Sample Solutions to Problem Set 3

(The solutions to the exercises from the book are adapted from those provided by the authors.)

1. (10 points) DAGs and cycles

Chapter 3, Exercise 3, page 107.

Answer: We apply the toplogical sort algorithm (for example, the one in the text). Recall that the algorithm repeatedly finds a node that has not incoming edge, "outputs" it to the topological sorted list, and removes the outgoing edges. If the algorithm is always able to find a node that has no incoming edge, then on termination we have a topological sorted order of nodes in a DAG. But if the algorithm reaches an iteration where every node has at least one incoming edge, then this indicates a cycle which we have to find.

We start from any node and follow an incoming edge (backward). Since every node has at least one incoming edge, we keep doing this until we return to a node v we have already passed. The set of nodes encountered between these successive visits to the node v form a cycle (traversed in the reverse direction).

2. (10 points) Number of shortest paths in social networks

Chapter 3, Exercise 10, page 110. (*Hint:* Use breadth-first search.)

Answer: We will solve the more general problem of finding the number of shortest paths from v to every other node.

We perform a BFS from v, obtaining a set of layers $L_0, L_1, \ldots$, where $L_0 = \{v\}$ and L_i is the set of nodes i hops away from v. For a node x, say in L_i, every path from v to x that goes through a node in L_1, then a node in L_2,..., then a node in L_{i-1}, and then to x is a shortest path. (This follows from the definition of BFS.)

We use the BFS routine to compute the number of shortest paths for each node x. Let $S(x)$ denote this number for a node x. For each node x in L_1, $S(x)$ is 1 since the only shortest-path consists of the single edge from v to x. Now consider a node y in layer L_j, for $j > 1$. The shortest paths from v to y all have the following form: they are a shortest path to some node x in L_{j-1}, and then they take one more step to get to y. Thus, $S(y)$ is the sum of $S(x)$ over all nodes x in layer L_{j-1} with an edge to y.

After performing BFS, we can thus compute all these values in order of the layers; the time spent in computing $S(y)$ is at most the degree of y. Since the sum of the degrees is twice the number of edges, the total running time is $O(m)$ plus the running time of BFS, which is $O(n + m)$. So the total running time is $O(n + m)$.

3. (10 points) Scheduling daily jobs

Chapter 4, Exercise 17, page 197.

Answer: In the algorithm we saw in class for interval scheduling, we arranged the jobs in order of increasing finish times, and then applied a greedy procedure. In the current context, there is no inherent order since time is "cyclic".

In order to use our original approach, we need to break symmetry somehow. Here is one way to do this. Suppose the intervals are labeled I_1 through I_n. Suppose we focus on solutions that contain interval I_1. We will try to find the best solution that contains I_1. We can do this by removing I_1 and all its overlapping intervals and find the best solution among the remaining intervals. The instance consisting of the remaining intervals has all intervals starting after the finish time of I_1 and finishing before the start time of I_1 in clockwise order. Such an instance can be handled by the greedy algorithm we studied in class.

The same way as we found the best solution containing I_1, we can find the best solution that contains I_2, the best solution containing I_3, etc. Then we compare the pick the best of these n possible solutions.

The running time is at most n times the time taken to pick the best solution containing I_1. If the intervals are sorted in clockwise order according to their finish times, then such a solution takes $O(n)$ time. We need to sort once overall, which will take time $O(n \log n)$. So the total time is $O(n^2 + n \log n) = O(n^2)$.

4. (15 points) Project management

Suppose you are a high-level manager in a software firm and you are managing n software projects. You are asked to assign m of the programmers in your firm among these n projects. Assume that all of the programmers are equally competent.

After some careful thought, you have figured out how much benefit i programmers will bring to project j. View this benefit as a number. Formally put, for each project j, you have computed an array $A_j[0..m]$ where $A_j[i]$ is the benefit obtained by assigning i programmers to project j. Assume that $A_j[i]$ is nondecreasing with increasing i. Further make the economically sound assumption that the marginal benefit obtained by assigning an ith programmer to a project is nonincreasing as i increases. Thus, for all j and $i \geq 1$, $A_j[i+1] - A_j[i] \leq A_j[i] - A_j[i-1]$.

Design a greedy algorithm to determine how many programmers you will assign to each project such that the total benefit obtained over all projects is maximized. Justify the correctness of your algorithm and analyze its running time.

Answer: We assign programmers one at a time, from 1 to m. At the ith step, we greedily assign the ith programmer to the project to which the programmer will bring the most marginal benefit. Note that the marginal benefit that the ith programmer brings to a project j depends on the particular assignment of the previous $i-1$ programmers; it may not be the same as $A_j[i]$.

We now prove that the above greedy choice leads to an optimal solution using a standard "swapping argument". Our main claim is this: for all i, the assignment obtained after the ith step is the subset of an optimal assignment. We prove the claim by induction.

The induction basis is trivial since we start with an empty assignment which is clearly a subset of an optimal assignment. As the induction hypothesis, we assume that the claim holds after step $i-1$. That is, the assignment obtained after step $i-1$ is a subset of an optimal assignment, say S. We now consider the state after step i. Let us suppose that the greedy choice assigns the ith

programmer to project j, thus resulting in k programmers for project j. If S has assigned at least k programmers to project j, then there is nothing to prove. (Note that by the induction hypothesis, S has assigned at least $k-1$ programmers to project j.) Consider the case in which S has assigned only $k-1$ programmers. This implies that there must be some other project j' such that S has assigned more programmers than the greedy assignment has assigned thus far. By the definition of our greedy choice and the assumption about marginal benefits, it follows that if we modify S by removing one programmer from project j' and move it to j, the change in total benefit cannot decrease. Therefore, the new assignment thus obtained by modifying S is optimal. It also contains the current greedy assignment, thus establishing the induction step.

We now consider the running time of the greedy algorithm. We have m iterations and in each iteration we are determining the project for which the marginal benefit will be maximum. Thus, a naive n-element comparison to determine the maximum will take $O(n)$ time, leading to a runtime of $\Theta(nm)$. One can do better by maintaining a priority queue in which we store an element for each project, representing the marginal benefit the next programmer will bring to the project. Initially, we add n entries to a priority queue, one for each of the projects, representing the marginal benefit that the first programmer brings to the project. This takes $O(n \log n)$ time counting $O(\log n)$ time for each insertion. (One can actually do this step faster in $O(n)$ time.) Each iteration then translates to one EXTRACT-MAX operation (to find the project to which the programmer will be assigned) and an INSERT operation to insert a new element giving the marginal benefit the next programmer will bring to the chosen project. Each iteration thus takes $O(\log n)$ time, thus leading to a total runtime of $O(n \log n + m \log n)$.

5. (15 points) Planning an expedition

Chapter 4, Exercise 18, page 197.

Answer: If the travel times along an edge do not vary with time, then clearly, the problem is the same as finding shortest path within a graph and we could use something like Dijkstra's algorithm.

Let us look at our current problem more closely. Consider going from the source s to a node v along a path in which the last step is to take edge $e = (u, v)$ from u to v. Although the time taken to traverse e varies over time, if we want to reach v at the earliest possible time by going through u, then we should aim at reaching u at the earliest possible time. This is because $f_e(t)$ is a monotonic function of t. So if we incrementally find nodes that we can reach the earliest (like in Dijkstra's algorithm), we could solve this problem. The other condition about f_e, that is $f_e(t) \geq t$, also seems helpful since this is equivalent to "no negative weights". We will now extend Dijkstra's algorithm to the given problem.

Let S be the set of explored nodes.
For each $u \in S$, we store the earliest time $d(u)$ when we can arrive at u
and the last site $r(u)$ before u on the fastest path to u
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from S for which
    $d'(v) = \min_{e=(u,v):u \in S} f_e(d(u))$ is as small as possible.
    Add v to S and define $d(v) = d'(v)$ and $r(v) = u$

We will prove by induction on the size of S that for all $v \in S$, the value $d(v)$ is the earliest time

that we can arrive at v.

The case $|S| = 1$ is clear since we know that we are at the source at time 0. Suppose the claim holds for $|S| = k \geq 1$; we now grow S to size $k+1$ by adding the node v. For each $x \in S$, let P_x be a path taken to reach x at time $d(x)$. Let (u, v) be the final edge on the path P_v from s to v.

Now, consider any other s-v path P; we wish to show that the time at which v is reached using P is at least $d(v)$. In order to reach v, this path must leave the set S somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

Let P' be the sub-path of P up to node y. By the choice of node v in iteration $k+1$, the travel time to y using P' is at least as large as the travel time to v using P_v. Since the time-varying edges do not allow traveling back in time, this means that the route to v through y is at least $d(v)$, as desired.

The algorithm can be implemented using a priority queue, as in Dijkstra's algorithm. The total running time is at most $O(m \log n)$.