算法设计与分析(2024年春季学期)

Final Review

授课老师: 栗师

南京大学计算机科学与技术系

Outline

What is an Algorithm?

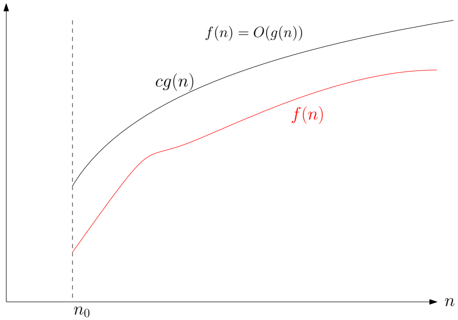
- Donald Knuth: An algorithm is a finite, definite effective procedure, with some input and some output.
- Algorithm: "abstract", can be specified using computer program, English, pseudo-codes or flow charts.
- Computer program: "concrete", implementation of algorithm, using a particular programming language

Def. $f : \mathbb{N} \to \mathbb{R}$ is an asymptotically positive function if:

- $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

O**-Notation** For a function $g(n)$,

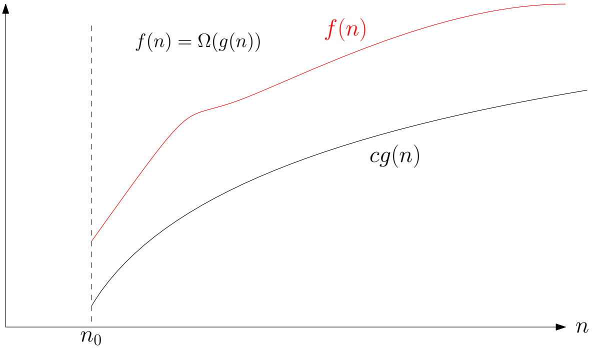
$$O(g(n)) = \{\text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \le cg(n), \forall n \ge n_0\}.$$



- We use "$f(n) = O(g(n))$" to denote "$f(n) \in O(g(n))$"

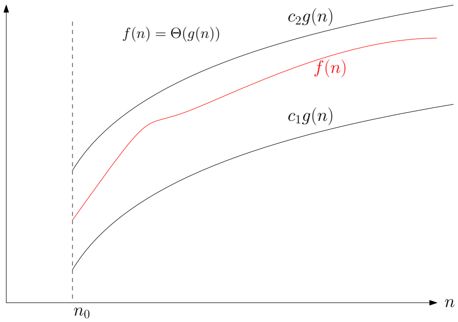
Ω-Notation For a function $g(n)$,

$$\Omega(g(n)) = \big\{\text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0\big\}.$$



Θ-Notation For a function $g(n)$,

$$\Theta(g(n)) = \{\text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}.$$



o-**Notation** For a function $g(n)$,

$$o(g(n)) = \{\text{function } f : \forall c > 0, \exists n_0 > 0 \text{ such that } f(n) \le cg(n), \forall n \ge n_0\}.$$

ω-**Notation** For a function $g(n)$,

$$\omega(g(n)) = \{\text{function } f : \forall c > 0, \exists n_0 > 0 \text{ such that } f(n) \ge cg(n), \forall n \ge n_0\}.$$

Running Times

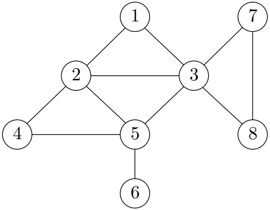
- Linear time: compute the summation/max/min of an array of n elements
- $O(n \log n)$ Running Time: merge-sort, counting-inversion, D&C algorithms with recurrence $T(n) = 2T(n/2) + O(n)$.
- $O(n^2)$ (Quadratic) time: enumerating pairs of elements in an array of size n
- $O(n^3)$ (Cubic) time: naive algorithm for computing matrix multiplication
- Beyond polynomial time: $2^{O(n)}$ and $O(n!)$

Outline

2 Graph Basics

- Connectivity and Graph Traversal
- Topological Ordering
- Finding Bridges

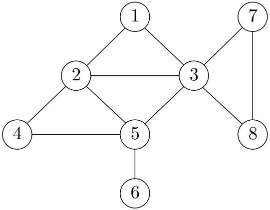
Representation of Graphs



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbours of v.

Representation of Graphs



1: 2 → 3
2: 1 → 3 → 4 → 5
3: 1 → 2 → 5 → 7 → 8
4: 2 → 5
5: 2 → 3 → 4 → 6
6: 5
7: 3 → 8
8: 3 → 7

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbours of v.

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n - 1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbours of v	$O(n)$	$O(d_v)$

Outline

2 Graph Basics

- Connectivity and Graph Traversal
- Topological Ordering
- Finding Bridges

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Implementing BFS using a Queue

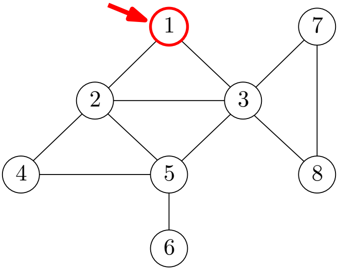
BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
while $head \leq tail$ **do**
  $v \leftarrow queue[head], head \leftarrow head + 1$
  for all neighbours u of v **do**
    if u is "unvisited" **then**
      $tail \leftarrow tail + 1, queue[tail] = u$
      mark u as "visited"

- Running time: $O(n + m)$.

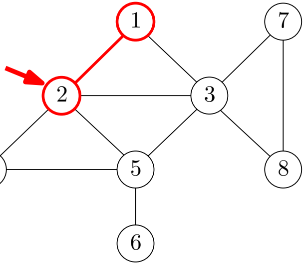
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



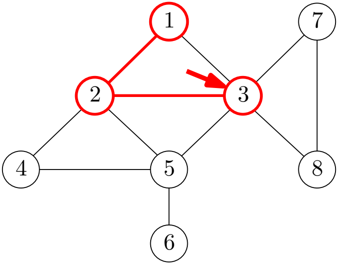
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



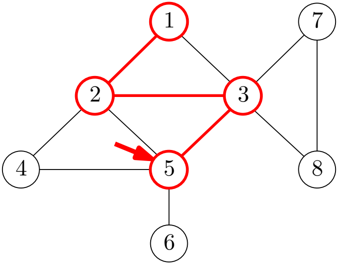
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



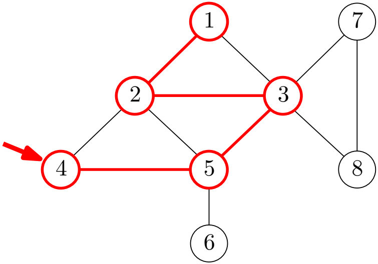
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



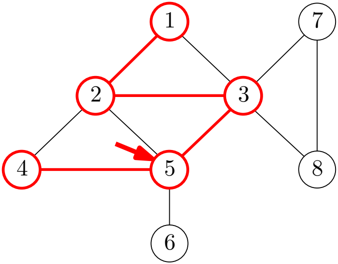
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



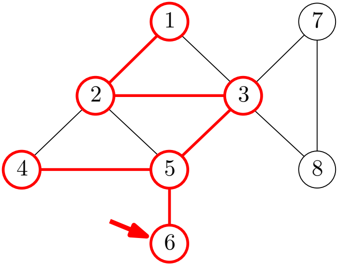
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



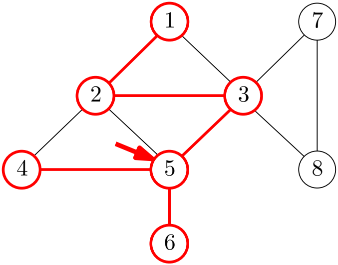
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



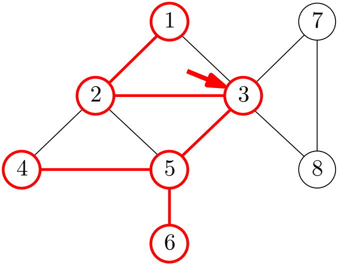
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



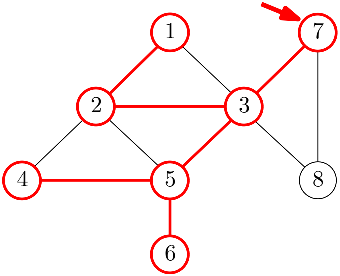
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



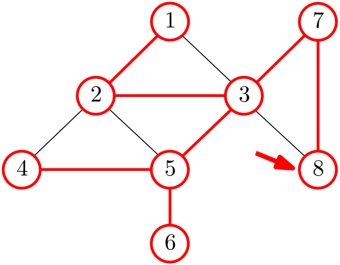
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Implementing DFS using Recurrsion

DFS(s)

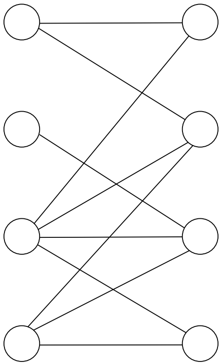
```
mark all vertices as "unvisited"
recursive-DFS(s)
```

recursive-DFS(v)

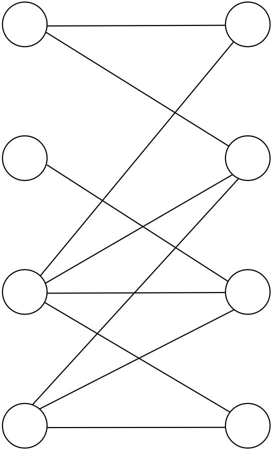
```
mark v as "visited"
for all neighbours u of v do
    if u is unvisited then recursive-DFS(u)
```

Testing Bipartiteness: Applications of BFS

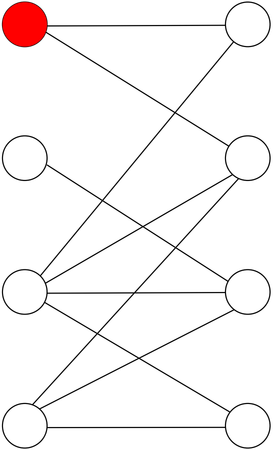
Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$.



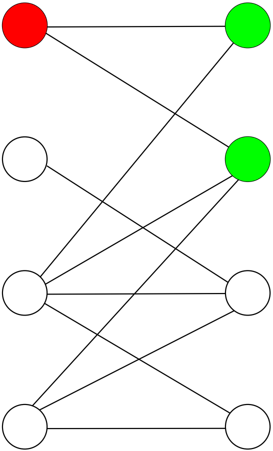
Test Bipartiteness



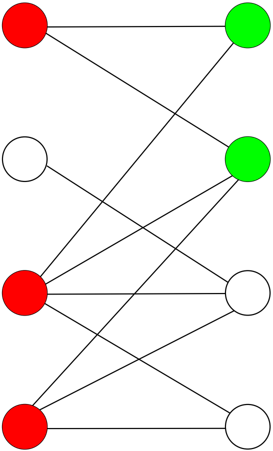
Test Bipartiteness



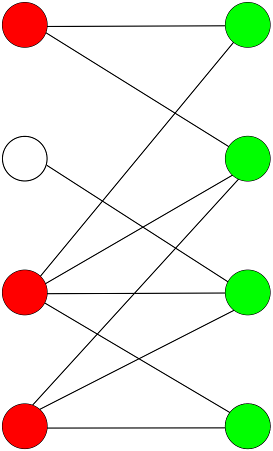
Test Bipartiteness



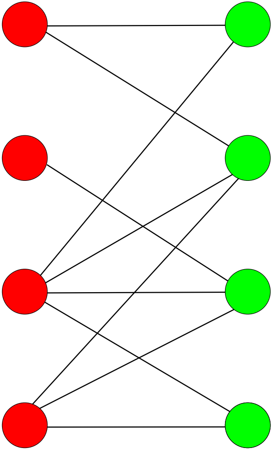
Test Bipartiteness



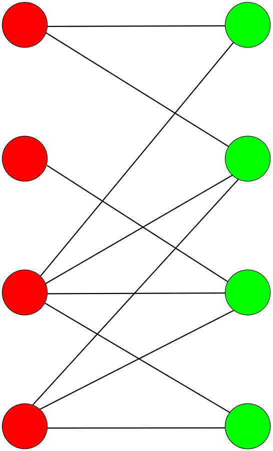
Test Bipartiteness

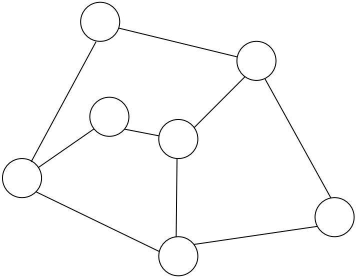


Test Bipartiteness

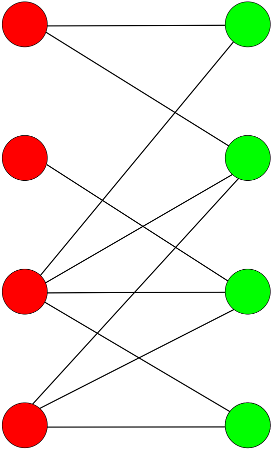


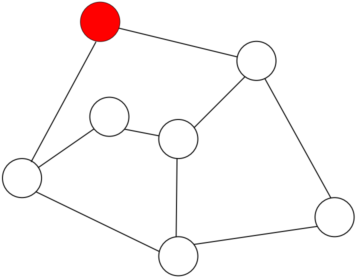
Test Bipartiteness



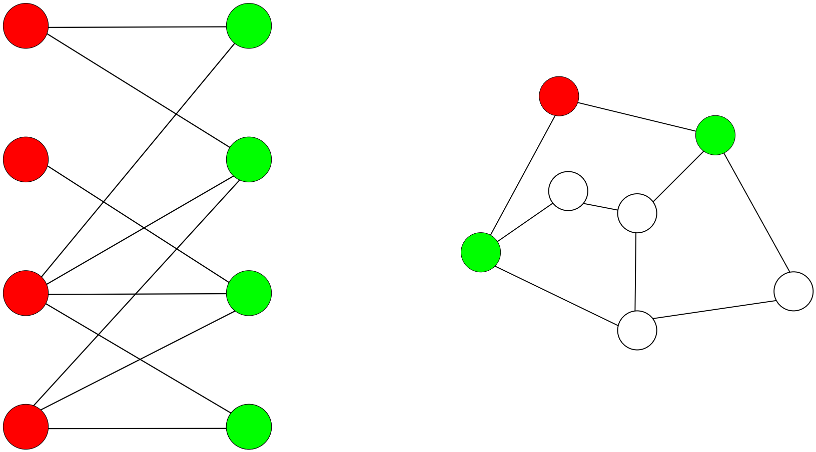


Test Bipartiteness

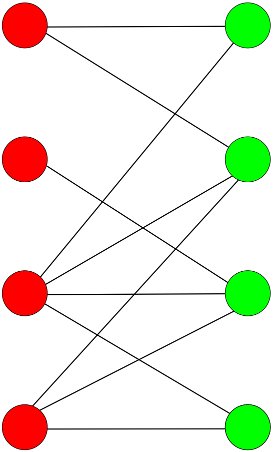


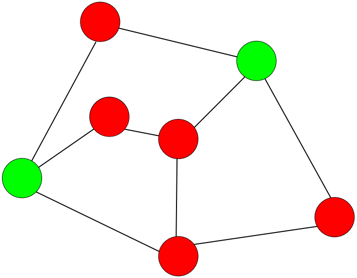


Test Bipartiteness

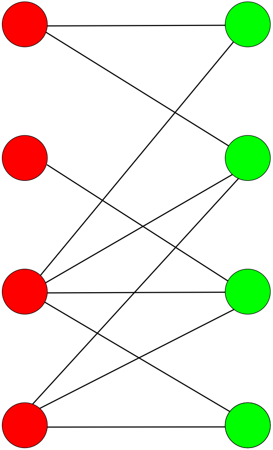


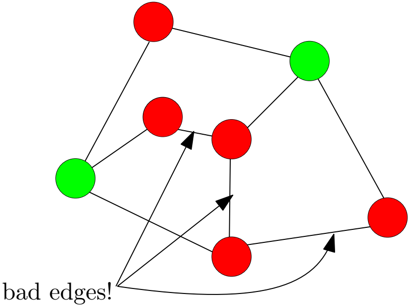
Test Bipartiteness





Test Bipartiteness





Testing Bipartiteness using BFS

BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
$color[s] \leftarrow 0$
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbours u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"
            $color[u] \leftarrow 1 - color[v]$
        else if $color[u] = color[v]$ **then**
            print("G is not bipartite") and exit

Testing Bipartiteness using BFS

mark all vertices as "unvisited"
for each vertex $v \in V$ **do**
    if v is "unvisited" **then**
        test-bipartiteness(v)
print("G is bipartite")

Obs. Running time of algorithm $= O(n+m)$

Outline

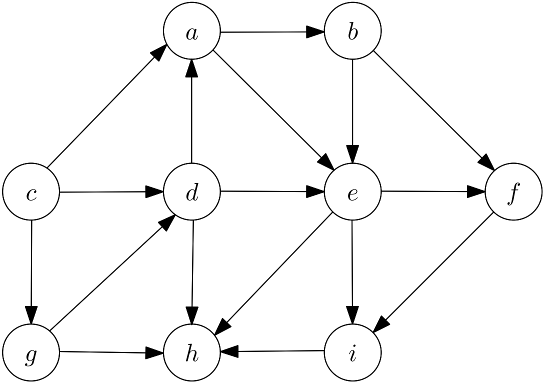
2. Graph Basics
   - Connectivity and Graph Traversal
   - Topological Ordering
   - Finding Bridges

Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$

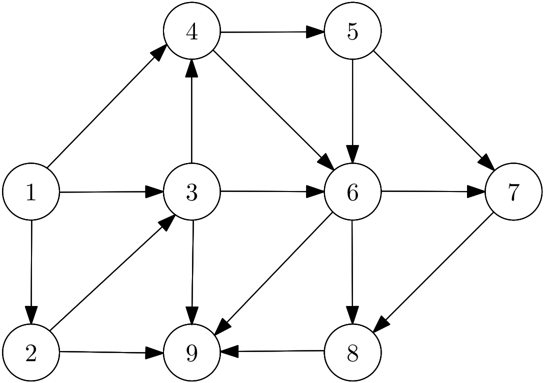


Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

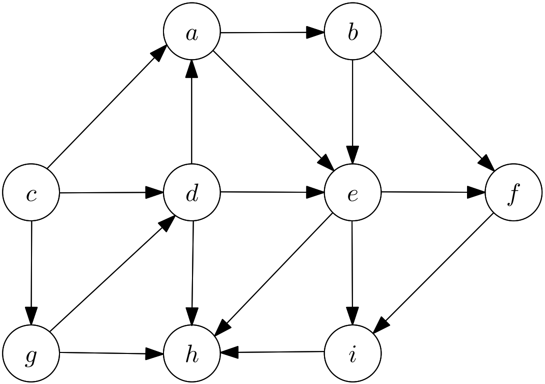
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$



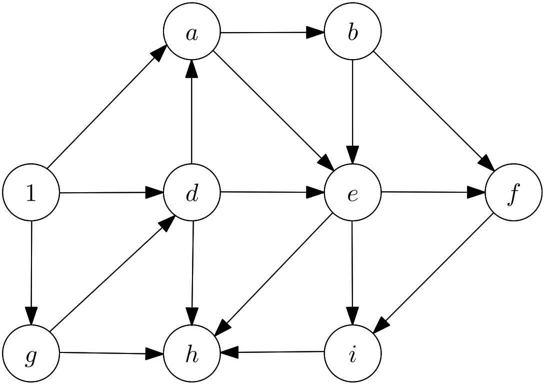
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



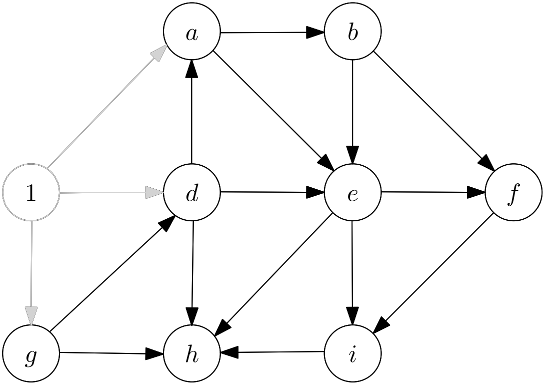
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



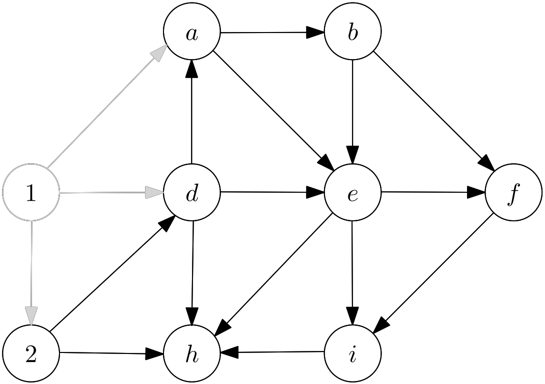
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



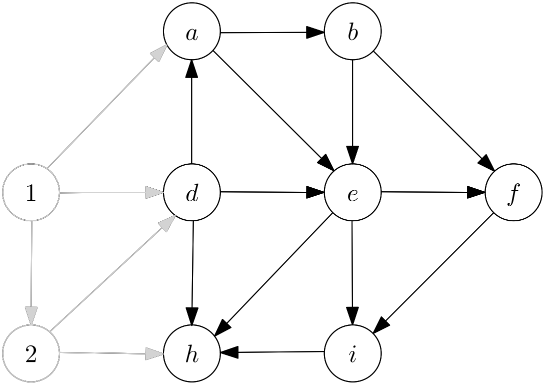
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



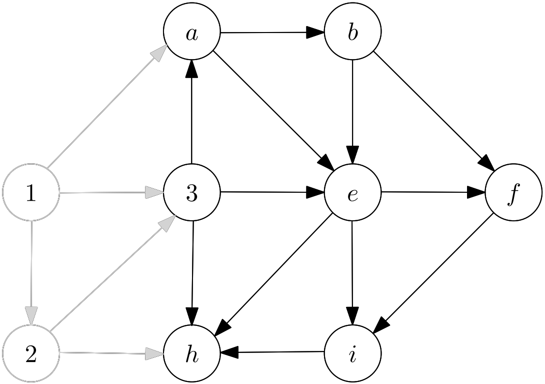
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

Q: How to make the algorithm as efficient as possible?

A:

- Use linked-lists of outgoing edges
- Maintain the in-degree d_v of vertices
- Maintain a queue (or stack) of vertices v with $d_v = 0$

topological-sort(G)

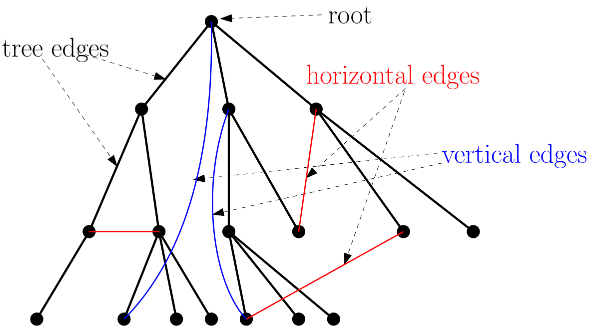
let $d_v \leftarrow 0$ for every $v \in V$
for every $v \in V$ **do**
  for every u such that $(v, u) \in E$ **do**
    $d_u \leftarrow d_u + 1$
$S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
while $S \neq \emptyset$ **do**
  $v \leftarrow$ arbitrary vertex in S, $S \leftarrow S \setminus \{v\}$
  $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
  for every u such that $(v, u) \in E$ **do**
    $d_u \leftarrow d_u - 1$
    if $d_u = 0$ **then** add u to S
if $i < n$ then output "not a DAG"

- S can be represented using a queue or a stack
- Running time $= O(n + m)$

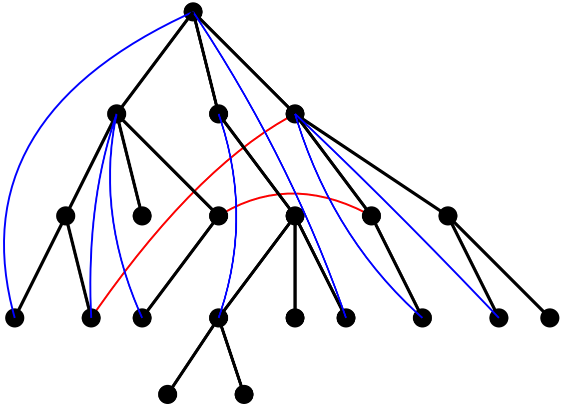
Outline

Vertical and Horizontal Edges

- $G = (V, E)$: connected graph
- $T = (V, E_T)$: rooted spanning tree of G
- $(u, v) \in E \setminus E_T$ is
  - vertical if one of u and v is an ancestor of the other in T,
  - horizontal otherwise.



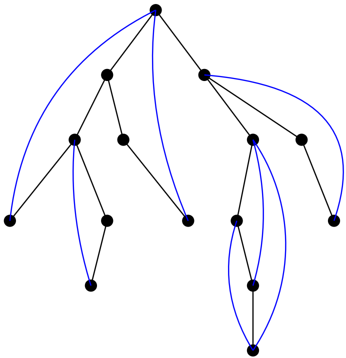
- $G = (V, E)$: connected graph T: a DFS tree for G



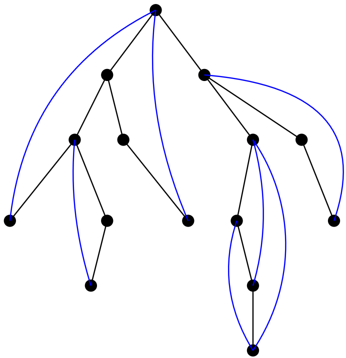
Q: Can there be a horizontal edges (u, v) w.r.t T?

A: No!

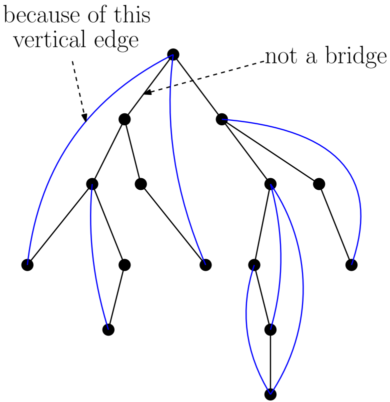
- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges



- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges
- vertical edges: not bridges



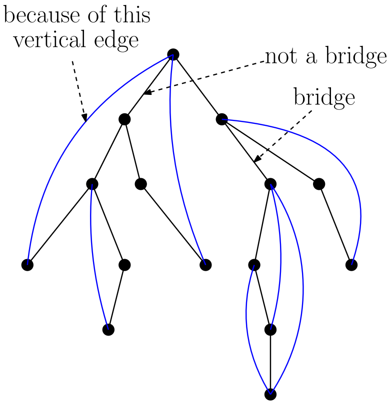
- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges
- vertical edges: not bridges



Lemma

- $(u, v) \in T$, u is parent
- (u, v) is not a bridge $\iff \exists$ vertical edge connecting an (inclusive) descendant of v and an (inclusive) ancestor of u

- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges
- vertical edges: not bridges



Lemma

- $(u, v) \in T$, u is parent
- (u, v) is not a bridge $\iff \exists$ vertical edge connecting an (inclusive) descendant of v and an (inclusive) ancestor of u

Outline

3. Greedy Algorithms
   - Interval Scheduling
   - Scheduling to Minimize Lateness
   - Weighted Completion Time Scheduling
   - Offline Caching
   - Data Compression and Huffman Code

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is "safe" (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Def. A strategy is safe: there is always an optimum solution that agrees with the decision made according to the strategy.

Generic Greedy Algorithm

```
while the instance is non-trivial do
    make the choice using the greedy strategy
    reduce the instance
```

Exchange argument: Proof of Safety of a Strategy

- let S be an arbitrary optimum solution.
- if S is consistent with the greedy choice, done.
- otherwise, show that it can be modified to another optimum solution S' that is consistent with the choice.

- The procedure is not a part of the algorithm.

Outline

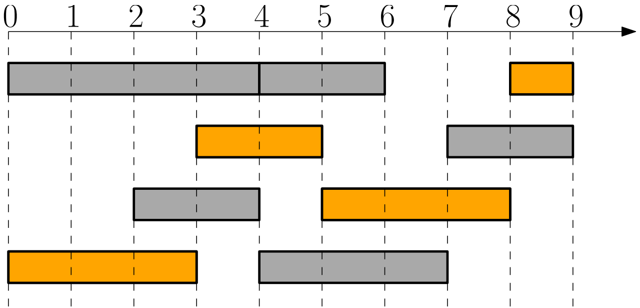
3. Greedy Algorithms
   - Interval Scheduling
   - Scheduling to Minimize Lateness
   - Weighted Completion Time Scheduling
   - Offline Caching
   - Data Compression and Huffman Code

Interval Scheduling

Input: n jobs, job i with start time s_i and finish time f_i

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: A maximum-size subset of mutually compatible jobs

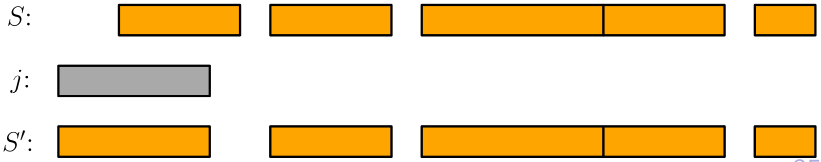


Greedy Algorithm for Interval Scheduling

Lemma It is safe to schedule the job j with the earliest finish time: There is an optimum solution where the job j with the earliest finish time is scheduled.

Proof.

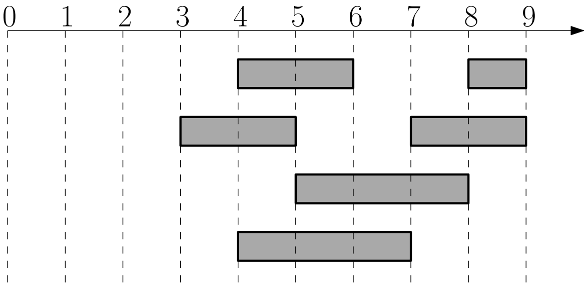
- Take an arbitrary optimum solution S
- If it contains j, done
- Otherwise, replace the first job in S with j to obtain another optimum schedule S'. $\square$



Greedy Algorithm for Interval Scheduling

Lemma It is safe to schedule the job j with the earliest finish time: There is an optimum solution where the job j with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule j?
- Is it another instance of interval scheduling problem? Yes!



Outline

Scheduling to minimize lateness

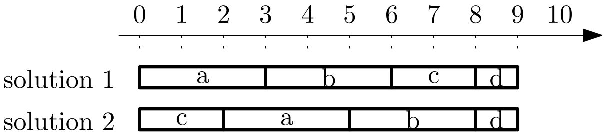
Input: n jobs, each job $j \in [n]$ with a processing time p_j and deadline d_j

Output: schedule jobs on 1 machine, to minimize the max. lateness

C_j: completion time of j lateness $l_j := \max\{C_j - d_j, 0\}$

- Example input:

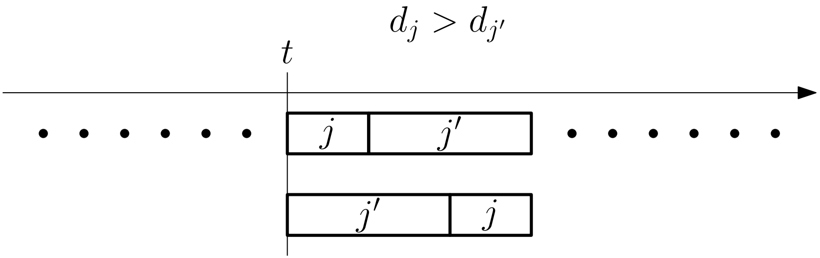
j	a	b	c	d
p_j	3	3	2	1
d_j	5	7	4	8



- solution 1: max lateness $= \max\{0, 3-5, 6-7, 8-4, 9-8\} = 4$
- solution 2: max lateness $= \max\{0, 2-4, 5-5, 8-7, 9-8\} = 1$
- solution 2 is better

Lemma The ascending order of deadlines d_j (the Earliest Deadline First order or the EDF order) is the optimum schedule.

- maximum lateness $= \max\left\{0, \max_{j\in[n]}\{C_j - d_j\}\right\}$.



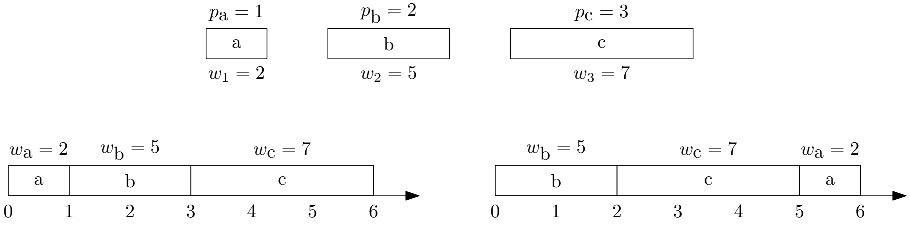
- before: $\max\{t+p_j-d_j, t+p_j+p_{j'}-d_{j'}\} = t+p_j+p_{j'}-d_{j'}$
- after: $\max\{t+p_{j'}-d_{j'}, t+p_j+p_{j'}-d_j\}$
- $p_{j'}-d_{j'} < p_j+p_{j'}-d_{j'}$ and $p_j+p_{j'}-d_j < p_j+p_{j'}-d_{j'}$
- $\max\{t+p_{j'}-d_{j'}, t+p_j+p_{j'}-d_j\} < t+p_j+p_{j'}-d_{j'}$
- after swapping, the maximum of the two terms strictly decreases

Outline

Scheduling to Minimize Weighted Completion Time

Input: A set of n jobs $[n] := \{1, 2, 3, \cdots, n\}$
each job j has a weight w_j and processing time p_j

Output: an ordering of jobs so as to minimize the total weighted completion time of jobs



$\text{cost} = 2 \times 1 + 5 \times 3 + 7 \times 6 = 59$

$\text{cost} = 5 \times 2 + 7 \times 5 + 2 \times 6 = 57$

Def. The Smith ratio of a job is w_j/p_j.

Lemma The descending order of Smith ratios (the Smith rule) is optimum.

Outline

3 Greedy Algorithms
- Interval Scheduling
- Scheduling to Minimize Lateness
- Weighted Completion Time Scheduling
- Offline Caching
- Data Compression and Huffman Code

Offline Caching

- Cache that can store k pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
- Goal: minimize the number of cache misses.

page sequence		cache		
1	✗	1		
5	✗	1	5	
4	✗	1	5	4
2	✗	1	2	4
5	✗	1	2	5
3	✗	1	2	3
2	✓	1	2	3
1	✓	1	2	3

$\text{misses} = 6$

Optimum Offline Caching

Furthest-in-Future (FF)

- Algorithm: every time, evict the page that is not requested until furthest in the future, if we need to evict one.
- The algorithm is not an online algorithm, since the decision at a step depends on the request sequence in the future.

Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.
- For each page p, use a linked list (or an array with dynamic size) to store the time steps in which p is requested.
  - We can find the next time a page is requested easily.
- Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.

Outline

3. Greedy Algorithms
    - Interval Scheduling
    - Scheduling to Minimize Lateness
    - Weighted Completion Time Scheduling
    - Offline Caching
    - Data Compression and Huffman Code

Encoding Letters Using Bits

- 8 letters a, b, c, d, e, f, g, h in a language
- need to encode a message using bits
- idea: use 3 bits per letter

a	b	c	d	e	f	g	h
000	001	010	011	100	101	110	111

$$deacfg \to 011100000010101110$$

Q: Can we have a better encoding scheme?

- Seems unlikely: must use 3 bits per letter

Q: What if some letters appear more frequently than the others?

Q: If some letters appear more frequently than the others, can we have a better encoding scheme?

A: Using variable-length encoding scheme might be more efficient.

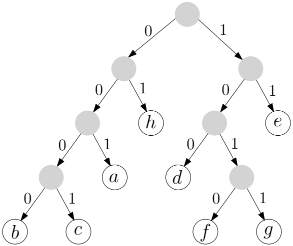
Idea

- using fewer bits for letters that are more frequently used, and more bits for letters that are less frequently used.

Prefix Codes

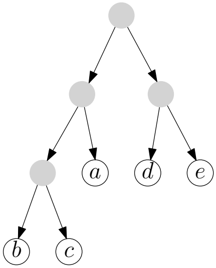
Def. A prefix code for a set S of letters is a function $\gamma : S \to \{0,1\}^*$ such that for two distinct $x, y \in S$, $\gamma(x)$ is not a prefix of $\gamma(y)$.

a	b	c	d
001	0000	0001	100
e	f	g	h
11	1010	1011	01

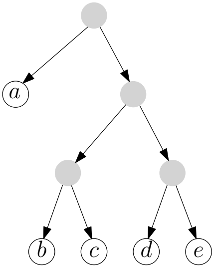


example

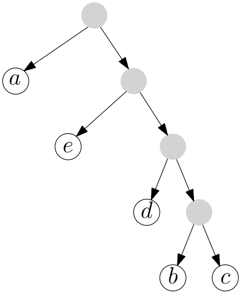
letters	a	b	c	d	e	
frequencies	18	3	4	6	10	
scheme 1 length	2	3	3	2	2	total $=$ 89
scheme 2 length	1	3	3	3	3	total $=$ 87
scheme 3 length	1	4	4	3	2	total $=$ 84



scheme 1



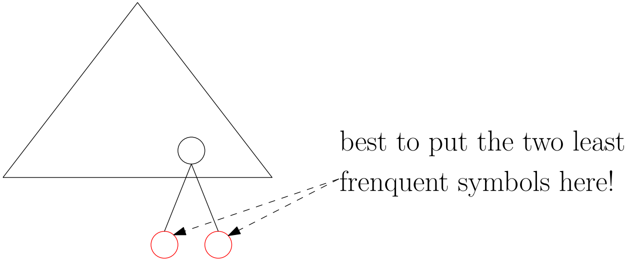
scheme 2



scheme 3

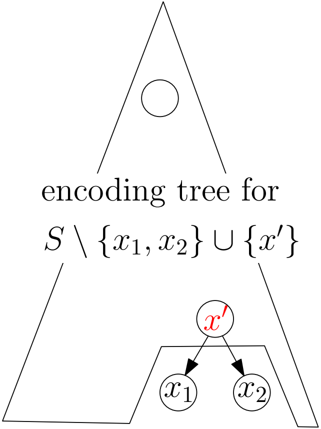
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the "structure" of the optimum encoding tree
- There are two deepest leaves that are brothers



Lemma It is safe to make the two least frequent letters brothers.

- f_x: the frequency of the letter x in the support.
- x_1 and x_2: the two letters we decided to put together.
- d_x the depth of letter x in our output encoding tree.



Def: $f_{x'} = f_{x_1} + f_{x_2}$

$$
\begin{aligned}
&\sum_{x \in S} f_x d_x \\
= &\sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
= &\sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
= &\sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1) \\
= &\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\end{aligned}
$$

In order to minimize

$$\sum_{x \in S} f_x d_x,$$

we need to minimize

$$\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x,$$

subject to that d is the depth function for an encoding tree of $S \setminus \{x_1, x_2\}$.

- This is exactly the best prefix codes problem, with letters $S \setminus \{x_1, x_2\} \cup \{x'\}$ and frequency vector f!

Outline

4. Divide-and-Conquer
   - Counting Inversions
   - Solving Recurrences
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
   - Polynomial Multiplication
   - Strassen's Algorithm for Matrix Multiplication
   - Finding Closest Pair of Points in 2D Euclidean Space
   - Computing n-th Fibonacci Number

Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

Outline

4. Divide-and-Conquer

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Count Inversions between B and C

- Procedure that merges B and C and counts inversions between B and C at the same time

merge-and-count(B, C, n_1, n_2)

```
count ← 0;
A ← array of size n1 + n2; i ← 1; j ← 1
while i ≤ n1 or j ≤ n2 do
    if j > n2 or (i ≤ n1 and B[i] ≤ C[j]) then
        A[i + j − 1] ← B[i]; i ← i + 1
        count ← count + (j − 1)
    else
        A[i + j − 1] ← C[j]; j ← j + 1
return (A, count)
```

Sort and Count Inversions in A

- A procedure that returns the sorted array of A and counts the number of inversions in A:

sort-and-count(A, n)

- Divide: trivial
- Conquer: 4, 5
- Combine: 6, 7

1: **if** $n = 1$ **then**
2:     **return** $(A, 0)$
3: **else**
4:     $(B, m_1) \leftarrow$ sort-and-count$\Big(A\big[1..\lfloor n/2\rfloor\big], \lfloor n/2\rfloor\Big)$
5:     $(C, m_2) \leftarrow$ sort-and-count$\Big(A\big[\lfloor n/2\rfloor + 1..n\big], \lceil n/2\rceil\Big)$
6:     $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2\rfloor, \lceil n/2\rceil)$
7:     **return** $(A, m_1 + m_2 + m_3)$

sort-and-count(A, n)

```
if n = 1 then
    return (A, 0)
else
    (B, m_1) ← sort-and-count(A[1..⌊n/2⌋], ⌊n/2⌋)
    (C, m_2) ← sort-and-count(A[⌊n/2⌋+1..n], ⌈n/2⌉)
    (A, m_3) ← merge-and-count(B, C, ⌊n/2⌋, ⌈n/2⌉)
    return (A, m_1 + m_2 + m_3)
```

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$
- Running time $= O(n \log n)$

Outline

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } c < \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^c) & \text{if } c > \log_b a \end{cases}$$

Outline

4. Divide-and-Conquer
   - Counting Inversions
   - Solving Recurrences
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
   - Polynomial Multiplication
   - Strassen's Algorithm for Matrix Multiplication
   - Finding Closest Pair of Points in 2D Euclidean Space
   - Computing n-th Fibonacci Number

Quicksort vs Merge-Sort

	Merge Sort	**Quicksort**
Divide	Trivial	Separate small and big numbers
Conquer	Recurse	Recurse
Combine	Merge 2 sorted arrays	Trivial

Quicksort

quicksort(A, n)

```
if n ≤ 1 then return A
x ← lower median of A
A_L ← array of elements in A that are less than x            \\ Divide
A_R ← array of elements in A that are greater than x         \\ Divide
B_L ← quicksort(A_L, length of A_L)                          \\ Conquer
B_R ← quicksort(A_R, length of A_R)                          \\ Conquer
t ← number of times x appear A
return concatenation of B_L, t copies of x, and B_R
```

- Recurrence $T(n) \leq 2T(n/2) + O(n)$
- Running time $= O(n \log n)$

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

A:

1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
2. Choose a pivot randomly and pretend it is the median (it is practical)

Quicksort Using A Random Pivot

quicksort(A, n)

if $n \le 1$ **then return** A
$x \leftarrow$ a random element of A (x is called a pivot)
$A_L \leftarrow$ array of elements in A that are less than x \\ Divide
$A_R \leftarrow$ array of elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return concatenation of B_L, t copies of x, and B_R

Lemma The expected running time of the algorithm is $O(n \log n)$.

Outline

4 Divide-and-Conquer

- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen's Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing n-th Fibonacci Number

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

- To sort, we are only allowed to compare two elements
- We can not use "internal structures" of the elements

Number of comparisons is at least $\log_2 n! = \Theta(n \log n)$

Outline

4. Divide-and-Conquer
   - Counting Inversions
   - Solving Recurrences
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
   - Polynomial Multiplication
   - Strassen's Algorithm for Matrix Multiplication
   - Finding Closest Pair of Points in 2D Euclidean Space
   - Computing n-th Fibonacci Number

Selection Problem

Input: a set A of n numbers, and $1 \le i \le n$

Output: the i-th smallest number in A

- Sorting solves the problem in time $O(n \log n)$.
- Our goal: $O(n)$ running time

Selection Algorithm with Median Finder

selection(A, n, i)

if $n = 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \leq A_L$.size **then**
    return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R$.size **then**
    return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
    return x

- Recurrence for selection: $T(n) = T(n/2) + O(n)$
- Solving recurrence: $T(n) = O(n)$

Randomized Selection Algorithm

selection(A, n, i)

if $n = 1$ **thenreturn** A
$x \leftarrow$ random element of A (called pivot)
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \leq A_L$.size **then**
  return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R$.size **then**
  return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
  return x

- expected running time $= O(n)$

Outline

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

$$\begin{aligned} pq &= \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right) \\ &= p_H q_H x^n + \left(p_H q_L + p_L q_H\right)x^{n/2} + p_L q_L \end{aligned}$$

- $p_H q_L + p_L q_H = (p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L$

Divide-and-Conquer for Polynomial Multiplication

$$
\begin{aligned}
r_H &= \mathsf{multiply}(p_H, q_H)\\
r_L &= \mathsf{multiply}(p_L, q_L)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{multiply}(p, q) = {} & r_H \times x^n \\
& + \big(\mathsf{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L\big) \times x^{n/2} \\
& + r_L
\end{aligned}
$$

- Solving Recurrence: $T(n) = 3T(n/2) + O(n)$
- $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Outline

4 Divide-and-Conquer

- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen's Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing n-th Fibonacci Number

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

$$A = \begin{array}{|c|c|l} \multicolumn{1}{c}{} & \multicolumn{1}{c}{\overbrace{\qquad}^{n/2}} & \\ \hline A_{11} & A_{12} & \}\, n/2 \\ \hline A_{21} & A_{22} & \\ \hline \end{array} \qquad B = \begin{array}{|c|c|l} \multicolumn{1}{c}{} & \multicolumn{1}{c}{\overbrace{\qquad}^{n/2}} & \\ \hline B_{11} & B_{12} & \}\, n/2 \\ \hline B_{21} & B_{22} & \\ \hline \end{array}$$

- $C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$

- $M_1 \leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- $M_2 \leftarrow (A_{21} + A_{22}) \times B_{11}$
- $M_3 \leftarrow A_{11} \times (B_{12} - B_{22})$
- $M_4 \leftarrow A_{22} \times (B_{21} - B_{11})$
- $M_5 \leftarrow (A_{11} + A_{12}) \times B_{22}$
- $M_6 \leftarrow (A_{21} - A_{11}) \times (B_{11} + B_{12})$
- $M_7 \leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22})$

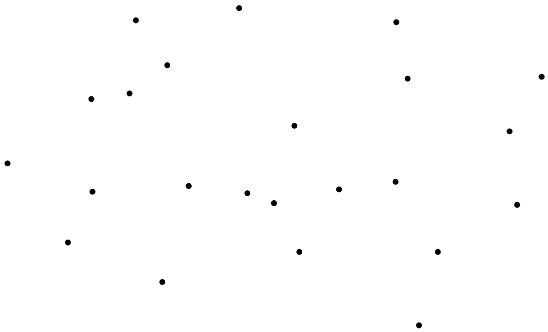
- $C_{11} \leftarrow M_1 + M_4 - M_5 + M_7$
- $C_{12} \leftarrow M_3 + M_5$
- $C_{21} \leftarrow M_2 + M_4$
- $C_{22} \leftarrow M_1 - M_2 + M_3 + M_6$

Outline

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

Output: the pair of points that are closest

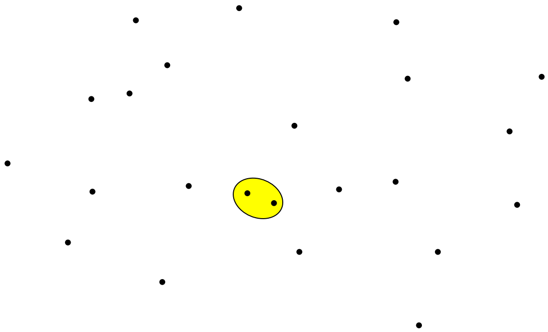


- Trivial algorithm: $O(n^2)$ running time

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

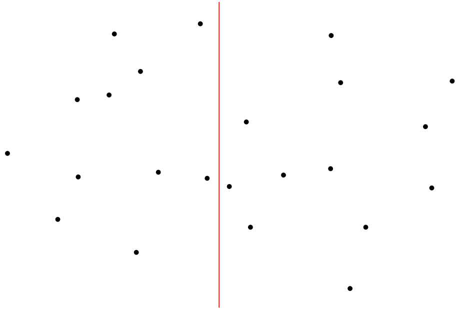
Output: the pair of points that are closest



- Trivial algorithm: $O(n^2)$ running time

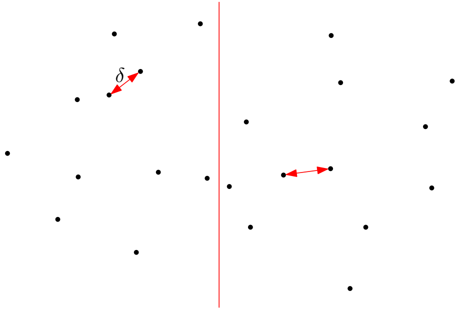
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half



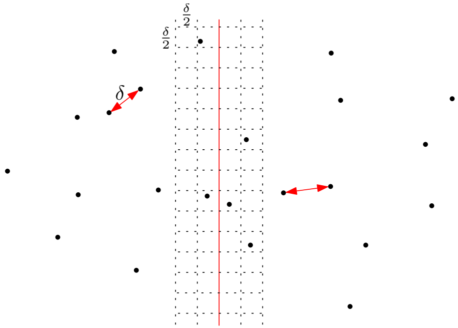
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half

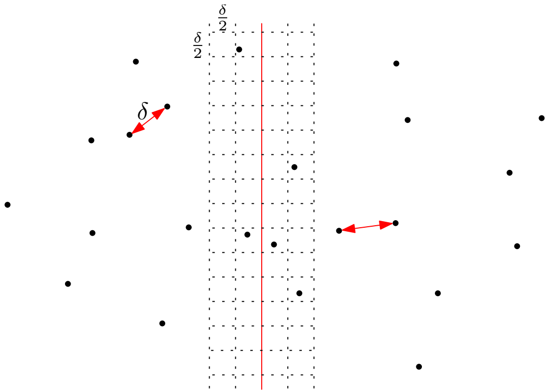


Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half



Divide-and-Conquer Algorithm for Closest Pair



- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- Implementation: Sort points inside the stripe according to y-coordinates
- For every point, consider $O(1)$ points around it in the order

Outline

4. Divide-and-Conquer
   - Counting Inversions
   - Solving Recurrences
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
   - Polynomial Multiplication
   - Strassen's Algorithm for Matrix Multiplication
   - Finding Closest Pair of Points in 2D Euclidean Space
   - Computing n-th Fibonacci Number

Fibonacci Numbers

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$

n-th Fibonacci Number

Input: integer $n > 0$

Output: F_n

Computing F_n: Even Better Algorithm

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} F_{n-2} \\ F_{n-3} \end{pmatrix}$$

$$\cdots$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

power(n)

if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$R \leftarrow$ power($\lfloor n/2 \rfloor$)
$R \leftarrow R \times R$
if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
return R

Fib(n)

if $n = 0$ then return 0
$M \leftarrow$ power($n - 1$)
return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$
- $T(n) = O(\log n)$

Outline

5. Dynamic Programming
   - Weighted Interval Scheduling
   - Segmented Least Squares
   - Subset Sum and Knapsack Problems
   - Longest Common Subsequence
   - Shortest Paths in Directed Acyclic Graphs
   - Matrix Chain Multiplication
   - Optimum Binary Search Tree

Outline

5. Dynamic Programming
   - Weighted Interval Scheduling
   - Segmented Least Squares
   - Subset Sum and Knapsack Problems
   - Longest Common Subsequence
   - Shortest Paths in Directed Acyclic Graphs
   - Matrix Chain Multiplication
   - Optimum Binary Search Tree

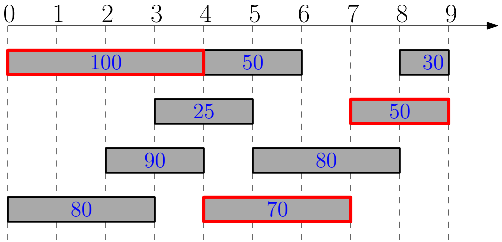
Weighted Interval Scheduling

Input: n jobs, job i with start time s_i and finish time f_i

each job has a weight (or value) $v_i > 0$

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-weight subset of mutually compatible jobs



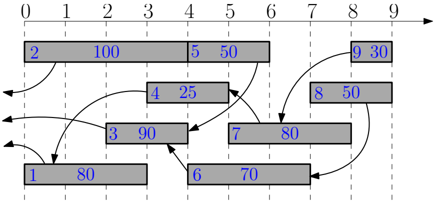
Optimum value $= 220$

Designing a Dynamic Programming Algorithm

- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$



Dynamic Programming

sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    $opt[i] \leftarrow \max\{opt[i-1], v_i + opt[p_i]\}$

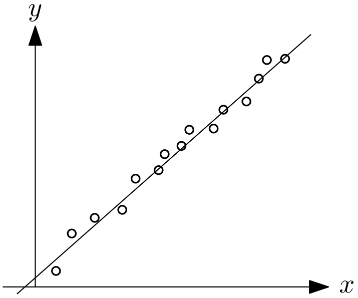
- Running time sorting: $O(n \lg n)$
- Running time for computing p: $O(n \lg n)$ via binary search
- Running time for computing $opt[n]$: $O(n)$

Outline

Linear Regression

- $P = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}, x_1 < x_2 < \cdots < x_n$
- $L : y = ax + b$

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Linear Regression

- find L, minimize $\text{Error}(L, P)$

$$a := \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}$$

$$b := \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

Input: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n), x_1 < x_2 < \cdots < x_n$

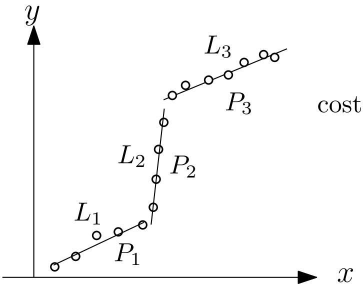
penalty parameter $C > 0$

Output: partition into $k \geq 1$ (k unknown) segments,

minimize cost := error + penalty

error: sum of squared error over all the k segments

penalty: kC

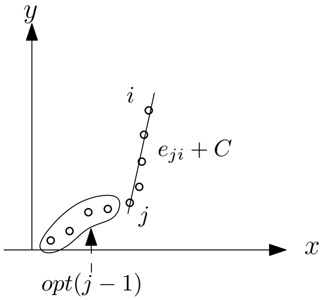


$$\begin{aligned}\text{cost} = &\ \text{error}(L_1, P_1)\\ &+\text{error}(L_2, P_2)\\ &+\text{error}(L_3, P_3)\\ &+3C\end{aligned}$$

Dynamic Programming

- $e_{ji}, 1 \leq j \leq i \leq n$: minimum error for $(x_j, y_j), \cdots, (x_i, y_i)$ using 1 line
- $opt[i]$: minimum cost for the instance with first i points

$$opt[i] = \begin{cases} 0 & \text{if } i = 0 \\ \min_{j:1 \leq j \leq i}(opt[j-1] + e_{ji}) + C & \text{if } i \geq 1 \end{cases}$$



- running time $= O(n^2)$.

Outline

Subset Sum Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

Output: a subset S of items that

$$\text{maximizes} \sum_{i \in S} w_i \qquad \text{s.t.} \sum_{i \in S} w_i \leq W.$$

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \begin{Bmatrix} opt[i-1, W'] \\ opt[i-1, W'-w_i] + w_i \end{Bmatrix} & i > 0, w_i \leq W' \end{cases}$$

Running Time of Algorithm

```
for W' ← 0 to W do
    opt[0, W'] ← 0
for i ← 1 to n do
    for W' ← 0 to W do
        opt[i, W'] ← opt[i − 1, W']
        if w_i ≤ W' and opt[i − 1, W' − w_i] + w_i ≥ opt[i, W'] then
            opt[i, W'] ← opt[i − 1, W' − w_i] + w_i
return opt[n, W]
```

- Running time is $O(nW)$
- Running time is pseudo-polynomial because it depends on value of the input integers.

Knapsack Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} v_i \qquad \text{s.t.} \sum_{i \in S} w_i \le W.$$

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \le W' \end{cases}$$

Outline

5 Dynamic Programming
- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree

- $A = bacdca$ $\qquad C = adca$
- C is a subsequence of A

Def. Given two sequences $A[1 .. n]$ and $C[1 .. t]$ of letters, C is called a subsequence of A if there exists integers $1 \leq i_1 < i_2 < i_3 < \ldots < i_t \leq n$ such that $A[i_j] = C[j]$ for every $j = 1, 2, 3, \cdots, t$.

Longest Common Subsequence

Input: $A[1 .. n]$ and $B[1 .. m]$

Output: the longest common subsequence of A and B

Example:

- $A = \text{`}bacdca\text{'}$
- $B = \text{`}adbcda\text{'}$
- $\text{LCS}(A, B) = \text{`}adca\text{'}$

Dynamic Programming for LCS

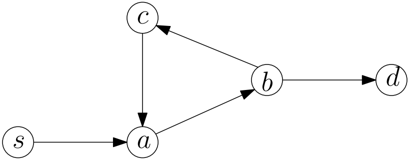
- $opt[i,j], 0 \le i \le n, 0 \le j \le m$: length of longest common sub-sequence of $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ or $j = 0$, then $opt[i,j] = 0$.
- if $i > 0, j > 0$, then

$$opt[i,j] = \begin{cases} opt[i-1, j-1] + 1 & \text{if } A[i] = B[j] \\ \max \begin{cases} opt[i-1, j] \\ opt[i, j-1] \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

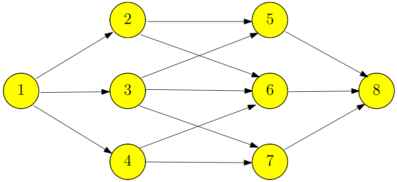
Outline

Directed Acyclic Graphs

Def. A directed acyclic graph (DAG) is a directed graph without (directed) cycles.



not a DAG



a DAG

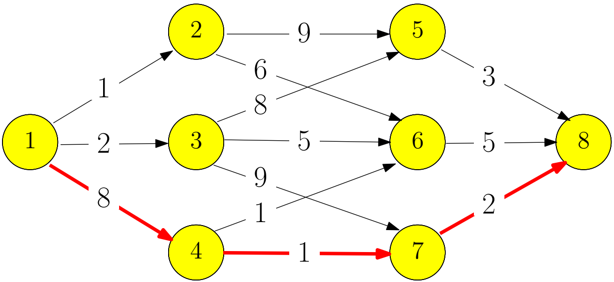
Lemma A directed graph is a DAG if and only its vertices can be topologically sorted.

Shortest Paths in DAG

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the shortest path from 1 to i, for every $i \in V$



Shortest Paths in DAG

- $f[i]$: length of the shortest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ \min_{j:(j,i)\in E} \{f(j) + w(j,i)\} & i = 2, 3, \cdots, n \end{cases}$$

Outline

5 Dynamic Programming

- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree

Matrix Chain Multiplication

Matrix Chain Multiplication

Input: n matrices $A_1, A_2, \cdots, A_n$ of sizes $r_1 \times c_1, r_2 \times c_2, \cdots, r_n \times c_n$, such that $c_i = r_{i+1}$ for every $i = 1, 2, \cdots, n-1$.

Output: the order of computing $A_1 A_2 \cdots A_n$ with the minimum number of multiplications

Fact Multiplying two matrices of size $r \times k$ and $k \times c$ takes $r \times k \times c$ multiplications.

- $opt[i, j]$: the minimum cost of computing $A_i A_{i+1} \cdots A_j$

$$opt[i, j] = \begin{cases} 0 & i = j \\ \min_{k: i \le k < j} \left(opt[i, k] + opt[k+1, j] + r_i c_k c_j \right) & i < j \end{cases}$$

Outline

5 Dynamic Programming

- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree

Optimum Binary Search Tree

- n elements $e_1 < e_2 < e_3 < \cdots < e_n$
- e_i has frequency f_i
- goal: build a binary search tree for $\{e_1, e_2, \cdots, e_n\}$ with the minimum accessing cost:

$$\sum_{i=1}^{n} f_i \times (\text{depth of } e_i \text{ in the tree})$$

- $opt[i,j]$: the optimum cost for the instance $(f_i, f_{i+1}, \cdots, f_j)$
- In general, $opt[i,j] =$

$$\begin{cases} 0 & \text{if } i = j+1 \\ \min_{k: i \le k \le j} \big(opt[i, k-1] + opt[k+1, j]\big) + \sum_{\ell=i}^{j} f_\ell & \text{if } i \le j \end{cases}$$

Outline

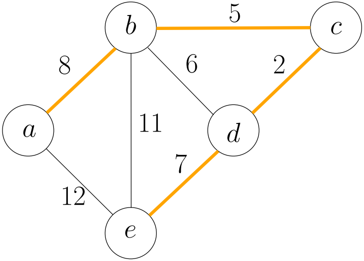
6. Graph Algorithms
   - Minimum Spanning Tree, Kruskal's Algorithm, Prim's Algorithm
   - Shortest Path Algorithms
   - Minimum Cost Arborescence

Outline

Minimum Spanning Tree (MST) Problem

Input: Graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$

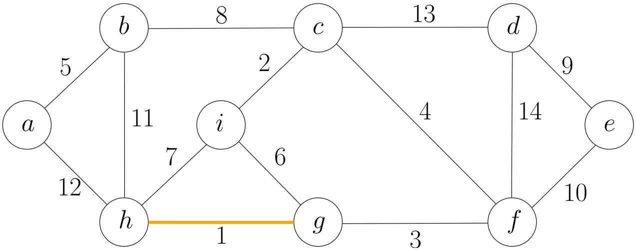
Output: the spanning tree T of G with the minimum total weight



Two Classic Greedy Algorithms for MST

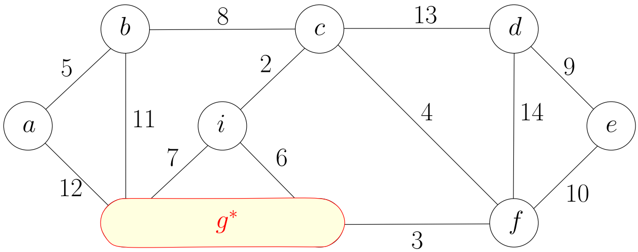
- Kruskal's Algorithm
- Prim's Algorithm

Kruskal's Algorithm

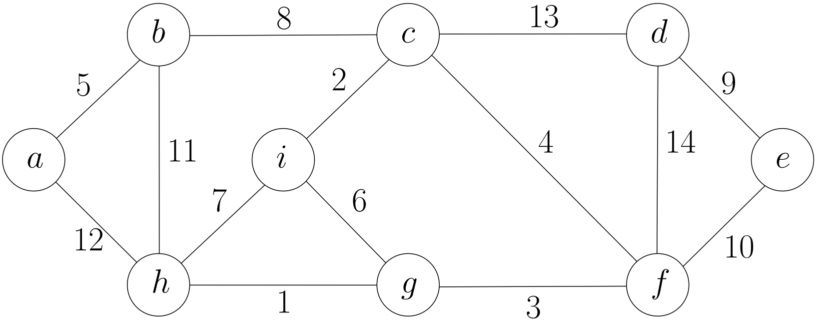


Lemma It is safe to include the lightest edge in the MST.

Residual Problem by Contraction

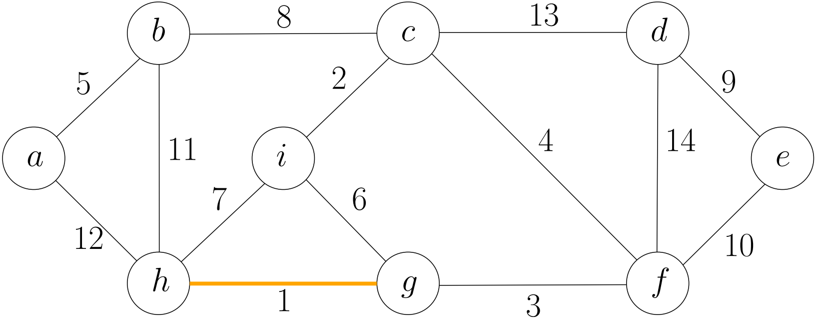


Kruskal's Algorithm: Example



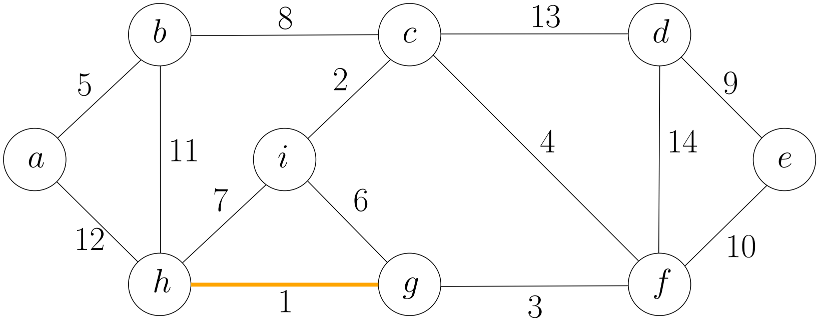
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$

Kruskal's Algorithm: Example



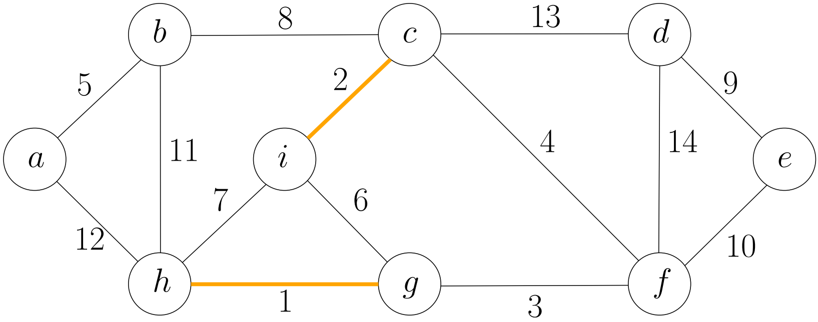
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$

Kruskal's Algorithm: Example



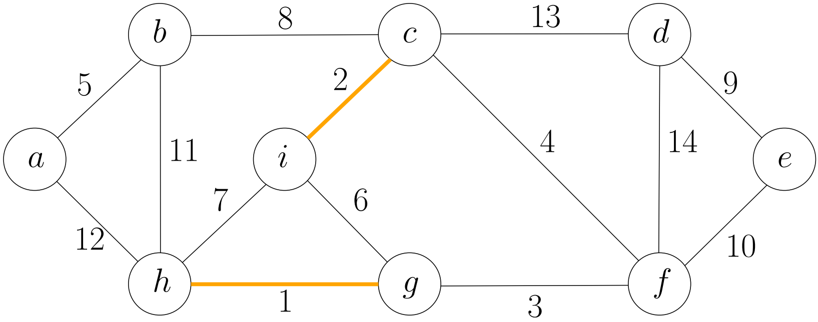
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$

Kruskal's Algorithm: Example



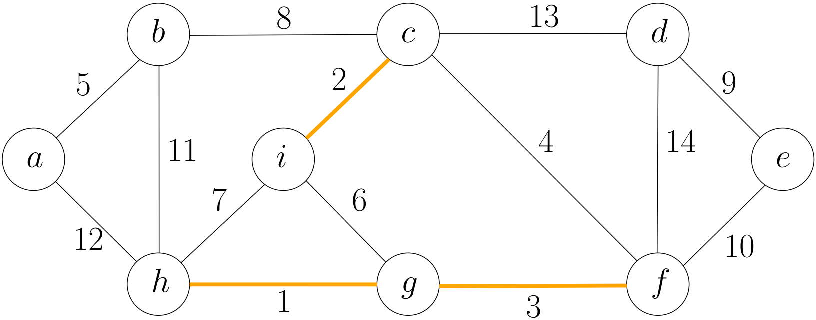
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$

Kruskal's Algorithm: Example



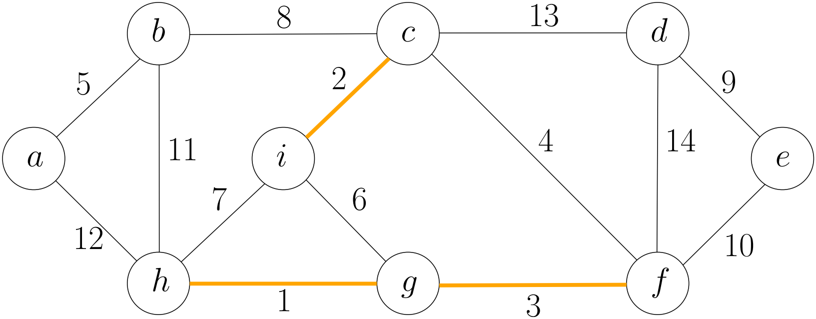
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$

Kruskal's Algorithm: Example



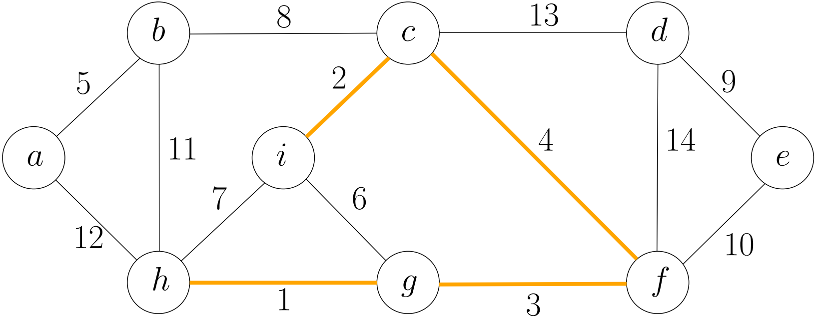
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$

Kruskal's Algorithm: Example



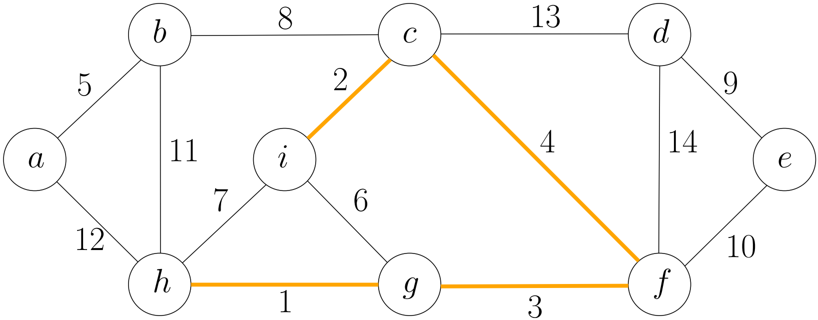
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Kruskal's Algorithm: Example



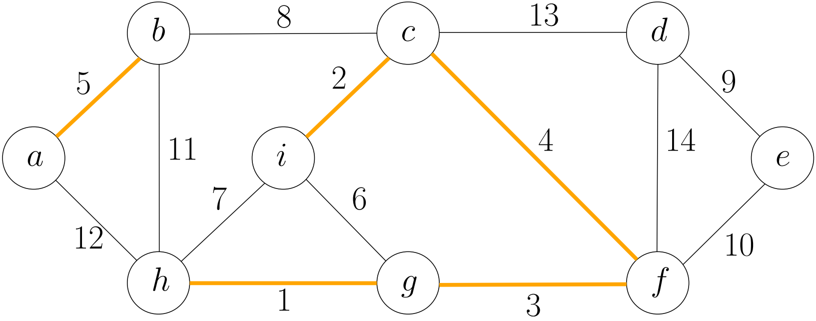
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Kruskal's Algorithm: Example



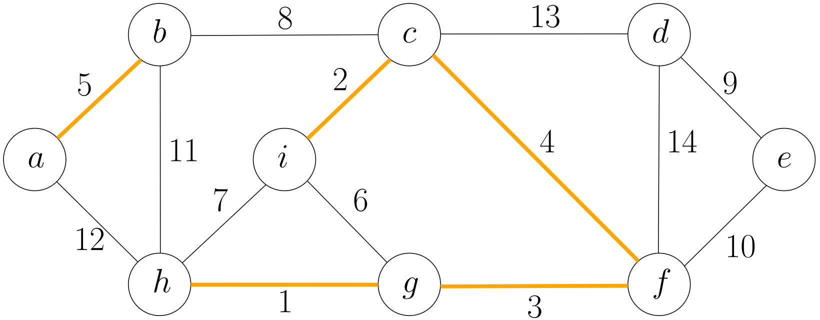
Sets: $\{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



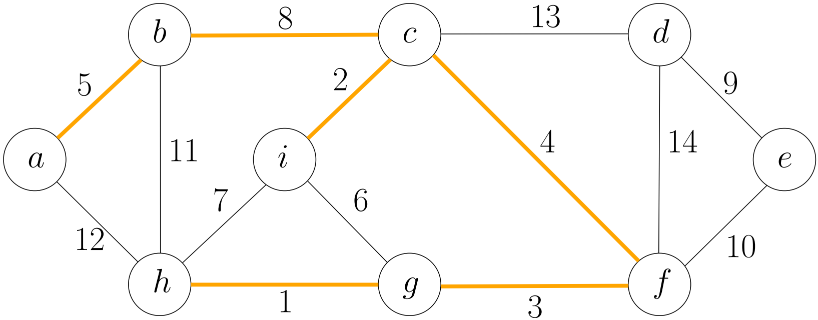
Sets: $\{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



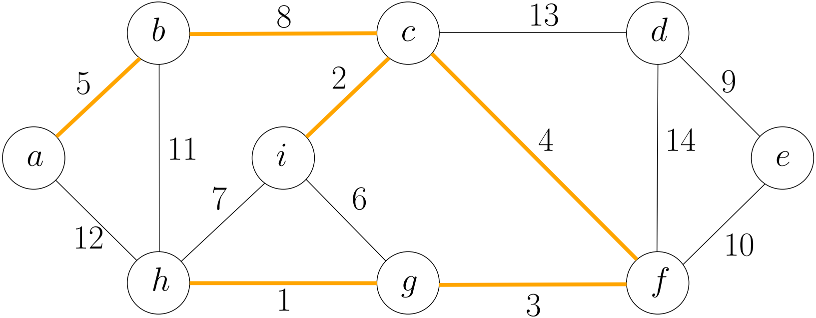
Sets: $\{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



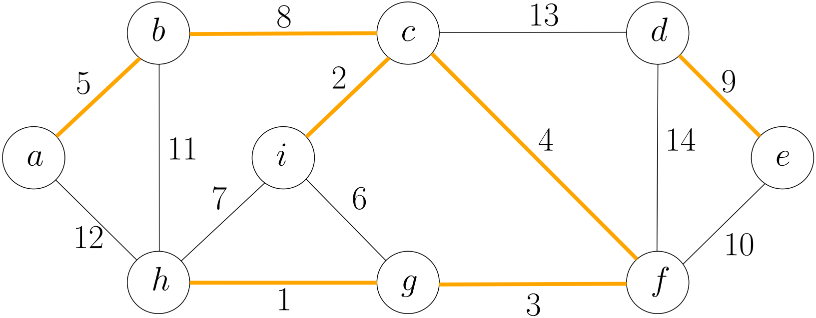
Sets: $\{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



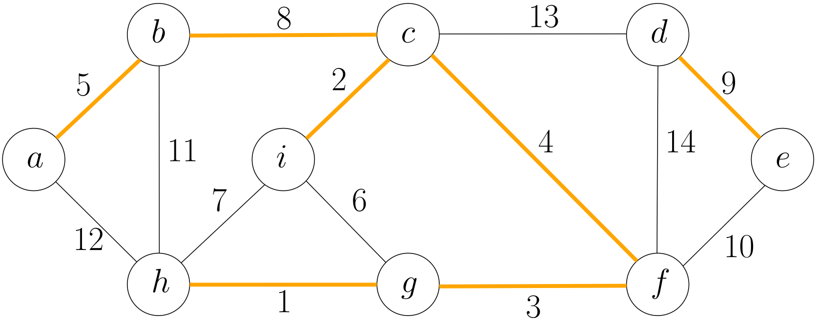
Sets: $\{a, b, c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



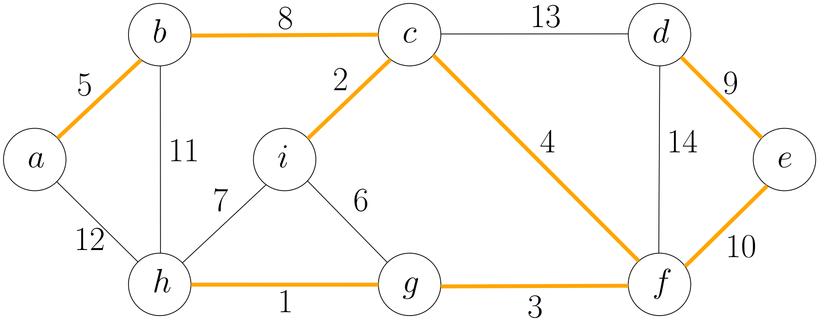
Sets: $\{a, b, c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



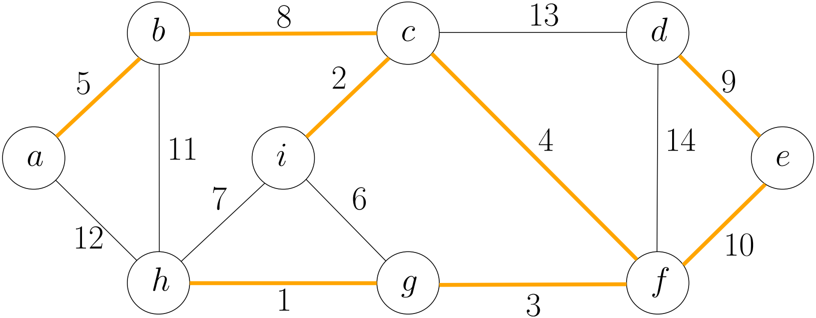
Sets: $\{a, b, c, i, f, g, h\}, \{d, e\}$

Kruskal's Algorithm: Example



Sets: $\{a, b, c, i, f, g, h\}, \{d, e\}$

Kruskal's Algorithm: Example



Sets: $\{a, b, c, i, f, g, h, d, e\}$

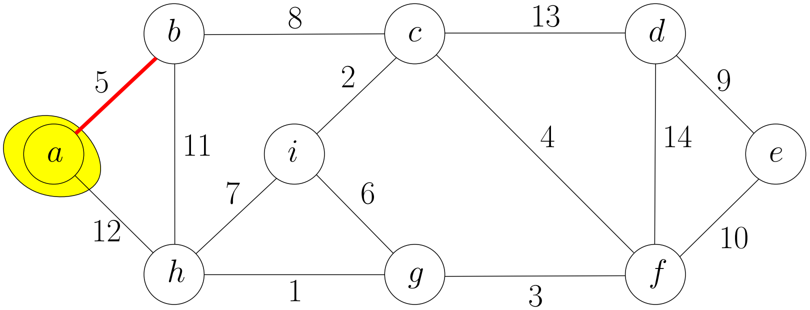
Running Time of Kruskal's Algorithm

MST-Kruskal(G, w)

```
1: F ← ∅
2: 𝒮 ← {{v} : v ∈ V}
3: sort the edges of E in non-decreasing order of weights w
4: for each edge (u, v) ∈ E in the order do
5:     S_u ← the set in 𝒮 containing u
6:     S_v ← the set in 𝒮 containing v
7:     if S_u ≠ S_v then
8:         F ← F ∪ {(u, v)}
9:         𝒮 ← 𝒮 \ {S_u} \ {S_v} ∪ {S_u ∪ S_v}
10: return (V, F)
```

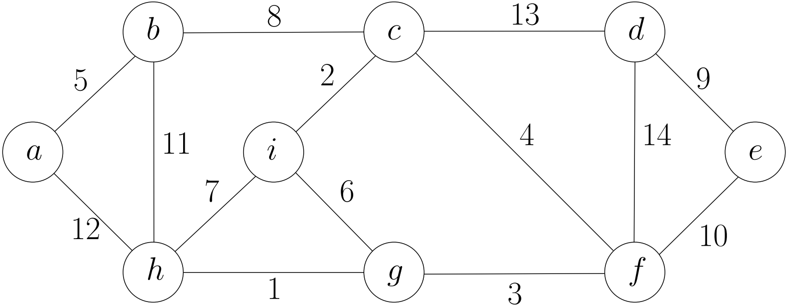
Use union-find data structure to support 2, 5, 6, 7, 9.

Prim's Algorithm

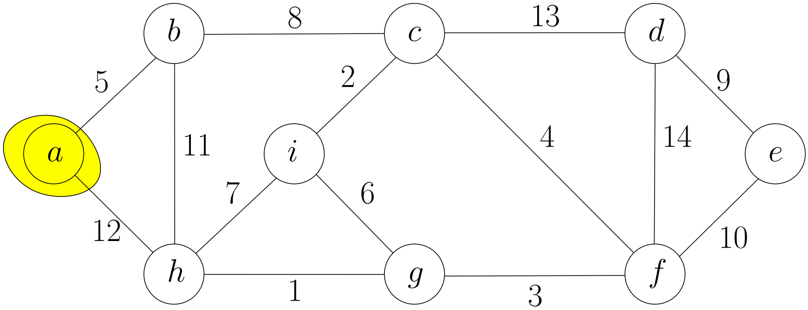


- Greedy strategy for Prim's algorithm: choose the lightest edge incident to a.

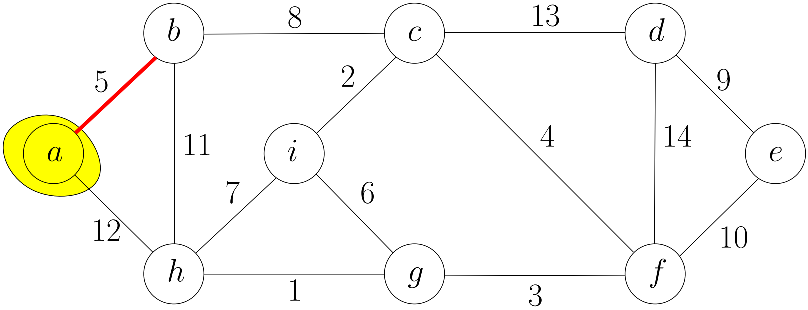
Prim's Algorithm: Example



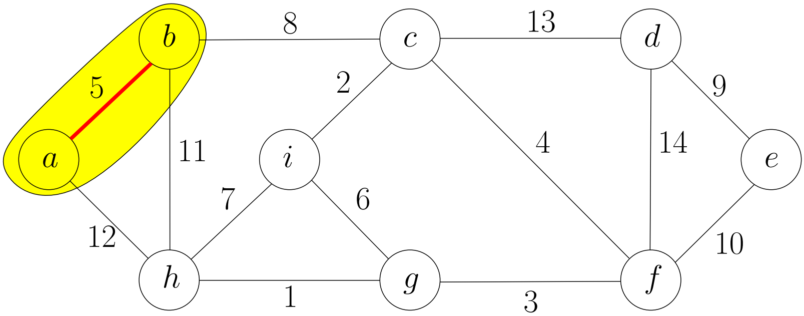
Prim's Algorithm: Example



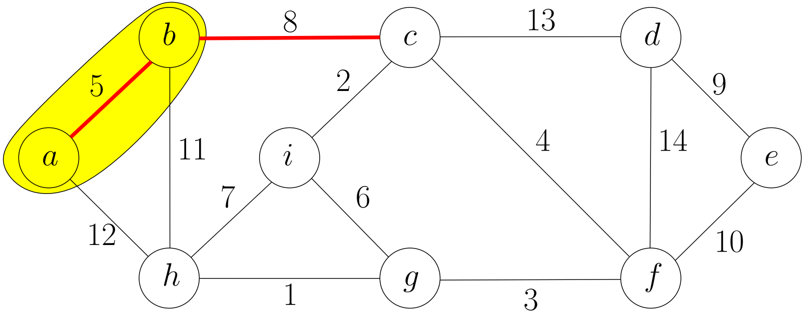
Prim's Algorithm: Example



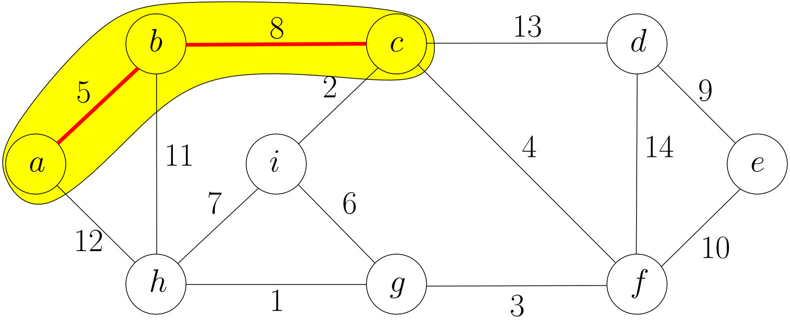
Prim's Algorithm: Example



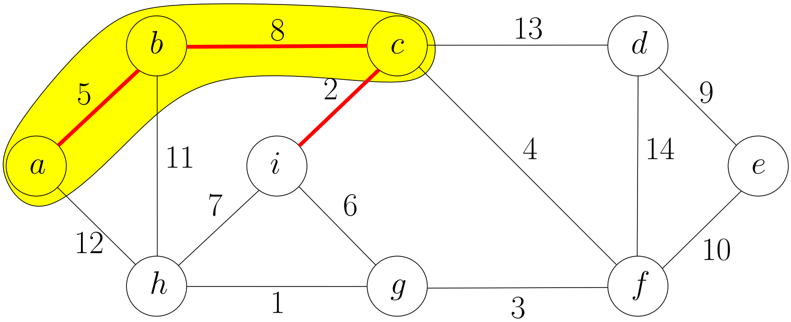
Prim's Algorithm: Example



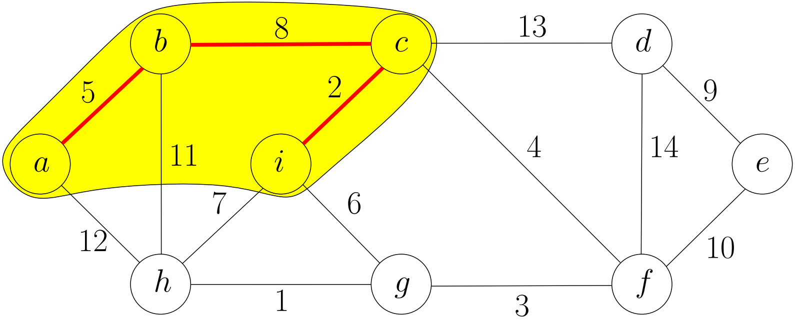
Prim's Algorithm: Example



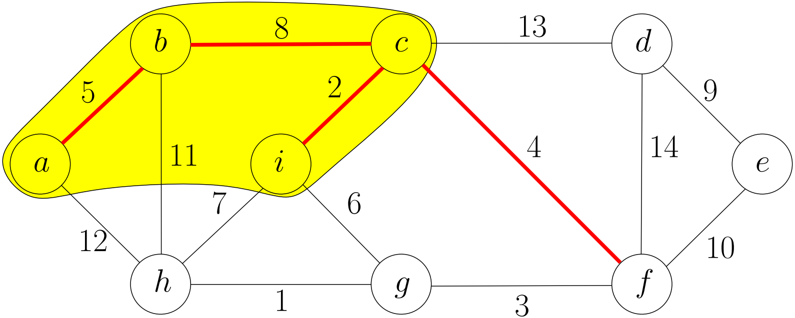
Prim's Algorithm: Example



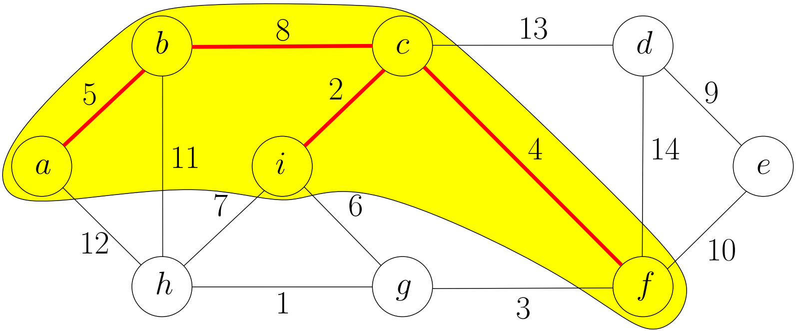
Prim's Algorithm: Example



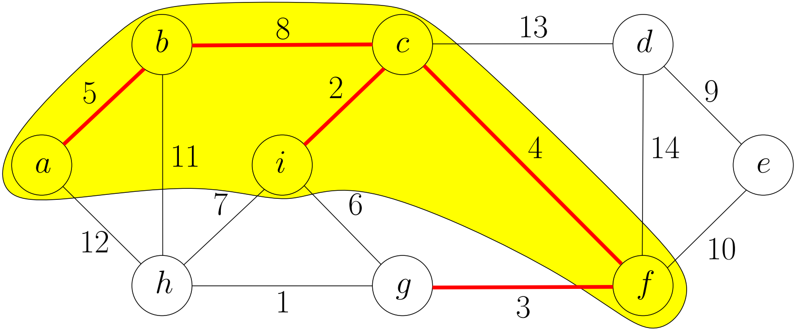
Prim's Algorithm: Example



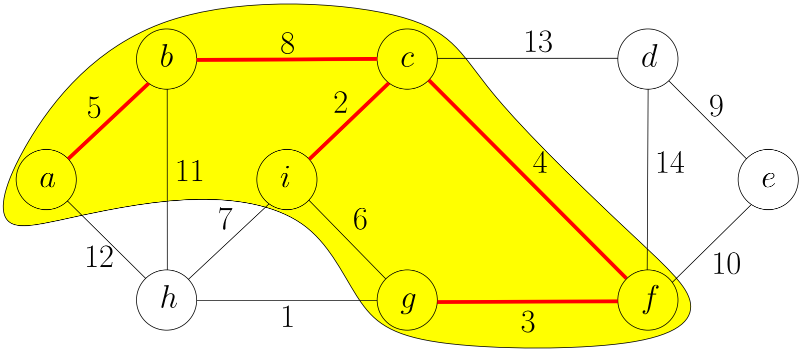
Prim's Algorithm: Example



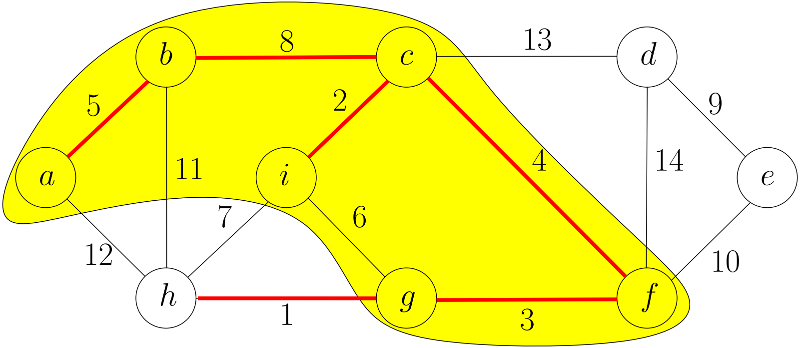
Prim's Algorithm: Example



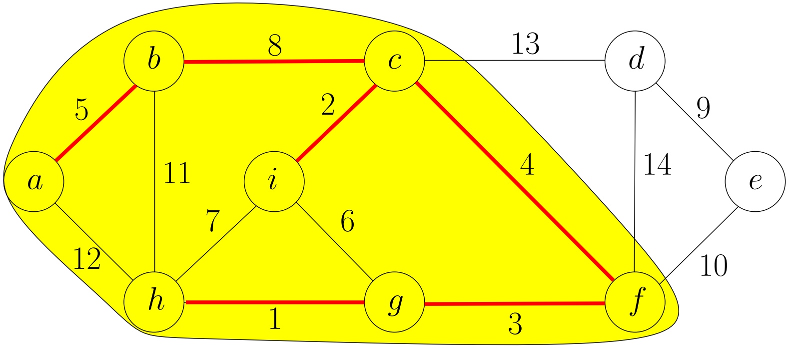
Prim's Algorithm: Example



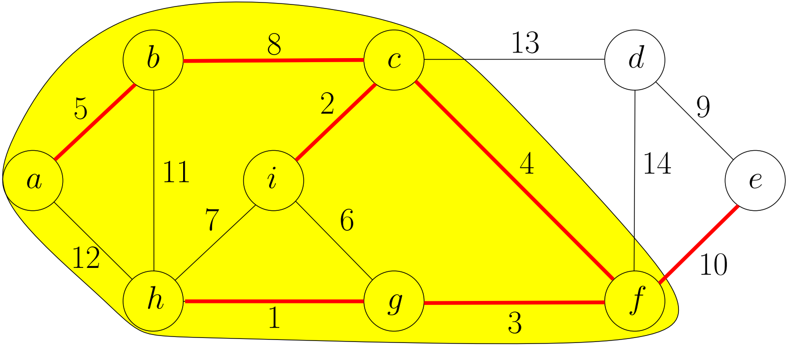
Prim's Algorithm: Example



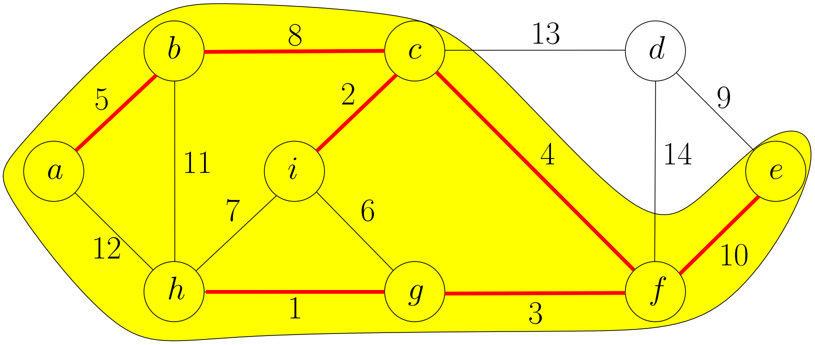
Prim's Algorithm: Example



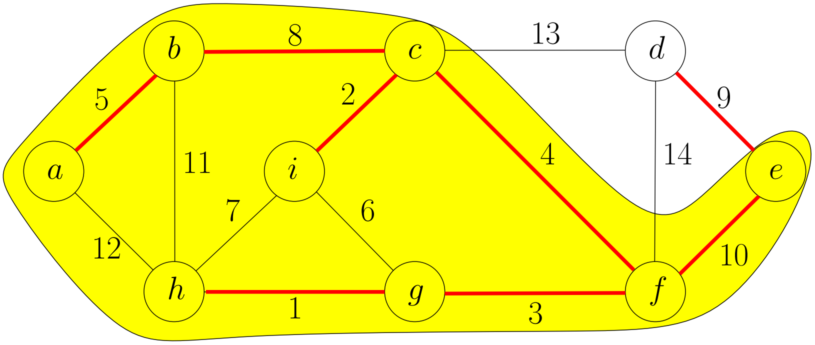
Prim's Algorithm: Example



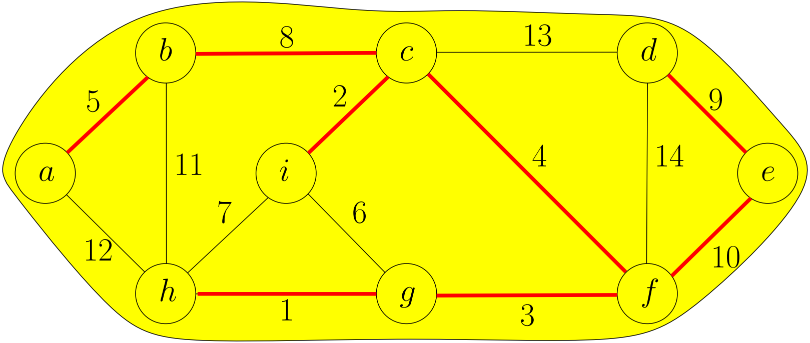
Prim's Algorithm: Example



Prim's Algorithm: Example



Prim's Algorithm: Example



Prim's Algorithm

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d[v] \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ **do**
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d[u]$
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $w(u, v) < d[v]$ **then**
            $d[v] \leftarrow w(u, v)$
            $\pi[v] \leftarrow u$
return $\big\{(u, \pi[u]) | u \in V \setminus \{s\}\big\}$

Running Time of Prim's Algorithm Using Priority Queue

$O(n)\times$ (time for extract_min) $+\ O(m)\times$ (time for decrease_key)

concrete DS	extract_min	decrease_key	overall time
heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

"Evidence" for $e \in$ MST or $e \notin$ MST

Assumption Assume all edge weights are different.

- $e \in$ MST $\leftrightarrow$ there is a cut in which e is the lightest edge
- $e \notin$ MST $\leftrightarrow$ there is a cycle in which e is the heaviest edge

Exactly one of the following is true:

- There is a cut in which e is the lightest edge
- There is a cycle in which e is the heaviest edge

Outline

6. Graph Algorithms
   - Minimum Spanning Tree, Kruskal's Algorithm, Prim's Algorithm
   - Shortest Path Algorithms
   - Minimum Cost Arborescence

algorithm	graph	weights	SS?	running time
Simple DP	DAG	$\mathbb{R}$	SS	$O(n+m)$
Dijkstra	U/D	$\mathbb{R}_{\geq 0}$	SS	$O(n \log n + m)$
Bellman-Ford	U/D	$\mathbb{R}$	SS	$O(nm)$
Floyd-Warshall	U/D	$\mathbb{R}$	AP	$O(n^3)$

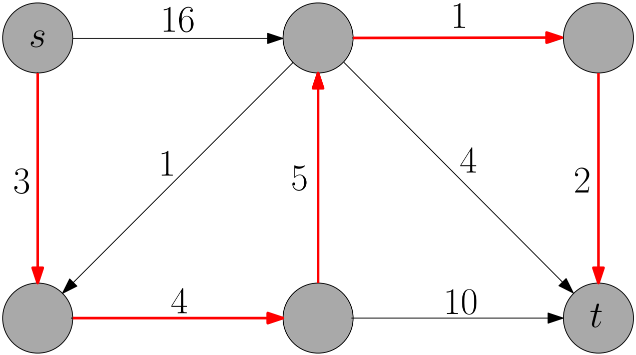
- DAG = directed acyclic graph   U = undirected   D = directed
- SS = single source   AP = all pairs

s-t Shortest Paths

Input: directed graph $G = (V, E)$, $s, t \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest path from s to t



Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: $\pi[v], v \in V \setminus s$: the parent of v in shortest path tree

$d[v], v \in V \setminus s$: the length of shortest path from s to v

Improved Running Time using Priority Queue

Dijkstra(G, w, s)

```
s ← arbitrary vertex in G
S ← ∅, d(s) ← 0 and d[v] ← ∞ for every v ∈ V \ {s}
Q ← empty queue, for each v ∈ V: Q.insert(v, d[v])
while S ≠ V do
    u ← Q.extract_min()
    S ← S ∪ {u}
    for each v ∈ V \ S such that (u, v) ∈ E do
        if d[u] + w(u, v) < d[v] then
            d[v] ← d[u] + w(u, v), Q.decrease_key(v, d[v])
            π[v] ← u
return (π, d)
```

Running time:
$O(n) \times$ (time for extract_min) $+ O(m) \times$ (time for decrease_key)

Priority-Queue	extract_min	decrease_key	Time
Heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci Heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

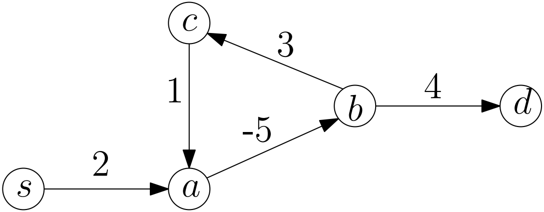
Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

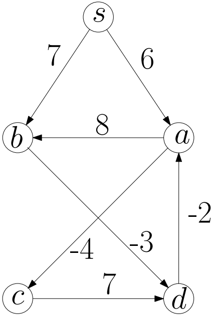
assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$



- Unfortunately, computing the shortest simple path between two vertices is an NP-hard problem.
- Cases: no negative cycles, need to detect negative cycles



- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \begin{Bmatrix} f^{\ell-1}[v] \\ \min_{u:(u,v)\in E} \left(f^{\ell-1}[u] + w(u,v)\right) \end{Bmatrix} & \ell > 0 \end{cases}$$

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    updated ← false
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
            updated ← true
    if updated = false then break
return f
```

All-Pair Shortest Paths

All Pair Shortest Paths

Input: directed graph $G = (V, E)$,
$w : E \to \mathbb{R}$ (can be negative)

Output: shortest path from u to v for every $u, v \in V$

1: **for** every starting point $s \in V$ **do**
2: run Bellman-Ford(G, w, s)

- Running time $= O(n^2 m)$

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

Floyd-Warshall(G, w)

```
f ← w
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if f[i,k] + f[k,j] < f[i,j] then
                f[i,j] ← f[i,k] + f[k,j]
```

Lemma Assume there are no negative cycles in G. After iteration k, for $i, j \in V$, $f[i,j]$ is exactly the length of shortest path from i to j that only uses vertices in $\{1, 2, 3, \cdots, k\}$ as intermediate vertices.

- Running time $= O(n^3)$.

Outline

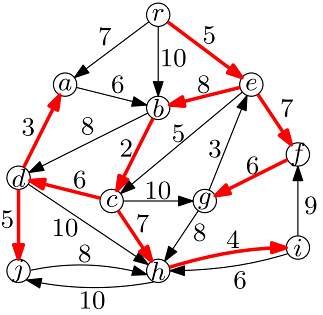
6 Graph Algorithms
- Minimum Spanning Tree, Kruskal's Algorithm, Prim's Algorithm
- Shortest Path Algorithms
- Minimum Cost Arborescence

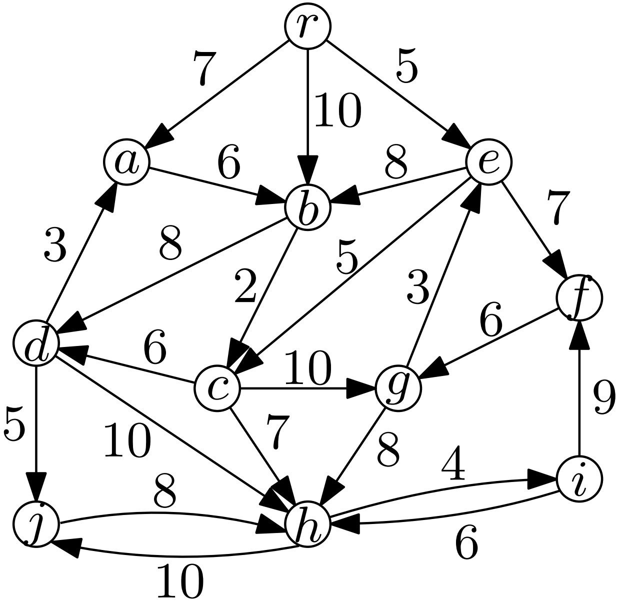
Def. An arborescence is directed rooted tree, where all edges are directed away from the root.

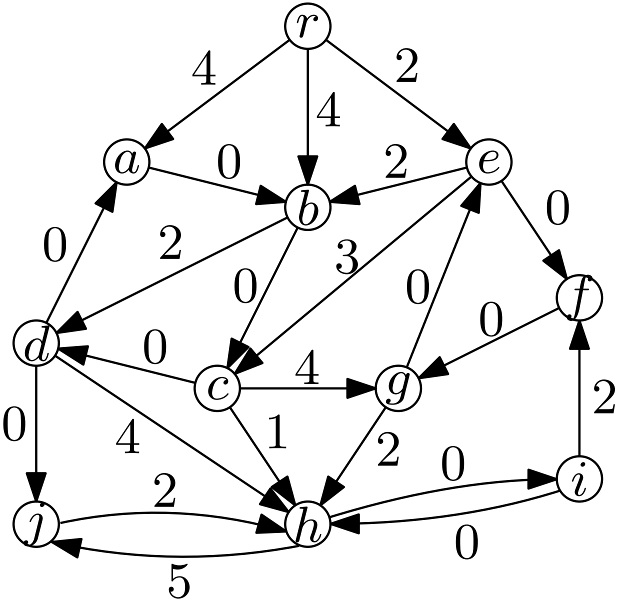
Minimum Cost Arborescence Problem

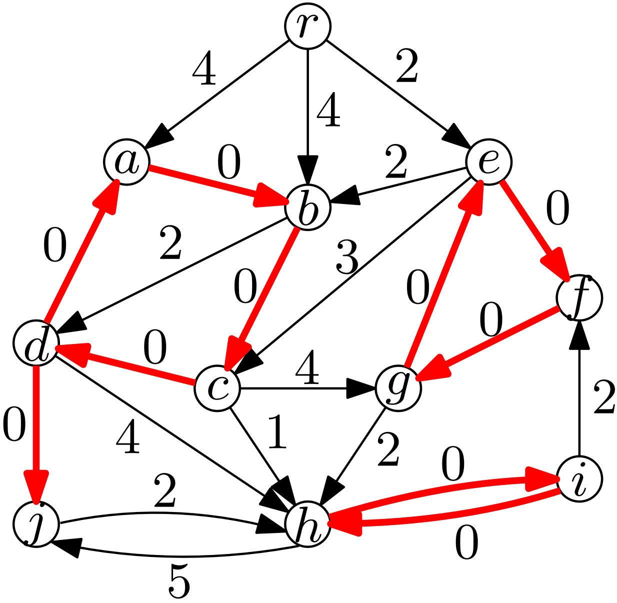
Input: a directed graph $G = (V, E)$,
edge weights $w : \mathbb{E} \to \mathbb{R}_{\geq 0}$
root $r \in V$

Output: a minimum-cost sub-graph $T = (V, E')$ of G that is an arborescence with root r

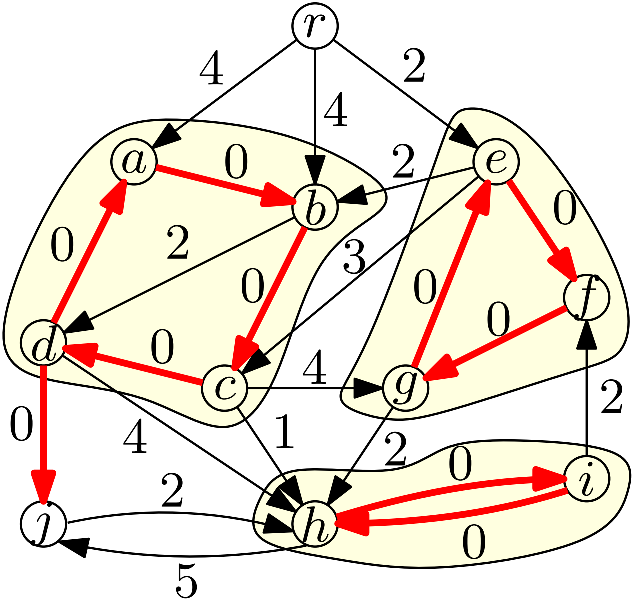



r
7
10
5
a
6
b
8
e
3
8
7
5
2
3
f
6
d
6
c
10
g
9
5
10
7
8
4
i
8
j
h
6
10

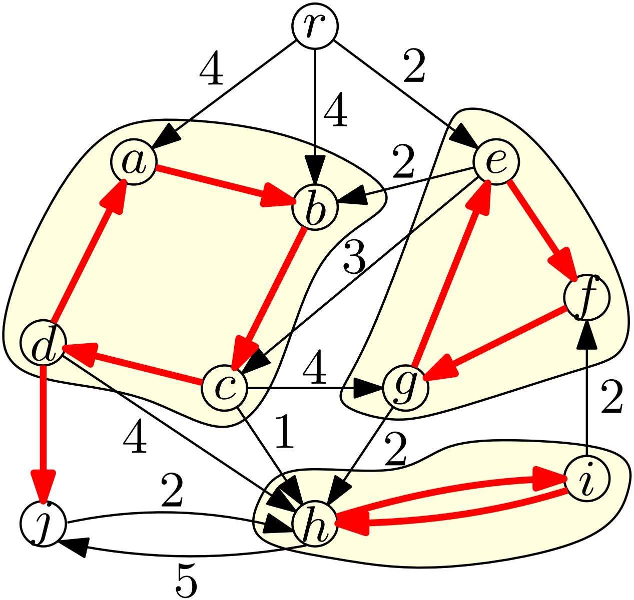

r
a
b
e
f
d
c
g
j
h
i
4
4
2
0
2
0
0
2
3
0
0
0
0
4
0
4
1
2
2
0
0
2
5



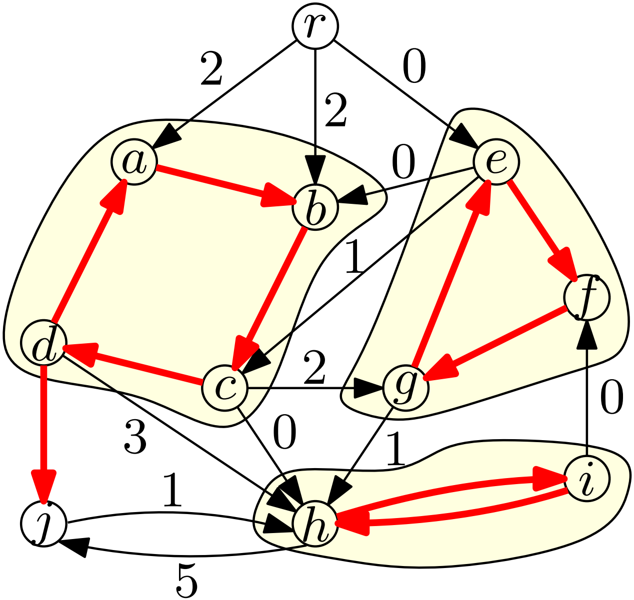
r
a
b
e
d
c
g
f
j
h
i
4
4
2
0
2
0
2
0
3
0
0
0
0
4
0
4
1
2
0
2
2
5
0



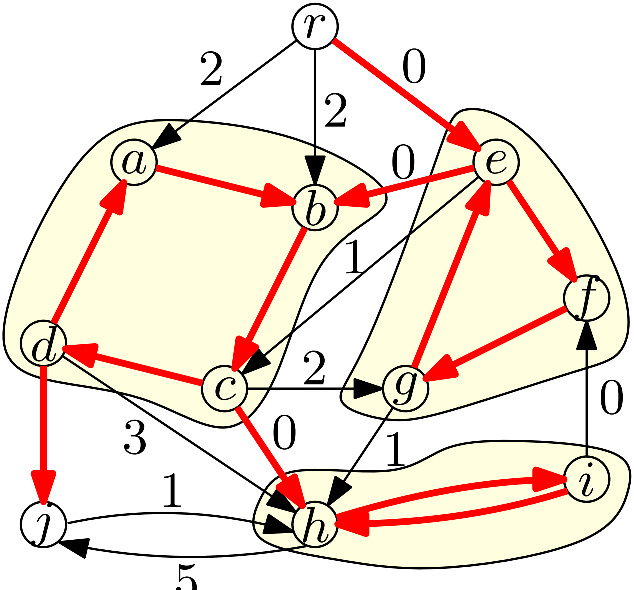
r
a
b
e
d
c
f
g
j
h
i
4
4
2
0
2
2
0
3
0
0
0
0
0
4
0
4
1
2
2
0
2
0
5



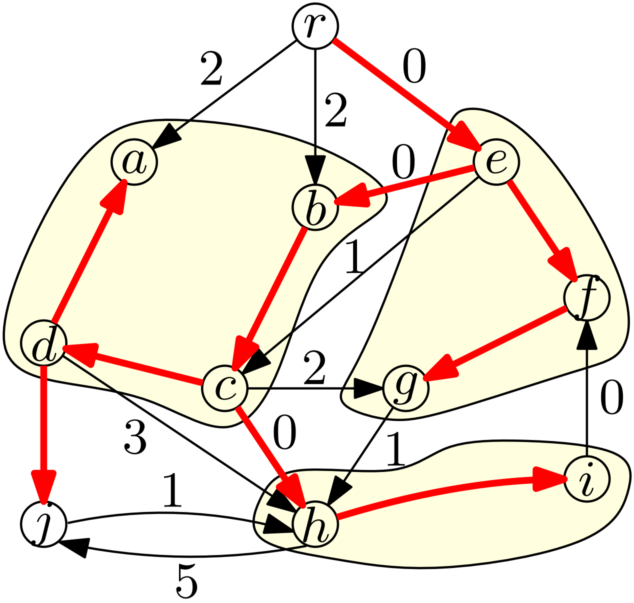
r
4
4
2
a
b
e
2
3
f
d
c
4
g
2
4
1
2
i
2
j
h
5

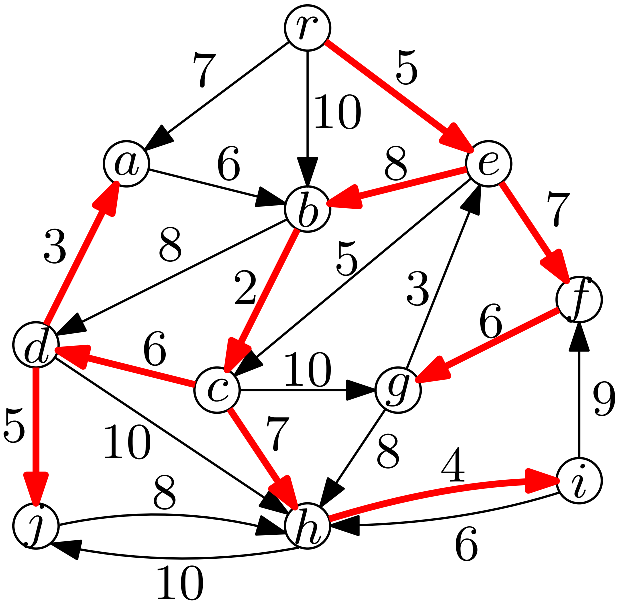


r
2
2
0
a
b
e
0
1
f
d
c
2
g
0
3
0
1
1
i
j
h
5



r
2
2
0
a
b
0
e
1
f
d
c
2
g
0
3
0
1
1
i
j
h
5


r
a
b
e
f
d
c
g
j
h
i
2
2
0
0
1
2
3
0
1
1
5
0

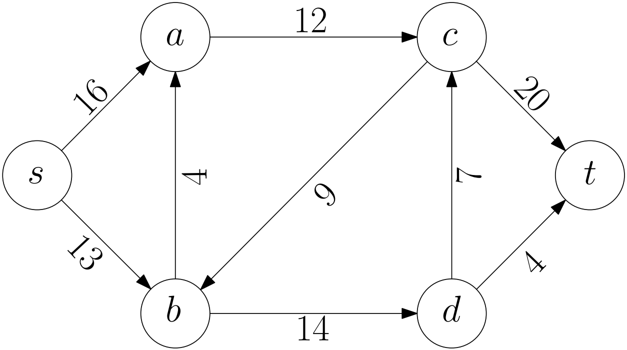

r
7
10
5
a
6
b
8
e
3
8
5
7
2
3
f
6
d
6
c
10
g
5
10
7
8
4
9
i
8
j
h
6
10

Outline

7. Network Flow
   - Ford-Fulkerson Method
   - Running Time of Ford-Fulkerson-Type Algorithm
   - Bipartite Matching Problem
   - s-t Edge-Disjoint Paths Problem
   - More Applications

Flow Network

- Abstraction of fluid flowing through edges
- Digraph $G = (V, E)$ with source $s \in V$ and sink $t \in V$
  - No edges enter s
  - No edges leave t
- Edge capacity $c_e \in \mathbb{R}_{>0}$ for every $e \in E$



Def. An s-t flow is a function $f : E \to \mathbb{R}$ such that

- for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

The value of a flow f is

$$\text{val}(f) := \sum_{e \in \delta_{\text{out}}(s)} f(e).$$

Maximum Flow Problem

Input: directed network $G = (V, E)$, capacity function $c : E \to \mathbb{R}_{>0}$, source $s \in V$ and sink $t \in V$

Output: an s-t flow f in G with the maximum $\text{val}(f)$

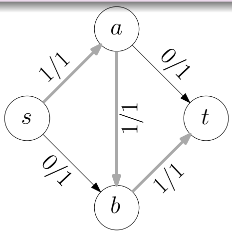
Outline

7. Network Flow
   - Ford-Fulkerson Method
   - Running Time of Ford-Fulkerson-Type Algorithm
   - Bipartite Matching Problem
   - s-t Edge-Disjoint Paths Problem
   - More Applications

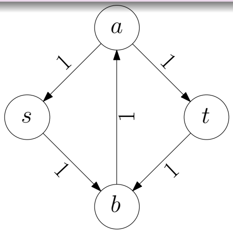
Assumption (u, v) and (v, u) are not both in E

Def. For a s-t flow f, the residual graph G_f of $G = (V, E)$ w.r.t f contains:

- the vertex set V,
- for every $e = (u, v) \in E$ with $f(e) < c_e$, a forward edge $e = (u, v)$, with residual capacity $c_f(e) = c_e - f(e)$,
- for every $e = (u, v) \in E$ with $f(e) > 0$, a backward edge $e' = (v, u)$, with residual capacity $c_f(e') = f(e)$.



Original graph G and f



Residual Graph G_f

Agumenting Path

Augmenting the flow along a path P from s to t in G_f

Augment(P)

1: $b \leftarrow \min_{e \in P} c_f(e)$
2: **for** every $(u, v) \in P$ **do**
3: **if** (u, v) is a forward edge **then**
4: $f(u, v) \leftarrow f(u, v) + b$
5: **else** ▷ (u, v) is a backward edge
6: $f(v, u) \leftarrow f(v, u) - b$
7: **return** f

Ford-Fulkerson's Method

Ford-Fulkerson(G, s, t, c)

let $f(e) \leftarrow 0$ for every e in G
while there is a path from s to t in G_f **do**
    let P be any simple path from s to t in G_f
    $f \leftarrow$augment(f, P)
return f

Correctness of Ford-Fulkerson's Method

1. The procedure augment(f, P) maintains the two conditions:
   - for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
   - for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

2. When Ford-Fulkerson's Method terminates, val(f) is maximized
3. Ford-Fulkerson's Method will terminate

Coro. $$\text{val}(f) \le \min_{s\text{-}t \text{ cut } (S,T)} c(S,T) \text{ for every } s\text{-}t \text{ flow} f.$$

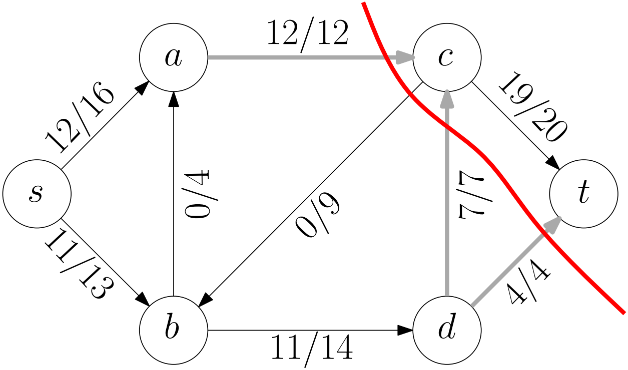
Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies
$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Corollary and Main Lemma implies

Maximum Flow Minimum Cut Theorem
$$\sup_{s\text{-}t \text{ flow } f} \text{val}(f) = \min_{s\text{-}t \text{ cut } (S,T)} c(S,T).$$

Maximum Flow Minimum Cut Theorem

$$\sup_{s\text{-}t \text{ flow } f} \mathsf{val}(f) = \min_{s\text{-}t \text{ cut } (S,T)} c(S,T).$$



Outline

7 Network Flow

- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- s-t Edge-Disjoint Paths Problem
- More Applications

Shortest Augmenting Path

shortest-augmenting-path(G, s, t, c)

let $f(e) \leftarrow 0$ for every e in G
while there is a path from s to t in G_f **do**
    $P \leftarrow$ breadth-first-search(G_f, s, t)
    $f \leftarrow$augment(f, P)
return f

Due to [Dinitz 1970] and [Edmonds-Karp, 1970]

Capacity-Scaling Algorithm

- Idea: find the augment path from s to t with a sufficiently large bottleneck capacity
- Assumption: Capacities are integers between 1 and C

capacity-scaling(G, s, t, c)

let $f(e) \leftarrow 0$ for every e in G
$\Delta \leftarrow$ largest power of 2 which is at most C
while $\Delta \geq 1$ do **do**
    while there exists an augmenting path P with bottleneck capacity at least Δ **do**
        $f \leftarrow$augment(f, P)
    $\Delta \leftarrow \Delta/2$
return f

Outline

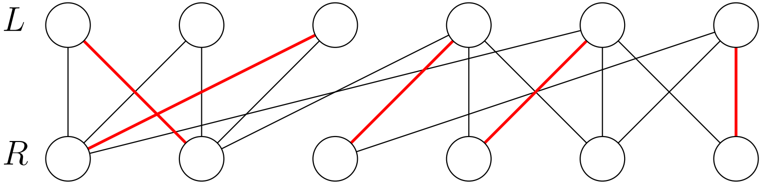
7 Network Flow
- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- s-t Edge-Disjoint Paths Problem
- More Applications

Def. Given a bipartite graph $G = (L \cup R, E)$, a matching in G is a set $M \subseteq E$ of edges such that every vertex in V is an endpoint of at most one edge in M.

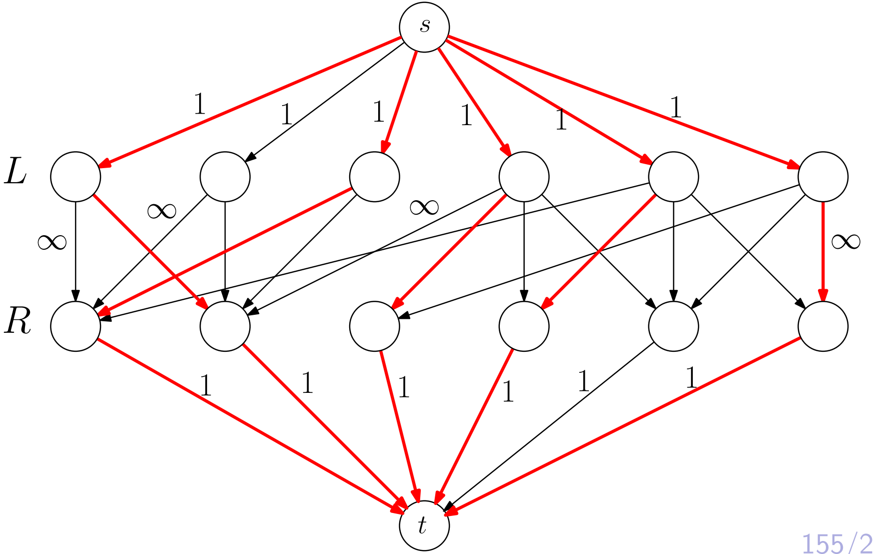
Maximum Bipartite Matching Problem

Input: bipartite graph $G = (L \cup R, E)$

Output: a matching M in G of the maximum size



Reduce Maximum Bipartite Matching to Maximum Flow Problem



Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

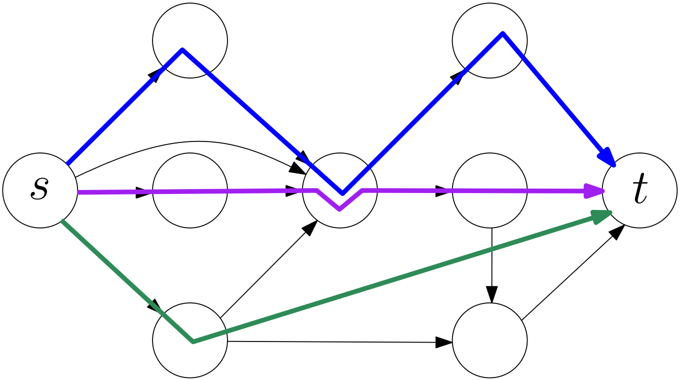
Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Outline

s-t Edge Disjoint Paths

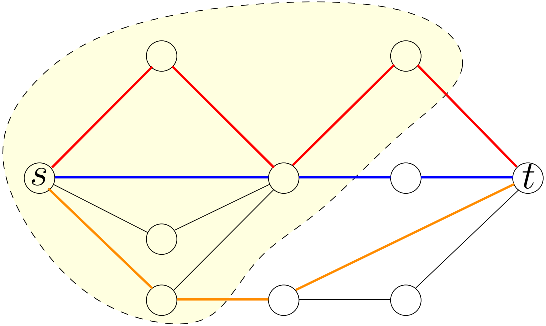
Input: a directed (or undirected) graph $G = (V, E)$ and $s, t \in V$

Output: the maximum number of edge-disjoint paths from s to t in G



Menger's Theorem

Menger's Theorem In an undirected graph, the maximum number of edge-disjoint paths between s to t is equal to the minimum number of edges whose removal disconnects s and t.

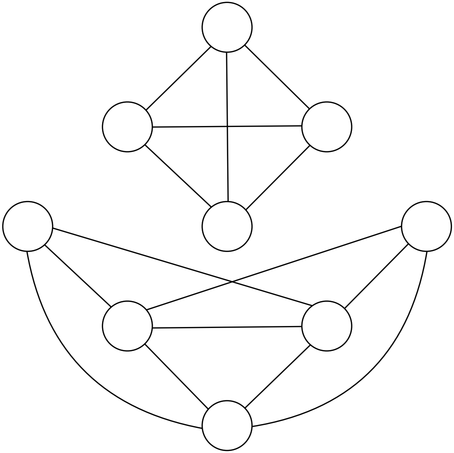


s-t connectivity measures how well s and t are connected.

Global Min-Cut Problem

Input: a connected graph $G = (V, E)$

Output: the minimum number of edges whose removal will disconnect G



Outline

Extension of Network Flow: Circulation Problem

Input: A digraph $G = (V, E)$

capacities $c \in \mathbb{Z}_{\geq 0}^{E}$

supply vector $d \in \mathbb{Z}^{V}$ with $\sum_{v \in V} d_v = 0$

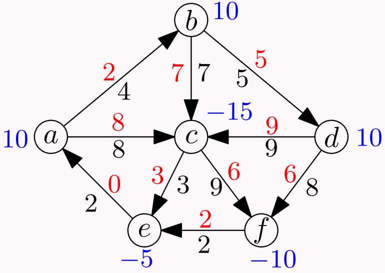
Output: whether there exists $f : E \to \mathbb{Z}_{\geq 0}$ s.t.

$$\sum_{e \in \delta^{\text{out}}(v)} f(e) - \sum_{e \in \delta^{\text{in}}(v)} f(e) = d_v \qquad \forall v \in V$$

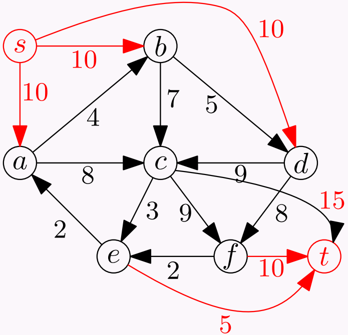
$$0 \leq f(e) \leq c_e \qquad \forall e \in E$$

- d_v denotes the net supply of a good
- $d_v > 0$: there is a supply of d_v at v
- $d_v < 0$: there is a demand of $-d_v$ at v
- problem: whether we can match the supplies and demands without violating capacity constraints

Example



Reduction



Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

Circulation Problem with Capacity Lower Bounds

Input: A digraph $G = (V, E)$

capacities $c \in \mathbb{Z}_{\geq 0}^{E}$

capacity lower bounds $l \in \mathbb{Z}_{\geq 0}^{E}$, $0 \leq l_e \leq c_e$

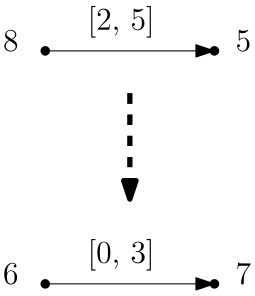
supply vector $d \in \mathbb{Z}^{V}$ with $\sum_{v \in V} d_v = 0$

Output: whether there exists $f : E \to \mathbb{Z}_{\geq 0}$ s.t.

$$\sum_{e \in \delta^{\mathrm{out}}(v)} f(e) - \sum_{e \in \delta^{\mathrm{in}}(v)} f(e) = d_v \qquad \forall v \in V$$

$$l_e \leq f(e) \leq c_e \qquad \forall e \in E$$

Removing Capacity Lower Bounds



handling $e = (u, v)$ with $l_e > 0$

- $d'_u \leftarrow d_u - l_e$
- $d'_v \leftarrow d_v + l_e$
- $c'_e \leftarrow c_e - l_e$
- $l'_e \leftarrow 0$

- in old instance: flow is $f(e) \in [l_e, c_e] \implies f(e) - l_e \in [0, c_e - l_e]$
- in new instance: flow is $f(e) - l_e \in [0, c'_e]$

Survey Design

Input: integers $n, k \geq 1$ and $E \subseteq [n] \times [k]$

integers $0 \leq c_i \leq c'_i, \forall i \in [n]$

integers $0 \leq p_j \leq p'_j, \forall j \in [k]$

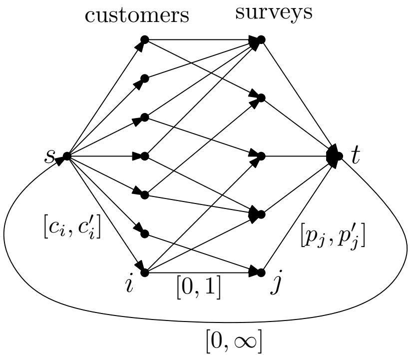
Output: $E' \subseteq E$ s.t.

$$c_i \leq |\{j \in [k] : (i,j) \in E'\}| \leq c'_i, \qquad \forall i \in [n]$$

$$p_j \leq |\{i \in [m] : (i,j) \in E'\}| \leq p'_j, \qquad \forall j \in [k]$$

Reduction to Circulation

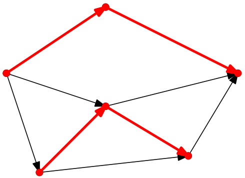
- vertices $\{s,t\} \uplus [n] \uplus [k]$,
- $(i,j) \in E$: (i,j) with bounds $[0,1]$
- $\forall i$: (s,i) with bounds $[c_i, c'_i]$
- $\forall j$: (j,t) with bounds $[p_j, p'_i]$
- (t,s) with bounds $[0,\infty]$



Airline Scheduling

Input: a DAG $G = (V, E)$

Output: the minimum number of disjoint paths in G to cover all vertices



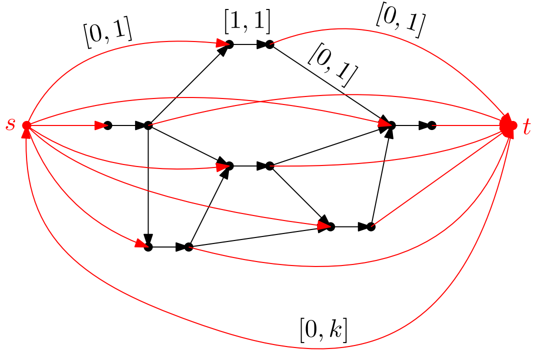
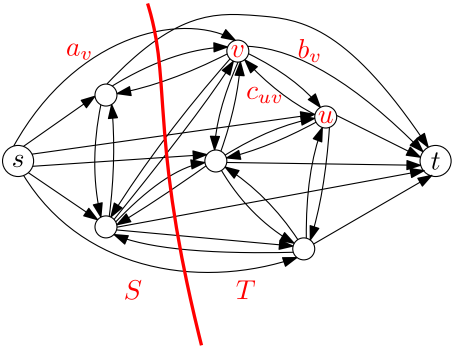


Image Segmentation

Input: A graph $G = (V, E)$, with edge costs $c \in \mathbb{Z}_{\geq 0}^{E}$
two reward vectors $a, b \in \mathbb{Z}_{\geq 0}^{V}$

Output: a cut (A, B) of G so as to maximize

$$\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\} \cap A| = 1} c_{(u,v)}$$



- The cut value of $(S = \{s\} \cup A, \{t\} \cup B)$ is

$$
\begin{aligned}
&\sum_{v \in V}(a_v + b_v) - \Big(\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\} \cap A| = 1} c_{(u,v)} \Big) \\
&= \sum_{v \in V}(a_v + b_v) - \big(\text{objective of } (A, B)\big)
\end{aligned}
$$

- So, maximizing the objective of (A, B) is equivalent to minimizing the cut value.

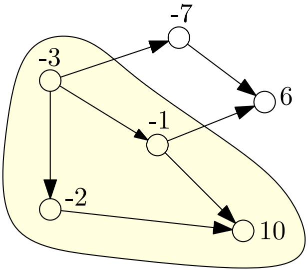
Project Selection

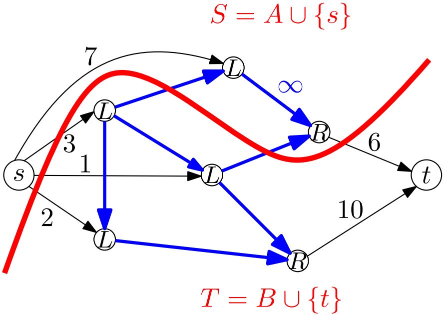
Input: A DAG $G = (V, E)$

revenue on vertices: $p \in \mathbb{Z}^V$; p_v's could be negative.

Output: A set $B \subseteq V$ satisfying the precedence constraints:

$$v \in B \implies u \in B, \quad \forall (u, v) \in E$$





- min-cut $(S = \{s\} \cup A, T = \{t\} \cup B)$
- no ∞-capacity edges from A to B
- cut value is

$$\sum_{v \in B \cap L} (-p_v) + \sum_{v \in A \cap R} p_v = -\sum_{v \in B \cap L} p_v - \sum_{v \in B \cap R} p_v + \sum_{v \in R} p_v$$
$$= \sum_{v \in R} p_v - \sum_{v \in B} p_v$$

More Applications

- Graph orientation
- maximum independent set (and minimum vertex cover) in a bipartite graph
- $\cdots$

Outline

8. Linear Programming
   - Linear Programming Duality
   - Integral Polytopes: Exact Algorithms Using LP

Standard Form of Linear Programming

Let $x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$, $\quad c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$,

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

Then, LP becomes $\min \quad c^{\mathrm{T}}x$ s.t.

$$Ax \geq b$$
$$x \geq 0$$

Standard Form of Linear Programming

$$\min \quad c^{\mathrm{T}} x \quad \text{s.t.}$$
$$Ax \geq b$$
$$x \geq 0$$

- Linear programmings can be solved in polynomial time

Algorithm	Theory	Practice
Simplex Method	Exponential Time	Works Well
Ellipsoid Method	Polynomial Time	Slow
Internal Point Methods	Polynomial Time	Works Well

Outline

Primal LP

$$\min \quad c^T x \qquad \text{s.t.}$$
$$Ax \geq b$$
$$x \geq 0$$

Dual LP

$$\max \quad b^T y \qquad \text{s.t.}$$
$$A^T y \leq c$$
$$y \geq 0$$

- $P =$ value of primal LP
- $D =$ value of dual LP

Theorem (weak duality theorem) $D \leq P$.

Theorem (strong duality theorem) $D = P$.

- Can always prove the optimality of the primal solution, by adding up primal constraints.

Outline

8. Linear Programming
   - Linear Programming Duality
   - Integral Polytopes: Exact Algorithms Using LP

Def. A polytope $P \subseteq \mathbb{R}^n$ is said to be integral, if all vertices of P are in $\mathbb{Z}^n$.

Maximum Weight Bipartite Matching

Input: bipartite graph $G = (L \uplus R, E)$

edge weights $w \in \mathbb{Z}^E_{>0}$

Output: a matching $M \subseteq E$ so as to maximize $\sum_{e \in M} w_e$

LP Relaxation

$$\max \quad \sum_{e \in E} w_e x_e$$

$$\sum_{e \in \delta(v)} x_e \leq 1 \quad \forall v \in L \cup R$$

$$x_e \geq 0 \quad \forall e \in E$$

Theorem The LP polytope is integral: It is the convex hull of $\{\chi^M : M \text{ is a matching}\}$.

LP for Maximum Flow

$$\begin{aligned} \max \quad & \sum_{e \in \delta_{\mathsf{in}}(t)} x_e \\ x_e &\le c_e & \forall e \in E \\ \sum_{e \in \delta_{\mathsf{out}}(v)} x_e - \sum_{e \in \delta_{\mathsf{in}}(v)} x_e &= 0 & \forall v \in V \setminus \{s, t\} \\ x_e &\ge 0 & \forall e \in E \end{aligned}$$

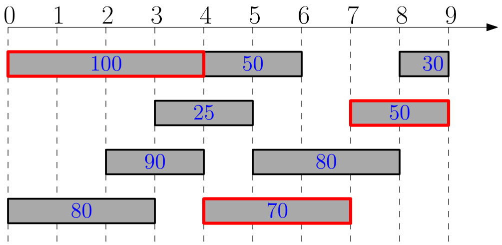
Theorem The LP polytope is integral.

Weighted Interval Scheduling Problem

Input: n activities, activity i starts at time s_i, finishes at time f_i, and has weight $w_i > 0$

i and j can be scheduled together iff $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: maximum weight subset of jobs that can be scheduled



- optimum value$= 220$

Weighted Interval Scheduling Problem

Linear Program

$$\max \quad \sum_{j\in[n]} x_j w_j$$

$$\sum_{j\in[n]:t\in[s_j,f_j)} x_j \leq 1 \qquad \forall t \in [T]$$

$$x_j \geq 0 \qquad \forall j \in [n]$$

Theorem The LP polytope is integral.

Def. A matrix $A \in \mathbb{R}^{m\times n}$ is said to be tototally unimodular (TUM), if every sub-square of A has determinant in $\{-1, 0, 1\}$.

Theorem If a polytope P is defined by $Ax \geq b, x \geq 0$ with a totally unimodular matrix A and integral b, then P is integral.

Lemma A matrix $A \in \{0, 1\}^{m\times n}$ where the 1's on every column form an interval is TUM.

Lemma Let $A' \in \{0, \pm 1\}^{n \times n}$ such that every row of A' contains at most one 1 and one -1. Then $\det(A') \in \{0, \pm 1\}$.

Lemma Let $A \in \{0, \pm 1\}^{m \times n}$ such that every row of A contains at most one 1 and one -1. Then A is TUM.

Coro. The matrix for s-t flow polytope is TUM; thus, the polytope is integral.

Lemma The edge-vertex incidence matrix A of a bipartite graph is totally-unimodular.

Outline

9. NP-Completeness Theory
   - P, NP and Co-NP
   - Polynomial Time Reductions and NP-Completeness
   - NP-Complete Problems
   - Dealing with NP-Hard Problems

Outline

Complexity Class P

Def. The complexity class P is the set of decision problems X that can be solved in polynomial time.

- The decision versions of interval scheduling, shortest path and minimum spanning tree all in P.

The Complexity Class NP

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t, and outputs 0 or 1.
- there is a polynomial function p such that, $X(s) = 1$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string t such that $B(s,t) = 1$ is called a certificate.

Def. The complexity class NP is the set of all problems for which there exists an efficient certifier.

The Complexity Class Co-NP

Def. For a problem X, the problem $\overline{X}$ is the problem such that $\overline{X}(s) = 1$ if and only if $X(s) = 0$.

Def. Co-NP is the set of decision problems X such that $\overline{X} \in$ NP.

P $\subseteq$ NP

- Let $X \in$ P and $X(s) = 1$
- The certificate is an empty string
- Thus, $X \in$ NP and P $\subseteq$ NP
- Similarly, P $\subseteq$ Co-NP, thus P $\subseteq$ NP $\cap$ Co-NP
- P $=$ NP? A famous, big, and fundamental open problem in computer science

Outline

9. NP-Completeness Theory
   - P, NP and Co-NP
   - Polynomial Time Reductions and NP-Completeness
   - NP-Complete Problems
   - Dealing with NP-Hard Problems

Polynomial-Time Reducations

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

Def. A problem X is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_\text{P} X$ for every $Y \in \text{NP}$.

Theorem If X is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

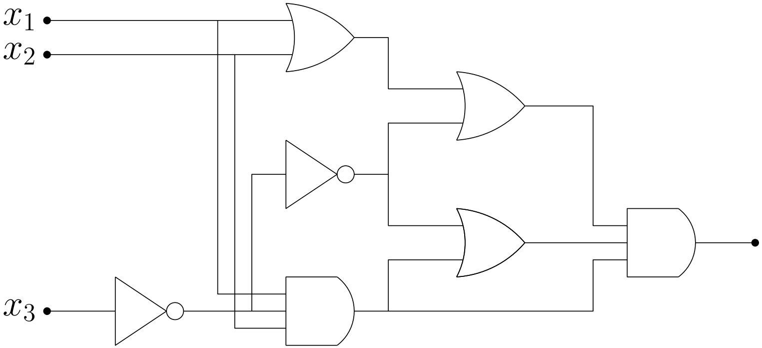
Outline

The First NP-Complete Problem: Circuit-Sat

Circuit Satisfiability (Circuit-Sat)

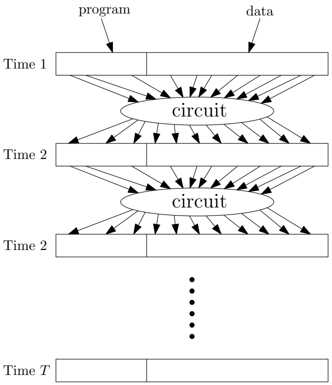
Input: a circuit

Output: whether the circuit is satisfiable



Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits



- Any problem $Y \in$ NP can be reduced to Circuit-Sat.

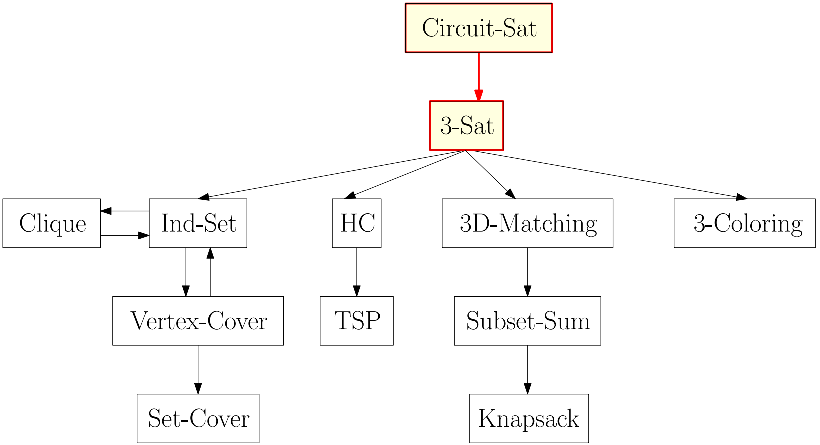
$Y \leq_P$ Circuit-Sat, For Every $Y \in$NP

- check-Y(s,t) returns 1 if t is a valid certificate for s.
- s is a yes-instance if and only if there is a t such that check-Y(s,t) returns 1

- Construct a circuit C' for the algorithm check-Y
- hard-wire the instance s to the circuit C' to obtain the circuit C
- s is a yes-instance if and only if C is satisfiable $\square$

Theorem Circuit-Sat is NP-complete.

Reductions of NP-Complete Problems



3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

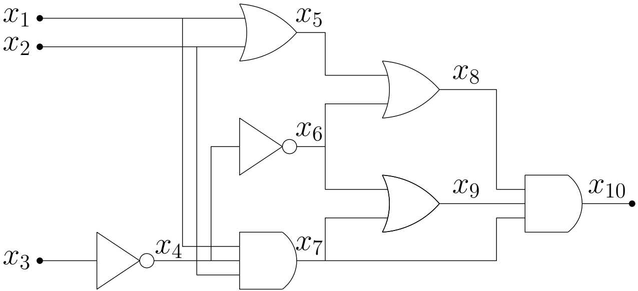
- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: x_i or $\neg x_i$
- Clause: disjunction ("or") of at most 3 literals: $x_3 \vee \neg x_4$, $x_1 \vee x_8 \vee \neg x_9$, $\neg x_2 \vee \neg x_5 \vee x_7$
- 3-CNF formula: conjunction ("and") of clauses: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

3-Sat

Input: a 3-CNF formula

Output: whether the 3-CNF is satisfiable

Circuit-Sat $\leq_P$ 3-Sat



- Associate every wire with a new variable
- The circuit is equivalent to the following formula:

$$
\begin{aligned}
&(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4) \\
&\wedge (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6) \\
&\wedge (x_9 = x_6 \vee x_7) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}
\end{aligned}
$$

Circuit-Sat $\leq_P$ 3-Sat

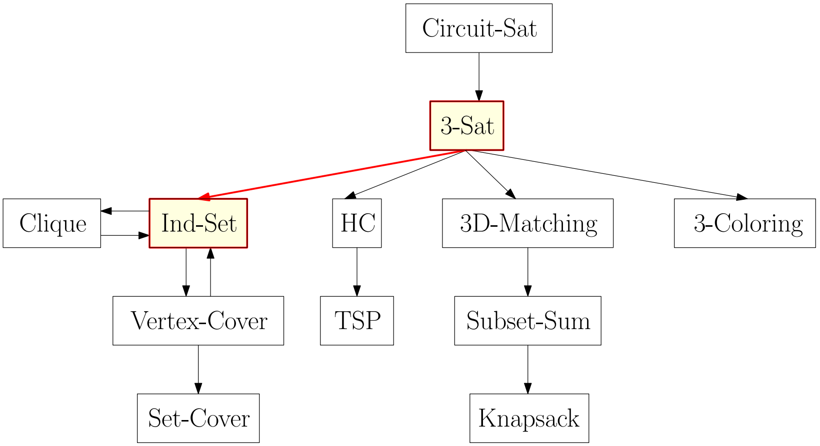
$$
\begin{aligned}
&(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4) \\
&\wedge (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6) \\
&\wedge (x_9 = x_6 \vee x_7) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}
\end{aligned}
$$

Convert each clause to a 3-CNF

$$
\begin{aligned}
&x_5 = x_1 \vee x_2 \quad \Leftrightarrow \\
&(x_1 \vee x_2 \vee \neg x_5) \quad \wedge \\
&(x_1 \vee \neg x_2 \vee x_5) \quad \wedge \\
&(\neg x_1 \vee x_2 \vee x_5) \quad \wedge \\
&(\neg x_1 \vee \neg x_2 \vee x_5)
\end{aligned}
$$

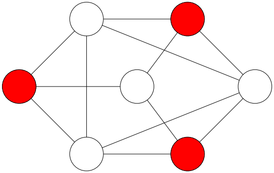
- Circuit $\Longleftrightarrow$ Formula $\Longleftrightarrow$ 3-CNF

Reductions of NP-Complete Problems



Recall: Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Independent Set (Ind-Set) Problem

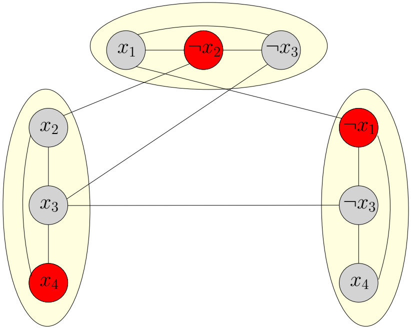
Input: $G = (V, E), k$

Output: whether there is an independent set of size k in G

3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

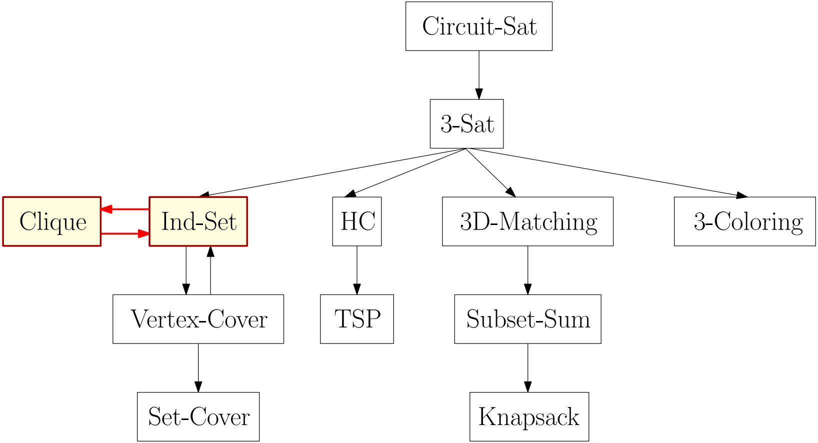
- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#$clauses



3-Sat instance is yes-instance $\Leftrightarrow$ Ind-Set instance is yes-instance:

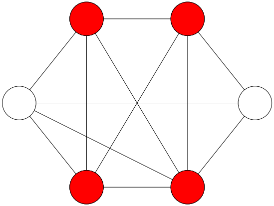
- satisfying assignment $\Longleftrightarrow$ independent set of size k

Reductions of NP-Complete Problems



Clique $=_P$ Ind-Set

Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$



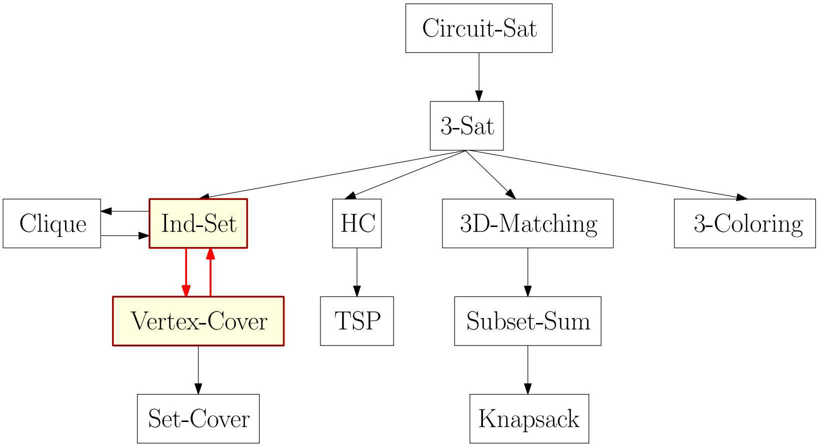
Clique Problem

Input: $G = (V, E)$ and integer $k > 0$,

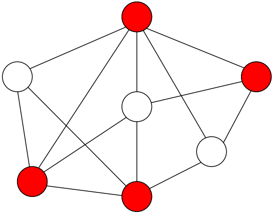
Output: whether there exists a clique of size k in G

Obs. S is an independent set in G if and only if S is a clique in $\overline{G}$.

Reductions of NP-Complete Problems



Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



Vertex-Cover Problem

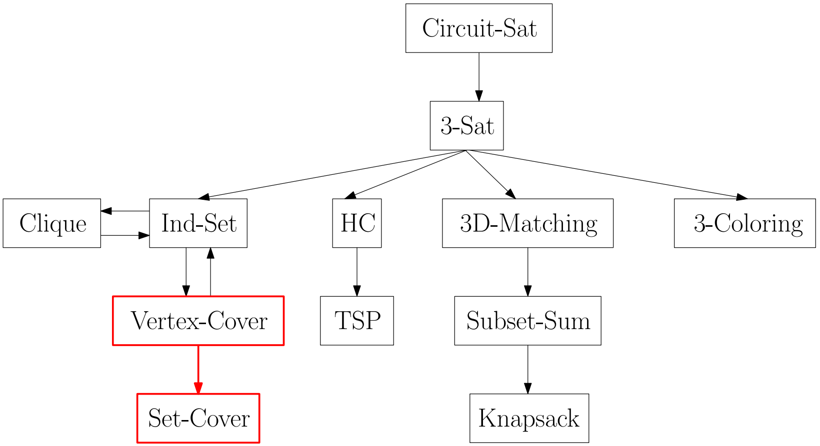
Input: $G = (V, E)$ and integer k

Output: whether there is a vertex cover of G of size at most k

Vertex-Cover $=_P$ Ind-Set

S is a vertex-cover of $G = (V, E)$ if and only if $V \setminus S$ is an independent set of G.

Reductions of NP-Complete Problems



Set Cover

Input: $S_1, S_2, \cdots, S_M \subseteq [N]$ with $\bigcup_{i \in [m]} S_i = [N]$

Output: The smallest set $I \subseteq [M]$ satisfying $\bigcup_{i \in I} S_i = [N]$

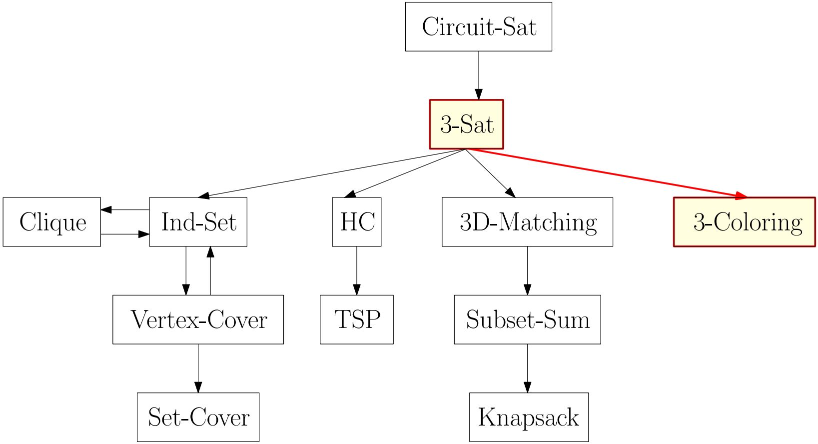
- decision version: given t, does there exist a solution I with $|I| \leq t$?

Vertex Cover $\leq_P$ Set Cover

- m edges $\Leftrightarrow$ N elements
- n vertices $\Leftrightarrow$ M sets
- vertex is incident to edge e $\Leftrightarrow$ set contains element

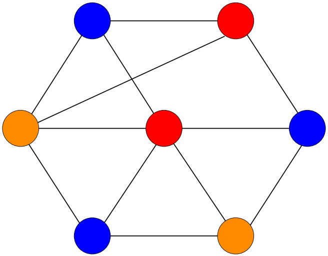
- Vertex cover is the special case of set cover where each element appears in exactly two sets.

Reductions of NP-Complete Problems



k-coloring problem

Def. A k-coloring of $G = (V, E)$ is a function $f : V \to \{1, 2, 3, \cdots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. G is k-colorable if there is a k-coloring of G.



k-coloring problem

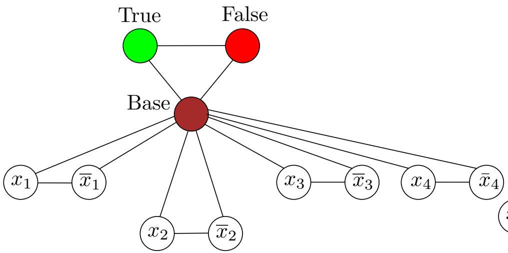
Input: a graph $G = (V, E)$

Output: whether G is k-colorable or not

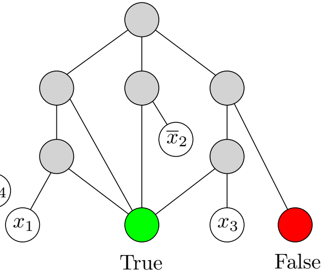
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

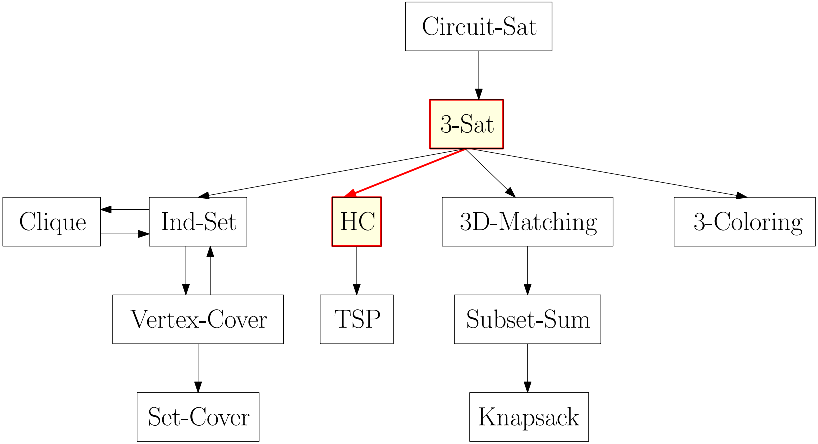
Base Graph



$x_1 \vee \neg x_2 \vee x_3$



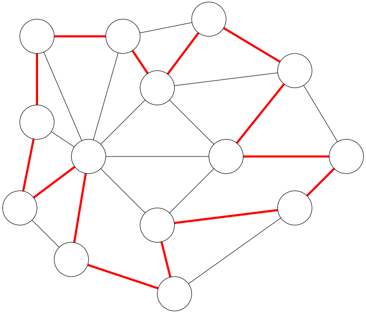
Reductions of NP-Complete Problems



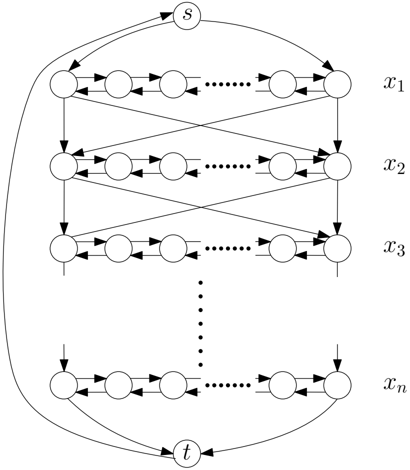
Recall: Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle

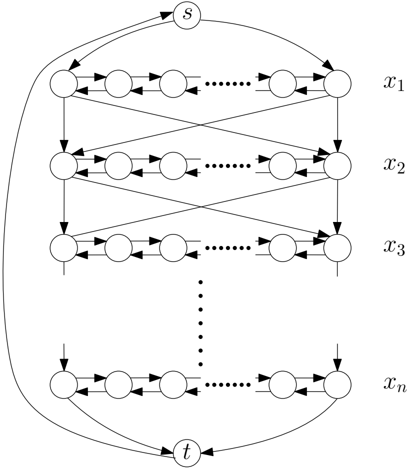


3-Sat $\leq_P$ Directed-HC



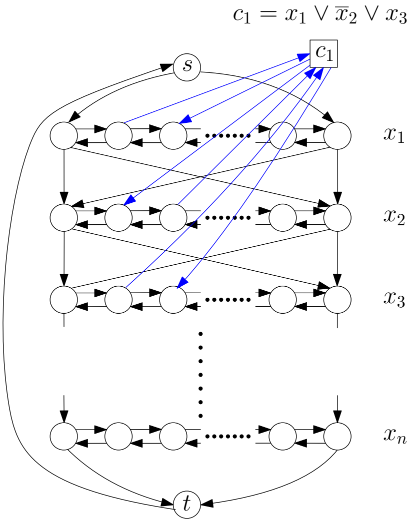
- Vertices s, t
- A long enough double-path P_i for each variable x_i
- Edges from s to P_1
- Edges from P_n to t
- Edges from P_i to P_{i+1}
- $x_i = 1 \iff$ traverse P_i from left to right
- e.g, $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$

3-Sat $\leq_P$ Directed-HC



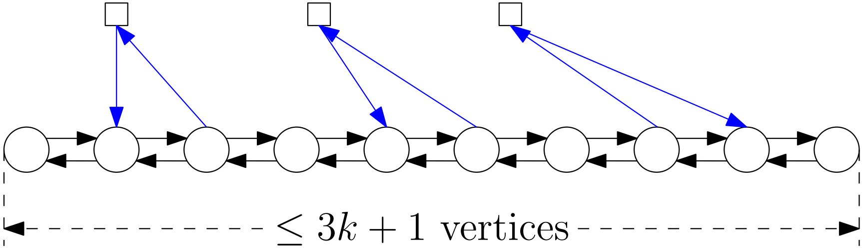
- There are exactly 2^n different Hamiltonian cycles, each correspondent to one assignment of variables
- Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.

3-Sat $\leq_P$ Directed-HC



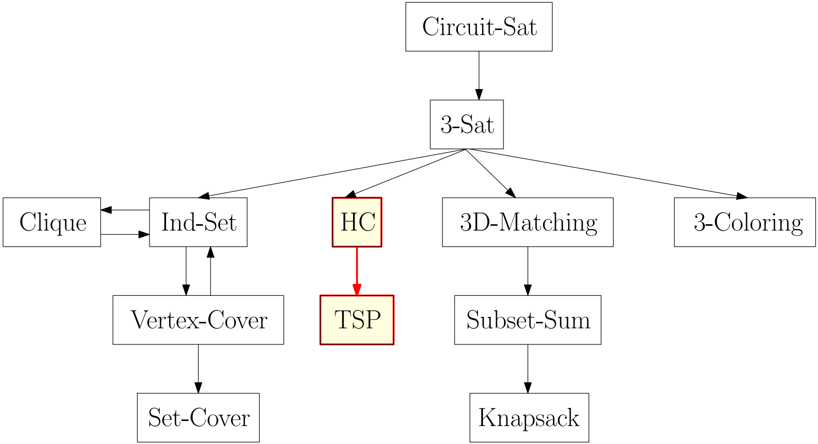
- There are exactly 2^n different Hamiltonian cycles, each correspondent to one assignment of variables
- Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.

A Path Should Be Long Enough



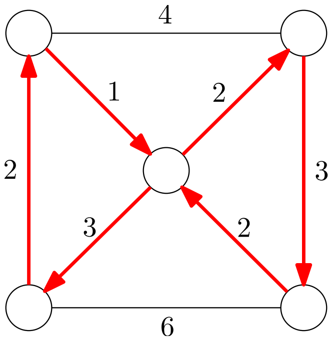
- k: number of clauses

Reductions of NP-Complete Problems



Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost

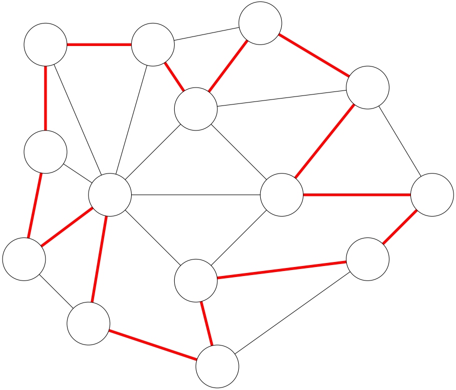


Travelling Salesman Problem (TSP)

Input: a graph $G = (V, E)$, weights $w : E \to \mathbb{R}_{\geq 0}$, and $L > 0$

Output: whether there is a tour of length at most D

HC $\leq_P$ TSP



Obs. There is a Hamilton cycle in G if and only if there is a tour for the salesman of length $n = |V|$.

A Strategy of Polynomial Reduction

- Given an instance s_Y of problem Y, show how to construct in polynomial time an instance s_X of problem such that:
  - s_Y is a yes-instance of $Y \Rightarrow s_X$ is a yes-instance of X
  - s_X is a yes-instance of $X \Rightarrow s_Y$ is a yes-instance of Y

Outline

9 NP-Completeness Theory
- P, NP and Co-NP
- Polynomial Time Reductions and NP-Completeness
- NP-Complete Problems
- Dealing with NP-Hard Problems

Dealing with NP-Hard Problems

- Faster exponential time algorithms
- Solving the problem for special cases
- Fixed parameter tractability
- Approximation algorithms

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:

- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$
- In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices

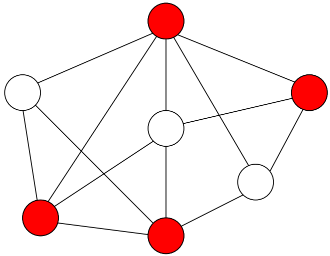
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees
- bounded tree-width graphs
- interval graphs
- $\cdots$

Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size k, for a small k (number of nodes is n, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time : $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some c independent of k
- Vertex-Cover is fixed-parameter tractable.



Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time
- There is an 2-approximation for the vertex cover problem: we can efficiently find a vertex cover whose size is at most 2 times that of the optimal vertex cover

Outline

10 Advanced Topics
- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual

Outline

Matrix Multiplication Verification

Input: 3 matrices $A, B, C \in \mathbb{Z}^{n\times n}$

Output: whether if $C = AB$

Freivald's Matrix Verification Algorithm: one experiment

1: randomly choose a vector $r \in \{0,1\}^n$
2: **return** $ABr = Cr$

- to compute $A(Br)$, need $O(n^2)$ time

	true	false
$AB = C$	1	0
$AB \neq C$	$\leq 1/2$	$\geq 1/2$

- probabilities with k experiments:

	true	false
$AB = C$	1	0
$AB \neq C$	$\leq 1/2^k$	$\geq 1 - 1/2^k$

- Frievald's algorithm is a Monta Carlo algorithm.

Def. A Monta Carlo algorithm is a randomized algorithm whose output may be incorrect with some probability.

Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the expected running time of the randomized quicksort algorithm on n elements
- $$T(n) = \frac{1}{n}\sum_{i=1}^{n}\big(T(i-1) + T(n-i)\big) + O(n)$$
$$= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + O(n)$$
- Can prove $T(n) \le c(n \log n)$ for some constant c by reduction

Indirect Analysis Using Number of Comparisons

- Running time $= O(\text{number of comparisons})$
- $\forall 1 \leq i < j \leq n$, $D_{i,j}$ indicates if we compared the i-th smallest element with the j-th smallest element
- number of comparisons $= \sum_{1\leq i<j\leq n} D_{i,j}$

Lemma
$\mathbb{E}[D_{i,j}] = \frac{2}{j-i+1} \implies \mathbb{E}\left[\text{number of comparisons}\right] O\left(n \log n\right)$.

Def. A Las-Vegas algorithm is a randomized algorithm that always outputs a correct solution but has randomized running time.

Randomized Selection Algorithm

selection(A, n, i)

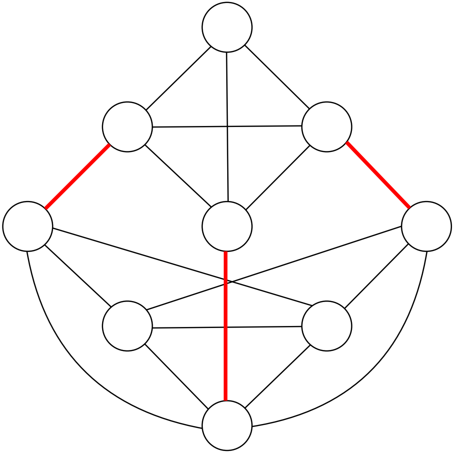
```
if n = 1 then return A
x ← random element of A (called pivot)
A_L ← elements in A that are less than x                          ▷ Divide
A_R ← elements in A that are greater than x                       ▷ Divide
if i ≤ A_L.size then
    return selection(A_L, A_L.size, i)                            ▷ Conquer
else if i > n − A_R.size then
    return selection(A_R, A_R.size, i − (n − A_R.size))           ▷ Conquer
else
    return x
```

- expected running time $= O(n)$

Global Min-Cut Problem

Input: a connected graph $G = (V, E)$

Output: the minimum number of edges whose removal will disconnect G



Karger's Randomized Algorithm for Min-Cut

$G' = (V', E') \leftarrow G$
while $|V'| > 2$ **do**
  pick $uv \in E'$ uniformly at random
  contract uv in G', keeping parallel edges, but not self-loops
return the cut in G correspondent to E'

- The probability that the algorithm succeeds is at least

$$\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{3}\right)$$
$$=\frac{n-2}{n}\times\frac{n-3}{n-1}\times\frac{n-4}{n-2}\times\cdots\times\frac{1}{3}=\frac{2}{n(n-1)}$$

Coro. Any graph G has at most $\frac{n(n-1)}{2}$ distinct minimum cuts.

- $A := \frac{n(n-1)}{2}$: algorithm succeeds with probability at least $\frac{1}{A}$
- Running the algorithm for Ak times will increase the probability to

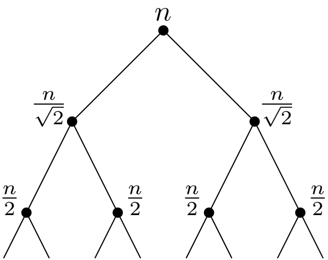
$$1-(1-\frac{1}{A})^{Ak} \geq 1-e^{-k}.$$

- To get a success probability of $1-\delta$, run the algorithm for $O(n^2 \log \frac{1}{\delta})$ times.

Karger-Stein: A Faster Algorithm

Karger-Stein$(G = (V, E))$

if $|V| \le 6$ **then return** min cut of G directly
repeat twice and return the smaller cut:
  run Karger(G) down to $\lceil n/\sqrt{2} \rceil$ vertices, to obtain G'
  consider the candidate cut returned by Karger-Stein(G')



- Running time:
  $T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2)$
- $T(n) = O(n^2 \log n)$

Max 3-SAT

Input: n boolean variables $x_1, x_2, \cdots, x_n$

m clauses, each clause is a disjunction of 3 literals from 3 distinct variables

Output: an assignment so as to satisfy as many clauses as possible

Example:

- clauses: $x_2 \vee \neg x_3 \vee \neg x_4, \quad x_2 \vee x_3 \vee \neg x_4,$
  $\neg x_1 \vee x_2 \vee x_4, \quad x_1 \vee \neg x_2 \vee x_3, \quad \neg x_1 \vee \neg x_2 \vee \neg x_4$
- We can satisfy all the 5 clauses: $x = (1, 1, 1, 0, 1)$

Outline

10 Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual

Finding Small Vertex Covers: Fixed Parameterized Tractability

Vertex-Cover Problem

Input: $G = (V, E)$

Output: a vertex cover C with minimum $|C|$

Lemma There is an algorithm with running time $O(2^k \cdot kn)$ to check if G contains a vertex cover of size at most k or not.

Vertex-Cover$(G' = (V', E'), k)$

1: **if** $|E'| = \emptyset$ **then return true**
2: **if** $k = 0$ **then return false**
3: pick any edge $(u, v) \in E'$
4: **return** Vertex-Cover$(G' \setminus u, k - 1)$ or Vertex-Cover$(G' \setminus v, k - 1)$

Outline

10 Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual

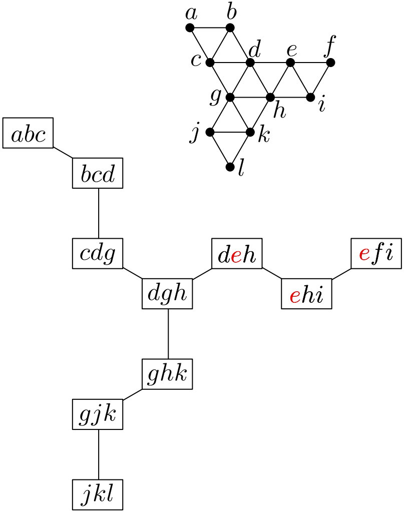
Bounded-Tree-Width Graphs

Def. A tree decomposition of a graph $G = (V, E)$ consists of

- a tree T with node set U, and
- a subset $V_t \subseteq V$ for every $t \in U$, which we call the bag for t,

satisfying the following properties:

- (Vertex Coverage) Every $v \in V$ appears in at least one bag.
- (Edge Coverage) For every $(u, v) \in E$, some bag contains both u and v.
- (Coherence) For every $u \in V$, the nodes $t \in U : u \in V_t$ induce a connected sub-graph of T.

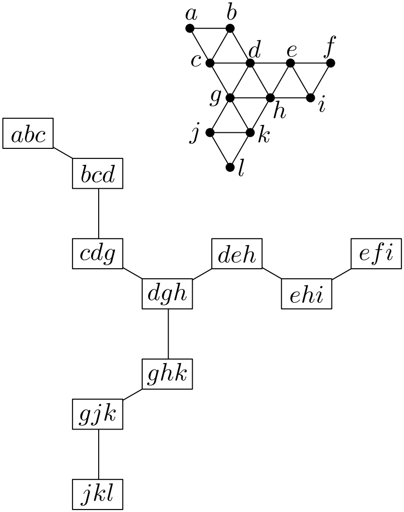


Bounded-Tree-Width Graphs

Def. The tree-width of the tree-decomposition $(T, (V_t)_{t\in U})$ is defined as $\max_{t\in U} |V_t| - 1$.

Def. The tree-width of a graph $G = (V, E)$, denoted as $\mathrm{tw}(G)$, is the minimum tree-width of a tree decomposition $(T, (V_t)_{t\in U})$ of G.

- The graph on the top right has tree-width 2.



- Many problems on graphs with small tree-width can be solved using dynamic programming.
- Typically, the running time will be exponential in $\mathrm{tw}(G)$.

Example: Maximum Weight Independent Set

- given $G = (V, E)$, a tree-decomposition $(T, (V_t)_{t \in U})$ of G with tree-width tw.
- vertex weights $w \in \mathbb{R}^V_{>0}$.
- find an independent set S of G with the maximum total weight.

- The running time of the dynamic programming: $O\big(2^{\mathsf{tw}} \cdot \mathsf{tw} \cdot n\big)$.
- It is efficient when tw is $O(\log n)$.

Outline

10 Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual

2-Approximation Algorithm for Vertex Cover

$E' \leftarrow E, C \leftarrow \emptyset$
while $E' \neq \emptyset$ **do**
    let (u, v) be any edge in E'
    $C \leftarrow C \cup \{u, v\}$
    remove all edges incident to u and v from E'
return C

- Can be extended to an f-approximation for set-cover problem with frequency f.

Set Cover

Input: $U, |U| = n$: ground set

$S_1, S_2, \cdots, S_m \subseteq U$

Output: minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$

Greedy Algorithm for Set Cover

1: $C \leftarrow \emptyset, U' \leftarrow U$
2: **while** $U' \neq \emptyset$ **do**
3: choose the i that maximizes $|U' \cap S_i|$
4: $C \leftarrow C \cup \{i\}, U' \leftarrow U' \setminus S_i$
5: **return** C

- g: minimum number of sets needed to cover U

Lemma Let $u_t, t \in \mathbb{Z}_{\geq 0}$ be the number of uncovered elements after t steps. Then for every $t \geq 1$, we have

$$u_t \leq \left(1 - \frac{1}{g}\right) \cdot u_{t-1}.$$

- In at most $\lceil g \ln(n+1) \rceil$ iterations, all elements will be covered.

Maximum Coverage

Input: $U, |U| = n$: ground set,
$S_1, S_2, \cdots, S_m \subseteq U$, $k \in [m]$

Output: $C \subseteq [m], |C| = k$ with the maximum $\bigcup_{i \in C} S_i$

Greedy Algorithm for Maximum Coverage

$C \leftarrow \emptyset, U' \leftarrow U$
for $t \leftarrow 1$ to k **do**
    choose the i that maximizes $|U' \cap S_i|$
    $C \leftarrow C \cup \{i\}, U' \leftarrow U' \setminus S_i$
return C

Theorem Greedy algorithm gives $(1 - \frac{1}{e})$-approximation for maximum coverage.

Outline

10 Advanced Topics
- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual

Knapsack Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} v_i \qquad \text{s.t.} \sum_{i \in S} w_i \le W.$$

- Let A be some integer to be defined later
- $v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item i
- Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i]: v'(S) \geq V'} w(S)$

$$f[i, V'] = \begin{cases} 0 & V' \leq 0 \\ \infty & i = 0, V' > 0 \\ \min \begin{Bmatrix} f[i-1, V'] \\ f[i-1, V' - v'_i] + w_i \end{Bmatrix} & i > 0, V' > 0 \end{cases}$$

- Output A times the largest V' such that $f[n, V'] \leq W$.

Makespan Minimization on Identical Machines

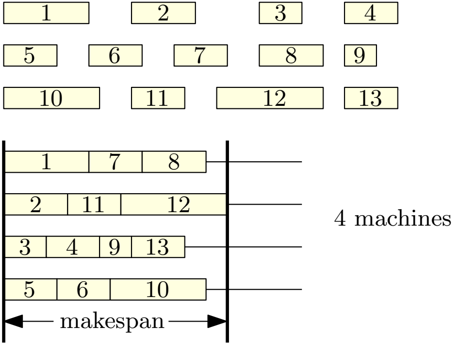
Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Output: schedule of jobs on machines with minimum makespan

$\sigma : [n] \to [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$



$1 + \epsilon$-Approximation

- Dynamic Programming for Big jobs:
  - Round job sizes so that there are only $k = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ distinct sizes
  - Running time exponential in k
- Greedily add small jobs.

Outline

10 Advanced Topics
- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual

Weighted Vertex-Cover Problem

Input: $G = (V, E)$ with vertex weights $\{w_v\}_{v \in V}$

Output: a vertex cover S with minimum $\sum_{v \in S} w_v$

- Linear programming relaxation for WVC:

$$(\mathrm{LP}_{\mathrm{WVC}}) \qquad \min \quad \sum_{v \in V} w_v x_v \qquad \text{s.t.}$$
$$x_u + x_v \geq 1 \qquad \forall (u, v) \in E$$
$$x_v \in [0, 1] \qquad \forall v \in V$$

Algorithm for Weighted Vertex Cover

Algorithm for Weighted Vertex Cover

1: Solving (LP_{WVC}) to obtain a solution $\{x^*_u\}_{u\in V}$
2: Thus, $\text{LP} = \sum_{u\in V} w_u x^*_u \leq \text{IP}$
3: Let $S = \{u \in V : x_u \geq 1/2\}$ and output S

Lemma S is a vertex cover of G.

Lemma $\text{cost}(S) := \sum_{u\in S} w_u \leq 2 \cdot \text{LP}$.

Algorithm for Weighted Vertex Cover

Algorithm for Weighted Vertex Cover

1: Solving (LP_{WVC}) to obtain a solution $\{x_u^*\}_{u \in V}$
2: Thus, $\text{LP} = \sum_{u \in V} w_u x_u^* \leq \text{IP}$
3: Let $S = \{u \in V : x_u^* \geq 1/2\}$ and output S

Lemma S is a vertex cover of G.

Lemma $\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot \text{LP}$.

LP Relaxation

$$\min \quad \sum_{v \in V} w_v x_v$$

$$x_u + x_v \geq 1 \qquad \forall (u,v) \in E$$

$$x_v \geq 0 \qquad \forall v \in V$$

Dual LP

$$\max \quad \sum_{e \in E} y_e$$

$$\sum_{e \in \delta(v)} y_e \leq w_v \qquad \forall v \in V$$

$$y_e \geq 0 \qquad \forall e \in E$$

- Algorithm constructs a maximal dual solution y, and returns the set of all vertices whose dual constraints are tight.