ESO207: Data Structures and Algorithms *Time : 3 hrs*

End-Semester Exam: 1 *Max. Marks: 100*

Instructions.

a. The exam is closed-book/closed-notes. No collaboration is permitted. You may not have your cell phone on your person.

b. You may use algorithms done in the class as subroutines and cite their properties, unless explicitly asked to prove them.

c. Describe your algorithms completely and precisely in English, preferably using pseudo-code.

d. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity of your arguments. **Always argue correctness of your algorithms.**

Problem 1.

(a) Give a linear time algorithm that takes as input a directed acyclic graph G and determines whether G contains a directed path that touches every vertex exactly once. (7) **Soln.** Let $v_1, v_2, \ldots, v_n$ be a topological order. If there is a directed path, then, this must be it. So, (1) obtain the topological order, and (2) check if there is an edge from v_i to v_{i+1}. This can be done in $O(|V| + |E|)$ time.

(b) Give a linear time algorithm that takes as input a weighted directed graph G and for every source vertex u, finds the length of the longest directed path in G originating at u. (8)

Soln. Let $l(u)$ denote the length of the longest path in G originating at u. Then,

$$l(u) = \max_{v \text{ adjacent to } u} l(v) + w(u,v)$$

This can be easily implemented by making a pass over the vertices in reverse topological order. Because, if there is an edge from u to v, then, in the reverse topological order, v will be processed and $l(v)$ will be computed before u is processed. Computation is obviously linear time (topological ordering + above computation).

Problem 2. A certain string processing language offers a primitive operation which splits a string into two pieces. The operation involves copying the original string and so, for a string of length n, it costs n units to split a string, regardless of the location of the cut. Suppose now that you want to break the string into multiple pieces. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a cost of 20, followed by the second cut at position 10 incurs a cost of 17, for a total of $20 + 17 = 37$. However, going for the first cut at 10, and the second cut at 3 incurs a cost of $20 + 10 = 30$, which is better.

Give a dynamic programming algorithm that, given the locations of the m cuts in a string of length n, finds the minimum cost of breaking it at those locations into $m+1$ pieces. Give recurrence relation, brief argument and pseudo-code. (20)

Soln. Let the locations of the cuts be at $1 < l_1 < l_2 \ldots < l_m < n$. For ease, let $l_0 = 1$ and $l_{m+1} = n$. For $0 \le i \le j \le m+1$, define $C(i,j)$ to be the least cost of cutting a string of size $l_j - l_i + 1$ into pieces at locations $l_{i+1} - l_i + 1, \ldots, l_{j-1} - l_i + 1$. Equivalently, in terms of the original "coordinates", $C(i,j)$ is the least cost of cutting the segment $[l_i, l_j]$ at locations $l_{i+1}, \ldots, l_{j-1}$.

Base Case (initialization): All diagonal entries $C(i,i)$ are 0. One above diagonal is also 0: $C(i, i+1) = 0$, for each $i = 0, \ldots, m$.

Induction Case: The optimal cut for $C(i,j)$ would be at index l_k, for some $i < k < j$. In which case, the cost is $C(i,j) = (l_j - l_i + 1) + C(i,k) + C(k,j)$. Since we do not know k, we minimize over all $k = i+1, \ldots, j-1$. That is,

$$C(i,j) = \min_{k=i+1}^{j-1} C(i,k) + C(k,j) + l_j - l_i + 1 \ .$$

Let s denote the size of the problem, namely, $s = j - i + 1$. We are interested in $C(0, n+1)$. We can translate the recurrence relation into the following pseudo-code.

```
1.  for  $i = 0$ to $m$
2.        $C[i, i+1] = 0$
3.  for $s = 2$ to $m + 1$
4.        for $i = 0$ to $m + 1 - s$
5.              $j = s + i$
6.              $mincost = -\infty$
7.              for $k = i + 1$ to $j - 1$
8.                    $mincost = \min(mincost, C[i,k] + C[k,j] + l_j - l_i + 1)$
```

Time complexity is $O(m^3)$.

Problem 3. Consider the following game. A "card dealer" produces a sequence $s_1, s_2, \ldots, s_n$ of "cards", face up, where the card s_i has value v_i (say). Two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect the cards of largest total value. (For example, you can think of the cards as bills of different denominations). Assume that n is even.

(a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. In other words, the greedy strategy is not optimal for this problem. (5)

Soln. Consider the sequence of values $1, 3, 10, 2$. The first player has a choice of choosing the first or the last, namely, 1 or 2. By greedy choice, suppose the player chooses 2. The sequence becomes $1, 3, 10$. Now player 2 chooses 10, and no matter what player 1 chooses 1 or 3, he loses.

(b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, such that, the first player should be able to make each move optimally in $O(1)$ time by looking up this precomputed information. (20)

Soln. Let us denote the first player as player 0 and the second player as player 1. Consider the i, j subsequence $s_i, s_{i+1}, \ldots, s_{j+1}$. For $j - i + 1$ even, let $E(i,j)$ denote the best total value that player 0 can hope to get assuming that player 1 plays optimally henceforth. In this case, let $O(i,j) = \sum_{k=i}^{j} v_k - E(i,j)$. For $j - i + 1$ odd, let $O(i,j)$ be the best total value that player 1 can hope to get assuming player 0 plays optimally henceforth. In this case, let $E(i,j) = \sum_{k=i}^{j} v_k - O(i,j)$. Let $M(i,j)$ be the best move of the corresponding player, that is, $M(i,j) = i$ means s_i is picked, and $M(i,j) = j$ means card s_j is picked (whoever's turn it is to pick).

Let $S(i,j)$ denote the sum of the values of the $[i,j]$ segment, $S(i,j) = \sum_{k=i}^{j} v_k$. The array $S(i,j)$, $1 \le i \le j \le n$ can be computed in time $O(n^2)$ easily. For each $1 \le i \le n$, compute $S(i,i) = v_i$ and $S(i,j+1) = S(i,j) + v_{j+1}$, for $j = i+1, \ldots, n$.

Base Case: $O(i,i) = v_i, E(i,i) = 0, M(i,i) = i$.

General Case:
We consider the computation of $E(i,j)$ and $O(i,j)$. Suppose $j-i+1$ is even. If card s_i is picked, then the remaining sequence is $s_{i+1} \ldots s_j$ and the value is $v_i + E(i+1,j)$. The other possibility is to pick s_j and then the value is $v_j + E(i,j-1)$. We take the better of these two, that is,

$$\begin{aligned}
&E(i,j) = \max(v_i + E(i+1,j), v_j + E(i,j-1)), \quad j-i+1 \text{ even}\\
&O(i,j) = S(i,j) - E(i,j) \ .\\
&M(i,j) = \begin{cases} i & \text{if } v_i + E(i+1,j) \ge v_j + E(i,j-1) \\ j & \text{otherwise.} \end{cases}
\end{aligned}$$

Correspondingly, the case when $j-i+1$ is odd is the following.

$$\begin{aligned}
&O(i,j) = \max(v_i + O(i+1,j), v_j + O(i,j-1)), \quad j-i+1 \text{ odd}\\
&E(i,j) = S(i,j) - O(i,j) \ .\\
&M(i,j) = \begin{cases} i & \text{if } v_i + O(i+1,j) \ge v_j + O(i,j-1) \\ j & \text{otherwise.} \end{cases}
\end{aligned}$$

The problem size is $t = j-i+1$, which ranges from 1 to n. The computation proceeds by solving smaller size problems to larger size problems (in fact, size increases by 1 only in each iteration). The pseudo-code is given below. It clearly runs in $O(n^2)$ time. The move is always found by looking at $M(i,j)$, where the current card sequence is $s_i \ldots s_j$. This is $O(1)$.

```
ComputeOEM($v$) // input $v_1, \ldots, v_n$ given as array
1.  Compute the matrix $S(i,j)$, $1 \le i \le j \le n$.
2.  for $i = 1$ to $n$
3.      $E(i,i) = 0$, $O(i,i) = v_i, M(i,i) = i$
4.  for $t = 1$ to $n - 1$
5.      for $i = 1$ to $n - t$
6.          $j = i + t$
7.          if $j - i + 1$ is even
8.              if $v_i + E(i+1,j) \ge v_j + E(i,j-1)$
9.                  $E(i,j) = v_i + E(i+1,j)$
10.                 $M(i,j) = i$
11.             else
12.                 $E(i,j) = v_j + E(i,j-1)$
13.                 $M(i,j) = j$
14.             $E(i,j) = S(i,j) - E(i,j)$
15.         else
16.             if $v_i + O(i+1,j) \ge v_j + O(i,j-1)$
17.                 $O(i,j) = v_i + O(i+1,j)$
18.                 $M(i,j) = i$
19.             else
20.                 $O(i,j) = v_j + O(i,j-1)$
21.                 $M(i,j) = j$
22.             $E(i,j) = S(i,j) - O(i,j)$
```

Problem 4. Let $G = (V, E)$ be an undirected connected graph with edge weights $w(e)$, for edges e.

(a) Prove the following property: if e is the heaviest edge in some simple cycle C of G, then there is a minimum spanning tree that does not contain e. Argue carefully. (5)

Soln. Suppose there is an MST T that contains $e = \{a, b\}$. We will show that we can construct an alternative MST that does not contain e. Deleting e from T will break T into two connected sets of edges T_1 and T_2: Let S be the set of vertices corresponding to T_1 and $V - S$ be the set of vertices corresponding to the other. Consider the cut $(S, V - S)$. The edge e crosses the cut. Now e lies on a simple cycle C. This cycle has at least one other edge $e' = \{c, d\}$ crossing this cut.

Consider $T' = (T - \{e\}) \cup \{e'\}$. We claim that this is connected. To show this, it suffices to show that connectivity is maintained between a and b in T' (since that is the edge that was deleted). But note that in T_1, being a tree, there is a path P_1 from a to c and in T_2, there is a path P_2 from d to b. So the path $P_1\ e'\ P_2$ is a path from a to b in T'. Further the cost of T' is

$$\mathit{cost}(T') = \mathit{cost}(T) - (w(e) - w(e'))$$

Since e is the heaviest edge on the cycle C, $w(e) \ge w(e')$ and so $\mathit{cost}(T') \le \mathit{cost}(T)$.

Thus we have shown that there is an MST T that does not contain e.

(b) Here is an alternative MST algorithm. (5)

```
New_MST($G$)
```

```
1.  sort the edges in decreasing order of weight
2.  for each edge $e \in E$ in decreasing order of weight
3.      if $e$ is part of a cycle in $G$
4.          Remove $e$ from $G$: $G{:=}G - \{e\}$
5.  return $G$
```

Prove that the algorithm is correct.

Soln. Let $e_1, e_2, \ldots e_m$ be the sequence of edges in decreasing order of weights as considered by the algorithm. After considering the kth edge in decreasing order of weight, let G_k be the resulting graph. Let $G_0 = G$. The algorithm maintains the invariant that there is an MST of G in G_k. Base case: for $k = 0$ this is obvious. Suppose it is true for $k - 1$, for $\leq k \leq m$. Consider G_k. If e_k creates a cycle, then by deleting e_k from G_{k-1}, there is an MST of G in $G_{k-1} - \{e_{k-1}\}$, by the above property (a). But then, G_k is $G_{k-1} - \{e_{k-1}\}$. The other possibility is that e_k does not produce a cycle in G_{k-1} in which case $G_k = G_{k-1}$ and the invariant holds from the induction hypothesis.

(c) On each iteration, the algorithm must check whether a specific edge e is part of a cycle. Give a linear-time algorithm for this task and argue its correctness. (5)

Soln. This is a HW4b problem. Do a DFS from any vertex in the current graph and check if (i) e is a back-edge, or (ii) a back edge $e' = (c, d)$ is discovered and at that the time e' is traversed, both a, b are on stack, and d is an ancestor (may be proper or not) of both a and b. A pseudo-code is given in HW4b-soln.

The last check: whether ancestor or not is easy. Maintain an explicit stack for DFS. When DFS_Visit(u) is called, insert u on stack. Define a *level* attribute for each vertex. Initialize a global variable *level* to 0, and whenever, DFS_Visit(u) is called, in addition to all that the standard algorithm does, increment *level* and set $u.level =$ *level*. This is the current position of u in the stack when it was visited. Similarly, when u is finished and DFS_Visit is about to end (vertex is colored black), pop the stack, and decrement *level*. The global variable level is depth of the stack.

Now to check if d is an ancestor of a check if both are grey and $d.level \leq a.level$.

(d) What is the overall time taken by this algorithm in terms of $|V|$ and $|E|$. (5)

Soln. The bottleneck is the repeated invocation of DFS. This takes time $O((|V|+|E|)\,|E|)$. Deletion of an edge takes $O(1)$ time assuming doubly linked adjacency list or just mark it as "deleted".

Problem 5. In the 2-SAT problem, you are given a set of *clauses*, where each clause is the disjunction (i.e., OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). The problem is to assign each boolean variable to either TRUE or FALSE so that *all* clauses are satisfied (i.e., evaluates to true under normal Boolean logic). A clause is satisfied if there is at least one true literal in each clause. Here is an instance of 2SAT:

$$(x_1 \vee \bar{x_2}) \;\wedge\; (\bar{x}_1 \vee \bar{x}_3) \;\wedge\; (x_1 \vee x_2) \;\wedge\; (\bar{x}_3 \vee x_4) \;\wedge\; (\bar{x}_1 \vee x_4)$$

This instance has a satisfying assignment as follows: set x_1, x_2, x_3 and x_4 to TRUE, FALSE, FALSE and TRUE respectively.

(a) Give an example of a 2-SAT formula with four variables and with no satisfying assignment. (5)

Soln. Consider the following set of implications which are clearly unsatisfiable.

$$(x_1 \Rightarrow x_2) \wedge (x_2 \Rightarrow x_3) \wedge (x_3 \Rightarrow x_4) \wedge (x_4 \Rightarrow \bar{x}_1)$$

Since, $a \Rightarrow b$ is equivalent to $\bar{a} \vee b$, this is can be equivalently written as

$$(\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_4 \vee \bar{x}_1) \ .$$

You will now design an algorithm to solve the 2-SAT problem efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables (assume $x_1, \ldots, x_n$) and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: For each clause $(\alpha \vee \beta)$ of I, where, α and β are literals, place an edge from $\bar{\alpha}$ to β in E and place another edge from $\bar{\beta}$ to α in E. (Note that $\alpha \vee \beta$ is logicall equivalent to $\bar{\alpha} \Rightarrow \beta$ and also equivalent to $\bar{\beta} \Rightarrow \alpha$.

(b) Show that if G_I has a strongly connected component containing both x and $\bar{x}$, then I has no satisfying assignment. (5)

Soln. There is a directed path from x to $\bar{x}$ and a directed path from $\bar{x}$ to x. Then, the first directed path is of the form $x, l_1, l_2, \ldots, l_k, \bar{x}$ and the second path is $\bar{x}, m_1, m_2, \ldots, m_r, x$. Here l_i's and m_j's are literals themselves as they occur on the path.

Each edge of the form (x, l_1) or (l_i, l_j) represents the clause $x \Rightarrow l_1$ or $l_i \Rightarrow l_j$ respectively. Similarly, corresponding to the second path the edges $(\bar{x}, m_1)$ or (m_i, m_{i+1}) represents the clauses $(x \Rightarrow m_1)$ and $m_i \Rightarrow m_{i+1}$. So if x is set to true, then, along the first path, each of the literals $l_1, l_2, \ldots, l_k$ is set to true forcing $\bar{x}$ to be true, which is impossible. On the other hand, if x is set to false, then, $\bar{x}$ is true, and then by the second path, each of the literals $m_1, m_2, \ldots m_r$ is set to true, which forces x to be true, again an impossibility.

We therefore conclude that I has no satisfying assignment.

(c) Now show the converse of (b) above, namely that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I is satisfiable. (*Hint*: Repeatedly pick a sink strongly connected component in G_I. Assign TRUE value to all literals in this sink component, and assign FALSE to their negations, and delete all of them. Argue that this discovers a satisfying assignment). (10)

Soln. Consider a strongly connected component of G_I. If any literal in this strongly connected component is assigned true, then all literals in this strong component must be assigned true. It follows that either all literals in a strong component are assigned true or all are assigned false. In this sense, a strongly connected component may be thought of as being assigned true (all literals in it are true) or false (all literals in it are false).

Now suppose that in the graph G_I^{SCC}, there is no component containing a literal and its negation. Let C be any strongly connected component. Then, we note that there is a component corresponding to C which we denote as $\bar{C}$. For every literal $l \in C$, there is the literal $\bar{l} \in \bar{C}$ and for the edge (l, m) corresponding to the implication $l \Rightarrow m$, there is the edge $(\bar{m}, \bar{l})$ corresponding to the equivalent implication $\bar{m} \Rightarrow \bar{l}$. Essentially, all literals are negated and all edge directions between literals are reversed between their negations. It is easy to see that $\bar{C}$ is a strong connected component and $\bar{\bar{C}} = C$.

Now consider the suggested algorithm. Let G_I^{SCC} be the strongly connected component graph of G_I. Consider a sink component C of G_I^{SCC}. Set all literals of C to true. This implies that the negations of every literal in C must be set to false. So for every strongly connected component C' such that C' has a literal whose negation is in C, we set every literal of C' to false.

Also G^{SCC} is symmetric. If there is an edge from C to C' in G^{SCC}, then, there is an edge from $\bar{C}'$ to $\bar{C}$. (Analogous argument). So, if C is a sink component, then, $\bar{C}$ is a source component. The algorithm assigns true to all literals in a sink component C and assigns false to its literals in $\bar{C}$. $\bar{C}$ is a source, and has no predecessor, so this is very safe: $\bar{C}$ is deleted.

$\bar{C}$ gets an all false assignment. (Note that all corresponding clauses evaluate to true!) This is a safe assignment in the following sense: suppose there is an edge $\bar{C}$ to C'' for some strong component C''. Now, C'' continues to have the option of being all true or all false, that is, $\bar{C}$ being false does not force C'' one way or the other. (For example, if there is an implication $l_0 \Rightarrow l_1$, and l_0 is false, then it does not force l_1 to be either true or false). The same can be said if there is a path from $\bar{C}$ to C''.

Thus removing $\bar{C}$ by assigning it false and removing them is always safe. Once a sink component C and source $\bar{C}$ have been removed, the graph is reduced and contains fewer variables. Now the same process can be repeated until the graph becomes empty.