Homework 7

Early deadline: Wednesday, October 21 at 11:59 PM
No penalty later deadline: Saturday, October 24 at 11:59 PM

Student: Justin DiEmmanuele
Collaborators: Shilpen Patel, George Padavick, Matthew Gilgo

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and, if helpful, pseudocode,
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Points might be subtracted for illegible handwriting and for solutions that are too long. Incorrect solutions will get from 0 to 30% of the grade, depending on how far they are from a working solution. Correct solutions with possibly minor flaws will get 70 to 100%, depending on the flaws and clarity of the write up.

Exercises (do not hand in) The following exercises are meant for practicing using the cut and cycle properties.

- Given an edge e in the MST of a graph G, find and output a cut S for which it is the lightest edge in the cutset.
- Given an edge e not in the MST, find and output a cycle C on which it is the heaviest edge.

Homework problems to hand in

1. (**MST Updates**) Suppose you are given a graph G (with unique edge weights) and its associated minimum spanning tree M, along with an edge $(u, v) \in G$ that we want to remove. Call the resulting graph with the edge removed G'. Assume that G' is still connected.

   (a) Argue that it's sufficient to change at most one edge in the MST for G so that it's the MST for G'. How do the cycle and cut properties help you find the edge to change? *Hint:* Think about two cases separately: $(u, v) \in M$ and $(u, v) \notin M$.

   (b) Write an algorithm that takes G, the MST of G, and an edge e in G, and outputs the minimum spanning tree of G' (again, assuming G' is still connected). Your algorithm should run in $O(n + m)$ time (do not recompute the MST from scratch; instead modify the MST you are given).

   Below are some examples of graphs with MST edges labeled, and the resulting G' and MST of G'

   (a) **It is sufficient to change at most one edge in the MST**

   - Case 1: $(u, v) \notin M$
     We claim that if the deleted edge (u, v) is not in the MST M we do not need to change M. For any edge e in M, we can remove e to create two separate trees, neither of which are spanning any longer. Let us call the two sets of nodes created A and B. By removing any edge e and taking the cut set (A, B), we can use the cut property outlined in the textbook to prove that the minimum weight edge in that set must be included in the new MST M'. Since the previous graph was an MST and the only deleted edge is one that was not in M M' must be equivalent to M or else the incoming tree M must not be an MST.
   - Case 2: $(u, v) \in M$
     The argument from case 1 still applies and shows that any edge in M that has not been deleted should be kept in M'. Now, we must decide which edge should take the deleted edge's place. Since we are guaranteed in the problem statement that the graph will still be connected after the edge is deleted we know there will be at least one candidate to connect the disconnected trees A and B created by deleting $e \in M$. We can use the cut property on the cut set (A, B) to add the lightest edge that connects those two sub-graphs. By the cut property we know that edge must be in M'.

   (b) **Write an algorithm to find the new MST**

- **A precise description of the algorithm**

Algorithm 1: NewMST(G, M, (u, v), W)

Input: G A graph in adjacency list form - tuples of (adj_node, weight)
Input: M Minimum spanning tree of graph G in adjacency list form
Input: (u, w) edge to be deleted in G
Input: W Weight of edge i

```
1  G' = G ;
2  G'[u][v] = 0;
3  min_weight = inf;
4  if M[u][v] = 0 then
       Output: M
5  group_1 = BFS(G', u );
6  group_2 = BFS(G', v );
7  for i = 0 to len(G) − 1 do
8      for j = 0 to len(G[i]) − 1 do
9          if (group_1[u] not null and group_2[v] not null) or (group_1[v] not null and group_2[u]
             not null) then
10             if G'[i][j][1] < min_weight then
11                 min_weight = G'[i][j][1];
12                 new_edge = (i, j);
13 M' = M ;
14 M'[u][v] = 0 ;
15 M'[new_edge[0]][new_edge[1] = 1 ;
```

Output: M'

The algorithm first checks in line 4 if the edge exists in the MST - if not it returns the old MST. It then creates a record of the nodes in the two disconnected trees created if the deleted node is in M. For each edge we then check if one node exists in group 1 while the other node exists in group 2. If this is true, we check the edge weight and update our minimum weight edge if this edge weight is lower than the known lowest edge weight. Finally, we add the new lowest weight edge that connects groups 1 and 2 to the new MST M' and return it.

- **Proof of correctness**

We have already proved that any edge in M after deletion will be in M' and that if the deleted edge is in M, we must simply take the lightest edge in the cut set between the two remaining tees.

This is exactly what the algorithm sets out to do, we must only prove that it achieves that goal. First, we initialize variables and set the new graph G' this is straightforward. Next, we find the disconnected trees in group 1 and 2. Since an MST is a tree, definitionally, deleting an edge in it will create two disconnected trees. We know that BFS will return the lengths to any node in a connected component so we

use that to find which nodes exist in which sub tree of the original MST.

We then iterate through all edges and only take a new lowest weight if the edge has one node in each of the disconnected trees. Since this iterates through every edge we are guaranteed to find the edge of lowest weight that connects the groups.

Finally, we add that edge to M' and return it to the user. Each step of the algorithm does what is needed to reach the stated goal.

- **Time and space complexity**
  Time Complexity: Up until line 5 we only call operations that run in $O(1)$ time. We then call BFS twice which we know runs in $O(m+n)$ time. We then reach nested for loops that will only run $O(m)$ times. All the operations within the innermost for loop are $O(1)$, setting variables and accessing arrays to make logical comparisons. Finally, the last few operations are all $O(1)$ time. The algorithm therefore runs in $O(m+n)$ time - limited by the BFS performed.
  Space Complexity: The Space complexity is the space it takes to store the graph or $O(m+n)$

2. (**Maximum clearance routes**) Every year, trucks get stuck on Storrow Drive because of the low bridges (sometimes called "storrowing"). Suppose you're running a trucking company in Boston with n warehouses. Each section of road $e \in E$ will be annotated with the clearance of the lowest bridge on that portion of road c_e. Assume all of the roads are undirected, and that all of the clearances are unique. We'll say that the *clearance of a route* from i to j is the minimum clearance of all the road segments on the route (you can't drive a truck that's taller than the lowest bridge on the route). Next, we'll say that a route is a *maximum-clearance route between i and j* if it is the route between i and j with largest route clearance out of all possible routes from i to j. Intuitively, the maximum-clearance route between two nodes tells you the tallest truck that you can drive between the two locations without getting stuck under a bridge, as well as which path to use.

   You want to find an algorithm that computes a tree for max-clearance routes from a given node s. After giving it some thought, you think it might be related to the idea of a maximum spanning tree with respect to the edge clearances.

   Note: the **maximum** spanning tree of a graph G is just a tree that connects all vertices in G (hence "spanning") while maximizing the sum of the edge weights in the tree. Cycles are not allowed, as we still want a *tree*.

   (a) State the cut and cycle property for *maximum* spanning trees (MaxST).

   (b) Show that in any graph G with unique edge clearances, the path given by using edges from the maximum spanning tree of G (with respect to the edge weights given by the clearances c_e) gives the maximum clearance route between any two warehouses. [*Hint:* There are a couple of natural ways to do this. One way is to proceed by contradiction: suppose that the highest-clearance path is not part of the MaxST; show that together with edges in the MaxST it creates a cycle that contradicts the cycle property above.]

   (c) Give a polynomial-time algorithm that takes a graph G with distinct edge clearances and outputs the maximum spanning tree. *Hint:* Modify any of the minimum-weight spanning tree algorithms from class.

   (a) **Cut and cycle property for maximum spanning trees**

   - **Cut Property**
     The cut property for MaxSTs is the following: Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the maximum cost edge with one end in S and the other in $V - S$. Then every maximum spanning tree contains the edge e. (Based on the property as defined in the textbook)
   - **Cycle Property**
     Assume that all edge costs are distinct. Let C be any cycle in G, and let edge $e = (v, w)$ be the least expensive edge belonging to C. Then e does not belong to any minimum spanning tree of G. (Based on the property as defined in the textbook)

   (b) **Show MaxST contains minimum clearance path between any two nodes**
   Assume M is the MaxST of graph $G(E, V)$ and there is an edge e that over which we can find higher clearance path from $u \in V$ to $v \in V$. If we add e to M we must create a cycle

since M is a spanning tree and e is an edge between two nodes in V. To keep the highest clearance path in the cycle we must remove the lowest weight edge in the newly created cycle per the cycle property stated in part a. If M was a MaxST before this, e would not have been added because it is a smaller weight path. Therefore, by contradiction, the MaxST must comprise of the highest clearance paths between all nodes.

(c) **Polynomial-time algorithm that outputs the MaxST**

- **A precise description of the algorithm** To write an algorithm that finds the maximum spanning tree all we need to do is modify Kruskal's algorithm to initially sort the edges in descending order instead of the ascending order used for minimum spanning trees.
- **Proof of correctness**
  To prove this algorithm: first we must prove the cut property in which Kruskal's algorithm relies on. For our MaxST version of the algorithm we are using the property as described in part a.
  - Let T be a spanning tree that does not contain an edge e but contains an edge e' which has a lower cost than e. We must show that we can exchange e' for e to create a new spanning tree with a higher cost. Let edge $e = (u, v)$ and $e' = (u', v')$.
  - Since T is a spanning tree, we know removing edge e' would create a gap between two connected trees that are no longer spanning. One tree must have u in it and the other must have v. We will call the set containing u S and the set containing v is therefore $V - S$ where V is the set of all the vertices in T.
  - Since removing e' creates a gap between S and $V - S$ the edge we chose must be from the set S to the set $V - S$ in order for the resulting graph to not have any cycles and be spanning.
  - Since e has one node $u \in S$ and the other $v \in V - S$ we can add e to create a spanning tree. Since e weighs more than e' the tree is a heavier spanning tree. If e is the heaviest edge bridging these sets it must me in the MaxST since this argument can be made against any other node.

  Kruskal's relies on the cut property for adding each node. When sorting by descending edge weight instead of ascending, we add the greatest edge spanning two sets that are disconnected. We know they are disconnected because we check that adding the new edge does not create a cycle. This principle is consistent with the maximum spanning tree rule as described in part a and proved above.
- **Time and Space Complexity**
  Time Complexity: Since the only change in Kruskal's algorithm is the way in which the edges are sorted, which has the same time complexity either way, the time complexity for the algorithm stays at $m \log n$
  Space Complexity: The Space complexity is the space it takes to store the graph or $O(m + n)$