

Summary of COMP523 Advanced Algorithm

May 8, 2023

Chapter 1

Symmetry Notation

1.1 Asymptotic Notation

Asymptotic notation is a way of describing the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, asymptotic notation is frequently used to describe the running time or space usage of an algorithm.

- O -notation: $f(n) = O(g(n))$ if there exist constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- Ω -notation: $f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
- Θ -notation: $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.
- o -notation: $f(n) = o(g(n))$ if for any constant $c > 0$, there exists a constant n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
- ω -notation: $f(n) = \omega(g(n))$ if for any constant $c > 0$, there exists a constant n_0 such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

1.2 Comparing Functions

1.2.1 Transitivity

- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ implies $f(n) = \Omega(h(n))$.
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$.

For example, $n^2 = O(n^3)$ and $n^3 = O(n^4)$ implies $n^2 = O(n^4)$.

1.2.2 Reflexivity

- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.
- $f(n) = \Theta(f(n))$.

For example, $n^2 = O(n^2)$.

1.2.3 Symmetry

- $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
- $f(n) = \Omega(g(n))$ implies $g(n) = \Omega(f(n))$.
- $f(n) = \Theta(g(n))$ implies $g(n) = \Theta(f(n))$.
- $f(n) = o(g(n))$ implies $g(n) = \omega(f(n))$.
- $f(n) = \omega(g(n))$ implies $g(n) = o(f(n))$.

For example, $n^2 = O(n^3)$ implies $n^3 = \Omega(n^2)$.

1.2.4 Transpose Symmetry

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.
- $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.

For example, $n^2 = O(n^3)$ if and only if $n^3 = \Omega(n^2)$.

1.2.5 sum and maximum

$$f_1(n) + f_2(n) + \cdots + f_k(n) = \Theta(\max(f_1(n), f_2(n), \dots, f_k(n)))$$

where k is a constant positive integer.

Let $f_j(n) = j$, $k = n$, then

$$f_1(n) + f_2(n) + \cdots + f_k(n) = n(n+1)/2 = \Theta(n^2)$$

1.2.6 Running time hierarchy

- logarithmic: $O(\log n)$
- linear: $O(n)$
- $n \log n$: $O(n \log n)$
- quadratic: $O(n^2)$
- polynomial: $O(n^k)$
- exponential: $O(c^n)$
- constant: $O(1)$
- superconstant: $\omega(1)$
- sublinear: $o(n)$
- superlinear: $\omega(n)$
- superpolynomial: $\omega(n^k)$
- subexponential: $o(c^n)$

1.3 Expect of algorithms

Correctness: An algorithm is correct if it halts with the correct output for every input instance.

Termination: An algorithm is terminating if it halts for every input instance.

Efficiency: An algorithm is efficient if it halts with the correct output for every input instance and runs in polynomial time.

Chapter 2

Recursion and Divide and Conquer techniques

2.1 Finding Majority in array

The pseudocode of the algorithm is shown in Algorithm 2.1.

Algorithm 1 Finding Majority in array

```
1: procedure MAJORITY( $A$ )
2:    $n \leftarrow$  length of  $A$ 
3:   if  $n = 0$  then
4:     return  $-1$ 
5:   end if
6:   if  $n = 1$  then
7:     return  $A[1]$ 
8:   end if
9:   if  $n \neq 1$  and  $n$  is odd then
10:
11:   end if
12:   Array  $B$  of size  $n/2$ 
13:   set  $j=0$ 
14:   for  $i = 1$  to  $n/2$  do
15:     if  $A[2i - 1] = A[2i]$  then
16:        $B[j] \leftarrow A[2i - 1]$ 
17:        $j \leftarrow j + 1$ 
18:     end if
19:   end for
20:    $m \leftarrow$  MAJORITY( $B$ )
21:    $count \leftarrow 0$ 
22:   for  $i = 1$  to  $n$  do
23:     if  $A[i] = m$  then
24:        $count \leftarrow count + 1$ 
25:     end if
26:   end for
27:   if  $count > n/2$  then
28:     return  $m$ 
29:   else
30:     return  $-1$ 
31:   end if
32: end procedure
```

Correctness:

Lemma: If A has a majority element, then the majority element of A is also the majority element of B .

Base case: $n = 1$, the majority element is $A[1]$.

Induction hypothesis: Assume that the lemma is true for $n = k$, we will prove that the lemma is true for $n = k + 1$.

Induction step: If A has a majority element, then the majority element of A is also the majority element of B .

Case 1 (A has a majority element m): Then by the lemma, it is also the majority element of B . Then m appears more than $k/2$ times in B . Then m appears more than $(k + 1)/2$ times in A .

Case 2 (A has no majority element): Then B has no majority element. Then A has no majority element.

Proof the lemma:

proof by contradiction. Assume that A has a majority element m and B has a majority element m' , but $m \neq m'$.

Let x be the numbers of occurrence of m in A .

Let y be the numbers of occurrence of m' in B .

Then $2y$ times from pairs that are represented in B by a value different from m' , and $x - 2y$ times, since each occurrence of m in A that is not paired with another occurrence of m in A is paired with an occurrence of m' in B .

In total, this gives $2y + x - 2y = x$ occurrences of m in A , which is a contradiction.

Running time:

Recursive formula for the running time:

$$T(n) \leq T(n/2) + cn$$

where c is a constant.

The solution to the recurrence is $T(n) = O(n)$.

2.2 Searching in logarithmic time

Searching faster with BinarySearch.

It is a particular case of the divide-and-conquer paradigm.

Input: A sorted array A of n elements and a value x .

Output: An index i such that $A[i] = x$ or the special value -1 if x does not appear in A .

Pseudocode is shown in Algorithm 2.2.

Algorithm 2 BinarySearch

```

1: procedure BINARYSEARCH( $x, i, j$ )
2:   if  $i = j$  then
3:     if  $A[i] = x$  then
4:       return  $i$ 
5:     else
6:       return  $-1$ 
7:     end if
8:   else
9:     if  $x = A[\lfloor (i + j)/2 \rfloor]$  then
10:      return  $\lfloor (i + j)/2 \rfloor$ 
11:    else if  $x < A[\lfloor (i + j)/2 \rfloor]$  then
12:      return BINARYSEARCH( $x, i, \lfloor (i + j)/2 \rfloor$ )
13:    else
14:      return BINARYSEARCH( $x, \lfloor (i + j)/2 \rfloor + 1, j$ )
15:    end if
16:  end if
17: end procedure

```

Running time:

The number of comparisons performed by BinarySearch is:

$$T(n) \leq T(n/2) + 4$$

Keep calculate:

$$\begin{aligned}
T(n) &\leq T(n/2) + 4 \\
&\leq T(n/4) + 4 + 4 \\
&\leq T(n/8) + 4 + 4 + 4 \\
&\leq T(n/2^k) + 4k \\
&\leq T(n/2^{\log(n-1)}) + 4\log(n-1) \\
&= T(2) + 4(\log n - 1) \\
&\leq 4\log n - 4 \\
&= 4\log n
\end{aligned}$$

proof $T(n) \leq 4\log n$:

Base case: $n = 1, T(1) = 0 \leq 4\log 1 = 0$.

Induction hypothesis: Assume that the lemma is true for $n = k$, we will prove that the lemma is true for $n = k + 1$.

Induction step: $T(k + 1) \leq 4\log(k + 1)$.

$$\begin{aligned}
T(k + 1) &\leq T(k/2) + 4 \\
&\leq 4\log(k/2) + 4 \\
&= 4\log k - 4 + 4 \\
&= 4\log k \\
&\leq 4\log(k + 1)
\end{aligned}$$

Memory usage:

The memory usage of BinarySearch is:

$$M(n) = O(\log n)$$

Comparing BinarySearch and LinearSearch:

$$\begin{aligned}
T_{\text{BinarySearch}}(n) &= O(\log n) \\
T_{\text{LinearSearch}}(n) &= O(n) \\
T_{\text{BinarySearch}}(n) &= O(\log n) < O(n) = T_{\text{LinearSearch}}(n) \\
M_{\text{BinarySearch}}(n) &= O(\log n) < O(1) = M_{\text{LinearSearch}}(n)
\end{aligned}$$

2.3 Running time of Divide and Conquer algorithms

The Master Theorem:

Suppose that $T(n)$ satisfies the recurrence:

$$T(n) \leq aT(n/b) + cn^d$$

where $a \geq 1, b > 1, c > 0$ and $d \geq 0$ are constants.

Then $T(n)$ has the following asymptotic bounds:

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

This theorem is useful for solving recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and $f(n)$ is an asymptotically positive function.

Example:

$$T(n) = 8T(n/2) + 100n^2$$

$a = 8$, $b = 2$, $f(n) = 100n^2$, $d = 2$, $\log_b a = \log_2 8 = 3$.

$d = 2 < \log_b a = 3$, so $T(n) = O(n^{\log_b a}) = O(n^3)$.

2.4 Finding pair of points closest to each other

Input: A set P of n points in the plane.

Output: The pair of points in P that are closest to each other.

Pseudocode is shown in Algorithm 2.4. **Running time:**

Algorithm 3 ClosestPair

```

1: procedure CLOSESTPAIR( $P_1, \dots, P_n$ )
2:   Construct  $P_x$  and  $P_y$ .  $P_x$  is sorted by  $x$ -coordinate,  $P_y$  is sorted by  $y$ -coordinate.
3:   return CLOSESTPAIRREC( $P_x, P_y$ )
4: end procedure

```

Algorithm 4 ClosestPairRec

```

1: procedure CLOSESTPAIRREC( $P_x, P_y$ )
2:   if  $|P_x| = |P_y| \leq 3$  then
3:     For each pair of points  $(P_i, P_j)$ , compute  $d(P_i, P_j)$ 
4:     return the pair of points with the smallest distance
5:   end if
6:   Construct  $Q_x, Q_y, R_x$  and  $R_y$ .
7:    $(l_1, l_2) = \text{CLOSESTPAIRREC}(Q_x, Q_y)$ 
8:    $(r_1, r_2) = \text{CLOSESTPAIRREC}(R_x, R_y)$ 
9:    $\delta = \min\{d(l_1, l_2), d(r_1, r_2)\}$ 
10:   $x^* =$  the largest  $x$ -coordinate in  $Q_x$ 
11:   $L = \{(x, y) : x = x^*\}$ 
12:   $S = \{p \in P : p \in L \text{ and } p \text{ is within } \delta \text{ of } L\}$ 
13:  Construct  $S_v$ 
14:  for  $p \in S$  do
15:    Let  $q$  be the point in  $S_v$  closest to  $p$ 
16:    if  $d(p, q) < \delta$  then
17:       $\delta = d(p, q)$ 
18:       $(s_1, s_2) = (p, q)$ 
19:    end if
20:  end for
21:  if  $d(s_1, s_2) < \min\{d(l_1, l_2), d(r_1, r_2)\}$  then
22:    return  $(s_1, s_2)$ 
23:  end if
24:  if  $d(l_1, l_2) < d(r_1, r_2)$  then
25:    return  $(l_1, l_2)$ 
26:  else
27:    return  $(r_1, r_2)$ 
28:  end if
29: end procedure

```

$$T(n) \leq 2T(n/2) + O(n \log n) = O(n \log n)$$

Example:

Chapter 3

Graph Algorithms

3.1 Graph Definitions

Graph: A graph G consists of a set V of vertices and a set E of edges, where each edge is associated with a pair of vertices.

Directed Graph: A directed graph G consists of a set V of vertices and a set E of directed edges, where each directed edge is associated with an ordered pair of vertices.

Undirected Graph: An undirected graph G consists of a set V of vertices and a set E of undirected edges, where each undirected edge is associated with an unordered pair of vertices.

Neighbours of a vertex v : Set of vertices that are connected to v by an edge.

Degree of a vertex v : number of neighbours of v , denoted by $deg(v)$.

Path: A sequence of (non-repeating) nodes with consecutive nodes being connected by an edge.
length = node count - 1 = edge count.

Distance between two nodes: The number of edges in the shortest path between the two nodes.

Graph diameter: The maximum distance between any two nodes in the graph.

Lines, cycles, trees and cliques:

Line: A graph with n vertices and $n - 1$ edges.

Cycle: A graph with n vertices and n edges.

cliques: A graph with n vertices and $n(n - 1)/2$ edges.

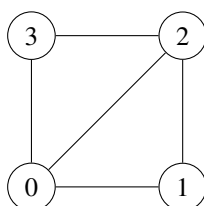
Tree: A graph with n vertices and $n - 1$ edges.

Graph representations:

Adjacency matrix: A $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge between i and j , and $A_{ij} = 0$ otherwise.

examples of adjacency matrices:

Given the following graph:



The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency matrix for directed graphs: A $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge from i to j , and $A_{ij} = 0$ otherwise.

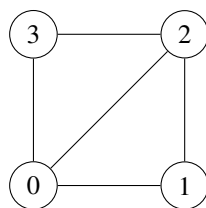
examples of adjacency matrices for directed graphs:
 Given the following graph:



The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

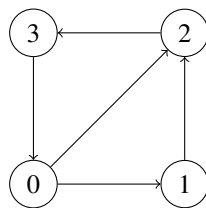
Adjacency list: A list of lists, where the i th list contains the neighbours of vertex i .
 Given the following graph:



The adjacency list is:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & \\ 0 & 1 & 3 \\ 0 & 2 & \end{bmatrix}$$

Adjacency list for directed graphs: A list of lists, where the i th list contains the neighbours of vertex i .
 Given the following graph:



The adjacency list is:

$$\begin{bmatrix} 1 & 2 \\ 2 & \\ 3 & \\ 0 & \end{bmatrix}$$

Adjacency matrix vs adjacency list:

Adjacency matrix	Adjacency list
$O(1)$ to check if there is an edge between i and j	$O(\min(\deg(i), \deg(j)))$ to check if there is an edge between i and j
$O(n)$ to find the neighbours of i	$O(\deg(i))$ to find the neighbours of i
$O(n^2)$ space	$O(n + m)$ space

3.2 Depth-first search

Depth-first search: A graph search algorithm that explores the neighbours of a vertex before exploring the neighbours of its neighbours.

example of depth-first search:



The depth-first search sequence is:

0, 1, 2, 3, 5, 4

Depth-first search algorithm:

Algorithm 5 Depth-first search algorithm

```
1: procedure DFS( $G, v$ )
2:   for  $e \in V$  do
3:     if  $e$  is unexplored then
4:        $u = \text{head of } e$ 
5:       if  $u$  is unexplored then
6:          $e$  is a tree edge
7:         DFS( $G, u$ )
8:       else
9:          $e$  is a back edge
10:      end if
11:    end if
12:  end for
13: end procedure
```

Running time of depth-first search: $O(n + m)$

3.3 Breadth-first search

Breadth-first search: A graph search algorithm that explores the neighbours of a vertex before exploring the neighbours of its neighbours.

example of breadth-first search:



The breadth-first search sequence starting from vertex 0 is 0, 1, 2, 3, 4, 5.

Breadth-first search algorithm:

Algorithm 6 Breadth-first search algorithm

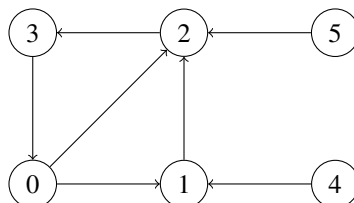
```
1: procedure BFS( $G, s$ )
2:   initial empty list  $L$ 
3:    $L \leftarrow s$ 
4:    $i \leftarrow 0$ 
5:   while  $L[i] \neq \emptyset$  do
6:      $L_{i+1} \leftarrow \text{emptylist}$ 
7:     for  $v \in L[i]$  do
8:       for edges  $(e)$  incident to  $v$  do
9:         if  $e$  is unexplored then
10:            $w \leftarrow$  the other end of  $e$ 
11:           if  $w$  is unexplored then
12:             label  $e$  as a tree edge
13:             add  $w$  to  $L_{i+1}$ 
14:           else
15:             label  $e$  as a cross edge
16:           end if
17:         end if
18:       end for
19:     end for
20:      $i \leftarrow i + 1$ 
21:   end while
22: end procedure
```

Running time of breadth-first search: $O(n + m)$

3.4 Strong Connectivity

Directed graph: A graph where the edges have a direction.

Examples:



DFS and BFS on directed graphs:

Very similar to undirected graphs, except that we only consider edges that go out of a vertex.

Running time is $O(n + m)$

For example graph above the DFS sequence is 0, 1, 2, 3.

The BFS sequence is 0, 1, 2, 3.

3.4.1 Connectivity

Weak connectivity: If we ignore the direction for all edges, there would be a path from any vertex to any other vertex.

Strong Connectivity: For every two nodes u and v , there is a path from u to v and a path from v to u .

3.4.2 Mutual Reachability

Two nodes u and v are mutually reachable if there is a path from u to v and a path from v to u .

Strong connectivity: For every pair of nodes u and v , these two nodes are mutually reachable.

Transitivity: If u is mutually reachable with v and v is mutually reachable with w , then u is mutually reachable with w .

3.4.3 Testing strong connectivity

Algorithm 7 Testing strong connectivity

```
1: procedure TESTSTRONGCONNECTIVITY( $G$ )
2:   define  $G^R$  to be the graph with the same vertices as  $G$  but with all edges reversed
3:   Select a node  $s$  in  $G$ 
4:   BFS( $G, s$ ), BFS( $G^R, s$ )
5:   for each node  $v$  do
6:     if  $v$  is unexplored in either BFS then
7:       return False
8:     end if
9:   end for
10:  return True
11: end procedure
```

3.5 Testing bipartiteness

Bipartite graph: A graph $G = (V, E)$ is bipartite if and only if the vertices can be partitioned into two sets V_1 and V_2 such that every edge has one end in V_1 and the other end in V_2 .

A Graph $G = (V, E)$ is bipartite if and only if it has no odd cycles. (odd cycle: a cycle with odd number of edges)

Testing bipartiteness:

Given a graph $G = (V, E)$, we want to test if G is bipartite.

Given a graph $G = (V, E)$, decide if it is 2-colourable.

Given a graph $G = (V, E)$, decide if it has an odd cycle.

Colouring the nodes It is quite familiar with BFS:

Algorithm 8 Colouring the nodes

```
1: procedure COLOURING( $G, s$ )
2:   initial empty list  $L$ 
3:   initial empty list  $C$ 
4:    $L \leftarrow s$ 
5:    $C[s] \leftarrow red$ 
6:    $i \leftarrow 0$ 
7:   while  $L[i] \neq \emptyset$  do
8:      $L_{i+1} \leftarrow emptylist$ 
9:     for  $v \in L[i]$  do
10:      for edges ( $e$ ) incident to  $v$  do
11:        if  $e$  is unexplored then
12:           $w \leftarrow$  the other end of  $e$ 
13:          if  $w$  is unexplored then
14:            label  $e$  as a tree edge
15:            add  $w$  to  $L_{i+1}$ 
16:            if  $i + 1$  is odd then
17:               $C[w] \leftarrow green$ 
18:            else
19:               $C[w] \leftarrow red$ 
20:            end if
21:          else
22:            label  $e$  as a cross edge
23:            if  $C[v] = C[w]$  then
24:              return False
25:            end if
26:          end if
27:        end if
28:      end for
29:    end for
30:     $i \leftarrow i + 1$ 
31:  end while
32:  for  $e(v, w) \in G$  do
33:    if  $C[v] = C[w]$  then
34:      return False
35:    end if
36:  end for
37:  return True
38: end procedure
```

Running time of colouring the nodes: $O(n + m)$

Correctness of colouring the nodes:

Proof by contradiction.

Suppose that G is not bipartite.

Then G has an odd cycle.

Suppose to the contrary that the algorithm return True.

That means that the algorithm did not detect the odd cycle.

3.6 DAGs and Topological Ordering

DAG: A directed acyclic graph (DAG) is a directed graph with no directed cycles.
examples of DAGs:



Topological ordering: Given a graph $G = (V, E)$, a topological ordering of G is an ordering of the nodes u_1, u_2, \dots, u_n such that for every edge (u_i, u_j) , we have $i < j$.

Intuitively, a topological ordering is an ordering of the nodes such that every edge goes from left to right.

example of topological ordering based on given graph above:

3, 0, 1, 2, 4, 5

Topological ordering implies DAG:

- If G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering u_1, u_2, \dots, u_n but G also has a cycle C .
- Let u_j be the smallest element of C in the topological ordering.
- Let u_i be its predecessor in C .
- u_i must appear before u_j in the topological ordering.
- This contradicts the fact that u_j is the smallest element of C in the topological ordering.

DAG implies topological ordering:

Proof by induction: Base case: If G has one or two nodes, then G has a topological ordering.

Induction steps: Assume that a DAG up to k nodes has a topological ordering (induction hypothesis). we will prove that a DAG with $k + 1$ nodes has a topological ordering.

- By our lemma, there is at least one source node in G , and let u be the node.
- Put u at the beginning of the topological ordering.
- Consider the graph G' , obtained by G by removing u and its incident edges.
- G' is a DAG with k nodes.
- It has a topological ordering u_1, u_2, \dots, u_k by the induction hypothesis.
- Append this ordering to u to get a topological ordering of G .

Here is the algorithm:

Algorithm 9 Topological Sorting

```

1: procedure TOPOLOGICALSORTING( $G$ )
2:   find a source vertex  $u$ 
3:   set  $u$  as the first element of the topological ordering
4:    $G' \leftarrow G$  with  $u$  and its incident edges removed
5:    $L \leftarrow$  TOPOLOGICALSORTING( $G'$ )
6:   append  $L$  to  $u$ 
7: end procedure

```

Running time of the algorithm is $O(n^2)$

Modified Topological Sorting:

Running time of the algorithm is $O(n + m)$

Algorithm 10 Modified Topological Sorting

```
1: procedure MODIFIEDTOPOLOGICALSORTING( $G$ )
2:    $L \leftarrow \text{emptylist}$ 
3:    $S \leftarrow$  set of all source vertices
4:   while  $S \neq \emptyset$  do
5:     remove a vertex  $u$  from  $S$ 
6:     append  $u$  to  $L$ 
7:     for each edge  $(u, v)$  do
8:       remove edge  $(u, v)$  from  $G$ 
9:       if  $v$  is a source vertex then
10:        add  $v$  to  $S$ 
11:       end if
12:     end for
13:   end while
14:   if  $G$  has edges then
15:     return  $G$  has a cycle
16:   else
17:     return  $L$ 
18:   end if
19: end procedure
```

3.7 Finding strongly connected components

connected components: A connected component of an undirected graph is subgraph of the graph where any two nodes are connected by a path.

strongly connected components: A strongly connected component of a directed graph is a subgraph of the graph where any two nodes are mutually reachable.(mutually reachable: there is a path from u to v and a path from v to u)

Finding strongly connected components:

Kosaraju's algorithm:

Algorithm 11 Kosaraju's algorithm

```
1: procedure KOSARAJU( $G$ )
2:   Initialise stack  $S$ 
3:   Select a arbitrary node  $s$ 
4:   DFS_tree=DFS( $G, s$ )
5:    $S \leftarrow$  nodes in DFS_tree
6:    $G^R \leftarrow$  nodes in order of  $S$ 
7:   DFS( $G^R, s$ )
8:   return the nodes in the DFS tree
9: end procedure
```

Running time of Kosaraju's algorithm: $O(n + m)$

Correctness of Kosaraju's algorithm:

- Define a meta-graph of G , called $G^{SCC} = (V^{SCC}, E^{SCC})$.
- Supposed that G has strongly connected components (SCCs) C_1, C_2, \dots, C_k , for some k .
- $V^{SCC} = \{C_1, C_2, \dots, C_k\}$ contains some of the SCCs of G .
- There is an edge (C_i, C_j) in E^{SCC} if G contains a directed edge (x, y) such that $x \in C_i$ and $y \in C_j$, crossing different components.

Examples:



The SCCs are $\{0, 1, 2, 3\}$ and $\{4, 5\}$.
 The meta-graph is:



Chapter 4

Greedy Algorithms

The greedy approach:

- The goal is to find a global solution to a problem.
- The solution will be built up in small consecutive steps.
- For each step, we choose the best option available to us at that moment.

4.1 Interval Scheduling

Interval Scheduling:

A set of requests $R = \{1, 2, \dots, n\}$.

- Each request i has a start time s_i and a finish time f_i .
- Alternative view: every request is an interval $[s_i, f_i]$.

Two requests i and j are compatible if $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap.

Goal: Find a maximum-size subset of compatible requests.

Example:

Interval scheduling.

- Job j starts at s_j and finishes at f_j .
- Two jobs **compatible** if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Figure 4.1: Interval Scheduling

Interval Scheduling Algorithm:

Algorithm 12 Interval Scheduling Algorithm

```
1: procedure INTERVALSCHEDULING( $[s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$ )
2:    $R$  is the set of requests
3:    $A \leftarrow \emptyset$ 
4:   while  $R \neq \emptyset$  do
5:     select a request  $i$  in  $R$  with the smallest finishing time
6:     add  $i$  to  $A$ 
7:     remove all requests from  $R$  that are incompatible with  $i$ 
8:   end while
9:   return  $A$ 
10: end procedure
```

Running time of Interval Scheduling Algorithm: $O(n \log n)$

Correctness of Interval Scheduling Algorithm: Since the algorithm always selects the request with the smallest finishing time, it is clear that the algorithm will always select a compatible request.

Arguing optimality:

4.2 Minimum Spanning Trees

Consider a connected graph $G = (V, E)$, such that each edge $e = (v, w)$ of E , there is an associated cost c_e .

Goal: Find a spanning tree T of E so that the graph $G' = (V, T)$ has minimum cost.

Example:



Greedy approach 1:

- Start with an empty set of edges T .
- Repeat until T forms a spanning tree:
 - Select an edge e of minimum cost.
 - If $T \cup \{e\}$ does not contain a cycle, then add e to T .

krukals algorithm:

Algorithm 13 Krukals algorithm

```
1: procedure KRUKALS( $G$ )
2:    $T \leftarrow \emptyset$ 
3:   while  $T$  is not a spanning tree do
4:     select an edge  $e$  of minimum cost
5:     if  $T \cup \{e\}$  does not contain a cycle then
6:       add  $e$  to  $T$ 
7:     end if
8:   end while
9:   return  $T$ 
10: end procedure
```

Running time of Krukals algorithm: $O(m \log n)$

Greedy approach 2:

- Start with an empty set of edges T .
- Start with a node s .
 - Add an edge $e = (s, v)$ of minimum cost to T .
- Repeat until T forms a spanning tree:

Prims algorithm:

Algorithm 14 Prims algorithm

```
1: procedure PRIMS( $G$ )
2:    $T \leftarrow \emptyset$ 
3:    $s \leftarrow$  an arbitrary node
4:   while  $T$  is not a spanning tree do
5:     add an edge  $e = (s, v)$  of minimum cost to  $T$ 
6:      $s \leftarrow v$ 
7:   end while
8:   return  $T$ 
9: end procedure
```

Running time of Prims algorithm: $O(m \log n)$

minimum spanning tree of example graph:

the minimum spanning tree sequence is d, a, c, b, e, f, h, g .

Greedy approach 3:

- Start with the full graph $G = (V, E)$.
- Delete an edge from G
 - the edge of maximum cost
- Repeat until G forms a spanning tree:

Reverse-delete algorithm:

Algorithm 15 Reverse-delete algorithm

```
1: procedure REVERSEDELETE( $G$ )
2:    $T \leftarrow G$ 
3:   while  $T$  is not a spanning tree do
4:     delete an edge  $e$  of maximum cost from  $T$ 
5:   end while
6:   return  $T$ 
7: end procedure
```

For when two edges have the same cost, use distinct labels to distinguish them.

Optimal with Priority Queue:

Add PQ to Prim's algorithm.

Algorithm 16 Optimal with Priority Queue

```
1: procedure OPTIMAL( $G$ )
2:    $T \leftarrow \emptyset$ 
3:    $s \leftarrow$  an arbitrary node
4:    $PQ \leftarrow$  empty priority queue
5:   for each node  $v$  do
6:     add  $v$  to  $PQ$  with key  $\infty$ 
7:   end for
8:   decrease key of  $s$  to 0
9:   while  $PQ$  is not empty do
10:     $v \leftarrow$  node with minimum key in  $PQ$ 
11:    add an edge  $e = (s, v)$  of minimum cost to  $T$ 
12:     $s \leftarrow v$ 
13:    for each edge  $e = (v, w)$  incident to  $v$  do
14:      if  $w$  is in  $PQ$  then
15:        decrease key of  $w$  to  $c_e$ 
16:      end if
17:    end for
18:  end while
19:  return  $T$ 
20: end procedure
```

Running time of Optimal with Priority Queue: $O(m \log n)$

4.3 Clustering

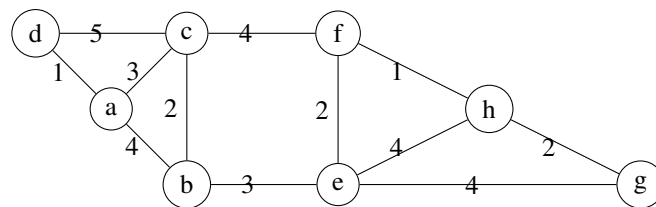
- a collection of n objects
- they have different degrees of similarity
- we want to organise them into coherent groups
- there is a notion of distance between objects

Definition:

- Given a set U of n elements, a k -clustering of U is a partition of U into non-empty subsets C_1, C_2, \dots, C_k .
- The spacing of a k -clustering is the minimum distance between any pair of points in different clusters.

Goal: Among all possible k -clusterings, find one with minimum spacing.

Example:



Greedy approach:

- Pick two objects p_i and p_j with minimum distance $d(p_i, p_j)$.
- Connect them with an edge $e = (p_i, p_j)$.

- Continue like this until we have k clusters.
- If the edge e under consideration connects two object p_i and p_j already in the same cluster, then discard e .

kruskals algorithm:

Algorithm 17 kruskals algorithm for clustering

Require: A graph $G = (V, E)$

Ensure: A minimum spanning tree of G with k clusters

```

1: procedure KRUSKAL( $G, k$ )
2:    $T \leftarrow \emptyset$ 
3:    $C \leftarrow \{\{v\} \mid v \in V\}$  ▷ Initial clusters
4:   Sort edges in  $E$  in increasing order of weight
5:   for  $\{u, v\} \in E$  do
6:     if  $C$  contains  $k$  clusters then
7:       break
8:     end if
9:     if clusters containing  $u$  and  $v$  are different in  $C$  then
10:       $T \leftarrow T \cup \{\{u, v\}\}$ 
11:      merge clusters containing  $u$  and  $v$  in  $C$ 
12:    end if
13:  end for
14:  return  $T$ 
15: end procedure

```

For Given example, the result of divide them into 3 clusters is:

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

Chapter 5

Dynamic Programming

The paradigm of dynamic programming: Given a problem P , define a sequence of subproblems, with the following properties:

- The subproblems are ordered from the simplest to the largest
- The largest problem is our original problem P
- The optimal solution of a subproblem can be structured from the optimal solutions of smaller subproblems.

Solve the subproblems from the smallest to the largest. When you solve a subproblem, store the solution and use it to solve larger subproblems.

5.1 Weighted Interval Scheduling

- A set of requests $R = \{1, 2, \dots, n\}$.
 - Request i has a start time s_i and a finish time f_i , and a value v_i .
 - Alternative view: every request is an interval $[s_i, f_i]$ associated with a value v_i .
- Two requests i and j are compatible if $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap.

build up a solution:

1. let O the optimal solution
2. O contains an optimal solution O' of the subproblem $R' = \{1, 2, \dots, i-1\}$
3. in order to find O , it suffices to look at smaller problems and find $O(1, 2, \dots, j)$ for some j
4. Let O_j be a shorthand for $O(1, 2, \dots, j)$ and let $OPT(j)$ be its total value.
5. Define $OPT(0) = 0$
6. Then $O = O_n$ with value $OPT(n)$
7. $OPT(j)$ can be computed from $OPT(j-1)$
8. $OPT(j) = \max\{OPT_{p_j} + v_j, OPT(j-1)\}$

Algorithm 18 ComputeOPT

```
1: procedure COMPUTEOPT( $j$ )
2:   if  $j = 0$  then
3:     return 0
4:   else
5:     return  $\max\{\text{COMPUTEOPT}(p_j) + v(j), \text{COMPUTEOPT}(j-1)\}$ 
6:   end if
7: end procedure
```

Correctness: ComputeOPT(j) correctly computes $OPT(j)$ for all $j = 0, 1, \dots, n$.

Proof by induction:

Base case: $OPT(0) = 0$ by definition.

Inductive step: Assume that it is true for all $i < j$. (Induction hypothesis)

return $\max\{\text{COMPUTEOPT}(p_j) + v(j), \text{COMPUTEOPT}(j - 1)\}$

Running time: $\Omega(2^n)$

Memoization:

- Compute ComputeOPT(j) for all $j = 0, 1, \dots, n$.
- Store it in an accessible place to use again later.
- Keep an array $M[0, \dots, n]$.
 - initially $M[j] = \text{EMPTY}$ for all $j = 0, 1, \dots, n$.
 - when ComputeOPT(j) is called, $M[j] = \text{ComputeOPT}(j)$.

Algorithm 19 M-ComputeOPT

```
procedure M-COMPUTEOPT(j)
  if j = 0 then
    return 0
  else if M[j] is not empty then
    return M[j]
  else
    M[j] ← max{M-COMPUTEOPT(pj) + v(j), M-COMPUTEOPT(j - 1)}
    return M[j]
  end if
end procedure
```

Running time: $O(n \log n)$

Algorithm 20 Find-Solution

```
procedure FIND-SOLUTION(j)
  if j = 0 then
    return ∅
  else
    if then v(j) + M-COMPUTEOPT(pj) > M-COMPUTEOPT(j - 1)
      return {j} ∪ FIND-SOLUTION(pj)
    else
      return FIND-SOLUTION(j - 1)
    end if
  end if
end procedure
```

Dynamic Programming vs Divide and Conquer:

Dynamic Programming:

- DP is an optimisation techniques and is only applicable to problems that have optimal substructure.
- DP splits the problem into parts, finds solutions to the parts and joins them. (The parts are not significantly smaller than the original problem and are overlapping.)
- In DP, the subproblems dependency can be represented by a directed acyclic graph.

Divide and Conquer:

- DC is not normally used for optimisation problems.
- DC splits the problem into parts, finds solutions to the parts and joins them. (The parts are significantly smaller than the original problem and are non-overlapping.)
- In DC, the subproblems dependency can be represented by a tree.

5.2 Subset Sum

Problem Description:

- Given a set of n items $1, 2, \dots, n$
- Each item i has a non-negative weight w_i .
- Given a bound W .
- **Goal:** select a subset S of items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximised.

Dynamic Programming: To find the optimal value of $OPT(n)$, we need

- the optimal value of $OPT(n-1)$ if item n is not selected.
- the optimal value of the solution on input $1, 2, \dots, n-1$ with weight bound $W - w_n$.

subproblems:

- Assumptions:
 - W is an integer
 - Every w_i is an integer
- subproblem for each $i = 0, 1, \dots, n$ and each integer $0 \leq w \leq W$.
- Let $OPT(i, w)$ be the optimal value of the solution on subset $1, 2, \dots, i$ with weight bound w .

Algorithm 21 SubsetSum

```
procedure SUBSETSUM( $n, w$ )
  Array  $M[0, \dots, n, 0, \dots, W]$ 
   $M[0, w] = 0$  for each  $w = 0, 1, \dots, W$ 
  for  $i = 1$  to  $n$  do
    for  $w = 0$  to  $W$  do
      if  $w_i > w$  then
         $M[i, w] = M[i-1, w]$ 
      else
         $M[i, w] = \max\{M[i-1, w], M[i-1, w-w_i] + w_i\}$ 
      end if
    end for
  end for
  return  $M[n, W]$ 
end procedure
```

Running time: $O(nW)$

5.3 knapSack

Problem Description:

- Given a set of n items $1, 2, \dots, n$
- Each item i has a non-negative weight w_i and a non-negative value v_i .
- Given a bound W .
- **Goal:** select a subset S of items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximised.

the fractional knapsack problem:

- Given a set of n items $1, 2, \dots, n$
- Each item i has a non-negative weight w_i and a non-negative value v_i .
- Given a bound W .
- **Goal:** select a fraction x_i of each item i such that $\sum_{i \in S} w_i x_i \leq W$ and $\sum_{i \in S} v_i x_i$ is maximised.

The 0/1 knapsack problem: Solution for 0/1 knapsack problem:

Algorithm 22 0/1 knapsack in dynamic programming

```
procedure 0/1 KNAPSACK( $n, W$ )  
  Array  $M[0, \dots, n, 0, \dots, W]$   
   $M[0, w] = 0$  for each  $w = 0, 1, \dots, W$   
  for  $i = 1$  to  $n$  do  
    for  $w = 0$  to  $W$  do  
      if  $w_i > w$  then  
         $M[i, w] = M[i - 1, w]$   
      else  
         $M[i, w] = \max\{M[i - 1, w], M[i - 1, w - w_i] + v_i\}$   
      end if  
    end for  
  end for  
  return  $M[n, W]$   
end procedure
```

Chapter 6

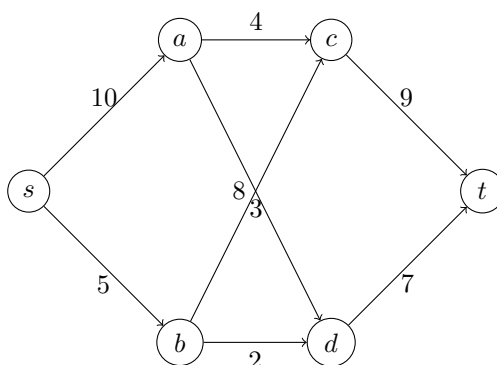
Network Flow

6.1 Network Flow Definitions

Flow network: A flow network is a directed graph $G = (V, E)$ with the following properties:

- Each edge $(u, v) \in E$ has a non-negative capacity c_e .
- There is a single source s in V .
- There is a single sink t in V .
- All other nodes in $V - \{s, t\}$ are called intermediate nodes.

example:



Further definitions:

- The source s has no incoming edges.
- The sink t has no outgoing edges.
- There is at least one edge incident to each node.
- All capacities are integers.

Flow: An $(s - t)$ flow is a function $f : E \rightarrow \mathbb{R}^+$, mapping each edge e to a non-negative real number $f(e)$. A feasible flow must satisfy the following conditions:

- Capacity: For each edge $e \in E$, $0 \leq f(e) \leq c_e$.
- Flow conservation: for each node $v \in V - \{s, t\}$, we have

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

The source s generates flow, and the sink t absorbs flow.

Value of a flow f , denoted $val(f)$, is the total amount of flow generated by the source s :

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

Generally, define $f^{out}(v)$ and $f^{in}(v)$ for the flow going out of (resp. going into) node v .

Similarly, define $f^{out}(S)$ and $f^{in}(S)$ for sets of nodes S .

6.2 Maximum Flow Problem

The maximum flow problem: Given a flow network $G = (V, E)$, find a flow of maximum possible value.

algorithm for maximum flow:

Idea: push flow forward on edges with leftover capacity, push flow backward on edges that are already carrying flow.

The residual graph G_f :

The residual graph G_f of G (also called the flow network) is defined as follows:

- The node set V_f of G_f is the same as the node set V .
- For each edge $(u, v) \in E$ which $f(e) < c_e$, there are $c_e - f(e)$ "leftover" units of capacity.
 - We will call this number the **residual capacity** of edge e .
 - We will call the edge e a forward edge.
- For each edge $(u, v) \in E$ with $f(e) > 0$, there is an edge $e' = (v, u)$ in E_f with a capacity of $f(e)$. We will call the edge e' a backward edge.

Working with residual graphs:

- Find an $(s - t)$ path P in G_f . This is called an **augmenting path**.
- Define the bottleneck of P ,
 - Denoted $bottleneck(P, f)$
 - to be the minimum residual capacity of any edge in P .
- Define the augmentation of flow f into flow f'
 - Denoted $augment(f, P)$.

Augmenting the flow:

Feasibility of capacity:

Algorithm 23 Augmenting the flow

```

1: procedure AUGMENT( $f, P$ )
2:    $b \leftarrow bottleneck(P, f)$ 
3:   for each edge  $e = (u, v) \in P$  do
4:     if  $e$  is a forward edge then
5:        $f(e) \leftarrow f(e) + b$ 
6:     else
7:        $f(e) \leftarrow f(e) - b$ 
8:     end if
9:   end for
10:  return  $f$ 
11: end procedure

```

consider an arbitrary edge $e = (u, v) \in P$. Suppose that e is a forward edge.

$$0 \leq f(e) \leq f'(e) = f(e) + b \leq f(e) + (c_e - f(e)) = c_e$$

Suppose that e is a backward edge.

$$c_e \geq f(e) \geq f'(e) = f(e) - b \geq f(e) - (f(e) - 0) = 0$$

The Ford-Fulkerson algorithm:

Algorithm 24 Max-flow algorithm

```

1: procedure MAX-FLOW( $G, s, t$ )
2:    $f(e) \leftarrow 0$  for all edges  $e \in E$ 
3:   while there exists an  $(s - t)$  path  $P$  in  $G_f$  do
4:      $f \leftarrow \text{augment}(f, P)$ 
5:      $f' \leftarrow \text{update}(f)$ 
6:      $G_f \leftarrow \text{update}(G_f, f)$ 
7:   end while
8:   return  $f$ 
9: end procedure

```

Running time of Ford-Fulkerson algorithm: $O(mC)$, where C is the maximum capacity of any edge in the network.

6.3 Min Cut theorem

A cut C is a partition of the nodes of G into two sets S and T such that $s \in S$ and $t \in T$.

The capacity of a cut $C = (S, T)$ of a cut C is the sum of the capacities of the edges "out of" S : these are edges (u, v) such that $u \in S$ and $v \in T$.

The min-cut theorem: In every flow network, the value of the maximum flow is equal to the capacity of the minimum cut.

A series of facts:

Fact 1: Let f be any $(s - t)$ flow and let (S, T) be any cut. Then $v(f) = f^{out}(S) - f^{in}(S)$.

1. By definition, $v(f) = f^{out}(s)$.
2. By definition $f^{in}(s) = 0$.
3. Hence, $v(f) = f^{out}(s) - f^{in}(s)$.
4. For every other node $v \neq s, t$, we have $f^{out}(v) = f^{in}(v)$.
5. Therefore, $v(f) = \sum_{v \in S} (f^{out}(v) - f^{in}(v))$.
6. rewrite as $v(f) = \sum_{v \in S} (f^{out}(v) - f^{in}(v)) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e) = f^{out}(S) - f^{in}(S)$.

Fact 2: Let f be any $(s - t)$ flow and let (S, T) be any $(s - t)$ cut. Then $v(f) = f^{out}(T) - f^{in}(T)$.

Fact 3: Let f be any $(s - t)$ flow and let (S, T) be any $(s - t)$ cut. Then $v(f) \leq c(S, T)$.

$$\begin{aligned}
 v(f) &= f^{out}(S) - f^{in}(S) \\
 &\leq f^{out}(S) \\
 &= \sum_{e \text{ out of } S} f(e) \\
 &\leq \sum_{e \text{ out of } S} c(e) \\
 &= c(S, T)
 \end{aligned}$$

Fact 4: Let f be any $(s - t)$ flow in G such that the residual graph G_f contains no augmenting paths. Then there exists an $(s - t)$ cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.

Proving fact 4: In the residual graph G_f , identify all nodes that are reachable from the source s . Let S^* be the set of these nodes, and let $T^* = V - S^*$.

1. $s \in S^*$ and $t \in T^*$.
2. No edge of G_f crosses from S^* to T^* .
3. Every edge of G_f crosses from T^* to S^* .
4. $f^{out}(S^*) = v(f)$.
5. $f^{in}(S^*) = 0$.

$$\begin{aligned}
 v(f) &= f^{out}(S^*) - f^{in}(S^*) \\
 &= \sum_{e \text{ out of } S^*} f(e) - \sum_{e \text{ into } S^*} f(e) \\
 &= \sum_{e \text{ out of } S^*} c(e) - 0 = c(S^*, T^*)
 \end{aligned}$$

Fact 5: If all capacities are integers, then there exists a maximum flow f for which $f(e)$ is an integer for every edge e .

6.4 Choosing Better Augmenting Paths

The Edmonds-Karp algorithm:

Algorithm 25 Edmonds-Karp algorithm

```

1: procedure EDMONDS-KARP( $G, s, t$ )
2:    $f(e) \leftarrow 0$  for all edges  $e \in E$ 
3:   while there exists an  $(s - t)$  path  $P$  in  $G_f$  do
4:      $P$  is a shortest  $(s - t)$  path
5:      $f \leftarrow \text{augment}(f, P)$ 
6:      $f' \leftarrow \text{update}(f)$ 
7:      $G_f \leftarrow \text{update}(G_f, f)$ 
8:   end while
9:   return  $f$ 
10: end procedure

```

Running time of Edmonds-Karp algorithm: $O(nm^2)$, where n is the number of nodes and m is the number of edges in the network.

The shortest path can be found in $O(m)$ time using BFS.

6.5 Modeling with Network Flows

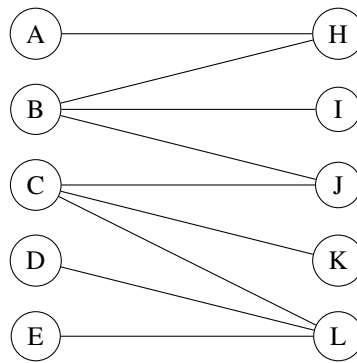
Bipartite graphs: A graph $G = (V, E)$ is bipartite if and only if it can be partitioned into two sets A and B such that every edge has one endpoint in A and one endpoint in B .

Maximum bipartite matching:

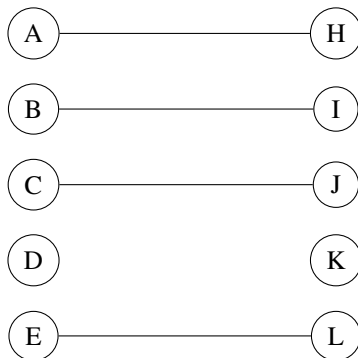
Matching: A subset M of edges E such that each node $v \in V$ appears in at most one edge $e \in M$.

Maximum matching: A matching with maximum cardinality.

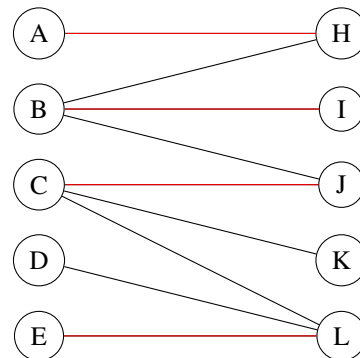
examples of bipartite graphs:



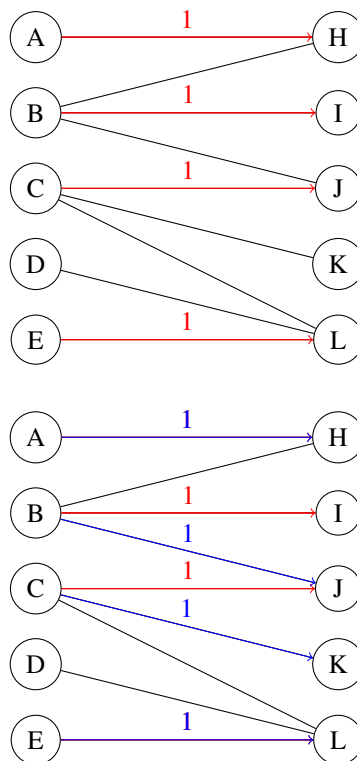
example of maximum bipartite matching:



example of maximal bipartite matching:



From matchings to flows:



Maximum Flow and Maximum Matching

The size of the maximum matching is equal to the value of the maximum flow.

The edges of M are the edges that carry flow from A to B in the residual network.

Running time: $O(mn)$

Baseball Elimination

- Given a set S of teams
- For each team x in S , the current number of wins w_x
- For teams x and y in S , they still have to play g_{xy} games against each other
- Given a designated team z
- Can z still win the tournament?

From Baseball Elimination to flows

- For each pair of teams x and y , create a vertex v_{xy}
- For each team x , create a vertex v_x
- For each pair of teams x and y , create an edge (s, v_{xy}) with capacity g_{xy}
- For each team x , create an edge (v_x, t) with capacity $w_z + g_{xz} - w_x$
- For each pair of teams x and y , create an edge (v_{xy}, v_x) with infinite capacity
- For each pair of teams x and y , create an edge (v_{xy}, v_y) with infinite capacity

Open pit mining

- Given a set S of blocks
- For each block x in S , the value v_x of the ore in the block
- For each block x in S , the cost c_x of mining the block
- For each block x in S , the set N_x of blocks that are neighbors of x
- Given a designated block z
- What is the maximum value of ore that can be mined?

From open pit mining to flows

- For each block x in S , create a vertex v_x
- For each block x in S , create an edge (s, v_x) with capacity v_x
- For each block x in S , create an edge (v_x, t) with capacity c_x
- For each block x in S , create an edge (v_x, v_y) with infinite capacity for each block y in N_x

Chapter 7

NP-Completeness

7.1 NP-Completeness

Polynomial time reduction

- Given a problem A to solve
- Reduce solving A to solving B
- Assume there is an algorithm ALG^B that solves B at cost $O(1)$
- Construct an algorithm ALG^A that solves A , which uses ALG^B as a subroutine
- If ALG^A runs in polynomial time, then this is a polynomial time reduction

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B .

I can try come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for A .

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves A .

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B .

B is "at least as hard to solve as" A , because if I could solve B , I could also solve A .

Types of reductions

- Turing reduction: $A \leq_T B$
 - A reduction which solves A using (potentially many) calls to an oracle for B
 - As known as Cook reduction
- Many-one reduction: $A \leq_m B$
 - A reduction which converts instances of A to instances of B
 - Also known as Karp reduction

Problem classification

Problems in P :

Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, weighted interval scheduling, etc.

Problems in NP :

subset sum, knapSack, weighted interval scheduling, Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, etc.

NP-hardness

A problem B is NP-hard if for every problem A in NP , $A \leq^p B$.

If every problem in NP is polynomial time reducible to B , then this captures the fact that B is at least as hard as any problem in NP .

3 SAT

- A CNF formula with m clauses and k literals.

$$\varphi = (x_1 \vee \neg x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \cdots \wedge (x_3 \vee x_8 \vee x_{12})$$

- each clause has exactly three literals
- Truth assignment: A value in $\{0, 1\}$ for each variable x_i
- Satisfying assignment: A truth assignment which makes the formula evaluate to 1
- Computational problem 3 SAT: Decide if the input formula φ has a satisfying assignment.

3 SAT is NP-complete

- 3 SAT is in NP
 - Given a truth assignment, we can check in polynomial time if it is satisfying
- 3 SAT is NP-hard
 - Given a CNF formula φ , we can construct a polynomial time reduction to 3 SAT

Proving NP-completeness Suppose that you are given a problem A and you want to prove that A is NP-complete.

First, prove that A is in NP .

Usually by observing that a solution is efficiently verifiable.

Then prove that A is NP-hard.

construct a polynomial time reduction from a known NP-complete problem P .

7.2 NP-completeness of the vertex cover problem

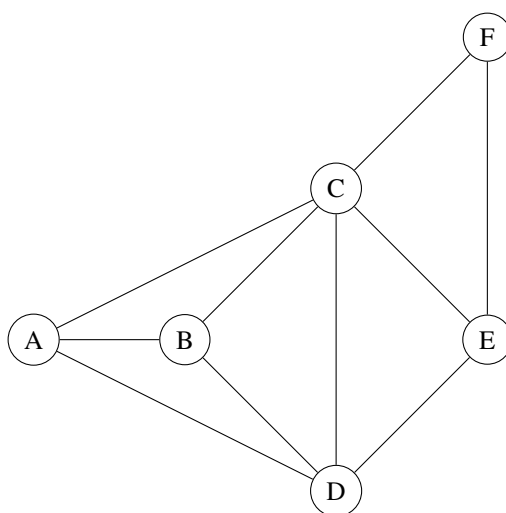
Vertex cover Definition: A vertex cover C of a graph $G = (V, E)$ is a subset of vertices $C \subseteq V$ such that for every edge e in E , at least one of the endpoints of e is in C .

Definition: A minimum vertex cover is a vertex cover of smallest possible size.

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

Output: A minimum vertex cover

Example



the vertex cover $\{A, C, E\}$ is not a minimum vertex cover
the minimum vertex cover is $\{A, E\}$.

Vertex cover is NP-hard: construct a polynomial time reduction from 3 SAT to vertex cover.

Let φ be a 3 CNF formula with m clauses and d variables.

Construct in polynomial time an instance $\langle G, k \rangle$ of vertex cover, with $k = d + 2m$.

if φ is satisfiable, then G has a vertex cover of size at most k

Let (y_1, y_2, \dots, y_d) in $\{0, 1\}^d$ be a satisfying assignment for φ

For the nodes on the top: If $y_i = 1$. Include node x_i in the vertex cover C , otherwise include node $\neg x_i$ in C .

For the nodes on the bottom: in each triangle, choose a node x_i that has been picked on the top and do not include it in the vertex cover. Include the other two nodes.

if φ is not satisfiable, then G has no vertex cover of size at most k

Let C be a vertex cover of size $k = d + 2m$ in G .

Since it is a vertex cover, it must include at least two out of three nodes in each "clause gadget" at the bottom.

this means that at most d nodes can be picked on the top.

To satisfy the edges at the top, in each "variable gadget", at least one node must be picked.

7.3 Further reductions in NP

Form optimization to decision

Given an optimization problem P , introduce a threshold k .

The decision version P_d becomes: Given an instance of P and the threshold k as input, is there a solution to P with value at most k ?

If P solved in polynomial time, then P_d is also solved in polynomial time.

If P_d solved in polynomial time, then P is also solved in polynomial time.