The Hong Kong University of Science & Technology

Department of Computer Science

COMP 271, Section 2: Design and Analysis of Algorithms

final examination (ANSWER)

Date: 12:30 - 3:30 p.m., 12 December, 1998.

Venue: LG1 Sports Hall Lobby

Student name: ____________________

Student ID: ____________________

e_mail: ____________________

Question	*Grade*
1	/24 pts
2	/12 pts
3	/8 pts
4	/12 pts
5	/8 pts
Total	/64 pts

Problem 1: Answer the following short questions.

(a) Mr. Careless claims that the running time of depth-first search on a graph G can always be done in $O(|V| + |E|)$ time, which is independent to the input graph representation. Discuss his claim.

If the input graph is represented by adjacent matrix, then the running time is $O(|V|^2)$.

(b) What is topological sort? Will the topological ordering of a directed acyclic graph be unique? Give an example to illustrate your answer.

A topological sort of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering. Note that in general, there may be many legal topological orders for a given DAG.

(c) During the operation of Dijkstra's algorithm, we will perform relaxation process. What is the meaning of relaxation?

Dijkstra's algorithm maintains an estimate of the shortest path for each vertex, call this $d[v]$. We will always require that $d[v]$ equals the cost of some known path. As the algorithm is running, it attempts to update $d[v]$ for each vertex in the graph, until all the $d[v]$ values converge to the true shortest distances. The process by which an estimate is updated is called relaxation.

(d) Discuss how some real-life problem that can be modeled as minimum spanning tree problem.

A common problem in communications networks and circuit design is that of connecting together a set of nodes (communication sites or circuit components) by a network of minimal total length (where length is the sum of the lengths of connecting wires).

(e) Discuss the difference between dynamic programming and divide and conquer algorithms.

Dynamic programming approach divides the problem into recursive subproblems (as in divide-and-conquer) but unlike divide-and-conquer there is often overlap between the subproblems.

(f) Does $P \subseteq NP$? Argue your answer briefly.

P is a subset of NP (i.e., $P \subseteq NP$). Since we can solve a problem in polynomial time, then we can certainly verify the correctness of the solution in polynomial time (or else we can't solve the problem in polynomial time).

(g) What is Cook's theorem? What is the importance of Cook's theorem?

Cook's theorem: SAT is NP complete.
Suppose we don't have Cook's theorem, according to definition of NP-complete, if we want to prove π is NP-complete, we have to reduce EVERY problem in NP to π. But there are infinitely many such problems, we have no hope to do this. Thanks to Cook's theorem, we can now take SAT is NP-complete and deduce from SAT to prove π is NP-complete.

(h) Give the definitions of intractable problems and NP-Complete problems, compare the difference between them.

Intractable Problems: No algorithm can exist that runs in polynomial time.

NP-Complete Problems: There exist algorithms for these problems, but they run in time exponential in the input size. People suspect that no polynomial-time algorithms exist for these problems.

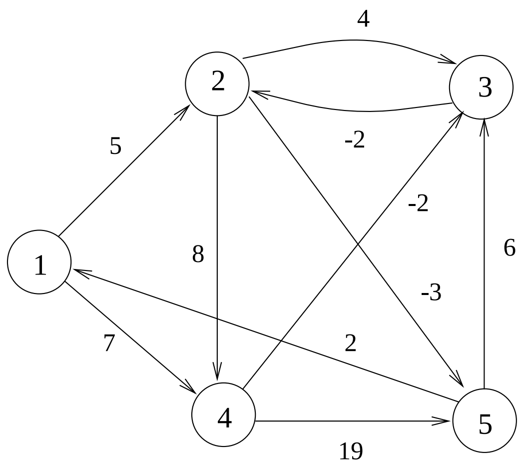
The main difference between them is we believe NP-complete problems are intractable, but nobody has come up with a proof yet.

Problem 2:

(a) What is the difference between single-source shortest-path problem and all-pairs shortest-path problem?

Given a graph $G = (V, E)$, the single-source shortest-path problem only requires the shortest path from a given source vertex $s \in V$ to every vertex $v \in V$. But in the case of all-pairs shortest-path problem, we need to find the shortest path from u to v for every pair of vertices u and $v \in V$.

(b) Given the following graph $G = (V, E)$, start with source 1, trace the actions of Dijkstra's algorithm. For each iteration, write down the d value of each vertex.



(c) For the same graph, trace the actions of Floyd-Warshall algorithm, show the matrix $D^{(k)}$ during each iteration.

```
D(0)=
0        5        infty    7        infty
infty    0        4        8        -3
infty    -2       0        infty    infty
infty    infty    -2       0        19
2        infty    6        infty    0

D(1)=
0        5        infty    7        infty
infty    0        4        8        -3
infty    -2       0        infty    infty
infty    infty    -2       0        19
2        7        6        9        0

D(2)=
0        5        9        7        2
infty    0        4        8        -3
infty    -2       0        6        -5
infty    infty    -2       0        19
2        7        6        9        0

D(3)=
0        5        9        7        2
infty    0        4        8        -3
infty    -2       0        6        -5
infty    -4       -2       0        -7
2        4        6        9        0

D(4)=
0        3        5        7        0
infty    0        4        8        -3
infty    -2       0        6        -5
infty    -4       -2       0        -7
2        4        6        9        0

D(5)=
0        3        5        7        0
-1       0        3        6        -3
-3       -2       0        4        -5
-5       -4       -2       0        -7
2        4        6        9        0
```

(d) Compare your answers in part (b) and part (c), does the Dijkstra's algorithm correctly find the shortest paths? Why?

The Dijkstra's algorithm failed to find the shortest path from v_1 to v_2 and v_1 to v_5. It is because the correctness of Dijkstra's algorithm can only be ensured when the weight of each edge is positive. But in our case, some edges have negative weight and therefore Dijkstra's algorithm fails to work.

Problem 3:

(a) What is the meaning of prefix-free code? Why is it so important to ensure the property of the prefix-free codes?

Prefix-free code (or prefix code) ensures no codeword is a prefix of some other codewords. If the code is prefix-free, no codeword is a prefix of any other codewords. But if a codeword is a prefix of some other codewords, then it will be ambiguous.

(b) Huffman invented a greedy algorithm to construct an optimal prefix-free code called Huffman code. What is being optimized in Huffman coding procedure? Argue briefly why Huffman code is prefix-free.

Let the frequence of each character c_i be f_i, and $d(c_i)$ be the length of codeword for character c_i. The Huffman code optimizes the cost of the code, i.e., $\sum_i f(c_i) \times d(c_i)$.

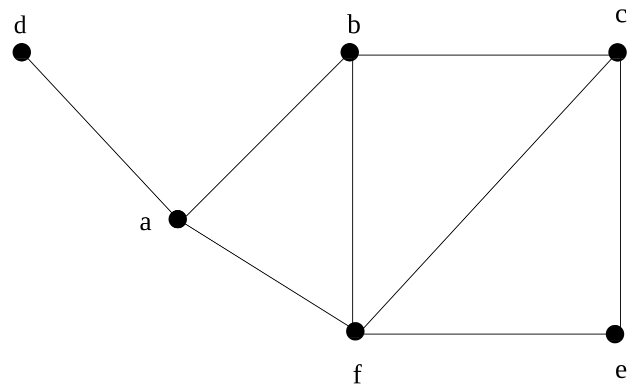
Huffman encoding algorithm ensures the codes are prefix-free. It is because we trace down the Huffman tree from the root to the leave (not internal node) to assign the codeword for each character. This process ensures the codeword is prefix-free.

(c) Build an optimal Huffman tree for the following seven characters with the frequencies specified in parentheses: A(440), E(400), O(300), U(250), F(150), R(100), T(50).

Problem 4:

An independent set of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident in at most one vertex in V'. The optimization version of the independent set problem is to find a maximum-size independent set of G.

(a) Find the maximum-size independent set of the following graph.



$\{b, d, e\}$

(b) What is the related decision version for the independent set problem?

Given an undirected graph $G = (V, E)$ and an integer k, does G has an independent set of size at least k?

(c) Prove that the independent set problem is in NP.

Given G and k, to verify that G has an independent set of size k we can treat that k vertices as the independent set. Given this candidate solution, it is easy to verify that they form an independent set or not. For each every edge, we need to check if both the incidence vertices are in the candidate solution or not. Hence the verification procedure can be completed in $O(|V| \cdot |E|)$.

(d) Prove that the independent set problem is NP-complete. You need not to argue the correctness of the reduction.

(*Hint* : Reduce from the clique problem.)

clique $\propto$ independent-set:
Define the complement of a graph G to be the graph G to be the graph $\bar{G} = (V, \bar{E})$ where

$$\bar{E} = \{(u, v) | (u, v) \notin E\}.$$

In other words, $\bar{G}$ is the graph containing exactly those edges that are not in G.

Define the following mapping f that takes an problem instance of clique (G, k) as input, and output $(\bar{G}), k)$ as problem instance of independent-set problem.

Combine with the answer in part (c), we show that independent-set problem is NP-complete.

Problem 5:

Give an $O(n^2)$ dynamic programming algorithm which finds the longest monotonically increasing subsequence of a sequence of n numbers. Your algorithm should be self-contained, i.e., it should not call any algorithm as a subroutine. Analyze the running time of your algorithm.

In other words, the algorithm should find the largest subset of the sequence such that each successive number is greater than its predecessor. For the sequence $< 1, 5, 2, 7, 3, 6 >$, the algorithm should produce $< 1, 2, 3, 6 >$.

(You need not to argue the correctness of your algorithm.)

Let $X =< x_1, \ldots, x_n >$ be the given sequence of n numbers. We need to find the longest monotonically increasing subsequence of X.

We first give an algorithm which finds the length of the longest monotonically increasing subsequence; later, we will modify it to produce the longest monotonically increasing subsequence.

Let $X_i =< x_1, \ldots, x_i >$ denotes the prefix of X consisting of the first i items. Define $c[i]$ to be the length of the longest monotonically increasing subsequence that ends with x_i. It is clear that the length of the longest increasing subsequence in X is given by $max_{1 \leq i \leq n} c[i]$.

The longest monotonically increasing subsequence that ends with x_i has the form $< Z, x_i >$ where Z is the longest monotonically increasing subsequence that ends with x_r for some $r < i$ and $x_r \leq x_i$. Thus, we have the following recurrence relation:

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r \geq x_i \text{ for } 1 \leq r < i \\ max_{1 \leq r < i, x_r < x_i}(c[r] + 1) & \text{if } i > 1 \end{cases}$$

The basis follows from the fact the longest monotonically increasing subsequence in a sequence consisting of one number is that number itself. The recurrence relation says that if all the numbers to the left of i are greater than x_i then the length of the longest increasing subsequence ending in x_i is 1. Otherwise, the length of the longest monotonically increasing subsequence ending in x_i is 1 more than the length of the longest monotonicall increasing subsequence ending at a number x_r to the left of x_i such than x_r is smaller than x_i.

We store the $c[i]$'s in an array whose entries are computed in order of increasing i. After computing the c array we run through all the entries to find the maximum value. This is the length of the longest monotonicall increasing subsequence in X.

In order to report the optimal subsequence, we need to store for each i, not only $c[i]$ but also the value of r which achieves the maximum value in the recurrence relations. Denote this by $r[i]$. We then can trace the solution as follows. Let $c[k]$=$max_{1 \leq i \leq n} c[i]$. Then x_k is the last number in the optimal subsequence. The second to last number is $x_{r[k]}$, the third to last number is $x_{r[r[k]]}$ and so on, until we have found all numbers of the optimal subsequence.

Running time: Since it takes $O(i)$ time to compute the i-th entry of the c array, the total time to compute the c array is $O(\sum i) = O(n^2)$. It takes $O(n)$ time to find the maximum in the c array. Finally, the time to trace the solution is $O(n)$. Thus, the running time is dominated by the time it takes to compute the c array, which is $O(n^2)$.