Quiz 2 Solutions

Problem 1. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name there.

(b) [1 point] Flip back to the cover page. Circle your recitation section.

Problem 2. Short Answer [38 points] (4 parts)

(a) [9 points] Give an example of a graph such that running Dijkstra on it would give incorrect distances.

Solution: Below is one example of such a graph. There needs to be a vertex u such that when it is extracted, the distance to it is not the weight of the shortest path. But this alone is not enough: there needs to be a vertex v adjacent to u whose shortest path is through u. Since the edges from u get relaxed only once, then even though the distance to u could later be updated to the correct shortest distance, the distance to v will not be. Dijkstra will also yield incorrect distances for a graph with a negative-weight cycle.

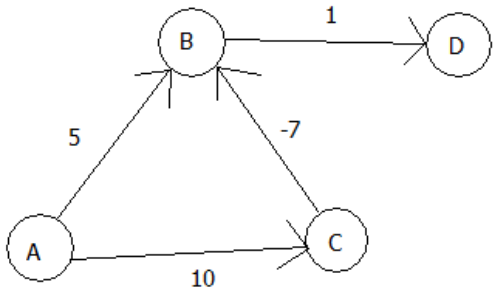

Figure 1: A gets extracted first, after which edges (A, B) and (A, C) are relaxed, and the distances are $d[A] = 0, d[B] = 5, d[C] = 10$. B is extracted next, leading to edge (B, D) being relaxed, and $d[D]$ becomes 6. D is extracted next, but it has no edges to relax. Finally, C is extracted, relaxing edge (C, B) and making $d[B] = 3$. The shortest path to D has weight 4, but $d[D] = 6$.

(b) [9 points] Give an efficient algorithm to sort n dates (represented as month-day-year and all from the 20^{th} century), and analyze the running time.

Solution: Use radix sort. First sort by day using counting sort with an array of size 31, then sort by month using counting sort with an array of size 12, and finally sort by year using counting sort with an array of size 100, where the counter in slot i corresponds to year $1900 + i$. The running time of radix sort is $\Theta(d(n + k))$. In this case, $d = 3$ and k is maximum 100, so the running time is $\Theta(n)$.

(c) [10 points] Give an $O(V + E)$-time algorithm to remove all the cycles in a directed graph $G = (V, E)$. Removing a cycle means removing an edge of the cycle. If there are k cycles in G, the algorithm should only remove $O(k)$ edges.

Solution: Do a DFS of the graph, and at the end, remove all the back edges. As you traverse the graph, you can check whether the edge you are trying to relax goes to a node that has been seen but is not yet finished, and if so, then it is a back edge and you can store it in a set. After the DFS, remove all the edges that are in this set.

(d) [10 points] Let $G = (V, E)$ be a weighted, directed graph with exactly one negative-weight edge and no negative-weight cycles. Give an algorithm to find the shortest distance from s to all other vertices in V that has the same running time as Dijkstra.

Solution: Let's say the negative-weight edge is (u, v). First, remove the edge and run Dijkstra from s. Then, check if $d_s[u] + w(u, v) < d_s[v]$. If not, then we're done. If yes, then run Dijkstra from v, with the negative-weight edge still removed. Then, for any node t, its shortest distance from s will be $min(d_s[t], d_s[u] + w(u, v) + d_v[t])$.

Problem 3. Path Problems [20 points] (2 parts)

We are given a directed graph $G = (V, E)$, and, for each edge $(u, v) \in E$, we are given a probability $f(u, v)$ that the edge may fail. These probabilities are independent. The reliability $\pi(p)$ of a path $p = (u_1, u_2, \ldots u_k)$ is the probability that no edge fails in the path, i.e.
$\pi(p) = (1 - f(u_1, u_2)) \cdot (1 - f(u_2, u_3)) \ldots \cdot (1 - f(u_{k-1}, u_k))$. Given a graph G, the edge failure probabilities, and two vertices $s, t \in V$, we are interested in finding a path from s to t of maximum reliability.

(a) [10 points] Propose an efficient algorithm to solve this problem. Analyze its running time.

Solution: Since the logarithm is a monotonic increasing function, maximizing the reliability $\pi(p) = (1 - f(u_1, u_2))(1 - f(u_1, u_2)) \cdots (1 - f(u_{k-1}, u_k))$ of a path is equivalent to maximizing $\log \pi(p) = \log(1 - f(u_1, u_2)) + \log(1 - f(u_1, u_2)) + \cdots + \log(1 - f(u_{k-1}, u_k))$, equivalent to minimizing $-\log \pi(p)$. Assign each edge a weight $w(u, v) = -\log(1 - f(u, v))$. These weights are all non-negative - and so we can apply Dijkstra.

Alternatively, simply modify the Dijkstra's algorithm (appropriately defining and initializing $d[u]$, replacing extract-min by extract-max, and using the relaxation step "if $d[v] < d[u](1 - f(u, v))$, then $d[v] = d[u](1 - f(u, v))$". These modifications work since $0 \leq f(u, v) \leq 1$ for all edge $(u, v) \in E$.

(b) [10 points] You tend to be risk-averse and in addition to finding a most reliable simple path from s to t, you also want to find a next-most reliable simple path, and output these two paths. Propose an algorithm to solve the problem, argue its correctness, and give its asymptotic running time.

Solution: We are not asking for a most efficient algorithm, simply a correct one. First notice that if the graph has no more than one simple path from s to t, then the problem has no next-most reliable simple path, and our algorithm should indicate this. We first found a most reliable simple path from s to t, if one exists. A next most reliable simple path must differ from it by at least one edge. So repeatedly resolve the problem after removing each edge of the initial path from G, one at a time, and chose among all these solutions the one that maximizes the reliability (if for each edge removal s is not connected to t anymore, the algorithm output "no next-most reliable path from s to t). This algorithm works since it will find a next-most reliable simple path that has to differ from the first one. It takes up to $k \leq n - 1$ iterations of Dijkstra, where k is the number of edges in the initial most reliable path.

Problem 4. Flight Plans [20 points]

When an airline is compiling flight plans to all destinations from an airport it serves, the flight plans are plotted through the air over other airports in case the plane needs to make an emergency landing. In other words, flights can be taken only along pre-defined edges between airports. Two airports are adjacent if there is an edge between them. The airline also likes to ensure that all the airports along a flight plan will be no more than three edges away from an airport that the airline regularly serves.

Given a graph with V vertices representing all the airports, the subset W of V which are served by the airline, the distance $w(u, v)$ for each pair of adjacent airports u, v, and a base airport s, give an algorithm which finds the shortest distance from s to all other airports, with the airports along the path never more than 3 edges from an airport in W.

Solution: As written, this problem asked that all nodes in the paths be within 3 edges of a node in W. So, we can solve this in two steps. First, we eliminate the nodes that are further than 3 edges away from a node in W. The most efficient way to do this is to create a supernode connected to all nodes in W and then run BFS to only four levels, eliminating all nodes not encountered. Alternately, using BFS the algorithm could run BFS as normal but start with a queue filled with all nodes in W. Other slower options include running BFS from every node in W or running Bellman-Ford with edge weights of 1 and nodes in W with starting weight 0. After eliminating the nodes which are further than 3 edges from a node in W, we can just run Dijkstra as normal. The running time of BFS in $O(V + E)$ is overtaken by Dijkstra's running time of $O(E + V \log V)$ giving that as the total. Solutions suggesting using multiple runs of BFS but skipping already-visited nodes from previous runs were invalid because the already-visited node may be reached at a shorter number of edges from a node in W, allowing more of its children to be included in the graph. Similarly, an algorithm just using Dijkstra but also tracking the distance from a node in W while running Dijkstra fails because there may have been a longer earlier path which would have run through a node in W.

Another interpretation of this question which was also accepted was that every node in the path must be within 3 edges of a node in W which is also in the same path. In this case, valid solutions used a graph transformation, making copies of each node for edge counts away from W, with each edge linking either to a higher-distance node or back to 0 if the node was in W. Similarly, another valid solution to this interpretation was to keep track of the shortest path thus far for each valid edge count away from W.

Problem 5. Tree Searches [20 points] (3 parts)

In this problem we consider doing a depth first search of a perfect binary search tree B. In a perfect binary search tree a node p can have either 2 or 0 children (but not just one child) with the usual requirement that any node in the left subtree of p is less than p and node in the right subtree is greater than p. In addition, all nodes with no children (leaves) must be at the same level of the tree. To make B into a directed graph, we consider the nodes of B to be the vertices of the graph. For each node p, we draw a directed edge from p to its left child and from p to its right child. An example of a perfect binary search tree represented as a graph is shown in Figure 2.

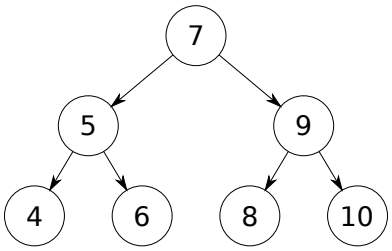

Figure 2: An example of a perfect binary search tree represented as a directed graph.

(a) [6 points] We structure our adjacency function such that at a node p, we first run DFS-VISIT on the left child of p and then on the right child. When we have finished expanding a node (i.e. just before we return from DFS-VISIT), we print the node. What is the first node printed? What is the last node printed? Give a short defense of your answer.

Solution: The first node printed will be the smallest node in the tree because DFS goes all the way down the tree before finally returning from a DFS-VISIT so that the first node it prints is in the bottom row. Since DFS first visits the left children, this will be the leftmost node in the bottom row. Since the bottom row is full, this node is the left child of its parent (which was the left child of its parent, etc) so it is the smallest node in the tree.

The last node printed will be the root node since this is the last node DFS finishes. Since the tree is perfectly binary, the root node is also the median of the tree.

Grading:

- 6/6: For something like the above answer. If you just did it on the example tree you received full credit provided you gave an explanation that showed you understood the order in which DFS expands node.
- 3/6: If got only one of the two right.

(b) [7 points] Does DFS print out the nodes of the tree in increasing or decreasing order? If yes, give a proof. If no, give a small counter example where the algorithm fails to print out the nodes in increasing or decreasing order and show the output of DFS on your example.

Solution: For the tree shown in Figure 2, DFS prints out

$$4, 6, 5, 8, 10, 9, 7$$

Grading:

- 7/7: For any counter-example
- 5/7: For a counter example that was not a perfect binary search tree.
- 4/7: If you showed a counter-example where DFS does not print out the nodes in order, but gave the wrong order or failed to give the output.
- 2/7: If you said DFS does not print out the nodes in order but gave no counter-example.
- 1-2/7: If you said DFS prints out the nodes in order, but gave a reasonable justification for why you might think that.
- 0/7: If you said DFS prints out the nodes in order and gave no justification.

(c) [7 points] Recall that usually when doing depth first search, we use the *parent* structure to keep track of which vertices have been visited. During the search, if a vertex v is in *parent*, the search will not run DFS-VISIT(v) again. Aspen Tu declares that *parent* is unnecessary when doing a DFS of B. She says that whenever the algorithm checks if a vertex v is in *parent*, the answer is always false. Do you agree with Aspen? If you do, prove that she is correct. If you do not, give a small counter-example where a depth first search through B will see a vertex twice. Remember, B is a directed graph.

Solution: Aspen is correct. Each node has only one incoming edge. When doing a DFS on a directed graph, we traverse each edge only once. Therefore, we can only see each node once in the search. You could also say that the search only produces tree edges and seeing a node twice requires a cross, forward, or back edge.

If you assumed the search did not start from the root then you do need the parent structure. This answer with correct justification also received full credit.

This question may have been confusingly worded. *Every* seen node is put into the *parents* data structure; not just the parents on the current path. This should have been clear from context, but naming the structure *parents* was a little misleading. It was done this way because that is how it was shown in lecture in class.

Grading:

- 7/7: For a correct answer with a good justification.
- 5/7: For saying the tree is a DAG but making it clear that you thought *parent* just stored current parents on the path. This is not true and we were trying to use the notation from class, but it was misleading.
- 4/7: For saying the tree is a DAG. This *does not* show you will not see a node twice. DAGs can have forward edges.
- 3/7 if you showed an example in which DFS would see a node twice that was not a perfect balanced binary search tree.
- 2/7: For the correct answer with no or very little justification.
- 1-3/7: For the wrong answer, but a justification that shows some knowledge of how DFS works.
- 0/7: For the wrong answer and no justification.

Problem 6. Computing Minimum Assembly Time [20 points]

As you might have heard, NASA is planning on deploying a new generation of space shuttles. Part of this project is creating a schedule according to which the prototype of the space shuttle will be assembled.

The assembly is broken down into atomic actions – called *jobs* – that have to be performed to build the prototype. Each job has a *processing time* and a (possibly empty) set of *required jobs* that need to be completed before this job can start – we will refer to this set as *precedence constraint*. Given such specification, we call an assembly schedule *valid* if it completes all the jobs and all the precedence constraints are satisfied.

Now, as the plan of the whole undertaking is being finalized, NASA has to compute the *minimum assembly time* of the prototype. This time is defined as the minimum, taken over all the valid assembly schedules, of the time that passes since the processing of the first scheduled job starts until the processing of the last job finishes. (Note that we allow jobs to be processed in parallel, as long as their precedence constraints are satisfied.)

As the prototype assembly is an immensely complex task, can you help NASA by designing an algorithm that computes the minimum assembly time efficiently? Prove the correctness of your algorithm and analyze its running time in terms of the number of jobs n and the total length of the required jobs lists m.

Formally, the assembly is presented as a list of n jobs $J_1, \ldots, J_n$, and each job J has a specified processing time, and the set of required jobs. We assume that there always is at least one valid assembly schedule corresponding to the given specification.

Example:

Job:	Processing time:	Required jobs:
J_1	1	$\{J_6, J_7\}$
J_2	6	$\emptyset$
J_3	4	$\{J_2, J_5\}$
J_4	2	$\{J_2, J_3\}$
J_5	3	$\emptyset$
J_6	5	$\emptyset$
J_7	7	$\emptyset$

Here, $n = 7$ and $m = 6$.

Solution: The minimum assembly time is 12.

(The corresponding schedule starts jobs J_2, J_5, J_6, J_7 at time 0, J_3 at time 6, J_1 at time 7, and J_4 at time 10.)

Solution: We start by augmenting the set of our jobs with two dummy jobs J_0 and J_{n+1} that have processing times equal zero. Furthermore, we make all the original jobs require J_0 to be

completed, and we define the set of required jobs of J_{n+1} to contain all the rest of the jobs. In this way, J_0 can be thought of as a "start" job and J_{n+1} as a "finish" job.

Next, we create a dependency graph that has one vertex per each job in our (augmented) set. For two jobs J_i, J_j, we put a directed edge from vertex J_i to vertex J_j if J_i is required by J_j. We set the weight of this edge (J_i, J_j) to be equal to the processing time of J_i. Note that the fact that there must exist at least one valid assembly schedule means that there is no circular precedence constraints and thus the dependency graph has to be a DAG.

It is easy to see that the minimum assembly time is just the *maximum* length of J_0-J_{n+1} path in this dependency graph.

This length can be computed by negating all the weights of the edges and finding the J_0-J_{n+1} distance $\delta(J_0, J_{n+1})$ in resulting graph – the minimum assembly time will be equal to $(-\delta(J_0, J_{n+1}))$. Since this graph is a DAG, we can compute this distance by running an appropriately modified version of Bellman-Ford that works in $O(m + n)$ time. (This version of Bellman-Ford was presented both in the lecture and in the recitations.) The total running time of this algorithm will be $O(m + n)$, as desired.