ESO207: Data Structures and Algorithms *Time: 4 hrs.*

End Semester Exam Solutions *Maximum Marks: 250*

Instructions.

a. There are 6 questions, each carrying 50 marks. You may answer any 5 questions. The first two questions have parts with different problems. The remaining questions each pertain to a single problem.

b. Please answer all parts of a question together. You may leave blank spaces if you wish to return later to complete parts of the question.

c. The exam is closed-book and closed-notes. Collaboration of any kind is ***not*** permitted. You may not have your cell phone on your person. Please use the mounted clock in the rear of the examination hall for time reference.

d. You may cite and use results done in the class. If you are modifying an algorithm done in the course, you may emphasize on your modifications and its consequences.

e. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity with which you express it. Please be neat. Argue correctness of your algorithms.

f. For the problems pertaining to graphs, assume that the graph is represented as an adjacency list.

Problem 1. Answer to the point, with clear and concise correctness arguments and complexity analysis. You may refer to results done in the class, and emphasize any modifications or additions.

1. Give an $O(|V| + |E|)$-time algorithm to find the connected components of an undirected graph. (12.5)

   Soln. Run DFS on the graph and output each DFS tree. Running time is $O(V + E)$ for DFS. (Can run BFS with an outer loop that runs BFS from each vertex that is white. Output BFS trees).

2. Give an $O(|V| + |E|)$-time algorithm to detect whether a given directed graph has a cycle. (12.5)

   Soln. Run DFS and check if there is a back-edge. (u, v) is a back edge if v's color is gray when the edge (u, v) is traversed. Report graph to have a cycle iff there is a back-edge. Proof. *if*: Obvious. Back-edge (u, v) together with the tree edges from v to u is a cycle. *only if*: Let $C = \langle u_1, u_2, \ldots, u_k, u_1\rangle$ be a cycle. Let u_i be the earliest vertex of the cycle that is discovered. Consider the vertex u_{i-1}. There is a path $u_i, u_{i+1}, \ldots, u_{i-1}$ along the cycle from u_i to u_{i-1}, all of whose vertices are white at the time u_i is discovered (since u_i is the first vertex discovered in the cycle). Hence, u_{i-1} is a descendant of u_i. Thus the edge (u_{i-1}, u_i) is a back-edge. Hence each cycle gives a back-edge.

3. In the DFS of a directed graph, suppose there is an edge (u, v) such that $u.d > v.f$. Can this edge be part of a cycle in G? Give an example, or prove that it is not possible. (12.5)

   Soln. An example graph and a possible DFS tree is given below.

4. Consider a modification of the algorithm discussed in the class for finding the strongly connected components in a directed graph. Suppose instead, we first run *DFS* on the edge reversed graph G^R and order the vertices in decreasing order of their finish times. Next we run *DFS* on the original graph but consider the vertices in this order, and output each DFS tree as it is produced. Does this modification output the strongly connected components? Give a proof or a counterexample. (12.5)

   Soln. Let C, C' be distinct strongly connected components of G^{SCC}. As shown in the class, if we run DFS on G and there is an edge from some vertex $u \in C$ to some vertex $v \in C'$, then, $f(C) < f(C')$.

Figure 1: A DFS tree is shown to the right. The dotted edges are the back edges or the cross edges, the normal edges are the tree edges. For every vertex, the start and finish times are shown as s/f. The DFS starts at a, visits v first and then visits u.

Thus, if we run DFS on the edge reversed graph, then, (a) the strongly connected components do not change, and, (b) $f(C') > f(C)$. Proof Outline: Let C be the component with the largest finishing time after running DFS on G^{rev}. Then, there is no edge from any vertex of C to any vertex of any other component D, otherwise, $f(D)$ would be $> f(C)$. By doing the DFS from the vertex $u \in C$ such that $f(u) = f(C)$, all vertices in C are discovered in the DFS tree. Since there are no edges to any other component, this DFS tree is output, which is the component C. The formal proof proceeds by induction. Let C_k be the component with the k th largest finishing time. The induction hypothesis is that for any $k \geq 1$, the kth DFS tree returns C_k. For $k = 1$, the proof was just given. For any general k, note that there can be edges from C_k only to C_j, for some $j < k$, but not to any $j > k$. Since all vertices in $C_1 \cup \ldots C_{k-1}$ are visited and are black, the DFS tree for C_k returns only C_k.

Problem 2. Answer to the point, with clear and concise correctness arguments and complexity analysis. You may refer to results done in the class, and emphasize any modifications or additions.

1. Given a *weighted directed acyclic graph* $G = (V, E, w)$, give an $O(|V| + |E|)$ algorithm for finding the longest path in G. Argue correctness and state and prove the complexity of the algorithm. (25)

   Soln. Add a dummy source vertex s and a dummy target vertex t. Add zero-weight edges from s to each vertex with no incoming edges. Add zero-weight edges from each vertex with no outgoing edges to t. Both these steps can be done in time $O(V+E)$. The longest path in the graph is now the same as the longest path from s to t. Negate the edge weights and run the single source shortest path algorithm from source s for directed acyclic graphs, as discussed in the class. This runs in time $O(V + E)$, by considering the vertices in a topological order and relaxing all the edges adjacent to a vertex. Return the negative of this shortest path length from s to t.

2. You are given a weighted directed graph $G = (V, E, w)$ with non-negative edge-weights, a set of source vertices S and a set of target vertices T. The shortest distance from S to T is defined as the minimum of the shortest distances from any vertex in S to any vertex in T, that is,

   $$\delta(S, T) = \min_{u \in S, v \in T} \delta(u, v) \ .$$

   The problem is to design an $O(|E| \log |V|)$-time algorithm to compute $\delta(S, T)$. Your algorithm must also be able to output one path from some vertex $s \in S$ to some vertex $t \in T$ such that the weight of this path equals $\delta(S, T)$. Give the algorithm (either explain clearly or give pseudo-code) and discuss its complexity. *Note*: An $O(|S|\,|E| \log |V|)$-time algorithm will not receive much credit. (25)

   Soln. Modify the graph to add a new dummy vertex s and add directed zero-weight edges to each vertex in S. Add a new dummy target vertex t and add directed zero-weight edges from each vertex in T to t. This can be done in time $O(|S| + |T|)$ by modifying the adjacency list appropriately. Now run Dijkstra's single-source shortest path algorithm from s. Then, $\delta(S, T) = \delta(s, t) = t.d$. Take the shortest path from s to t as returned by the algorithm and remove s and t to get a shortest path from some vertex in S to some vertex in T. The correctness is rather obvious.

Problem 3. *Divide and Conquer.* Given an array of n *distinct* numbers $a[1], a[2], \ldots, a[n]$, we say that (i, j) is an inversion if $1 \leq i < j \leq n$ and $a[i] > a[j]$. Note that if $a[1 \ldots n]$ is sorted in ascending order, then the number of inversions is 0 and if it is sorted in descending order, then the number of inversions is $\binom{n}{2} = n(n-1)/2$. In general the number of inversions may lie anywhere in between these limits. The problem is to design an $O(n \log n)$ time algorithm to count the number of inversions. Argue its correctness and analyze its complexity. A possible design approach is given below–you may choose to give an independent solution as well. (50)

Hint: (Let $n/2$ denote $\lfloor n/2 \rfloor$.) The number of inversions in the array $a[1 \ldots n]$ is the number of inversions in the array $a[1 \ldots n/2]$ + the number of inversions in the array $a[n/2 + 1 \ldots n]$ + the number of "cross-inversions", namely, the number of inversions of the form (i, j) such that i is an index in the first half and j is an index in the second half. Show that the number of "cross inversions" remains unchanged if each of the halves was sorted. Now give an $O(n)$ time algorithm to compute the number of "cross inversions" assuming each half of the array is sorted. Use this routine in the Conquer phase of a divide-and-conquer algorithm that both counts the number of inversions and sorts the array.

We first design an algorithm to count the number of "cross-inversions" between the array segment $a[i, \ldots, k]$ and $a[k+1, \ldots, j]$, assuming that both the segments are sorted in non-decreasing order. For any l, $i \leq l \leq k$, the number of cross-inversions of the form (l, r) is the number of indices $r \in \{k+1, \ldots, j\}$ that are smaller than $a[l]$. One way is to find the lowest index r_l, for a given $l \in \{i, \ldots, k\}$ such that $a[r_l] \geq a[l]$. Then, the number of cross-inversions of the form (l, r) is $r_l - (k+1)$. Further, as l increases, r_l increases, since the arrays are sorted. This can be written as follows.

```
Count-Cross-Inversions(a, i, k, j)      // assumes i < j
1.  l = i
2.  r = k + 1
3.  count = 0
4.  for  l = i to  k  {
5.       while  r ≤ j and a[l] > a[r] {
6.            r = r + 1
7.       }
8.       count = count +r − (k + 1)
9.  }
10. return  count
```

The complexity of this procedure is $O(j - i + 1)$, that is, it is linear time, since we make a single pass over the left segment and a single pass over the right segment.

We can now use the above algorithm to design a divide and conquer based algorithm that counts the number of inversions and sorts the array simultaneously. The top level call is *Count-Inversion-and-Sort*$(a, 1, n)$. We will assume the *Merge*(a, i, k, j) procedure from *Merge-Sort* that merges the sorted sub-arrays $a[i \ldots k]$ and $a[k+1 \ldots j]$, assuming $j > i$.

```
Count-Inversion-and-Sort(a, i, j)
1.  if  i  == j
2.       return 0
3.  k = ⌊(i + j)/2⌋
4.  leftcount = Count-Inversion-and-Sort(a, i, k)
5.  rightcount = Count-Inversion-and-Sort(a, k + 1, j)
6.  count = leftcount + rightcount + Count-Cross-Inversions(a, i, k, j)
7.  Merge(a, i, k, j)
```

8. **return** *count*

The recurrence relation is $T(n) = 2T(n/2) + O(n)$, since, both the *Merge* and the *Count-Cross-Inversion* procedures take time $O(n)$ when $j - i + 1 = n$. The solution to this recurrence is $O(n \log n)$ with $T(1) = 0$.

Problem 4. *Dynamic Programming.* You are given n intervals $I_1, I_2, \ldots, I_n$, where the ith interval $I_i = [s_i, f_i]$ signifies a request for a resource from time s_i to time f_i, where, $0 \leq s_i < f_i$ and s_i, f_i are non-negative real numbers. Associated with the ith interval is its value v_i, which is non-negative. Intervals numbered i and j are said to be compatible if they have no (non-trivial) overlap. We have to select a subset $S \subset \{1, \ldots, n\}$ of the intervals that are all mutually compatible, that is, no pair of intervals in S overlap, and the value $\sum_{i \in S} v_i$ is maximized. Assume that the intervals are numbered in increasing order of their finish times, that is, $f_1 \leq f_2 \leq \ldots \leq f_n$. For any interval i, let $p(i)$ be the largest value $j \geq 0$ that is less than i such that the intervals I_j and I_i are disjoint. If no such interval exists, then $p(i) = 0$. The problem is to give an efficient algorithm for computing the maximum value among all mutually compatible sets of intervals. You may follow the steps below or give an independent algorithm, argue its correctness, complexity, and output a candidate optimal set S of compatible intervals.

1. Give an $O(n \log n)$ time algorithm to compute $p(1), \ldots, p(n)$. (10) Let $f[1 \ldots n]$ be an array of the finishing times, which is sorted, by assumption. Set $p(1) = 0$. For each $i \in \{1, 2, \ldots, n\}$, do a binary search to find the largest value of j such that $f_j \leq s_i$. If there is no such value, return 0. This takes $O(\log n)$ time for each i, for a total of $O(n \log n)$ time.

2. For $1 \leq i \leq n$, let $W(i)$ denote the maximum value of mutually compatible intervals from 1 to i. That is,

$$W(i) = \max_{\substack{S \subset \{1,\ldots,i\} \\ \text{Intervals with indices in } S \text{ are non-overlapping}}} \sum_{i \in S} v_i$$

   Design a recurrence relation for $W(i)$ and argue its correctness. Where exactly is the substructure optimality property used? (15)

   Either the interval I_i is included in an optimal compatible set for the first i intervals or it is not. Suppose it is included. Then, none of the intervals with indices in the range $p(i) + 1, p(i) + 2, \ldots, j - 1$ can be selected into the optimal subset S since they overlap with I_i. Further, every interval with index at most $p(i)$ does not overlap with I_i. That is, the set of intervals with indices in the range $\{1, 2, \ldots, p(i)\}$ is precisely the subset of $\{1, \ldots, i - 1\}$ that does not overlap with I_i. Hence any compatible set in this case must be a compatible set from the range $\{1, 2, \ldots, p(i)\}$, and therefore, must be the optimal compatible set from the first $p(i)$ intervals. Hence, the value must be $W(i) = W(p(i)) + v_i$ when I_i is included in the optimal compatible set (substructure optimality)

   Otherwise, I_i is not included in the optimal compatible set for the first i intervals. Then, $W(i)$ must be the optimal compatible set for the first $i - 1$ intervals, $W(i - 1)$ (substructure optimality). Combining, we get the recurrence relation:

$$W(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(v_i + W(p(i)), W(i-1)) & \text{otherwise.} \end{cases}$$

3. Translate the recurrence relation into a dynamic programming based pseudo code. Analyze the complexity of your algorithm, assuming that $p(1), \ldots, p(n)$ is already computed. (15)

   The procedure *Find-Opt-Cost* assumes that the $p[]$ array is already computed.

   Find-Opt-Cost$(v[1, \ldots, n], p[1, \ldots, n])\{$

```
1.  Create the array W[0, ..., n]
2.  W[0] = 0
3.  for  i = 1 to n {
4.       if  W[i − 1] < v[i] + W[p[i]]
5.            W[i] = v[i] + W[p[i]]
6.       else
7.            W[i] = W[i − 1]
8.  }
}
```

The complexity is $O(n)$.

4. Add lines to pseudo code to produce a set of intervals that have no pair-wise overlap and have maximum value. (10)

Create another array $S[1, \ldots, n]$ that for each i is 1 if I_i is chosen for the maximum value compatible subset for the first i intervals, and is 0 otherwise. The following modification records the choice.

```
Find-Opt-Cost(v[1, ..., n], p[1, ..., n]){
1.  Create the arrays W[0, ..., n], S[1, ..., n].
2.  W[0] = 0
3.  for  i = 1 to n {
4.       if  W[i − 1] < v[i] + W[p[i]]{
5.            S[i] = 1
6.            W[i] = v[i] + W[p[i]]
7.       }
8.       else {
9.            W[i] = W[i − 1]
10.           S[i] = 0
11.      }
12. }
}
```

We can now use the choice array S to output the subset.

```
Find-Opt-Soln(S, p[1, ..., n]){
1.  i = n
2.  while  i > 0 {
3.       if   S[i] == 1 {
4.            print interval I_i
5.            i = i − p[i]
6.       }
7.       else
8.            i = i − 1
9.  }
}
```

Problem 5. *Graph Search.* Let $G = (V, E)$ be a connected undirected graph. An edge $\{u, v\}$ is a bridge if its removal disconnects the graph. In the DFS of G, for each vertex v, define $v.l$ to be

$$v.l = \min(v.d, \min\{u.d \mid (v, u) \text{ is a backedge in DFS tree }\}, \\ \min\{w.l \mid w \text{ is a descendant of } v \text{ in the DFS tree }\}) \ .$$

1. Show that $\{u, v\}$ is a bridge iff either $u.d < v.d$, (u, v) is a tree edge and $v.l > u.d$, OR, $v.d < u.d$, (v, u) is a tree edge and $u.l > v.d$. (15)

   Proof: *if*: Without loss of generality, let $u.d < v.d$–the other case is symmetric. Then, v is a child of u. Consider the DFS tree and let A be the set of vertices including u or its ancestors in the tree and let T denote the set of vertices including v and its descendants. Since, $v.l > u.d$, there is no back-edge from v or any of its descendants to either u or any of its ancestors. In the DFS of an undirected graph, there are no cross edges. Hence, all paths from A to T pass through the edge $\{u, v\}$. Deleting this edge creates two components, one including A and another including T. Hence, $\{u, v\}$ is a bridge.

   only if: Suppose $\{u, v\}$ is a bridge. In the DFS of G let $u.d < v.d$, without loss of generality. Since, $\{u, v\}$ is an edge, v is a descendant of u (white path theorem). However, there are no other paths from u to v, since $\{u, v\}$ is a bridge. Hence, (u, v) must be a tree edge. [Otherwise, (v, u) would be a back edge, implying that $\{u, v\}$ edge lies on a cycle which contradicts that it is a bridge.] Let A and T be defined as in the earlier paragraph. Since $\{u, v\}$ is a bridge, all paths from A to T must pass through $\{u, v\}$. Hence, for any vertex w that is either v or a descendant of v, there cannot be a back-edge (w, x) such that $x \in A$. In other words, $w.l \leq v.d < u.d$, for every $w = v$ or w descendant of v in the DFS tree.

2. Add lines of pseudo-code to the standard DFS algorithm (given later) to compute $v.l$ for all vertices v. (15)

3. Further modify the code to find and print all bridges. (10)

```
DFS(G){        // G = (V, E)
1.   for each vertex v ∈ V {
2.        v.color = white
3.        v.π = NIL
4.   }
5.   time = 0
6.   for each vertex v ∈ V {
7.        if v.color == white
8.             DFS-Visit(G, v)
9.   }
}

DFS-Visit(G, u) {      // G = (V, E)
1.   u.color = gray
2.   time = time + 1
3.   u.d = time
4.   u.l = u.d
5.   for each vertex v ∈ Adj[u] {
6.        if v.color == white{
7.             v.π = u
8.             DFS-Visit (G, v)
9.             if v.l > u.d
10.                 print {u, v} is a bridge
11.            u.l = min(u.l, v.l)
12.       else     // v.color == gray
13.            u.l = min(u.l, v.d)
14.       }
15.  }
16.  time = time + 1
```

17. $u.f = time$
18. $u.color = black$
}

4. Analyze the complexity of your algorithm. (10)

   This is linear time $O(V + E)$ algorithm, with constant additional processing per edge compared to standard DFS.

Problem 6. *Infection Spreading.* Consider a directed graph $G = (V, E, w)$, where the vertices are computers and there is an edge from one computer to another if the first computer has sent a message to the second one. Further, for an edge (u, v), $w(u, v)$ denotes the *time* when this message was sent (assume that at most one message is sent from one computer to another). Let s be a given source vertex that is known to be infected by a virus at a given time t_0. If s communicates to another computer u at time t_0 or later, then, we assume that u is infected with the virus. In general, if a computer u is infected at time t_u and sends a message to v at time t_u or later, then it infects the computer v. Given the graph and the timestamps on the edges, the first infected vertex s and its time of infection t_0, design an efficient algorithm that determines, for every vertex, the time that it is first infected (which may be set to ∞ if the vertex is not infected). Prove the correctness of your algorithm and analyze its complexity.

Soln. A valid path of length k from s to another vertex u_k is a sequence $P = \langle s = u_0, u_1, u_2, \ldots, u_k \rangle$, where, (u_i, u_{i+1}) are edges, for $i = 0, 1, \ldots, k-1$ *and* the time stamps on the edges of the path are monotonic, that is,

$$w(u_0, u_1) \leq w(u_1, u_2) \leq \ldots \leq w(u_{k-1}, u_k) \ .$$

The cost of this path is defined as

$$w(P) = w(u_{k-1}, u_k) = \max_{i=0}^{k-1}(w(u_i, u_{i+1})) \ .$$

There are two possible solutions: One uses Bellman-Ford type sequence of edge relaxations, and the other uses Dijkstra's algorithm. Both solutions are discussed below and use the following notion of edge relaxation.

Bellman-Ford based solution.

For $u \in V$, let $u.d$ denote an upper bound (estimate) on the earliest time of infection of u. Initially, $u.d$ is set to ∞ for all vertices except s. $s.d$ is set to t_0.

INITIALIZE-SINGLE-SOURCE(G, s, t_0) {
1. **for** each vertex $u \in V$ {
2. $\quad u.d = \infty$
3. $\quad u.\pi =$ NIL
4. }
5. $s.d = t_0$

We change the notion of relaxation of edge is changed to the following. Suppose there is a valid path from s to u and $u.d$ is the current earliest time of infection of u. To relax the edge (u, v), we must ensure that (a) $w(u, v) \geq u.d$ so that the infection can spread from u to v, and (b) the distance to v reduces, that is, this path infects v earlier than it already it (if at all), or, $w(u, v) < v.d$.

RELAX$(G, (u, v), w)$ { // relax the edge (u, v)
1. **if** $w(u, v) \geq u.d$ and $v.d > w(u, v)$

2. $v.d = w(u, v)$
3. $v.\pi = u$
}

The basic idea is rather simple if there no multiple edges with the same timestamp (edge-weight). First consider this case. Since any valid path must have monotonic edge-weights, we can relax the edges in this order. Thus every path is considered, and the shortest path from the source to each vertex (or the earliest time of infection) is found.

Earliest-Infection-Time(G, s, t_0) { // no duplication in edge-weights
1. SINGLE-SOURCE-INITIALIZE(G, s, t_0)
2. Sort edges in E in non-decreasing order of their weights.
3. **for** edges (u, v) in E in sorted order of weights do
4. RELAX(G, u, v) }

This is an $O(|E| \log |E|)$ algorithm, the cost is dominated by the cost of sorting.

If there are multiple edges with the same time-stamp, we can use the Bellman-Ford algorithm to get $O(VE)$ time complexity. Although this may be an overkill, it is clearly correct.

Dijkstra's algorithm based solution The observation is that Dijkstra's algorithm works correctly with this notion of relaxation and distance (earliest time of infection).

DIJKSTRA(G, s, t_0)
1. *Single-Source-Initialize* (G, s, t_0)
2. Let Q be a min priority queue of vertices with key $u.d$
3. Q.*Make-Priority-Queue*(V)
4. **while** Q is not empty {
5. $v = Q$.*Extract-Min*$()$
6. **for** each vertex $x \in Adj[v]$ {
7. **if** $v.d \leq w(v, x)$ and $w(v, x) < x.d$ {
8. RELAX(v, x, w)
9. Q.*Update-key*$(x, x.d)$
10. }
11. }
12. }
}

Let $\tau(s, v)$ denote the earliest time of infection of the vertex v due to vertex s. *Proof of correctness*: The proof will proceed by induction on the number of the vertices dequeued. The induction hypothesis is as follows. Let S be the set of vertices not in the priority queue at the beginning of an iteration of the while loop, that is, $S = V - Q$. Hypothesis: For every vertex $u \in S$, $u.d = \tau(s, v)$. The hypothesis is clearly true initially, and after the first iteration, since $S = \phi$ initially, and after the first iteration, $S = \{s\}$ and $s.d = t_0 = \tau(s, s)$.

Suppose the vertex v is inserted in the $k + 1$st iteration. Consider the scenario just prior to the *Extract-Min* operation. For sake of contradiction, assume that there is a shorter path from s to v. In that case, let P denote the shortest path from s to v. Since $v \notin S$, this path P must leave the set S–let v' be the first vertex on P that is not in S and let u' be the vertex just prior to it in P. (u' could be s). Now, we know that by definition of the cost of path $w(P)$,

$$w(P) \geq w(u', v') \geq v'.d$$

where the last inequality follows because we have relaxed the edge (u', v') when u' was inducted into S. However, we know that $v'.d \geq v.d$, since the *Extract-Min* operation chose v to be a vertex with the minimum distance in Q. Combining, we have,

$$w(P) \geq w(u', v') \geq v'.d \geq v.d$$

But we assumed that $w(P)$ was shorter than the current path to v whose distance is $v.d$–contradiction. Hence, we conclude that $v.d$ is actually the shortest distance, that is, $v.d = \tau(s, v)$.

The complexity of the algorithm is $O(E \log V)$ as in Dijkstra's algorithm.