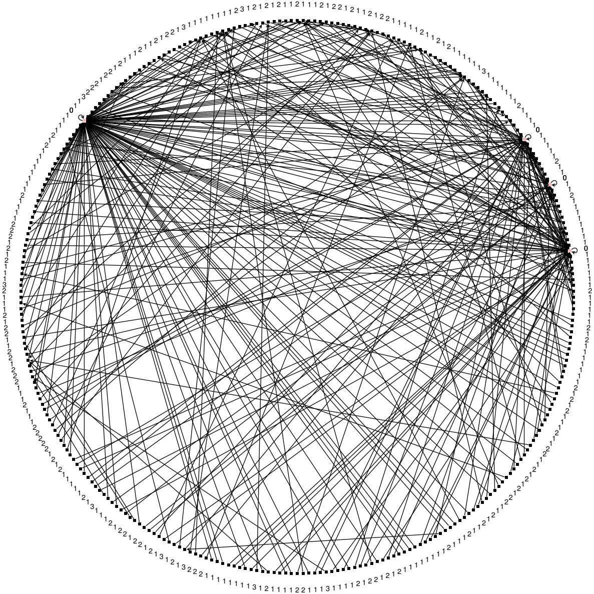
Union by Rank with Path Compression



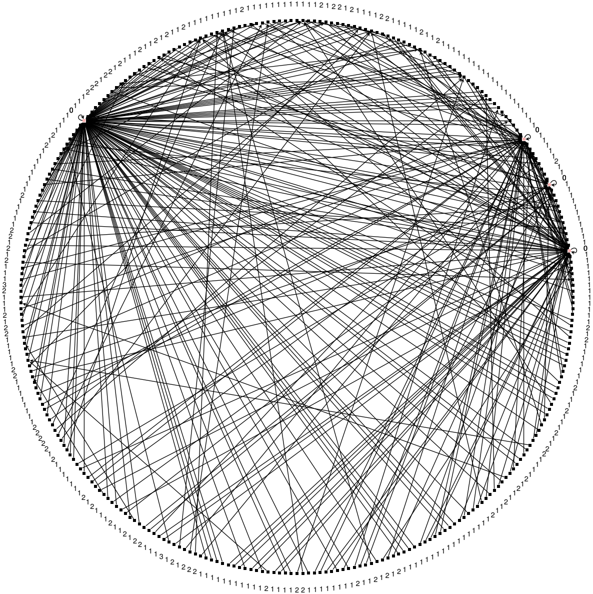
Union by Rank with Path Compression (100 additional FINDSET)



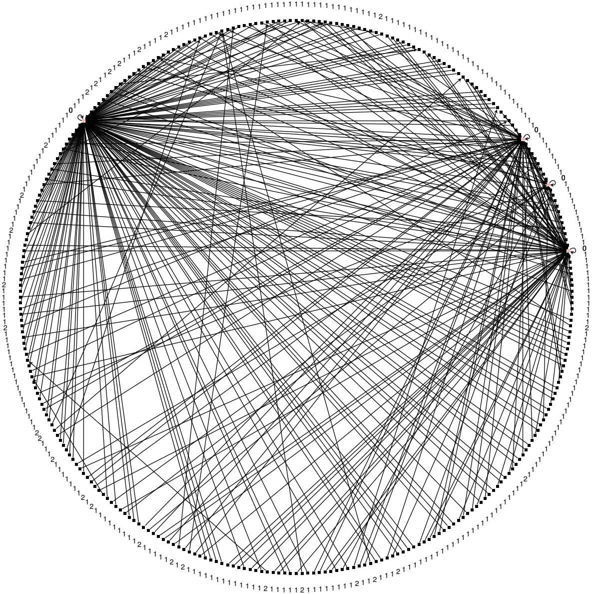
Union by Rank with Path Compression (200 additional FINDSET)



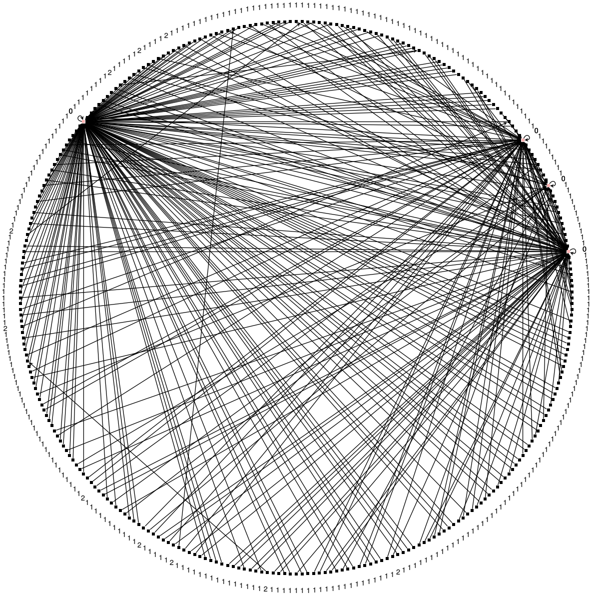
Union by Rank with Path Compression (300 additional FINDSET)

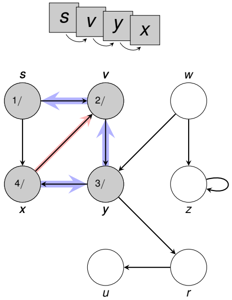


Union by Rank with Path Compression (600 additional FINDSET)



Union by Rank with Path Compression (900 additional FINDSET)





6.1 & 6.2: Graph Searching

Frank Stajano

Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

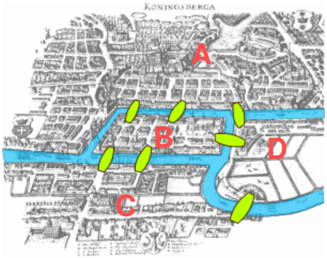
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

Topological Sort

Origin of Graph Theory

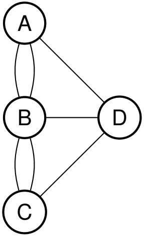


Source: Wikipedia

Seven Bridges at Königsberg 1737

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**?
$\rightsquigarrow$ 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)

Undirected Graph

A graph $G = (V, E)$ consists of:

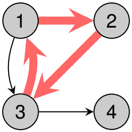
- V: the set of vertices
- E: the set of (undirected) edges

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

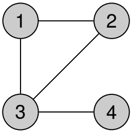
Path $p = (1, 2, 3)$, which is a cycle

G is not connected



$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

G is connected



$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

Later: edge-weighted graphs $G = (V, E, w)$

Representations of Directed and Undirected Graphs

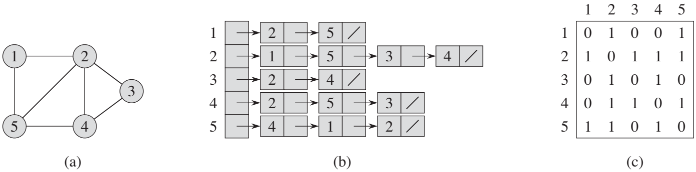


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Most times we will use the adjacency-list representation!

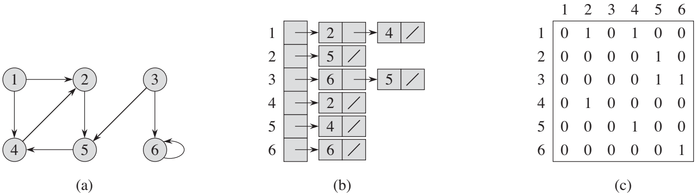
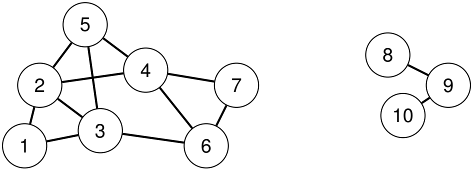


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

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

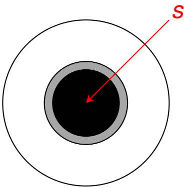
Introduction to Graphs and Graph Searching

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
- 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"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:

  White = Unvisited

  Grey = Visited, but not all neighbors

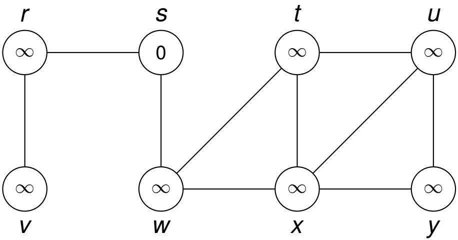
  Black = Visited and all neighbors
- Runtime $O(V + E)$

Assuming that all executions of the FOR-loop for u takes $O(|u.adj|)$ (**adjacency list model!**)

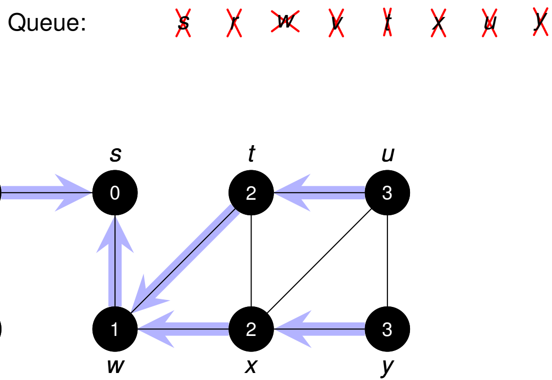
$$\sum_{u \in V} |u.adj| = 2|E|$$

Execution of BFS (Figure 22.3) (1/2)

Queue:



Execution of BFS (Figure 22.3) (2/2)



Outline

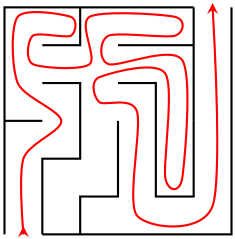
Introduction to Graphs and Graph Searching

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
- 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
```

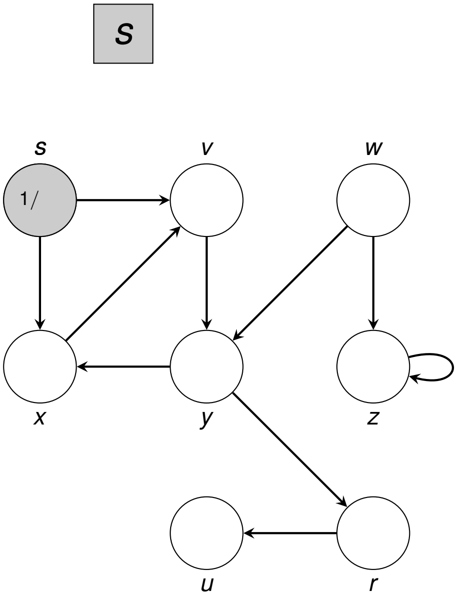
- We always go deeper before visiting other neighbors
- Discovery and Finish times, *.d* and *.f*
- Vertex Colours:

  White = Unvisited

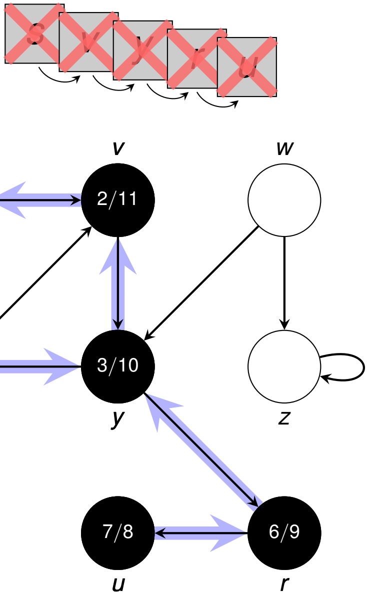
  Grey = Visited, but not all neighbors

  Black = Visited and all neighbors
- Runtime $O(V + E)$

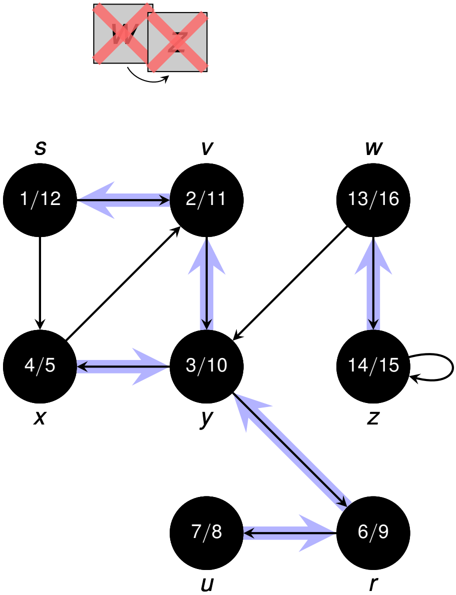
Execution of DFS (1/3)



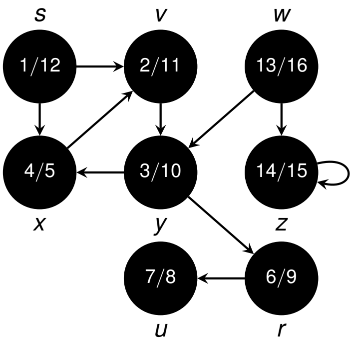
Execution of DFS (2/3)

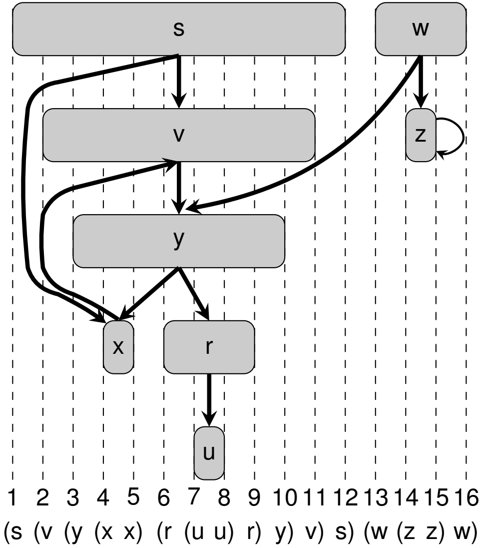


Execution of DFS (3/3)



Paranthesis Theorem (Theorem 22.7)





Outline

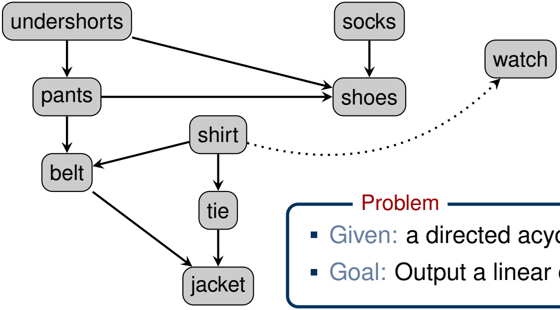
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

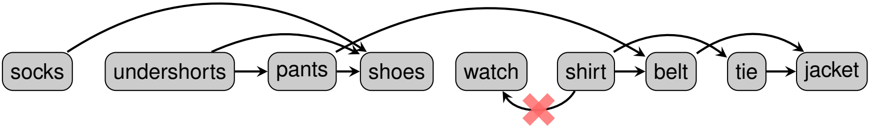
Topological Sort

Topological Sort

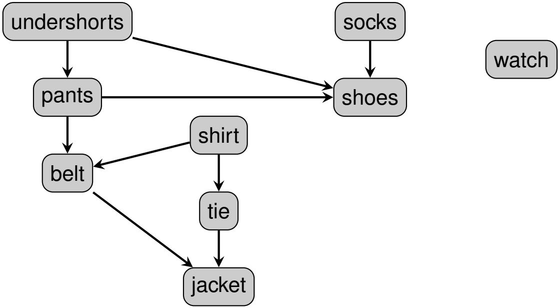


Problem

- Given: a directed acyclic graph (DAG)
- Goal: Output a linear ordering of all vertices



Solving Topological Sort



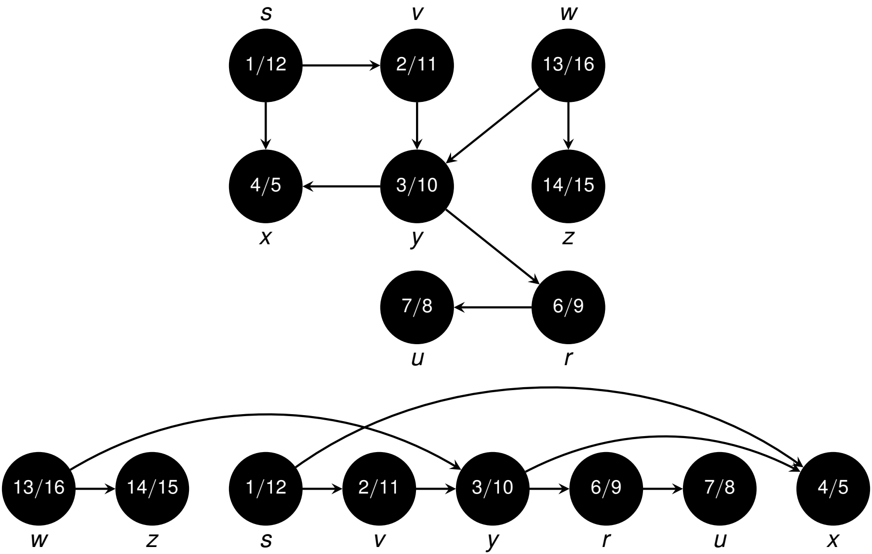
Knuth's Algorithm (1968)

- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

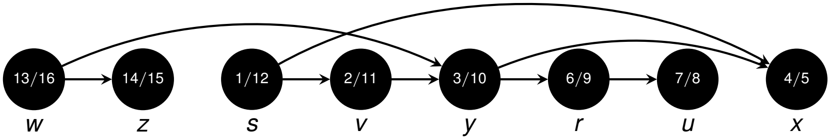
Runtime $O(V + E)$

Don't need to sort the vertices – use DFS directly!

Execution of Knuth's Algorithm



Correctness of Topological Sort using DFS (1/3)



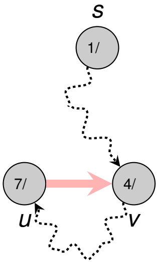
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

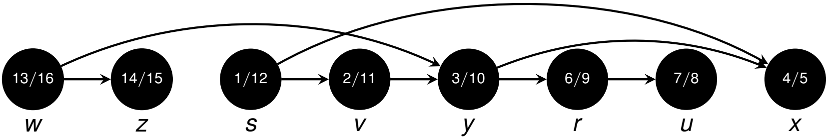
Proof:

- 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 (2/3)



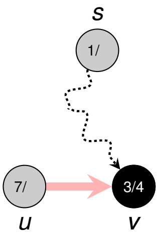
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

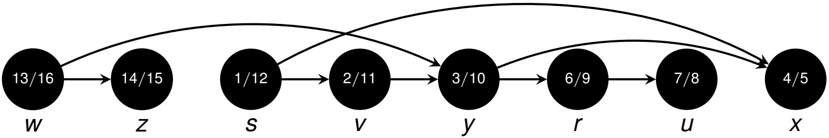
Proof:

- 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 (3/3)



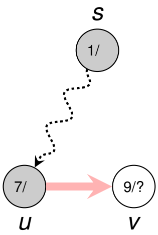
Theorem 22.12

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 $v.f < u.f$.*

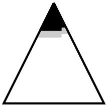


$\Rightarrow$ In all cases $v.f < u.f$, so v appears after u. $\square$

Summary of Graph Searching

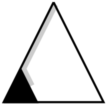
Breadth-First-Search

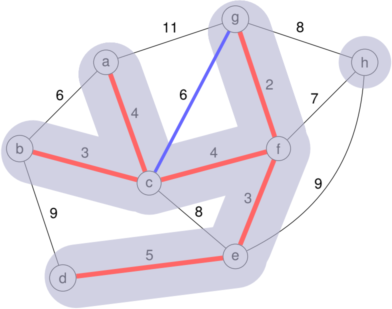
- vertices are processed by a queue
- computes distances and shortest paths
  $\rightsquigarrow$ similar idea used later in Prim's and Dijkstra's algorithm
- Runtime $\mathcal{O}(V + E)$



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

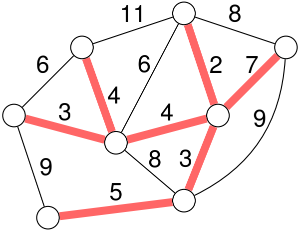
UNIVERSITY OF CAMBRIDGE

Minimum Spanning Tree Problem

Minimum Spanning Tree Problem

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

Must be necessarily a tree!



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
```

Definition

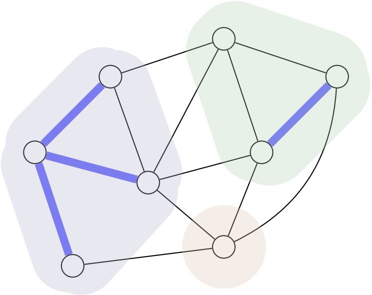
An edge of G is safe if by adding the edge to A, the resulting subgraph is still a subset of a minimum spanning tree.

How to find a safe edge?

Finding safe edges

Definitions

- a cut is a partition of V into at least two disjoint sets
- 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.

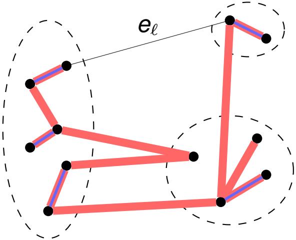
Proof of Theorem (1/3)

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



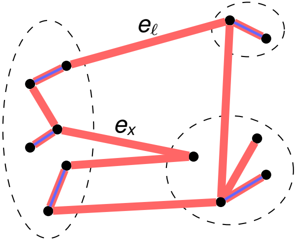
Proof of Theorem (2/3)

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



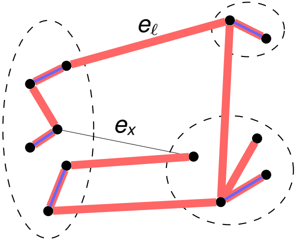
Proof of Theorem (3/3)

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. □

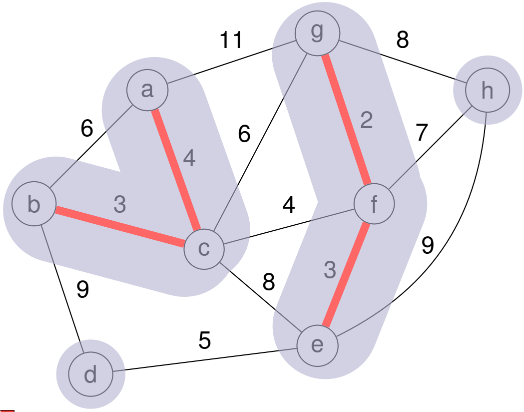


Glimpse at Kruskal's Algorithm (1/2)

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

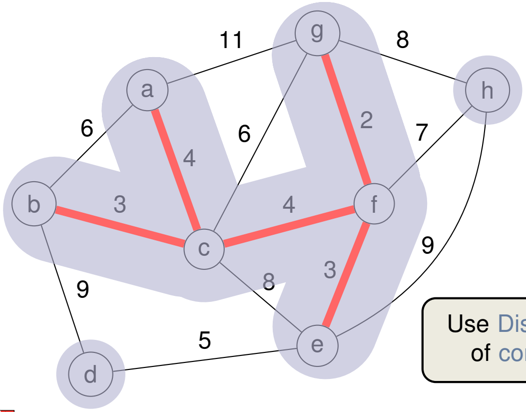


Glimpse at Kruskal's Algorithm (2/2)

Basic Strategy

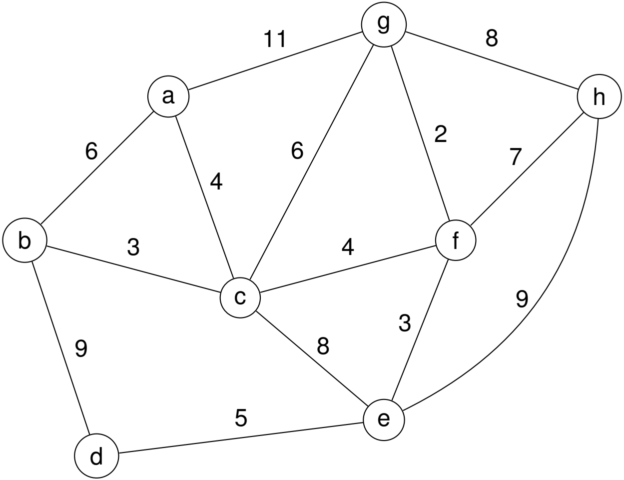
- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle

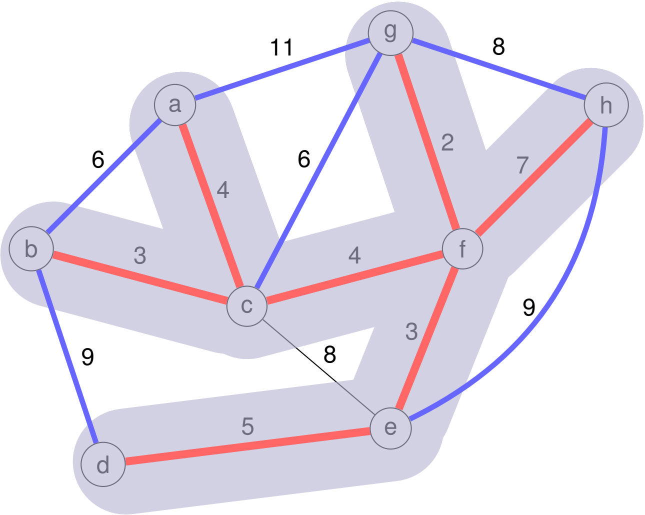


Use Disjoint Sets to keep track of connected components!

Execution of Kruskal's Algorithm (1/2)



Execution of Kruskal's Algorithm (2/2)



Details of Kruskal's Algorithm (1/2)

```
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 (2/2)

```
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 (1/4)

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 (2/4)

Basic Strategy

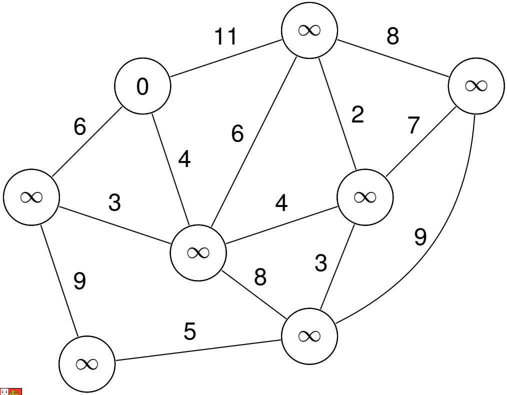
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Implementation

- 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

Basic Strategy

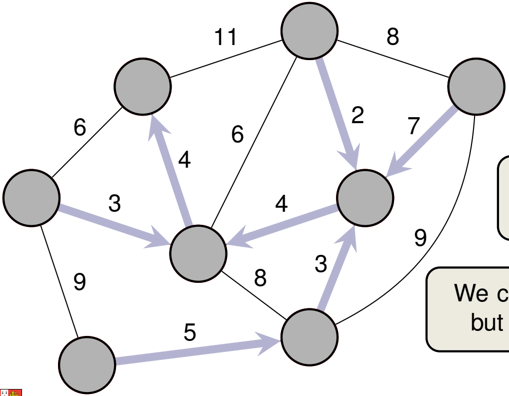
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Prim's Algorithm (4/4)

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



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

```
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: $\mathcal{O}(V \log V + E)$
  Amortized Cost Amortized Cost
- 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

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

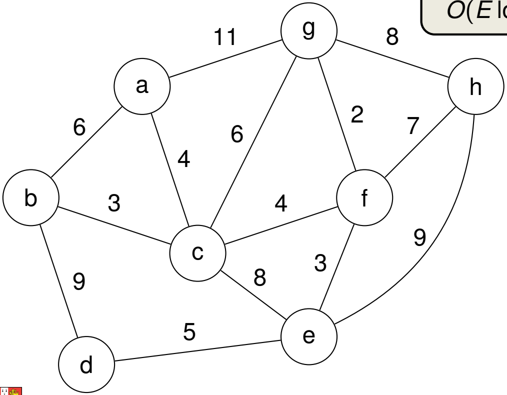
- Gradually extends a tree into a MST by adding incident edges
- invokes Fibonacci heaps (priority queue)
- Runtime $\mathcal{O}(V \log V + E)$

Outlook: Reverse-Delete Algorithm (1/2)

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]

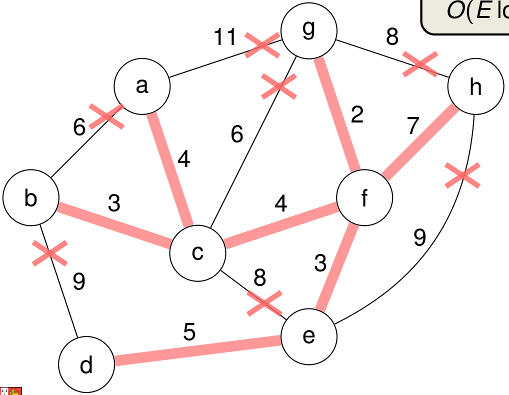


Outlook: Reverse-Delete Algorithm (2/2)

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]



Does a linear-time MST algorithm exist?

Karger, Klein, Tarjan, JACM'1995

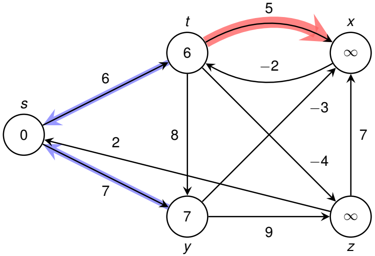
- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

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

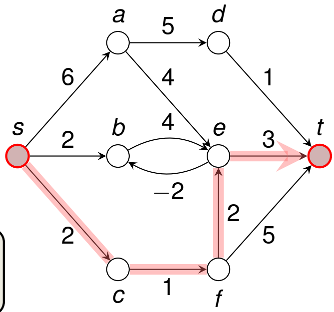
Dijkstra's Algorithm

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.



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

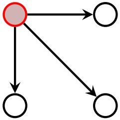
Applications

- Car Navigation, Internet Routing, Arbitrage in Concurrency Exchange

Variants of Shortest Path Problems

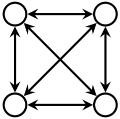
Single-source shortest-paths problem (SSSP)

- Bellman-Ford Algorithm
- Dijsktra Algorithm

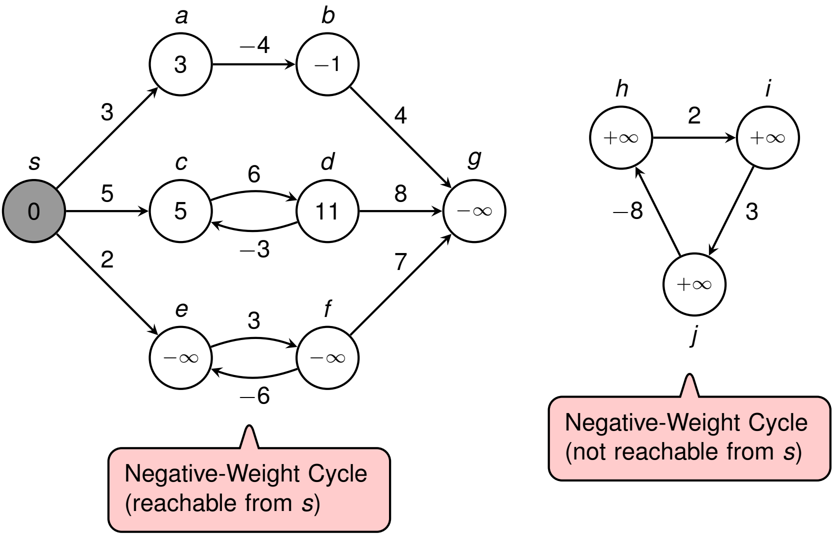


All-pairs shortest-paths problem (APSP)

- Shortest Paths via Matrix Multiplication
- Johnson's Algorithm



Distances and Negative-Weight Cycles (Figure 24.1)



Outline

Introduction

Bellman-Ford Algorithm

Dijkstra's Algorithm

Relaxing Edges (1/2)

Definition

Fix the source vertex $s \in V$

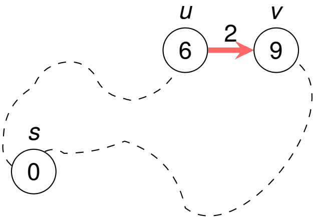
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- 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 (2/2)

Definition

Fix the source vertex $s \in V$

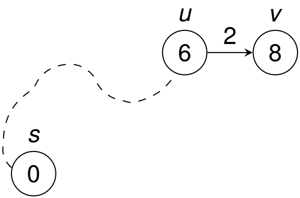
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- 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)

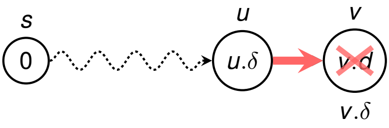
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

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.



$$
\begin{aligned}
v.d &\leq u.d + w(u, v) \\
&= u.\delta + w(u, v) \\
&= v.\delta
\end{aligned}
$$

Since $v.d \geq v.\delta$, we have $v.d = v.\delta$. $\square$

Path-Relaxation Property

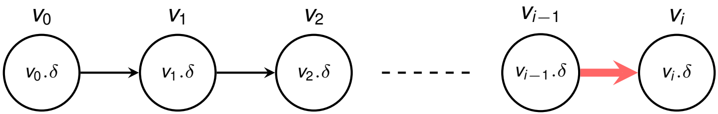
"Propagation": By relaxing proper edges, set of vertices with $v.\delta = v.d$ gets larger

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

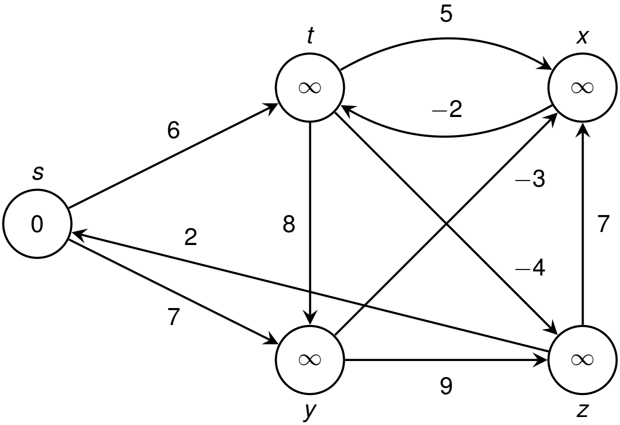
```
 BELLMAN-FORD(G,w,s)
0: assert(s in G.vertices())
1: for v in G.vertices()
2:    v.predecessor = None
3:    v.d = Infinity
4: s.d = 0
5:
6: repeat |V|-1 times
7:    for e in G.edges()
8:    Relax edge e=(u,v):  Check if u.d + w(u,v) < v.d
9:       if e.start.d + e.weight.d < e.end.d:
10:         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) (1/5)

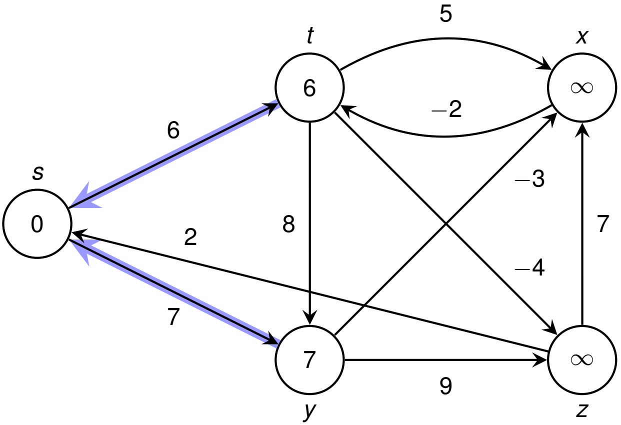
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4) (2/5)

Pass: 1

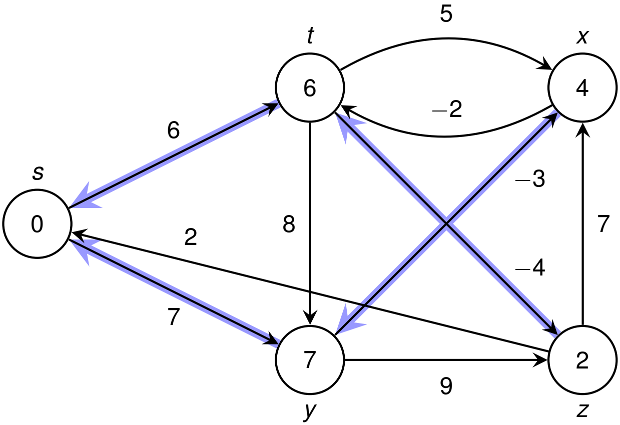
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4) (3/5)

Pass: 2

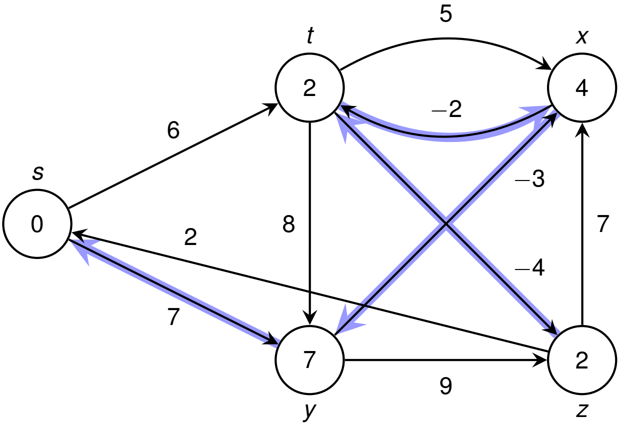
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Execution of Bellman-Ford (Figure 24.4) (4/5)

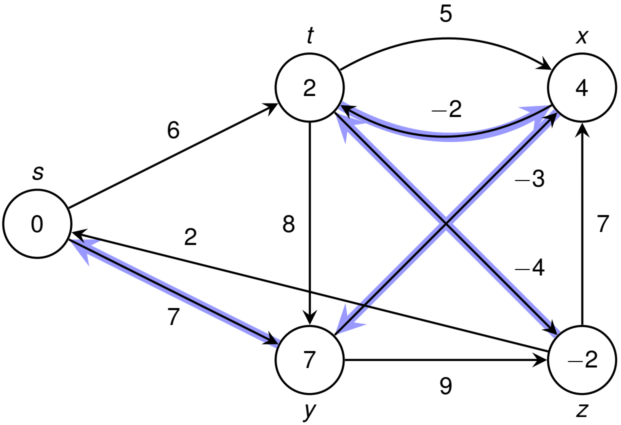
Pass: 3

Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



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.

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.

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 (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 $i + 1$.

Outline

Introduction

Bellman-Ford Algorithm

Dijkstra's Algorithm

Historical Remarks

Source: Wikipedia

Edsger Wybe Dijkstra (1930-2002)

- Dutch computer scientist
- developed Dijkstra's shortest path algorithm in 1956 (and published in 1959)
- many more fundamental contributions to computer science and engineering
- Turing Award (1972)

Some Quotes

“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”

“If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.”

“FORTRAN’s tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes.”

“Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.”

Recap: Prim's Algorithm (1/4)

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 incident edge connecting to A

Use a Priority Queue!

Recap: Prim's Algorithm (2/4)

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

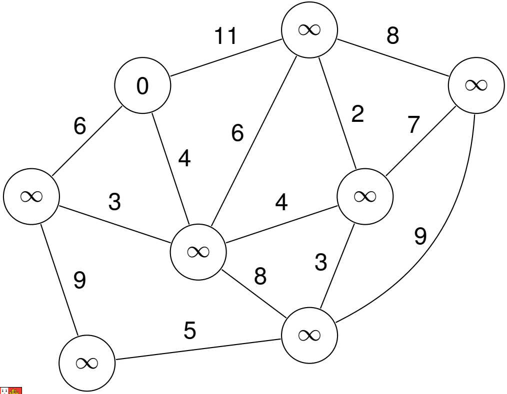
Implementation

- 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

Recap: Prim's Algorithm (3/4)

Basic Strategy

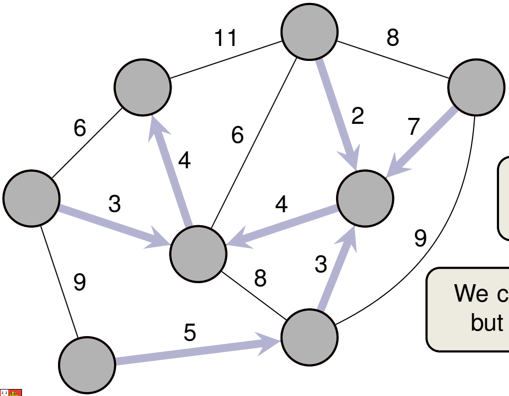
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Recap: Prim's Algorithm (4/4)

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



Final MST is given (implicitly) by the pointers!

We computed same MST as Kruskal, but in a completely different order!

Prim's Algorithms vs. Dijsktra's Algorithm

Prim's Algorithm

- Grows a tree that will eventually become a (minimum) spanning tree
- A is the set of vertices which have been connected so far
- Value of a vertex:
  - If $u \in A$, then it has no value.
  - If $u \notin A$, then it is equal to the smallest weight of an edge connecting to A (if such edge exists, otherwise ∞.)

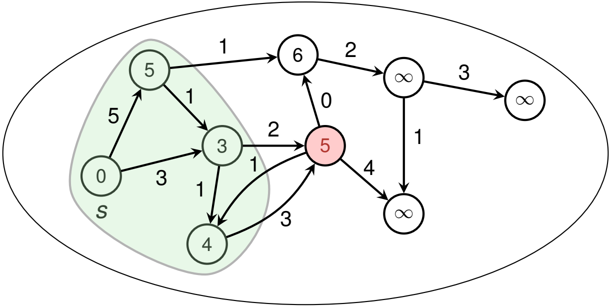
Dijsktra's Algorithm

- Grows a tree that will eventually become a shortest-path tree
- S is the set of vertices in the (current) shortest-path tree
- Value of a vertex:
  - If $u \in S$, then it is the actual distance from the source s to u.
  - If $u \notin S$, then it may be any value (including ∞) that is at least the distance from the source s.

Dijkstra's Algorithm (1/2)

Overview of Dijkstra

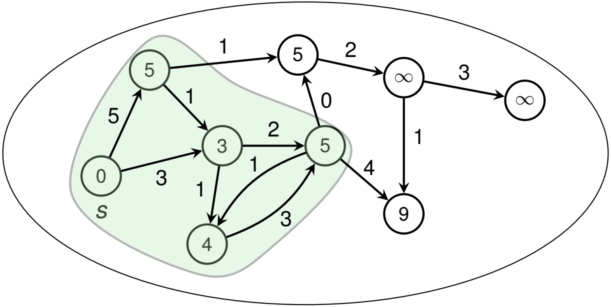
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = v.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$



Dijkstra's Algorithm (2/2)

Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = v.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

DIJKSTRA(G,w,s)

```
0: INITIALIZE(G,s)
1:   S = ∅
2:   Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$
- DecreaseKey (l. 7): $\mathcal{O}(E \cdot 1)$

$\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$

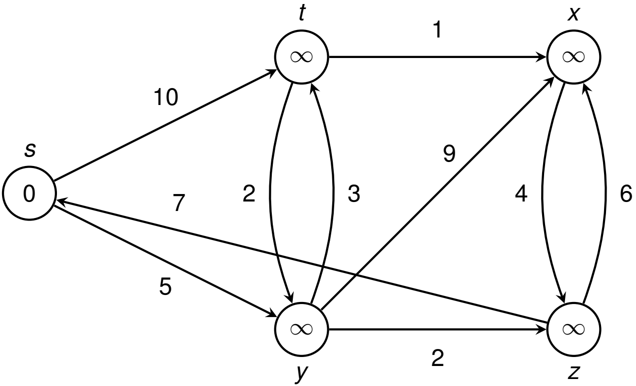
With a binary heap instead, the overall runtime would be $O(E \cdot \log V)$!

Prim's algorithm has the same runtime!

Execution of Dijkstra (Figure 24.6) (1/6)

Priority Queue Q:

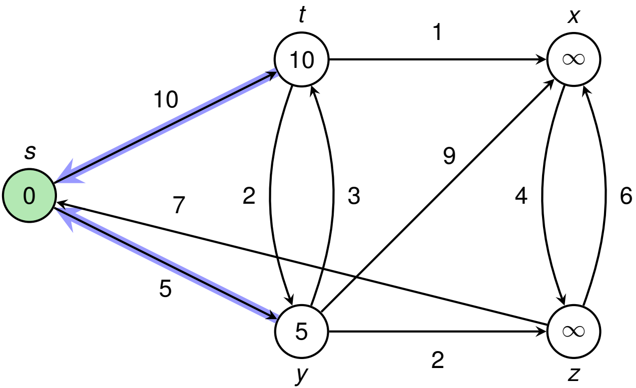
$(s, 0), (t, \infty), (x, \infty), (y, \infty), (z, \infty)$



Execution of Dijkstra (Figure 24.6) (2/6)

Priority Queue Q:

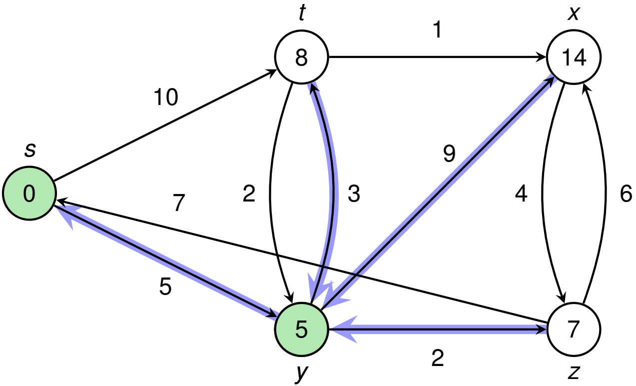
~~$(s, 0)$~~, $(t, 10), (x, \infty), (y, 5), (z, \infty)$



Execution of Dijkstra (Figure 24.6) (3/6)

Priority Queue Q:

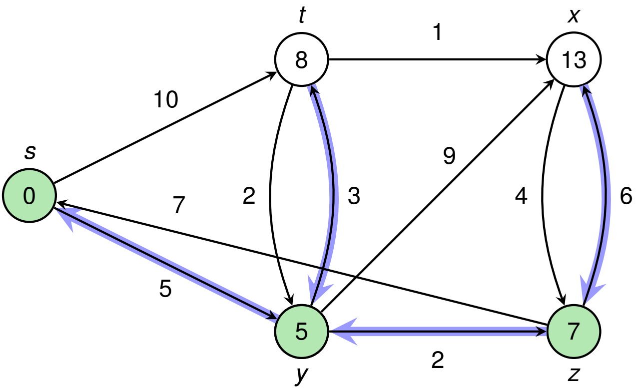
$(t, 8), (x, 14), \sout{(y, 5)}, (z, 7)$



Execution of Dijkstra (Figure 24.6) (4/6)

Priority Queue Q:

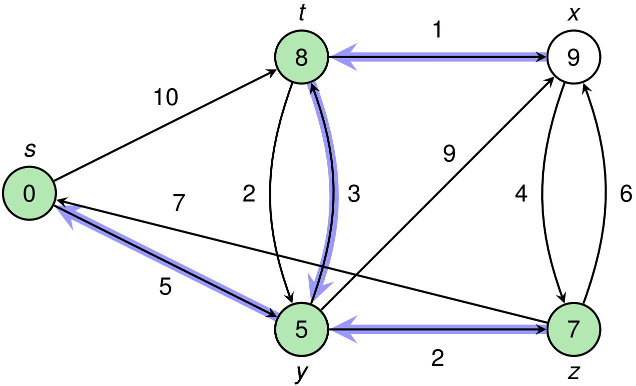
$(t, 8), (x, 13), \sout{(z, 7)}$



Execution of Dijkstra (Figure 24.6) (5/6)

Priority Queue Q:

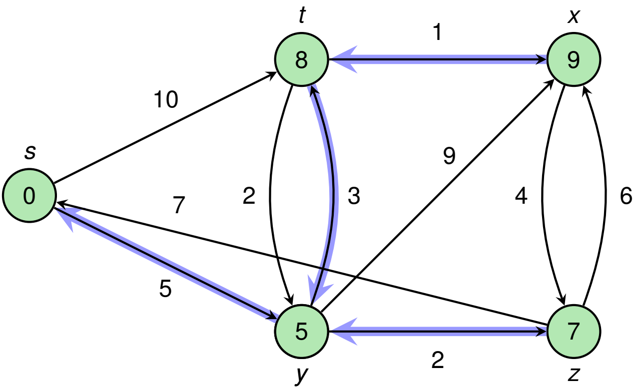
~~$(t, 8)$~~, $(x, 9)$



Execution of Dijkstra (Figure 24.6) (6/6)

Priority Queue Q:

~~(x, 9)~~



Dijkstra's Algorithm: Correctness (1/5)

Correctness (Theorem 24.6)

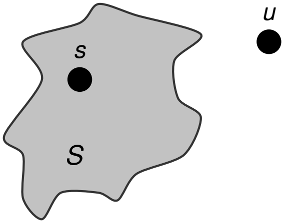
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \rightarrow \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property



Dijkstra's Algorithm: Correctness (2/5)

Correctness (Theorem 24.6)

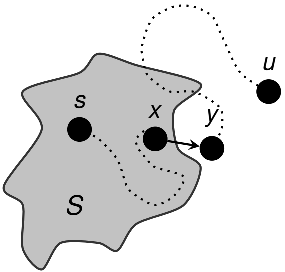
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$



Dijkstra's Algorithm: Correctness (3/5)

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

- Suppose there is $u \in V$, when extracted,

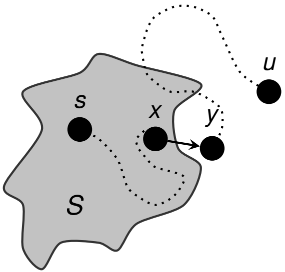
$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.d \leq y.d$$

u is extracted before y



Correctness (Theorem 24.6)

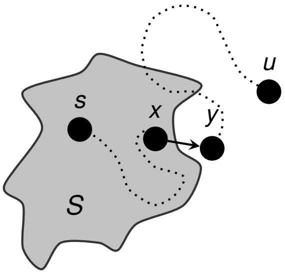
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$



$\Rightarrow$

$$u.d \leq y.d = y.\delta$$

since $x.d = x.\delta$ when x is extracted, and then (x, y) is relaxed $\Rightarrow$ Convergence Property

Dijkstra's Algorithm: Correctness (5/5)

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

There are edge cases like $s = x$ and/or $y = u$!

Proof: Invariant: If v is extracted, $v.d = v.\delta$

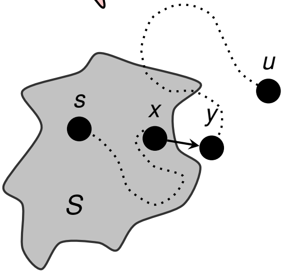
- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.\delta < u.d \le y.d = y.\delta$$

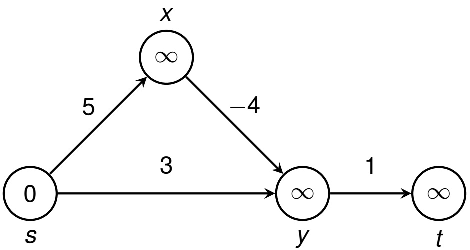


This contradicts that y is on a shortest path from s to u. □

Why negative-weight edges don't work (1/6)

Priority Queue Q:

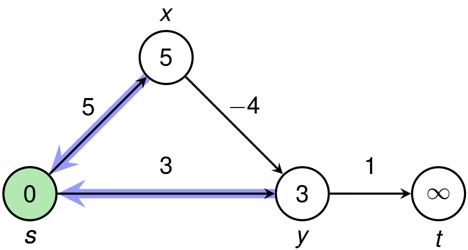
$(s, 0), (t, \infty), (x, \infty), (y, \infty)$



Why negative-weight edges don’t work (2/6)

Priority Queue Q:

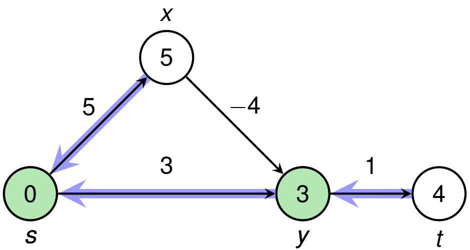
~~$(s, 0)$~~, $(t, \infty), (x, 5), (y, 3)$



Why negative-weight edges don't work (3/6)

Priority Queue Q:

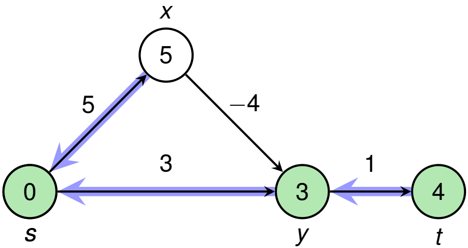
$(t, 4), (x, 5),$ ~~$(y, 3)$~~



Why negative-weight edges don't work (4/6)

Priority Queue Q:

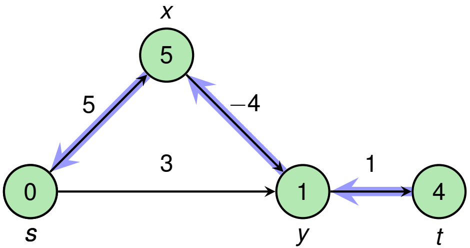
~~$(t, 4)$~~, $(x, 5)$



Why negative-weight edges don't work (5/6)

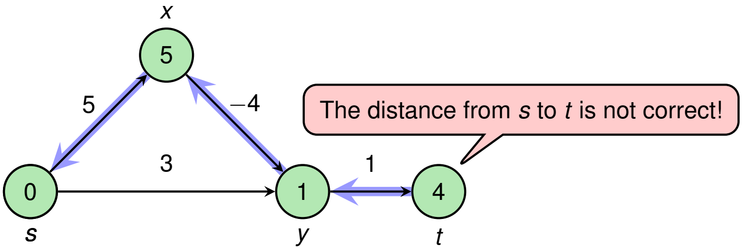
Priority Queue Q:

~~(x, 5)~~



Why negative-weight edges don't work (6/6)

Priority Queue Q:



Summary of Single-Source Shortest Paths

Overview

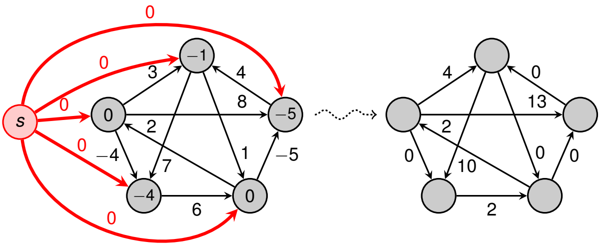
- studied two algorithms for SSSP (single-source shortest path)
- basic operation: relaxing edges

Bellman-Ford Algorithm

- detects negative-weight cycles
- V passes of relaxing all edges (arbitrary order)
- Runtime $\mathcal{O}(V \cdot E)$

Dijkstra's Algorithm

- requires non-negative weights
- Greeedy strategy to choose which edge to relax (similar to Prim)
- Using Fibonacci Heaps $\Rightarrow$ Runtime $\mathcal{O}(V \log V + E)$



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:

$$w_{i,j} = \begin{cases} \text{weight of edge } (i,j) & \text{for an edge } (i,j) \in E, \\ \infty & \text{if there is no edge from } i \text{ to } j, \\ 0 & \text{if } i = j. \end{cases}$$

- 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

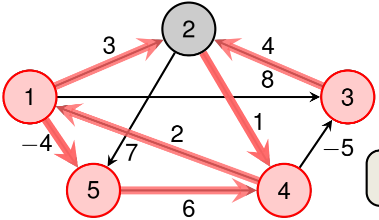
- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

$$\ell_{i,j}^{(2)} = \min\left(\ell_{i,j}^{(1)}, \min_{1 \leq k \leq n} \ell_{i,k}^{(1)} + w_{k,j}\right)$$

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)



$$\ell_{1,4}^{(2)} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\ell_{3,5}^{(4)} = \min\{7-4, 4+7, 0+\infty, 5+\infty, 11+0\}$$

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 $L^{(m)}$:

$$\ell_{i,j}^{(m)} = \min_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} + w_{k,j} \right)$$

$$(L^{(m-1)} \cdot W)_{i,j} = \sum_{1 \leq k \leq n} \left(\ell_{i,k}^{(m-1)} \times w_{k,j} \right)$$

$L^{(m)}$ can be computed in $\mathcal{O}(n^3)$

- The correspondence is as follows:

$$\min \Leftrightarrow \sum$$

$$+ \Leftrightarrow \times$$

$$\infty \Leftrightarrow 0$$

$$0 \Leftrightarrow 1$$

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

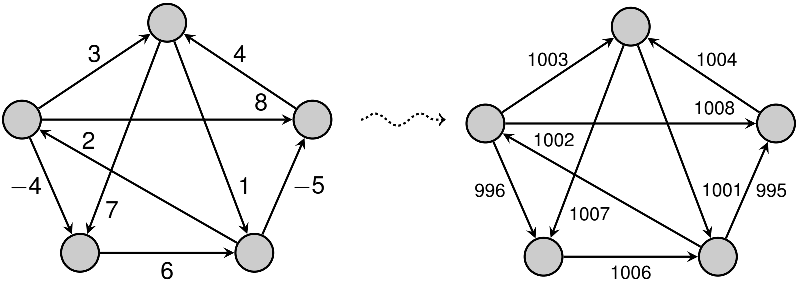
Johnson's Algorithm

Johnson's Algorithm

Overview

- 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!

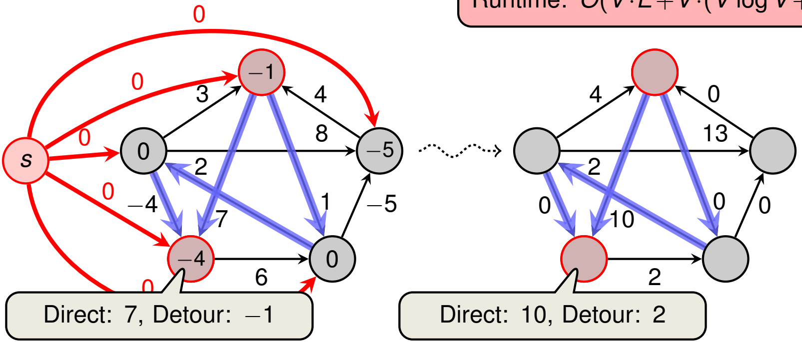


How Johnson's Algorithm works

Johnson's Algorithm

1. Add a new vertex s and directed edges (s, v), $v \in V$, with weight 0
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: $O(V \cdot E + V \cdot (V \log V + E))$



Correctness of Johnson's Algorithm (1/2)

$$\widetilde{w}(u,v) = w(u,v) + u.\delta - v.\delta$$

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

$$u.\delta + w(u,v) \geq v.\delta \qquad \text{(triangle inequality)}$$

$$\Rightarrow \qquad \widetilde{w}(u,v) + u.\delta + w(u,v) \geq w(u,v) + u.\delta - v.\delta + v.\delta$$

$$\Rightarrow \qquad \widetilde{w}(u,v) \geq 0$$

□

Correctness of Johnson's Algorithm (2/2)

$$\widetilde{w}(u,v) = w(u,v) + u.\delta - v.\delta$$

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

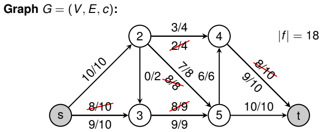
- In the original graph, the weight is $\sum_{i=1}^{k} w(v_{i-1}, v_i) = w(p)$.
- In the reweighted graph, the weight is

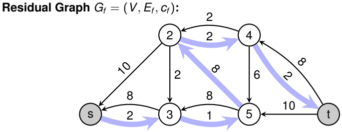
$$\sum_{i=1}^{k} \widetilde{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + v_{i-1}.\delta - v_i.\delta\right) = w(p) + v_0.\delta - v_k.\delta \quad \square$$

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





6.6: Maximum flow

Frank Stajano

Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Introduction

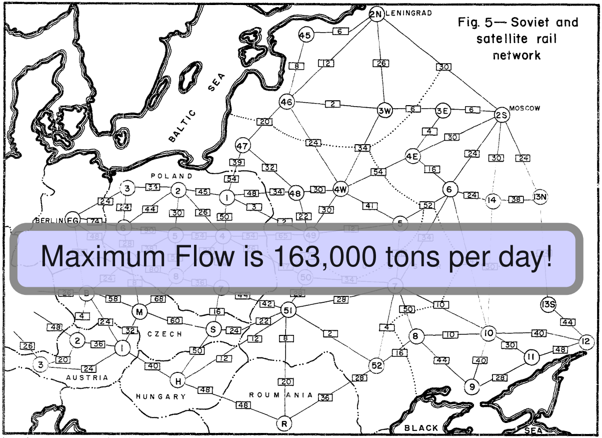
Ford-Fulkerson

A Glimpse at the Max-Flow Min-Cut Theorem

Analysis of Ford-Fulkerson

Matchings in Bipartite Graphs

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.

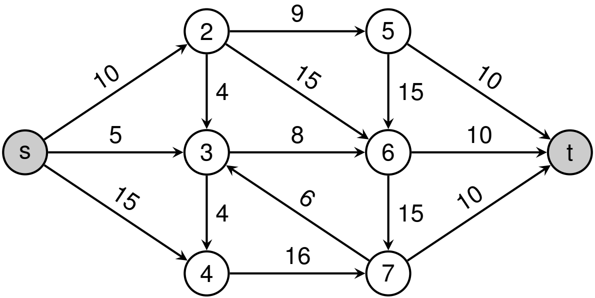
Flow Network (1/4)

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 $c : V \times V \to \mathbb{R}^+$

$c(u, v) = 0 \Leftrightarrow (u, v) \not\in E$



Flow Network (2/4)

Flow

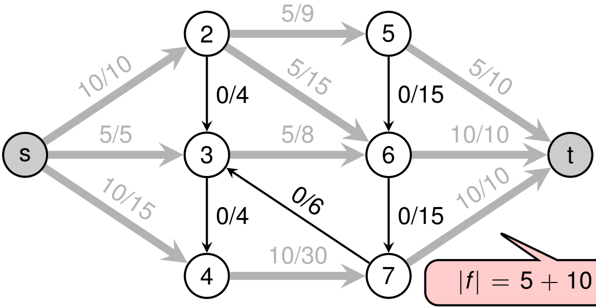
A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- 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 Conservation

$\sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$



$|f| = 5 + 10 + 10 = 25$

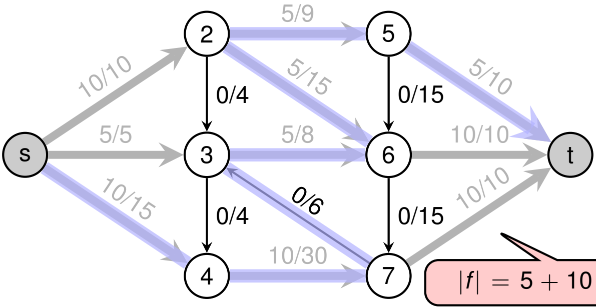
Flow Network (3/4)

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 (4/4)

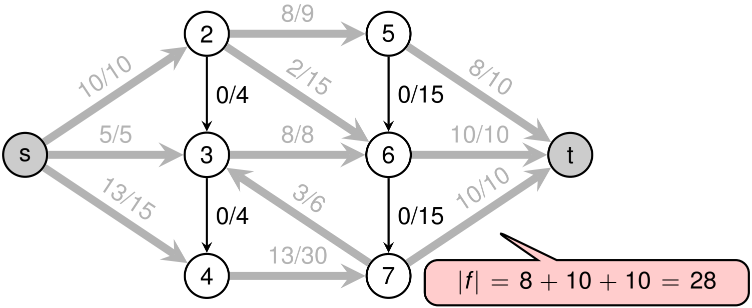
Flow

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 $|f| = \sum_{v \in V} f(s, v)$

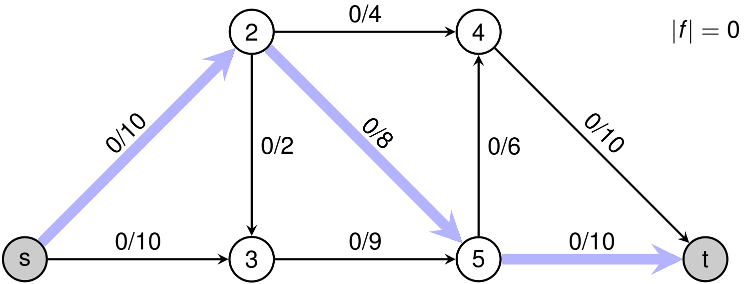
How to find a Maximum Flow?



A First Attempt (1/5)

Greedy Algorithm

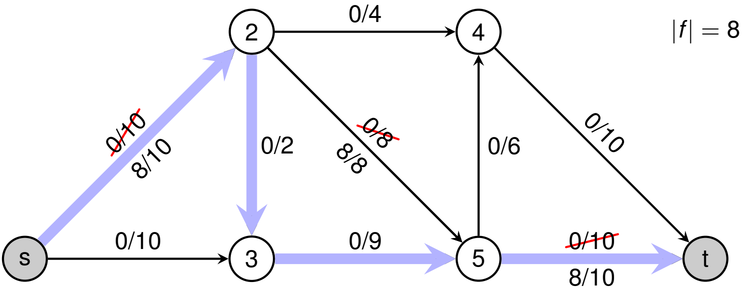
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt (2/5)

Greedy Algorithm

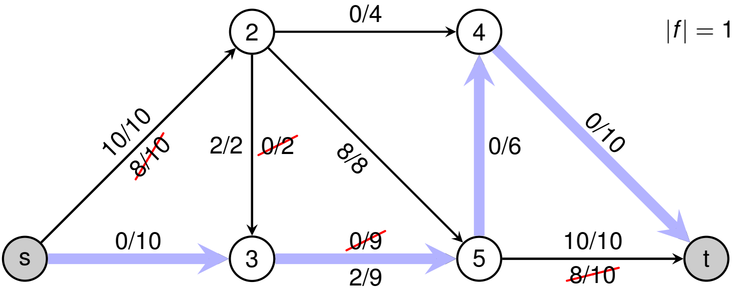
- Start with $f(u,v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s,t)-path p where each edge $e = (u,v)$ has $f(u,v) < c(u,v)$
  - Augment flow along p



A First Attempt (3/5)

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

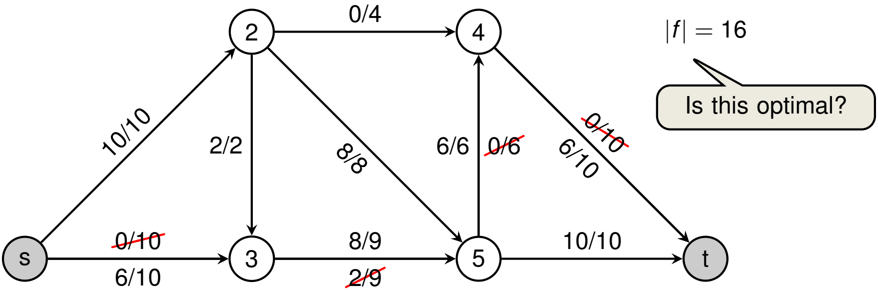


$|f| = 10$

A First Attempt (4/5)

Greedy Algorithm

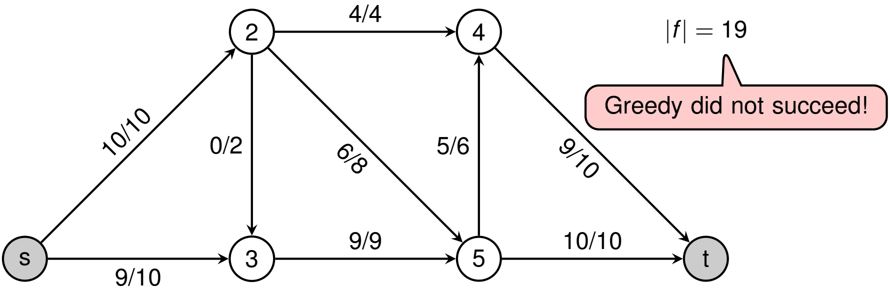
- Start with $f(u,v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s,t)-path p where each edge $e = (u,v)$ has $f(u,v) < c(u,v)$
  - Augment flow along p



A First Attempt (5/5)

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

Matchings in Bipartite Graphs

Residual Graph

Original Edge

Edge $e = (u, v) \in E$

- flow $f(u, v)$ and capacity $c(u, v)$

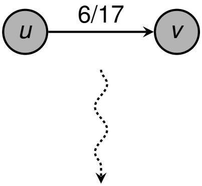
Residual Capacity

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E, \\ f(v, u) & \text{if } (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

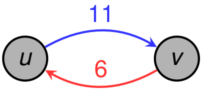
Residual Graph

- $G_f = (V, E_f, c_f)$, $E_f := \{(u, v)\colon c_f(u, v) > 0\}$

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 $c(u, v)$

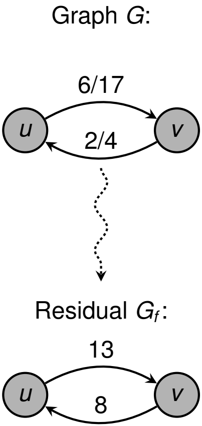
Residual Capacity

For every pair $(u, v) \in V \times V$,

$$c_f(u, v) = c(u, v) - f(u, v).$$

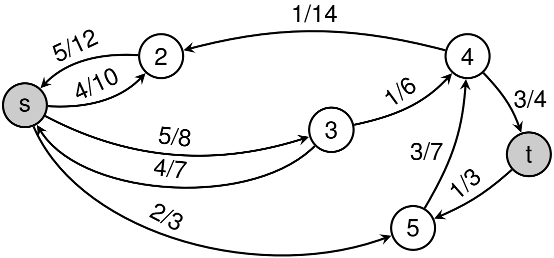
Residual Graph

- $G_f = (V, E_f, c_f)$, $E_f := \{(u, v) \colon c_f(u, v) > 0\}$

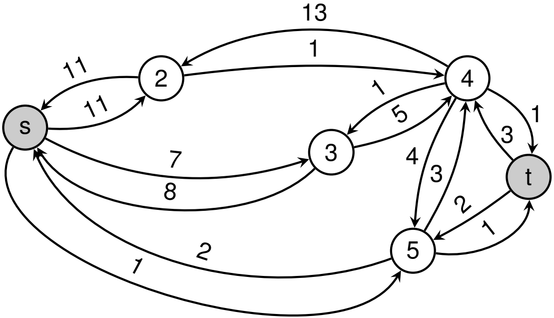


Example of a Residual Graph (Handout)

Flow network G



Residual Graph G_f



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
```

Augmenting path: Path from source to sink in G_f

If f' is a flow in G_f and f a flow in G, then $f + f'$ is a flow in G

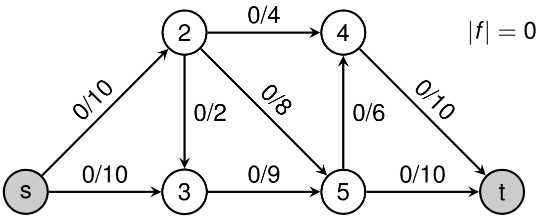
Using BFS or DFS, we can find an augmenting path in $O(V + E)$ time.

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 (1/7)

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

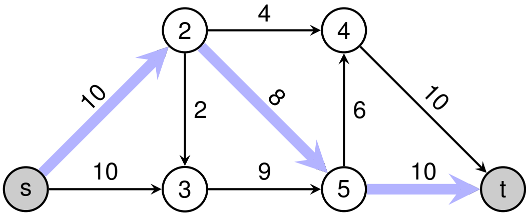
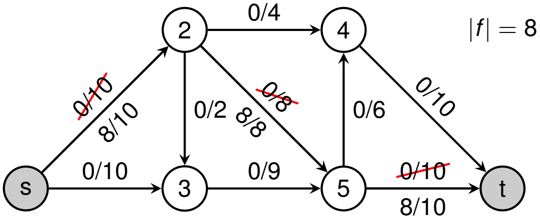


Illustration of the Ford-Fulkerson Method (2/7)

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

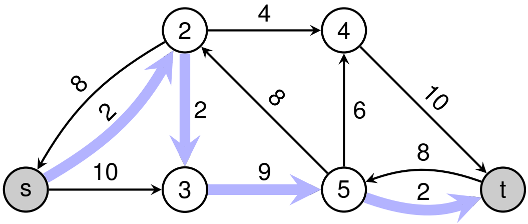
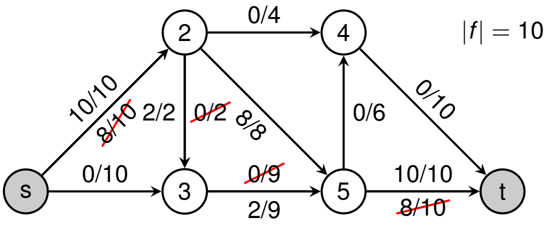
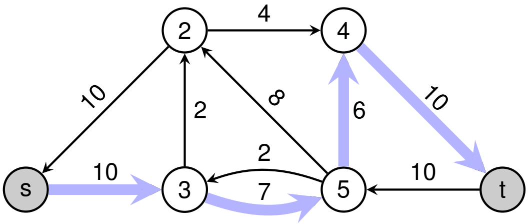


Illustration of the Ford-Fulkerson Method (3/7)

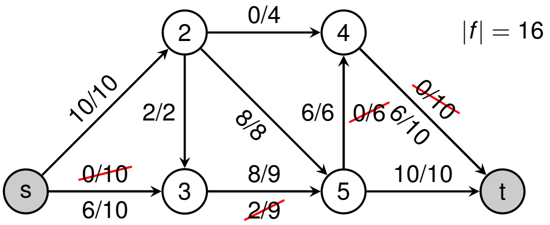
Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**



Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

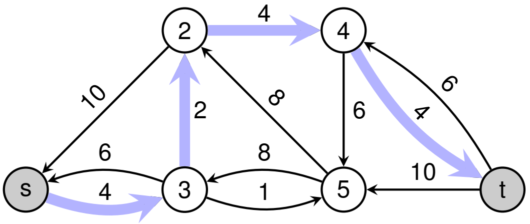
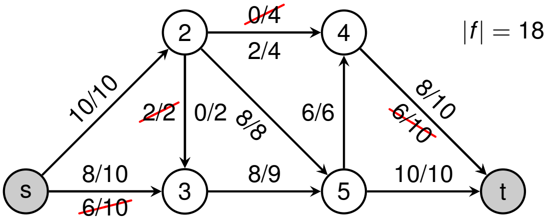


Illustration of the Ford-Fulkerson Method (5/7)

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

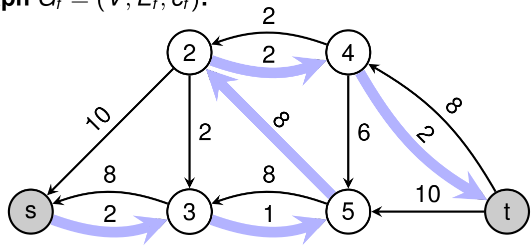
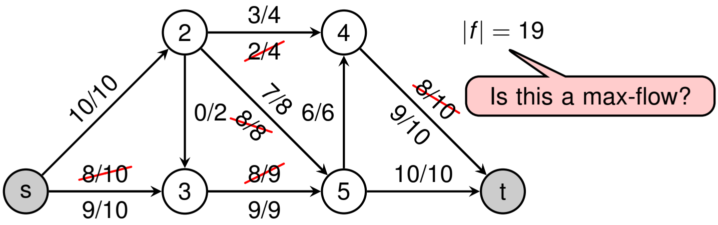


Illustration of the Ford-Fulkerson Method (6/7)

Graph $G = (V, E, c)$**:**



Residual Graph $G_f = (V, E_f, c_f)$**:**

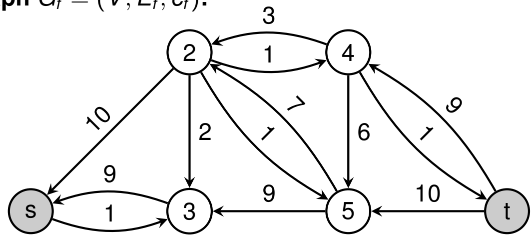
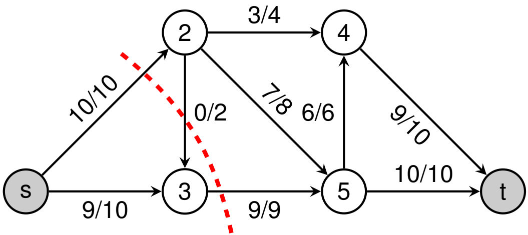
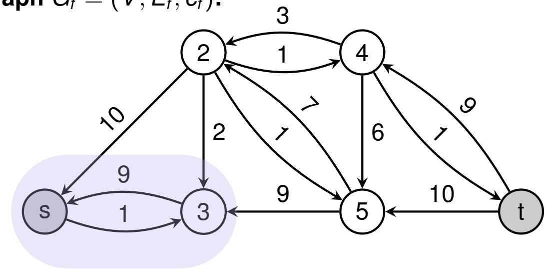


Illustration of the Ford-Fulkerson Method (7/7)

Graph $G = (V, E, c)$**:**



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

Matchings in Bipartite Graphs

From Flows to Cuts (1/3)

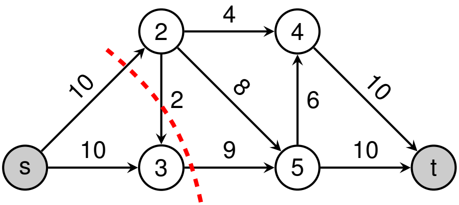
Cut

- 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 mininum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Graph $G = (V, E, c)$**:**



$\mathrm{c}(\{s, 3\}, \{2, 4, 5, t\}) = 10 + 9 = 19$

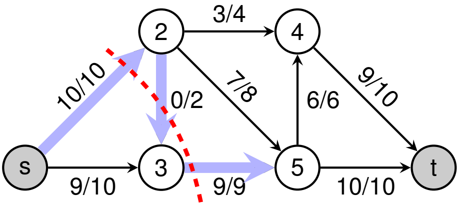
From Flows to Cuts (2/3)

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 $G = (V, E, c)$**:** $|f| = 19$



$10 - 0 + 9 = 19$

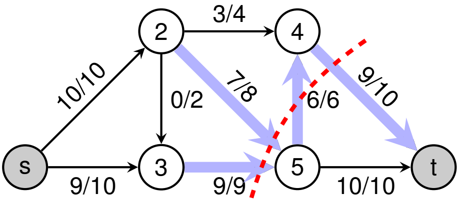
From Flows to Cuts (3/3)

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

Matchings in Bipartite Graphs

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.

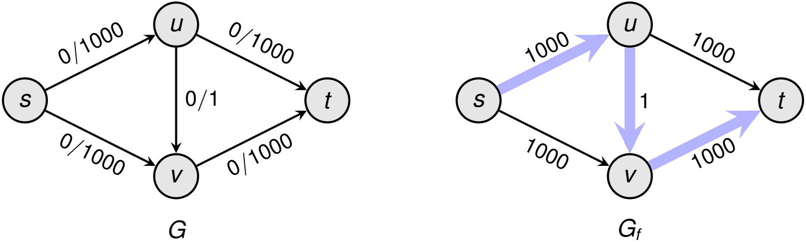
Flow before iteration integral
& capacities in G_f are integral
$\Rightarrow$ Flow after iteration integeral

Theorem

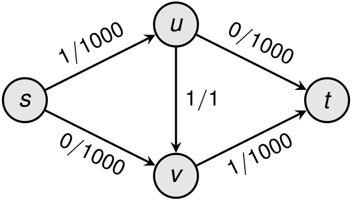
For integral capacities $c(u, v)$, Ford-Fulkerson terminates after $C := \max_{u,v} c(u, v)$ iterations and returns the maximum flow.

(proof omitted here, see CLRS3)

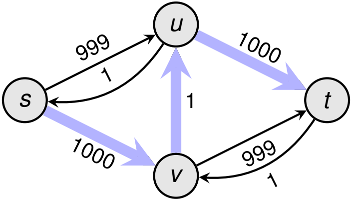
Slow Convergence of Ford-Fulkerson (Figure 26.7) (1/3)



Slow Convergence of Ford-Fulkerson (Figure 26.7) (2/3)

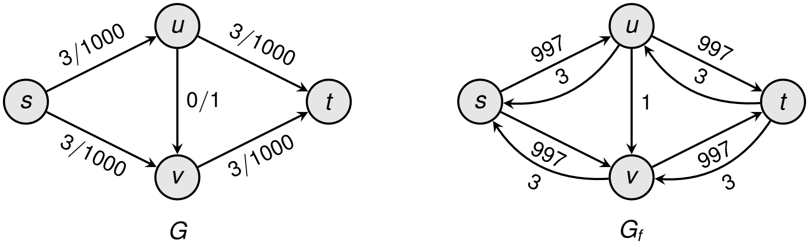


G



G_f

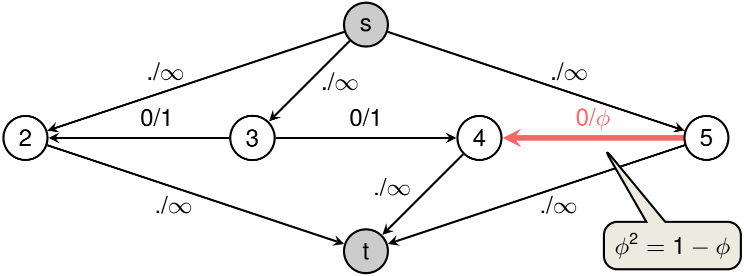
Slow Convergence of Ford-Fulkerson (Figure 26.7) (3/3)

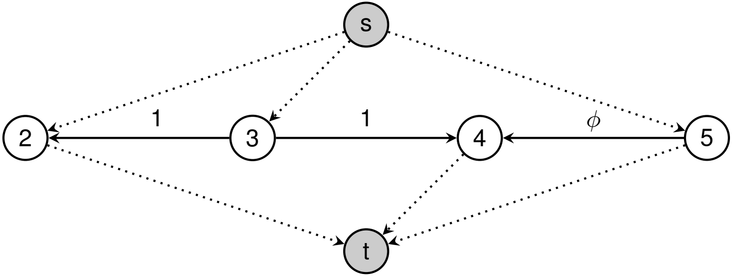


Number of iterations is $C := \max_{u,v} c(u,v)$!

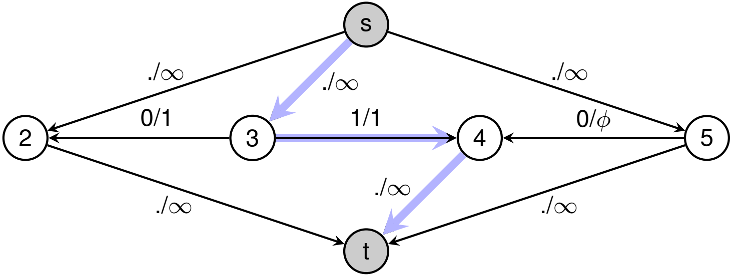
For irrational capacities, Ford-Fulkerson may even fail to terminate!

Non-Termination of Ford-Fulkerson for Irrational Capacities (1/8)

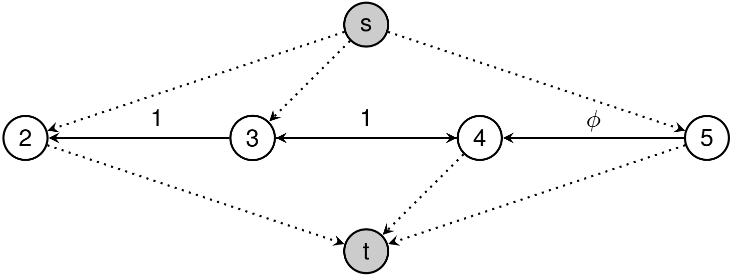




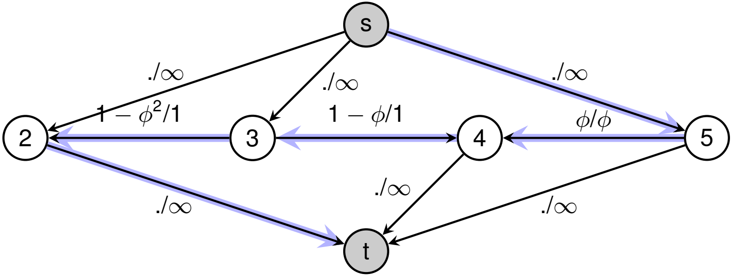
Non-Termination of Ford-Fulkerson for Irrational Capacities (2/8)



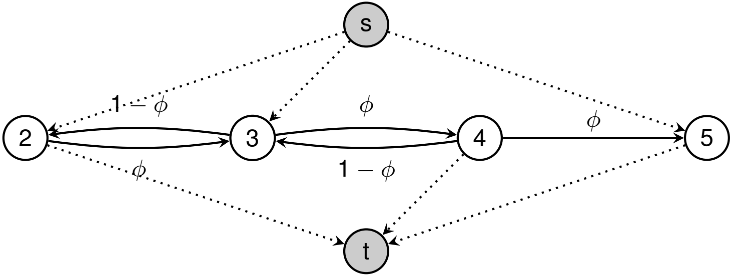
Iteration: 1, $|f| = 1$



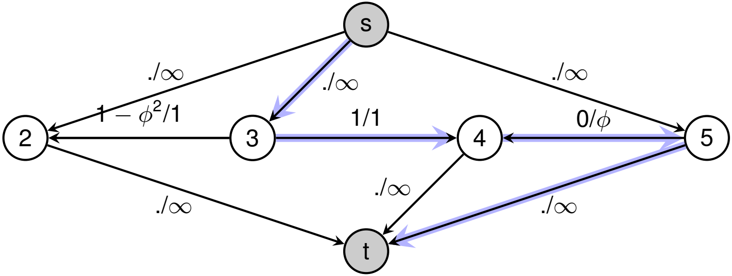
Non-Termination of Ford-Fulkerson for Irrational Capacities (3/8)



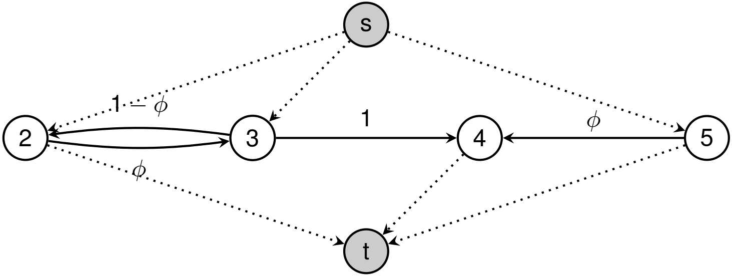
Iteration: 2, $|f| = 1 + \phi$



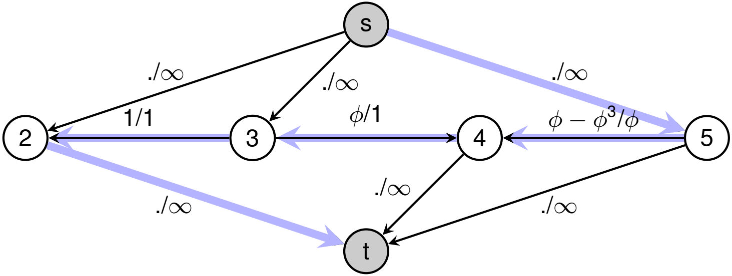
Non-Termination of Ford-Fulkerson for Irrational Capacities (4/8)



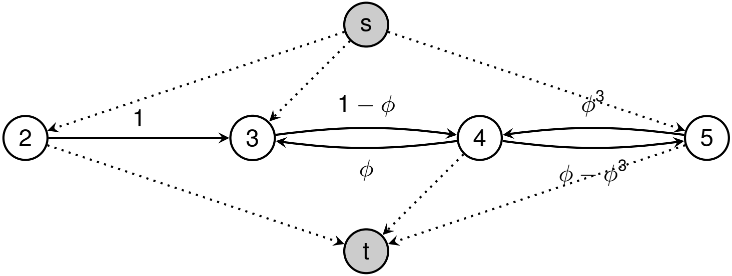
Iteration: 3, $|f| = 1 + 2 \cdot \phi$



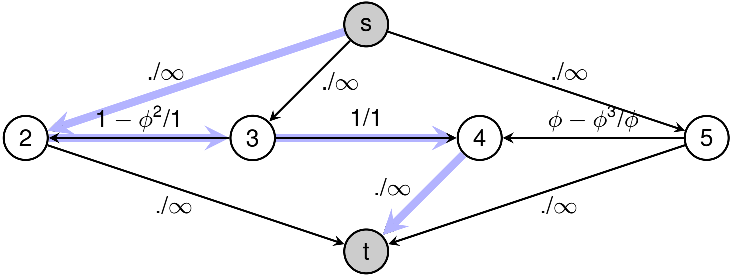
Non-Termination of Ford-Fulkerson for Irrational Capacities (5/8)



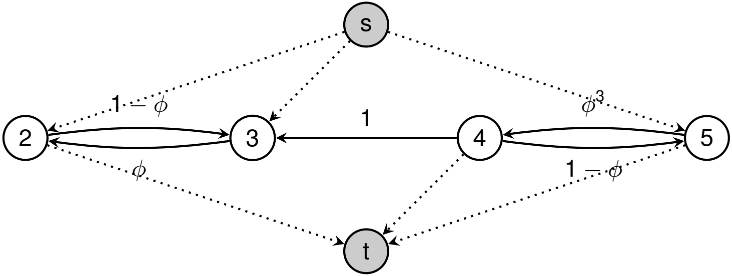
Iteration: 4, $|f| = 1 + 2 \cdot \phi + \phi^2$

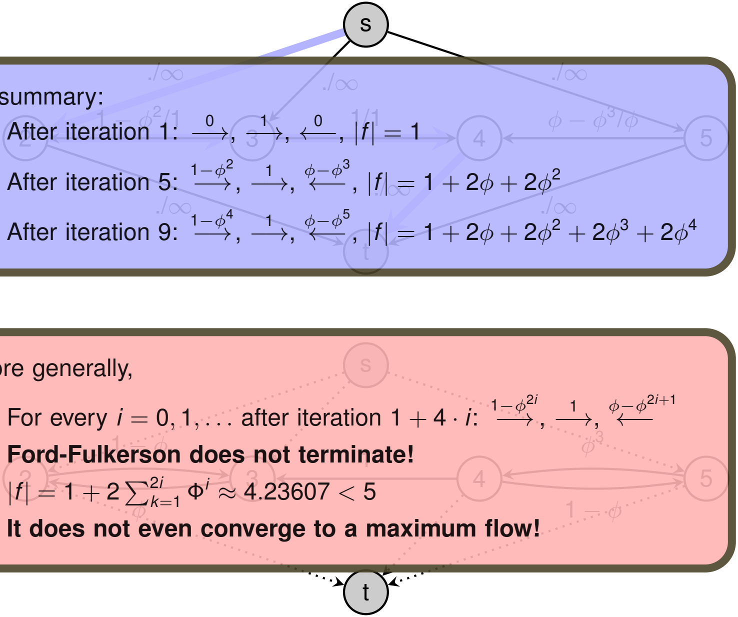


Non-Termination of Ford-Fulkerson for Irrational Capacities (6/8)



Iteration: 5, $|f| = 1 + 2 \cdot \phi + 2 \cdot \phi^2$





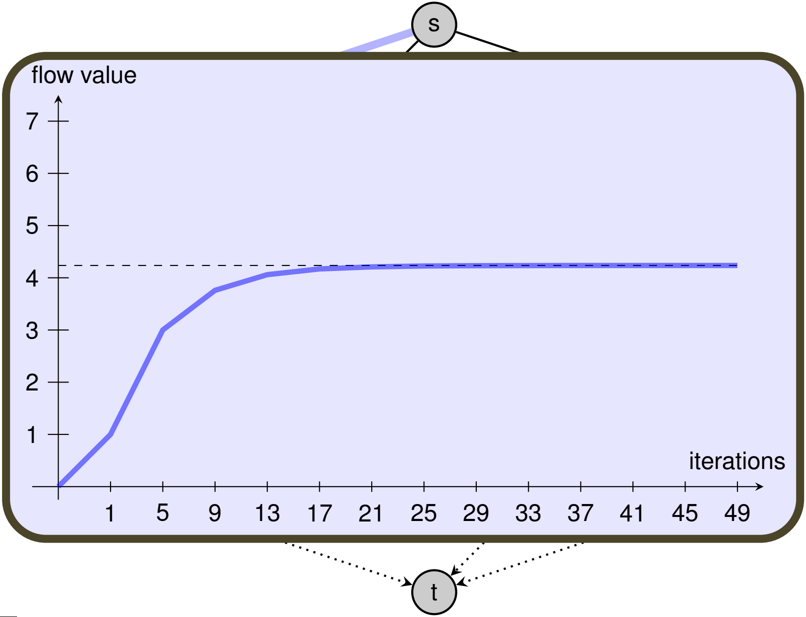
In summary:

- After iteration 1: $\xrightarrow{0}$, $\xrightarrow{1}$, $\xleftarrow{0}$, $|f| = 1$
- After iteration 5: $\xrightarrow{1-\phi^2}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^3}$, $|f| = 1 + 2\phi + 2\phi^2$
- After iteration 9: $\xrightarrow{1-\phi^4}$, $\xrightarrow{1}$, $\xleftarrow{\phi-\phi^5}$, $|f| = 1 + 2\phi + 2\phi^2 + 2\phi^3 + 2\phi^4$

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!**
- $|f| = 1 + 2\sum_{k=1}^{2i} \Phi^i \approx 4.23607 < 5$
- **It does not even converge to a maximum flow!**

Non-Termination of Ford-Fulkerson for Irrational Capacities (8/8)



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

Introduction

Ford-Fulkerson

A Glimpse at the Max-Flow Min-Cut Theorem

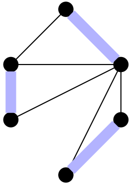
Analysis of Ford-Fulkerson

Matchings in Bipartite Graphs

Application: Maximum-Bipartite-Matching Problem

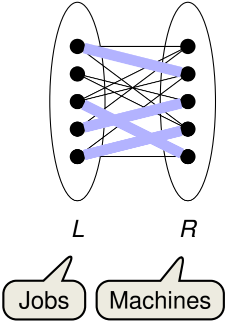
Matching

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.



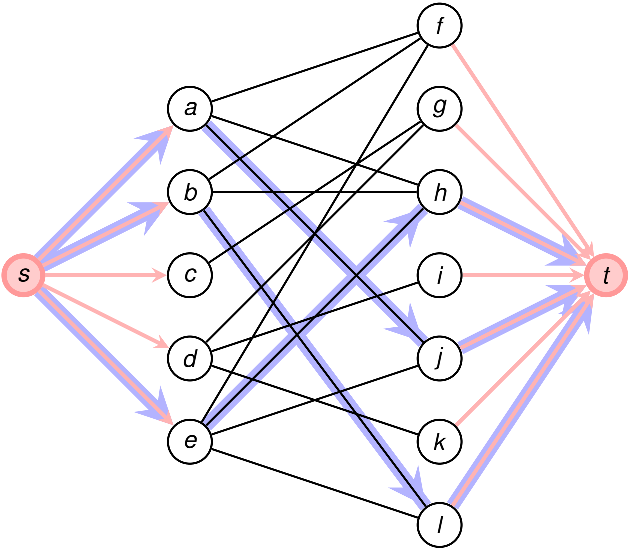
Bipartite Graph

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.

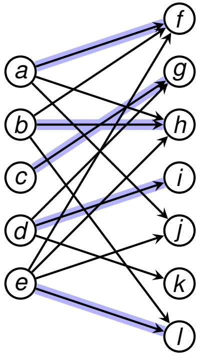
Matchings in Bipartite Graphs via Maximum Flows



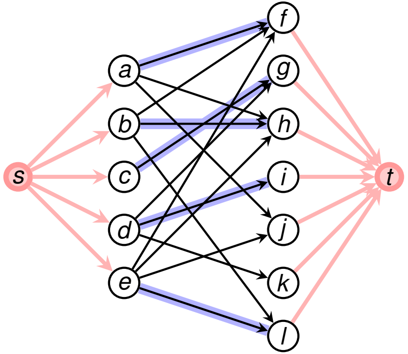
Correspondence between Maximum Matchings and Max Flow

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



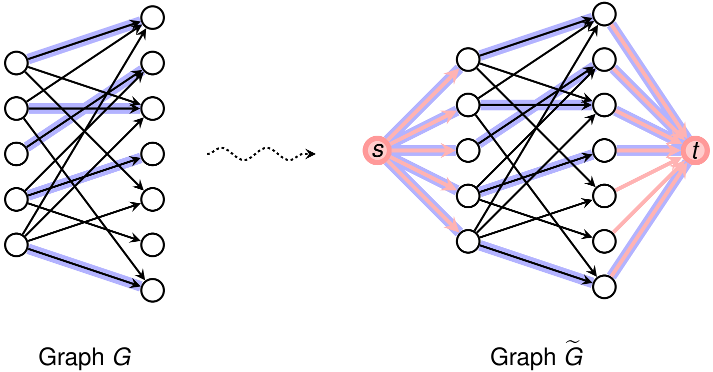
Graph $\widetilde{G}$

From Matching to Flow

- Given a maximum matching of cardinality k
- 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



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

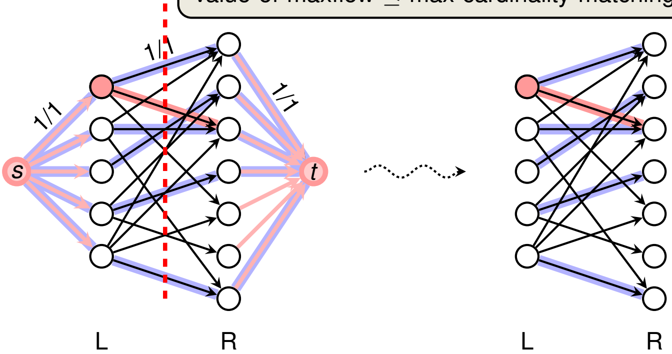
a) Flow Conservation $\Rightarrow$ every node in L sends at most one unit

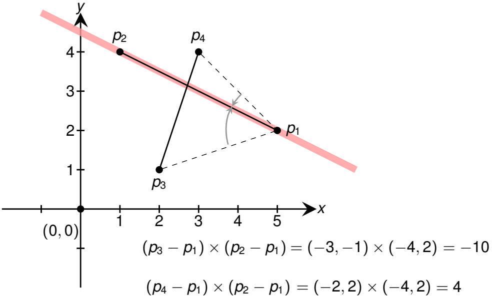
b) Flow Conservation $\Rightarrow$ every node in R receives at most one unit

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





7: Geometric Algorithms

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Introduction and Line Intersection

Convex Hull

Introduction

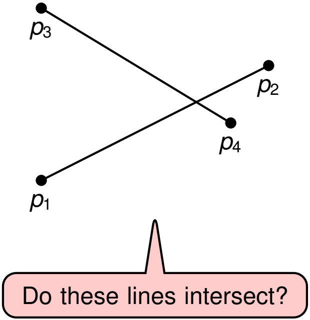
Computational Geometry

- Branch that studies algorithms for geometric problems
- typically, input is a set of points, line segments etc.

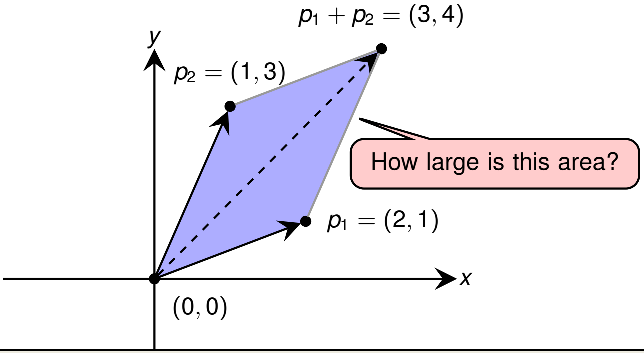
Applications

- computer graphics
- computer vision
- textile layout
- VLSI design

⋮



Cross Product (Area)



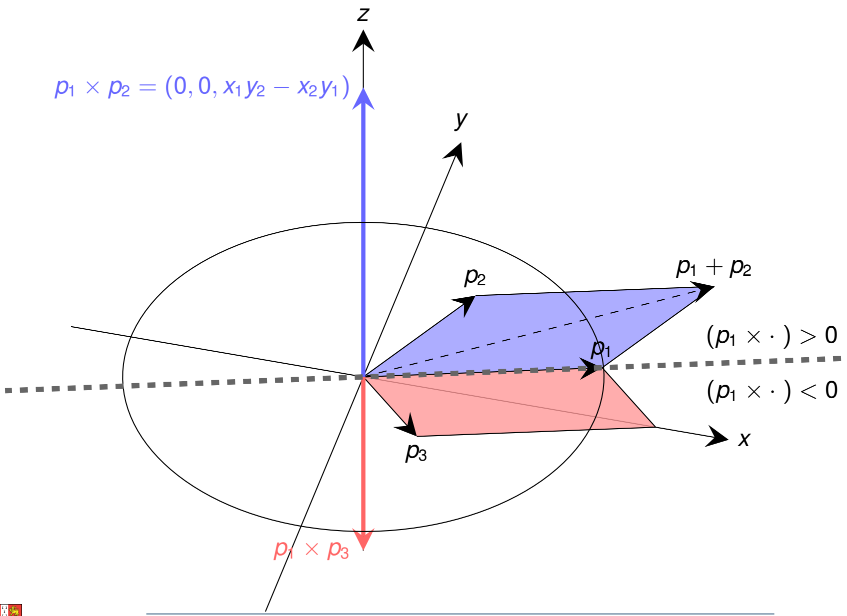
Alternatively, one could take the dot-product (but not used here):

$$p_1 \cdot p_2 = \|p_1\| \cdot \|p_2\| \cdot \cos(\phi).$$

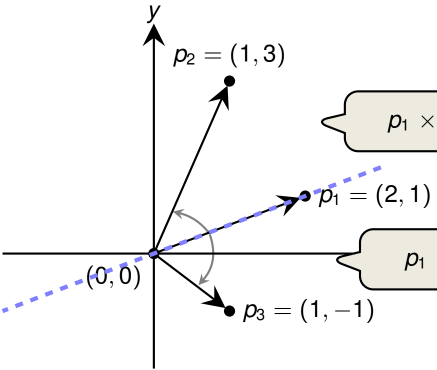
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

$$p_2 \times p_1 = y_1 x_2 - y_2 x_1 = -(p_1 \times p_2) = -5$$

Cross Product in 3D



Using Cross product to determine Turns



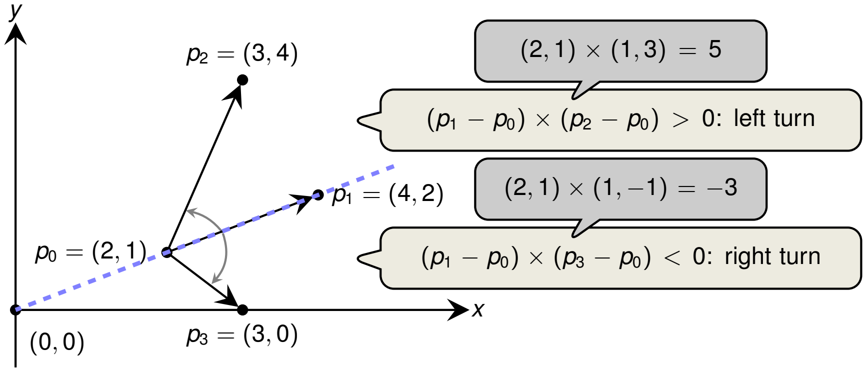
$p_1 \times p_2 > 0$: left (counterclockwise) turn

$p_1 \times p_3 < 0$: right (clockwise) turn

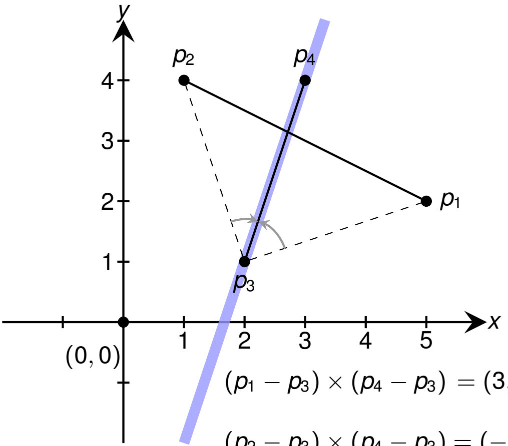
Sign of cross product determines turn!

Cross product equals zero iff vectors are colinear

Using Cross product to determine Turns (origin shifted)



Solving Line Intersection (1/4)

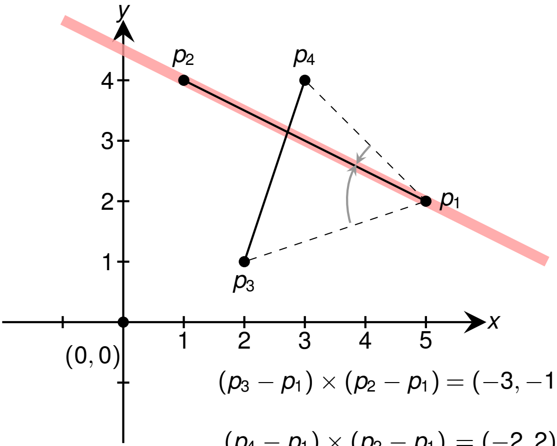


$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 3) = 8$$

$$(p_2 - p_3) \times (p_4 - p_3) = (-1, 3) \times (1, 3) = -6$$

Opposite signs $\Rightarrow \overline{p_1 p_2}$ crosses (infinite) line through p_3 and p_4

Solving Line Intersection (2/4)



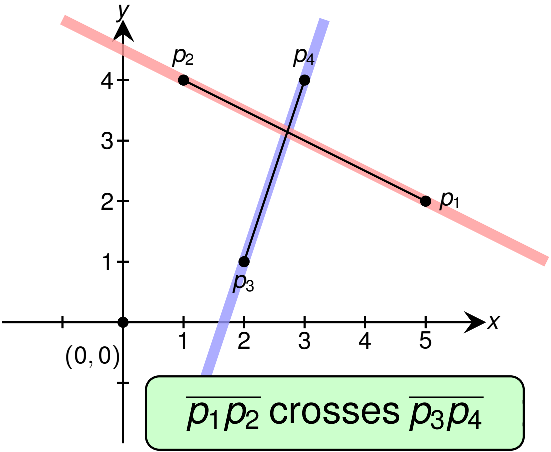
$$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (-4, 2) = -10$$

$$(p_4 - p_1) \times (p_2 - p_1) = (-2, 2) \times (-4, 2) = 4$$

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 (3/4)

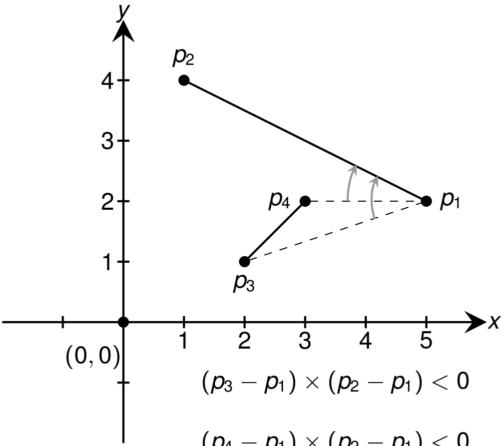


$\overline{p_1 p_2}$ crosses $\overline{p_3 p_4}$

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 (4/4)

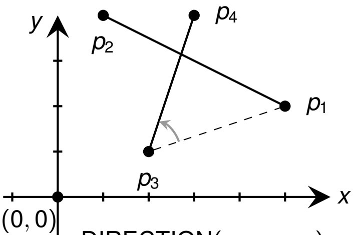


$$(p_3 - p_1) \times (p_2 - p_1) < 0$$

$$(p_4 - p_1) \times (p_2 - p_1) < 0$$

$\overline{p_1 p_2}$ does **not** cross $\overline{p_3 p_4}$

Solving Line Intersection



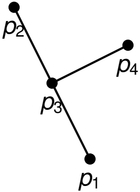
$$\text{DIRECTION}(p_3, p_4, p_1) = (p_1 - p_3) \times (p_4 - p_3)$$

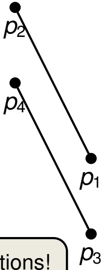
```
DIRECTION(p_i, p_j, p_k)
    return (p_k − p_i) × (p_j − p_i)
```

```
SEGMENTS-INTERSECT(p_i, p_j, p_k)
    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!

Lines could touch or be colinear



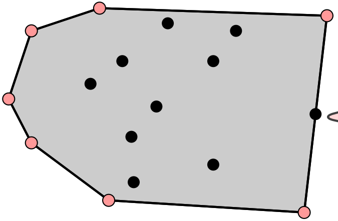


Outline

Introduction and Line Intersection

Convex Hull

Convex Hull



Vertex lies on the convex hull, but is not part of the polygon!

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 Problem

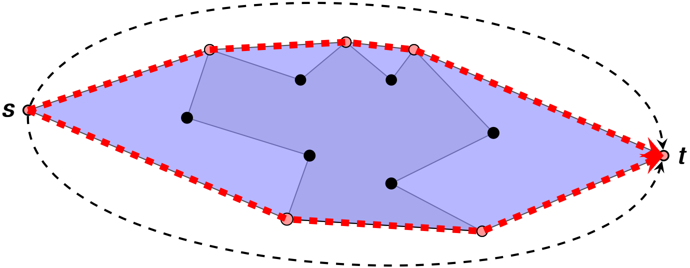
- Input: set of points Q in the Euclidean space
- Output: return points of the convex hull in counterclockwise order

Application of Convex Hull

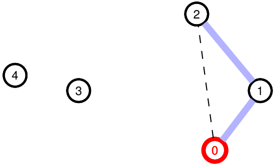
Robot Motion Planning

Find shortest path from s to t which avoids a polygonal obstacle.

can be solved by computing the Convex hull!



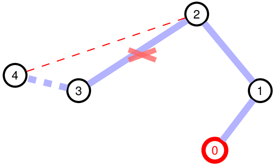
Graham's Scan (1/4)



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 ✓

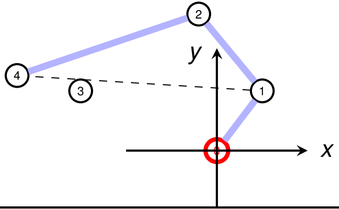
Graham's Scan (2/4)



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 (3/4)



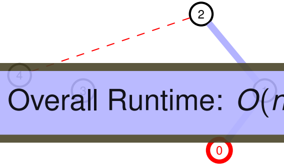
Use Cross Product!

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 ✓

Graham's Scan (4/4)



Overall Runtime: $O(n \log n)$

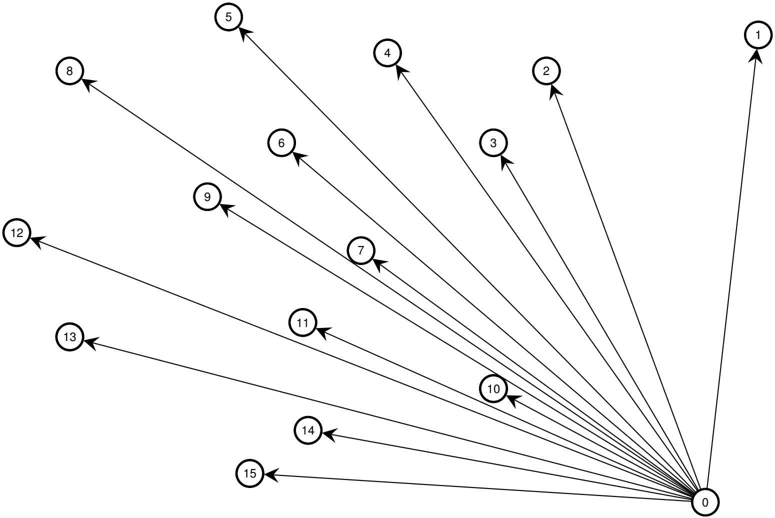
GRAHAM-SCAN(Q)
  Let p_0 be the point with minimum y-coordinate
  Let $(p_1, p_2, \ldots, p_n)$ be the other points sorted by polar angle w.r.t. p_0
  If $n < 2$ return false
  $S = \emptyset$
  PUSH(p_0,S)
  PUSH(p_1,S)
  PUSH(p_2,S)
  For $i = 3$ to n
    While angle of NEXT-TO-TOP(S),TOP(S),p_i makes 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 (1/2)

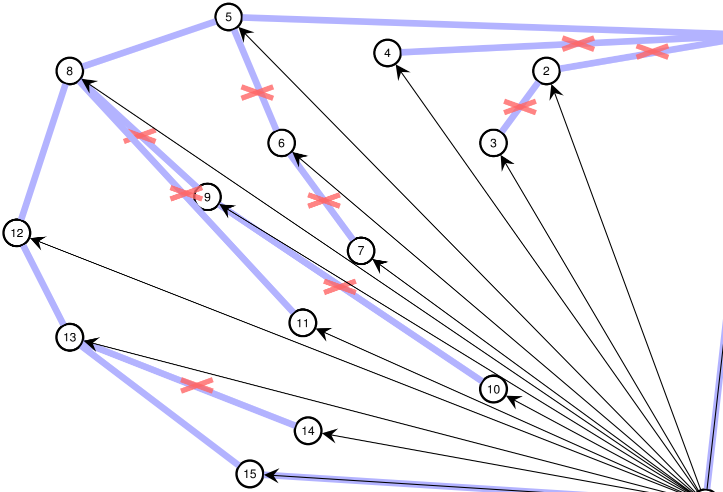
$i = 0$



Execution of Graham's Scan (2/2)

$i = 15$ ✓

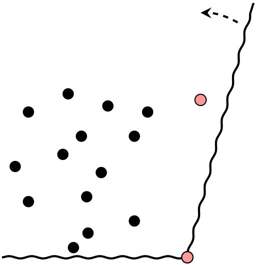
0	1	5	8	12	13	15



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

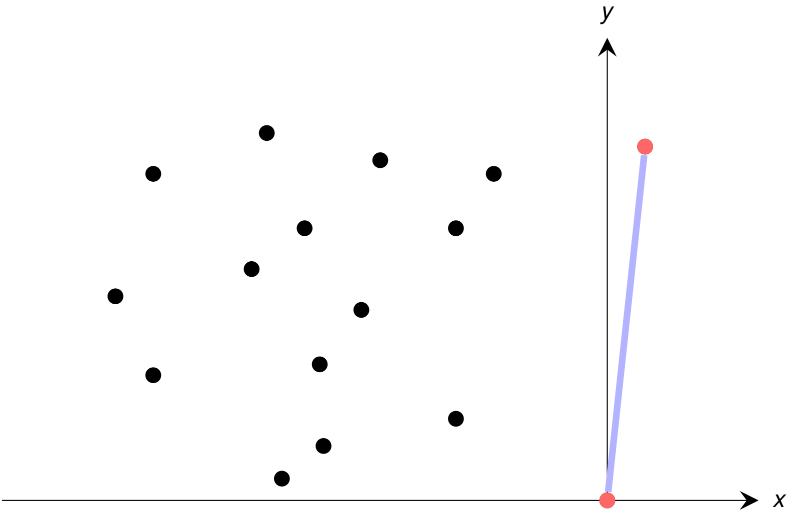
1. Let p_0 be the lowest point
2. Next point the one with smallest angle w.r.t. p_0
3. Continue until highest point p_k
4. Next point the one with smallest angle w.r.t. p_k
5. Continue until p_0 is reached

Here, we rotate the coordinate system by 180!

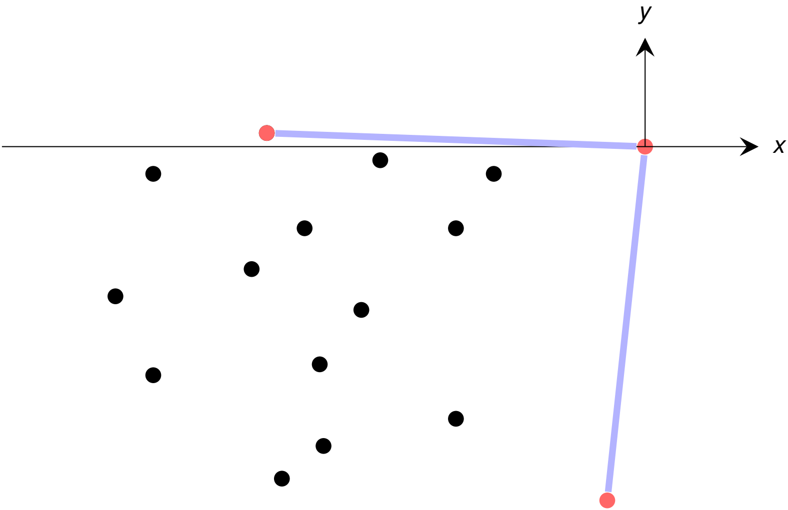
Runtime: $O(n \cdot h)$, where h is no. points on convex hull.

Output sensitive algorithm!

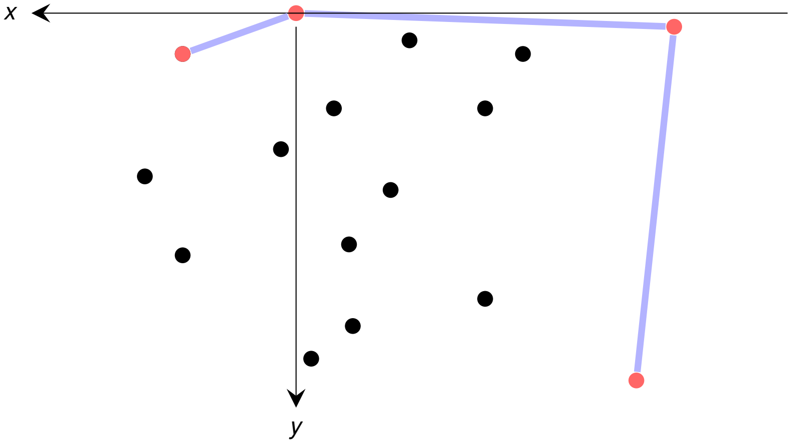
Execution of Jarvis' March (1/4)



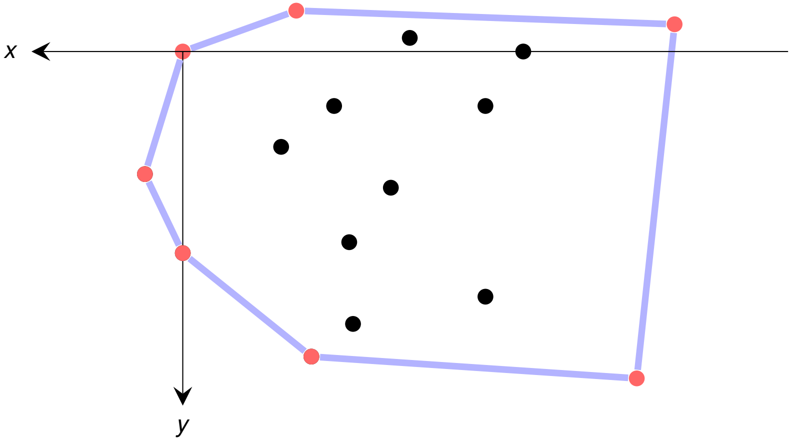
Execution of Jarvis' March (2/4)



Execution of Jarvis' March (3/4)



Execution of Jarvis' March (4/4)



Computing Convex Hull: Summary

Graham's Scan

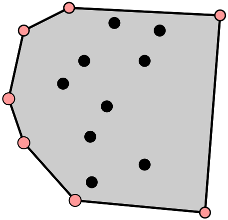
- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

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

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`