Problem 1 (10 Points). Sometimes, there are many shortest paths between nodes (i.e. multiple paths that are at least as short as any other path). Show how to compute the number of shortest paths from a specific node v to each other node. Your algorithm should have the same running time as Dijkstra's algorithm

Solution: For this problem, we will need to assume all edge weights are positive.

Every shortest path to w has one of w's ancestors, u, as the second to last node in the path, and the path up until that point is a shortest path to u. Thus, the number of shortest paths to w is the sum of the number of shortest paths to each parent u that is part of some shortest path to w.

Thus, we modify Dijsktra's algorithm as follows: Keep track of a $numPaths(u)$ value for every node u. At each iteration, $numPaths(u)$ will be the number of shortest paths to u from v where all intermediate nodes have already been processed. At the beginning, we set $numPaths(v) = 1$ since there is only 1 shortest path from v to itself (any other path will be a cycle with positive length). For the rest, set $numPaths(u)$ to be 0. Then, when we process a node u, we modify the update to neighbor w as follows:

- If $dist(w) > dist(u) + weight(u, w)$:
  - $dist(w) = dist(u) + weight(u, w)$
  - $numPaths(w) = numPaths(u)$
- If $dist(w) = dist(u) + weight(u, w)$: $numPaths(w) = numPaths(w) + numPaths(u)$

The extra work per update is constant time, so the running time is the same as Dijsktra's algorithm.

To prove correctness, we need to prove that $numPaths(u)$ always contains the number of shortest paths from v to u where the intermediate nodes have all been processed. This is clearly true at the start, since no nodes have been processed, so there is only 1 shortest path to v, and none to any $u \neq v$.

Now suppose it is true before we process a node u. When we process u, we update the value of $dist(w)$ and $numPaths(w)$ for all edges (u, w) leaving u.

First, we argue that the shortest path to w with all intermediate nodes being processed and that goes through u must have u as the last intermediate node. Otherwise, there is some node x in the path after u. But x must already be processed since it is an intermediate node, and therefore there must be a path p to x whose length is at most the distance to u. But then the path to x through u will be strictly longer than the length of this path (since all edge weights are positive). Thus the path to w through u and then x cannot be a shortest path.

Now there are three cases:

- $dist(w) < dist(u) + weight(u, w)$: in this case, the shortest paths to w through the processed nodes including u must not pass through u, so $dist(w)$ and $numPaths(w)$ are already set correctly

- $dist(w) > dist(u) + weight(u, w)$: in this case, the shortest paths to w through the processed nodes not including u are longer than the ones through u, so we update the distance. At this point, all shortest paths to w pass through u as the last node, so $numPaths(w) = numPaths(u)$.
- $dist(w) = dist(u) + weight(u, w)$: in this case, there are shortest paths through the processed nodes not including u, and some additional shortest paths through u as the last node. Thus, we need to add $numPaths(u)$ to $numPaths(w)$.

Problem 2 (10 Points). Suppose that, in addition to weights on edges $weight(u, v)$, we also have weights on nodes $weight(u)$, and the length of a path is the sum of the weights of the edges and nodes on the path (including endpoints). We are now interested in computing the shortest path between two nodes using this new notion of length.

(a) In a directed graph, show how to reduce this problem to the standard shortest path problem. That is, show how to give new weights to edges $weight'(u, v)$ such that a shortest path in the standard shortest path problem using weights $weight'(u, v)$ is the same as the shortest path in the more general problem using weights $weight(u, v)$ and $weight(u)$. How are the lengths of the path under the two measures related?

(b) Do the same for an undirected graph.

Solution:

(a) Any path from u to v in the original problem has weight equal to the sum of the weights of all nodes and edges on the path. In the new graph without node weights, we want the weight of every path to be approximately the same as in the original graph. Therefore, we add the node weights to, say, all outgoing edges. That is, $weight'(u, v) = weight(u, v) + weight(u)$. In the new graph, the weight of a path is the sum of the new weights for all the edges. But this is just the sum of the old weights, plus the sum of all the nodes except the last node in the path. Thus, if p is a path from u to v, and $weight(p)$ is the weight in the original graph and $weight'(p)$ is the weight in the new graph, then

$$weight'(p) = weight(p) - weight(v)$$

Since all paths from u to v are modified by the same value, the ordering stays the same. Thus, the shortest path in the modified graph is the shortest path in the original graph, and the weight just differs by $weight(v)$, so we can easy recover the length in the original graph given the weight in the modified graph.

(b) Now we don't have a notion of incoming or outgoing edges, so we need to do something different. One possibility to to construct a directed graph where each undirected edge is replaced by one directed edge going in each direction, each with the same weight as the undirected edge. Then, we apply the transformation in part (a) to reduce the problem to solving the shortest paths problem on directed graphs.

We can also do a reduction to the undirected problem as follows: Instead of assigning all of $weight(u)$ to some set of edges, we assign half of it to each edge. Any path through u will

pass through 2 of the edges incident on u, so the total weight of u will be included in the path. The only exceptions are the endpoints, which only have half of their weight on the path. Thus,

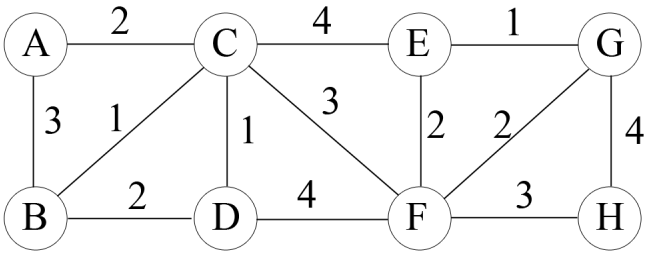
$$weight'(p) = weight(p) - (weight(u) + weight(v))/2$$

Just like before, all paths from u to v are changed by a constant amount, so the ordering is the same. Thus shortest paths in the new graph are shortest paths in the old graph, and we can easily get the length in the old graph by adding $(weight(u) + weight(v))/2$ to the weight in the new graph.

Problem 3 (0 Points). Removed

Solution:

Problem 4 (10 Points). Consider the following graph:



(a) In what order does Prim's algorithm add edges to the MST? Whenever there is a choice of nodes, choose the one that comes alphabetically first. Whenever there is a choice of edges, choose the one with the alphabetically first endpoint (if both share an alphabetically first endpoint, use the alphabetical ordering on the other end point).

(b) In what order does Krustkal's algorithm add edges to the MST? Use the same method for breaking ties as above.

Solution:

(a) (A, C), (C, B), (C, D), (C, F), (F, E), (E, G), (F, H).

(b) (B, C), (C, D), (E, G), (A, C), (E, F), (C, F), (F, H).

Problem 5 (30 Points). The following statements may or may not be correct. In each case, either prove it (if it is correct) or give a counter example (it it isn't correct). Always assume the graph $G = (V, E)$ is undirected and connected. Do not assume that edge weights are distinct unless this is specifically stated.

(a) If graph G has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a minimum spanning tree.

(b) If G has a cycle with a unique heaviest edge e, then e cannot be part of any MST.

(c) Let e be any edge of minimum weight in G. Then e must be part of some MST.

(d) If the lightest edge in a graph is unique, then it must be part of every MST.

(e) If e is part of some MST of G, then it must be a lightest edge across some cut of G.

(f) If G has a cycle with a unique lightest edge e, then e must be part of every MST

(g) The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST.

(h) The shortest path between two nodes is necessarily part of some MST.

(i) Prim's algorithm works correctly when there are negative edge weights

(j) (For any $r \geq 0$, define an r-path to be a path whose edges all have weight $< r$). If G contains an r-path from node s to t, then every MST of G must also contain an r-path from node s to t.

Solution:

(a) False. Suppose the heaviest edge is a bridge (removing it leaves two connected components). Then e is part of every tree, and hence every MST.

(b) True. Let T be any tree containing e. If we remove e from T, we disconnect it into two components. Consider the cut where S is one component, and $V - S$ is the other. Now, the cycle in question has e go across this cut, so it must have some other edge e' also going across this cut, and $weight(e') < weight(e)$. We add e', getting a new tree T' of lighter weight. Therefore, T could not have been a MST.

(c) True. Consider running Kruskal's algorithm, There is a way to sort the edges of G so that e is first. The tree produced by Kruskal's then is an MST containing e.

(d) True. Suppose T is a tree not containing e. If we add e, we get a cycle. Every other edge in that cycle has a higher weight than e, so remove one of them . WE are left with a new tree T' of lighter weight, meaning T could not have been a MST.

(e) True. Suppose we remove e from the MST T. This induces a cut where the two sets are the components of T. If e is not the lightest edge across this cut, we can add the lightest edge across this cut, getting a new tree T' of lighter weight.

(f) False. Consider a graph consisting of two cycles C_1 and C_2 of the same length which are disjoint except for the edge e and its endpoints. Suppose edges in C_1 have weight 1, edges in C_2 have weight 3, and e has weight 2. Then the unique lightest edge in C_2 is e. However, Kruskal's algorithm will first pick the edges of C_1, and then will skip e (since it forms a cycle).

(g) False. Consider a graph with three nodes u, v, and w, and $weight(u, v) = 11$, $weight(u, w) = 10$, and $wieght(v, w) = 2$. The MST clearly has the edges (u, w) and (v, w). However, if we run Dijsktra's from u, the shortest paths tree will consist of edges (u, v) and (u, w) since the shortest path from u to v or w is the direct path.

(h) False. In the previous example, the shortest path from u to v is the edge (u, v), which is not part of the MST

(i) True. If the most negative edge weight is $-N$¡ we can add $N + 1$ to each edge weight. Prim's algorithm will return the same MST in both the original and modified graph. This is because the cost of a node u is only compared to individual edge weights, which have all been increased by the same amount. Further, since all spanning trees have $|V| - 1$ edges, the weight of all trees has increased by the same amount, $(N + 1)(|V| - 1)$. Thus, the ordering of trees is the same, and thus the MST Prim's finds in the modified graph is an MST in the original graph.

(j) True. Suppose G has an r-path from s to t, and some tree T does not have an r-path from s to t. The path in T from s to t must be unique, otherwise we'd have a cycle. Let e be an edge on the path from s to t in T that has weight at least r. Remove e from T, which induces a cut separating the two components of T. Since there is an r-path from s to t, and because s and t will lie in different components, there is an edge e' of weight $< r$ crossing this cut. Adding this edge will result in a new tree T' of lighter weight than T, meaning T could not have been a MST.

Problem 6 (10 Points). In a spanning tree, a bottleneck is an edge with highest weight. A spanning tree is a *minimum bottleneck spanning tree*, or MBST, if no other spanning tree of the graph has a smaller bottleneck.

(a) Show that every minimum spanning tree is also a minimum bottleneck spanning tree.

(b) Is every minimum bottleneck spanning tree a minimum spanning tree? Prove or give a counter example.

Solution:

(a) Suppose an MST T is not a bottleneck spanning tree. That is, if w is the maximum weight edge in T, then there is some other tree T' whose edge weights are all less than w. Let e be the edge of weight w in T. If we remove e from T, we get a cut separating the two components of T. Since there is some tree with all edge weight being less than w, there is come edge e' of weight less than w crossing this cut. Adding e' to T gives us a tree of lighter total weight, meaning T could not have been an MST.

(b) Consider a graph with 4 nodes: A,B, and C form a cycle, with $weight(A, B) = weight(B, C) = 1$, and $weight(A, C) = 2$. There is also a node D connected to C with $weight(C, D) = 2$. Clearly, the MST consists of (A, B), (B, C), and (C, D), and the bottleneck in this tree has weight 2. Another tree consists of edges (A, B), (A, C), and (C, D). This graph is not an MST, but the bottleneck is still 2. Thus we have a tree that has the minimum bottleneck of 2, but it not an MST.

Problem 7 (10 Points). A small business — say, a photocopying service with a single large machine — faces the following scheduling problem. Each morning, they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e. an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$, and if job j is done right after job i, we could have $C_j = C_i + t_j$. Each customer also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of completion times, $\sum_{i=1}^{n} w_i C_i$.

Solution: Compute $r_i = t_i/w_i$, and do the tasks in order of increasing r_i. This clearly takes $O(n \log n)$ time, since the bottleneck of the algorithm is the sorting.

We will prove correctness in two steps: First, we will show that all solutions with no gaps and tasks ordered by r_i have the same cost. Then, we will show that the optimal solution has tasks ordered by r_i.

Suppose we have a solution with no gaps, and two adjacent tasks i and j with i immediately before i. Let T be the time just before we start task i. Then $C_i = T + t_i$, $C_j = T + t_i + t_j$.

The cost of this solution is

$$Cost = \sum_{k=1}^{n} w_k C_k = \sum_{k \neq i,j} w_k C_k + w_i(T + t_i) + w_j(T + t_i + t_j)$$

Now, suppose we swap i and j, obtaining a new potential solution. The completion times of tasks other than i and j is not affected. The completion time of i becomes $T + t_i + t_j$, and the completion time of j becomes $T + t_j$. Thus, the new cost is

$$Cost' = \sum_{k \neq i,j} w_k C_k + w_i(T + t_i + t_j) + w_j(T + t_j)$$

The difference in these costs is

$$Cost - Cost' = w_j t_i - w_i t_j = w_i w_j (r_i - r_j)$$

Let O_1 and O_2 be some orderings with tasks ordered by r_i. The only way O_1 can differ from O_2 is by reordering tasks with the same r_i. We can obtain any reordering of tasks with the same r_i by swapping adjacent tasks with the same r_i. But as shown above, swapping two adjacent tasks i and j with $r_i = r_j$ does not change the cost of the solution. Thus, O_1 and O_2 have the same cost.

Now we will show that the optimal solution has r_i in increasing order with no gaps. If there are gaps, we can move tasks earlier, decreasing the cost. If the tasks are not in order, there is some adjacent tasks i and j where i comes immediately before j, but $r_i > r_j$. The swapping i with j will result in a cost that has been decreased by $w_i w_j (r_i - r_j) > 0$, meaning the original solution could not have been optimal.

Thus, the optimal solution has no gaps and tasks in order of increasing r_i, so it has the same cost as all such solutions, including the greedy solution.

Total points: 80