COMP 3001 - Algorithms

LECTURER:	Dr. Antonis Symvonis
OFFICE:	Madsen Building G34a, Phone:9351-4291
OFFICE HOURS :	Wednesday 10-12am.
TEXT:	Cormen, Leiserson, Rivest, *Introduction to Algorithms*

ASSESSMENT

3 written assignments	20%
Mid-term test (13 April)	25%
Final Exam	55%

IMPORTANT !! No late assignments/projects will be marked.

IMPORTANT DATES

24 March (Wednesday, 5pm) :	Assignment #1 is due
12 April (Monday, 2pm):	Assignment #2 is due
13 April (Tuesday):	**Mid-term test**
26 May (Wednesday, 5pm):	Assignment #3 is due

OTHER

- The final exam will be comprehensive
- The mid-term test will cover all the material taught **before** the lecture on Monday 12 April.
- Results will be scaled

What is an algorithm?

Input $\Longrightarrow$ ALGORITHM $\Longrightarrow$ Output

- An **algorithm** is a well defined computational procedure which takes some value, or a set of values, as **input**, and produces some value, or set of values, as **output**.

EXAMPLES

- Given a list of names, produce a new list that contains the same names sorted in alphabetical order.
- Given a list of numbers, compute the maximum and the minimum among them.

- A **computational problem** specifies the desired input/output relationship.
- An **algorithm** describes a specific computational procedure for achieving the input/output relationship.

Why do we study algorithms?

The obvious solution to a problem is not always efficient.

EXAMPLE

You are given a map with n cities and the cost for travelling between each pair of cities. Find the cheapest way to travel from city A to city B.

A simple solution

1. Compute the cost of each path from A to B.
2. Select the cheapest one.

Question: What is the number of possible $A \longrightarrow B$ paths?

Answer:

Visit	no	intermediate city	1
"	1	"	$(n-2)$
"	2	intermediate cities	$(n-3)(n-2)$
"	3	"	$(n-4)(n-3)(n-2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
"	k	"	$\binom{n-2}{k}k! = \frac{(n-2)!}{(n-2-k)!\,k!}k! = \frac{(n-2)!}{(n-2-k)!}$

- So, there are **at least** $(n-2)!$ paths.
- For large n, it is impossible to check all paths!

  $20! \approx 2.43 \times 10^{18}$
  $50! \approx 3.04 \times 10^{64}$
  $100! =$ **E** (for my pocket calculator)

We need to try more sophisticated solutions.

Quiz

You are given 12 balls of identical shape all of which, but one, are of equal weight. The ball of different weight might be heavier or lighter. You are also given a balance.

By using the balance **only 4 times**, find the ball of irregular weight and the type of irregularity (heavier/lighter).

The sorting problem

Input: A sequence of numbers $< a_1,\ a_2,\ a_3,\ \cdots\ a_n >$.

Output: A permutation (reordering) $< a'_1,\ a'_2,\ a'_3,\ \cdots\ a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq a'_3 \leq\ \cdots \leq a'_n$.

Example

$$< 6, 5, 3, 4, 1, 2 > \Longrightarrow \boxed{\text{SORTING}} \Longrightarrow < 1, 2, 3, 4, 5, 6 >$$

How do we sort?

Algorithm 1: *Insertion Sorting*

- Process one input number at a time.
- Maintain a sorted sequence of the already processed numbers.
- **Insert** the currently processed number in the correct position of the maintained sorted sequence.

Example Input:$< 6, 5, 3, 4, 1, 2 >$

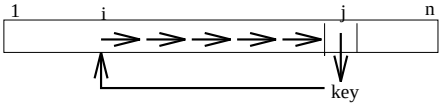
Current element	Sorted list
6	<>
5	$< 6 >$
3	$< 5, 6 >$
4	$< 3, 5, 6 >$
1	$< 3, 4, 5, 6 >$
2	$< 1, 3, 4, 5, 6 >$
NIL	$< 1, 2, 3, 4, 5, 6 >$

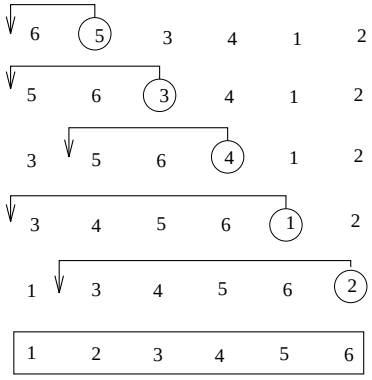
Question: How **good** is insertion sort?

Good usually means **fast**!

```
    Insertion_Sort(A)

1   for j ← 2 to length(A) do
2       key ← A[j]
3       /* Insert A[j] into the sorted sequence A[1...j − 1] */
4       i ← j − 1
5       while i > 0 and a[i] > key do
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← key
```





Note: Insertion sort works "**in place**".

	Insertion_Sort(A)	Cost	Times
1	**for** $j \leftarrow 2$ **to** $length(A)$ **do**	c_1	n
2	$key \leftarrow A[j]$	c_2	$n-1$
3	/* ... */	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ **and** $a[i] > key$ **do**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

where, t_j is the number of times the **while loop** is executed for the value j.

- $T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n}(t_j - 1)$

- The running time depends on n, the input size.
- For inputs of the same size, the running time might be different.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

- If already sorted input, $t_j = 1$, for $j = 2, 3, \cdots, n$.

$$\begin{aligned} \Longrightarrow \; T(n) = & \quad c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5(n-1) + 0 \\ = & \quad c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1) \\ = & \quad (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$ **[linear on n]**

- If the input is sorted in decreasing order, $t_j = j$, for $j = 2, 3, \cdots, n$.

$$\begin{aligned} \Longrightarrow \; T(n) = & \quad c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5(\tfrac{n(n+1)}{2} - 1) + (c_6 + c_7)\tfrac{n(n-1)}{2} \\ & \quad \vdots \\ = & \quad \tfrac{(c_5 + c_6 + c_7)}{2} n^2 + (c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_8)n \\ & \quad -(c_2 + c_4 + c_5 + c_8) \\ = & \quad an^2 + bn + c \quad a, b, c \text{ are constants} \end{aligned}$$ **[quadratic on n]**

In this course, we will be interested in the **order of growth** of the running time.

Worst case analysis

- Gives an upper bound on the running time for any input.
- For some algorithms the worst case occurs very often (eg. searching).
- Some times the average case is as bad as the worst case.

Average case analysis

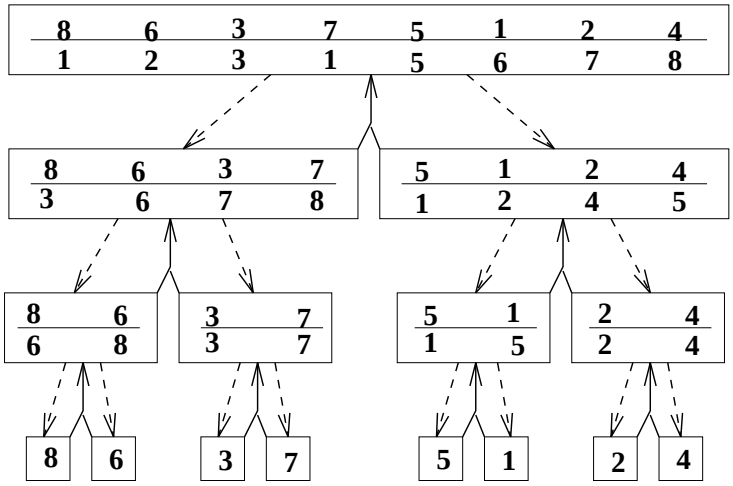
- Assumes that all inputs of a given size are equally likely to occur.
- For insertion sort: $t_j = j/2 \implies$
  $T(n) = \cdots = a'n^2 + b'n + c' = \Theta(n^2)$

  Worst case for insertion sort is also $\Theta(n^2)$.

Algorithm 2: *Merge Sort*

- Split the input into 2 parts.
- **Recursively** sort each of them.
- **Merge** the two sorted parts.

Example



How do we merge?

Input: 2 sorted lists A and B of a and b elements, respectively.

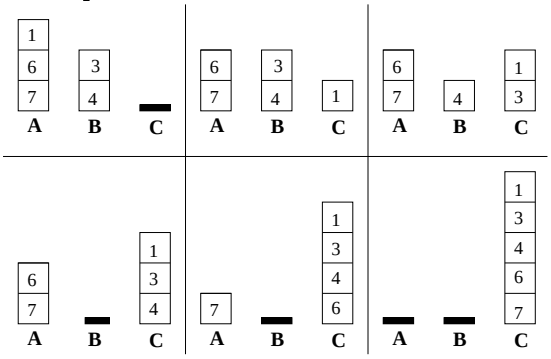
Output: A sorted list C which contains all the elements of A and B. C is of length $n = a + b$.

Merge(A, B, C)

while there are still elements in A or B **do**

- Compare the "first" elements of A and B.
- Move the minimum of them from its corresponding list to the end of C.

Example



Time complexity: $\Theta(n)$

Merge-Sort(A, p, r)

if $p < r$ **then**

- $q \leftarrow \lfloor (p+r)/2 \rfloor$
- Merge-Sort(A, p, q)
- Merge-Sort$(A, q+1, r)$
- Merge(A, p, q, r)

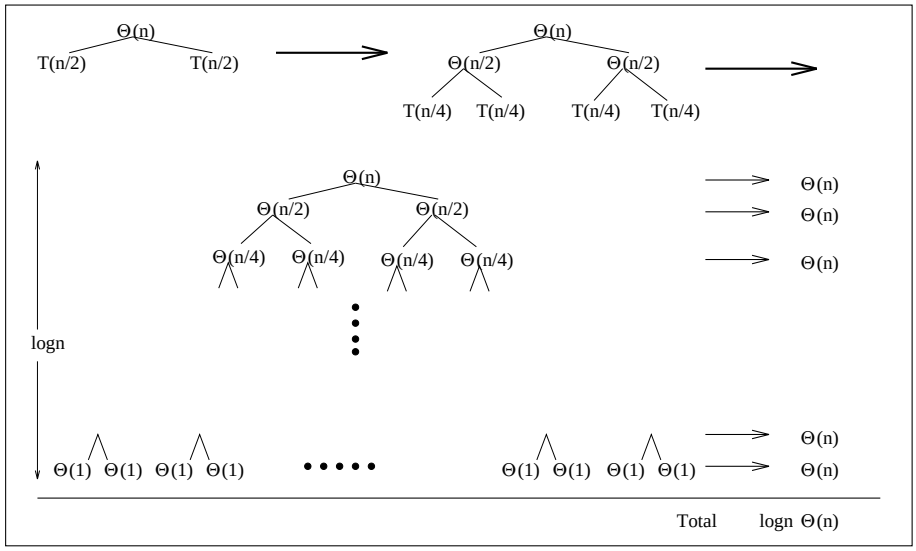
Time analysis

- If the problem size is small, say $n \leq c$ for some constant c, we can solve the problem in constant, i.e., $\Theta(1)$ time.
- Let $T(n)$ be the time needed to sort for input of size n.
- Let $C(n)$ be the time needed to merge 2 lists of total size n. We know that $C(n) = \Theta(n)$.
- Assume that the problem can be splitten into 2 subproblems in constant time and that $c = 1$. Then

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Question: Is there a closed formula for $T(n)$?

W.l.o.g., assume $n = 2^k$ (or, $\log n = k$).



$\Longrightarrow T(n) = \Theta(n \log n)$

Observations:

- For merge-sort we have that:

$$\left.\begin{array}{l}\text{Worst case time complexity}\\ \text{Average case time complexity}\\ \text{Best case time complexity}\end{array}\right\} = \Theta(n \log n)$$

- For small values of n, insertion-sort might be faster than merge-sort.

> We are interested in the performance of algorithms when the input size $n \longrightarrow \infty$.

We call this kind of analysis **asymptotic**.

In this course ...

- We will compare different algorithms for the same problem with respect to their asymptotic behaviour.

  In that sense,

A	$\Theta(n)$	algorithm is better than a
	$\Theta(n \log n)$	algorithm, which is better than a
	$\Theta(n \log^2 n)$	"
	$\Theta(n^2)$	"
	$\Theta(n^3)$	"
	$\Theta(2^n)$	"
	$\Theta(n2^n)$	"
	$\Theta(n!)$	"
	$\Theta(2^{2^n})$	"

Reading material

Introduction (from Cornem, Leiserson, Rivest).
Insertion sort (pp. 2-11).
Merge sort (pp. 11-15).

Suggested reading:

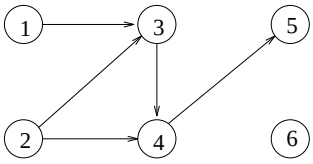
Mathematical foundations (pp. 21-135).
Pay special attention to the *"Growth of functions"* (pp. 23-41).

Note You are supposed to (AT LEAST) have seen the material in pp. 21-135.

Graphs

Definition A *graph* G is a pair (V, E) where V is the set of *vertices* and E is the set of *edges*.

Example



V={1,2,3,4,5,6}

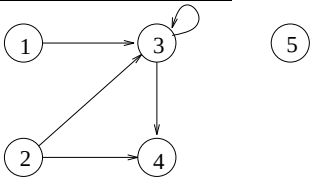
E={(1,3), (2,3), (2,4), (3,4), (4,5)}

Each edge is an **ordered** pair of vertices.

Why do we study graphs?

- To model a huge number of real life problems.
- They are an important tool in the design of algorithms.

Directed graphs (digraphs)

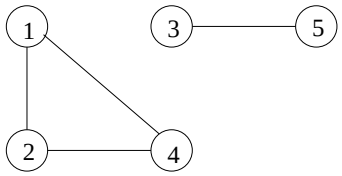


V={1,2,3,4,5}

E={(1,3), (2,3), (2,4), (3,3), (3,4)}

- Vertex 5 is an *isolated* vertex.
- Edge $(3, 3)$ is a *self-loop*.
- The *in-degree* of vertex v is the number of edges entering vertex v (denoted $d^-(v)$).
$$d^-(1) = d^-(2) = d^-(5) = 0, \quad d^-(3) = 3, \quad d^-(4) = 2$$
- The *out-degree* of vertex v is the number of edges leaving vertex v (denoted $d^+(v)$).
$$d^+(1) = 1, \quad d^+(2) = d^+(3) = 2, \quad d^+(4) = d^+(5) = 0$$
- $\sum_{v \in V} d^-(v) = \sum_{v \in V} d^+(v) = |E|$

Undirected graphs



V={1,2,3,4,5}

E={(1,2), (1,4), (2,4), (3,5)}

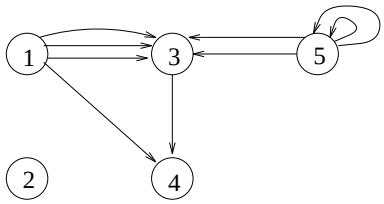
Each edge $e \in E$ is an **unordered** pair of vertices.

- The *degree* of vertex v is the number of edges incident to v (denoted $d(v)$).
$$d(1) = d(2) = d(4) = 2, \quad d(3) = d(5) = 1$$

- $\sum_{v \in V} d(v) = 2 \cdot |E|$

Multigraphs (directed or undirected)

- E is allowed to contain the same pair of vertices (u, v) more than 1 time.



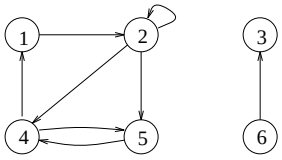
V={1,2,3,4,5}

E={(1,3), (1,3), (1,3), (1,4),
(3,4),
(5,3), (5,3), (5,5), (5,5)}

Paths in graphs

A *path of length* k from a vertex u to a vertex u' in graph $G = (V, E)$ is a sequence $< v_0,\ v_1,\ v_2, \cdots,\ v_k >$ of vertices such that $u = v_0,\ u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \cdots, k$.

- The *length* of a path is the number of edges in it.
- A path is *simple* if all the vertices in it are distinct.



$< 2, 4, 5, 4, 1 >$ is a path of length 4 (not simple).
$< 2, 4, 1 >$ is a simple path of length 2.

- A path $< v_0,\ v_1, \cdots,\ v_k >$ forms a *cycle* if $v_0 = v_k$ and at least 1 edge is in the path.

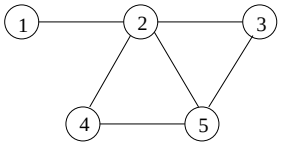
  $< 2, 4, 5, 4, 1, 2 >$ is a cycle.

- A cycle is *simple* if all the vertices in it are distinct.

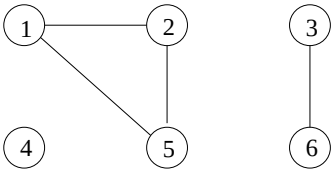
- A graph with no cycles is called *acyclic*.

Connected components

- An undirected graph is *connected* if every pair of vertices is connected by a path.



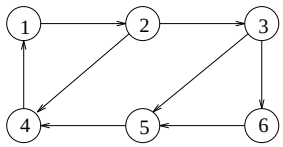
- The *connected components* of an undirected graph are the equivalence classes of vertices under the *"is reachable from"* relation.



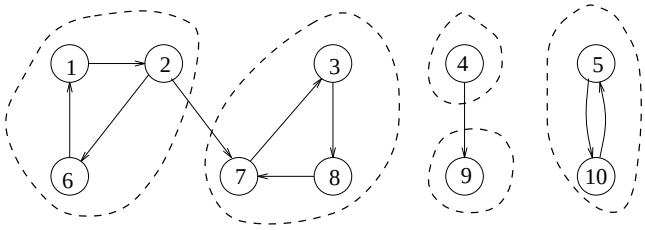
Connected components:
{1,2,5}
{3,6}
{4}

Strongly connected components

- A directed graph is *strongly connected* if every 2 vertices are reachable from each other.



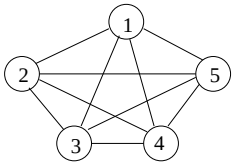
- The *strongly connected components* of a directed graph are the equivalence classes of vertices under the *"are mutually reachable"* relation.



Strongly connected components {1,2,6} {3,7,8} {4} {9} {5,10}

Complete graphs (cliques)

- An undirected graph in which every pair of vertices is connected by an edge.

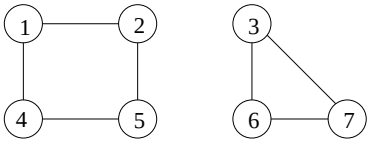


$$|E| = \frac{|V| \cdot (|V| - 1)}{2}$$

$$d(v) = |V| - 1, \forall v \in V$$

d-regular graphs

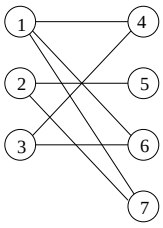
- An undirected graph in which all vertices are of degree d.



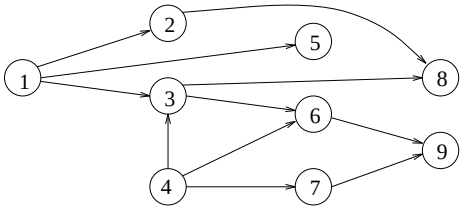
A clique is a $(|V| - 1)$-regular graph.

Bipartite graphs

- An undirected graph $G = (V, E)$ in which V is partitioned into two sets, V_1 and V_2, such that $(u, v) \in E$ implies that either ($u \in V_1$ and $v \in V_2$) or ($u \in V_2$ and $v \in V_1$).

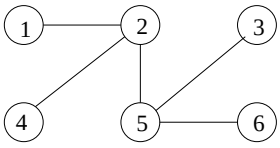


DAGs (Directed Acyclic Graphs)



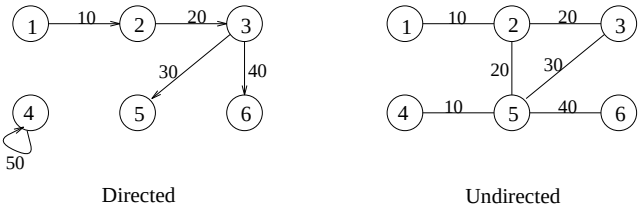
Trees

- An undirected connected graph with no cycles.



Weighted graphs

- A graph in which each edge is associated with a weight.

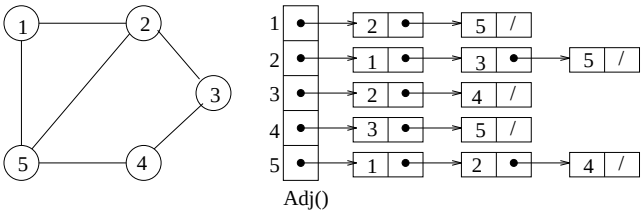


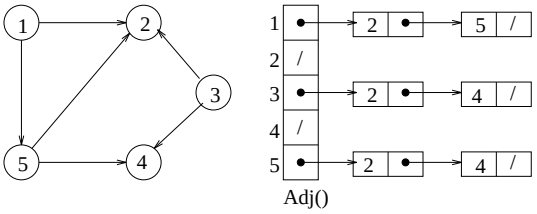
Notational convention In asymptotic notation ($\Theta()$, $\Omega()$, $O()$) the symbol V denotes $|V|$ and the symbol E denotes $|E|$.

Representation of graphs

Adjacency list representation

- An array Adj of $|V|$ lists, one for each vertex in V.
- For each $u \in V$, the adjacency list $Adj(u)$ contains all the vertices v such that there is an edge (u, v) in E.

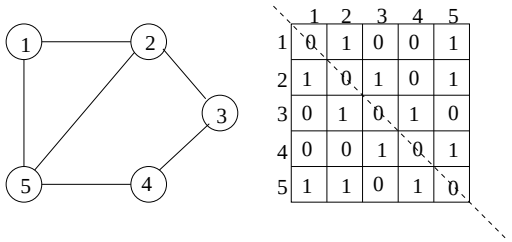




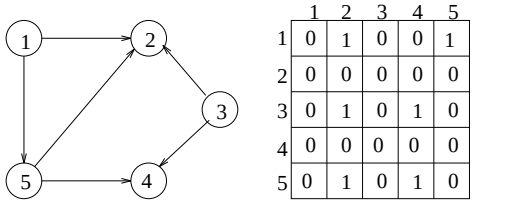
Required space: $O(V + E)$.

Adjacency matrix representation

- A $|V| \times |V|$ matrix $A = (a_{ij})$ such that $a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$



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



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

- The adjacency matrix of an undirected graph is symmetric along the diagonal.

Required space: $\Theta(V^2)$.

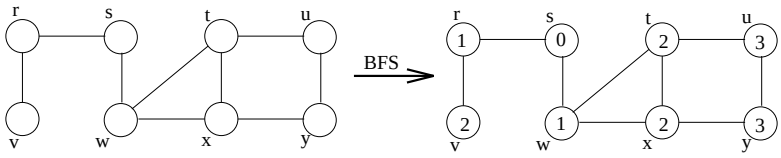
- Use the adjacency list representation for *sparse* graphs ($|E|$ is much less than $|V|^2$).
- Use the adjacency matrix representation for *dense* graphs ($|E|$ is close to $|V|^2$).
- When the adjacency list representation is used, it takes $O(|V|)$ time to check if a particular edge (u, v) exists.
- When the adjacency matrix representation is used, it takes $O(1)$ time to check if a particular edge (u, v) exists.

Breadth-First Search (BFS)

Given a graph $G = (V, E)$ and a distinguished source vertex s, we want to systematically explore the edges of G and to "discover" every vertex that is reachable from s.

- BFS computes the distances (smallest number of edges) to all vertices reachable from s.
- BFS builds a *breadth-first tree* from s which contains all such reachable vertices.

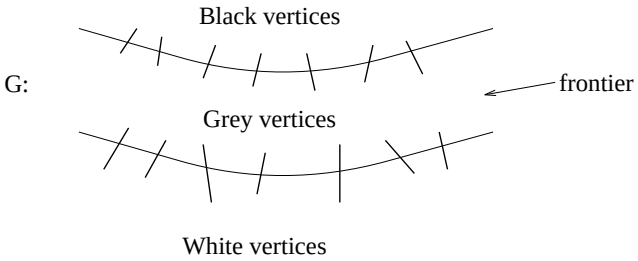
Example



- BFS expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

- It discovers all vertices at distance k from s before discovering any vertex at distance $k + 1$.

- During its execution, BFS colours the vertices *white, grey*, or *black*.
  White vertices: undiscovered.
  Grey vertices: discovered, will be used to discover new vertices.
  Black vertices: discovered, will not be used to discover new vertices.



BFS(G, s)

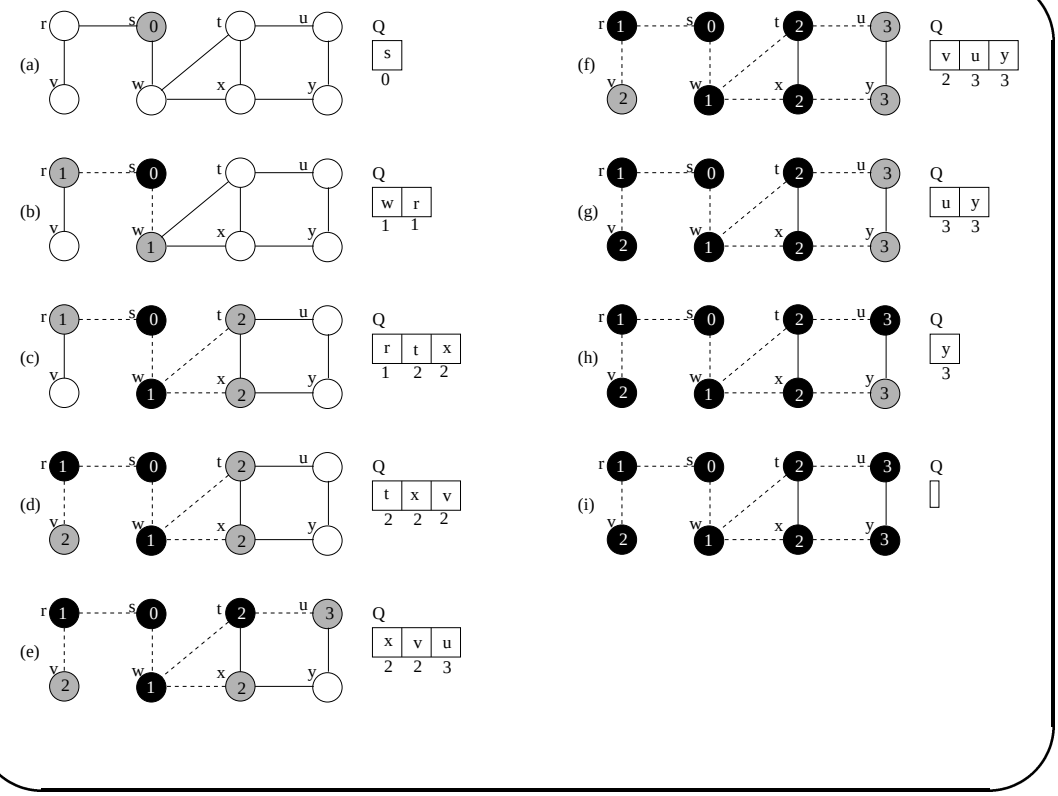
for each vertex $u \in V[G] - \{s\}$
  do $color[u] = white$
    $d[u] = +\infty$
    $\pi[u] = \text{NIL}$

$color[s] = grey$
$d[s] = 0$
$\pi[s] = \text{NIL}$

$Q \leftarrow \{s\}$ /* Q is a FIFO queue */
while $Q \neq \emptyset$
  do $u = head(Q)$
    for each vertex $v \in Adj[u]$
      do if $color[v] = white$
        then $color[v] = grey$
          $d[v] = d[u] + 1$
          $\pi[v] = u$
          $enqueue(Q, v)$
      $dequeue(Q)$
      $color[u] = black$

Analysis $O(V + E)$

(Provided that the adjacency list representation is used.)



Properties of BFS

- $d[u] = \delta(s, u)$
  ($\delta(s, u) \equiv$ The shortest path distance from s to u, is the minimum number of edges in any path from s to u).

- BFS produces a *breadth-first search tree* consisting of vertices reachable from s such that, for each vertex u in the tree, the simple path from s to u is also the shortest path.

How to recover the shortest paths

Print_Path(G, s, v)

if $v = s$
  then print s
  else if $\pi[v] =$ NIL
    then print " no path from s to v exists"
    else Print_Path$(G, s, \pi[v])$
      print v

Analysis: Linear time on the number of vertices in the path.

- BFS can be applied to **both** directed and undirected graphs.

Reading Material

§5.4 "Graphs" pp. 86-90.

§23.1 "Representation of graphs" pp. 465-469.

§23.2 "Breadth-First Search" pp.469-477.
(Proofs of properties are not required but are highly recommended.)

Suggested Reading

§5.5 "Trees" pp. 91-97

Suggested Exercises

5.4-1 – 5.4-7 pp.91-97.

23.1-1 – 23.1-7 pp.468.

23.2-1 – 23.2-6 pp. 476.

Depth-First Search (DFS)

- DFS searches "deeper" in the graph whenever possible.

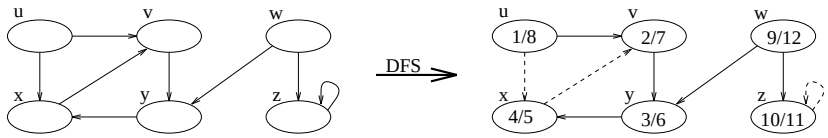
- Edges are explored out of the most recently discovered vertex u that still has unexplored vertices leaving it.

- DFS time-stamps each vertex.

  $d[u]$: the time u is first discovered.
  $f[u]$: the time u is finished, i.e., u's adjacency list has been completely examined.

- DFS builds a *depth-first forest* composed of several *depth-first trees*.

Example



- During its execution, DFS colours the vertices *white*, *grey*, or *black*.

  White vertices: undiscovered vertices.

  Grey vertices: discovered vertices that still have unexplored edges leaving them.

  Black vertices: discovered vertices of which the adjacency lists are completely examined.

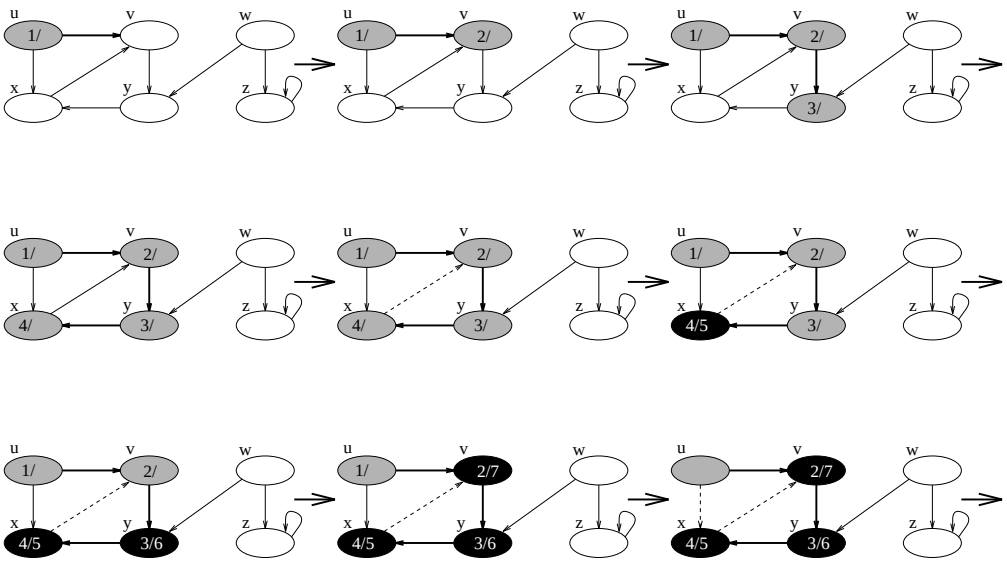
DFS(G)

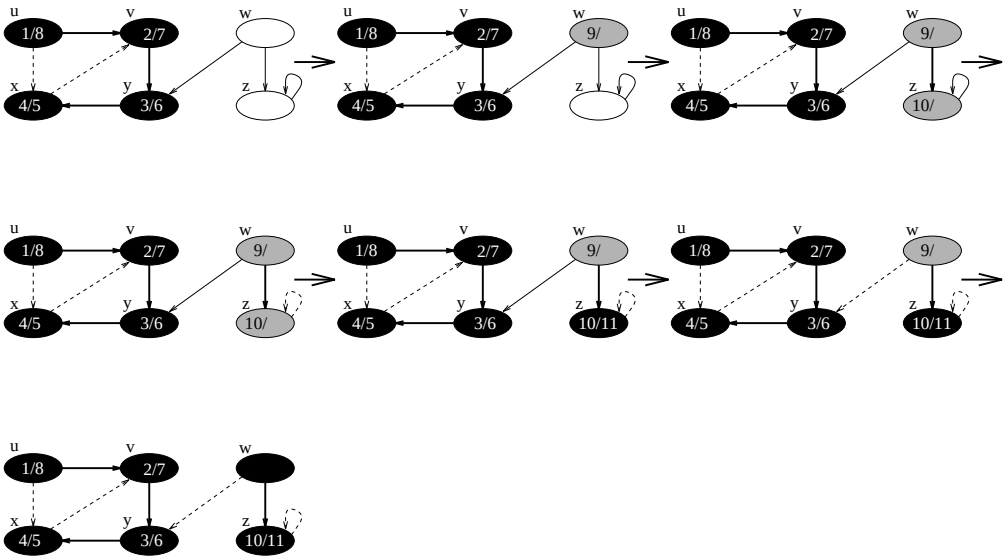
for each vertex $u \in V[G]$
  do $color[u] = white$
    $\pi[u] = \text{NIL}$
$time = 0$
for each vertex $u \in V[G]$
  do if $color[u] = white$
    then DFS_visit(u)

DFS_visit(u)

$color[u] = grey$
$time = time + 1$
$d[u] = time$
for each vertex $v \in Adj[u]$
  do if $color[v] = white$
    then $\pi[v] = u$
      DFS_visit(v)
$color[u] = black$
$time = time + 1$
$f[u] = time$

Analysis $O(V + E)$





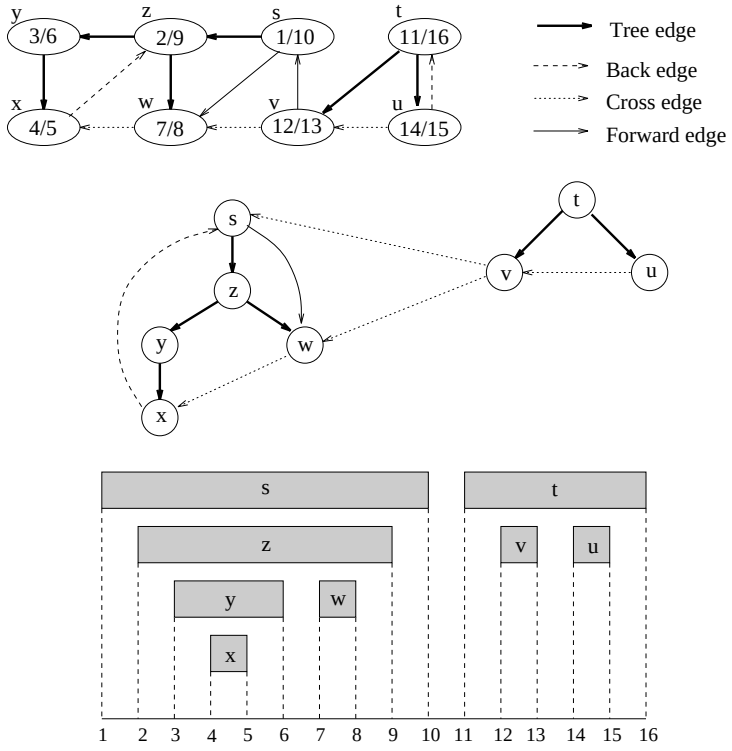
Properties of DFS

- **Classification of edges**

  1. *Tree edges* are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was discovered by exploring edge (u, v).
  2. *Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in the depth-first tree.
  3. *Forward edges* are those edges (u, v) connecting a vertex u to a descendant v in the depth-first tree.
  4. *Cross edges* are all other edges. They are edges between vertices in the same tree, as long as one vertex is not the ancestor of the other, or edges between vertices in different depth-first trees.

- **Parenthesis structure**

  The *discovery* and *finishing* times have a parenthesis structure.



How to classify edges

- Each edge (u, v) can be classified by the colour of vertex v that is reached when the edge is first explored.

$$(u,v) \text{ is } \begin{cases} \textit{tree edge} & \text{if } v \text{ is white.} \\ \textit{back edge} & \text{if } v \text{ is grey.} \\ \textit{forward edge} & \text{if } v \text{ is black and } d[u] < d[v]. \\ \textit{cross edge} & \text{if } v \text{ is black and } d[u] > d[v]. \end{cases}$$

Theorem In a depth-first search of an undirected graph G, every edge in G is either a tree edge or a back edge.

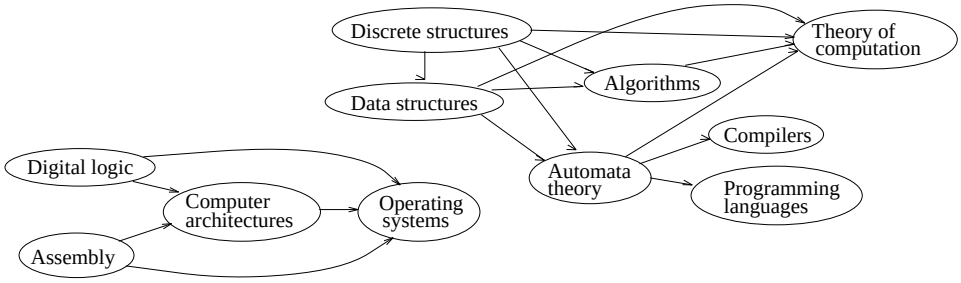
Topological sort

- A topological sort of a DAG $G = (V, E)$ is a linear ordering of all of its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.

- If the graph is not acyclic, then no linear ordering is possible.

Application Denotes precedences among events.

Example Course prerequisites.

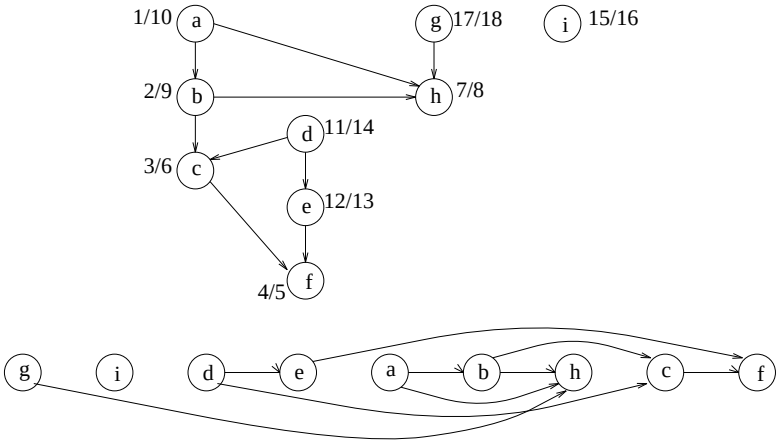


Topological_Sort(G)

1. Call DFS(G) to compute the finishing times $f[u]$ for each vertex u.
2. As each vertex is finished, insert it at the front of a linked list.
3. Return the linked list of the vertices.

Time analysis $O(V + E)$

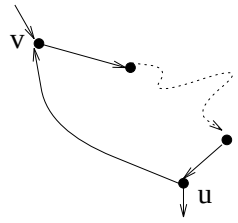
Example



Lemma A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

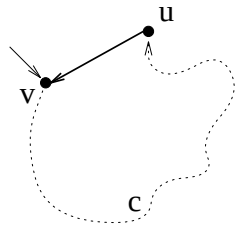
Proof

"$\Longrightarrow$" G is acyclic $\Rightarrow$ G has no back edges



Suppose there is a back edge (u, v). Then, v is an ancestor of u in the depth-first forest. Thus, there is a path from v to u in G, That path together with (u, v) form a cycle. $\rightarrow\leftarrow$

"$\Longleftarrow$" G has no back edges $\Rightarrow$ G is acyclic



Assume that G contains a cycle c. We will show that a depth-first search yields a back edge.
Let v be the first edge discovered in c. By appropriately arranging the vertices of c in the adjacency lists, we can make the DFS reach u and thus, u becomes a descendant of v. Then, (u, v) becomes a back edge. $\rightarrow\leftarrow$

Theorem Topological_Sort(G) produces a topological sort of a DAG G

Proof

It suffices to show that for any pair of vertices $u,\ v \in V$, if there is an edge in G from u to v , then $f[v] < f[u]$.

Consider any edge (u, v) explored by DFS(G).

When (u, v) is explored, v **cannot be grey**. (v would be an ancestor of u, and thus, (u, v) a back edge).

Thus, v is either white or black.

- **If v is white**, it becomes a descendant of u. $\Longrightarrow f[v] < f[u]$.
- **If v is black**, then v is finished. $\Longrightarrow f[v] < f[u]$.

□

Reading Material

§23.3 "Depth-first search".

§23.4 "Topological sort".

Suggested Exercises

23.3-1, 23.3-2, 23.3-4, 23.3-6, 23.3-7, 23.3-8.

23.4-1, 23.4-2, 23.4-3, 23.4-5.

The Divide-and-Conquer Approach

- A recursive approach
  - **Divide** the problem into a number of subproblems.
  - **Conquer** the subproblems by solving them recursively.
  - **Combine** the solutions of the subproblems into the solution of the original problems.

Example: Merge Sort

Analysis: Based on *recurrence relations*.

The n^{th} Power of an $n \times n$ Matrix

Input: An $n \times n$ matrix A.

Output: $A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$

Fact: $A^2 = A \times A$ can be computed in $O(n^3)$ time.

- **A simple iterative approach**

```
(1)   RESULT = A
(2)   for i = 2 to  n do
(3)         RESULT = RESULT × A
```

- **Analysis**

Step 1: $\Theta(n^2)$

Step 2: $\Theta(n)$

Step 3: $\Theta(n^4)$ [$n-1$ operations of $O(n^3)$ each]

Total: $\Theta(n^4)$

Question: Can we do better?

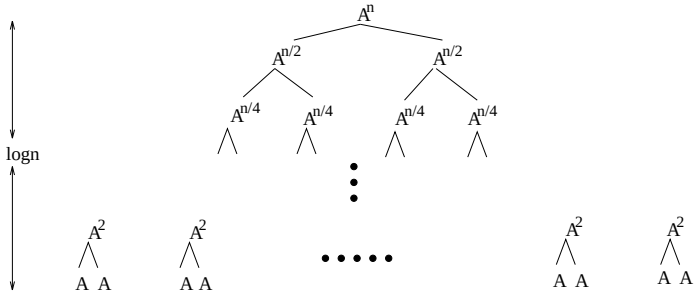
A divide-and-conquer approach

Assume that $n = 2^k$ (or, $\log n = k$). Then,

$$A^n = \overbrace{A \times A \times \cdots \times A}^{n \text{ times}} = (\overbrace{A \times A \times \cdots \times A}^{\frac{n}{2} \text{ times}}) \times (\overbrace{A \times A \times \cdots \times A}^{\frac{n}{2} \text{ times}}) \quad \text{or,}$$

$$A^n = (A^{\frac{n}{2}}) \times (A^{\frac{n}{2}}) = \left((A^{\frac{n}{4}}) \times (A^{\frac{n}{4}})\right) \times \left((A^{\frac{n}{4}}) \times (A^{\frac{n}{4}})\right) = \cdots \quad \text{or,}$$

$$A^n = (A^{\frac{n}{2}})^2 = \left((A^{\frac{n}{4}})^2\right)^2 = \left(\left((A^{\frac{n}{8}})^2\right)^2\right)^2 = \underbrace{\left(\left((A^2)^2\right)^2 \cdots\right)^2}_{\log n}$$



- **A divide-and-conquer algorithm**

```
RESULT = A
for i = 1 to log n do
        RESULT = RESULT × RESULT
```

Time analysis: $\Theta(n^3 \log n)$

More generally: If matrix multiplication can be done on $O(f(n))$ time then A^n can be computed in $O(f(n) \log n)$ time.

Comment: $f(n) = \Omega(n^2)$, i.e., to multiply two $n \times n$ matrices we need to perform $\Omega(n^2)$ operations.

Binary search

Input: A sorted array A of n elements and an element key.

Output: The position of key in A. If key is not in A we return 0.

- **A trivial algorithm**

Search the array from the beginning to the end.
If you meet key during the search report its position.
Otherwise return 0.

Analysis: $\Theta(n)$

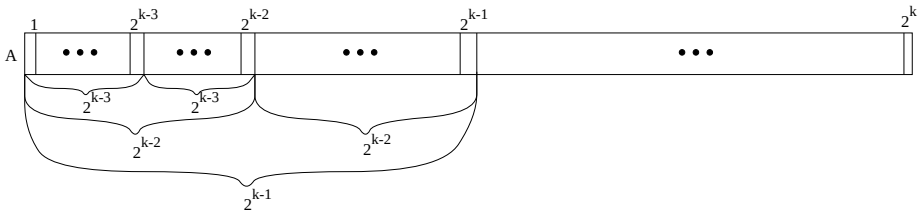
This is a **bad design** since we never took into account the fact that the array is initially sorted.

A divide-and-conquer approach

Assume that $n = 2^k$ (or, $\log n = k$) and that A is sorted in increasing order.

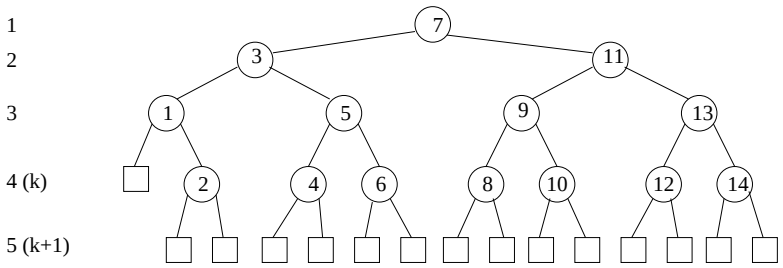
Binary_search(A, i, j, key) /* Searches for key in $A[i..j]$ */

if $i > j$ **return**(0) **and exit**
if $i \leq j$ **do**
  $middle = \lfloor (i + j)/2 \rfloor$
  if $key = A[middle]$ **then return**$(middle)$ **and exit**
  else if $key < A[middle]$ **then** *Binary_search*$(A, i, middle - 1, key)$
  else *Binary_search*$(A, middle + 1, j, key)$



Theorem If n is in the range $[2^{k-1}, 2^k)$, *Binary_search* makes at most k element comparisons for a successful search and either $k-1$ or k comparisons for an unsuccessful search.

Proof Consider the binary decision tree for *Binary_search*.



- A successful search ends at a ◯ node.
- An unsuccessful search ends at a □ node.
- ◯ nodes exist at levels $1..k$.
- □ nodes exist at levels k and $k+1$.

- The time for a successful search is $O(\log n)$ while the time for an unsuccessful search is $\Theta(\log n)$.

In conclusion

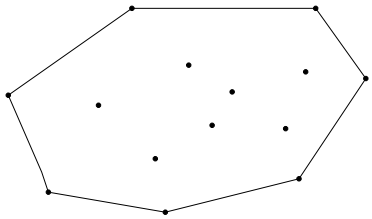
	Successful searches	**Unsuccessful searches**
Best case	$\Theta(1)$	$\Theta(\log n)$
Worst case	$\Theta(\log n)$	$\Theta(\log n)$
Average case	$\Theta(\log n)$	$\Theta(\log n)$

Question Will it help if we split the list into 3 parts (*ternary search*)?

Finding the convex hull

- The *convex hull* of a set of n points in the plane is a sequence of points from the set which defines a convex figure enclosing all points.

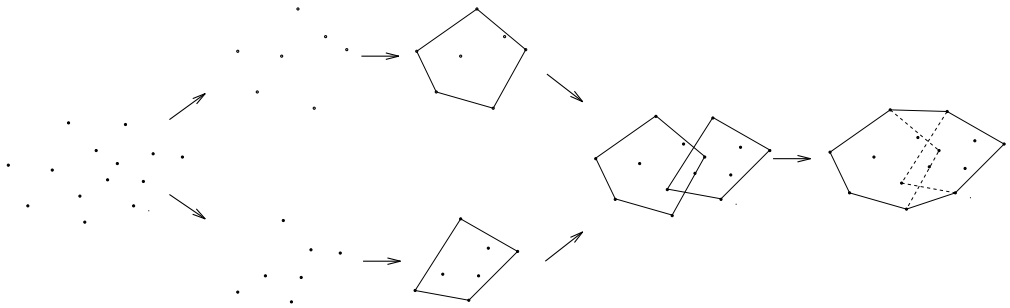
Example



- **A divide-and-conquer approach**

• Split the points into 2 sets, each containing $n/2$ points.
• Recursively compute the two convex hulls.
• Combine the 2 hulls.

Example



Problem It is not easy to combine the solutions.

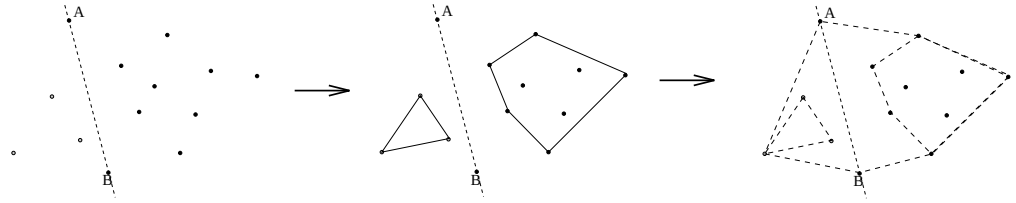
Alternative Split the points differently.

Property The points with the maximum and minimum y-coordinates belong in the convex hull. Let these be points A and B, respectively.

- Split the points into 2 sets based on their location with respect to line AB.

  Solve the smallest problems.

  Combine them.



Time analysis

Average analysis: $O(n \log n)$

Worst case analysis: $O(n^2)$

Reading Material

§1.3 pp. 11–15.

§4.3 pp. 66–69 *Kingston's* book, (Convex hull material).

"Binary Search", "A^n" are not in the book.

Suggested Reading

§4.3 "The master method" pp. 61–64.

§4.4 "Proof of the master theorem" pp. 64–72.

Matrix multiplication

Input: Two $n \times n$ matrices A and B.

Output: Matrix C such that $C = A \times B$.

- **The conventional method**

$$C[i,j] = \sum_{1 \le k \le n} A[i,k] \cdot B[k,j]$$

```
/* Conventional matrix multiplication. C = A × B */

for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C[i,j] := C[i,j] + A[i,k] · B[k,j]
```

Analysis: $\Theta(n^3)$

- **A divide and conquer approach**

Assume that $n = 2^k$ (or, $k = \log n$).

If $A \times B$ is

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} \quad (1)$$

then

$$\begin{aligned} C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} \\ C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ C_{2,1} &= A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} \\ C_{2,2} &= A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{aligned} \quad (2)$$

Time analysis

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(\frac{n}{2}) + cn^2 & n > 2 \end{cases} \quad (3)$$

Is this method better?

$$T(n) = 8T(\tfrac{n}{2}) + cn^2 \;=\; 8\left(8T(\tfrac{n}{4}) + c(\tfrac{n}{2})^2\right) + cn^2 \;=$$
$$= 8^2 T(\tfrac{n}{4}) + cn^2\left(1 + 8(\tfrac{1}{2})^2\right) \;=$$
$$= 8^2\left(8T(\tfrac{n}{8}) + c(\tfrac{n}{4})^2\right) + cn^2\left(1 + 8(\tfrac{1}{2})^2\right) \;=$$
$$= 8^3 T(\tfrac{n}{4}) + cn^2\left(1 + 8(\tfrac{1}{2})^2 + 8^2(\tfrac{1}{2^2})^2\right) \;=$$
$$= \cdots \;=$$
$$= 8^k T(1) + cn^2\left(1 + 8(\tfrac{1}{2})^2 + 8^2(\tfrac{1}{2^2})^2 + \cdots + 8^{k-1}(\tfrac{1}{2^{k-1}})^2\right)$$
$$= \Theta(n^3)$$

- $8^k \;=\; 8^{\log_2 n} \;=\; n^{\log_2 8} \;=\; n^3$
- $cn^2\left(1 + 8(\tfrac{1}{2})^2 + 8^2(\tfrac{1}{2^2})^2 + \cdots + 8^{k-1}(\tfrac{1}{2^{k-1}})^2\right) \;=\; cn^2\sum_{j=0}^{\log_2 n - 1} 8^j(\tfrac{1}{2^j})^2 \;=$
$= cn^2\sum_{j=0}^{\log_2 n - 1} 2^j \le cn^3$

- If instead of (**??**) we had:

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(\tfrac{n}{2}) + cn^2 & n > 2 \end{cases} \qquad (4)$$

$$T(n) = 7T(\tfrac{n}{2}) + cn^2 \;=\; 7\left(7T(\tfrac{n}{4}) + c(\tfrac{n}{2})^2\right) + cn^2 \;=$$
$$= 7^2 T(\tfrac{n}{4}) + cn^2\left(1 + 7(\tfrac{1}{2})^2\right) \;=$$
$$= 7^2\left(7T(\tfrac{n}{8}) + c(\tfrac{n}{4})^2\right) + cn^2\left(1 + 7(\tfrac{1}{2})^2\right) \;=$$
$$= 7^3 T(\tfrac{n}{8}) + cn^2\left(1 + 7(\tfrac{1}{2})^2 + 7^2(\tfrac{1}{2^2})^2\right) \;=$$
$$= \cdots \;=$$
$$= 7^k T(1) + cn^2\left(1 + 7(\tfrac{1}{2})^2 + 7^2(\tfrac{1}{2^2})^2 + \cdots + 7^{k-1}(\tfrac{1}{2^{k-1}})^2\right)$$
$$= \Theta(n^{\log_2 7}) \;=\; \Theta(n^{2.71})$$

- $cn^2\left(1 + 7(\tfrac{1}{2})^2 + 7^2(\tfrac{1}{2^2})^2 + \cdots + 7^{k-1}(\tfrac{1}{2^{k-1}})^2\right) \;=\; cn^2\sum_{j=0}^{\log_2 n - 1} 7^j(\tfrac{1}{2^j})^2 \;=$
$= cn^2\sum_{j=0}^{\log_2 n - 1}(\tfrac{7}{4})^j \;=\; cn^2 c'(\tfrac{7}{4})^{\log_2 n} = cc' n^{\log_2 7}$

Strassen's matrix multiplication

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$P = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$
$Q = (A_{2,1} + A_{2,2}) \times B_{1,1}$
$R = A_{1,1} \times (B_{1,2} - B_{2,2})$
$S = A_{2,2} \times (B_{2,1} - B_{1,1})$
$T = (A_{1,1} + A_{1,2}) \times B_{2,2}$
$U = (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2})$
$V = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$

$C_{1,1} = P + S - T + V$
$C_{1,2} = R + T$
$C_{2,1} = Q + S$
$C_{2,2} = P + R - Q + U$

- 7 multiplications and 18 additions/subtractions.
- $T(n) = \Theta(n^{2.71})$ [From the solution of (**??**).]

Some results ...

- We **cannot** multiply two 2×2 matrices using less that 7 multiplications.
- We can do matrix multiplication in $O(n^{2.376})$ time.

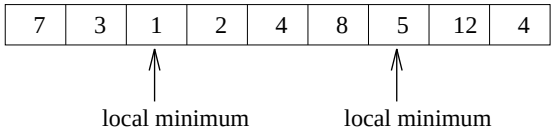
Which method to use?

- For very small matrices use the "traditional" method.
- For moderate size matrices use Winograd's method.
- For very large values use Strassen's method.

Computing a local minimum

Definition Given an array $A[0..n+1]$ we say that $A[i],\ 1 \leq i \leq n$, is a local minimum if $A[i-1] \geq A[i] \leq A[i+1]$

Example



Input: An array $A[0..n+1]$ such that $A[0] = A[n+1] = +\infty$

Output: The position of a local minimum.

- **A trivial algorithm**

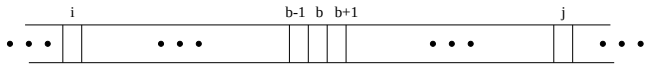
> Traverse the array from left to right and for every 3 consecutive elements check if $element\_1 \geq element\_2 \leq element\_3$.

Analysis $\Theta(n)$

A divide-and-conquer approach

Fact If in the sub-array $A[i..j],\ j - i \geq 2$ it holds that $A[i] \geq A[i+1]$ and $A[j-1] \leq A[j]$ then in $A[i..j]$ there exists a local minimum.

Because $A[0] = A[n+1] = +\infty$ we must have a local minimum in A.



> ...
> **if** $A[b] \geq A[b+1] \implies \exists$ a local minimum in $A[b..j]$
> **else if** $A[b] \geq A[b-1] \implies \exists$ a local minimum in $A[i..b]$
> **else** $A[b]$ is a local minimum
> ...

Analysis $O(\log n)$

Reading Material

§31.1 "Operations on matrices" pp. 733–735.

§31.2 "Strassen's algorithm..." pp. 739–745
(the notation is different from that used in lectures).

"Local minimum" is not in the book.

Suggested Reading

§4.3 "The master method" pp. 61–64.

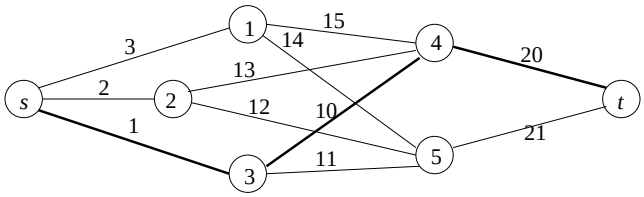
§4.4 "Proof of the master theorem" pp. 64–72.

The Greedy Method

Optimization problems

- In such problems there can be many possible solutions.
- Each solution has a *value*. We wish to find a solution with the *optimal* (maximum or minimum) value.

Example Find the shortest path from s to t.



Path	**Cost**
$s-2-4-t$	$2+13+20=35$
$s-1-5-2-4-t$	$3+14+12+13+20=62$
$s-3-4-t$	$1+10+20=31$

A greedy algorithm always makes the choice which looks best at the moment.

It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

NOTE When we claim that we have a greedy algorithm that computes an optimal solution to an optimization problem **we must give a proof for our claim**.

We will study:

- Activity selection problem.
- Huffman codes.
- Prim's algorithm for minimum spanning trees.
- Dijkstra's algorithm for single source shortest paths.

An activity selection problem

Input: A set $S = \{1, 2, \cdots, n\}$ of n proposed activities all of which wish to use a resource. For each activity i, its *start time* s_i and its *finishing time* f_i, $f_i > s_i$.

Output: A maximum size set of mutually compatible activities.

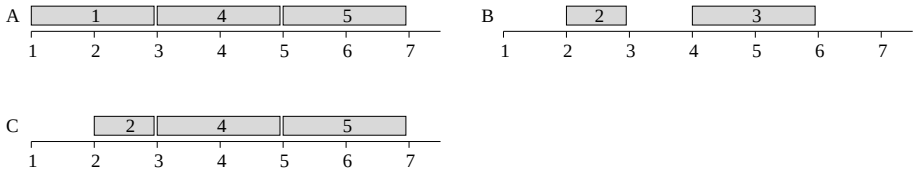
- Activities i and j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Example

Resource: A lecture room.

Activities: Lect1 [1..3)pm Lect2 [2..3)pm Lect3 [4..6)pm
Lect4 [3..5)pm Lect5 [5..7)pm

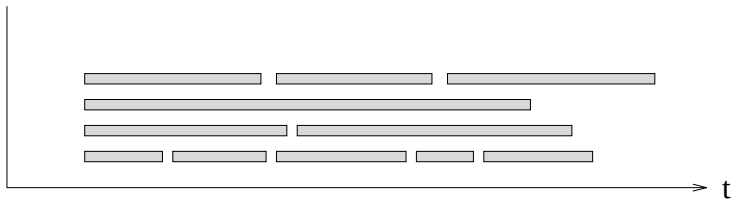
Possible schedules



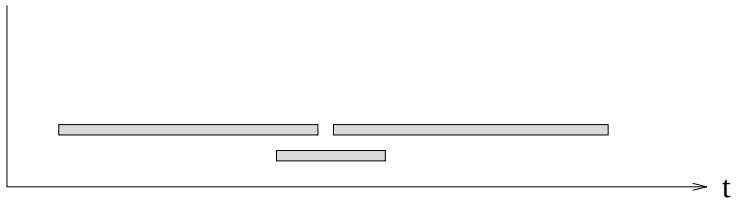
A first greedy approach ...

> Select the activity of least duration from those that are compatible with previously selected activities.

- It works for the following instance:



- It fails for the instance:

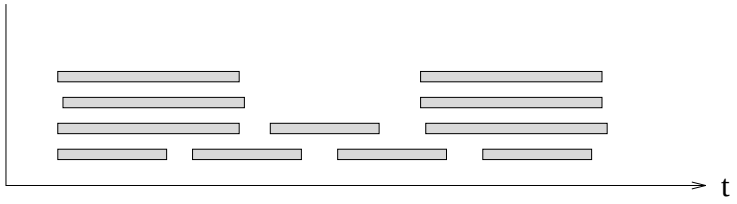


A second greedy approach ...

> Select the activity which overlaps the fewest activities out of those that are compatible with previously selected ones.

- It works for the following instance:

- It fails for the instance:



A greedy solution

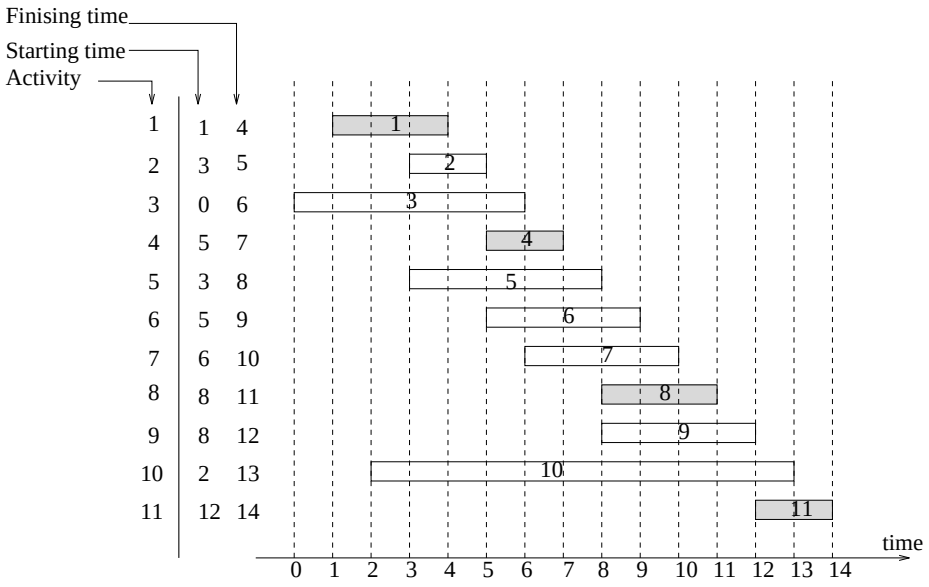
Assume that the activities are in order of increasing finishing times, i.e., $f_1 \leq f_2 \leq \cdots \leq f_n$. (If they are not, we sort them in $O(n \log n)$ time.)

> Always select the activity with the earliest finishing time that can be legally scheduled.

```
Greedy_activity_selection(s, f)
n = length(s)
A = {1}
j = 1
for i = 2 to n do
    if s_i ≥ f_j then
        A = A ∪ {i}
        j = i
return A
```

- Our greedy choice is the one that maximizes the amount of unscheduled remaining time.
- Set A collects the selected activities.
- j specifies the most resent addition to A.

Time complexity: $\Theta(n)$



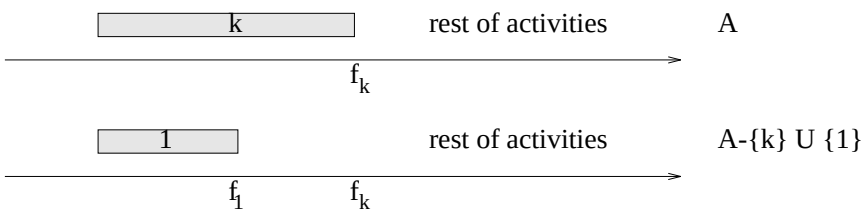
Proving the greedy method correct

Theorem 1 Algorithm *Greedy_activity_selection* produces solutions of maximum size for the activity selection problem.

Lemma 1 There is an optimal solution that begins with a greedy choice, i.e., activity 1.

Proof Let $A \subseteq S$ be an optimal solution of the activity selection problem.

- Order the activities of A by finishing time.
- Let k be the first activity.
- If $k = 1$ we are done.
- If $k \neq 1$ then $A - \{k\} \cup \{1\}$ is also an optimal solution ($f_1 \leq f_k$ and $\{1\} \notin A$).



□

Lemma 2 If A is an optimal solution of the original problem S, then $A' = A - \{1\}$ is an optimal solution of the activity selection problem $S' = \{i \in S : s_i \geq f_1\}$.

Proof

- Let A contain m activities. $\Longrightarrow$ A' contains $m-1$ activities.
- Assume that A' is not optimal, i.e.,
  there exists a set B of activities that is a valid schedule that contains more activities than A', $|B| > m-1$.

Then, $B \cup \{1\}$ is a valid schedule for S and $|B \cup \{1\}| > m$. Thus, A is not optimal. This is a clear contradiction since we assumed it is.

□

Proof of Theorem 1 (By induction)

- Basis: Lemma 1
- Induction step: Lemma 2

After each greedy choice is made, we are left with an optimization problem of the same form as the original problem.

By induction on the number of choices made, making the greedy choice at every step produces the optimal solution.

□

Reading Material

§17.1 "An activity-selection problem" pp. 329–333.

§17.2 "Elements of the greedy strategy" pp. 333-334
(up to "Greedy versus dynamic programming").

Suggested Exercises

17.1-2, 17.1-3.

Huffman codes

We consider the problem of designing a binary character code wherein each character is represented by a unique binary string.

- Fixed-length codeword.
- Variable-length codeword.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- We need 300,000 bits to encode a file consisting of 100,000 characters (a-f only) if coded by the fixed-length codewords.
- If the variable-length codewords are used we need:

$$1000 \cdot (45 \cdot 1 \ + \ 13 \cdot 3 \ + \ 12 \cdot 3 \ + \ 16 \cdot 3 \ + \ 9 \cdot 4 \ + \ 5 \cdot 4) = 224,000.$$

About 25% savings of space.

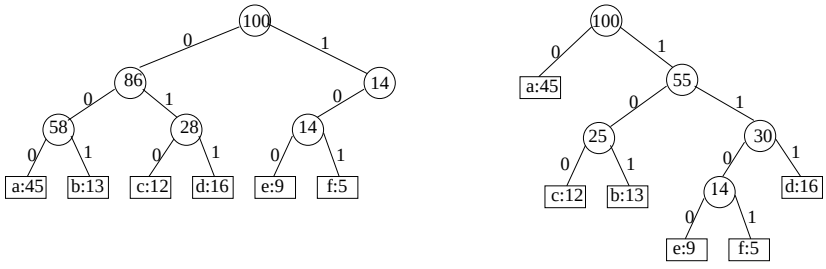
Prefix-free codes: Codes in which no codeword is also a prefix of another codeword.

- Prefix-free codes simplify encoding and decoding.

Encoding: "abc" : 000 001 010 (fixed-length code)
: 0 101 100 (variable-length code)

Decoding: We use binary trees.

Fixed-length (non optimal) Variable-length



- An optimal code for a file is always represented by a full binary tree in which every non-leaf node has two children.
- A tree of an optimal prefix code for a set C of characters has exactly $|C|$ leaves and $|C| - 1$ internal nodes.

Consider a tree T corresponding to a prefix-free code for alphabet C. Also consider a file over C and let:

- $f(c)$ denote the frequency of c in the file, and
- $d_T(c)$ denote the depth of c's leaf in the tree T.

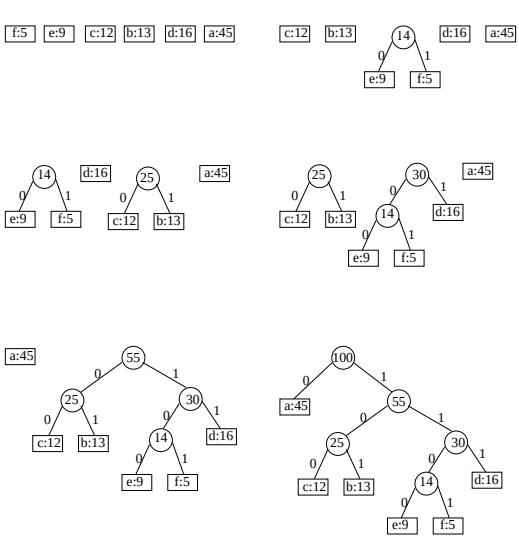
The number of bits required to encode a file is:

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

$B(T)$ is defined to be the cost of tree T.

Problem Given an alphabet C and a set of frequencies for the characters in C find a coding scheme (tree) of minimum cost.

$Huffman(C)$
/* Q is a priority queue */
$n = |C|$
$Q = C$
for $i = 1$ **to** $n - 1$ **do**
    $z = allocate\_node()$
    $x = extract\_min(Q)$
    $y = extract\_min(Q)$
    $left[z] = x$
    $right[z] = y$
    $f[z] = f[x] + f[y]$
    $insert(Q, z)$
return $extract\_min(Q)$



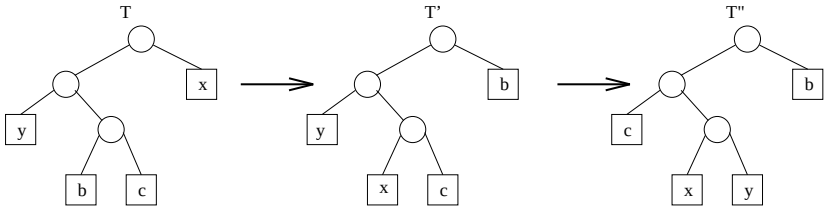
Time complexity $O(n \log n)$ (if Q is implemented as a binary heap).

Correctness of Huffman's algorithm

Theorem 1 Algorithm $Huffman$ produces an optimal prefix-free code.

Lemma 1 Let x and y be two characters in alphabet C having the lowest frequencies. Then, there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof Take a tree T representing an arbitrary optimal prefix code. Modify it to satisfy the lemma.

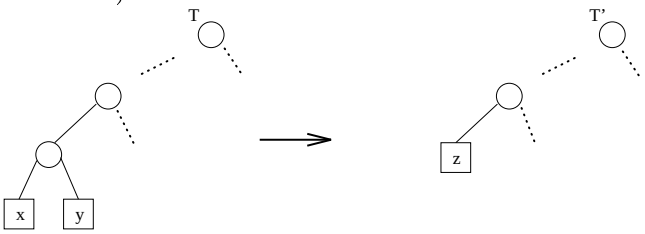


- Let b, c be two characters that are sibling leaves of maximum depth in T.
- Assume that $f(b) \leq f(c)$ and $f(x) \leq f(y)$.
- Exchange b with x. $(\Longrightarrow B(T) \geq B(T'))$
- Exchange c with y. $(\Longrightarrow B(T') \geq B(T''))$

□

Lemma 2 Let T be a full binary tree representing an optimal prefix code over an alphabet C. Consider any 2 characters x and y that appear as sibling leaves in T, and let z be their parent. Then, considering z as a character with frequency $f(z) = f(x) + f(y)$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for alphabet $C' = C - \{x, y\} \cup \{z\}$.

Proof (by contradiction)



- $B(T) = B(T') + f(x) + f(y)$.
- Assume that T' is not optimal. $\Longrightarrow$ $\exists$ T" for $C' : B(T") < B(T')$.
- We can obtain from T" a prefix code for C with cost $B(T") + f(x) + f(y) < B(T)$. This is a clear contradiction since we assumed that $B(T)$ is optimal.

□

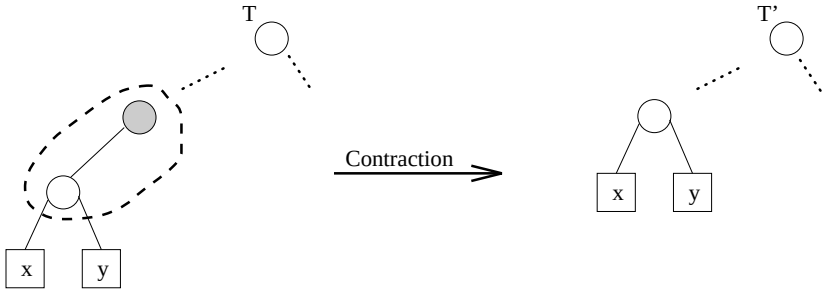
Proof of Theorem 1 Follows from Lemmata 1 and 2.

□

Exercise 17.3-1 Prove that a binary tree T which is not full cannot represent an optimal prefix code.

Proof (by contradiction)

- Assume that we have a binary tree which is not full and it represents an optimal prefix code.
- There is an internal node of T which contains only 1 child.
- By contracting that node with its child we get a binary tree T' that represents a code for the same file and is of less cost than T. This is a clear contradiction since T was assumed to be of minimum cost.



□

Reading Material

§17.3 "Huffman codes" pp. 337–344.

Suggested Exercises

17.3-2, 17.3-4, 17.3-6.

Single-source shortest path problems

Motivation

- They have a greedy solution (Dijkstra's algorithm).
- They have many real-life applications.

Examples

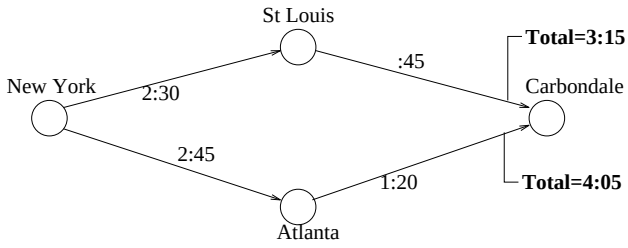
- Given a road map of North America (or of any other place) find the distance from New York to all other cities on the map.

  Represent cities as vertices,
  roads as edges, and
  distances between cities as edge weights.

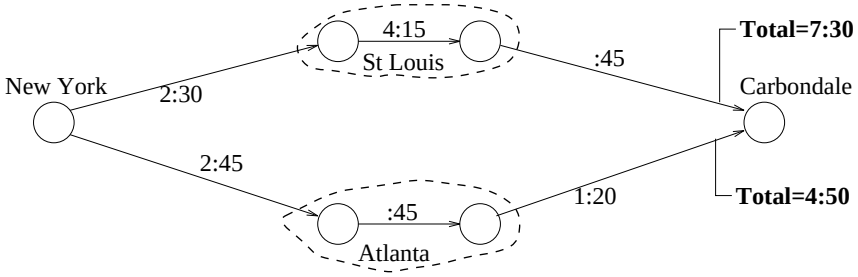
  Then, find the shortest paths to all vertices (representing cities) from the vertex that represents New York.

- Given the schedules of all airline flights and their flying-times find what is the fastest way to go from New York to Carbondale, Illinois.

Solution
Represent cities as vertices, flights as edges, and flying-times as edge weights.



- What if we want to take waiting time between flights into account?



Single-source shortest path problems

Input:
- A weighted, undirected graph $G = (V, E)$ with weight function $w : E \to \mathcal{R}$ mapping edges to real valued weights.
- A source node s.

Output: The shortest paths from s to any other vertex in G.

Definitions

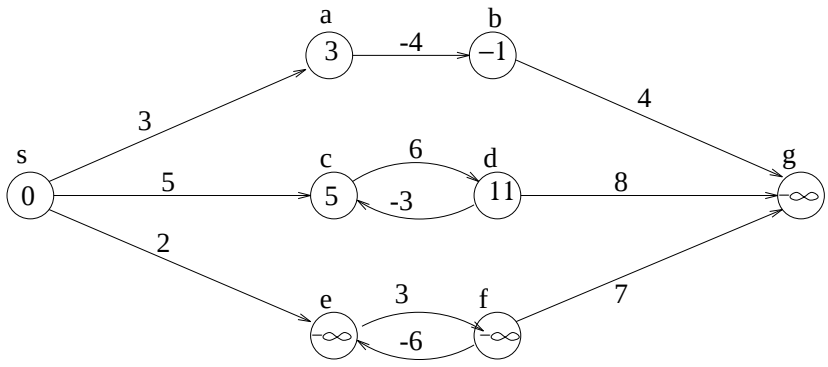
The *weight of path* $p = (v_0, v_1, \cdots, v_k)$ is the sum of the weights of the constituent edges

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

The *shortest path weight* from u to v is defined by:

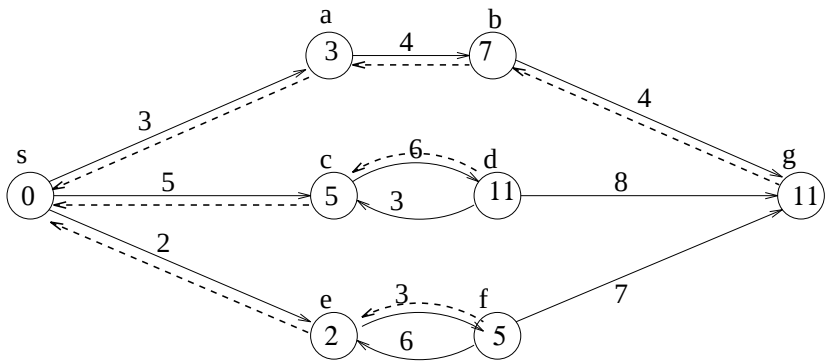
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from u to v} \\ +\infty & \text{otherwise} \end{cases}$$

Negative weight edges



Cycles that contain negative weight edges:
- cost of cycle ≥ 0.
- cost of cycle < 0.

How to represent the shortest paths

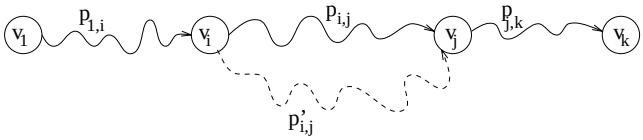


- In the following we concentrate only on finding the costs of the shortest paths.
- The actual paths can be constructed with the same algorithm we used to construct paths from a breadth first search tree.

Optimal substructure of a shortest path

Lemma (*Sub-paths of shortest paths are shortest paths*)
Assume a weighted directed graph $G = (V, E)$ with weight function w and let $p = (v_1, v_2, \cdots, v_k)$ be a shortest path from v_1 to v_k. For $1 \leq i \leq j \leq k$ let $p_{i,j} = (v_i, \cdots, v_j)$ be the sub-path of p from v_i to v_j. Then $p_{i,j}$ is a shortest path.

Proof (By contradiction)



- $w(p) = w(p_{1,i}) + w(p_{i,j}) + w(p_{j,k})$ and $w(p)$ is minimum.
- Assume $p_{i,j}$ is not minimum. Let $p'_{i,j}$ be a minimum path.
- $w(p'_{i,j}) < w(p_{i,j})$.
- Then p is mot a minimum path. A clear contradiction.
  (The path p' : $p_{1,i} - p'_{i,j} - p_{j,k}$ has smaller weight than p.)

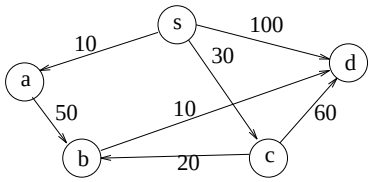
□

Dijkstra's algorithm

- It solves the single-source shortest paths problem on weighted graph $G = (V, E)$ with nonnegative edge weights.
- S: set of vertices whose final shortest path weights from the source s have been determined.
- $d[v]$: an upper bound on the weight of a shortest path from source s to v.
- Q: a priority queue containing vertices in $V - S$ sorted by their d values.

$Dijkstra(G, w, s)$
for each vertex $v \in V$ **do**
    $d[v] = +\infty$
$d[s] = 0$
$Q = V$
while $Q \neq \emptyset$ **do**
    $u = Extract\_min(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in adj[u]$ **and** $v \in V - S$ **do**
        if $d[v] > d[u] + w[u, v]$ **then**
            $d[v] = d[u] + w[u, v]$

Example



Iteration	S	u	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$
initial	$\emptyset$	-	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	$\{s\}$	s	0	10	$+\infty$	30	100
2	$\{s, a\}$	a	0	10	60	30	100
3	$\{s, a, c\}$	c	0	10	50	30	90
4	$\{s, a, b, c\}$	b	0	10	50	30	60
5	$\{s, a, b, c, d\}$	d	0	10	50	30	60

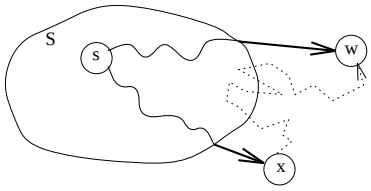
Time analysis

- If Q is maintained as an array: $O(n^2)$.
- If Q is implemented as a binary heap: $O(E \log V)$.
  (This is better for sparse graphs.)

Correctness of Dijkstra's algorithm

- $d[v]$, for any vertex v, contains the cost of a path that passes only through vertices in S.
- Whenever we insert a new vertex w in S, $d[w]$ contains the cost of the shortest path from source s to w.

Proof



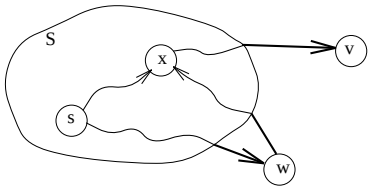
Consider a hypothetical shorter path from s to w that leaves S to go to x and then it reaches w.

If this path is shorter, then x had to be selected by the algorithm instead of w. Thus we didn't execute the algorithm correctly.
(Note how crucial is the fact that we do not have negative weights.)

□

- $d[v]$ is **always** the cost of the shortest path from s to v that passes from vertices in S.

Proof



When we add w in S we have to make sure that if there is a shortest path to v that passes through w, we update $d[v]$ correctly.

• If the path goes through the old S to w and then directly to v, we are updating it correctly.

• What if the path goes to w, then to $x \in S$ and then to v? (The algorithm does nothing to cover this case.)

This case cannot occur!

Since x was placed in S before w, the shortest of all paths from the source s to x runs through the old S alone. Thus, the path to x through w is no shorter than the path directly trough S. So, we cannot improve $d[v]$.

□

Reading Material

§25 "Introduction" pp. 514–518.

Lemma 25.1 pp. 519.

§25.2 "Dijkstra's algorithm".

Additional Reading (optional)

§25.1 "Shortest paths and relaxation".

Suggested Exercises

25.2-1, **25.2-2**.

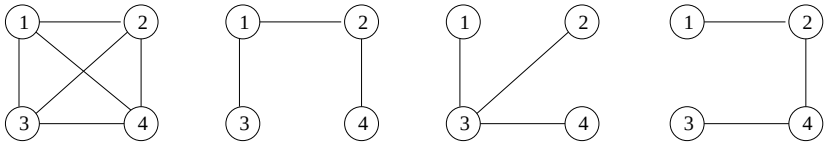
Minimum spanning trees

Motivation

- Nice greedy solutions.
- They have many real-life applications.

Definition Let $G = (V, E)$ be an undirected connected graph. A subgraph $T = (V, E')$ of G is a *spanning tree* of G **iff** T is a tree.

Example A complete graph of 4 nodes together with 3 of its spanning trees.



- If a weight is associated with each edge of the graph, we are interested in finding a *minimum- weight spanning tree.*

- The total weight of a spanning tree T of graph G is:

$$w(T) = \sum_{e \in T} w(e)$$

where e is an edge and $w(e)$ is its weight.

Application In the design of electronic circuits we often want to make n pins electrically equivalent by wiring them. We want to find an "efficient" wiring, i.e., one of minimum cost. The cost to connect any two pins is given.

Solution

- Represent each pin by a vertex.
- Represent each possible connection by an edge of appropriate weight.

Then, find a minimum spanning tree.

Growing a spanning tree

Input: A connected undirected graph $G = (V, E)$ with a weight function $w : E \to \mathcal{R}$.

Output: A minimum spanning tree for G.

- **A generic algorithm**

  $Generic\_MST(G, w)$
  $A = \emptyset$
  while A does not form a spanning tree **do**
  Find an edge (u, v) that is safe for A.
  $A = A \cup \{(u, v)\}$
  return A

  Algorithm Invariant A is always a subset of some minimum spanning tree.

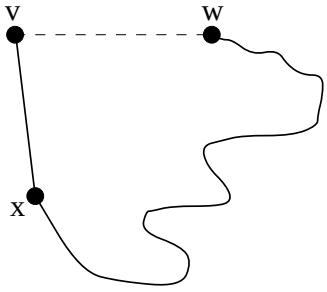
- An edge (u, v) is a *safe edge* for A if it can be safely added to A without violating the invariant.

  If A does not form a (minimum) spanning tree, we can always find a safe edge for A.

How to find a safe edge for A

Lemma Consider any vertex v. Let (v, w) be the incident edge on v of minimum weight. Then, there exists a minimum spanning tree which contains (v, w).

Proof (By contradiction)



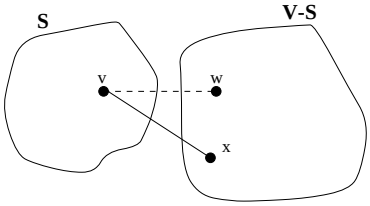
- Assume that such a tree does not exist.
- Consider any minimum spanning tree T.
  Let (v, x) be the edge of T incident to v.
- By adding (v, w) in T we introduce a cycle. We can break it by removing (v, x). This gives us the new spanning tree $T' = T - \{(v, x)\} \cup \{(v, w)\}$.
- $w(T') \leq w(T)$ since $w((v, w)) \leq w((v, x))$. Thus, T' contains (v, w) and does not have weight greater than the minimum spanning tree T. A clear contradiction.

□

$\Rightarrow$ By the above lemma **we know how to find the first safe edge**.

Lemma Let A be a subset of a minimum spanning tree and S be the set of vertices that are adjacent to edges in A. Let (v, w) be an edge of minimum weight such that $v \in S$ and $w \in V - S$. Then, (v, w) is a safe edge for A.

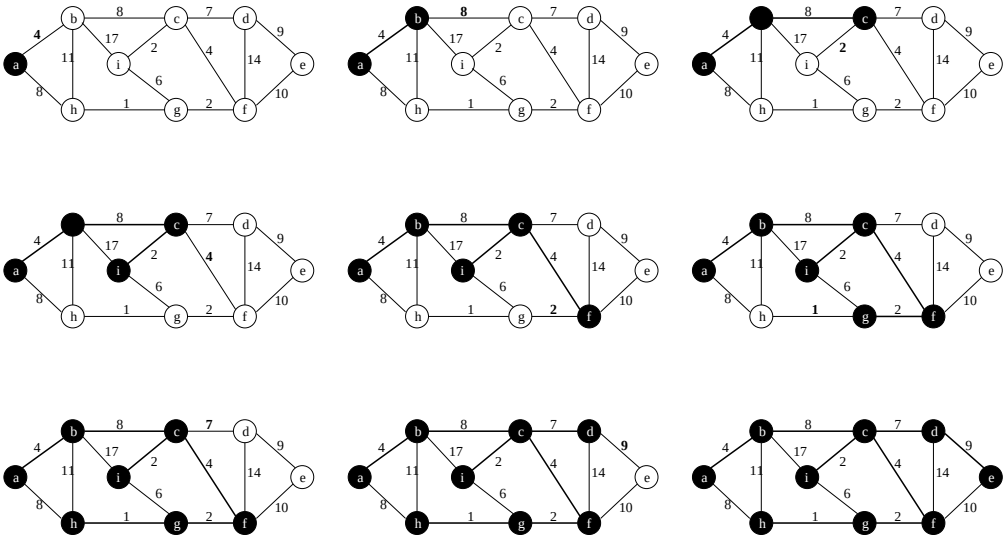
Proof (By contradiction)



- Let T be a minimum spanning tree with edge (v, x) connecting S and $V - S$.
- $T' = T - \{(v, x)\} \cup \{(v, w)\}$ is a spanning tree of weight smaller or equal to that of T and also contains (v, w). A clear contradiction.

□

Example



Prim's algorithm

- During execution of the algorithm all vertices not currently in the tree reside in a priority queue Q based on a key field.
- For each vertex $v \in V, key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree.
- $\pi[v]$ denotes the "parent" of v in the tree.

```
MST_prim(G, w, r)    /* r is the root of the spanning tree */
for each u ∈ V do    key[u] = +∞
key[r] = 0
π[r] = nil
Q = V
while Q ≠ ∅ do
    u = extract_min(Q)
    for each v ∈ adjacent_list(u) do
        if v ∈ Q and w[u, v] < key[v] then
            π[v] = u
            key[v] = w(u, v)
```

Analysis: $O(V \log V + E \log V) = O(E \log V)$ if the queue is implemented as a heap.

$O(V \log V + E)$ if the queue is implemented as a Fibonacci heap.

Conclusion

- A lot of optimization problems suggest a trivial algorithm. Usually it is **not** efficient.

  Examples

  - Minimum Spanning Trees

    Construct all Spanning Trees.
    Find the one of minimum weight.

    Analysis: $O\left(C\left(|E|, n-1\right) \cdot n\right)$

  - Single-source shortest paths (nonnegative weights)

    For each vertex v, construct all paths from the source s to v.
    Find the one of minimum length.

    Analysis: $O\left(C\left(|E|, n-1\right) \cdot n \cdot n\right)$

- Finding properties of the optimal solution leads to efficient algorithms.

Reading Material

§24 "Minimum spanning trees" pp. 498–499.

§24.1 "Growing a minimum spanning tree" pp. 499–504.
(The presentation in the book is different from that made in class.)

§24.2 "Prim's algorithm" (only) pp. 505–511.

Suggested Exercises

24.1-1, 24.1-5, 24.2-2.

Dynamic Programming

Characteristics

- Optimal substructure.
- Overlapping subproblems.

We will examine:

- Fibonacci numbers.
- Matrix-chain multiplication.
- Longest common subsequence.
- All-pairs shortest paths.

Fibonacci numbers

The n^{th} *Fibonacci number* $F(n)$ is defined by the recurrence relation:

$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

$F(2) = 1,\ F(3) = 2,\ F(4) = 3,\ F(5) = 5,\ F(6) = 8,\ F(7) = 13,\ F(8) = 21,\ \cdots$

An obvious algorithm to compute $F(n),\ n \geq 1$

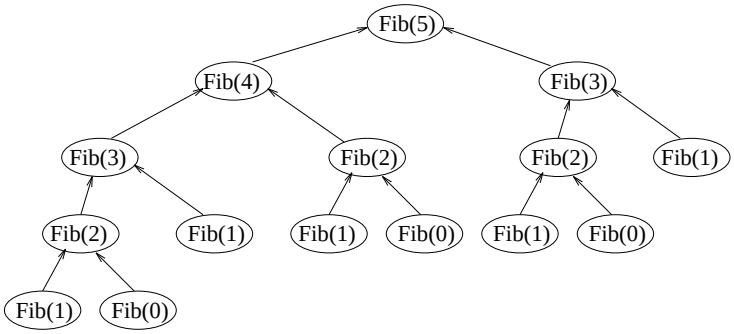
Fibinacci(n)
if $n < 2$ **then return**(n)
else return$(Fibonacci(n-1) + Fibonacci(n-2))$

Analysis $T(n) = \begin{cases} 1 & n < 2 \\ T(n-1) + T(n-2) & n \geq 2 \end{cases}$

$$\Longrightarrow \quad T(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} \quad \Longrightarrow \quad T(n) = \Theta(\phi^n)$$

$\phi = \frac{1+\sqrt{5}}{2} = 1.61803, \quad \hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803$

- For $n = 5$, the recursion tree is:



We compute: $F(3)$ 2 times,
$F(2)$ 3 times,
$F(1)$ 5 times,
$F(0)$ 3 times.

This is redundant work!

A better solution

- Compute $F(n)$ in a bottom-up manner.

$Non\_Recursive\_Fibonacci(n)$
/* $A[1..n]$ is an array in which we store $F(1) \cdots F(n)$ */
$A[0] = 0$
$A[1] = 1$
for $i = 2$ **to** n **do**
    $A[i] = A[i-1] + A[i-2]$
return $A[n]$

Analysis: $\Theta(n)$

Matrix-chain multiplication

We are given a sequence (chain) $< A_1,\ A_2,\ \cdots,\ A_n >$ of n matrices and we want to compute the product $A_1 \times A_2 \times \cdots \times A_n$.

- $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$, i.e., matrix multiplication is associative.

  $\Longrightarrow$ All parenthesizations will give the same result.

Example For $n = 4$, there are 5 distinct parenthesizations:

$$(A_1 \times (A_2 \times (A_3 \times A_4)))$$

$$(A_1 \times ((A_2 \times A_3) \times A_4))$$

$$((A_1 \times A_2) \times (A_3 \times A_4))$$

$$((A_1 \times (A_2 \times A_3)) \times A_4)$$

$$(((A_1 \times A_2) \times A_3) \times A_4)$$

Definition A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products surrounded by parentheses.

Note We can generalize the definition to any kind of binary operator and operands.

Exercise Show that a full parenthesization of an n-element expression has exactly $n - 1$ pairs of parentheses. (A binary operator is assumed.)

```
Matrix_multiply(A, B)
if columns(A) ≠ rows(B)
then error "incompatible dimension"
else for i = 1 to rows(A) do
        for j = 1 to columns(B) do
            C[i, j] = 0
            for k = 1 to columns(A) do
                C[i, j] = C[i, j] + A[i, k] · B[k, j]
```

- If A is a $p \times q$ matrix and B is a $q \times r$ matrix then C is a $q \times r$ matrix.
- The algorithm performs exactly pqr multiplications.

Example $A_1 = 10 \times 100, \quad A_2 = 100 \times 5, \quad A_3 = 5 \times 50$

$((A_1 \times A_2) \times A_3)$ needs $(10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) = 7,500$ multiplications.
$(A_1 \times (A_2 \times A_3))$ needs $(10 \cdot 100 \cdot 50) + (100 \cdot 5 \cdot 50) = 75,000$ multiplications.

$\Longrightarrow$ The order in which we perform the multiplications is very important!

Question What is the number of possible parenthesizations?

Answer Denote the answer by $P(n)$.

$$\underbrace{(A_1 \times A_2 \times \cdots \times A_k)}_{P(k)} \times \underbrace{(A_{k+1} \times \cdots \times A_n)}_{P(n-k)}$$

We can write the recurrence relation:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution is the sequence of Catalan numbers $P(n) = C(n-1)$ where,

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$$

(Exponential on n!)

The structure of the optimal parenthesization

- $A_{i..j}$ denotes the matrix that results from evaluating the product $A_i \times \cdots \times A_j$.

Observation An optimal parenthesization of $A_1 \times \cdots \times A_n$ splits the product between A_k and A_{k+1} for some integer $k,\ 1 \leq k < n$.

$$((A_1 \times A_2 \times \cdots \times A_k) \ \times \ (A_{k+1} \times \cdots \times A_n))$$

Theorem The parenthesization of the *prefix* subchain $A_1 \times \cdots \times A_k$ within an optimal parenthesization of $A_1 \times \cdots \times A_n$ must be an optimal parenthesization of $A_1 \times \cdots \times A_k$.

Proof If we can find a better parenthesization for $A_1 \times \cdots \times A_k$ then, we can construct a better than the optimal parenthesization for $A_1 \times \cdots \times A_n$. This is a clear contradiction.

□

Note We can obtain a similar theorem for the *postfix* chain $A_{k+1} \times \cdots \times A_n$.

A recursive solution

- $m[i,j]$ denotes the minimum number of multiplications needed to compute $A_{i..j}$.
- We want to find the parenthesization which takes $m[1,n]$ multiplications to compute$A_{1..n}$.
- We assume that A_i is of dimensions $[p_{i-1} \times p_i]$.

We can compute $m[1,n]$ by using the recurrence relation:

$$m[i,j] = \begin{cases} 0 & \text{if } \ i = j \\ \min_{i \leq k \leq j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j & \text{if } \ i < j \end{cases}$$

- A recursive algorithm based on the above recurrence relation will take **at least** exponential time!

A dynamic programming solution

Observation We have to solve $\Theta(n^2)$ subproblems, one for each choice of i and j satisfying $1 \leq i \leq j \leq n$.

Idea: We will compute $m[1,n]$ in a bottom-up fashion. We will compute all $m[i,j]$ in order of increasing $(j-i)$.

```
Matrix_Chain_order(p)
/* p =< p_0, p_1, ..., p_n >, the dimensions of the matrices */
n = length(p) - 1
for i = 1 to n do
    m[i,i] = 0
for l = 2 to n do                /* l denotes the "length" of A_{i..j} */
    for i = 1 to n - l + 1 do
        j = i + l - 1
        m[i,j] = 0
        for k = i to j - 1 do
            q = m[i,k] + m[k+1,j] + p_{i-1} · p_k · p_j
            if q < m[i,j] then m[i,j] = q
return m
```

Analysis: $\Theta(n^3)$

Example

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

M (i \ j)	1	2	3	4	5	6
1	0	15,750	7,875	9,375	11,875	15,125
2		0	2,625	4,375	7,125	10,500
3			0	750	2,500	5,375
4				0	1,000	3,500
5					0	5,000
6						0

$m[1,2] = 30 \cdot 35 \cdot 15 = 15,750$

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 = 0 + 2,625 + 5,250 = \mathbf{7,875} \\ m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 = 15,750 + 0 + 2,250 = 18,000 \end{cases}$$

$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 4,375 + 10,500 = 14,875 \\ m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 15,750 + 750 + 4,500 = 21,000 \\ m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 = 7,875 + 0 + 1,500 = \mathbf{9,375} \end{cases}$$

Question How do we recover the actual parenthesization?

Answer By maintaining an additional matrix *split* where, $split[i,j]$ is the place in which we split $A_{i..j}$.

```
Matrix_Chain_order_1(p)
/* p =< p_0, p_1, ..., p_n >, the dimensions of the matrices */
n = length(p) - 1
for i = 1 to n do
    m[i,i] = 0
for l = 2 to n do                /* l denotes the "length" of A_{i..j} */
    for i = 1 to n - l + 1 do
        j = i + l - 1
        m[i,j] = 0
        for k = 1 to j - 1 do
            q = m[i,k] + m[k+1,j] + p_{i-1} . p_k . p_j
            if q < m[i,j] then
                m[i,j] = q
                split[i,j] = k
return m, split
```


The matrix multiplication algorithm

```
Matrix_Chain_Multiply(A, split, i, j)
/* Computes A_{i..j}. A is the set of matrices */

if j > i then  X = Matrix_Chain_Multiply(A, split, i, split[i,j])
               Y = Matrix_Chain_Multiply(A, split, split[i,j] + 1, j)
               return Matrix_Multiply(X, Y)
       else return A_i
```

- **Elements of dynamic programming**
  - Optimal substructure.
  - Overlapping subproblems.

- **Memoization** Use the recursive algorithm but store the solution of each subproblem for future use. In that way, we solve each subproblem once.

Reading Material

§16 "Dynamic programming" pp. 301–302.

§16.1 "Matrix-chain multiplication" pp. 302-309.

Suggested Reading

§16.2 "Elements of dynamic programming" pp. 309-314.

Suggested Exercises

16.1-1, 16.1-2, 16.1-4.

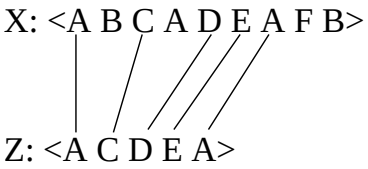
Longest Common Subsequence

Definition Given a sequence $X =< x_1, x_2, \cdots, x_m >$, another sequence $Z =< z_1, z_2, \cdots, z_k >$ is a subsequence of X if there exists a strictly increasing sequence $< i_1, i_2, \cdots, i_k >$ of indices of X such that for all $j = 1 \cdots k$ we have $x_{i_j} = z_j$.

Example $X =<$ A, B, C, A, D, E, A, F, B $>$

Subsequence of X	Sequence of indices of X
$Z =<$ A, C, D, E, A $>$	$<$ 1, 3, 5, 6, 7 $>$
$Z =<$ A, B $>$	$<$ 1, 2 $>$, $<$ 1, 9 $>$, $<$ 4, 9 $>$

- A better visual representation:



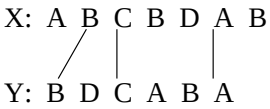
Definition Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

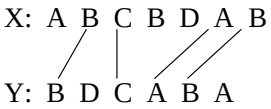
The longest common subsequence (LCS) problem

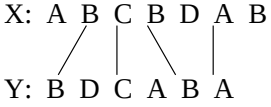
Input: Two sequences X and Y.

Output: Their longest common subsequence.

Examples







Why do we study the LCS problem?

- It has a nice dynamic programming solution.
- It is a very useful algorithm.

Application

Given two versions of a program locate the changes that happened from the first to the second version.

Solution

Find the LCS of the two programs. The text that is not part of the LCS represents the additions/deletions made from the one version to the other.

A brute-force approach

- Let S be the set of all possible subsequences of X and Y.
- Check all of them to find the longest common subsequence.

Question How large is S?

Answer Let $k = \min(legth(X), length(Y))$. Then, $|S| = 2^k$.

This makes the brute-force approach inefficient!

A recursive solution

Let $X =< x_1, x_2, \cdots, x_m >$ and $Y =< y_1, y_2, \cdots, y_n >$.

Denote by X_i the subsequence $X_i =< x_1, x_2, \cdots, x_i >$
and by Y_i the subsequence $Y_i =< y_1, y_2, \cdots, y_i >$.

Let $c[i, j]$ be the length of the LCS of the sequences X_i and Y_j.

We are interested in computing $c[m, n]$.

We can write the recurrence relation:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Question i) Is the above relation correct?
ii) How did we arrive to it?

Answer i) YES

ii) $$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- if $i = 0$ or $j = 0$:
  At least one sequence is empty. Thus, the only common subsequence is the empty one (denoted by ϵ).
- if $i, j > 0$ and $x_i = y_j$:

$$\begin{array}{r} < x_1, x_2, \cdots, x_{i-1}, x_i > \\ = \\ < y_1, y_2, \cdots, y_{j-1}, y_j > \end{array}$$

$LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) \oplus x_i$
$\Longrightarrow c[i,j] = c[i-1, j-1] + 1$

($\oplus$ denotes string concatenation)

- if $i, j > 0$ and $x_i \neq y_j$:

$$\begin{array}{r} < x_1, x_2, \cdots, x_{i-1}, x_i > \\ \neq \\ < y_1, y_2, \cdots, y_{j-1}, y_j > \end{array}$$

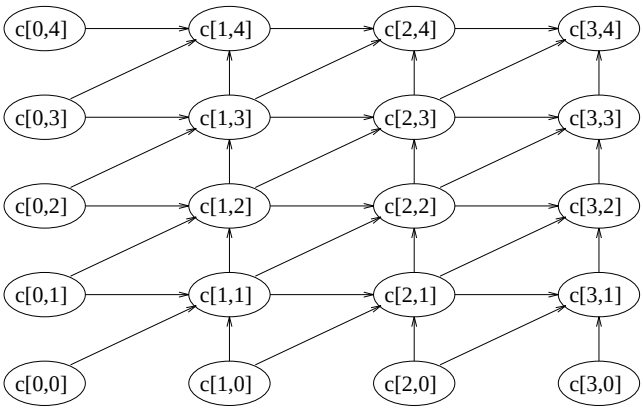
$$LCS[X_i, Y_j] = \begin{cases} LCS[X_i, Y_{j-1}] \\ \text{or} \\ LCS[X_{i-1}, Y_j] \end{cases}$$

$\Longrightarrow c[i,j] = \max(c[i, j-1], c[i-1, j])$

A recursive approach

- In the case where the 2 subsequences have no common subsequences (besides ϵ), the algorithm will take exponential time.

Example Draw the "recursion graph" for $C[3, 4]$.



Exercise Compute the number of times that the procedure which computes $c[i, j]$ is invoked during the computation of $c[m, n]$, where $0 \leq i \leq m, \quad 0 \leq j \leq n$.

- Obviously, we have overlapping subproblems that the recursive algorithm will solve several times.

(The recursive algorithm will take at least exponential time.)

Observation We can easily compute $c[m, n]$ in a bottom-up fashion.

(Fill matrix c in a row-major fashion from left to right.)

$LCS\_length(X, Y)$

$m = length(X)$
$n = length(Y)$
for $i = 1$ **to** m **do** $c[i, 0] = 0$
for $j = 1$ **to** n **do** $c[0, j] = 0$
for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do**
    if $x_i = y_j$
      then $c[i, j] = c[i-1, j-1] + 1$
        $b[i, j] = \swarrow$
      else if $c[i-1, j] \geq c[i, j-1]$
        then $c[i, j] = c[i-1, j]$
          $b[i, j] = \downarrow$
        else $c[i, j] = c[i, j-1]$
          $b[i, j] = \leftarrow$
return $c,\ b$

Note Matrix b is used to recover the common subsequence.

Example X= A B C B D A B
Y= B D C A B A

	j	0	1	2	3	4	5	6
i			(B)	D	(C)	A	(B)	(A)
7	B	0	↙ 1	↓ 2	↓ 2	↓ 3	↙ 4	↓ 4
6	(A)	0	↓ 1	↓ 2	↓ 2	↙ 3	↓ 3	↙ 4
5	D	0	↓ 1	↙ 2	↓ 2	↓ 2	↓ 3	↓ 3
4	(B)	0	↙ 1	↓ 1	↓ 2	↓ 2	↙ 3	← 3
3	(C)	0	↓ 1	↓ 1	↙ 2	← 2	↓ 2	↓ 2
2	(B)	0	↙ 1	← 1	← 1	↓ 1	↙ 2	← 2
1	A	0	↓ 0	↓ 0	↓ 0	↙ 1	← 1	↙ 1
0		0	0	0	0	0	0	0

Question 1 What is the time complexity of LCS()?

Answer: $O(mn)$

Question 2 How much time is required for constructing the LCS from matrix c?

Answer: $O(m+n)$

Question 3 What is the space requirement of LCS()?

Answer: $O(mn)$

Question 4 Can we improve the space complexity for the case where we ONLY want to compute the length of the LCS?

Answer: Yes! Only two rows are needed.
$\Longrightarrow$ $O(\min(m,n))$ space.

Note: We cannot trace our steps back to recover the LCS.

Reading Material

§16.3 "Longest common subsequence" pp. 314-319.

Suggested Exercises

16.3-1, 16.3-2, 16.3-3, 16.3-4, 16.3-5.

All-Pairs Shortest Paths

Input: A weighted, directed graph $G = (V, E)$ with a weight function $w : E \to \mathcal{R}$.

Output For every pair of vertices $u, v \in V$ a shortest path from u to v.

Application Create the "table of distances" between the cities of a road atlas.

A simple algorithm

$Trivial\_All\_Pairs(G, w)$

for each $v \in V(G)$ **do**
Run $Dijkstra()$ with v as the source vertex

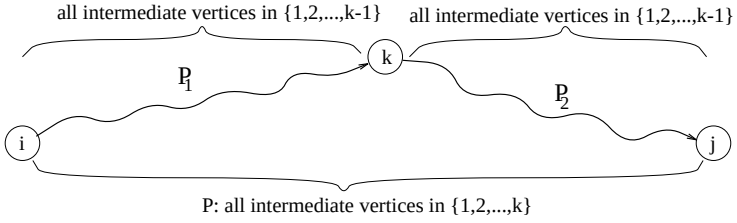
Complexity Linear array implementation: $O(V^3)$
Binary heap implementation: $O(VE \log V)$

Note We will only get the correct results if all weights are non-negative.

A dynamic programming approach

- **The structure of a shortest path**

Consider the shortest path from vertex i to vertex j which passes only through vertices in the set $\{1, 2, \cdots, k\}$.



- **A recursive solution**

Notation $d_{i,j}^{(k)}$ =the weight of the shortest path from i to j which passes only through vertices in $\{1, 2, \cdots, k\}$.

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{if } k = 0 \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

$Floyd\_Warshal(W)$

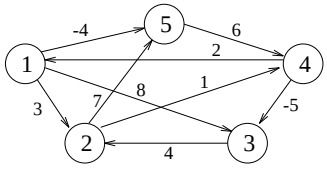
$n = rows(W)$
$d^{(0)} = W$
for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $d_{i,j}^{(k)} = \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$
return $d^{(n)}$

Time & space complexity: $O(V^3)$

Note 1 The space complexity can be reduced to $O(V^2)$. Simply observe that in the computation of $d^{(k)}$ we only use $d^{(k-1)}$.

Note 2 The $Floyd\_Warshal$ algorithm works only for graphs with **no negative weight cycles**. (Why?)

Note 3 Up to now, we only computed the costs of the shortest paths. We haven't show how to recover the paths. (How do we recover the paths?)



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

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

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

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

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

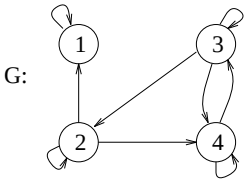
$$d^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

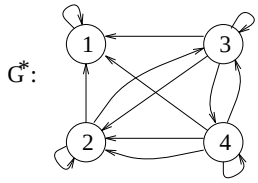
Transitive closure of directed graphs

Input: A directed graph $G = (V, E)$.

Output: A directed graph $G^\star = (V, E^\star)$ where
$E^\star = \{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}$.

Example

G: 

G^*: 

A simple algorithm

- Assign weight "1" to all edges of G.
- Run $Floyd\_Warshal()$.
- If the cost of the path from i to j is not ∞ then $(i, j) \in E^\star$.

A similar algorithm

$$t_{i,j}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

$$t_{i,j}^{(k)} = t_{i,j}^{(k-1)} \quad \vee \quad (t_{i,k}^{(k-1)} \ \wedge \ t_{k,j}^{(k-1)})$$

$Transitive\_Closure(G)$

$n = |V|$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    if $i = j$ **or** $(i, j) \in E$ **then** $t_{i,j}^{(0)} = 1$
    else $t_{i,j}^{(0)} = 0$
for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $t_{i,j}^{(k)} = t_{i,j}^{(k-1)} \quad \vee \quad (t_{i,k}^{(k-1)} \ \wedge \ t_{k,j}^{(k-1)})$
return $t^{(n)}$

Analysis: Identical to $Floyd\_Warshal$.

Advantages It is faster and uses less space (by a constant factor).

Reading Material

§26.2 "All-pairs shortest paths" pp. 558–562.
"Transitive closure of a directed graph" pp. 562-565.

Suggested Exercises

26.2-1, 26.2-2, 26.2-3, 26.2-5.

On Lower Bounds

When we are given a problem P and an algorithm A that solves P, we want to know: *"Is algorithm A the best possible algorithm for P?"*

Possible improvements on A may be:

- Reduction of required time.
- Reduction of required space.

We want a theorem of the form:

"All algorithms for problem P have complexity $T(n) \geq f(n)$."

- A function $f(n)$ appearing in such a theorem is called a *lower bound* on the complexity of problem P.
- If there exists an algorithm for P of complexity $f(n)$, that algorithm is *optimal*.

The model of computation must be well defined

It must be clear what is the model of computation we assume when arriving at a lower bound.

- The way we access data must be defined (random access vs sequential access).
- The possible operations on the data must be defined (the instruction set used by the algorithm).

Example

Assume a sorted list of elements is on a tape that can be accessed sequentially, the tape head is initially at the beginning of the tape, and that elementary operations are:

1. *move_head*(*direction*) (*direction* is either $Left$ or $Right$)
2. *compare*(*key*) (compares *key* with element under the tape-head)
3. **if** "condition" **then** "jump"

Assume that each elementary operation takes $O(1)$ time.

- A trivial lower bound for "search" under this model is $\Omega(n)$.

  If the *key* is the last element in the list, we have to execute at least $n-1$ *move_head*(*right*) statements.

Question If $\Omega(n)$ is a lower bound on the searching problem for the model stated above, why it is possible to have an $O(\log n)$ algorithm (as binary search)?

Answer Because the binary search algorithm is designed for a random access machine (RAM). A different model of computation!

When we state a lower bound for some problem, we must also describe the model of computation under which the lower bound was derived.

Input-output lower bounds

- When the input of a problem is of size n, we need at least n steps to read the input.

  Note: In several cases we assume that the data are already stored in main memory.

- When the output of a problem consists of $f(n)$ primitive elements, then we need at least $\Omega(f(n))$ time to report (output) them.

Examples

Sorting	$\Omega(n)$ lower bound.
Matrix multiplication	$\Omega(n^2)$ lower bound.
Single-source shortest paths	$\Omega(\|V\|)$ lower bound.
All-pairs shortest paths	$\Omega(\|V\|^2)$ lower bound.
Minimum spanning tree	$\Omega(\|V\|)$ lower bound.
Etc ...	

Adversary lower bounds

Problem Find the sum of n numbers.

Theorem Finding the sum of n numbers requires $\Omega(n)$ time.

Proof (Intuition: the algorithm must examine all elements of the input in order to find their sum).

- Assume that there is an algorithm that does not examine all elements.
- The adversary watches and, when the algorithm terminates, it moves in and changes an unexamined element.
- If the algorithm is run again, it will give the same result since it doesn't see the change. But, the true result has changed, and thus, the algorithm is incorrect!

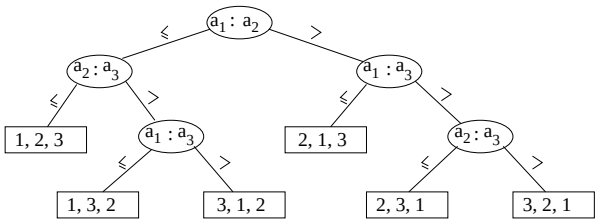
So, any correct summation algorithm examines every element and so is $\Omega(n)$.

Lower bound for sorting

Model of computation

- Random access machine (RAM).
- Only comparisons between elements are allowed.

- Algorithms for the above model are called *"comparison sorts"*.
- We can view them as *decision trees*:



Notation

$a : b$	Internal node. a is compared with b.
	Leaf. It contains the output of the algorithm.

- There are $n!$ possible permutations of the input. Thus, the decision tree must have at least $n!$ leaves.

  The length of the longest path from the root of the decision tree to any of its leaves represents the worst case number of comparisons the sorting algorithm performs.

Facts

1. A binary tree of height h has at most 2^h leaves.

2. $n! > \left(\frac{n}{e}\right)^n$

   $e = 2.71828...$

   The formula is derived from Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Theorem Any decision tree that sorts n elements has height $\Omega(n \log n)$.

Proof Let such a tree have height h. Thus,

$$n! \leq 2^h \qquad \text{(Fact 1)}$$

$$\Longrightarrow \quad h \geq \log(n!)$$

$$\Longrightarrow \quad h \geq \log\left(\frac{n}{e}\right)^n \qquad \text{(Fact 2)}$$

$$\Longrightarrow \quad h \geq \log\left(\frac{n^n}{e^n}\right) = n \log n - n \log e$$

$$\Longrightarrow \quad h = \Omega(n \log n)$$

□

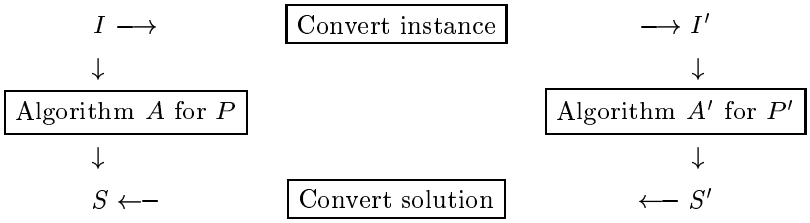
Theorem *MergeSort* is asymptotically optimal.

Proof *MergeSort* used the same model of computation as the one we used to obtain the $\Omega(n \log n)$ lower bound. Moreover, its time complexity is $O(n \log n)$.

□

Reductions (Transformations)

- When two problems are related, it may be possible to use an algorithm for the one to solve the other.
- Let P, P' be problems and I be an arbitrary instance of P. Also assume that we know an algorithm A' that solves P'. We may be able to solve P as follows:



- If there exist algorithms for converting I to I' and S' to S, we say that P reduces to P' or, P transforms to P'.
- We write $P \propto P'$.

Theorem Suppose that $P \propto P'$ and that the worst case time complexities for converting I to I' and S' to S are $f_1(n)$ and $f_2(n)$, respectively, where n is the size of instance I. Then,

1. For every algorithm for P' of worst case time complexity $W'(n)$, there is an algorithm for P of complexity $f_1(n) + W'(n) + f_2(n)$.

2. If $g(n)$ is a lower bound on the complexity of P, then $g(n) - f_1(n) - f_2(n)$ is a lower bound on the complexity of P'.

Proof (See previous figure)

1. By constructing the algorithm.
2. By contradiction.

□

A lower bound for the Huffman tree problem

Problem Given n weights $w_1,\ w_2, \ldots, w_n$ find a Huffman tree for them.

- A Huffman tree minimizes $\sum_{c \text{ is a leaf}} w(c) \cdot d_T(c)$.

Theorem Constructing Huffman trees is of complexity $\Theta(n \log n)$.

Proof Consider the following problem $Sort^\star$:

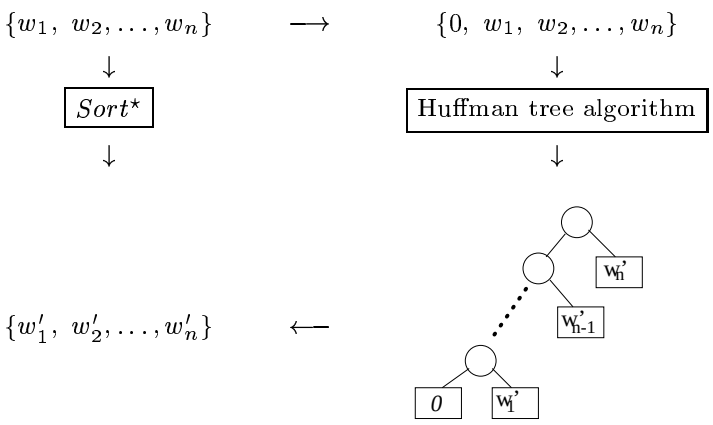
We are given a set of n positive numbers $w_1,\ w_2, \ldots, w_n$ for which there is a permutation $w'_1,\ w'_2, \ldots, w'_n$ of them such that

$$w'_i > \sum_{j=1}^{i-1} w'_j$$

for all i. Sort these numbers.

- It is easy to see that $Sort^\star = \Omega(n \log n)$. This is because, again, any decision tree must have at least $n!$ leaves.

$Sort^\star \propto$ Huffman tree problem



Thus, any decision tree algorithm for constructing Huffman trees is $\Omega(n \log n)$.

□

Reading Material

§9.1 "Lower bounds for sorting" pp. 172-174.

From "Algorithms and Data Structures" by Jeff Kingston:

§11.1 "Model of computation" pp.284-286.
§11.2 "Adversary bounds" pp. 286-288.
§11.3 "Transformations" pp. 297-301.

Suggested Exercises

9.1-4, 9.1-5.

Sorting in Linear Time

Counting-Sort

Input: n integers in the range $[1..k]$.

Output: The n integers sorted in increasing order.

Idea: Determine for each input element x, the number of elements which are $< x$.

- Counting-Sort uses 3 arrays:

  $A[1..n]$: Contains the input elements.

  $B[1..n]$: Contains the sorted output.

  $C[1..k]$: $C[i]$ contains the number of elements which are $\leq i$.


```
COUNTING-SORT(A, B, k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
/* C[i] now contains the number of */
/* elements equal to i (see (a) ). */
for i ← 2 to k
    do C[i] ← C[i] + C[i − 1]
/* C[i] now contains the number of */
/* elements ≤ i (see (b) ). */
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

(a)

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

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

(b)

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

(c)

	1	2	3	4	5	6	7	8
B							4	

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

(d)

	1	2	3	4	5	6	7	8
B		1					4	

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

(e)

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

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

(f)

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

Time complexity: $O(k+n)$ If $k = O(n) \Longrightarrow O(n)$ algorithm.

Note This result does not contradict with the $\Omega(n \log n)$ lower bound. We are not using the comparison model. Actually, COUNTING-SORT() never makes a comparison.

Definition A sorting algorithm is *stable* if elements with the same value appear at the output in the same order as they appear at the input.

Fact COUNTING-SORT() is stable.
(Prove it by examining the algorithm.)

Radix-Sort

Input: n elements such that each element consists of d digits where the 1^{st} digit is the lowest-order digit and the d^{th} digit is the highest-order digit.

Output: A sorted array containing the n elements in increasing order.

Idea: Sort the elements on their *least significant digit first.*

Example

329		72**0**		7**2**0		**3**29
457		35**5**		3**2**9		**3**55
657		43**6**		4**3**6		**4**36
839	$\Longrightarrow$	45**7**	$\Longrightarrow$	8**3**9	$\Longrightarrow$	**4**57
436		65**7**		3**5**5		**6**57
720		32**9**		4**5**7		**7**20
355		83**9**		6**5**7		**8**39

$RADIX - SORT(A, d)$
for $i = 1$ **to** d
do use a stable sorting algorithm to sort array A on digit i.

Proof of correctness (by induction on the number of passes)

- Assume that the lower order digits are sorted.
- Show that sorting on the next digit leaves the array correctly sorted.

- Consider any two elements:
  - If 2 digits in this position are different, sorting the elements on that position results to the correct ordering. The lower-order digits are irrelevant.
  - If 2 digits in this position are the same, the elements are already in the right order since they are already sorted on the lower-order digits. *Since we use a stable sorting algorithm, the elements stay in the right order.*

□

Analysis

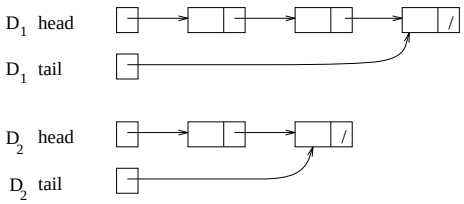
- Each pass takes $\Theta(n + k)$ time, where each digit takes values in the range $[1..k]$.
  $\Longrightarrow \Theta(d(n + k))$
- If $k = O(n) \Longrightarrow \Theta(dn)$, using COUNTING-SORT().

Bin-Sort(similar to Radix-Sort)

Input:	A linked list of n d-digit elements where each digit is in the range $[1..k]$.
Output:	A sorted linked list the n elements in increasing order.

Idea:
- Partition the elements into *"bins"*.
- Work similar to Radix-Sort.

Illustration ($k = 2$)



Example Sort the list $< 36,\ 9,\ 0,\ 25,\ 1,\ 49,\ 64,\ 16,\ 81,\ 4 >$ of 2-digit numbers.

Bin	Contents	Bin	Contents
0	0	0	0, 1, 4, 9
1	1, 81	1	16
2		2	25
3		3	36
4	64, 4	4	49
5	25	5	
6	36, 16	6	64
7		7	
8		8	81
9	9, 49	9	
	d = 1		**d = 2**

$BIN - SORT(L, d, k)$ /* L is the list of the input elements. */
/* Each element consists of d digits. */
/* Each digit is in the range $[1..k]$. */

for $i = 1$ **to** d **do**

- Process list L sequentially and place each element of L at the top of the list $L_j,\ 1 \leq j \leq k$, where j is the i^{th} digit of the element under consideration.
- Concatenate $L_1,\ L_2, \ldots,\ L_k$ into L.

Time complexity $\Theta(dn + dk)$

- If $k = O(n)$ and d is a constant $\implies \Theta(n)$.

Reading Material

§9.2 "Counting sort" pp. 175-177.
§9.3 "Radix sort" pp. 178-180.

Suggested Reading

§9.4 "Bucket sort" pp. 180-183.

Suggested Exercises

9.2-1, 9.2-2, 9.2-3.
9.3-1, 9.3-3, 9.3-4.

Graph Acyclicity Testing

Input: A directed graph $G = (V, E)$.

Output: Does G contain a cycle?

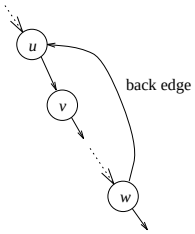
ACYCLICITY(G)

1. Run $DFS(G)$ and classify the edges as *tree edges, forward edges, cross edges* and *back edges.*
2. If there exists a back edge then G contains a cycle.

Time complexity: $O(V + E)$

Proof of correctness

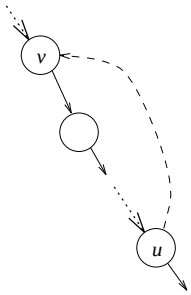
- If G has a back edge then, obviously, it contains a cycle.



$\Longrightarrow \langle u,\ v, \ldots,\ w,\ v \rangle$ is a cycle.

Claim If a directed graph has a cycle, then a back edge will always be encountered in any DFS of the graph.

- Suppose the graph has a cycle.
- Let v be the vertex with the smallest discovery time $d[v]$ out of the vertices that belong to the cycle.
- Consider an edge $u \longrightarrow v$ on some cycle containing v.



- Since u is on the cycle, u is a descendant of v in the DFS forest.
  $\Longrightarrow$ $u \longrightarrow v$ is not a cross edge.
- Since $d[v] < d[u] \Longrightarrow$ $u \longrightarrow v$ is not a tree edge or a forward edge.
  Thus, $u \longrightarrow v$ is a back edge.

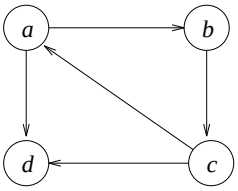
□

Strongly Connected Components

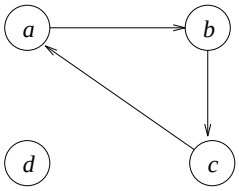
A *strongly connected component* of a directed graph $G = (V, E)$ is a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, there exists a path from u to v and a path from v to u.

Input: A directed graph $G = (V, E)$.
Output: The strongly connected components of G.

Example



A directed graph

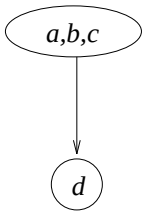


Its strongly connected components

- Every vertex of a directed graph is in some strongly connected component, but certain edges of the graph may be in no component. Such edges are called *cross-component edges*.

- We can represent the interconnections between the components by constructing a *reduced component graph*.

Example (continued)



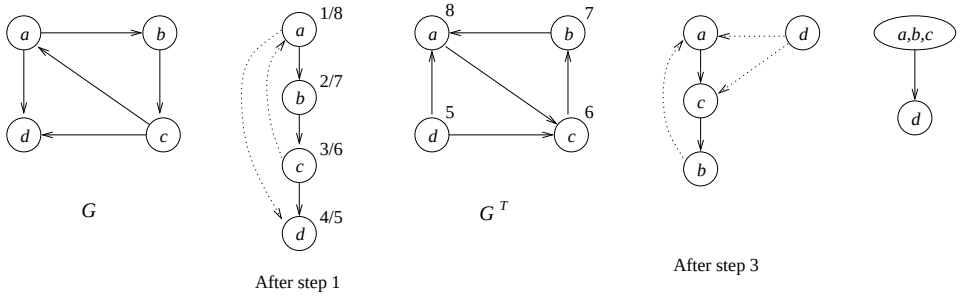
- The reduced component graph is ALWAYS acyclic, i.e., a DAG.

STRONGLY_CONNECTED_COMPONENTS(G)

1. Call $DFS(G)$ to compute the finishing times $f[u]$ for each vertex u of G.
2. Compute the transpose G^T of G.
3. Call $DFS(G^T)$, but in the main loop of the DFS consider the vertices in order of decreasing $f[]$ (as computed in 1).
4. Output the vertices of each tree in the DFS-forest produced in 3 as a separate strongly connected component.

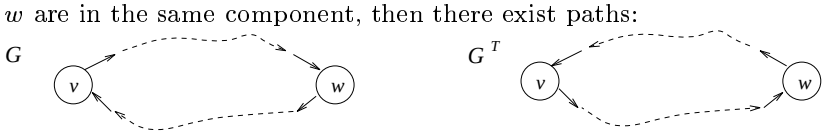
Time complexity: $O(V + E)$

Example



Proof of correctness

- If v and w are in the same component, then there exist paths:



- We have to show that:

v and w are in the same strongly connected component of G.	$\Longleftrightarrow$	v and w are in the same tree of the DFS- forest of G^T.

"$\Longrightarrow$"

- Suppose that in G^T we begin the search at some root x and we reach v. Then, w will end up in the same tree (since w is reachable from v).

"$\Longleftarrow$" (v and w are in the same tree of the DFS- forest of $G^T \Longrightarrow v$ and w are in the same strongly connected component of G.)

- Let x be the root of the tree in the DFS-forest of G^T in which v and w belong.
  $\Longrightarrow$ In G^T $\exists$ a path $x \longrightarrow v$ $\Longrightarrow$ In G $\exists$ a path $v \longrightarrow x$.

- In G^T, v was unvisited when x was discovered.
  $\Longrightarrow$ In G, $f[x] > f[v]$.
  $\Longrightarrow$ In the search of G, v is visited during the search of x.
  $\Longrightarrow$ v is a descendant of x in the DFS-forest of G.
  $\Longrightarrow$ In G $\exists$ a path $x \longrightarrow v$. $\Longrightarrow$ x and v are in the same strongly connected component.

- Similarly we prove that x and w are in the same strongly connected component.

□

Proof of correctness for algorithm STRONGLY_CONNECTED_COMPONENTS()

We have claimed that the vertices of a strongly connected component correspond precisely to the vertices of a tree in the spanning forest of the seconf depth-first search. To see why, observe that if v and w are vertices in the same strongly connected component, then there are paths in G from v to w and from w to v. Thus, there are also paths from v to w and from w to v in G^T.

Suppose that in the depth-first search of G^T we begin a search at some root x and either reach v or w. Since v and w are reachable from each other, both v and w will end up in the tree with root x.

Now suppose that v and w are in the same tree of the DFS-forest of G^T. We must show that v and w are in the same strongly connected component of G. Let x be the root of the tree containing v and w. Since v is a descendant of x, there is a path from x to v in G^T. Thus, there exists a path from v to x in G.

During the construction of the the depth-first search at x was initiated. Thus, x has a higher finishing timestamp than v, i.e., $f[x] > f[v]$. So, in the depth-first search of G, the recursive call at v terminated before the recursive call at x did. But, in the depth-first search of G, the search at v could not have started before x since the path in G from v to x would then imply that the search at x would start and end before the search at v ended.

We conclude that in the depth-first search of G, v is visited during the search of x and hence v is a descendant of x in the DFS-forest of G. Thus, there exists a path from x to v in G. Therefore, x and v are in the same strongly connected component. An identical argument shows that x and w are in the same strongly connected component and hence v and w are in the same strongly connected component, as shown by the path from v to x to w and the path from w to x to v.□

Reading Material

Lemma 23.10 "Acyclicity testing" pp. 486-487.

"Strongly connected components" from lecture notes and
§10.7 "Strongly connected components" pp. 248-249 from Jeff Kington's book.

Suggested Reading

§23.5 "Strongly connected components" pp. 488-494.
(A different proof was given at the lecture)

Suggested Exercises

23.4-3, 23.4-5.
23.5-1 – 23.5-7.

Minimum Spanning Trees: Kruskal's Algorithm

Input: A connected undirected graph $G = (V, E)$ with a weight function $w : E \to \mathcal{R}$.

Output: A minimum spanning tree for G.

- Recall the **generic** algorithm:

```
Generic_MST(G, w)
A = ∅
while A does not form a spanning tree do
        Find an edge (u, v) that is safe for A.
        A = A ∪ {(u, v)}
return A
```

Algorithm Invariant A is always a subset of some minimum spanning tree.

- An edge (u, v) is a *safe edge* for A if it can be safely added to A without violating the invariant.


```
MST_Kruskal(G, w)
1. A = ∅
2. for each vertex v ∈ V(G)
3.         do MAKE_SET(v)
4. Sort the the edges of E in non-decreasing weight w.
5. for each edge (u, v) ∈ E, in order by non-decreasing weight,
6.         do if FIND_SET(u) ≠ FIND_SET(v)
7.                 then A = A ∪ {(u, v)}
8.                      UNION(u, v)
9. return A
```

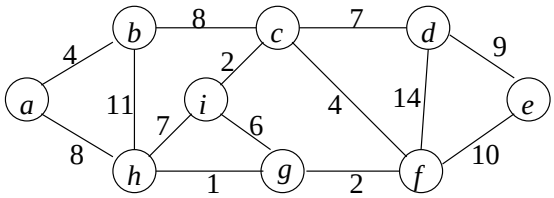
- We used an ADT that supports operations on disjoint sets.
  - $-MAKE\_SET(v)$:Creates a new set that contains element v.
  - $-FIND\_SET(v)$: Returns the set in which v belongs.
  - $-UNION(u, v)$:Creates a new set that is the union of sets $FIND\_SET(u)$ and $FIND\_SET(v)$.

- **Analysis** Steps 2-3: $O(V)$
  Step 4: $O(E \log E)$
  Steps 5-8: $O(E + V \log V)$ (See §22.2)
  TOTAL: $O(E \log E)$

Example

E is sorted in increasing order:

(h, g)	1
(i, c)	2
(g, f)	2
(a, b)	4
(c, f)	4
(i, g)	6
(c, d)	7
(h, i)	7
(a, h)	8
(b, c)	8
(d, e)	9
(f, e)	10
(b, h)	11
(d, f)	14



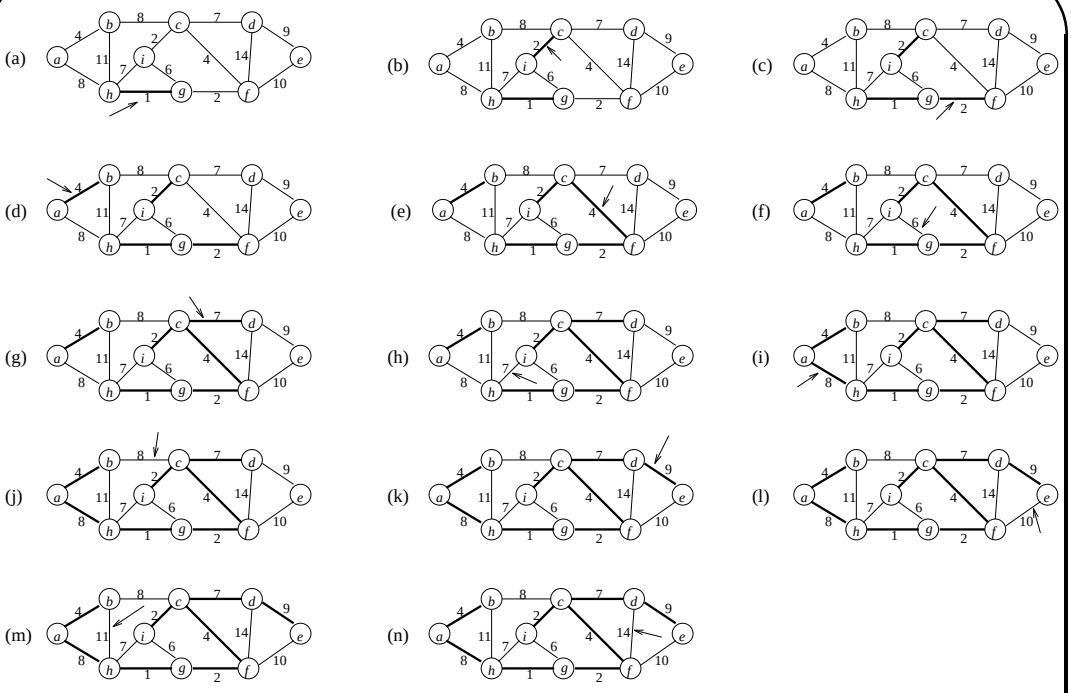
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

(m) (n)

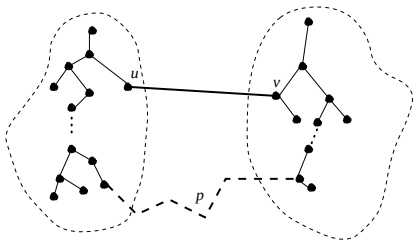


Proof of correctness for Kruskal's algorithm

Theorem If we decide to add edge (u, v) in A, (u, v) is a safe edge.

Proof We add (u, v) in A iff u and v belong to different trees of G (formed so far by the execution of Kruskal's algorithm).

- If (u, v) is not safe, then it does not belong in the MST. Thus, there exists a path p from the tree which contains u to the tree which contains v that belongs to the MST of G.



- Since p consists from edges with weights greater than or equal to $w(u, v)$, we can substitute any edge in p with (u, v) and obtain a spanning tree with the same weight as the assumed MST of G.

  Thus, (u, v) is safe.

□

Single-source shortest path problems

Input: • A weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathcal{R}$ mapping edges to real valued weights.
• A source node s.

Output: The shortest paths from s to any other vertex in G.

Relaxation

- For each vertex $v \in V$, $d[v]$ is an upper bound on the weight of a shortest path from the source s to v.
  $d[v]$ = shortest_path_estimate

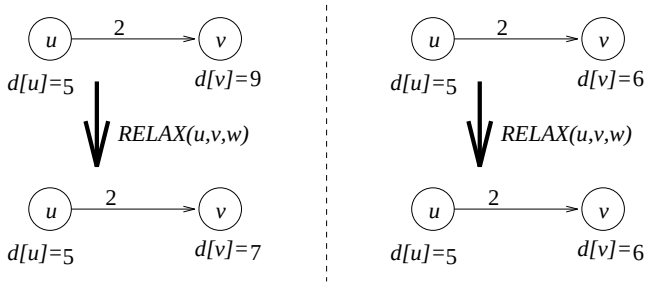
- We initialize $d[v]$, $v \in V$, as follows:

  INITIALIZE_SINGLE_SOURCE(G, s)
  for each vertex $v \in V$
  do $d[v] \leftarrow +\infty$
  $\pi[v] \leftarrow NIL$
  $d[s] \leftarrow 0$

While *relaxing edge* (u, v), we check if we can improve the shortest path from s to v by going through edge (u, v).

$RELAX(u, v, w)$
if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$

Example



- Immediately after relaxing edge (u, v) by executing $RELAX(u, v, w)$, we have: $d[v] \leq d[u] + w(u, v)$.

- Dijkstra's algorithm can be rewritten as:

$Dijkstra(G, w, s)$
1. $INITIALIZE\_SINGLE\_SOURCE(G, s)$
2. $S \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. **while** $Q \neq \emptyset$
5. **do** $u = extract\_min(Q)$
6. $S \leftarrow S \cup \{u\}$
7. **for** each vertex $v \in Adj[u]$
8. **do** $RELAX(u, v, w)$

Note 1: In Dijkstra's algorithm, each edge is relaxed *exactly 1 time.*

Note 2: Dijkstra's algorithm works only for graphs with non-negative edge weights.

Question: How do we deal with negative weights?

The BELLMAN-FORD algorithm

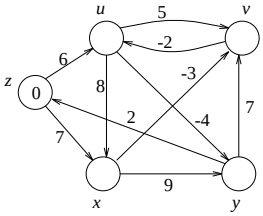
Input:
- A weighted, undirected graph $G = (V, E)$ with weight function $w : E \to \mathcal{R}$ mapping edges to real valued weights.
- A source node s.

Output:
- A boolean value indicating whether there exists a negative weight cycle reachable from the source s.
- If there is no negative weight cycle, the shortest paths (and their weights) from s to any other vertex in G.

```
BELLMAN_FORD(G, w, s)
1. INITIALIZE_SINGLE_SOURCE(G, s)
2. for i = 1 to |V(G)| − 1
3.         do for each edge (u, v) ∈ E(G)
4.                 do RELAX(u, v, w)
5. for each edge (u, v) ∈ E(G)
6.         do if d[v] > d[u] + w(u, v)
7.                 then return FALSE
8. return TRUE
```

Time complexity $O(VE)$ (Each edge is relaxed exactly $|V| - 1$ times.)

Example



We relax the edges in the following order:

(z, u)
(z, x)
(u, v)
(u, y)
(u, x)
(v, u)
(x, v)
(x, y)
(y, z)
(y, v)

- The order of relaxation is arbitrary. Each pass can use a different order.

Proof of correctness of the BELLMAN-FORD algorithm

- Let v be a vertex reachable from s.
- Let $p = \langle s, v_1, v_2, \ldots, v_{k-1}, v \rangle$ be a shortest path of length k from s to v, $k \leq |V| - 1$.
- We prove by induction that $d[v_i] = \delta(s, v_i)$ after the i^{th} pass over the edges.

Basis $d[s] = 0$.

Induction step

◇ **Assume** that $d[v_{i-1}] = \delta(s, v_{i-1})$ after the $(i-1)^{th}$ pass.

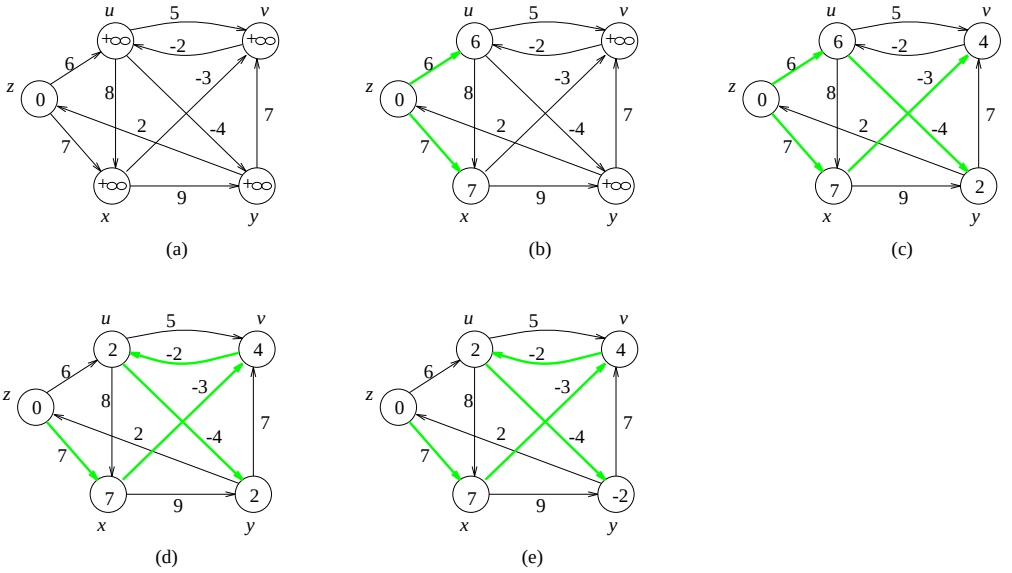
◇ **Prove** that $d[v_i] = \delta(s, v_i)$ after the i^{th} pass.

It follows from the facts:

1. We relax edge (v_{i-1}, v_i) during the i^{th} pass.
2. The sub-paths of shortest paths are shortest paths.

□

Example The edges are relaxed in lexicographic order.



Single-source Shortest Paths in DAGs

$DAGs\_Shortest\_Paths(G, w, s)$

1. Topologically sort the vertices of G
2. $INITIALIZE\_SINGLE\_SOURCE(G, s)$
3. **for** each vertex u (in the topologically sorted order)
4. **do for** each vertex $v \in Adj[u]$
5. **do** $RELAX(u, v)$

Time complexity $O(V + E)$

Reading Material

§24.1 "Growing a minimum spanning tree" pp. 498-503.
§24.2 "Kruskal's algorithm" pp. 504-505.
§25.1 "Single-source shortest paths - relaxation" pp. 514-526
§25.3 "The Bellman-Ford algorithm" pp. 532-535
§25.4 "Shortest-paths in DAGs" pp. 536-538

Suggested Reading

§22.1-2 "Disjoint-set operations" pp. 440-446.

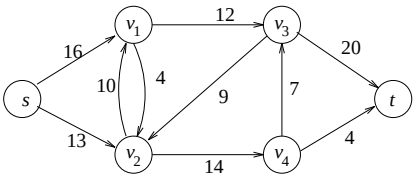
Suggested Exercises

24.2-4
25.3-5
25.4-1

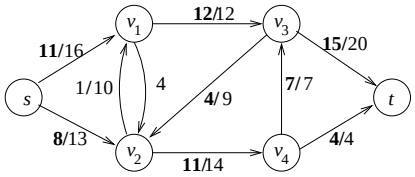
Maximum Flow

- We are given a *flow network*, a source s, and a sink t. We want to find the maximum amount of some commodity that can flow from s to t without violating the *flow capacities* of the edges of the network.

Example



A flow network.



A flow f with $|f| = 19$.

Applications

- We can model:
  1. The flow of liquids through pipes.
  2. The flow of parts through an assembly line.
  3. Electrical networks.
  4. Communication networks.

- Several other problems that seem to be unrelated to the maximum flow problem can be reduced to it (e.g., *maximal matching in bipartite graphs*).

Flow networks and flows

- A *flow network* $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non-negative *capacity* $c(u, v) \geq 0$. If $(u, v) \notin E$ then $c(u, v) = 0$.

- There are 2 distinguished vertices: A *source* s and a *sink* t.

  We assume that every vertex $v \in V - \{s, t\}$ is on some path from s to t.

  $\Longrightarrow$ $|E| \geq |V| - 1$ (the graph is connected).

- A *flow in* G is a real-valued function $f : V \times V \longrightarrow \Re$ satisfying the following properties:

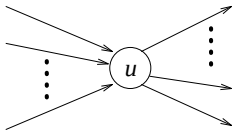
  1. *Capacity constraint:* $\forall\ u, v \in V, \quad f(u, v) \leq c(u, v)$.

  2. *Skew constraint:* $\forall\ u, v \in V, \quad f(u, v) = -f(v, u)$.

  3. *Flow conservation:* $\forall\ u \in V - \{s, t\}, \quad \sum_{v \in V} f(u, v) = 0$.

Flow conservation

- For all $u \in V - \{s, t\}$ we require $\sum_{v \in V} f(u, v) = 0$

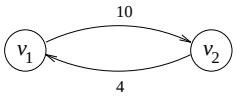


  - The flow that enters a vertex is equal to the flow that exits from it.
  - The flow for all outgoing edges is nonnegative while the flow for all incoming edges is non-positive.
  - The quantity $f(u, v)$ is called the *net flow from vertex u to vertex v.*

- The *value* of a flow f is defined as:

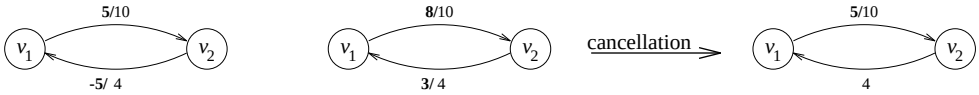
$$|f| = \sum_{v \in V} f(s, v) \qquad \left[= \sum_{v \in V} f(v, t) \right]$$

How to manipulate flows - Cancellation

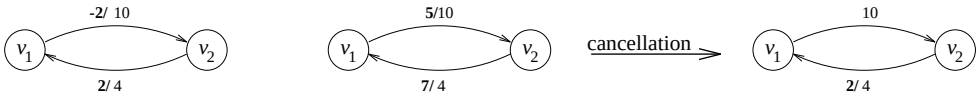


- Ship 8 units from $v_1 \longrightarrow v_2$.

- Ship 3 units from $v_2 \longrightarrow v_1$.



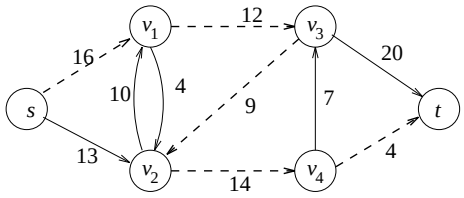
- Ship 7 units from $v_2 \longrightarrow v_1$.



The Ford-Fulkerson Method

Definition An *augmenting path* is a path from the source s to the sink t along which we can push more flow and thus, *augment the flow along this path.*

Example



$Ford\_Fulkerson\_Method(G, s, t)$

1. Initialize flow f to 0
2. **while** there exists an augmenting path p
3. **do** augment flow f along p
4. **return** f

Residual networks

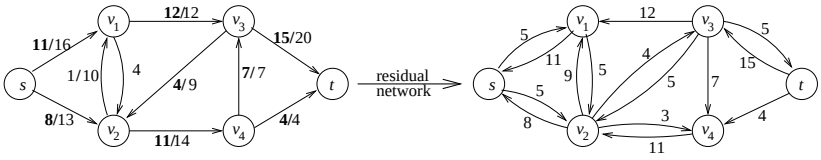
- Let f be a flow in G and consider a pair of vertices $u,\ v\ \in V$. The amount of additional net flow that we can push from u to v before exceeding the capacity $c(u,v)$ is the *residual capacity* $c_f(u,v)$ of (u,v).

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

Note: $c_f(u,v) \geq 0$ for all $u,\ v\ \in V$

- Given a flow network $G = (V,E)$ and a flow f, the *residual network of G induced by f* is $G_f = (V, E_f)$ where $E_f = \{(u,v) :\ c_f(u,v) > 0\}$.

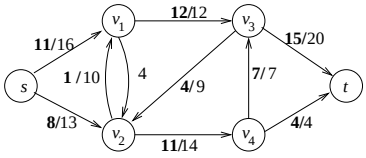
Example



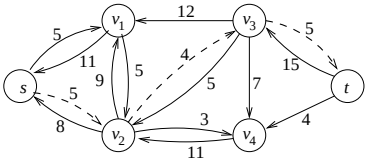
Observation $|E_f| \leq 2|E|$

This is because edge (u,v) is in E_f if at least one of (u,v) or (v,u) is in E.

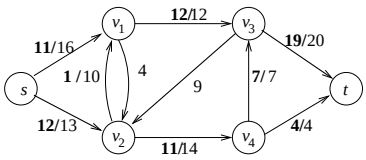
Example (1 iteration of $Ford\_Fulkerson\_Method$)



⟵ Flow f in G.



⟵ G_f and an augmenting path p.



⟵ The flow in G after augmenting along path p.

- Assume an augmenting path p. The maximum amount of flow that we can ship along the edges of p is called the *residual capacity of* p.

$$c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$$

$Ford\_Fulkerson(G, s, t)$

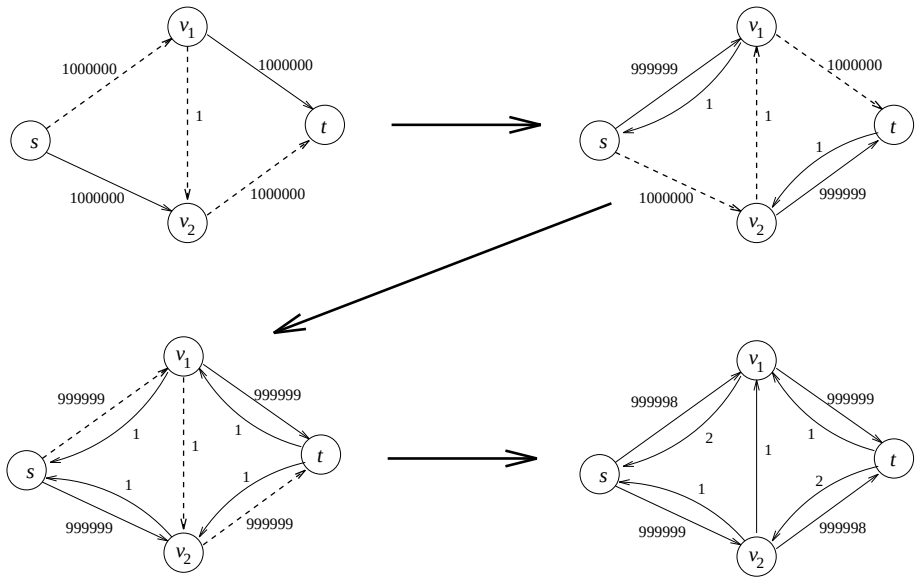
1. **for** each edge $(u,v) \in E$
2. **do** $f[u,v] = 0$
3. $f[v,u] = 0$
4. **while** there exists a path p from s to t in G_f
5. **do** $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
6. **for** each edge (u,v) in p
7. **do** $f[u,v] = f[u,v] + c_f(p)$
8. $f[v,u] = -f[u,v]$

Running time

- It depends on the way we choose the augmenting paths.
- The algorithm may not even terminate.
- If the capacities are **integral** the algorithm will terminate.

- An upper bound on the running time of $Ford\_Fulkerson()$ when the capacities are integral is $O(E|f^*|)$ where f^* is the maximum flow returned by the algorithm.

Example



- An augmenting path can be found in $O(E)$ time with either *depth first search* or *breadth first search*.

- The proof of correctness for Ford-Fulkerson's method follows from the theorem:

  Theorem If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following two statements are equivalent:

  1. f is a maximum flow in G.
  2. The residual network G_f contains no augmenting paths.

Edmonds-Karp algorithm

$Edmonds\_Karp(G, s, t)$

- Compute the flow as in $Ford\_Fulkerson()$.
- Use *breadth first search* to compute the augmenting paths.

Theorem The *Edmonds_Karp* algorithm makes $O(VE)$ flow augmentations.

Time complexity: $O(VE^2)$
(A total of $O(VE)$ calls to breadth first search.)

Note The capacities **must** be integral.

Reading material

§27 "Maximum Flow" pp. 579-580.
§27.1 "Flow networks" pp. 580-584
§27.2 "The Ford-Fulkerson method" pp. 587-600
(Read only the subsections covered at the lecture.)

Suggested reading:

The analysis of the Edmonds-Karp Algorithm (Theorem 27.9)

Suggested Exercises

27.1-1 – 27.1-3
27.2-2

Cuts on Flow Networks

Notation Let f be a flow in flow graph $G = (V, E)$ and $X,\ Y \subseteq V$.

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$$

Lemma

1. $f(X, X) = 0, \qquad X \subseteq V$

2. $f(X, Y) = -f(Y, X), \qquad X,\ Y \subseteq V$

3. $\begin{aligned} f(X \cup Y, Z) &= f(X, Z) + f(Y, Z) \\ f(Z, X \cup Y) &= f(Z, X) + f(Z, Y) \end{aligned} \qquad X,\ Y,\ Z \subseteq V,\ \ X \cap Y = \emptyset$

□

Lemma Let $G = (V, E)$ be a flow network with source s and sink t and let f be a flow in G. Let G_f be the residual network induced by f, and f' be a flow in G_f. Then, $f + f'$ is a flow in G with value $|f + f'| = |f| + |f'|$.

Proof $(f + f')(u, v) \stackrel{\text{def}}{=} f(u, v) + f'(u, v)$

1. Skew symmetry

$$\begin{aligned} (f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -\left(f(v, u) + f'(v, u)\right) \\ &= -(f + f')(v, u) \end{aligned}$$

2. Capacity constraint

$$\begin{aligned} (f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\leq f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v) \end{aligned}$$

Note: $f'(u, v) \leq c_f(u, v) \stackrel{\text{def}}{=} c(u, v) - f(u, v)$

3. Flow conservation

For all $u \in V - \{s, t\}$,

$$\begin{aligned}\textstyle\sum_{v \in V}(f + f')(u, v) &= \textstyle\sum_{v \in V} f(u, v) \; + \; \sum_{v \in V} f'(u, v) \\ &= 0 + 0 \\ &= 0\end{aligned}$$

The value of the new flow

$$\begin{aligned}|f + f'| &= \textstyle\sum_{v \in V}(f + f')(s, v) \\ &= \textstyle\sum_{v \in V} f(s, v) \; + \; \sum_{v \in V} f'(s, v) \\ &= |f| + |f'|\end{aligned}$$

□

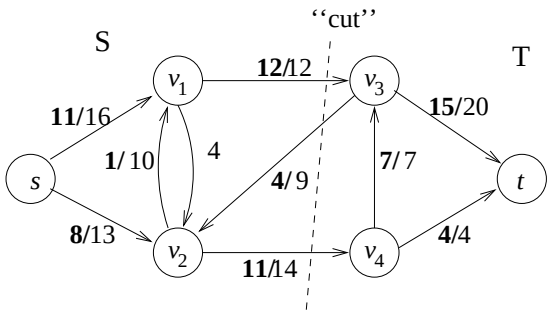
Definitions

- A *cut* (S, T) of a flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

- The *capacity of a cut* (S, T), $c(S, Y)$, is defined as:

$$c(S, T) = \sum_{x \in S} \sum_{y \in T} c(x, y)$$

Example



- $f(S, T) = f(v_1, v_3) + f(v_2, v_4) + f(v_2, v_3) = 12 + 11 + (-4) = 19$
- $c(S, T) = c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26$

Lemma Let f be a flow in a flow network G and let (S,T) be a "cut" in G. Then,

$$f(S,T) = |f|$$

Proof

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) - 0 \\ &= f(s,V) + f(S-\{s\},V) \\ &= f(s,V) \\ &= |f| \end{aligned}$$

□

Exercise Work out the details of the above proof. See also exercise 27.1-5.

Lemma The value of any flow f in a flow network G is *bounded from above* by the capacity of any "cut" of G.

Proof

Let (S,T) be an arbitrary "cut" of G and let f be any flow.

By the previous lemma,

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

□

Theorem *(Max-Flow Min-Cut Theorem)*
If f is a flow in a network $G = (V, E)$, then the following statements are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting path.
3. $|f| = c(S, T)$ for some "cut" (S, T) of G.

Proof

(1) $\Longrightarrow$ (2) (By contradiction)

Suppose that f is a maximum flow and that G_f **has** an augmenting path p (with flow f_p). Then,

$$|f + f_p| \;=\; |f| + |f_p| \;>\; |f|$$

a clear contradiction since f is a maximum flow.

(2) $\Longrightarrow$ (3)

- "G_f has an augmenting path" $\equiv$ "$\neg\exists$ a path from s to t"
- Define $S = \{v : v \text{ is reachable from } s\}$
  and $T = V - S$
- (S, T) is a "cut". Thus,
  "for all (u, v) such that $u \in S,\ v \in T$ we have that $f(u, v) = c(u, v)$"
  $\Longrightarrow\ |f| = f(S, T) = c(S, t)$

(3) $\Longrightarrow$ (1)

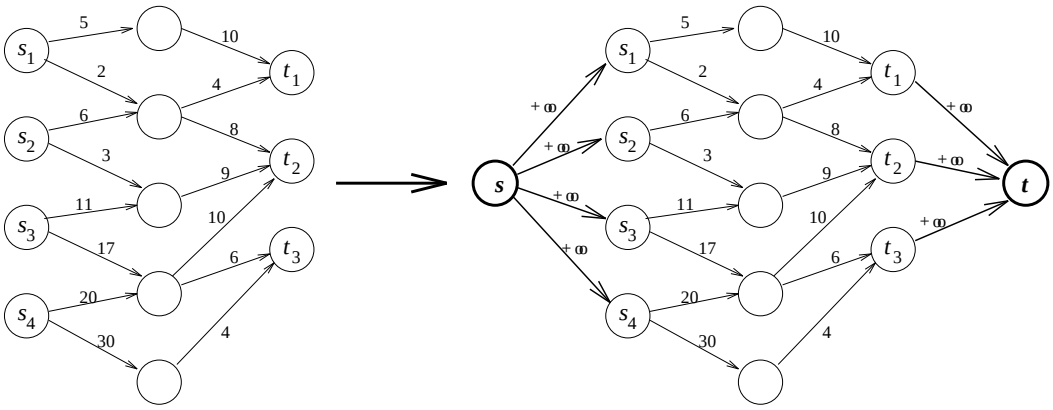
- $|f| \leq c(S, T)$ for all "cuts" (S, T) (Lemma on page 220.)

  Thus, the fact that $|f| = c(S, T)$ for some "cut" (S, T) implies that f is maximum.

□

Networks with multiple sources and sinks

- Reduces to a single-source single-sink maximum flow problem.



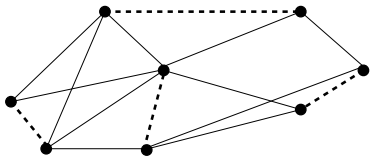
- In practice, instead of $+\infty$ we can set the capacity of the new edges to $c > \sum_{e \in E} c(e)$.

Maximum Bipartite Matching

Definitions

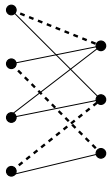
- Given an undirected graph $G = (V, E)$, a *matching* M is a subset of edges, $M \subseteq E$, such that for all vertices $v \in V$, at most one edge of M is incident to v.

Example



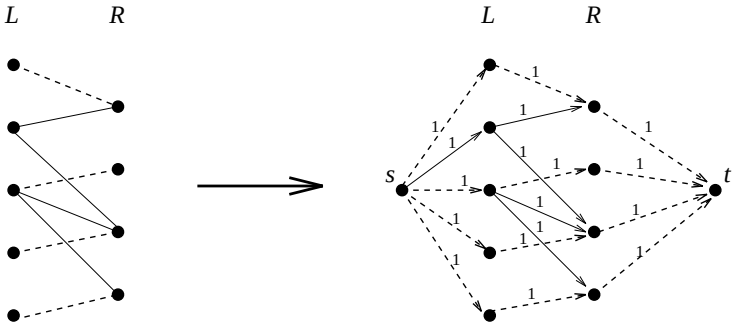
- A *maximum matching* is a matching of maximum cardinality.

- A *maximum bipartite matching* is a maximum matching of a bipartite graph.
  (Very useful in scheduling problems.)



- Maximum bipartite matching $\propto$ Maximum integral flow.
- Given a bipartite graph $G = (L, R, E)$ construct a flow network $G' = (V', E')$ as follows:
  - $V' = L \cup R \cup \{s, t\}$
  - $E' = \{(s, u) : u \in L\} \cup \{(u, v) : u \in L,\ v \in R,\ (u, v) \in E\} \cup \{(v, t) : v \in R\}$
  - $c(u, v) = \begin{cases} 1 & \text{for each } (u, v) \in E' \\ 0 & \text{otherwise} \end{cases}$

Example



Time analysis: $O(VE)$
Because:

1. The maximum matching of any graph has cardinality $O(V)$.
2. The time complexity of the *Ford_Fulkerson()* algorithm is $O(E|f^*|)$, where f^* is the maximum flow.

Reading material

§27.2 "The Ford-Fulkerson method" pp. 587-600
§27.3 "Maximum bipartite matching" pp. 600-604

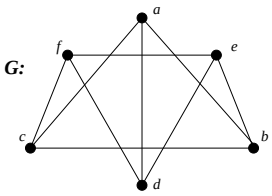
Suggested Exercises

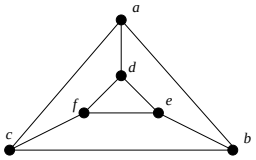
27.2-6
27.3-1, 27.3-3, [27.3-4, 27.3-5]
Problem 27-1 (pp 625)

Planar graphs

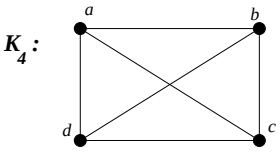
Definition A graph G is *planar* if it can be drawn on the plane with its edges intersecting only at vertices of G.

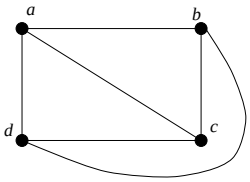
Examples





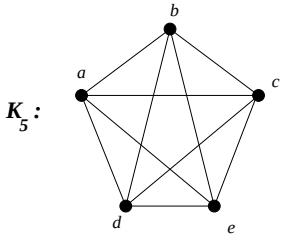
A *plane drawing* of G.



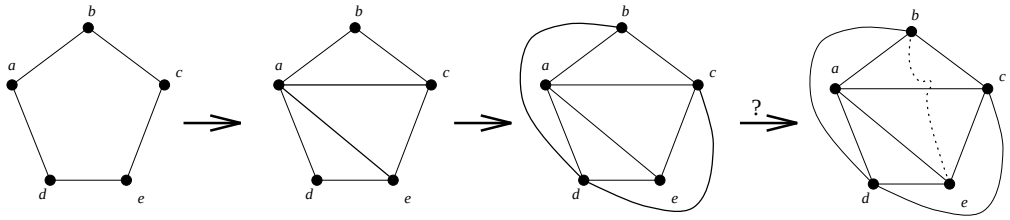


A *plane drawing* of K_4.
K_4 is planar $\Longrightarrow$
$\Longrightarrow$ K_3 and K_2 are also planar.

What about K_5?



- K_5 is not planar.

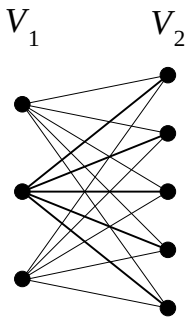


K_5 is not planar $\Longrightarrow$ $K_m,\ m \geq 5$ is not planar.

Definition A *complete bipartite graph* $K_{m,n}$ is a bipartite graph $G = (V_1, V_2, E)$ that satisfies the properties:

1. $|V_1| = m$
2. $|V_2| = n$
3. $degree(v) = \begin{cases} m & \text{if } v \in V_2 \\ n & \text{if } v \in V_1 \end{cases}$

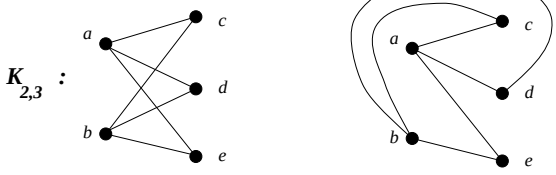
Example $K_{3,5}$



$|E(K_{m,n})| = mn$

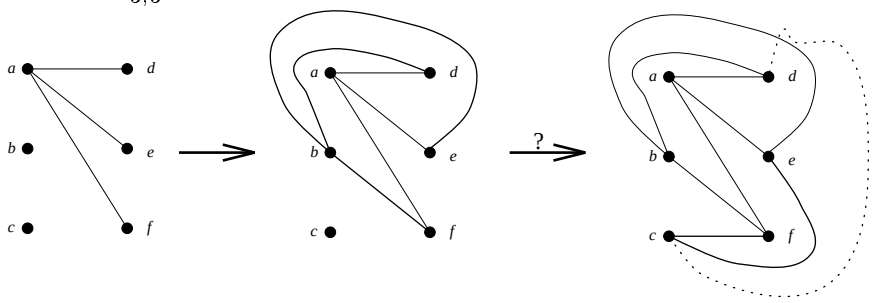
$|E(K_{3,5})| = 3 \cdot 5 = 15$

- Consider $K_{2,3}$



A plane drawing of $K_{2,3}$.

What about $K_{3,3}$?

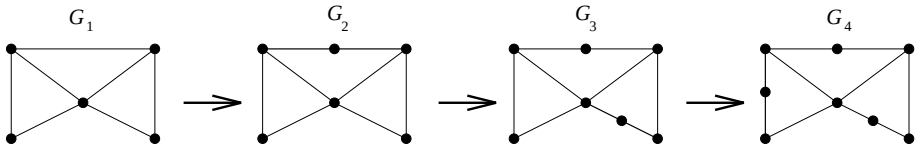


There is no way to draw edge (c, d) without intersecting another edge.
Thus, $K_{3,3}$ is not planar.

$\Longrightarrow$ $K_{m,n},\ m \geq 3,\ n \geq 3$ is not planar.

Definition Let $G = (V, E)$ be a loop-free undirected graph, where $|E| \neq 0$. We obtain an *elementary subdivision* of G when we remove an edge (u, w) from G and replace it by edges (u, v) and (v, w), where v is a new vertex.

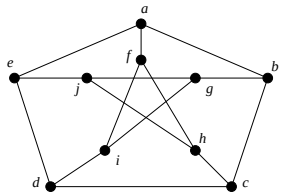
Example



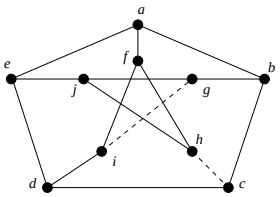
G^4 can be obtained from G^1 by a sequence of 3 elementary subdivisions.

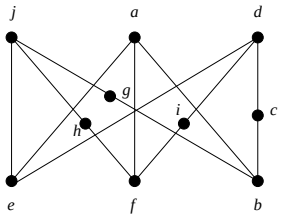
Theorem (Kuratowski's theorem) A graph is non-planar if and only if it contains a subgraph that can be obtained from K_5 or $K_{3,3}$ as a sequence of elementary subdivisions.

What about the following graph? (Petersen's graph)



Redraw the graph induced by the solid edges:





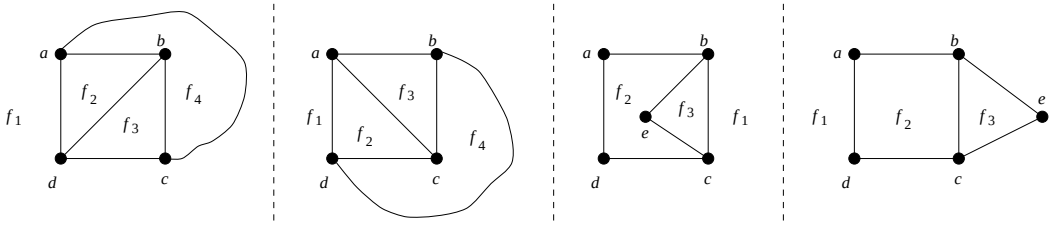
This is a graph obtained from $K_{3,3}$ by a sequence of 4 subdivisions.

$\Longrightarrow$ Petersen's graph is not planar.

Consider some plane drawing of a planar graph G.

Definition A *face* of G is defined to be a region of the plane bounded by edges such that any two points in the region can be connected by a continuous curve that meets no edges or vertices.

Examples



Theorem (Euler's theorem) Let $G = (V, E)$ be a connected planar graph. Let f be the number of faces of G. Then,

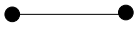
$$|V(G)| - |E(G)| + f = 2$$

Proof (By induction on $|E|$.)

- **Basis** If $|E| = 1$, then G is one of the two following graphs:

(a) 

For (a) we have: $|V| = 1,\ f = 2 \implies 1 - 1 + 2 = 2.\ \surd$

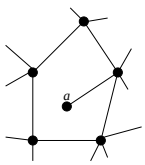
(b) 

For (b) we have: $|V| = 2,\ f = 1 \implies 2 - 1 + 1 = 2.\ \surd$

- **Induction step** Assume the result is true for any connected graph of k edges. Let G be a connected planar graph with $|V|$ vertices, f faces and $|E| = k + 1$ edges.

We will show that the formula $|V| - |E| + f = 2$ holds for G.

Case 1: G contains a vertex a of degree 1

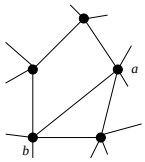


Let H be the graph obtained by deleting a and its incident edge from G.

- H still has f faces.
- $|V(H)| = |V(G)| - 1$
- $|E(H)| = |E(G)| - 1$

Because $|E(H)| = k$, the induction hypothesis holds. Thus,
$|V(H)| - |E(H)| + f = 2 \implies |V(G)| - 1 - |E(G)| + 1 + f = 2 \implies$
$\implies \quad |V(G)| - |E(G)| + f = 2$

Case 2: G does not contain a vertex of degree 1



Let H be the graph obtained by deleting any edge (a, b) from G.

- H has $f - 1$ faces.
- $|V(H)| = |V(G)|$
- $|E(H)| = |E(G)| - 1$

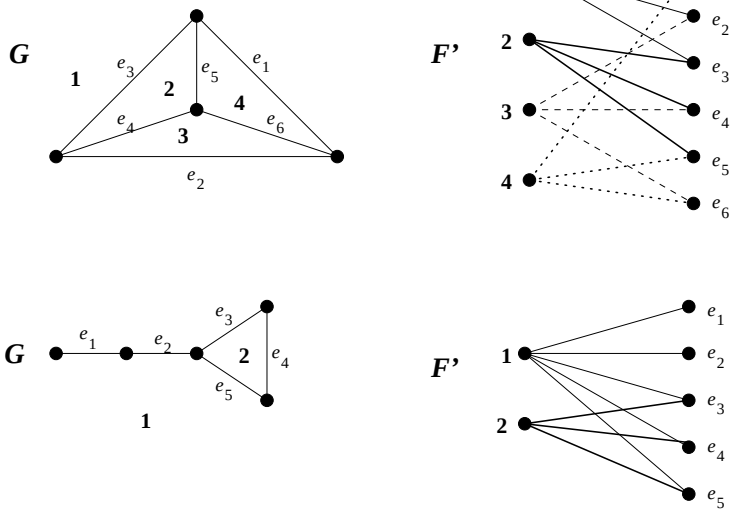
Because $|E(H)| = k$, the induction hypothesis holds. Thus,
$|V(H)| - |E(H)| + f = 2 \implies |V(G)| - |E(G)| + 1 + f - 1 = 2 \implies$
$\implies \quad |V(G)| - |E(G)| + f = 2$

□

Definition Given a plane drawing of a planar graph $G = (V, E)$ that has f faces, the *face-edge incidence graph* $F(V', E')$ *for* G is constructed as follows:

$-V' = \{v_i \mid 1 \leq i \leq f\} \cup \{u_i \mid 1 \leq i \leq |E(G)|\}$
$-E' = \{(v_i, u_j) \mid j$ is an edge at the boundary of face $i\}$

Examples

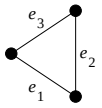


Theorem For any planar graph $G = (V, E)$, $|E| \geq 3$, it holds that $|E| \leq 3|V| - 6$.

Proof

- Assume that G has f faces. Construct the face-edge incident graph H for G.

- Since each face must be incident with *at least* 3 edges,
  $\Longrightarrow \quad |E(H)| \geq 3f \quad (1)$



- Since each edge can be adjacent with *at most* 2 faces,
  $\Longrightarrow \quad |E(H)| \leq 2|E| \quad (2)$

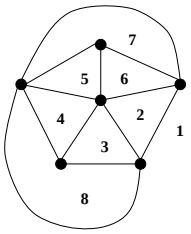
- (1), (2) $\Longrightarrow \quad 2|E| \geq 3f \quad \Longrightarrow \quad f \leq \frac{2|E|}{3} \quad (3)$

- From Euler's formula we have that $|V| - |E| + f = 2 \Longrightarrow$
  $\Longrightarrow \quad |E| = f + |V| - 2 \overset{(3)}{\Longrightarrow} |E| \leq \frac{2|E|}{3} + |V| - 2 \quad \Longrightarrow$
  $\Longrightarrow \quad 3|E| \leq 2|E| + 3|V| - 6 \quad \Longrightarrow \quad |E| \leq 3|V| - 6$

□

- The equality in $|E| \leq 3|V| - 6$ holds when all faces are triangles (*planar triangulated graphs*).

Example



$|E| = 12$
$|V| = 6$
$f = 8$

$|E| = 3|V| - 6 \leftrightarrow 12 = 3 \cdot 6 - 6 = 12\surd$

- The expression $|E| \leq 3|V| - 6$ is very important!

  An immediate implication is that the time analysis of all algorithms that involved E in their time complexity can be improved.

  - **For planar graphs**:

Depth-first search:	$O(V)$
Breadth-first search:	$O(V)$
Bellman-Ford:	$O(V^2)$
Edmonds-Karp:	$O(V^3)$
etc ...	

Theorem K_5 is not planar.

Proof

- K_5 has 5 vertices and $\frac{5 \cdot 4}{2} = 10$ edges.
- If K_5 was planar we must have that:

  $|E(K_5)| \leq 3|V(K_5)| - 6 \quad \Leftrightarrow \quad 10 \leq 3 \cdot 5 - 6 \quad \Leftrightarrow \quad 10 \leq 9$ which is **false**.

□

Question Can we prove with the same method that $K_{3,3}$ is not planar?

Theorem For any planar bipartite graph G it holds that $|E| \leq 2|V| - 4$.

Proof

- Assume that G has f faces. Construct the face-edge incident graph H for G.
- In a bipartite planar graph each face is incident to at least 4 edges.

  $\Longrightarrow \quad |E(H)| \geq 4f \quad (1)$
- Since each edge can be adjacent with *at most* 2 faces,

  $\Longrightarrow \quad |E(H)| \leq 2|E| \quad (2)$
- (1), (2) $\Longrightarrow \quad 2|E| \geq 4f \quad \Longrightarrow \quad f \leq \frac{|E|}{2} \quad (3)$
- From Euler's formula we have that $|V| - |E| + f = 2 \Longrightarrow$

  $\Longrightarrow \quad |E| = f + |V| - 2 \quad \stackrel{(3)}{\Longrightarrow} \quad |E| \leq \frac{|E|}{2} + |V| - 2 \quad \Longrightarrow$

  $\Longrightarrow \quad 2|E| \leq |E| + 2|V| - 4 \quad \Longrightarrow \quad |E| \leq 2|V| - 4$

□

Exercise Prove that $K_{3,3}$ is not planar.

Approximation Algorithms

Consider the problems:

- **3-SATISFIABILITY (3-SAT)**

  Instance: A collection $C = \{c_1, c_2, \ldots, c_m\}$ of clauses on a finite set U of n variables such that $|c_i| = 3$ for $1 \leq i \leq m$.

  Question: Is there a truth assignment for U that satisfies all the clauses in C?

Example

$U = \{x_1, x_2, x_3, x_4, x_5\} \quad C = \{c_1, c_2, c_3, c_4\}$

$c_1 = x_1 + x_2 + \overline{x_3}$

$c_2 = \overline{x_1} + x_4 + \overline{x_5}$

$c_3 = x_1 + \overline{x_4} + x_5$

$c_3 = x_3 + x_4 + x_5$

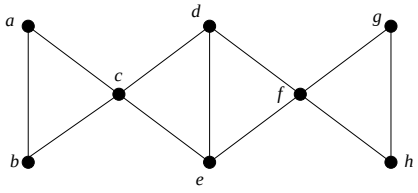
Answer: YES, $(x_1 = T/F, \quad x_2 = T, \quad x_3 = T/F, \quad x_4 = T, \quad x_5 = T)$

- **VERTEX COVER (VC)**

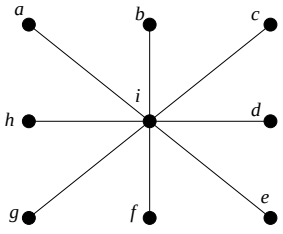
  Instance: A graph $G = (V, E)$ and a positive integer $k \leq |V|$.

  Question: Is there a vertex cover of size at most k for G, that is, *a subset $V' \subseteq V$ such that $|V'| \leq k$ and for each edge $(u, v) \in E$ at least one of u or v belongs to V'*?

Examples



	Answer
$k = 1$	NO
$k = 2$	NO
$k = 3$	NO
$k = 4$	NO
$k = 5$	**YES** (b, c, e, f, h)
$k \geq 6$	YES



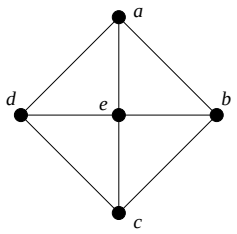
	Answer
$k = 1$	**YES** (i)
$k \geq 2$	YES

- **GRAPH k-COLOURABILITY (CHROMATIC NUMBER)**

Instance: A graph $G = (V, E)$ and a positive integer $k \leq |V|$.

Question: Is G k-colourable, that is, *does there exist a function* $f : V \to \{1, 2, \ldots, k\}$ *such that* $f(u) \neq f(v)$ *whenever* $(u, v) \in E$?

Example



	Answer
$k = 1$	NO
$k = 2$	NO
$k = 3$	**YES** ($a \to 1,\ b \to 2,\ c \to 1,\ d \to 2,\ e \to 3$)
$k \geq 4$	YES

It is "easy" to find an exponential algorithm for each of these problems.

3-SATISFIABILITY (3-SAT)

- Let S be the set of all possible assignments. ($|S| = 2^n$)
- **While** $S \neq \emptyset$ **do**
  - Let α be an assignment in S.
  - $S = S - \{\alpha\}$
  - **If** α satisfies all clauses **then**
    - ANSWER= **TRUE**
    - **Exit** and **Stop**.
- ANSWER= **FALSE**

Time complexity $O(2^n|C|) = O(m2^n)$

VERTEX COVER (VC)

- Let S be the set of all subsets of V of cardinality k. $(|S| = \binom{|V|}{k})$
- **While** $S \neq \emptyset$ **do**
  - Let α be a subset in S.
  - $S = S - \{\alpha\}$
  - **If** α "covers" all edges of E **then**
    - ANSWER= **TRUE**
    - **Exit** and **Stop**.
- ANSWER= **FALSE**

Time complexity $O(\binom{|V|}{k}|E|)$

VERTEX COVER (VC)

A similar algorithm can be obtained.

$\binom{n}{\lambda n} \leq 2^{nH(\lambda)}$ where,
$H(\lambda) = -\lambda \log \lambda - (1-\lambda)\log(1-\lambda),\ 0 \leq \lambda \leq 1$ (*entropy function).*

- A problem is *polynomial-time solvable* if there exists an algorithm to solve it in $O(n^c)$ time, for some constant c.

- The complexity class P is defined as the set of all decision problems that are solvable in polynomial time.

- All problems solvable in polynomial time are considered to be *tractable.*

- All problems that require *super-polynomial* time are considered to be *intractable.*

NP-Complete problems

- A class of problems the status of which is unknown.
- No polynomial-time algorithm has yet been discovered for an NP-Complete problem.
- No one has proven a super-polynomial lower bound for any NP-Complete problem.

$$P \stackrel{?}{=} NP$$

- The most famous open problem in Computer Science since 1971 (Cook).
- We conjecture that $P \neq NP$.

Reason: If a single NP-Complete problem is solved in polynomial time then ALL NP-Complete problems have a polynomial-time solution.

- *"Computers and Intractability. A Guide to the Theory of NP-Completeness"*, Michael R. Garey and David S. Johnson, W.H. Freeman and Company, 1979.

What do we do if a problem is NP-Complete?

- For optimization problems, a solution that is near optimal is desirable.
- An algorithm that returns near-optimal solutions is called an *approximation algorithm.*
- An approximation algorithm has *relative error bound* of $\epsilon(n)$ if

$$\frac{|C - C^*|}{C^*} \leq \epsilon(n)$$

where, C is the solution returned by the approximation algorithm and C^* is the optimal solution.

- We want to derive approximation algorithms which have a *fixed* relative error bound. (For several problems such algorithms do not exist!)

Planar Graph Colouring

Instance: A loop-free planar graph $G = (V, E)$, $|E| \geq 1$.

Question: What is the minimum number of colours needed to colour graph G?

Approx_Planar_Graph_Colouring(G)
- **return**("We need 3 colours")

- This is an approximation algorithm of fixed relative error bound.
  - Every planar graph can be coloured with 4 colours.
  - If $|E| \geq 1$, we always need at least 2 colours.

  $\Longrightarrow \quad |C - C^*| \leq 1 \quad \Longrightarrow \quad \frac{|C-C^*|}{C^*} \leq \frac{1}{2}$

- It might be impossible to colour G with 3 colours!

Note This was a stupid toy-example! The approximation algorithm for the *"minimum vertex cover"* that follows will convince you that this is not "easy staff".

Vertex cover

Instance: A graph $G = (V, E)$.

Output: A *cover* for G of minimum cardinality.

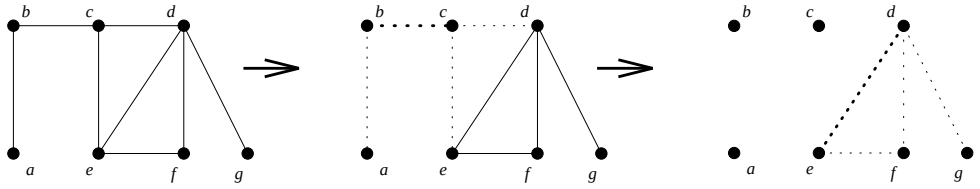
Approx_Vertex_Cover(G)

1. $C \leftarrow \emptyset$
2. $E' \leftarrow E(G)$
3. **while** $E' \neq \emptyset$ **do**
   - Let (u, v) be an arbitrary edge of E'.
   - $C \leftarrow C \cup \{u, v\}$
   - Remove from E' every edge incident on either u or v.

Time complexity $O(E)$

Note In the case where the optimal solution is of odd cardinality, the above approximation algorithm will never return the correct answer.

Example



Select edge (b, c).
$\Longrightarrow \quad C = \{b, c\}$

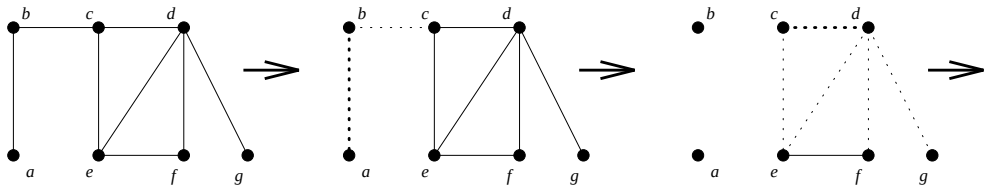
Select edge (d, e).
$\Longrightarrow \quad C = \{b, c, d, e\}$

$\Longrightarrow \quad C = \{b, c, d, e\}$

- Note that $C^* = \{b, e, d\}$.

- If we remove the edges in a different order we might get a "cover" C' with $|C'| > 4$.

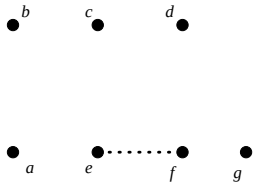
Example



Select edge (a, b).
$\Longrightarrow \quad C' = \{a, b\}$

Select edge (c, d).
$\Longrightarrow \quad C' = \{a, b, c, d\}$

Select edge (e, f).
$\Longrightarrow \quad C' = \{a, b, c, d, e, f\}$



$\Longrightarrow \quad C' = \{a, b, c, d, e, f\}, \ |C'| = 6$

Theorem Let C be the cover produced by *Approx_Vertex_Cover*(G), and C^* be an optimal cover. Then,

$$|C| \leq 2|C^*|$$

Proof

- Let A be the set of edges that were selected by the algorithm.
- The cover C consists of the endpoints of the edges in A.
- **The edges in A are vertex disjoint.**

  $\Longrightarrow$ Any cover will include at least one endpoint for every edge in A.

  $\Longrightarrow \quad |A| \leq |C^*|$

  $\Longrightarrow \quad 2|A| \leq 2|C^*|$

  $\Longrightarrow \quad |C| \leq 2|C^*|$

□

Heuristics

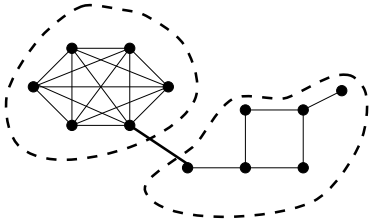
- There are several problems for which we do not know any approximation algorithm. For these problems, we try *heuristic methods* and hope for the best!

Graph Partitioning (Bisection)

Instance: A graph $G = (V, E)$ such that $|V| = 2k$.

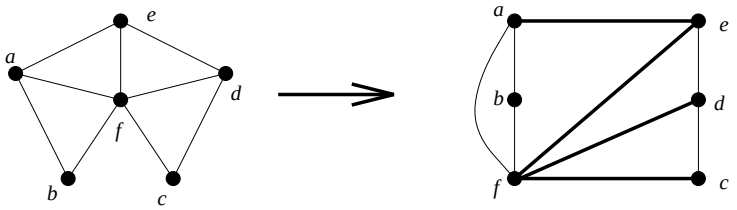
Output: A *partitioning* of V into two sets V_1 and V_2 such that $|V_1| = |V_2| = k$ and the number of edges that have one of their endpoint in V_1 and the other in V_2 is minimized.

Example



$Bisection = 1$

Example



$Bisection = 4$

The Kernighan-Lin algorithm

A simplified version of the algorithm is the following:

Simplified_Kernighan-Lin(G)

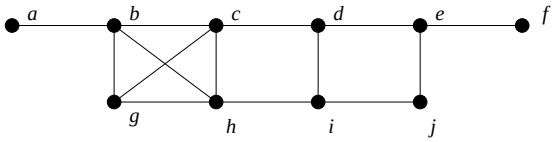
1. Start with an arbitrary initial partition V_1, V_2.
2. **while** there are sets $A \subset V_1$ and $B \subset V_2$ with $|A| = |B|$ such that when exchanged the bisection reduces **do**

$V_1 \leftarrow V_1 - A \cup B$

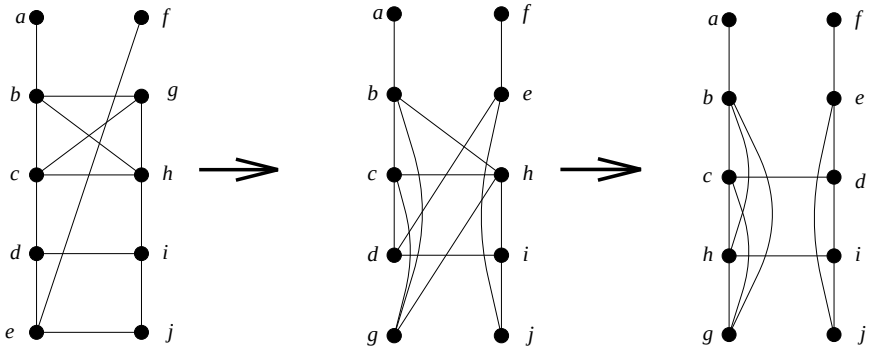
$V_2 \leftarrow V - V_1$

- The resulting partition depends on the initial partition.
- The resulting partition depends on the order of the exchanges.
- The algorithm works in practice. It is being used since 1971.
- A similar (popular and fast) algorithm is due to Fiduccia and Matheyses (1982).

Example



Let $V_1 = \{a, b, c, d, e\}$
$V_2 = \{f, g, h, i, j\}$



Bisection $= 7$
Exchange g with e.

Bisection $= 5$
Exchange h with d.

Bisection $= 2$

- In this example, we got the optimal solution. We cannot be that lucky all the time!

Reading Material

§37 "Approximation Algorithms" pp. 964–966.

§37.1 "The vertex-cover problem" pp. 966–968.

Additional Reading (optional)

§36 "NP-Completeness" pp. 916-946.

Suggested Exercises

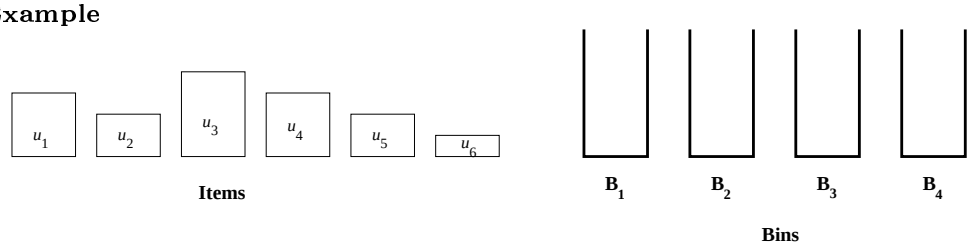
37.1-1, 37.1-2, 37.1-3.

Bin-Packing

Problem Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of *items* and *size-function* $s(u) \in [0 \ldots 1]$ for each item $u \in U$, find a partition of U into disjoint subsets $U_1, U_2, \ldots, U_k$ such that the sum of the sizes of the items in each (bin) U_i is at most 1, and k is as small as possible.

- View each U_i as a *unit-capacity bin.*

Example



- Let $OPT(I)$ denote the number of bins in an optimal solution of the bin-packing problem for instance I. Then,

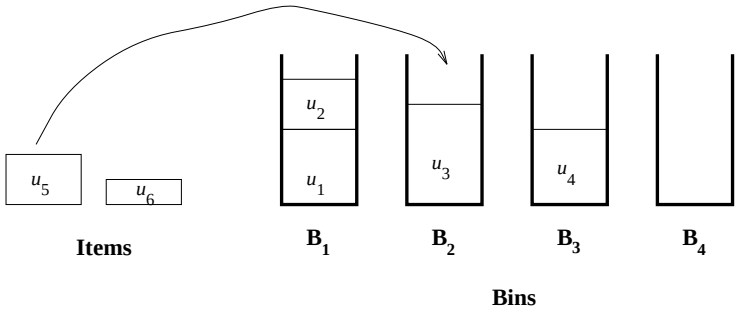
$$OPT(I) \geq \left\lceil \sum_{i=1}^{n} s(u_i) \right\rceil$$

- **First-Fit approach**

First-Fit(I)

1. Start with an infinite sequence $B_1, B_2, \ldots$ of unit-capacity bins, all of which are empty.
2. **For** $i = 1$ **to** n **do**
   Place item u_i into the bin of smallest index in which it can fit.

Example



Denote by $FF(I)$ the number of bins required by algorithm *First-Fit*() for instance I.

Theorem $FF(I) \leq 2 \cdot OPT(I)$

Proof

- We have that $FF(I) < \left\lceil 2 \sum_{i=1}^{n} s(u_i) \right\rceil$ (1)
  since there can be at most 1 nonempty bin with total contents $\frac{1}{2}$ or less.
- We also have that $OPT(I) \geq \left\lceil 2 \sum_{i=1}^{n} s(u_i) \right\rceil$ (2)
- (1), (2) $\Longrightarrow$ $FF(I) \leq 2 \cdot OPT(I)$

□

Theorem i) $FF(I) \leq \frac{17}{10} \cdot OPT(I) + 2$, for every instance I.

ii) There exist instances with $OPT(I)$ arbitrarily large such that $FF(I) \geq \frac{17}{10}(OPT(I) - 1)$

□

- **First-Fit-Decreasing approach**

  First-Fit-Decreasing(I)

  1. Sort the items in decreasing order of size.

  2. Apply *First-Fit*() in the sequence of items obtained from step 1.

- Denote by $FFD(I)$ the number of bins required by algorithm *First-Fit-Decreasing*() for instance I.

Theorem i) $FFD(I) \leq \frac{1}{9} \cdot OPT(I) + 4$, for every instance I.

ii) There exist instances with $OPT(I)$ arbitrarily large such that $FFD(I) = \frac{11}{9} \cdot OPT(I)$

□

Pseudo-Polynomial time algorithms

- **The "Partition"problem**

  Instance: A finite set A and a *size* $s(a) \in Z^+$ for each $a \in A$.

  Output: Is there a subset $A' \subset A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

Example

$A = \{a_1, a_2, a_3, a_4, a_5\}$

$s(a_1) = 1$
$s(a_2) = 9$
$s(a_3) = 5$
$s(a_4) = 3$
$s(a_5) = 8$

Let $A' = \{a_3, a_5\}$

$\Longrightarrow \sum_{a \in A'} s(a) = a_3 + a_5 = 5 + 8 = 13$

$\Longrightarrow \sum_{a \in A - A'} s(a) = a_1 + a_2 + a_4 = 1 + 9 + 3 = 13$

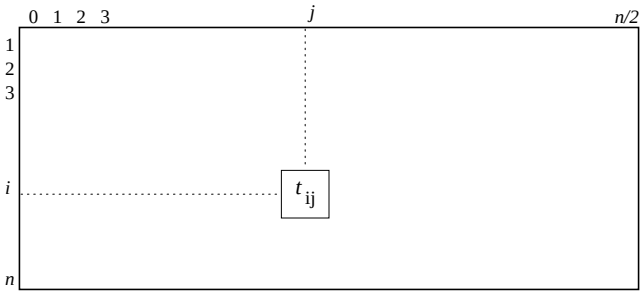
Let $B = \sum_{a \in A} s(a)$

- If B is odd, obviously no set A' satisfies the requirements of the problem.
- For $1 \leq i \leq n,\ 0 \leq j \leq \frac{B}{2}$ consider the statement:

  "There is a subset of $1, a_2, \ldots, a_i\}$ for which the sum of the item sizes is exactly j."

  Denote by $t(i, j)$ the truth value of this statement.

- The values for $t(i, j)$ can be arranged in a table of dimensions $n \times (\frac{B}{2} + 1)$.



(In our example, $n = 5$ and $B = \frac{26}{2} = 13$.)

- We can fill the table as follows:

  Row 1 $t(1,j) = \mathbf{T} \iff j = 0$ or $j = s(a_1)$

  Row i $t(i,j) = \mathbf{T} \iff t(i-1,j) = \mathbf{T}$ or $(s(a_i) \leq j$ and $t(i-1, j-s(a_i)) = \mathbf{T})$

In our example, $s(a_1) = 1$, $s(a_2) = 9$, $s(a_3) = 5$, $s(a_4) = 3$, $s(a_5) = 8$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	**T**	**T**												
2														
3														
4														
5														

Time complexity $O(nB)$

Fact: The *"partition"* problem is NP-complete.

Question Did we just prove that $P = NP$?

Answer NO!

- We are after a polynomial *to the size of the input* algorithm.

- Each integer $s(a_i)$ can be represented by a bit-string of size $O(\log s(a_i))$. Thus, the size of the input I has length

$$length(I) \;=\; \sum_{i=1}^{n} \log s(a_i) \;\leq\; \sum_{i=1}^{n} \log B \;=\; O(n \log B)$$

- nB is not bounded by a polynomial function of $n \log B$.

- Pseudo-polynomial algorithms are useful in practice.
  - In *scheduling problems*, numbers represent task-lengths and it is reasonable to assume that they are small.
  - In problems where *numbers represent empirically measured quantities*, limits on the precision of measurement limit their range.
  - In applications which no bounds on numbers are imposed, pseudo-polynomial algorithms display *"exponential behaviour"* only when confronted with instances containing *"exponentially large"* numbers.

Reading Material

"Computers and Intractability. A Guide to the Theory of NP-Completeness", Michael R. Garey and David S. Johnson, W.H. Freeman and Company, 1979.

"Bin-packing" pp. 123–128.

"Pseudo-polynomial algorithms" pp. 90–92.

To finish this module, remember:

- There are

LIES,

BIG LIES, and

BIG "O" NOTATION.

- ALWAYS consider the hidden constant in the asymptotic complexity of an algorithm you are apply.
- ALWAYS consider the properties that your input possesses.