CSE 40113: Algorithms
Homework 6

You may complete this homework in groups of 3 or less students. Note that the integrity policy applies: your group should write up your own work, although you're welcome to work on the problems in a larger group. If you have any questions, please re-read both the homework guidelines and the academic integrity policy carefully, and then come discuss any questions or concerns with me.

Required Problems

1. For this problem, you're doing to put yourself in the mindset of a network engineer, working for a company that is facing a routing question. You have a connected graph $G = (V, E)$, where the nodes in V represent sites that wish to communicate with each other. Each edge e is weighted with a *bandwidth* $b(e)$, which is the maximum amount of information it can transmit.

   For every pair of nodes, the goal is to select a single path between the nodes along which they can communicate. However, the rate of transfer is limited by the smallest edge on that path, which we sometimes called the bottleneck edge: so for a path P, the bandwidth of the path is equal to $\min_{e \in P} b(e)$. (If there are no paths from s to t, then the bandwidth is $-\infty$.)

   One of the network designers has suggested that instead of storing a path for each pair of vertices, you could instead store a single spanning tree such that the unique path between vertices in the tree in fact achieves the largest possible bandwidth of any path. While skeptical, after some attempts to prove your colleague incorrect, you must admit that this may in fact be true.

   (a) Prove that there is a spanning tree T such that, for every pair of vertices s and t in the graph, the tree contains the maximum bandwidth path between s and t, and give an algorithm to compute it.

   Solution: We prove that it must be a tree first. If the set of such paths do not form a tree, then they must be either disconnected or contain a cycle. If disconnected, then we do not have a path for every pair of vertices, so that is easily ruled out. To see that there is not a cycle: suppose for the purposes of contradiction that there is a cycle in the union of all paths for every pair of vertex, and consider the edges on that cycle. Assuming unique edge weights (or that we consistently break ties), there will be some minimum edge. Every pair of vertices on the cycle has two possible paths in the cycle, one containing the minimum edge and one which does not. Each such pair will choose the path that avoids the minimum edge, though, since that path will have a larger bottleneck. Therefore, the minimum weight edge on the cycle is not in any communication path, and hence cannot be in the union of the paths.

   In fact, we can prove that for any subset $S \subseteq V$, the largest bandwidth tree always contains the maximum weight edge with exactly one endpoint in S. The proof is identical to the one in the book on page 260: Fix a set S, and suppose not. Let $e = \{u, v\}$ be the largest edge. The maximum bandwidth tree T must have some other u to v path, which must utilize some other edge e' going from a vertex in S to a vertex outside of S.

Remove e' to get a forest with two components, and then readd e to get a new spanning tree $T - e' + e$. Since $w(e) > w(e')$ as it was the largest possible edge, our bottleneck distance from u to v, as well as on any path that now uses e instead of e', has only gotten larger.

Note that this proof immediately suggests adapting a minimum spanning tree algorithm to solve this bottleneck tree algorithm: in Kruskal's algorithm on path 267, instead of sorting by increasing weight, sort by decreasing weight instead. Then, inside the loop, we let uv be the i^{th} heaviest edge. Otherwise, the code is unchanged, and still runs in $O(E \log V)$ time.

(You could also have adapted Prim or Boruvka's algorithm; I will accept any correct algorithm for this problem.)

■

(b) Describe an algorithm to solve the following problem, as efficiently as possible: Given an undirected weighted graph G, two vertices s and t, and a weight W, is the bottleneck distance between s and t at most W?

Solution: For this problem, we can decide more quickly than running the algorithm from part a. Instead, we can remove all edges with weight $\leq W$. If s and t are still connected, then we know there is a path connecting them with minimum edge weight that is $\geq W$, so the answer is no. If they are no longer connected, then every path between them must have some edge that is weighted $< W$, so the answer is yes.

```
BottleneckBound($G = (V, E), s, t, W$):
  for each vertex in $v$
      for each edge $e$ in $v$'s adjacency list
          if $w(e) < W$
              remove $e$ from $v$'s adjacency list
  mark all vertices as not visited
  put $s$ into a queue $Q$
  while $Q$ is not empty
      remove $v$ from $Q$
      if $v$ is unmarked
          mark $v$
          for each edge $vw$
              add $w$ to $Q$
  if $t$ is unmarked
      return True
  else
      return False
```

Correctness follows immediately: if t is not marked in the BFS traversal, then all s to t paths must use an edge of weight $< W$, so the bottleneck distance is $< W$. If t is marked, there exists at least one s to t path found in the BFS traversal using edges $\geq W$, so the bottleneck distance is not at most W.

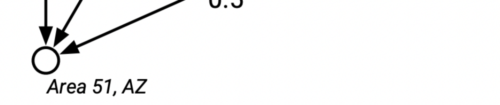
Runtime: the first loop takes $\sum\limits_{v \in V} d(v)$ time, which by the degree sum formula is $O(V + E)$, and the BFS traversal takes $O(V + E)$ as well, as it is identical to the one in Chapter 5. So, in total the algorithm takes $O(V + E)$ time.

Note: Using part a to solve this is slower, and hence would not get full credit. You also cannot loop over all edges and remove the edge from two adjacency lists in each iteration without taking $O(VE)$ time total, since removing something from a list requires finding it, taking $O(V)$ time each iteration.

■

2. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road won't be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area , Nevada. What route should he take so that he has the least chance of being abducted?

   More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The safety of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t. You may assume that all necessary arithmetic operations can be performed in $O(1)$ time.



   For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.)

Solution: This problem is asking you to modify Dijkstra. However, we can't use Dijkstra itself, as it minimizes the sum of edges, not the product. To solve it, you can either modify the graph with new weights, or can redo Dijkstra to do multiplication and prove that this correctly returns the safest path.

One solution is as follows: create G' with the same vertex and edge set as G, but for each edge e, re-weight the edge to be $\log(p(e))$ instead of $p(e)$. (This can be done in a simple for loop over the edges.) Then, run Dijkstra (page 285) to get the shortest path from s to t in G', and return that path.

Correctness: Consider two paths P_1 and P_2 using edges from E. If P_1 is shorter than P_2 in G', then P_1 has a lower product of probabilities of edges along the path than P_2.

Proof: If P_1 is shorter, than $\sum\limits_{e \in P_1} \log(p(e)) \leq \sum\limits_{e \in P_2} \log(p(e))$. Using the identify that $\log a + \log b = \log(ab)$, we can rewrite that inequality as $\prod\limits_{e \in P_1} \log(p(e)) \leq \prod\limits_{e \in P_2} \log(p(e))$, where $\prod$ denotes the product of each entry in the set (instead of sum of all the items in the set as in $\sum$). This means that the product of edge probabilities along P_1 must be less than in P_2, as desired.

Therefore, if we find the shortest path in G', this will also be the safest path in G.

Runtime: We convert edge weights in $O(V+E)$ time, by looping over each vertex's adjacency list in G and creating G' with new edge weights. We then call Dijkstra. Total run time is $O(V + E + E \log V)$.

■

3. After a grueling midterm, you are taking the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in South Bend. Unfortunately, no single bus will get you home, so you must change buses at least once. There are exactly b different buses. Each one starts running at 12:01am, makes exactly n stops throughout the day, and stops running at 11:50pm. Buses run exactly on schedule in this theoretical version of South Bend, and you have a perfectly on time watch to plan with. Finally, you are exhausted, so you don't want to walk between bus stops at all.

   (a) Describe and analyze an algorithm to determine the sequence of bus rides that get you home as *early* as possible. You goal is to minimize arrival time, NOT time spent traveling.

   Solution: There are several different ways to approach this problem, all boiling down to building a graph and running some version of Dijkstra. Here's one option:

   Build a graph G. First, create a vertex for each (bus, location, time) tuple on the schedule. If you have b buses, each making n stops, this will give $V = bn$. Label each vertex as its tuple.

   Next, for all vertex pairs, we add an edge from (b_i, l_j, t_k) to $(b_{i'}, l_{j'}, t_{k'})$ of weight $t_{k'} - t_k$ if:

   - $b_i = b_{i'}$ and $l_{j'}$ is the next stop on b_i's route after l_j: this corresponds to riding on the bus from one stop to the next
   - $l_j = l_{j'}$ and $t_k < t_{k'}$: this corresponds to waiting at a station for a later bus

Finally, add extra vertices s and t. Add 0 edges connecting s to any vertex that contains your starting location, weighted with the time difference between your test ending and the bus arrival time at that vertex. Next, add weight 0 edges from any copy of your home station to t.

This yields $E \leq (bn)^2$ edges, in the worst case.

Our algorithm will then just Dijkstra in $O(ElogV) = O((bn)^2 \log(bn))$ time to find the shortest s to t path, which we can turn into a schedule.

Claim (and proof of correctness for our reduction): any route home via the bus schedule will correspond exactly to a s to t path in G, and the length of the path is the amount of time spent returning home.

To prove the forward direction of claim: Consider a route home via the bus schedule, which is a series of buses to catch. Each such bus ride traces a path in our graph as follows: First, follow the edge from s to the vertex holding the station and bus arrival time of the first bus on your route. Then, when you stay on a single bus for multiple stops, you are following the first type of edges, and if you wait at the station for another bus if you take the second type of edge. Finally, at your "home" station, you can take the weight 0 edge to t. In all edges except the first and last, the weight of the edge is labeled with time spent waiting or riding, so the length of the path is time spent getting home.

For the other direction: consider any s to t path of length L in the graph. Such a path must go from s to some initial (bus, station, time) tuple, and then follow edges that correspond to either riding on a bus for multiple stops, or getting off at a station and waiting for a different bus (the second type of edge), before finally arriving at t along one final weight 0 edge. Since we have weighted the edges appropriately, the length of this path is still our time spent.

■

(b) Things just got stranger! There are now zombies infesting the city. The transit authority doesn't have the funding to zombie-proof the bus stops (although you're safe on a bus), so your goal just changed. Describe and analyze an algorithm to determine a sequence of bus rides that minimizes the *total time you spend waiting at bus stops*; you don't care how long you spend on a bus, or what time you get home. (Assume you can wait inside the building for that first bus to arrive, so you're safe for that waiting period - it's all the bus stops that are risky.)

Solution: For this problem, we can use an identical construction above, but the edges (b_i, l_j, t_k) to $(b_{i'}, l_{j'}, t_{k'})$ where $b_i = b_{i'}$ and $l_{j'}$ is the next stop on b_i's route after l_j (our first type of edge, which is riding on the bus) now are given weight 0. Things are still non-negative, and paths will only pay for time spent on the "waiting" edges. The runtime is unchanged, since we still run Dijkstra on the same graph, and the proof is essentially identical to above. ■

Hint: for this problem, not that I am NOT giving you a graph. So, if you're using any graph algorithm, you'll have to build the (presumably weighted) graph before you can use any of the algorithms we have covered! In other words, you're doing a reduction here, at least if you're using a graph algorithm (which I highly recommend).