Summary of COMP523 Advanced Algorithm

May 4, 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 $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ 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

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

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 pesudocode 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** $n1$ 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** $\text{BINARYSEARCH}(x, i, j)$
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** $\text{BINARYSEARCH}(x, i, \lfloor (i + j)/2 \rfloor)$
13:         **else**
14:             **return** $\text{BINARYSEARCH}(x, \lfloor (i + j)/2 \rfloor + 1, j)$
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:

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

proof $T(n) \le 4\log n$:
Base case: $n = 1$, $T(1) = 0 \le 4\log 1 = 0$.
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: $T(k + 1) \le 4\log(k + 1)$.

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

Memory usage:
The memory usage of BinarySearch is:

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

Comparing BinarySearch and LinearSearch:

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

2.3   Running time of Divide and Conquer algorithms

The Master Theorem:
Suppose that $T(n)$ satisfies the recurrence:

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

where $a \ge 1$, $b > 1$, $c > 0$ and $d \ge 0$ 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 piar 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) = CLOSESTPAIRREC(Q_x, Q_y)
8:      (r_1, r_2) = 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:     $S = \{p \in P : p \in L$ and p is within δ 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.

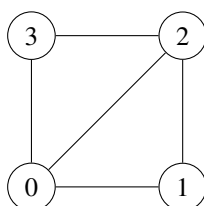**cliques**: A graph with n vertices and $n(n-1)/2$ edges.

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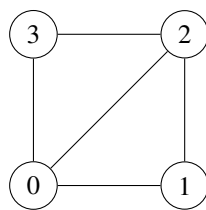
examples of adjacency matrices for directed graphs:
Given the following graph:

The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$
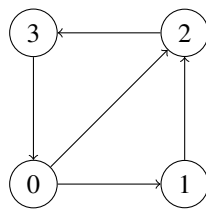
Adjacency list: A list of lists, where the ith list contains the neighbours of vertex i.
Given the following graph:

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 ith 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	$O(deg(j))$ 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 = 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.
exaqmple 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:          $L_{i+1} \leftarrow emptylist$
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:
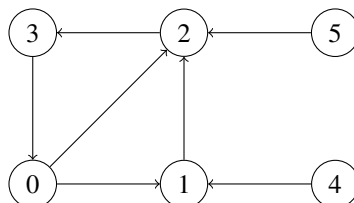
DFS and BFS on directed graphs:
Very similar to undirected graphs, except that we only consider edges that go out of a vertex.
Running time is $O(n + m)$
For example graph above the DFS sequence is 0, 1, 2, 3.
The BFS sequence is 0, 1, 2, 3.

3.4.1   Connectivity

Weak connectivity:If we ignore the direction for all edges, there would be a pah 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 any 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 ← s
 5:     C[s] ← red
 6:     i ← 0
 7:     while L[i] ≠ ∅ do
 8:         L_{i+1} ← emptylist
 9:         for v ∈ L[i] do
10:             for edges (e) incident to v do
11:                 if e is unexplored then
12:                     w ← 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] ← green
18:                         else
19:                             C[w] ← 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 ← i + 1
31:     end while
32:     for e(v, w) ∈ 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 $u_1, u_2, \ldots, u_n$ such that for every edge (u_i, u_j), we have $i < j$.

Intutively, 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 $u_1, u_2, \ldots, u_n$ 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 $u_1, u_2, \ldots, u_k$ 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:     $L \leftarrow emptylist$
 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) $C_1, C_2, \ldots, C_k$, for some k.

- $V^{SCC} = \{C_1, C_2, \ldots, C_k\}$ 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 $R = \{1, 2, \ldots, n\}$.

- 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:

Figure 4.1: Interval Scheduling

17

Interval Scheduling Algorithm:

Algorithm 12 Interval Scheduling Algorithm

1: **procedure** INTERVALSCHEDULING($[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$)
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 Priorty Queue:
Add PQ to Prim's algorithm.

Algorithm 16 Optimal with Priorty Queue

```
 1: procedure OPTIMAL(G)
 2:     T ← ∅
 3:     s ← an arbitrary node
 4:     PQ ← empty priority queue
 5:     for each node v do
 6:         add v to PQ with key ∞
 7:     end for
 8:     decrease key of s to 0
 9:     while PQ is not empty do
10:         v ← node with minimum key in PQ
11:         add an edge e = (s, v) of minimum cost to T
12:         s ← 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 Priorty 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 $C_1, C_2, \ldots, C_k$.

- 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:     $C \leftarrow \{\{v\} \mid v \in V\}$ $\triangleright$ 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:

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