Quiz 2 Solutions

Problem 1. True or False [30 points] (10 parts)

For each of the following questions, circle either T (True) or F (False). There is no penalty for incorrect answers.

(a) T F [3 points] For *all* weighted graphs and all vertices s and t, Bellman-Ford starting at s will *always* return a shortest path to t.

Solution: FALSE. If the graph contains a negative-weight cycle, then no shortest path exists.

(b) T F [3 points] If all edges in a graph have distinct weights, then the shortest path between two vertices is unique.

Solution: FALSE. Even if no two edges have the same weight, there could be two *paths* with the same weight. For example, there could be two paths from s to t with lengths $3 + 5 = 8$ and $2 + 6 = 8$. These paths have the same length (8) even though the edges $(2, 3, 5, 6)$ are all distinct.

(c) T F [3 points] For a directed graph, the absence of back edges with respect to a BFS tree implies that the graph is acyclic.

Solution: FALSE. It is true that the absence of back edges with respect to a *DFS* tree implies that the graph is acyclic. However, the same is not true for a BFS tree. There may be cross edges which go from one branch of the BFS tree to a lower level of another branch of the BFS tree. It is possible to construct a cycle using such cross edges (which decrease the level) and using forward edges (which increase the level).

(d) T F [3 points] At the termination of the Bellman-Ford algorithm, even if the graph has a negative length cycle, a correct shortest path is found for a vertex for which shortest path is well-defined.

Solution: TRUE. If the shortest path is well defined, then it cannot include a cycle. Thus, the shortest path contains at most $V - 1$ edges. Running the usual $V - 1$ iterations of Bellman-Form will therefore find that path.

(e) T F [3 points] The depth of any DFS tree rooted at a vertex is at least as much as the depth of any BFS tree rooted at the same vertex.

Solution: TRUE. Since BFS finds paths using the fewest number of edges, the BFS depth of any vertex is at least as small as the DFS depth of the same vertex. Thus, the DFS tree has a greater or equal depth.

(f) T F [3 points] In bidirectional Dijkstra, the first vertex to appear in both the forward and backward runs must be on the shortest path between the source and the destination.

Solution: FALSE. When a vertex appears in both the forward and backward runs, it may be that there is another vertex (on a different path) which is further away from the source but substantially closer to the destination. (This was covered in recitation.)

(g) T F [3 points] There is no edge in an undirected graph that jumps more than one level of any BFS tree of the graph.

Solution: TRUE. If such an edge existed, it would provide a shorter path to some node than the path found by BFS (in terms in the number of edges). This cannot happen, as BFS always finds the path with the fewest edges.

(h) T F [3 points] In an unweighted graph where the distance between any two vertices is at most T, any BFS tree has depth at most T, but a DFS tree might have larger depth.

Solution: TRUE. Since all vertices are connected by a path with at most T edges, and since BFS always finds the path with the fewest edges, the BFS tree will have depth at most T. A DFS tree may have depth up to $V - 1$ (for example, in a complete graph).

(i) T F [3 points] BFS takes $O(V + E)$ time irrespective of whether the graph is presented with an adjacency list or with an adjacency matrix.

Solution: FALSE. With an adjacency matrix representation, visiting each vertex takes $O(V)$ time, as we must check all N possible outgoing edges in the adjacency matrix. Thus, BFS will take $O(V^2)$ time using an adjacency matrix.

(j) T F [3 points] An undirected graph is said to be *Hamiltonian* if it has a cycle containing all the vertices. Any DFS tree on a Hamiltonian graph must have depth $V - 1$.

Solution: FALSE. If a graph has a Hamiltonian cycle, then it is *possible*, depending on the ordering of the graph, that DFS will find that cycle and that the DFS tree will have depth $V - 1$. However, DFS is not guaranteed to find that cycle. (Indeed, finding a Hamiltonian cycle in a graph is NP-complete.) If DFS does not find that cycle, then the depth of the DFS tree will be less than $V - 1$.

Problem 2. Neighborhood Finding in Low-Degree Graphs [20 points]

Suppose you are given an adjacency-list representation of an N-vertex graph undirected G with non-negative edge weights in which every vertex has at most 5 incident edges. Give an algorithm that will find the K closest vertices to some vertex v in $O(K \log K)$ time.

Solution: We use a modified version of Dijkstra's algorithm for shortest paths. Suppose that we were to run Dijkstra's algorithm from v until we visited the $(K+1)$-st vertex (i.e. v plus K more). Then, these K vertices (not including v) would be the vertices we want.

However, we must make a modification. In the version of Dijkstra presented in class, we create a binary heap (or Fibonacci heap) and initialize the distance of all N vertices to ∞. We can't do that here, as that would require $O(N)$ time to initialize and as subsequent heap operations would take $O(\log N)$ time instead of $O(\log K)$ time. Thus, we start with an empty heap. Then, when we relax an edge, we insert the destination of the edge into the heap if it isn't already there.

The total time for this algorithm can be determined by the number of operations we perform. As each vertex has degree at most 5 and as we visit K vertices, we perform at most $5K$ `Inserts`, $5K$ `DecreaseKeys`, and K `ExtractMins`. Since we perform at most $5K$ `Inserts`, the size of the heap is at most $5K$ and all heap operations take $O(\log 5K) = O(\log K)$ time. Thus, our modified Dijkstra takes $O(5K \log K + 5K \log K + K \log K) = O(K \log K)$. (Using a Fibonacci heap results in the same asymptotic runtime.)

[Note: Many students lost points on this problem for not explaining how the heap needs to start empty and how it never grows beyond $5K$ elements.]

Problem 3. Word Chain [15 points] (3 parts)

A word chain is a simple word game, the object of which is to change one word into another through the smallest possible number of steps. At each step a player may perform one of four specific actions upon the word in play — either *add a letter*, *remove a letter* or *change a letter* without switching the order of the letters in play, or *create an anagram* of the current word (an anagram is a word with exactly the same number of each letter). The trick is that each new step must create a valid, English-language word. A quick exaple would be FROG $\rightarrow$ FOG $\rightarrow$ FLOG $\rightarrow$ GOLF.

(a) [5 points] Give an $O(L)$-time algorithm for deciding if two English words of length L are anagrams.

Solution: Iterate through each of the L letters in each word, tracking the frequency of each letter. If both words have the same counts, the words are anagrams. This is similar to counting sort and runs in $O(L+S)$ time where S is the size of the alphabet.

(b) [2 points] Give an $O(L)$-time algorithm for deciding whether two words differ by one letter (added/removed/changed).

Solution: Iterate through each of the words comparing each letter as you go. If the letters do not match and one word is longer, then move to the next letter in that word. If a mismatch is found twice, return false, otherwise return true at the end.

Where the previous part asked about identifying anagrams, this asks about the one-off changes other than anagram listed above. Solutions which looked for a one-off anagram were not accepted.

(c) [8 points] Suppose you are given a dictionary containing N English words of length at most L and two particular words. Give an $O(N^2 \cdot L)$-time algorithm for deciding whether there is a word chain connecting the two words.

Solution: Construct a graph with each word in the dictionary being a node. For each node, create an edge to another node if the function from either a or b return true. Then use BFS on this graph to determine if one word can be reached from the other. Building the graph takes L time for each comparison, done comparing each node to each other node, N^2 time. This takes longer than BFS, so total time is $O(N^2 \cdot L)$.

Problem 4. Approximate Diameter [15 points]

The *diameter* of a weighted undirected graph $G = (V, E)$ is the maximum distance between any two vertices in G, i.e. $\Delta(G) = \max_{u,v \in V} \delta(u, v)$ where $\Delta(G)$ is the diameter of G and $\delta(u, v)$ is the weight of a shortest path between vertices u and v in G. Assuming that all edge weights in G are non-negative, give an $O(E + V \log V)$-time algorithm to find a value D that satisfies the following relation: $\Delta(G)/2 \leq D \leq \Delta(G)$. You must prove that the value of D output by your algorithm indeed satisfies the above relation.
Hint: For any arbitrary vertex u, what can you say about $\max_{v \in V} \delta(u, v)$?

Solution: We run Dijkstra's algorithm for single source shortest paths (using a Fibonacci heap) with an arbitrarily selected vertex u as the source. Since the vertices are removed from the heap in non-decreasing order of distance from u, the distance from u to the last vertex in the heap is $\max_{v \in V} \delta(u, v)$. Thus, we can find $\max_{v \in V} \delta(u, v)$ in $O(m + n \log n)$ time. We output $\max_{v \in V} \delta(u, v)$ as D. Since $\Delta \geq \delta(u, v)$ for all $u, v \in V$, $D \leq \Delta$. Further,

$$
\begin{aligned}
D &= \max_{v \in V} \delta(u, v) \\
&\geq \max_{v_1, v_2 \in V} \frac{\delta(u, v_1) + \delta(u, v_2)}{2} \\
&= \max_{v_1, v_2 \in V} \frac{\delta(v_1, u) + \delta(u, v_2)}{2} \quad \text{(since the graph is undirected)} \\
&\geq \max_{v_1, v_2 \in V} \frac{\delta(v_1, v_2)}{2} \quad \text{(by triangle inequality of } \delta\text{)} \\
&= \frac{\Delta}{2}.
\end{aligned}
$$

Problem 5. Triple Testing [20 points]

Consider the following problem: given sets A,B,C, each comprising N integers in the range $-N^k \ldots N^k$ for some constant $k > 1$, determine whether there is a triple $a \in A$, $b \in B$, $c \in C$ such that $a+b+c = 0$. Give a *deterministic* (e.g. no hashing) algorithm for this problem that runs in time $O(N^2)$.

Solution: Perhaps the simplest solution involved Radix-Sort. We start by generating a set D of all pairs $a + b, a \in A, b \in B$; this takes $O(N^2)$-time. Then we sort D in $O(N^2)$ time using Radix Sort; this is possible since after adding $2N^k$ to each element in D, all elements in D become integers in the range $0 \ldots 4N^k$, where k is constant. Let $D'[1 \ldots N^2]$ be the sorted array. Now, for each $c \in C$, we check whether $-c \in D$; this can be done in $O(\log N)$ time per element c using binary search on D'. Since $|C| = N$, we can perform all checks in $O(N \log N)$ time. Overall, the running time is $O(N^2)$.

There are many variants of the above solution. Here are common examples:

- Instead of searching for $-c$, $c \in C$, in D', one can search for $-d$, $d \in D$, in a sorted version of C. This solution is correct, but takes $O(N^2 \log N)$ time, so it received only a partial credit.
- To find collisions between elements in D and the inverses of elements in C, one can (i) label each element to denote whether it comes from D or is the inverse of an element in C, (ii) perform Radix Sort on the union of D and the inverses of the elements in C and (iii) check if the sorted array contains any consecutive elements that are equal, and have different labels. This takes $O(N^2)$ time.
- One can use hashing to check whether an element $-c$, $c \in C$, belongs to D. Unfortunately, this involves using hash functions, which are either randomized (as in universal hashing) or heuristic (there are some sets on which the hash function has bad performance). As such, hashing was explicitly disallowed by the problem statement. Still, solutions involving hashing received some partial credit.

Overall, the best way to approach this was problem was to use Radix Sort (or some other form of sorting). Some people attempted to map the problem into some shortest paths problem, where the elements of A , B and C are used as edge weights. However, finding a, b, c such that $a + b + c = 0$ would typically require finding a path of length 0, which is quite different from finding the *shortest* path.

Problem 6. Number of Shortest Paths [20 points]

You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V+E)$-time algorithm that finds the number of shortest paths between the airport (the source vertex s) and the hotel (the target vertex t).

Solution: First we view the map as an (unweighted) undirected graph with locations as vertices and two locations are connected if and only if there is a unit-distance connection between the two locations. Then we do a breadth-first search (BFS) starting from the airport. We augment the data structure so that each vertex has an additional field paths to count the number of shortest paths from the root to that vertex. Initially we set $\mathrm{paths}(s) = 1$ for the root vertex s (that is, the airport) and $\mathrm{paths}(v) = 0$ for all other vertices. After each vertex (say v) is explored during BFS, we check all the neighbors of v and set $\mathrm{paths}(v)$ to be the sum of the paths of its neighbor nodes whose level is one less than the level of v. That is (recall that, for every $v \in V(G)$, $N(v)$ denotes the set of vertices adjacent to the vertex v),

$$\mathrm{paths}(v) = \sum_{w \in N(v) \text{ and } \mathrm{level}(w) = \mathrm{level}(v) - 1} \mathrm{paths}(w).$$

The correctness of the algorithm follows from the fact that all the shortest paths from the root node s to a node t at level k must have length k, and each of the shortest path is of the form $\langle s, v_1, \ldots, v_k = t \rangle$, where node v_i is some node at level i in the BFS tree and $1 \leq i \leq k$. The only modification to the original BFS is about the counter $\mathrm{paths}(v)$ for every vertex v, it is clear that initialization takes $O(V)$ time and updating the counter at any vertex v takes $O(|N(v)|)$ time. Note that $\sum_{v \in V(G)} |N(v)| = 2E$ and an ordinary BFS takes time $O(V + E)$, therefore the total running time of our modified BFS is $O(V + E) + O(V) + O(E) = O(V + E)$.