5.1: Amortized Analysis

Frank Stajano   Thomas Sauerwald

Lent 2015

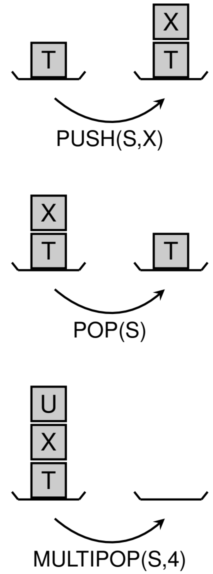
UNIVERSITY OF CAMBRIDGE

Motivating Example: Stack

Stack Operations

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

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



Motivating Example: Stack

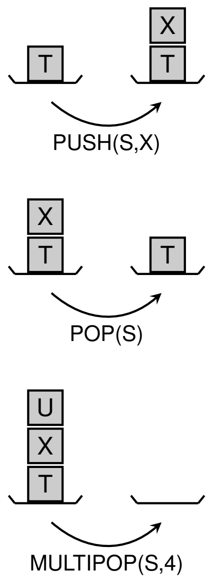
Stack Operations

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

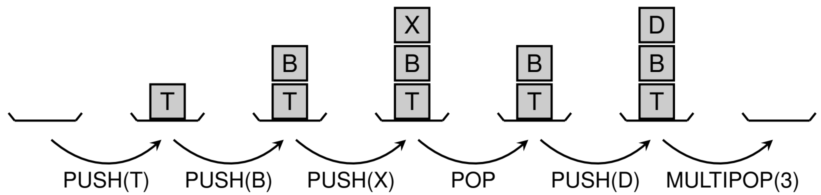
What is the largest possible cost of a sequence of n stack operations?

Simple Worst-Case Bound:

- 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

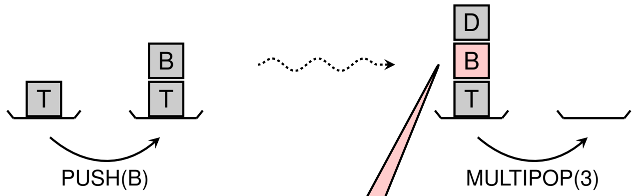
- Determine an upper bound $T(n)$ for the total cost of any sequence of n operations
- amortized cost of each operation is the average $\frac{T(n)}{n}$

Even though operations may be of different types/costs

Aggregate Analysis of the STACK

Simple Worst-Case Bound:

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

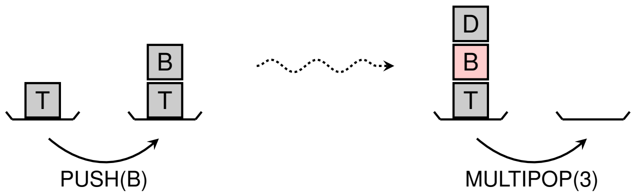


Every item which is POPPED from the stack had to be PUSHED earlier!

Aggregate Analysis of the STACK

Simple Worst-Case Bound:

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



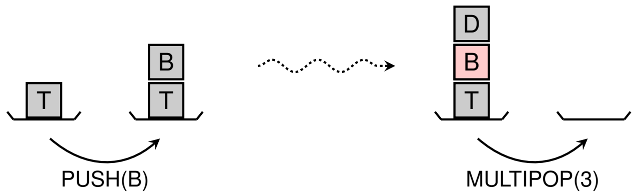
MULTIPOP(k) contributes $\min\{k, |S|\}$ to $T_{POP}(n)$

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

Aggregate Analysis of the STACK

Simple Worst-Case Bound:

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



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

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

Second Technique: Potential Method

Potential Method

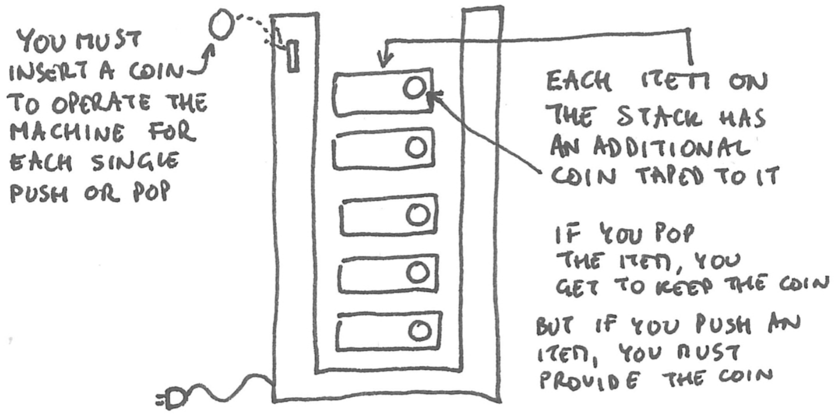
- allow different amortized costs

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

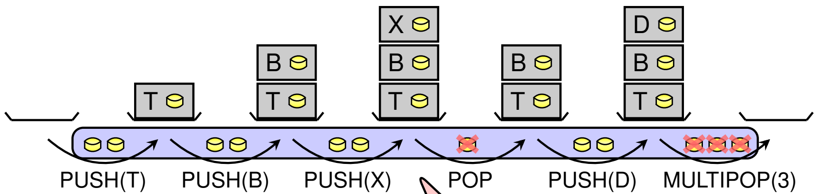
Potential of a data structure can be also thought of as

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

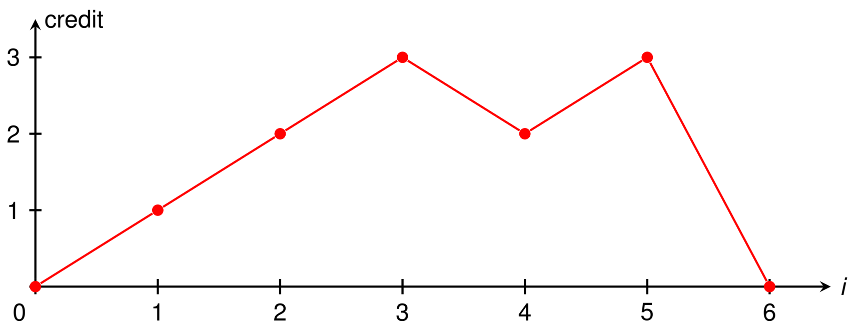
Stack as a coin-operated machine (p. 83)



Stack and Coins



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



Potential Method in Detail

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

- 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 ($\Phi_0 = 0$)

Function that maps states of the data structure to some value

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 ($\Phi_0 = 0$)

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

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

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 ($\Phi_0 = 0$)

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

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

Potential Method in Detail

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

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

$\Phi_1 - \Phi_0 + \Phi_2 - \Phi_1 + \cdots + \Phi_n - \Phi_{n-1}$

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

$$= \sum_{i=1}^{n} c_i + \Phi_n$$

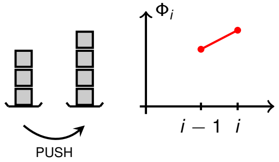
If $\Phi_n \geq 0$ for all n, sum of amortized costs is an upper bound for the sum of actual costs!

Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation

PUSH

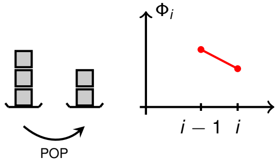
- actual cost: $c_i = 1$
- potential change: $\Phi_i - \Phi_{i-1} = 1$
- amortized cost: $\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2$



POP

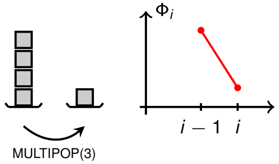
- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$

Stack is non-empty!



MULTIPOP(k)

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



Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation

PUSH

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

POP

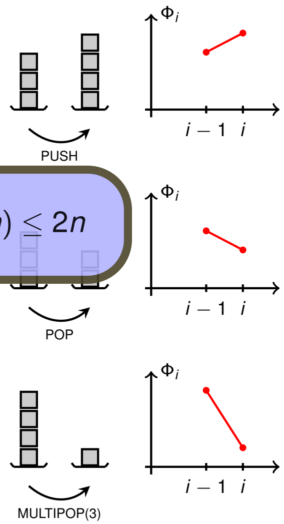
- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$

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

Stack is non-empty!

MULTIPOP(k)

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



Stack: Analysis via Potential Method

Φ_i = # objects in the stack after ith operation

PUSH

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

POP

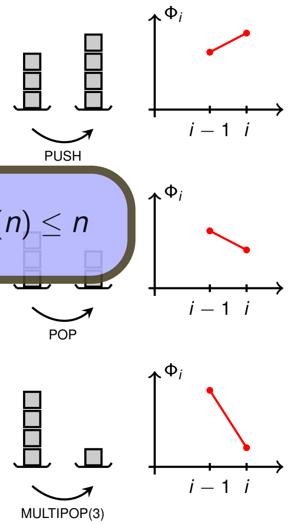
- $c_i = 1$
- $\Phi_i - \Phi_{i-1} = -1$
- $\widehat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 - 1 = 0$

$n/2$ 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: $\leq k$

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

1	0	1	1

11

INC

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

1	1	0	0

12

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: number of bit flips

```
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

Second Example: Binary Counter

Binary Counter

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

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

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

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

Φ_i = # ones in the binary representation of i

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

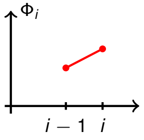
Increment without Carry-Over

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

1 1 0 0

INC

1 1 0 1



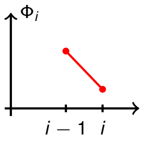
Increment with Carry-Over

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

0 1 1 1

INC

1 0 0 0



Binary Counter: Analysis via Potential Function

Φ_i = # ones in the binary representation of i

$\Phi_0 = 0$ ✓ $\Phi_i \geq 0$ ✓

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

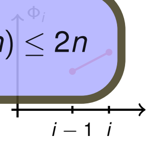
Increment without Carry-Over

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

1 1 0 0

INC

1 1 0 1



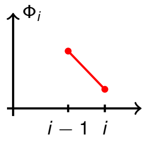
Increment with Carry-Over

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

0 1 1 1

INC

1 0 0 0



Summary

Amortized Analysis

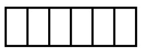
- Average costs over a sequence of n operations
- overcharge cheap operations and undercharge expensive operations
- no probability/average case analysis involved!

E.g. by bounding the number of expensive operations

Aggregate Analysis

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

$T(n)$

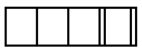


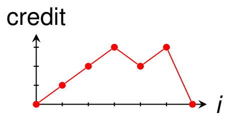
Full power of this method will become clear later!

Potential Method

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

$T(n)$





5.2 Fibonacci Heaps

Frank Stajano Thomas Sauerwald

Lent 2015

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

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	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Binomial Heap: n Inserts

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

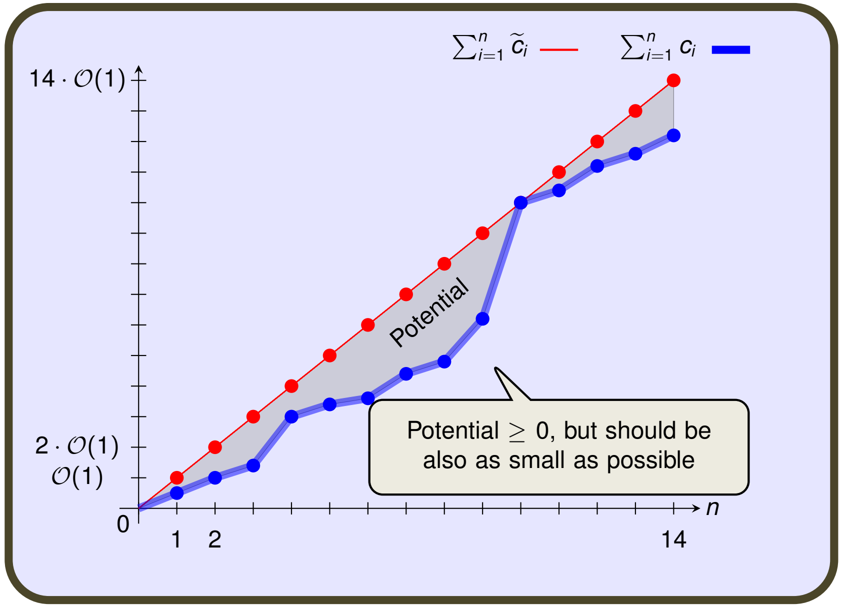
$\Rightarrow \sum_{i=1}^{n} c_i = \mathcal{O}(n \log n)$

Fibonacci Heap: n Inserts

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

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

Actual vs. Amortized Cost



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	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
UNION	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Can we perform EXTRACT-MIN better than $\mathcal{O}(\log n)$?

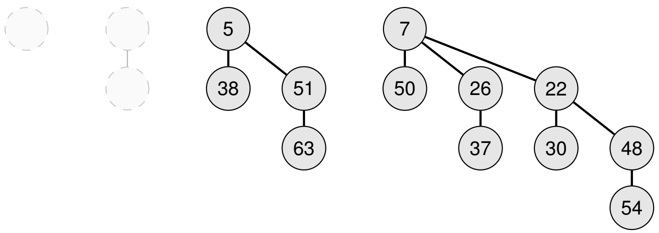
If this was possible, then there would be a sorting algorithm with runtime $o(n \log n)$!

Outline

Binomial Heap vs. Fibonacci Heap: Structure

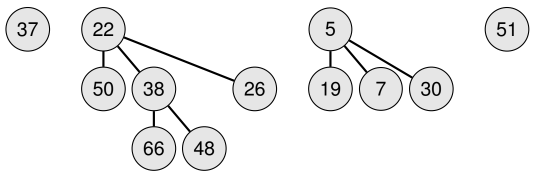
Binomial Heap:

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



Fibonacci Heap:

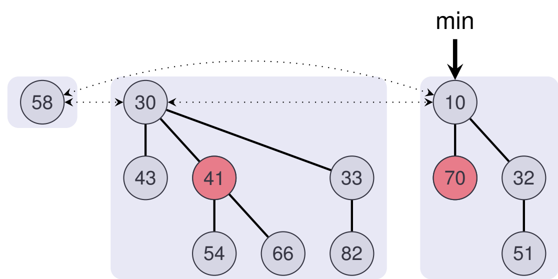
- forest of MIN-HEAPs
- lazily defer tidying up; do it on-the-fly when search for the MIN



Structure of Fibonacci Heaps

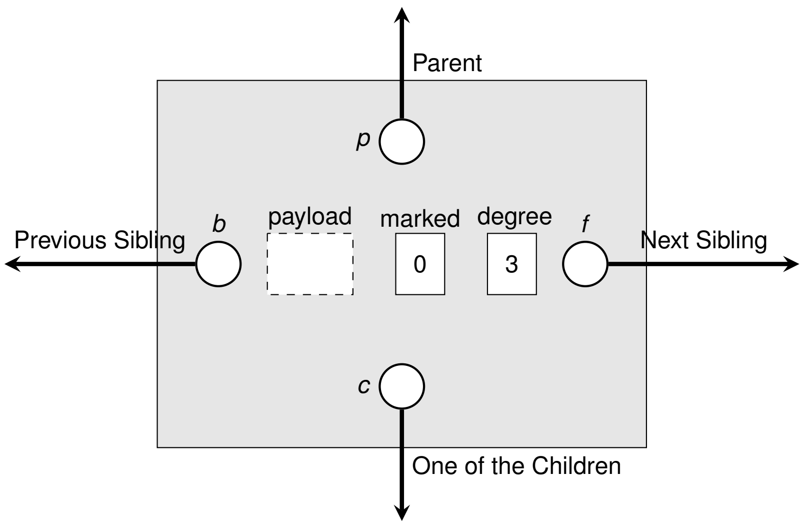
Fibonacci Heap

- Forest of MIN-HEAPs
- Nodes can be marked
  [slides/handout: roots are always unmarked, CLRS: roots can be marked, but have to be unmarked once they become a child.]
- Tree roots are stored in a circular, doubly-linked list
- Min-Pointer pointing to the smallest element

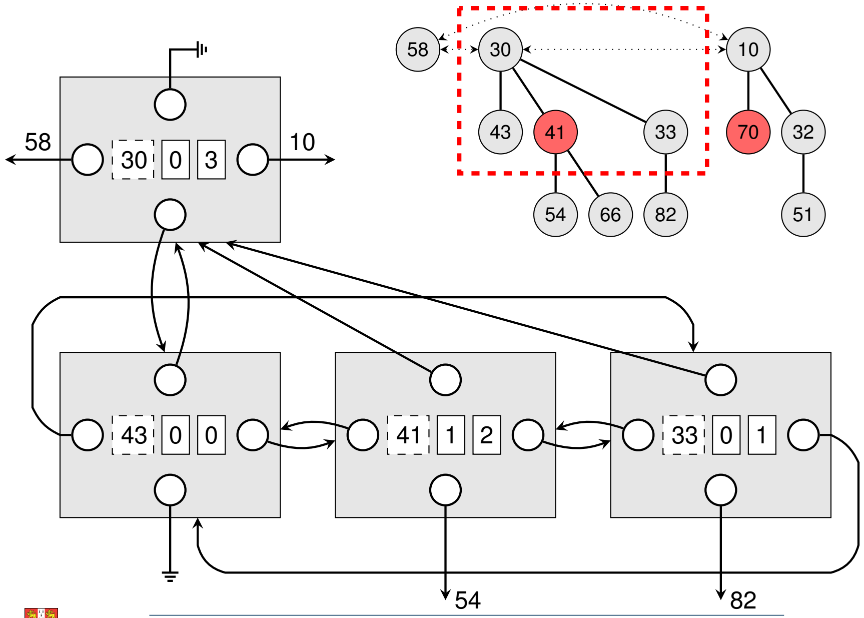


How do we implement a Fibonacci Heap?

A single Node



Magnifying a Four-Node Portion



Outline

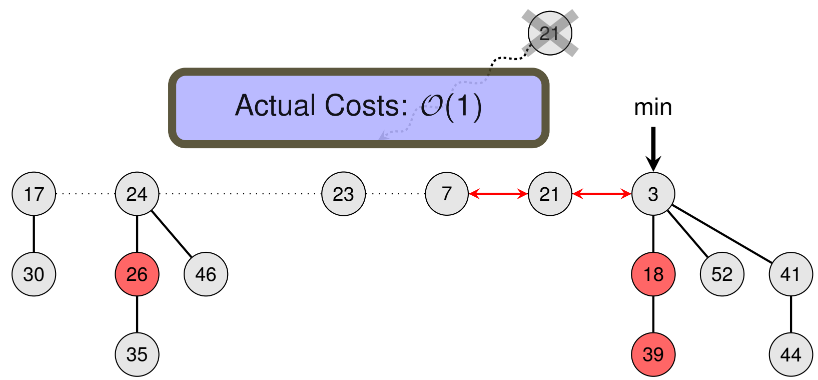
Structure

Operations

Glimpse at the Analysis

Fibonacci Heap: INSERT

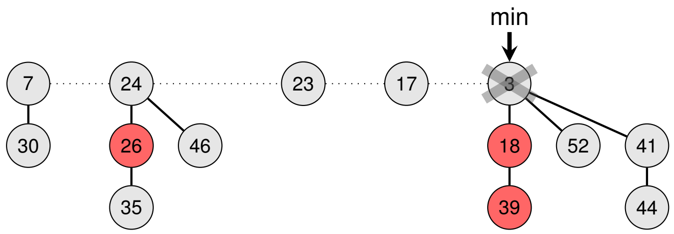
INSERT
- Create a singleton tree
- Add to root list and update min-pointer (if necessary)



Fibonacci Heap: Extract-Min

Extract-Min

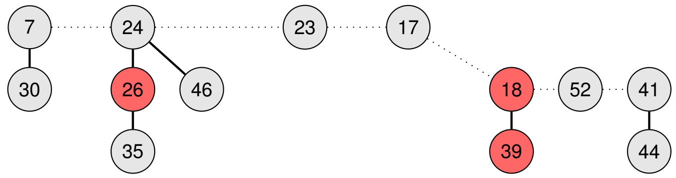
- Delete min



Fibonacci Heap: Extract-Min

Extract-Min

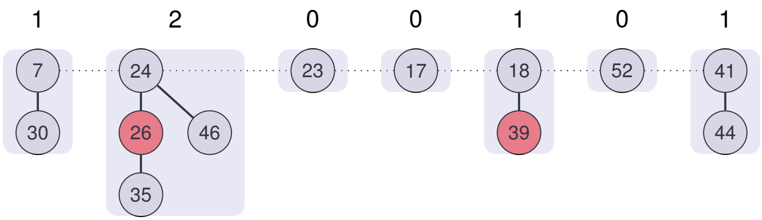
- Delete min ✓
- Meld childen into root list and unmark them



Fibonacci Heap: Extract-Min

Extract-Min

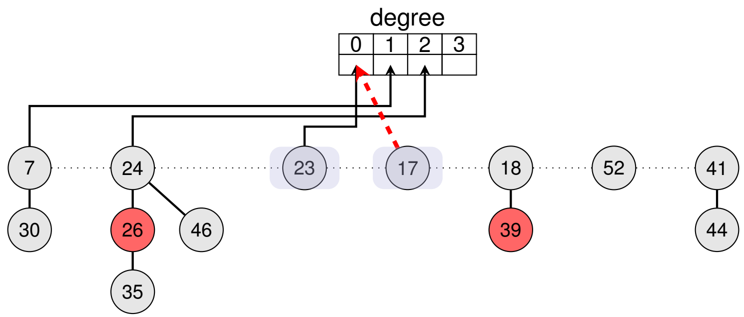
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: Extract-Min

Extract-Min

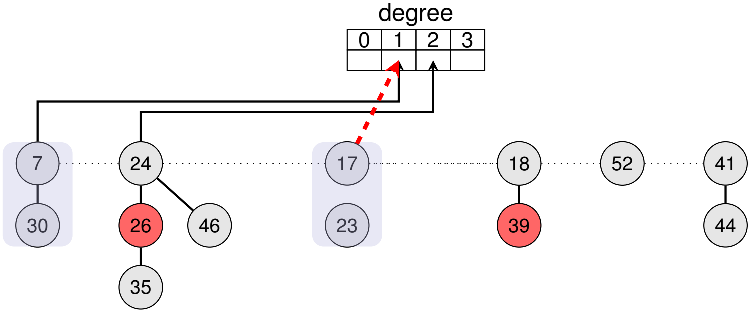
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

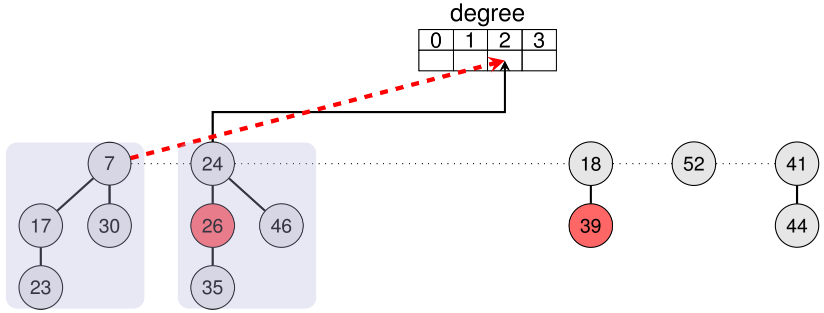
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

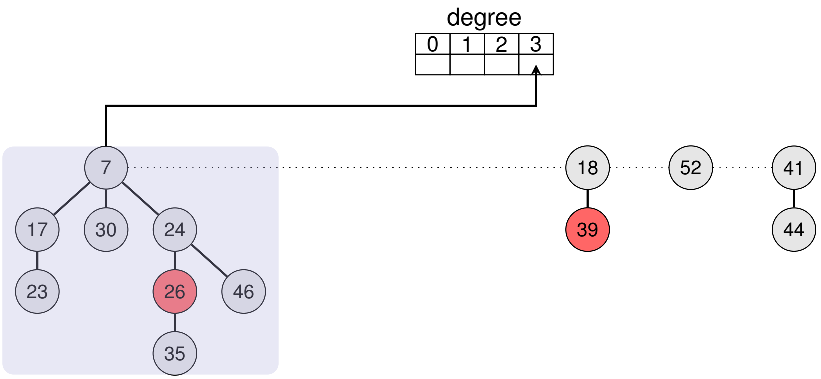
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

EXTRACT-MIN

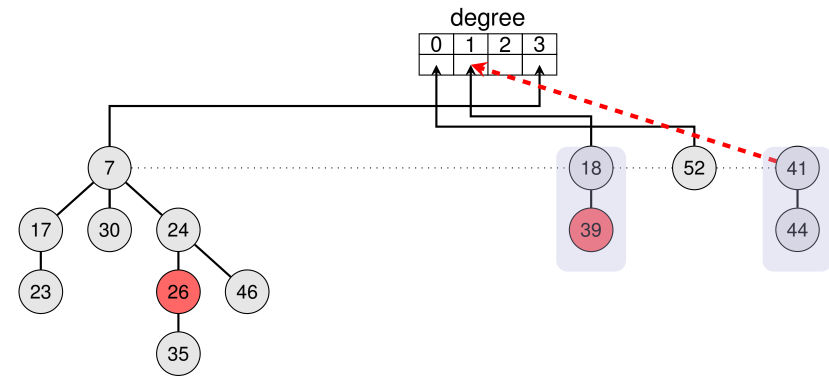
- Delete min ✓
- Meld childen into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)



Fibonacci Heap: EXTRACT-MIN

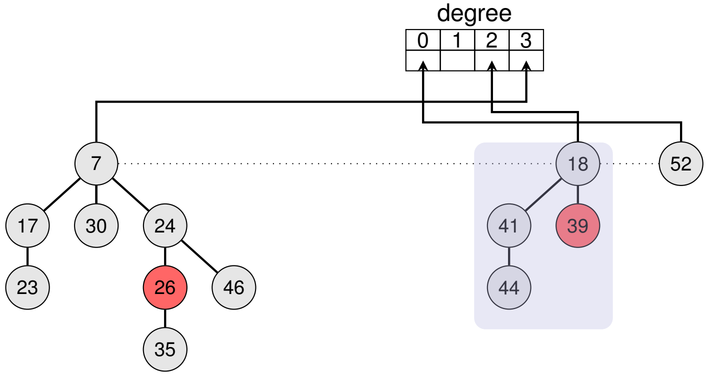
EXTRACT-MIN

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



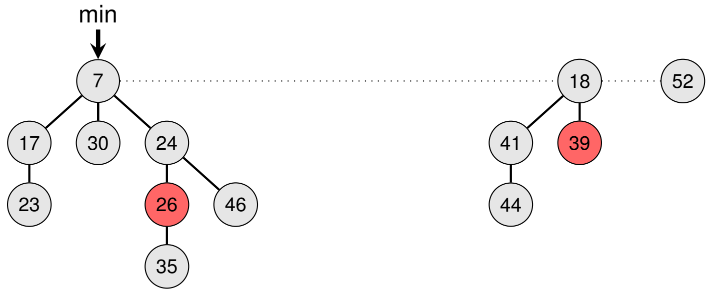
Fibonacci Heap: EXTRACT-MIN

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



Fibonacci Heap: EXTRACT-MIN

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

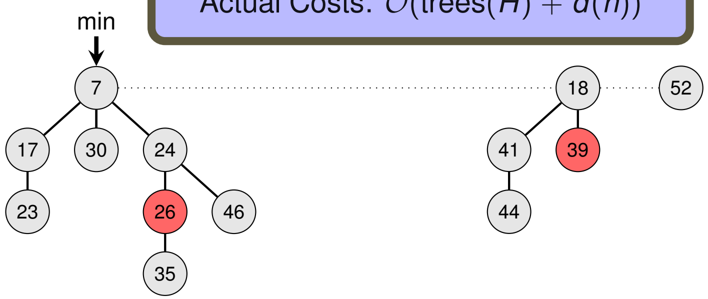


Fibonacci Heap: EXTRACT-MIN

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

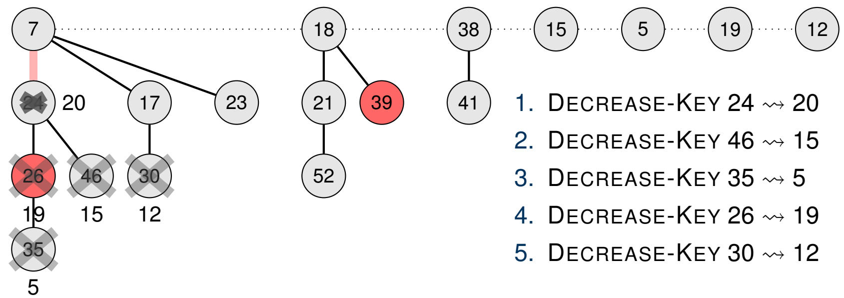
Every root becomes child of another root at most once!

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



Fibonacci Heap: DECREASE-KEY (First Attempt)

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.

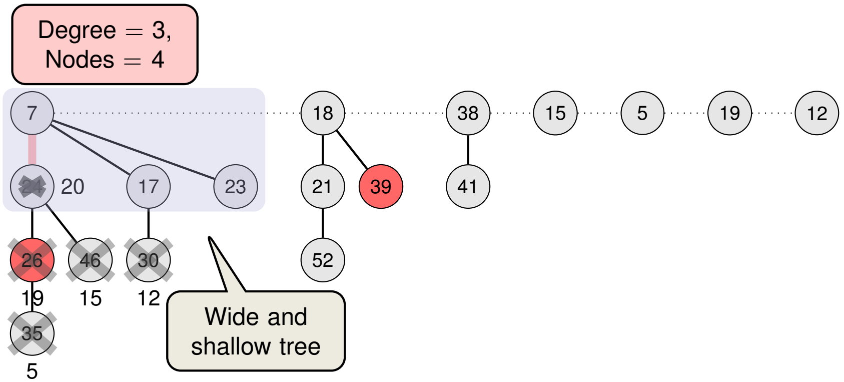


1. DECREASE-KEY 24 ⇝ 20
2. DECREASE-KEY 46 ⇝ 15
3. DECREASE-KEY 35 ⇝ 5
4. DECREASE-KEY 26 ⇝ 19
5. DECREASE-KEY 30 ⇝ 12

Fibonacci Heap: DECREASE-KEY (First Attempt)

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.

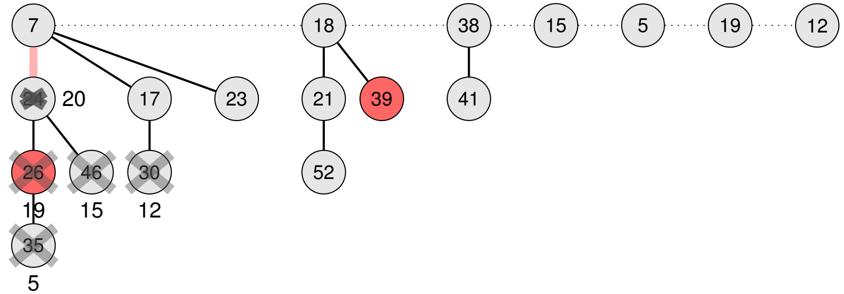


Fibonacci Heap: DECREASE-KEY (First Attempt)

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.

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



Fibonacci Heap: DECREASE-KEY

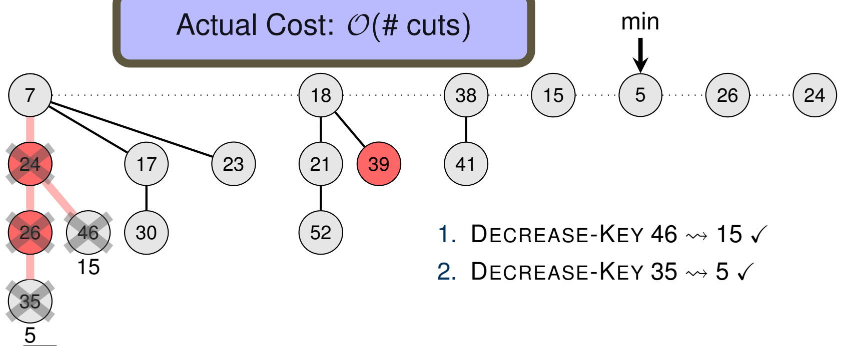
DECREASE-KEY of node x

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

$\Rightarrow$ Cut tree rooted at x, unmark x, and 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 ✓

Outline

Structure

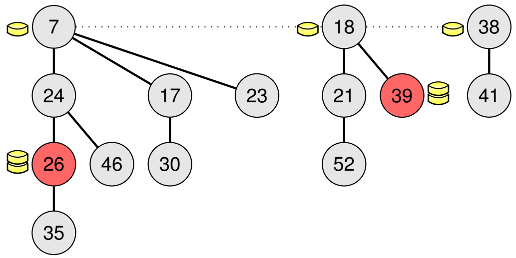
Operations

Glimpse at the Analysis

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 $\mathcal{O}(\#\text{ cuts})$ amortized $\mathcal{O}(1)$

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



Lent 2015

5.2 Fibonacci Heaps (Analysis)

Frank Stajano Thomas Sauerwald

UNIVERSITY OF CAMBRIDGE

Outline

Amortized Analysis

Bounding the Maximum Degree

Amortized Analysis of DECREASE-KEY

Actual Cost

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

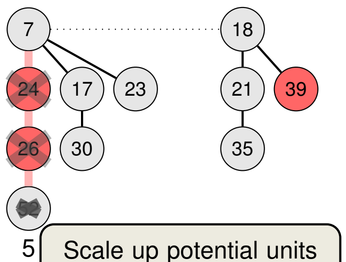
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

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

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

Amortized Analysis of EXTRACT-MIN

Actual Cost

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

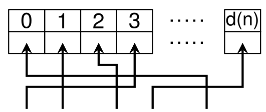
$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Change in Potential

- $\text{marks}(H') \leq \text{marks}(H)$
  $\text{trees}(H') \leq d(n) + 1$

$\Rightarrow$ $\Delta\Phi = d(n) + 1 - \text{trees}(H)$

degrees



Amortized Cost

$$\widetilde{c}_i = c_i + \Delta\Phi = \mathcal{O}(\text{trees(H)} + d(n)) + d(n) + 1 - \text{trees(H)} = \mathcal{O}(d(n))$$

How to bound $d(n)$?

Outline

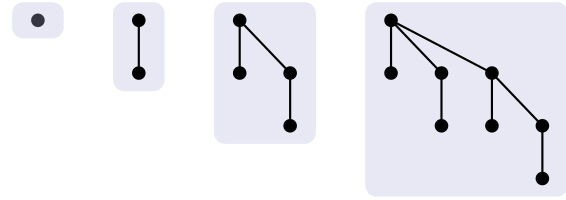
Amortized Analysis

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 $\varphi \approx 1.62$.

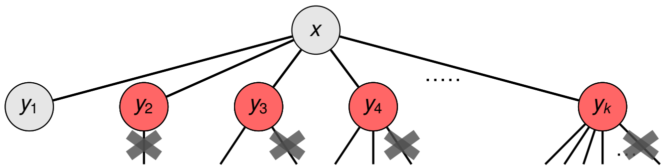
Lower Bounding Degrees of Children

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

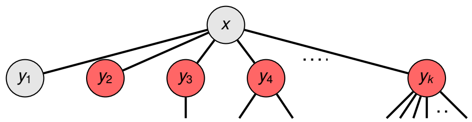
$$d(n) \leq \log_\varphi n$$

- Consider any node x (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



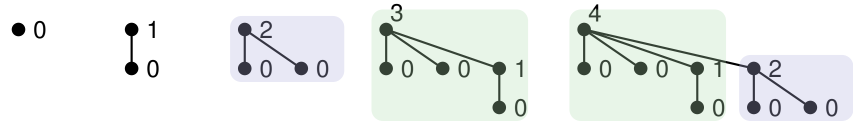
$$\forall 1 \leq i \leq k: \quad d_i \geq i - 2$$

Definition

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

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

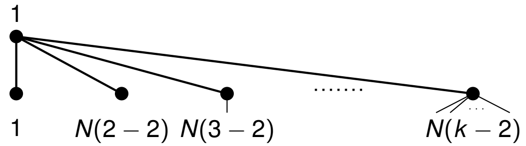
$N(0) = 1$ $N(1) = 2$ $N(2) = 3$ $N(3) = 5$ $N(4) = 8 = 5 + 3$



From Minimum Subtree Sizes to Fibonacci Numbers

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

$N(k) =$



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

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 $\varphi^1 < 1.619$ ✓
- 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 $\cancel{\mathcal{O}(d(n))}$ $\mathcal{O}(\log n)$
- DECREASE-KEY amortized cost $\mathcal{O}(1)$

$$\begin{aligned} & n \geq N(k) = F(k+2) \geq \varphi^k \\ \Rightarrow \quad & \log_\varphi n \geq k \end{aligned}$$

Summary

- Fibonacci Heaps were developed by Fredman and Tarjan in 1984
- Fibonacci Numbers were discovered >800 years ago

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

Outlook: A More Efficient Priority Queue

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 2015

UNIVERSITY OF CAMBRIDGE

Outline

Disjoint Sets

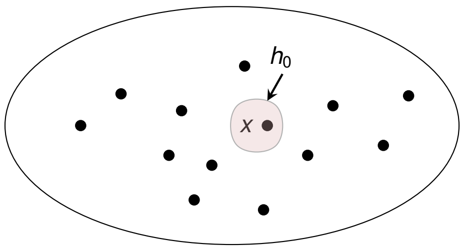
Graph Representations

Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle makeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

h_0=**`makeSet(x)`**

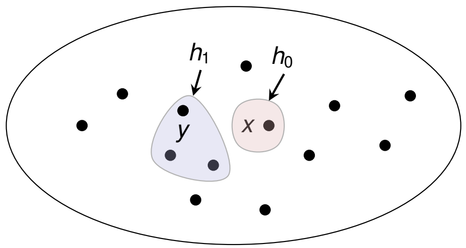


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle makeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle findSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

h_1=**`findSet(y)`**

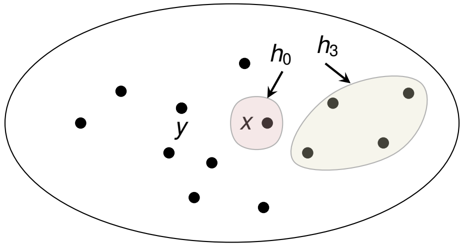


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle makeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle findSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union`(h_0, h_3)

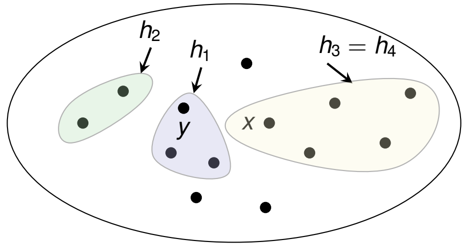


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle makeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle findSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union`(h_1, h_2)

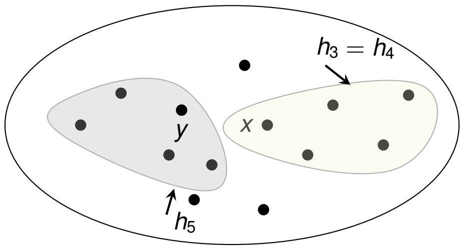


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle makeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle findSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union`(h_1, h_2)



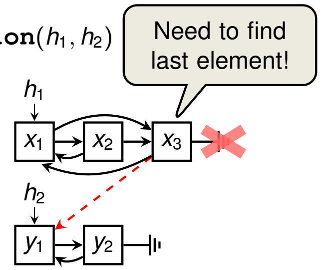
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

`Union`(h_1, h_2)



First Attempt: List Implementation

Union-Operation

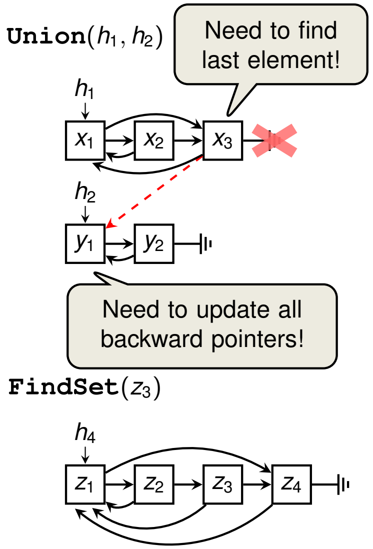
- Add extra pointer to the last element in each list

$\Rightarrow$ ~~Union takes constant time~~

Find-Operation

- Add backward pointer to the list head from everywhere

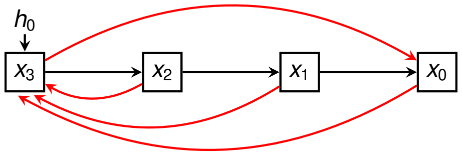
$\Rightarrow$ Find takes constant time



First Attempt: List Implementation (Analysis)

d = DisjointSet()
h_0 = d.MakeSet(x_0)

h_1 = d.MakeSet(x_1)
h_0 = d.union(h_1, h_0)
h_2 = d.MakeSet(x_2)
h_0 = d.union(h_2, h_0)
h_3 = d.MakeSet(x_3)
h_0 = d.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)

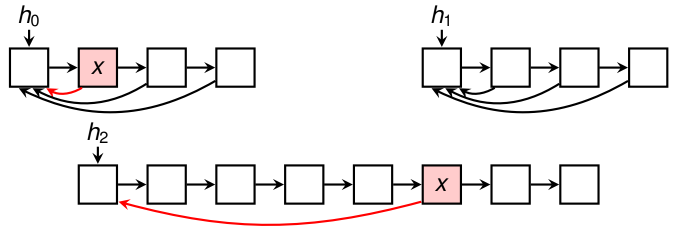
can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET 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 MAKE-SET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Can we improve on this further?

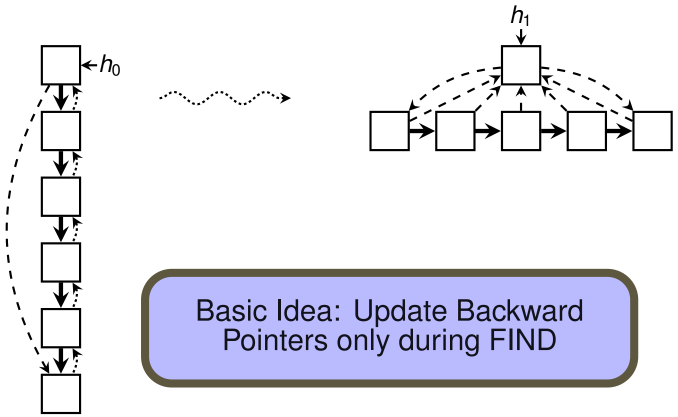
Proof:

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

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

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

How to Improve?

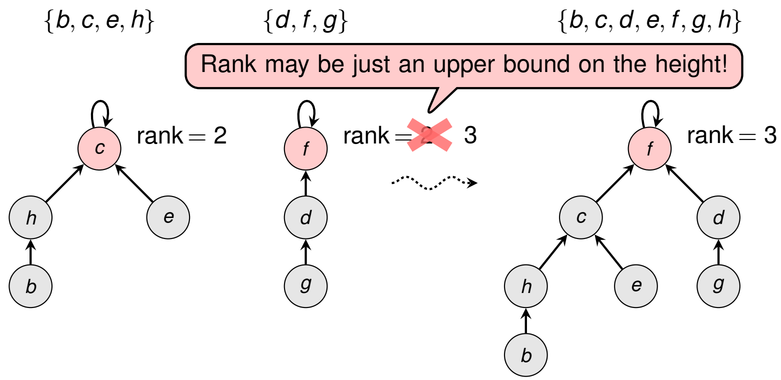


Basic Idea: Update Backward Pointers only during FIND

Doubly-Linked List
- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic
- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

Disjoint Sets via Forests

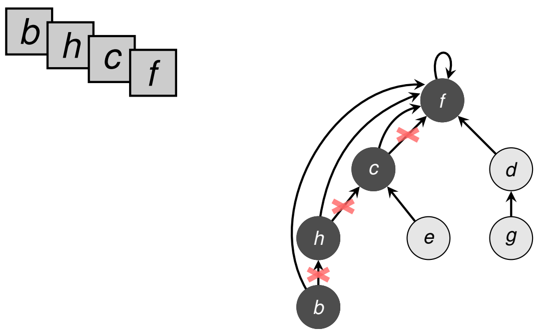


Rank may be just an upper bound on the height!

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 FIND-SET



```
FIND-SET(x)
  if x ≠ x.p
     x.p = FIND-SET(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 MAKE-SET, UNION, FIND-SET operations, n of which are MAKE-SET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

In practice, $\alpha(n)$ is a small constant

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

More than the number of atoms in the universe!

6.1 & 6.2: Graph Searching

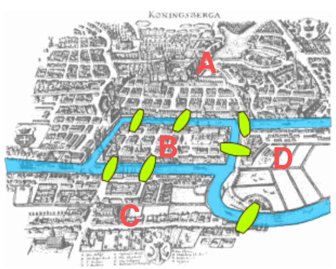
Frank Stajano   Thomas Sauerwald

28 February 2014

UNIVERSITY OF CAMBRIDGE

Outline

Origin of Graph Theory

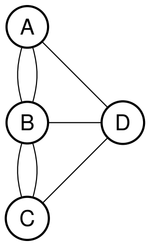


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

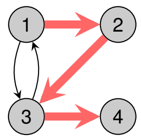
Is there a tour which visits every island **exactly once**?
$\rightsquigarrow$ 1B course: Complexity Theory

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:
- V: the set of vertices
- E: the set of edges (arcs)

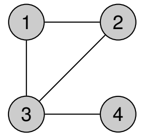
Path $p = (1, 2, 3, 4)$



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

Undirected Graph

A graph $G = (V, E)$ consists of:
- V: the set of vertices
- E: the set of (undirected) edges



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

Paths and Connectivity

- A sequence of edges between two vertices forms a path

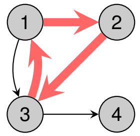
What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:
- V: the set of vertices
- E: the set of edges (arcs)

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

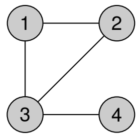
G is not a DAG



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

Undirected Graph

A graph $G = (V, E)$ consists of:
- V: the set of vertices
- E: the set of (undirected) edges



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

Paths and Connectivity

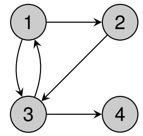
- A sequence of edges between two vertices forms a path

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:
- V: the set of vertices
- E: the set of edges (arcs)

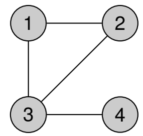
G is not (strongly) connected



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

Undirected Graph

A graph $G = (V, E)$ consists of:
- V: the set of vertices
- E: the set of (undirected) edges



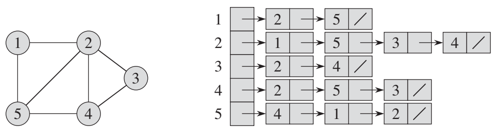
G is connected

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

Paths and Connectivity

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

Representations of Directed and Undirected Graphs

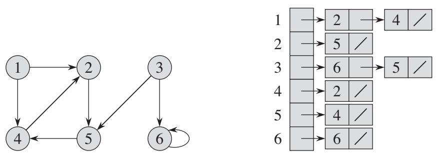


(a) (b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

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



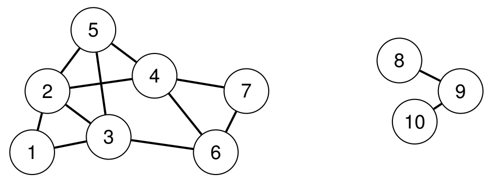
(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

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

Graph Searching



Overview

- Graph searching means taversing 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

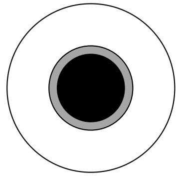
Introduction to Graphs and Graph Searching

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:

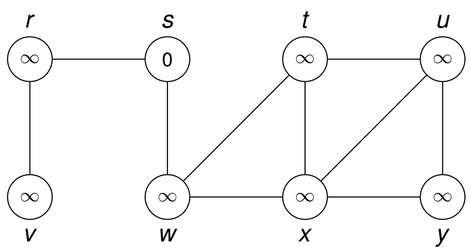
  White = Unvisited

  Grey = Visited, but not all neighbors (=adjacent vertices)

  Black = Visited and all neighbors

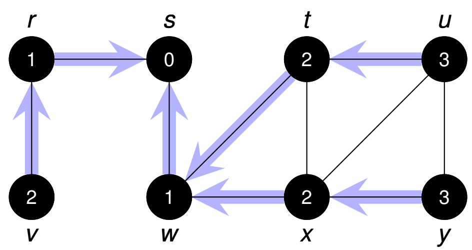
Complete Execution of BFS (Figure 22.3)

Queue:



Complete Execution of BFS (Figure 22.3)

Queue: ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~



Breadth-First-Search: Pseudocode

```
def bfs(G)
   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"
```

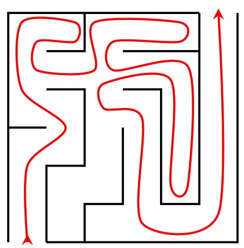
- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  White = Unvisited
  Grey = Visited, but not all neighbors
  Black = Visited and all neighbors
- Runtime $O(V + E)$

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

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

Outline

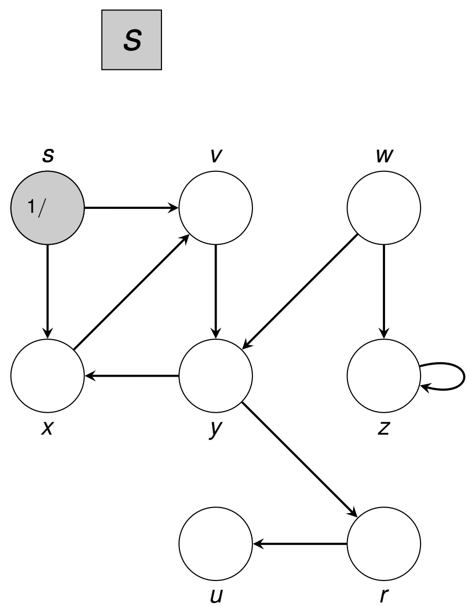
Depth-First Search: Basic Ideas



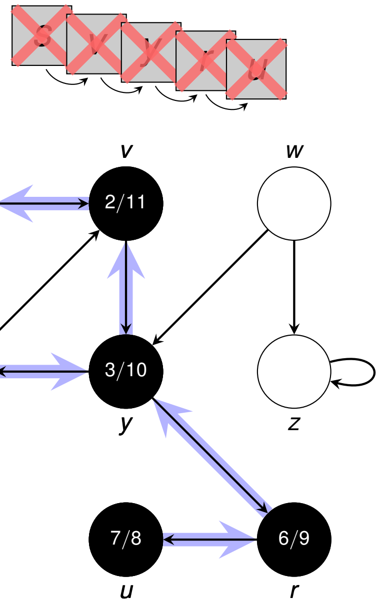
Basic Idea

- 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

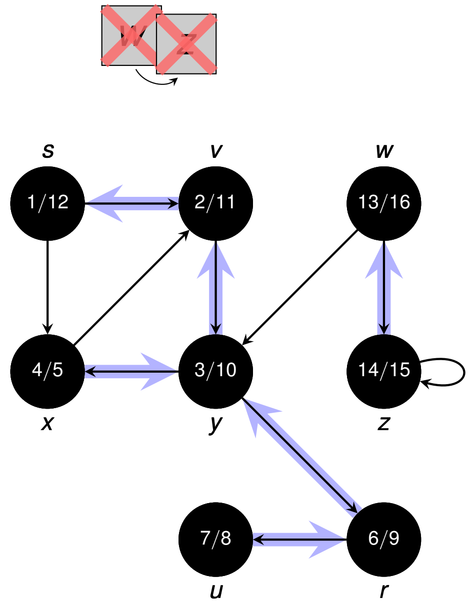
Complete Execution of DFS



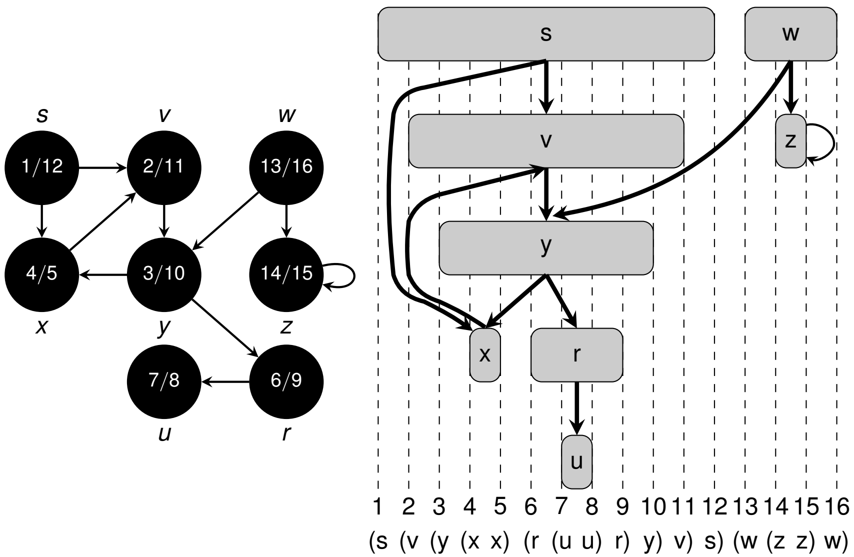
Complete Execution of DFS



Complete Execution of DFS



Paranthesis Theorem (Theorem 22.7)



Depth-First-Search: Pseudocode

```
def dfs(G):
    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 *.v*
- Vertex Colours:
  - White = Unvisited
  - Grey = Visited, but not all neighbors
  - Black = Visited and all neighbors
- Runtime $O(V + E)$

Outline

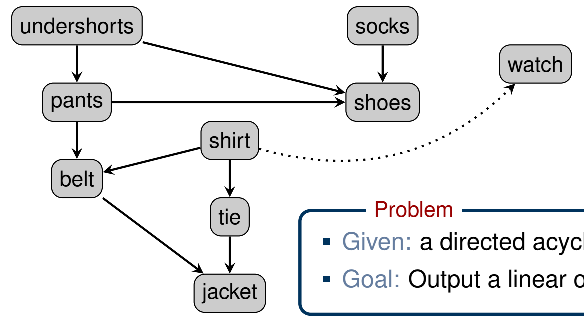
Introduction to Graphs and Graph Searching

Breadth-First Search

Depth-First Search

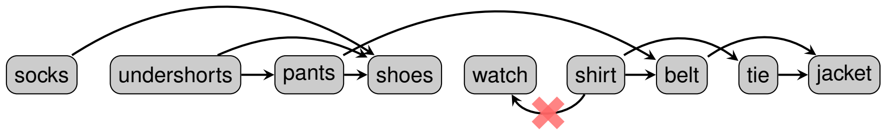
Topological Sort

Topological Sort

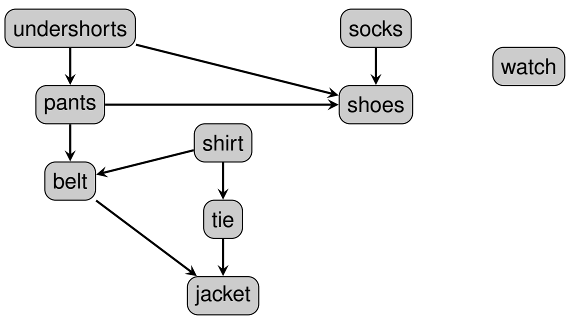


Problem

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



Solving Topological Sort



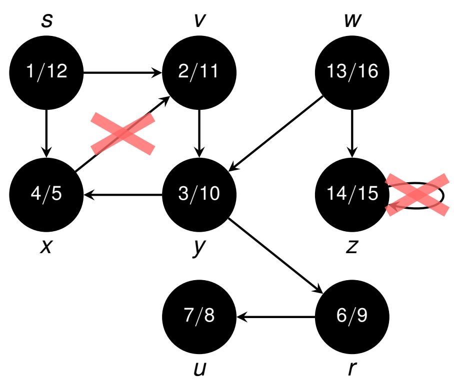
Knuth's Algorithm (1968)

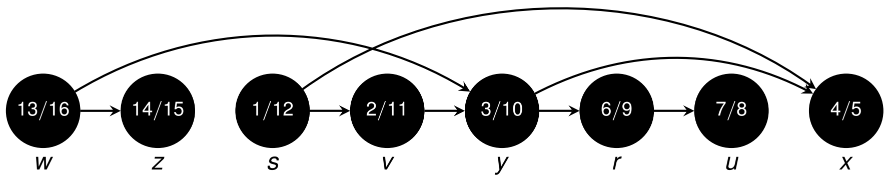
- Perform DFS's so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Runtime $O(V + E)$

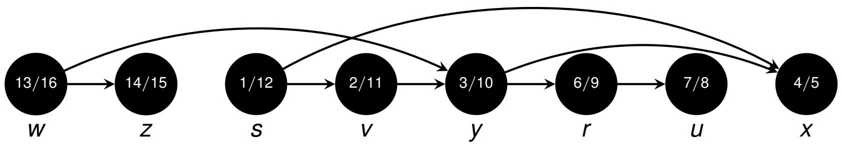
Don't need to sort the vertices – use DFS directly!

Execution of Knuth's Algorithm





Correctness of Topological Sort using DFS



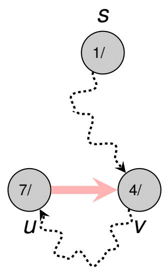
Theorem 22.12

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

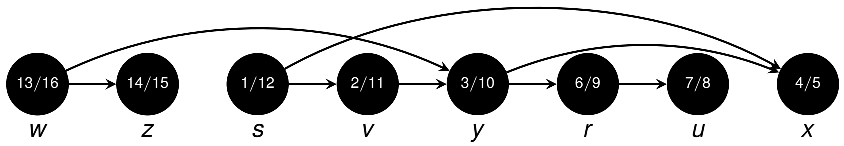
Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).



Correctness of Topological Sort using DFS



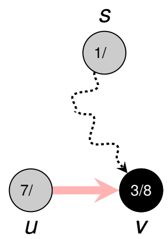
Theorem 22.12

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

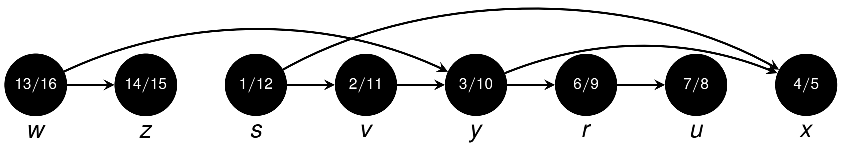
Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*



Correctness of Topological Sort using DFS



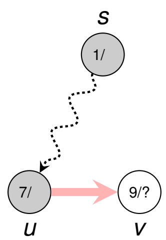
Theorem 22.12

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

Proof:

- Consider any edge $(u, v) \in E(G)$ being explored,
  $\Rightarrow$ u is grey and we have to show that $v.f < u.f$

  1. *If v is grey, then there is a cycle*
     (can't happen, because G is acyclic!).
  2. *If v is black, then $v.f < u.f$.*
  3. *If v is white, we call DFS(v) and $v.f < u.f$.*

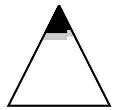


$\Rightarrow$ In all cases $v.f < v.u$, so v appears after u. □

Summary of Graph Searching

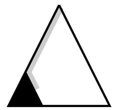
Breadth-First-Search

- 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 2015

UNIVERSITY OF CAMBRIDGE

Outline

Minimum Spanning Tree Problem

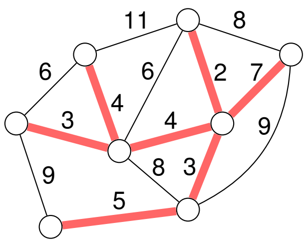
Single-Source Shortest Path

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

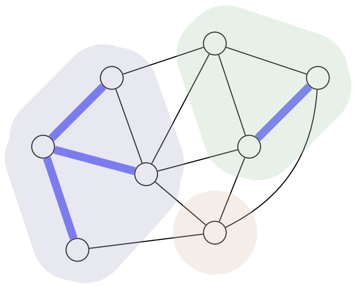
A edge of G is safe if by adding the edge to A, the resulting subgraph is still a subset of a minimum spanning tree.

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.

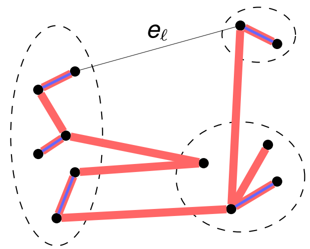
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done



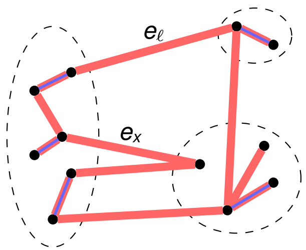
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x



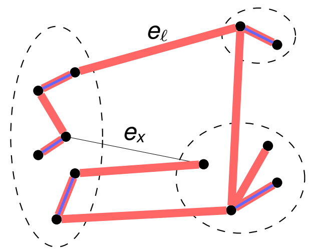
Proof of Theorem

Theorem

Let $A \subseteq E$ be a subset of a MST of G. Then for any cut that respects A, the lightest edge of G that goes across the cut is safe.

Proof:

- Let T be a MST containing A
- Let e_ℓ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding e_ℓ to T introduces cycle
- This cycle crosses the cut through e_ℓ and another edge e_x
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree
  - If $w(e_\ell) < w(e_x)$, then this spanning tree has smaller cost than T (can't happen!)
  - If $w(e_\ell) = w(e_x)$, then $T \cup e_\ell \setminus e_x$ is a MST. $\square$

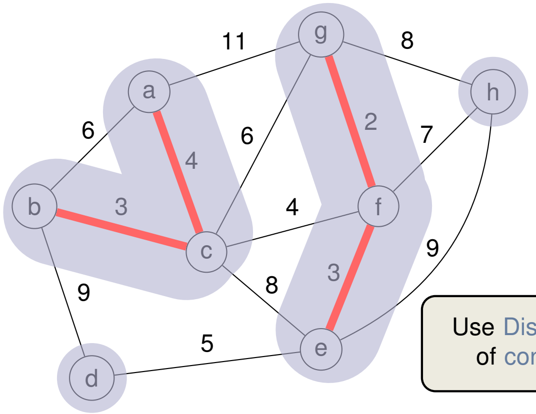


Glimpse at Kruskal's Algorithm

Basic Strategy

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

  Add lightest edge to A that does not introduce a cycle



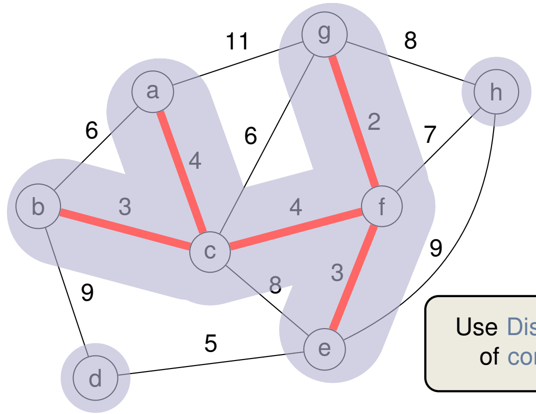
Use Disjoint Sets to keep track of connected components!

Glimpse at Kruskal's Algorithm

Basic Strategy

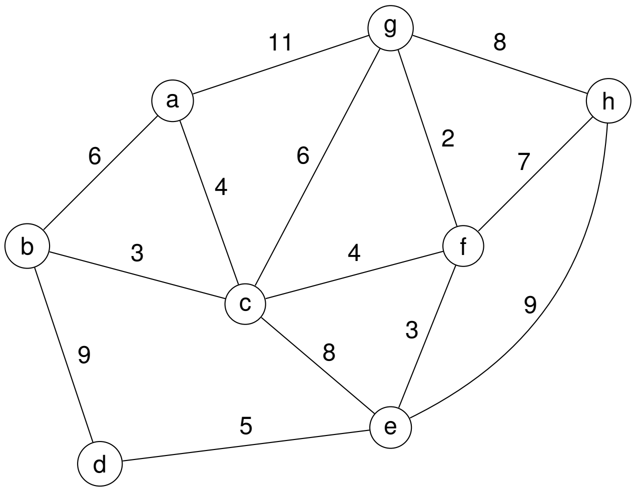
- Let $A \subseteq E$ be a forest, intially empty
- At every step, given A, perform:

  Add lightest edge to A that does not introduce a cycle

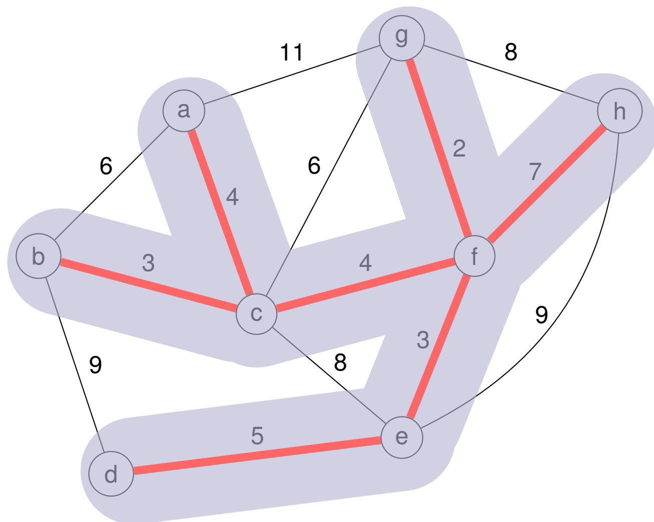


Use Disjoint Sets to keep track of connected components!

Complete Run of Kruskal's Algorithm



Complete Run of Kruskal's Algorithm



Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
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

- Initialization (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 $O(E \cdot \alpha(n))$!

Prim's Algorithm

Basic Strategy

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

Assign every vertex not connected to 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

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

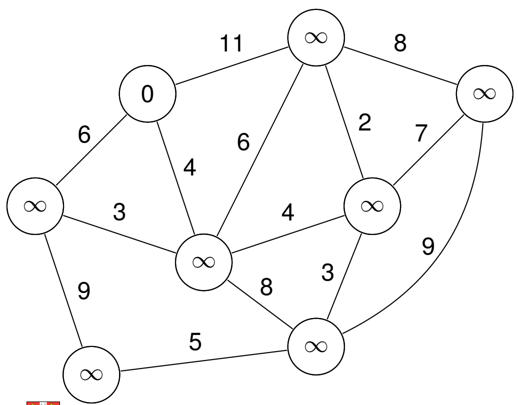
Implementation

- 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

Prim's Algorithm

Basic Strategy

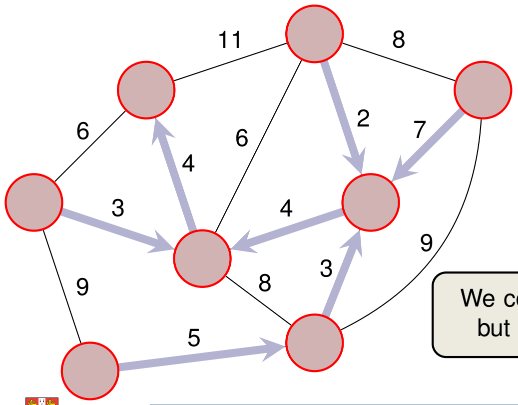
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Prim's Algorithm

Basic Strategy

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



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=v)
```

Time Complexity

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

6.3: Minimum Spanning Tree T.S. 11

6.4: Single-Source Shortest Paths

Frank Stajano Thomas Sauerwald

Lent 2015

UNIVERSITY OF CAMBRIDGE

Outline

Introduction

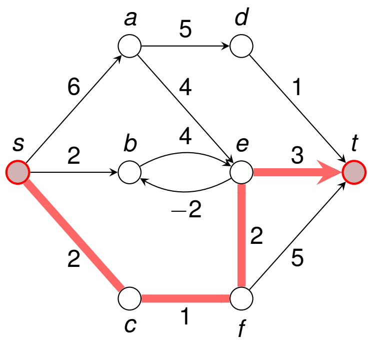
Bellman-Ford Algorithm

Dijkstra's Algorithm

Shortest Path Problem

Shortest Path Problem

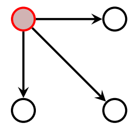
- Given: directed graph $G = (V, E)$ with edge weights, pair of vertices $s, t \in V$
- Goal: Find a path of minimum weight from s to t in G



Variants of Shortest Path Problems

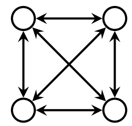
Single-source shortest-paths problem (SSSP)

- Bellman-Ford Algorithm
- Dijsktra Algorithm

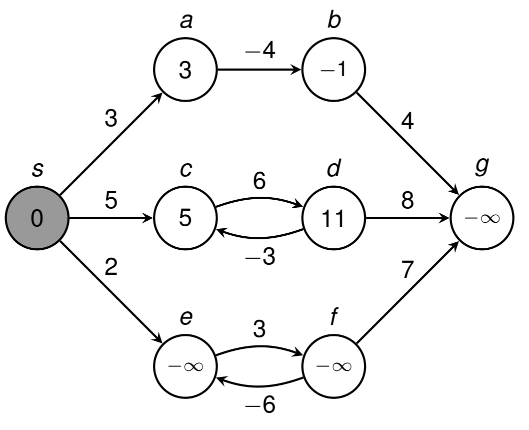


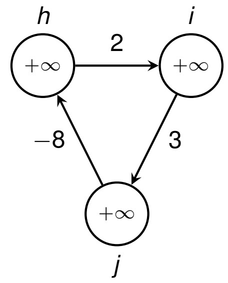
All-pairs shortest-paths problem (APSP)

- Shortest Paths via Matrix Multiplication
- Johnson's Algorithm



Distances and Negative-Weight Cycles





Relaxing Edges

Definition

Fix the source vertex $s \in V$

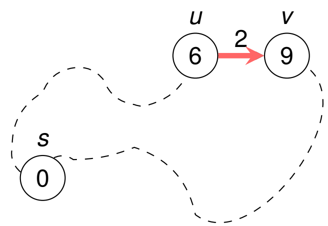
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



Relaxing Edges

Definition

Fix the source vertex $s \in V$

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

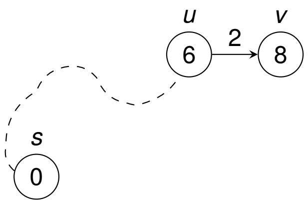
- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$

After relaxing (u, v):
$v.d \leq u.d + w(u, v)$



Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

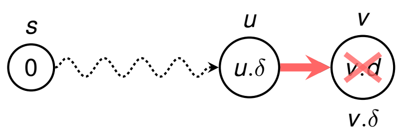
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

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 $v.d = v.\delta$. □

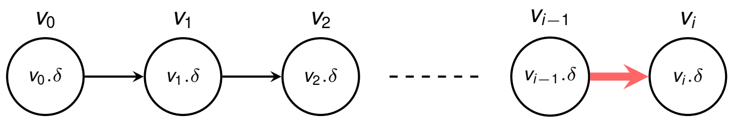
Path-Relaxation Property

Path-Relaxation Property (Lemma 24.15)

If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = v_k.\delta$ (regardless of the order of other relaxation steps).

Proof:

- By induction on i, $0 \leq i \leq k$:
  After the ith edge of p is relaxed, we have $v_i.d = v_i.\delta$.
- For $i = 0$, by the initialization $s.d = s.\delta = 0$.
  Upper-bound Property $\Rightarrow$ the value of $s.d$ never changes after that.
- Inductive Step ($i - 1 \to i$): Assume $v_{i-1}.d = v_{i-1}.\delta$ and relax (v_{i-1}, v_i).
  Convergence Property $\Rightarrow$ $v_i.d = v_i.\delta$ (now and at all later steps) $\square$



The Bellman-Ford Algorithm

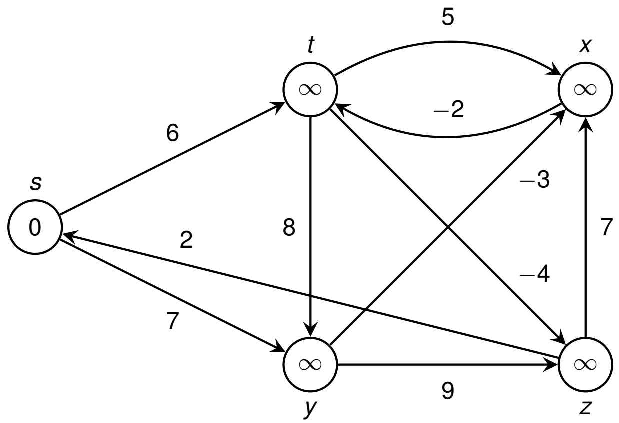
```
BELLMAN-FORD(G,w,s)
0: assert(s in G.vertices())
1: for v in G.vertices()
2:    v.predecessor = None
3:    v.d = Infinity
4: s.d = 0
5:
6: repeat |V|-1 times
7:    for e in G.edges()
8:    Relax edge e=(u,v): Check if u.d + w(u,v) < v.d
9:    if e.start.d + e.weight.d < e.end.d:
10:       e.end.d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

Complete Run of Bellman-Ford (Figure 24.4)

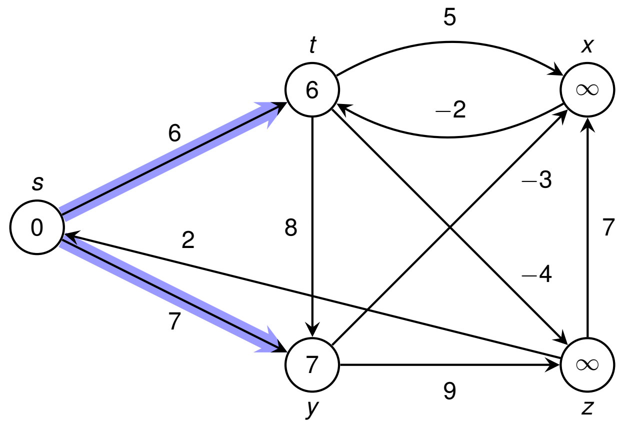
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 1

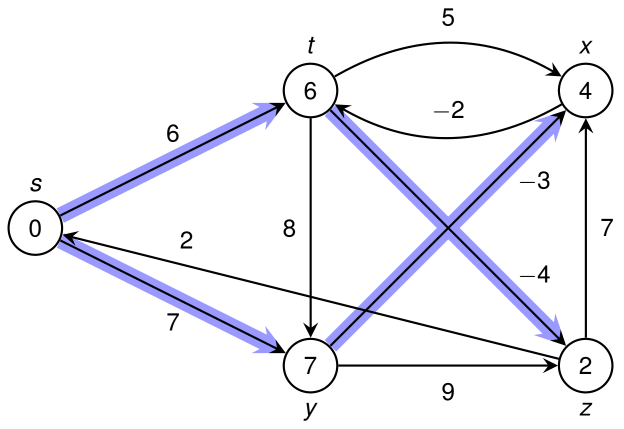
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 2

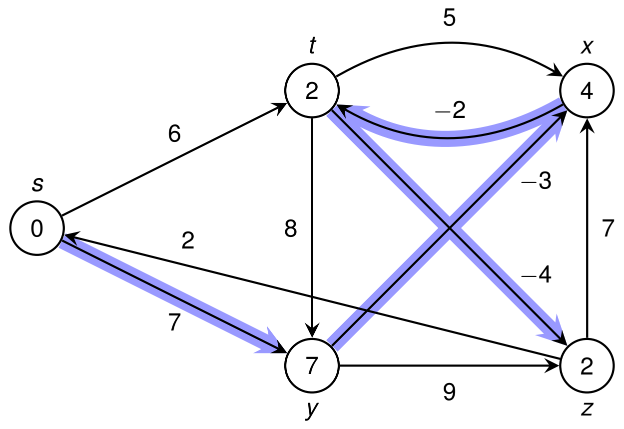
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 3

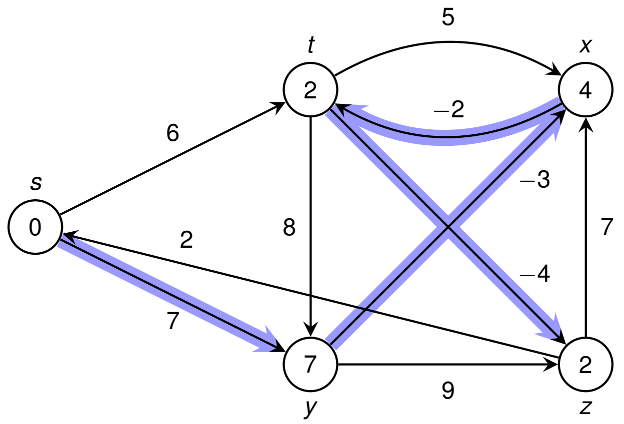
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 4

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



The 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

- Let $(u,v) \in E$ be any edge. After $|V|-1$ passes:

$$v.d = v.\delta \leq u.\delta + w(u,v) = u.d + w(u,v)$$

$\Rightarrow$ Bellman-Ford returns TRUE $\square$

The 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 $1 \leq i < k$,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

$$\Rightarrow \sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- This contradicts the assumption that c is a negative-weight cycle! $\square$

Outline

Relaxing Edges

Definition

Fix the source vertex $s \in V$

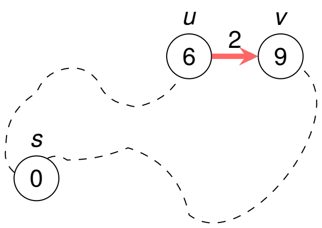
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



Relaxing Edges

Definition

Fix the source vertex $s \in V$

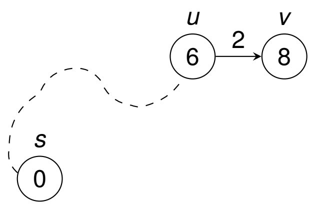
- $v.\delta$ is the length of the shortest path (distance) from s to v
- $v.d$ is the length of the shortest path discovered so far

- At the beginning: $s.d = s.\delta = 0$, $v.d = \infty$ for $v \neq s$
- At the end: $v.d = v.\delta$ for all $v \in V$

Relaxing an edge (u, v)

Given estimates $u.d$ and $v.d$, can we find a better path from v using the edge (u, v)?

$$v.d \overset{?}{>} u.d + w(u, v)$$



After relaxing (u, v), regardless of whether we found a shortcut: $v.d \leq u.d + w(u, v)$

Properties of Shortest Paths and Relaxations

Toolkit

Triangle inequality (Lemma 24.10)

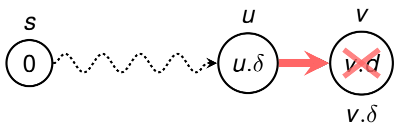
- For any edge $(u, v) \in E$, we have $v.\delta \leq u.\delta + w(u, v)$

Upper-bound Property (Lemma 24.11)

- We always have $v.d \geq v.\delta$ for all $v \in V$, and once $v.d$ achieves the value $v.\delta$, it never changes.

Convergence Property (Lemma 24.14)

- If $s \rightsquigarrow u \to v$ is a shortest path from s to v, and if $u.d = u.\delta$ prior to relaxing edge (u, v), then $v.d = v.\delta$ at all times afterward.



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

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

Path-Relaxation Property

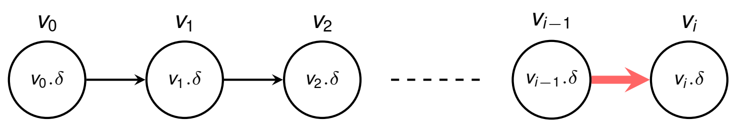
"Propagation": By relaxing proper edges, set of vertices with $v.\delta = v.d$ gets larger

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

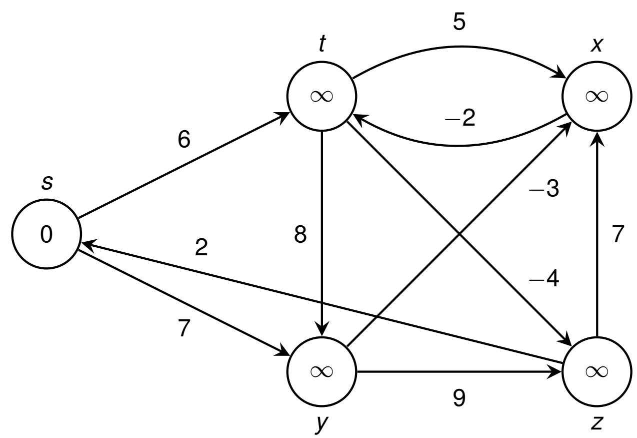
```
 BELLMAN-FORD(G,w,s)
0: assert(s in G.vertices())
1: for v in G.vertices()
2:    v.predecessor = None
3:    v.d = Infinity
4: s.d = 0
5:
6: repeat |V|-1 times
7:    for e in G.edges()
8:    Relax edge e=(u,v): Check if u.d + w(u,v) < v.d
9:    if e.start.d + e.weight.d < e.end.d:
10:       e.end.d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

Complete Run of Bellman-Ford (Figure 24.4)

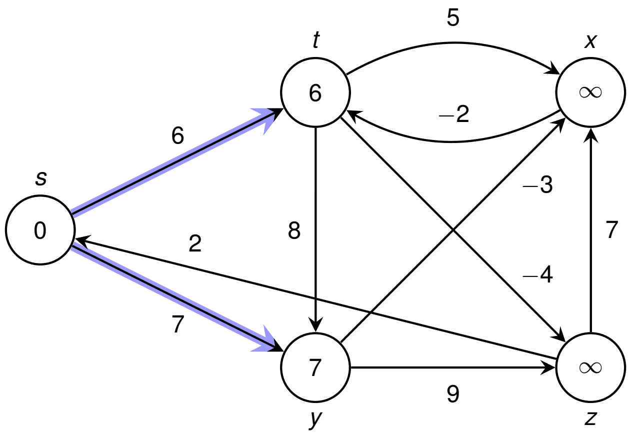
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 1

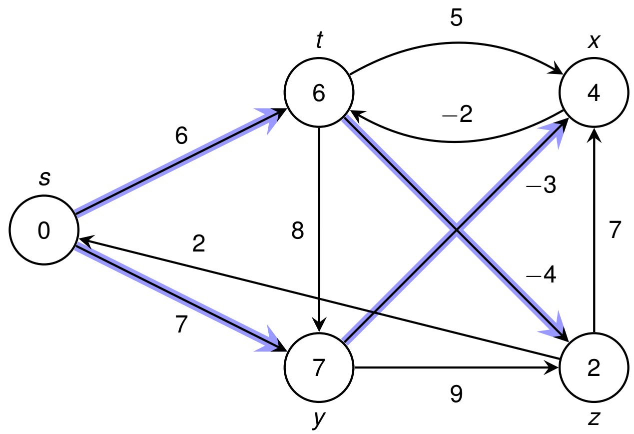
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 2

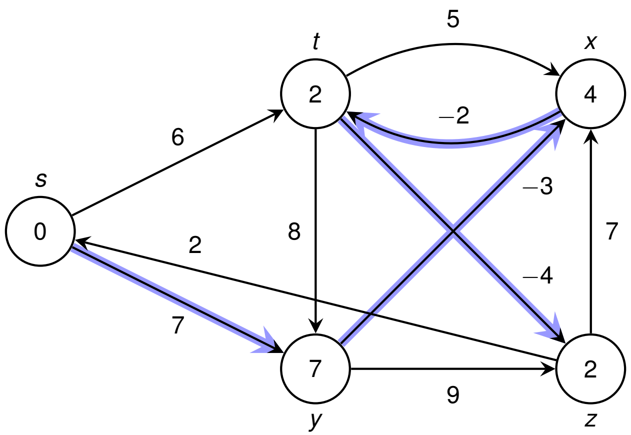
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 3

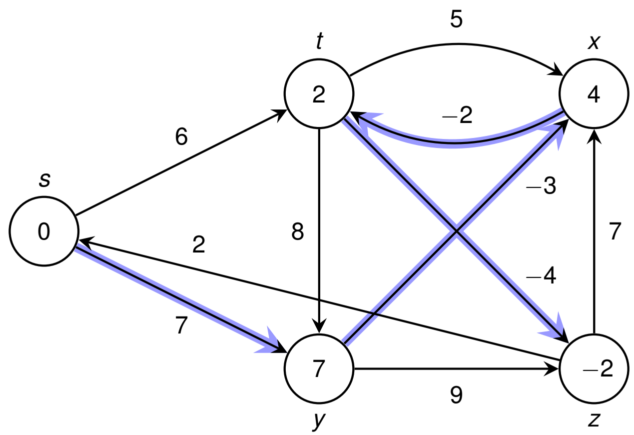
Relaxation Order: (t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)



Complete Run of Bellman-Ford (Figure 24.4)

Pass: 4

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



Bellman-Ford Algorithm: Correctness (1/2)

Lemma 24.2/Theorem 24.3

Assume that G contains no negative-weight cycles that are reachable from s. Then after $|V|-1$ passes, we have $v.d = v.\delta$ for all vertices $v \in V$ that are reachable and Bellman-Ford returns TRUE.

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 $1 \leq i < k$,

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

$$\Rightarrow \sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

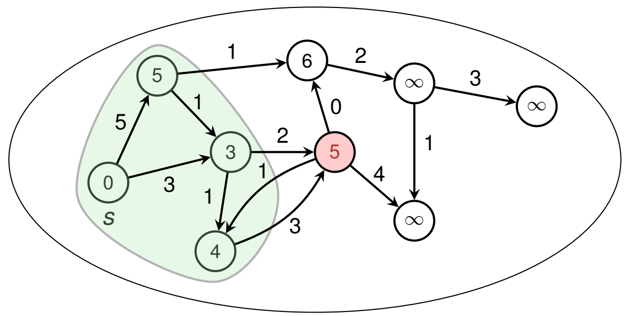
- This contradicts the assumption that c is a negative-weight cycle! $\square$

Outline

Dijkstra's Algorithm

Overview of Dijkstra

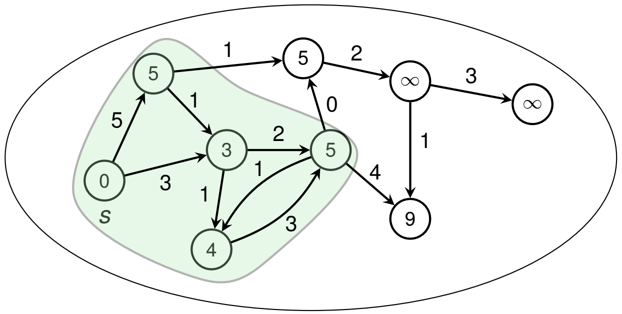
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = v.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$



Dijkstra's Algorithm

Overview of Dijkstra

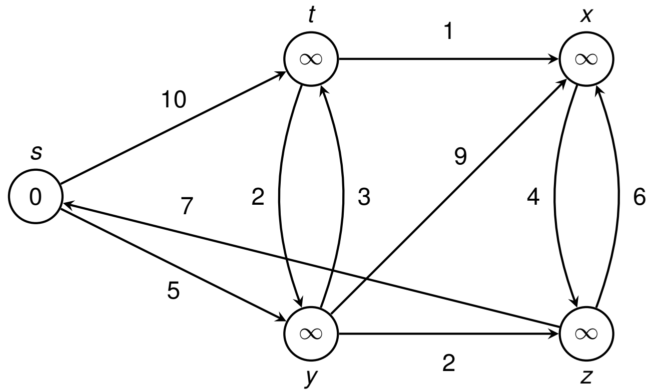
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = v.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3.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



Complete Run of Dijkstra (Figure 24.6)

Priority Queue Q:

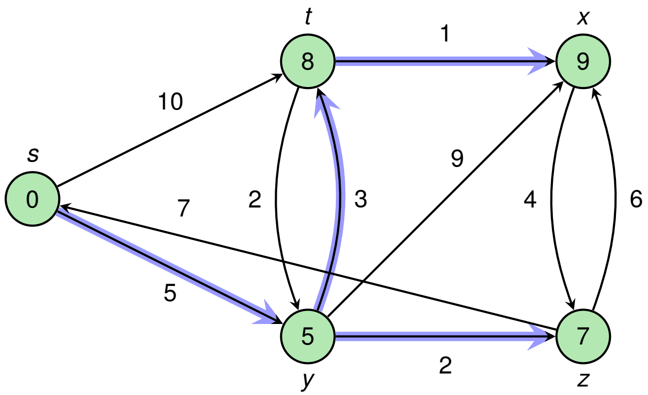
$(s,0),(t,\infty),(x,\infty),(y,\infty),(z,\infty)$



Complete Run of Dijkstra (Figure 24.6)

Priority Queue Q:

$\cancel{(x,9)}$



Runtime of Dijkstra's Algorithm

```
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 (using 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: $\mathcal{O}(V \log V + E)$

Correctness of Dijkstra's Algorithm

Theorem 24.6

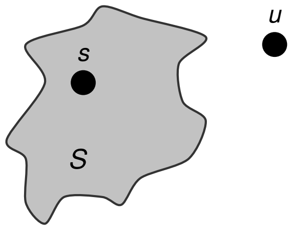
For any directed graph $G = (V, E, w)$ with non-negative edge weights and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

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

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

- Let u be the first vertex with this property



Correctness of Dijkstra's Algorithm

Theorem 24.6

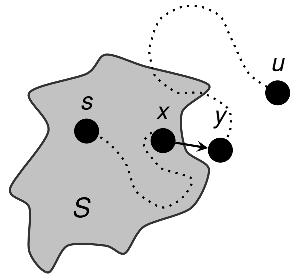
For any directed graph $G = (V, E, w)$ with non-negative edge weights and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- 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 $V \setminus S$



Correctness of Dijkstra's Algorithm

Theorem 24.6

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

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

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

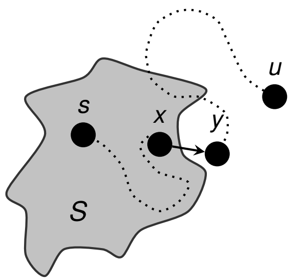
$$u.d > u.\delta$$

- 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 of Dijkstra's Algorithm

Theorem 24.6

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

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

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

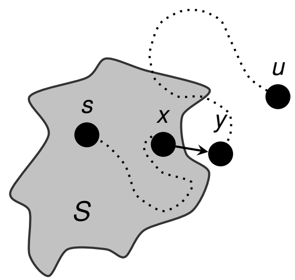
$$u.d > u.\delta$$

- 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



Correctness of Dijkstra's Algorithm

Theorem 24.6

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

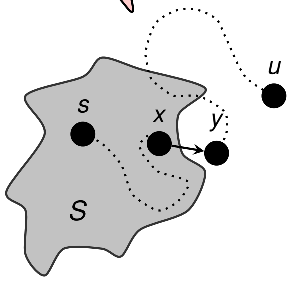
There are edge cases like $s = x$ and/or $y = u$!

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- 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 $V \setminus S$



$\Rightarrow$

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

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

6.5: All-Pairs Shortest Paths

Frank Stajano   Thomas Sauerwald

Lent 2015

UNIVERSITY OF CAMBRIDGE

Outline

Formalizing 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

$$l_{i,j} = \begin{cases} \text{weight of a shortest path from } i \text{ to } j, & \text{if } j \text{ is reachable from } i \\ \infty & \text{otherwise.} \end{cases}$$

Here we will only compute the weight of the shortest path without keeping track of the edges of the path!

Outline

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

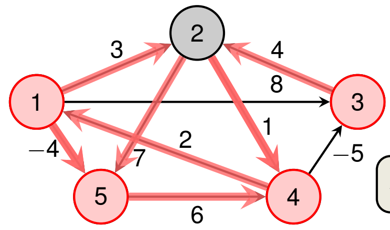
- Let $\ell_{i,j}^{(m)}$ be min. weight of any path from i to j with at most m edges
- Then $\ell_{i,j}^{(1)} = w_{i,j}$, so $L^{(1)} = W$
- How can we obtain $L^{(2)}$ from $L^{(1)}$?

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

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)



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

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

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

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

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

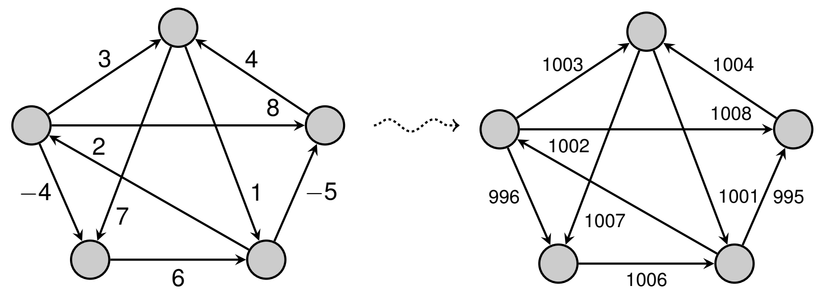
Johnson's Algorithm

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!

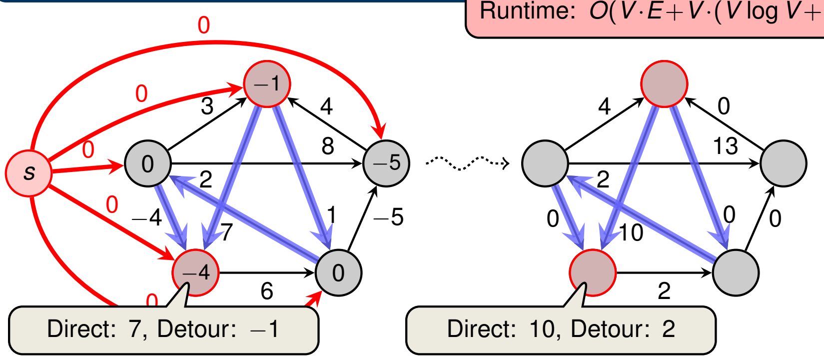


How Johnson's Algorithm works

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

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

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

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

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

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

□

Correctness of Johnson's Algorithm

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

Theorem

For any graph $G = (V, E, w)$ without negative-weight cycles:

1. After reweighting, all edges are non-negative
2. Shortest Paths are preserved

Proof of 2.

Let $p = (v_0, v_1, \ldots, v_k)$ be any path

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

6.6: Maximum flow

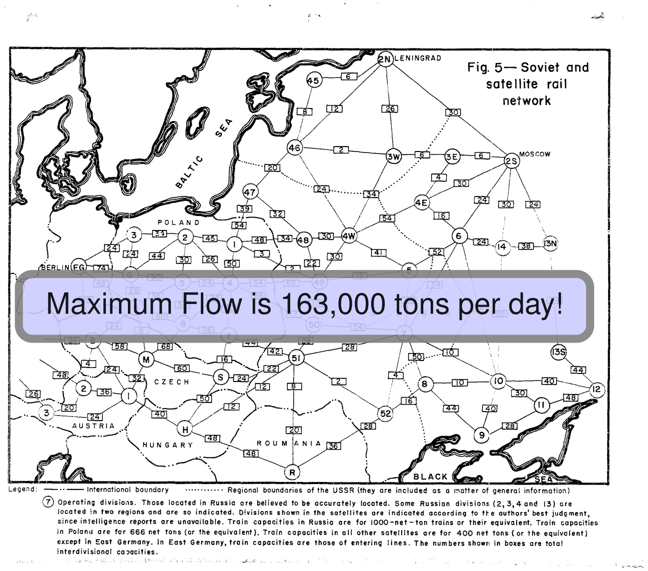
Frank Stajano Thomas Sauerwald

Lent 2015

UNIVERSITY OF CAMBRIDGE

Outline

History of the Maximum Flow Problem [Harris, Ross (1955)]



Maximum Flow is 163,000 tons per day!

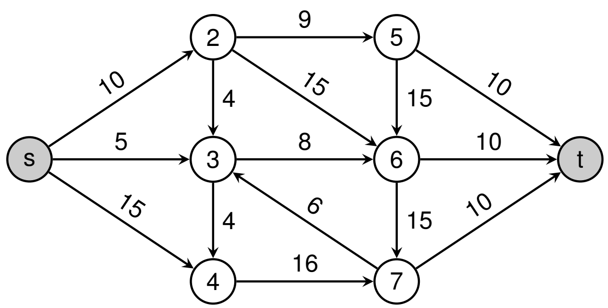
Flow Network

Flow Network

- Abstraction for material (one commodity!) flowing through the edges
- $G = (V, E)$ directed graph without parallel edges
- distinguished nodes: source s and sink t
- every edge e has a capacity $c(e)$

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

$c(u, v) = 0 \Leftrightarrow (u, v) \notin E$



Flow Network

Flow

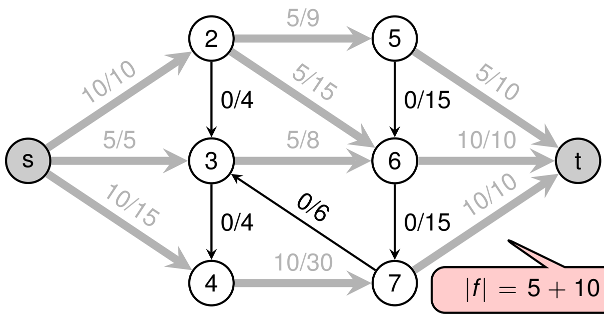
A flow is a function $f : V \times V \to \mathbb{R}$ that satisfies:

- For every $u, v \in V$, $f(u,v) \leq c(u,v)$
- For every $v \in V \setminus \{s,t\}$, $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,u)\in E} f(v,u)$
- $f(u,v) = -f(v,u)$

The value of a flow is defined as $|f| = \sum_{v\in V} f(s,v)$

Flow Conservation

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



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

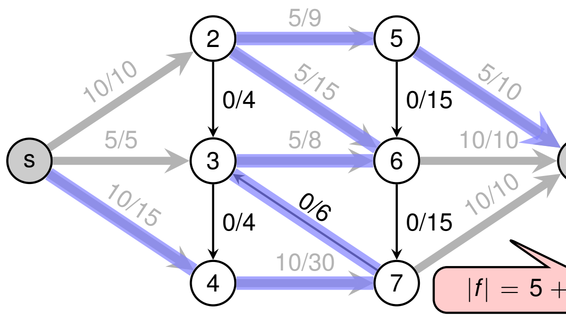
Flow Network

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 $v \in V \setminus \{s,t\}$, $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,u)\in E} f(v,u)$
- $f(u,v) = -f(v,u)$

The value of a flow is defined as $|f| = \sum_{v\in V} f(s,v)$



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

Flow Network

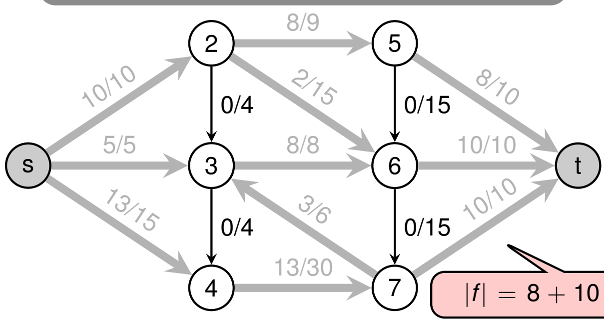
Flow

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

- For every $u, v \in V$, $f(u,v) \leq c(u,v)$
- For every $v \in V \setminus \{s,t\}$, $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,u)\in E} f(v,u)$
- $f(u,v) = -f(v,u)$

The value of a flow is defined as $|f| = \sum_{v\in V} f(s,v)$

How to find a Maximum Flow?

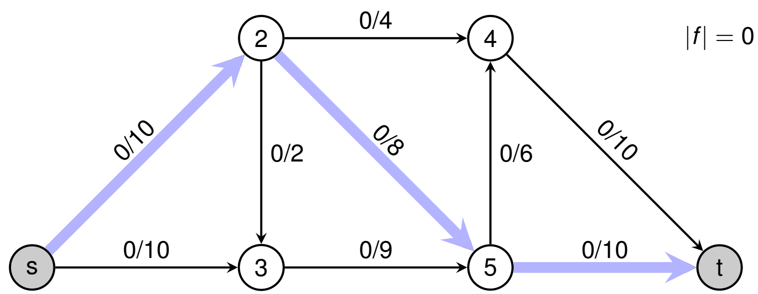


$|f| = 8 + 10 + 10 = 28$

A First Attempt

Greedy Algorithm

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

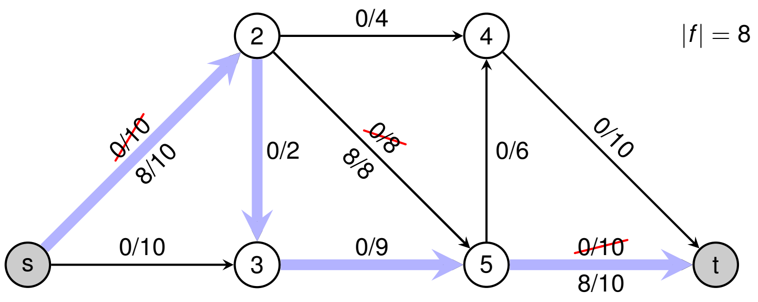


$|f| = 0$

A First Attempt

Greedy Algorithm

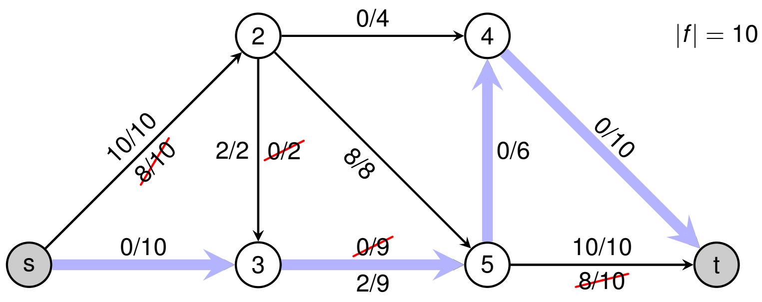
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

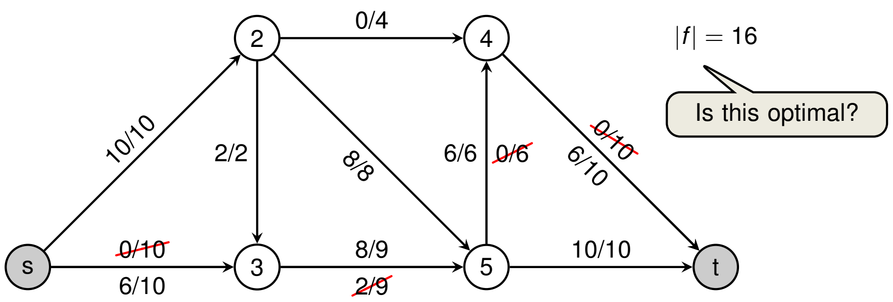
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

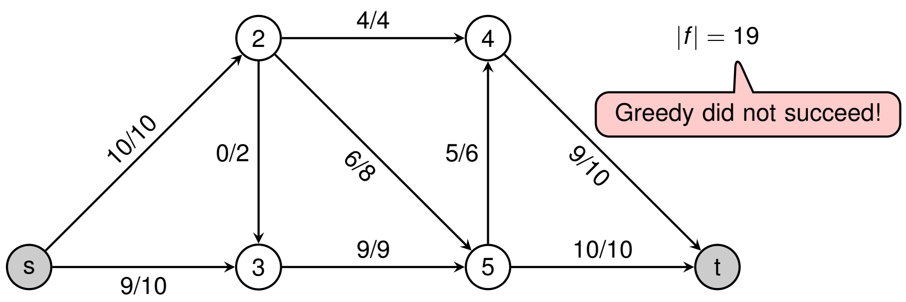
- Start with $f(u, v) = 0$ everywhere
- Repeat as long as possible:
  - Find a (s, t)-path p where each edge $e = (u, v)$ has $f(u, v) < c(u, v)$
  - Augment flow along p



A First Attempt

Greedy Algorithm

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



Outline

Residual Graph

Original Edge

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

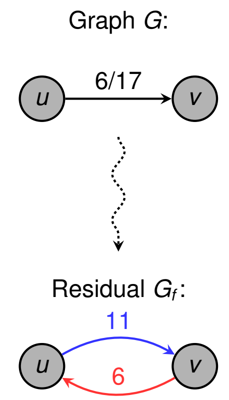
- flow $f(u, v)$ and capacity $c(u, v)$

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

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



The Ford-Fulkerson Method ("Enhanced Greedy")

```
def fordFulkerson(G)
    initialize flow to 0 on all edges
    while an augmenting path in G_f can be found:
       push as much extra flow as possible through it
```

Augmenting path: Path from source to sink in G_f

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

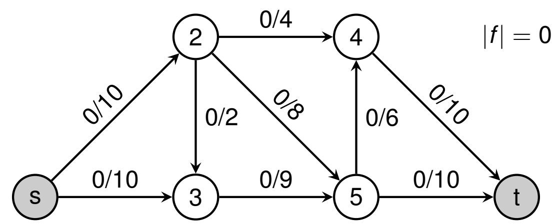
Using BFS or DFS, we can find an augmenting path in $O(V + E)$ time.

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

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



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

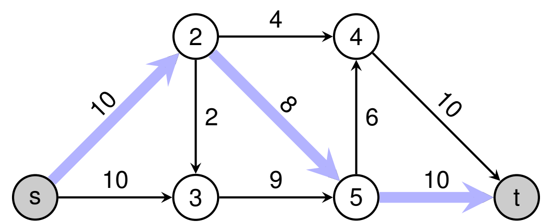
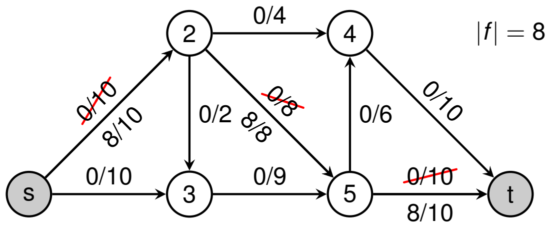


Illustration of the Ford-Fulkerson Method

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



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

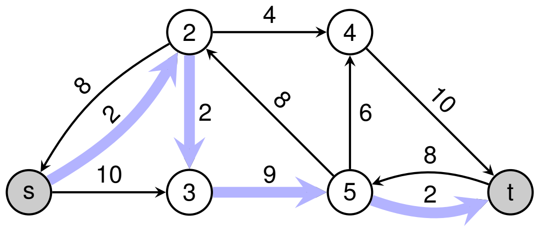
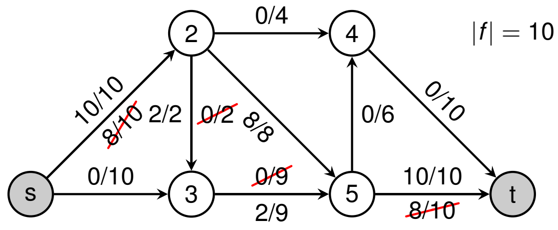


Illustration of the Ford-Fulkerson Method

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



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

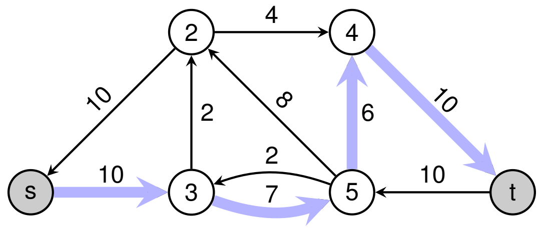
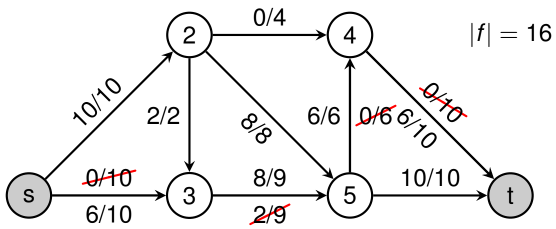


Illustration of the Ford-Fulkerson Method

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



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

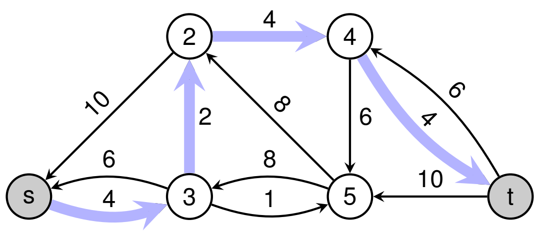
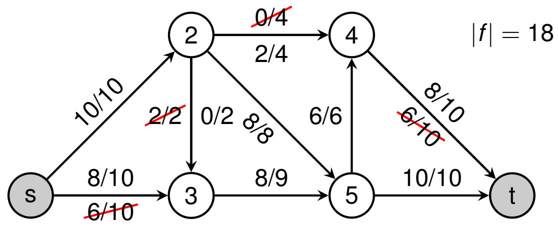


Illustration of the Ford-Fulkerson Method

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



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

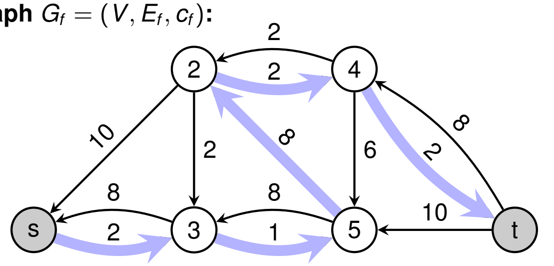
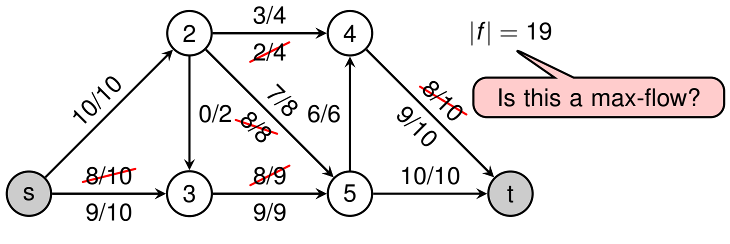


Illustration of the Ford-Fulkerson Method

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



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

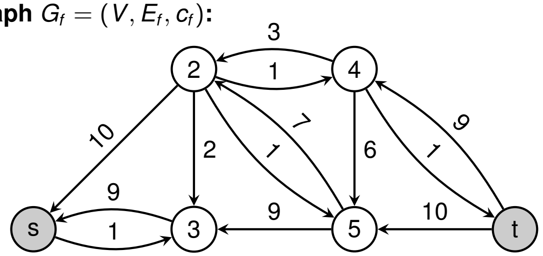
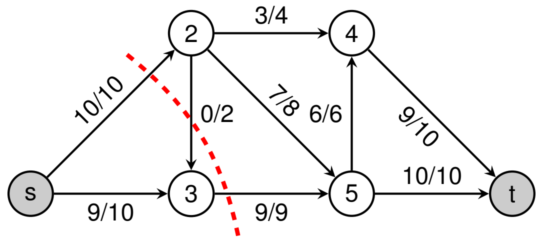
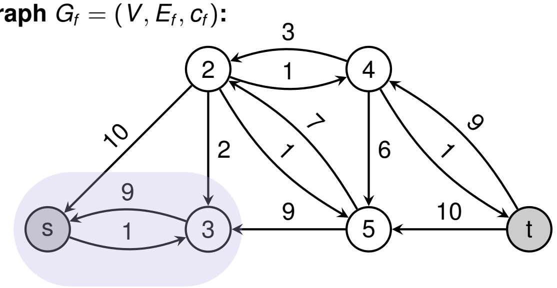


Illustration of the Ford-Fulkerson Method

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



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



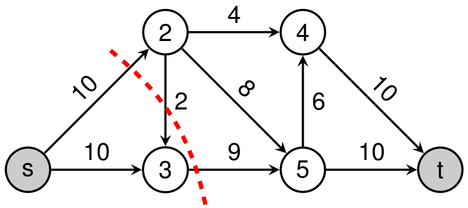
Outline

From Flows to Cuts

Cut

- A cut (S,T) is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.
- The capacity of a cut (S, T) is the sum of the capacities of the edges from S to T:
$$\mathrm{c}(S,T) = \sum_{u \in S, v \in T} c(s,t).$$
- A mininum cut of a network is a cut whose capacity is minimum over all cuts of the network.

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



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

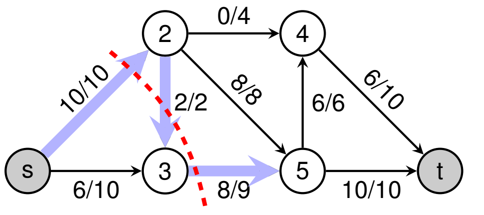
From Flows to Cuts

Flow Value Lemma (Lemma 26.4)

Let f be a flow with source s and sink t, and let (S, T) be any cut of G. Then the value of the flow is equal to the net flow across the cut, i.e.,

$$|f| = \sum_{(u,v)\in E(S,T)} f(u,v) - \sum_{(v,u)\in E(T,S)} f(v,u).$$

Graph $G = (V, E, c)$**:** $|f| = 16$



$10 - 2 + 8 = 16$

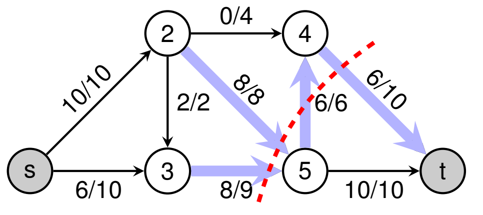
From Flows to Cuts

Flow Value Lemma (Lemma 26.4)

Let f be a flow with source s and sink t, and let (S, T) be any cut of G. Then the value of the flow is equal to the net flow across the cut, i.e.,

$$|f| = \sum_{(u,v)\in E(S,T)} f(u,v) - \sum_{(v,u)\in E(T,S)} f(v,u).$$

Graph $G = (V, E, c)$**:** $|f| = 16$



$8 + 8 - 6 + 6 = 16$

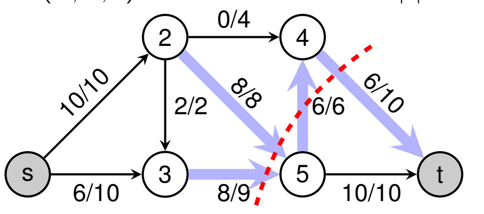
From Flows to Cuts

$$|f| = \sum_{(u,v)\in E(S,T)} f(u,v) - \sum_{(v,u)\in E(T,S)} f(v,u).$$

$$|f| = \sum_{(s,w)\in E} f(s,w) = \sum_{u\in S}\left(\sum_{(u,w)\in E} f(u,w) - \sum_{(w,u)\in E} f(w,u)\right)$$

$$= \sum_{(u,v)\in E(S,T)} f(u,v) - \sum_{(v,u)\in E(T,S)} f(v,u) \qquad \square$$

Graph $G = (V, E, c)$**:** $|f| = 16$



$8 + 8 - 6 + 6 = 16$

Weak Duality betwen Flows and Cuts

Weak Duality (Corollary 26.5)

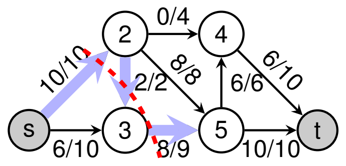
Let f be any flow and (S, T) be any cut. Then the value of f is bounded from above by the capacity of the cut (S, T), i.e.,

$$|f| \leq c(S, T).$$

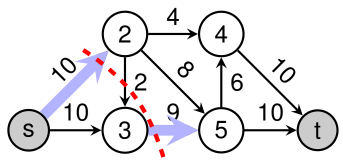
$$|f| = \sum_{(u,v)\in E(S,T)} f(u,v) - \sum_{(v,u)\in E(T,S)} f(v,u)$$

$$\leq \sum_{(u,v)\in E(S,T)} f(u,v)$$

$$\leq \sum_{(u,v)\in E(S,T)} c(u,v) = c(S,T). \qquad \square$$



$|f| = 10 - 2 + 8 = 16$



$c(S, T)$=10+9=19

Max-Flow Min-Cut Theorem

Theorem

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S,T).$$

Key Lemma

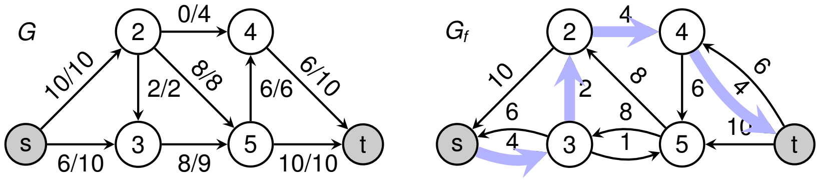
Key Lemma (Theorem 26.6)

The following three conditions are all equivalent for any flow f:

1. f is a maximum flow
2. There is no augmenting path in G_f
3. There exists a cut (S,T) such that $c(S,T) = |f|$

Proof $1 \Rightarrow 2$:

- For the sake of contradicion, suppose there is an augmenting path with respect to f.
- Then we can improve f by increasing the flow along this path.
- Hence f cannot be a maximum flow.



Key Lemma

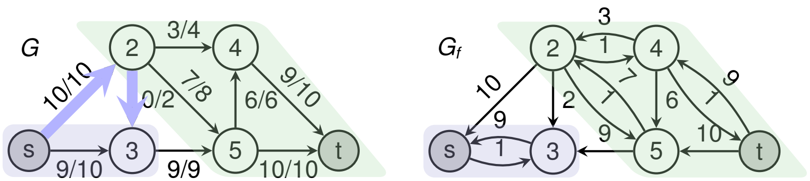
Key Lemma (Theorem 26.6)

The following three conditions are all equivalent for any flow f:

1. f is a maximum flow
2. There is no augmenting path in G_f
3. There exists a cut (S,T) such that $c(S,T) = |f|$

Proof $2 \Rightarrow 3$:

- Let f be a flow with no augmenting paths.
- Let S be the nodes reachable from s in G_f, $T := V \setminus S \Rightarrow s \in S, t \notin S$.
- $(u,v) \in E(S,T) \Rightarrow f(u,v) = c(u,v)$.
- $(v,u) \in E(T,S) \Rightarrow f(v,u) = 0$.



Key Lemma

Key Lemma (Theorem 26.6)

The following three conditions are all equivalent for any flow f:

1. f is a maximum flow
2. There is no augmenting path in G_f
3. There exists a cut (S,T) such that $c(S,T) = |f|$

Proof $2 \Rightarrow 3$:

- Let f be a flow with no augmenting paths.
- Let S be the nodes reachable from s in G_f, $T := V \setminus S \Rightarrow s \in S, t \notin S$.
- $(u,v) \in E(S,T) \Rightarrow f(u,v) = c(u,v)$.
- $(v,u) \in E(T,S) \Rightarrow f(v,u) = 0$.

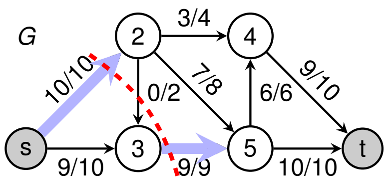
$$\begin{aligned} |f| &= \sum_{(u,v)\in E(S,T)} f(u,v) - \sum_{(v,u)\in E(T,S)} f(v,u) \\ &= \sum_{(u,v)\in E(S,T)} c(u,v) = c(S,T) \qquad \square \end{aligned}$$

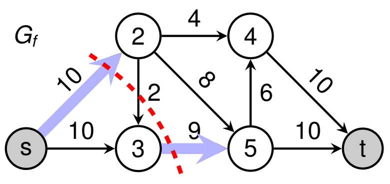
Key Lemma

Key Lemma (Theorem 26.6)

The following three conditions are all equivalent for any flow f:
1. f is a maximum flow
2. There is no augmenting path in G_f
3. There exists a cut (S, T) such that $c(S, T) = |f|$

Proof 3 $\Rightarrow$ 1:
- Suppose that (S, T) is a cut with $c(S, T) = |f|$
- By Corollary 26.5, for any flow $\widetilde{f}$,
- Hence f is a maximum flow.





Proof of the Max-Flow Min-Cut Theorem

Key Lemma

The following conditions are equivalent for any flow f:
1. There exists a cut (S, T) such that $c(S, T) = |f|$
2. f is a maximum flow
3. There is no augmenting path in G_f.

Theorem

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S, T).$$

Proof of "$\leq$":
- For any flow f and cut (S, T), $|f| \leq c(S, T)$ (Corollary 26.5)

$\Rightarrow$

$$\max_f |f| \leq \min_{S,T \subseteq V} c(S, T)$$

Proof of the Max-Flow Min-Cut Theorem

Key Lemma

The following conditions are equivalent for any flow f:
1. There exists a cut (S, T) such that $c(S, T) = |f|$
2. f is a maximum flow
3. There is no augmenting path in G_f.

Theorem

The value of the max-flow is equal to the capacity of the min-cut, that is

$$\max_f |f| = \min_{S,T \subseteq V} c(S, T).$$

Proof of "$\geq$":
- Let $f_{\max}$ be a maximum flow
- Key Lemma $\Rightarrow$ there is a cut (S, T) with $c(S, T) = |f_{\max}|$.

$\Rightarrow$

$$\max_f |f| = |f_{\max}| \geq c(S, T) \geq \min_{S,T \subseteq V} c(S, T) \qquad \square$$

Outline

Analysis of Ford-Fulkerson

```
def FordFulkerson(G)
   initialize flow to 0 on all edges
   while an augmenting path in G_f can be found:
      push as much extra flow as possible through it
```

Lemma

If all capacities $c(u, v)$ are integral, then the flow at every iteration of Ford-Fulkerson is integral.

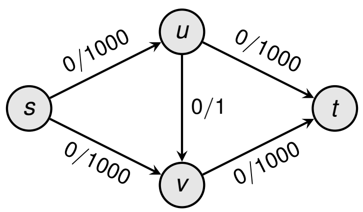
> Flow before iteration integral
> & capacities in G_f are integral
> $\Rightarrow$ Flow after iteration integral

Theorem

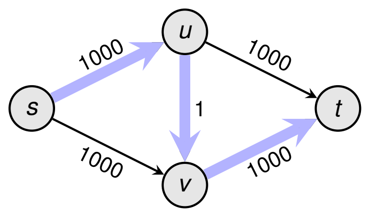
For integral capacities $c(u, v)$, Ford-Fulkerson terminates after $C := \max_{u,v} c(u, v)$ iterations and returns the maximum flow.

> at the time of termination, no augmenting path
> $\Rightarrow$ Ford-Fulkerson returns maxflow (Key Lemma)

Slow Convergence of Ford-Fulkerson (Figure 26.7)

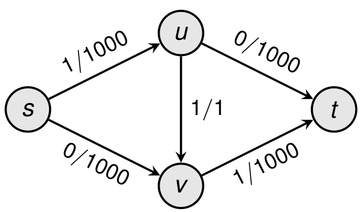


G

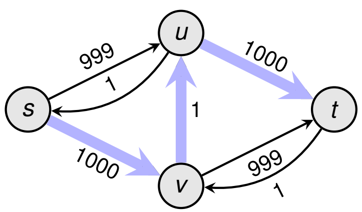


G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)

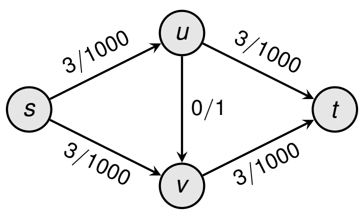


G

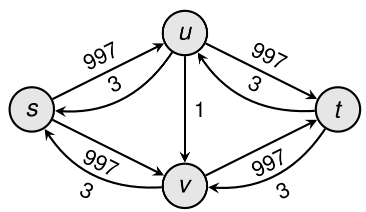


G_f

Slow Convergence of Ford-Fulkerson (Figure 26.7)



G



G_f

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

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

Summary and Outlook

Ford-Fulkerson Method

- works only for integral (rational) capacities
- Runtime: $O(E \cdot |f^*|) = O(E \cdot C)$

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

Max-Flow Min-Cut Theorem

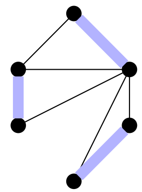
Analysis of Ford-Fulkerson

Matchings in Bipartite Graphs

Application: Maximum-Bipartite-Matching Problem

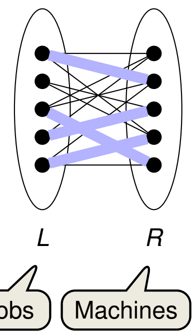
Matching

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.



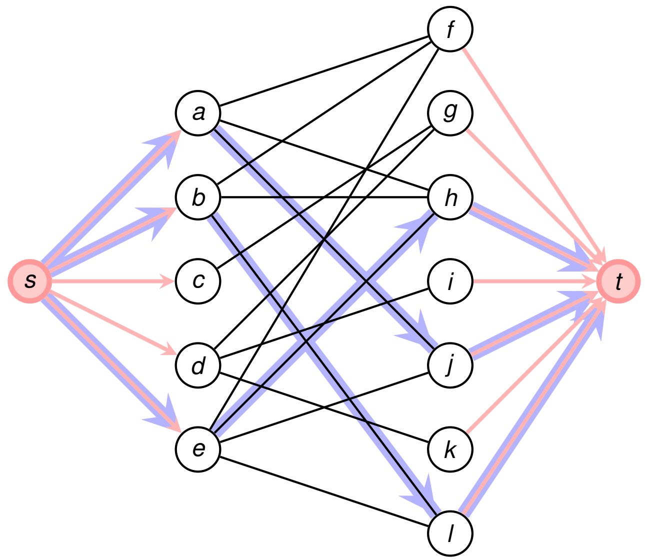
Bipartite Graph

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 $G = (V \cup L, E)$, find a matching of maximum cardinality.

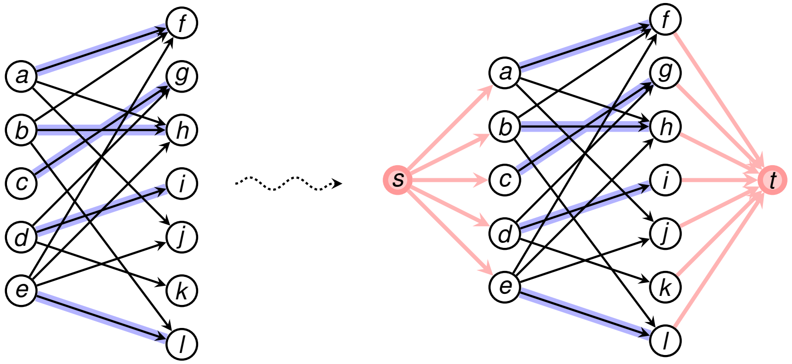
Matchings in Bipartite Graphs via Maximum Flows



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 $\widetilde{G}$.



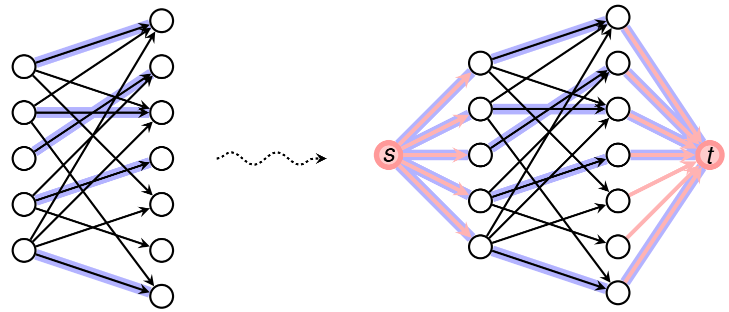
Graph G Graph $\widetilde{G}$

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



Graph G Graph $\widetilde{G}$

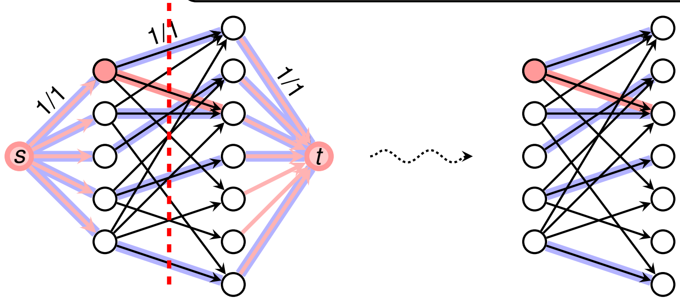
From Flow to Matching

- Let f be a maximum flow in 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 receives at most one unit
b) Flow Conservation $\Rightarrow$ every node in R sends 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



L   R   L   R

7: Geometric Algorithms

Frank Stajano Thomas Sauerwald

Lent 2015

UNIVERSITY OF CAMBRIDGE

Outline

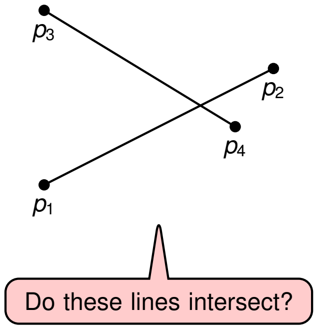
Introduction

Computational Geometry

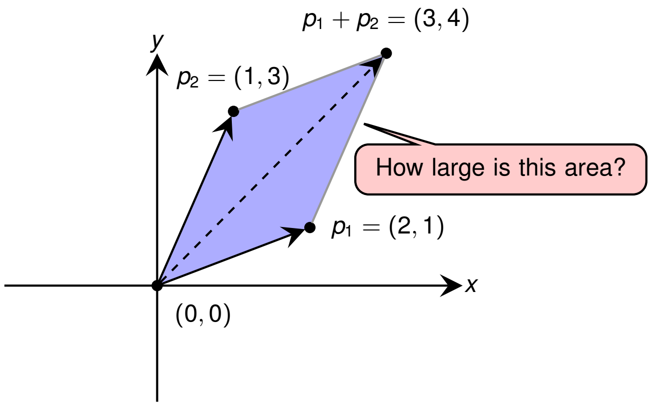
- Branch that studies algorithms for geometric problems
- typically, input is a set of points, line segments etc.

Applications

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



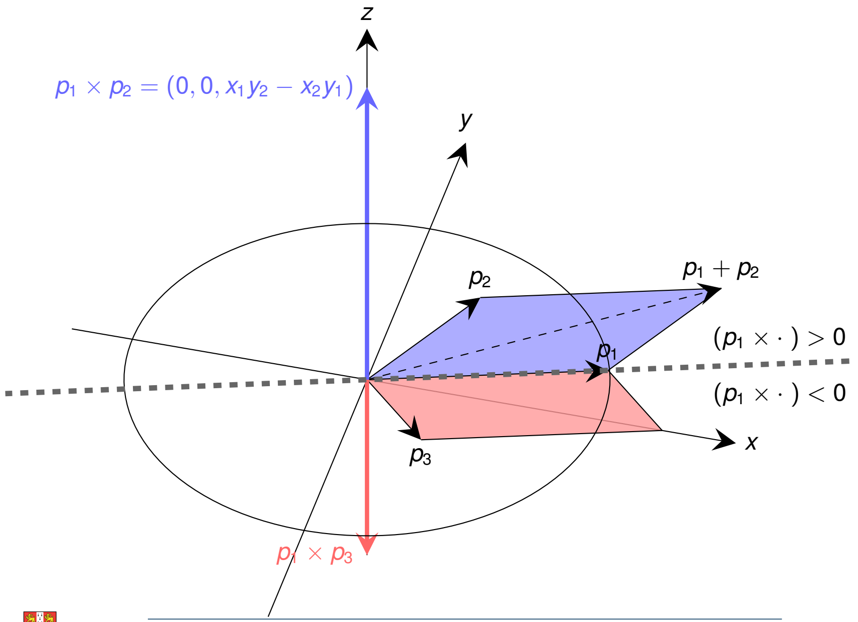
Cross Product (Area)



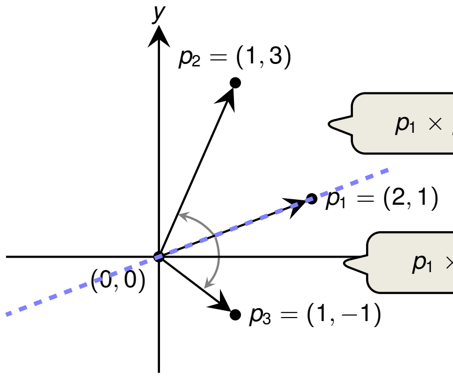
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = 2 \cdot 3 - 1 \cdot 1 = 5$$

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

Cross Product in 3D



Using Cross product to determine Turns (1/2)



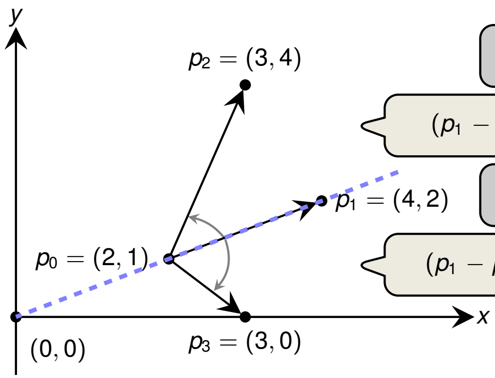
$p_1 \times p_2 > 0$: left (counterclockwise) turn

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

Sign of cross product determines turn!

Cross product equals zero iff vectors are colinear

Using Cross product to determine Turns (2/2)



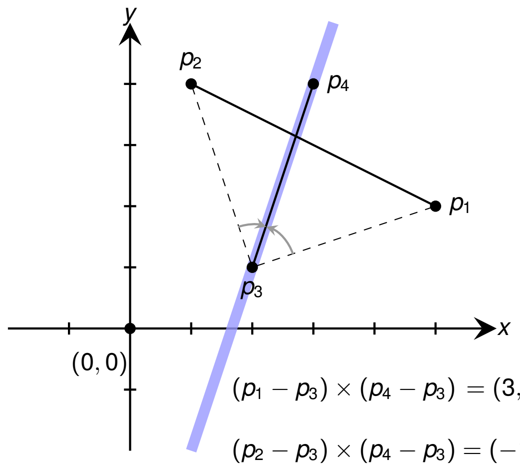
$(2, 1) \times (1, 3) = 5$

$(p_1 - p_0) \times (p_2 - p_0) > 0$: left turn

$(2, 1) \times (1, -1) = -3$

$(p_1 - p_0) \times (p_3 - p_0) < 0$: right turn

Solving Line Intersection (without Trigonometry and Division!)

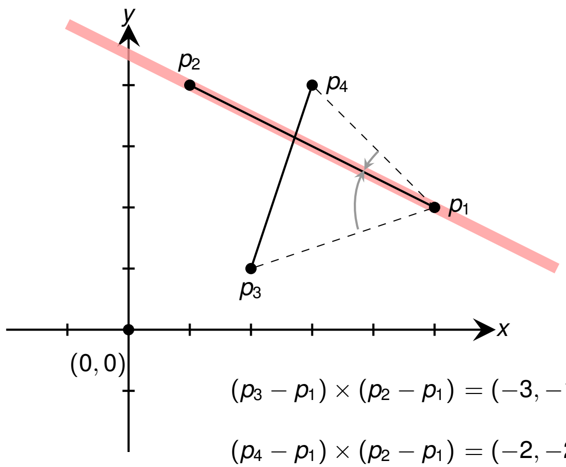


$$(p_1 - p_3) \times (p_4 - p_3) = (3, 1) \times (1, 4) = 11$$

$$(p_2 - p_3) \times (p_4 - p_3) = (-1, 3) \times (1, 4) = -7$$

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

Solving Line Intersection (without Trigonometry and Division!)



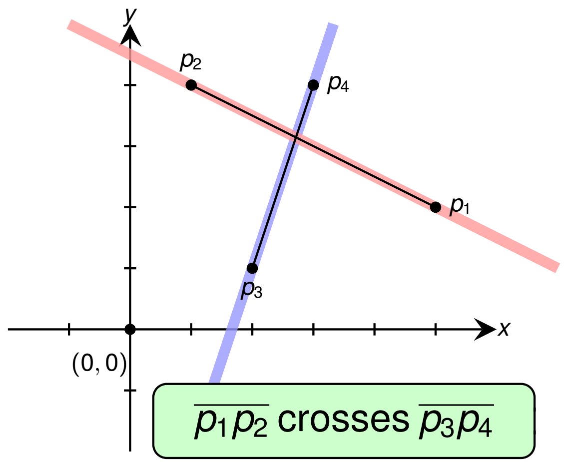
$(p_3 - p_1) \times (p_2 - p_1) = (-3, -1) \times (4, 2) = -10$

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

Opposite signs $\Rightarrow$ $\overline{p_1p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow$ $\overline{p_3p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection (without Trigonometry and Division!)

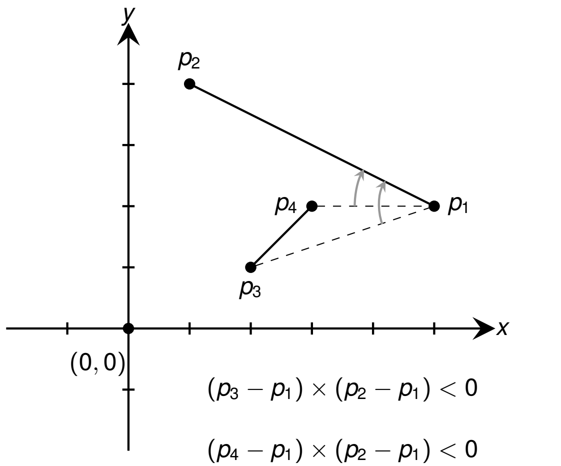


$\overline{p_1p_2}$ crosses $\overline{p_3p_4}$

Opposite signs $\Rightarrow$ $\overline{p_1p_2}$ crosses (infinite) line through p_3 and p_4

Opposite signs $\Rightarrow$ $\overline{p_3p_4}$ crosses (infinite) line through p_1 and p_2

Solving Line Intersection (without Trigonometry and Division!)

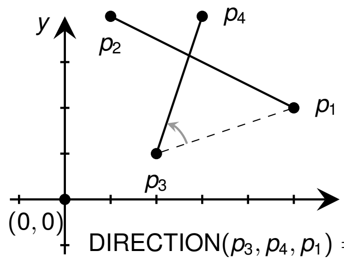


$(p_3 - p_1) \times (p_2 - p_1) < 0$

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

$\overline{p_1p_2}$ does **not** cross $\overline{p_3p_4}$

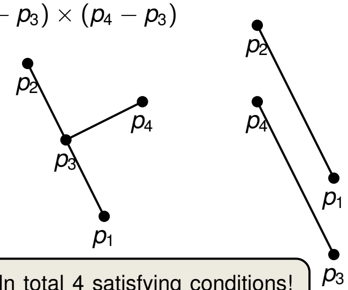
Solving Line Intersection



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

```
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 toch or be colinear

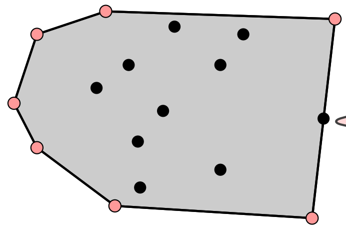
Outline

Introduction and Line Intersection

Convex Hull

Inside or Outside a Polygon?

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

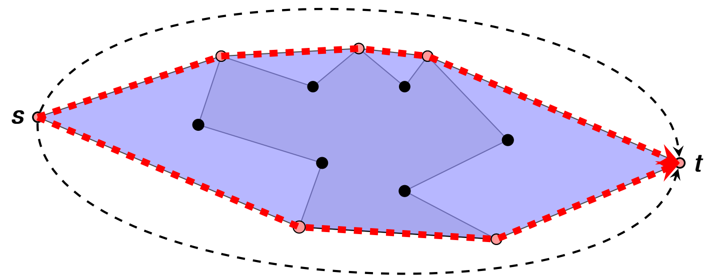
- Input: set of points Q in the Euclidean space
- Output: return points of the convex hull in counterclockwise order

Application of Convex Hull

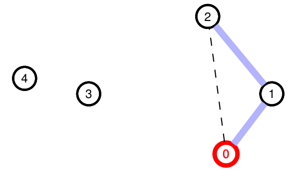
Robot Motion Planning

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

can be solved by computing the Convex hull!



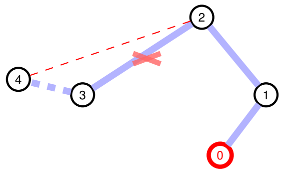
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓

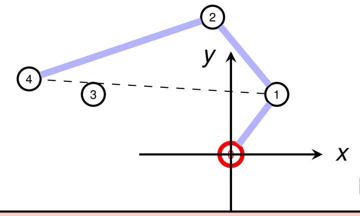
Graham's Scan



Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓
  - Otherwise, keep on removing recent points until point can be added

Graham's Scan



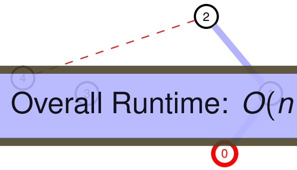
Use Cross Product!

Efficient Sorting by comparing (not computing!) polar angles

Basic Idea

- Start with the point with smallest y-coordinate
- Sort all points increasingly according to their polar angle
- Try to add next point to the convex hull
  - If it does not introduce non-left turn, then fine ✓

Graham's Scan



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

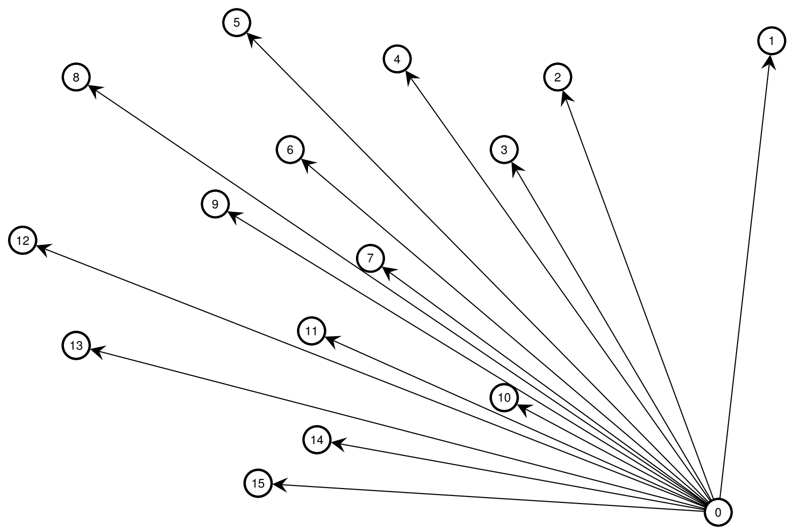
```
GRAHAM-SCAN(Q)
    Let p_0 be the point with minimum y-coordinate
    Let (p_1, p_2, ..., p_n) be the other points sorted by polar angle w.r.t. p_0
    If n < 2 return false
    S = ∅
    PUSH(p_0,S)
    PUSH(p_1,S)
    PUSH(p_2,S)
    For i = 3 to n
        While angle of NEXT-TO-TOP(S),TOP(S),p_i makes a non-left turn
            POP(S)
        End While
        PUSH(p_i,S)
    End For
    Return S
```

Takes $O(n \log n)$ time

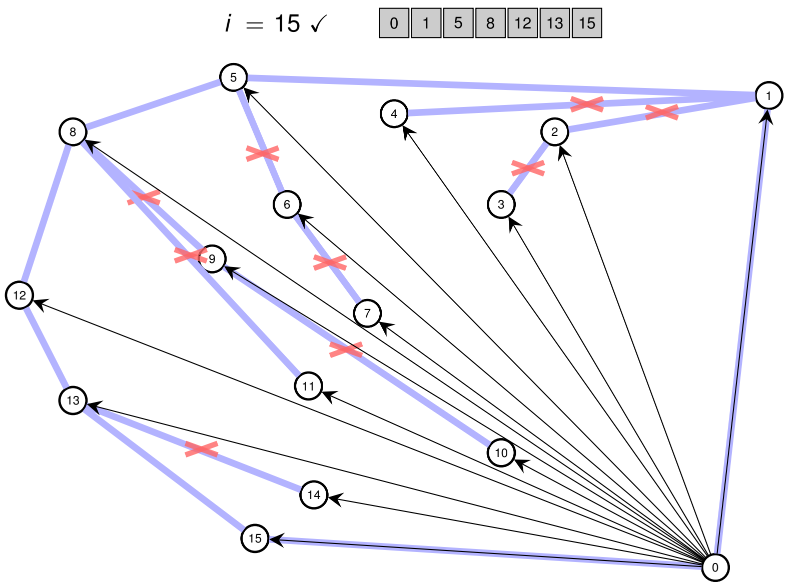
Takes $O(n)$ time, since every point is part of a PUSH or POP at most once.

Complete Run of Graham's Scan

$i = 0$



Complete Run of Graham's Scan



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

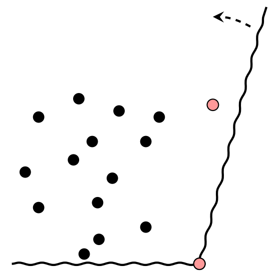
Algorithm

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

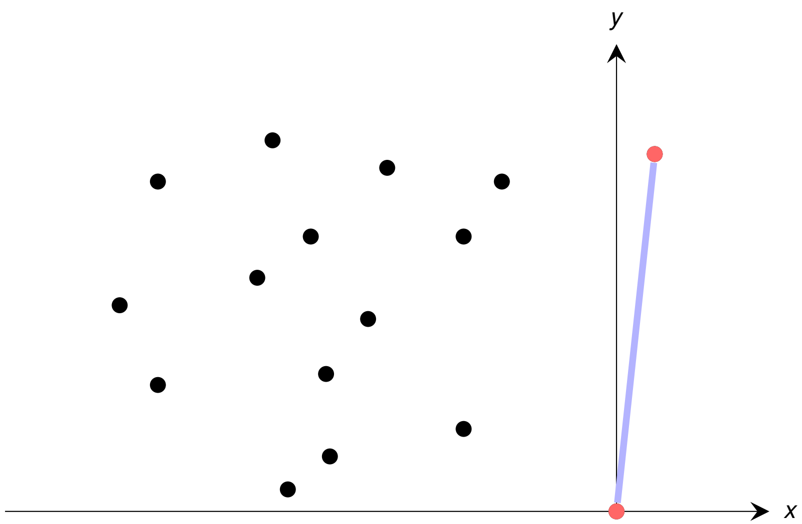
Here, we rotate the coordinate system by 180!

Runtime: $O(n \cdot h)$

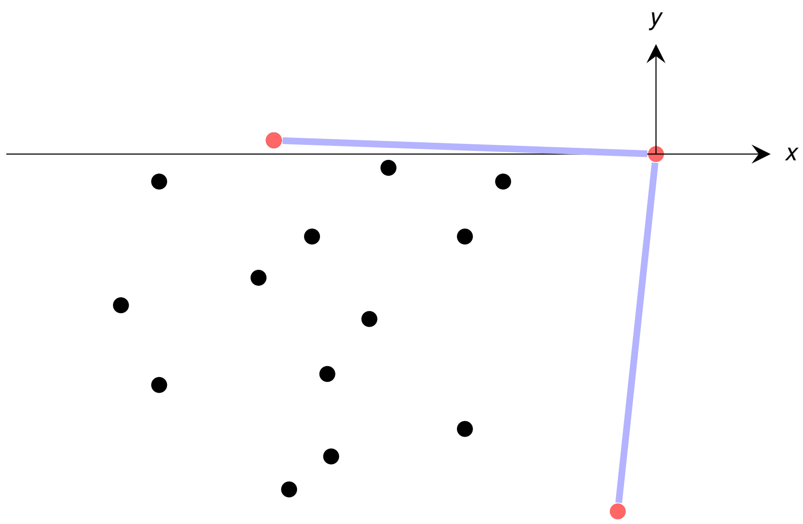
Output sensitive algorithm!



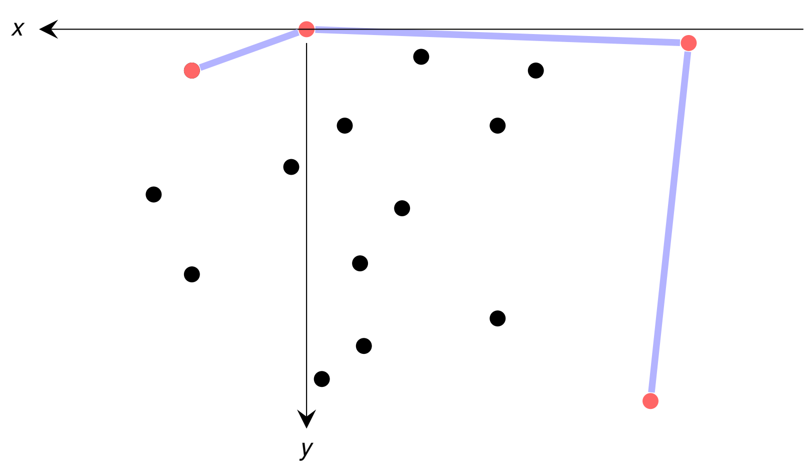
Execution of Jarvis' March



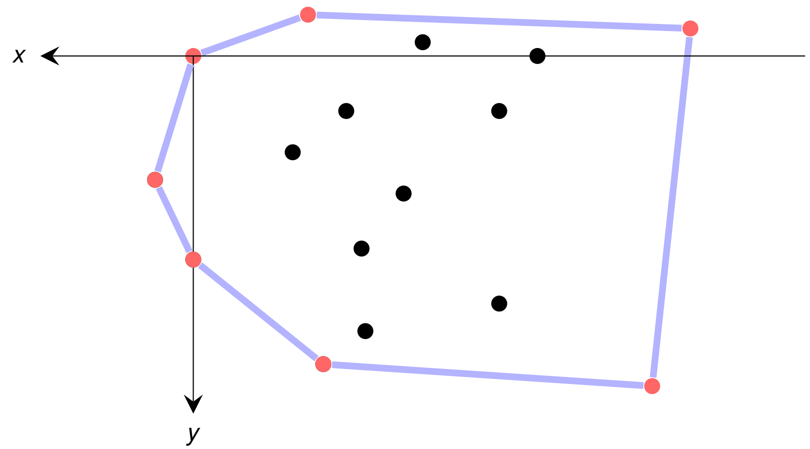
Execution of Jarvis' March



Execution of Jarvis' March



Execution of Jarvis' March



Computing Convex Hull: Summary

Graham's Scan

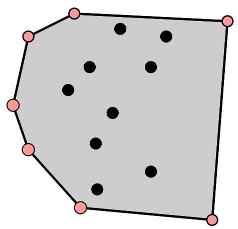
- natural backtracking algorithm
- cross-product avoids computing polar angles
- Runtime dominated by sorting $\rightsquigarrow O(n \log n)$

Jarvis' March

- proceeds like wrapping a gift
- 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
- take care of degenerate cases, numerical precision

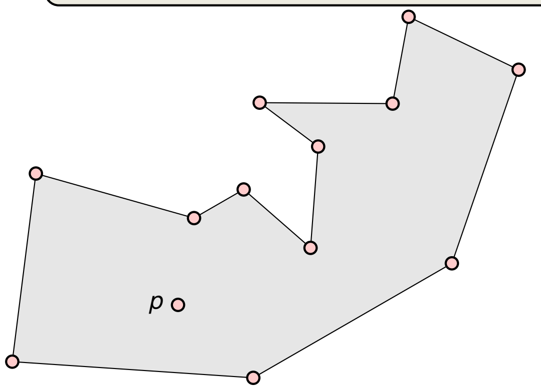
Outline

Another Problem: The Inside-or-Outside a Polygon Problem

Problem

- Given: A polygon as list of edges $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, and another point $p = (p_1, p_2)$
- Question: Is the point $p = (p_1, p_2)$ inside or outside the polygon?

Two-year old human can do it, but it's not so obvious how to program a computer...



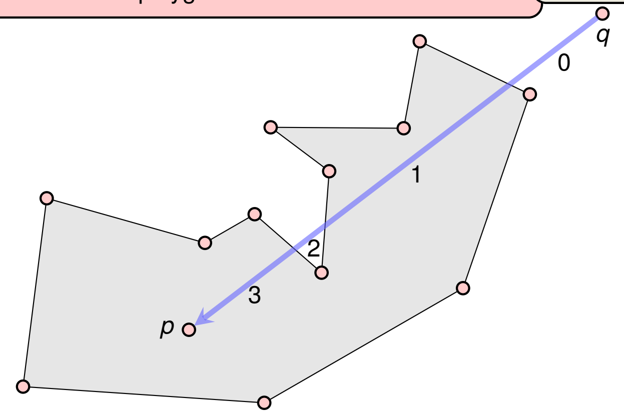
Another Problem: The Inside-or-Outside a Polygon Problem

Problem

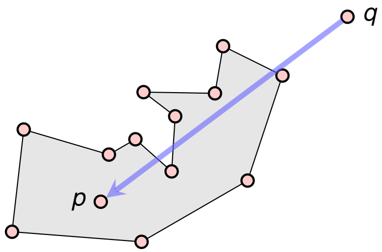
- Given: A polygon as list of edges $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, and another point $p = (p_1, p_2)$
- Question: Is the point $p = (p_1, p_2)$ inside or outside the polygon?

Observation: p is inside if a line drawn from "infinity" to p crosses the polygon an odd number of times

"infinity": can take $(\max_i x_i + 1, \max_i y_i + 1)$



Solution to the Inside-or-Outside a Polygon Problem



But this is just the line segment intersection problem...

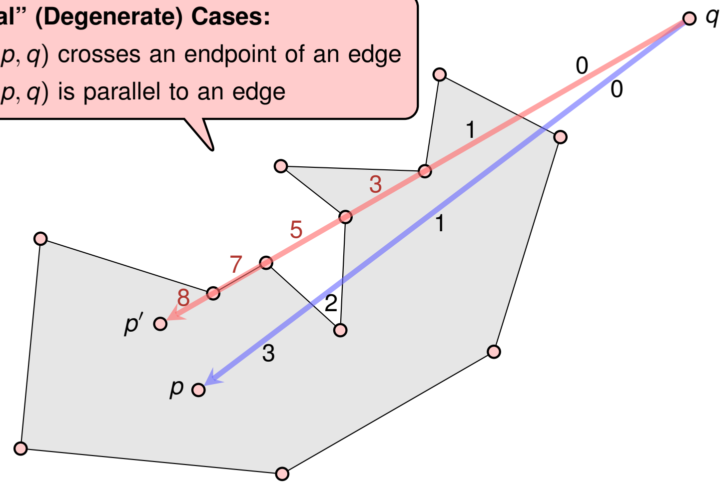
```
IsInside(point p, polygon G)
{   let q be a point outside G ('infinity')
    int count = 0;
    for E an edge of G do
        if (pq intersects E) count = count+1;
    if (count is odd) return YES;
    return NO;
}
```


Inside-or-Outside the Polygon Problem: Are we done yet?

Some extra tweaks are needed to make our program (fully) correct...

"Unusual" (Degenerate) Cases:

- line (p, q) crosses an endpoint of an edge
- line (p, q) is parallel to an edge



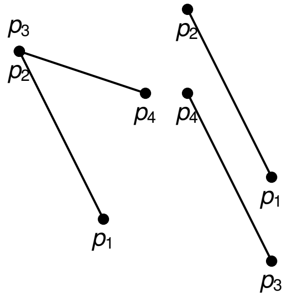
How likely are “unusual” cases?

First Idea

If we take four random points in a plane, then the mathematical probability of:

- two of them being identical; or
- any two lines connecting them being parallel

is vashingly small (even “probability zero”)!



So, I don’t have to worry about this when programming?

Wrong!

- Imagine people drawing squares or two objects together (Real examples may result in lots of horrible cases)
- Computer floating-point arithmetic is not exact (Problems when points are very close)

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 (typos, criticsm, praise etc.) to: **`tms41@cam.ac.uk`**