Homework 7

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

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.

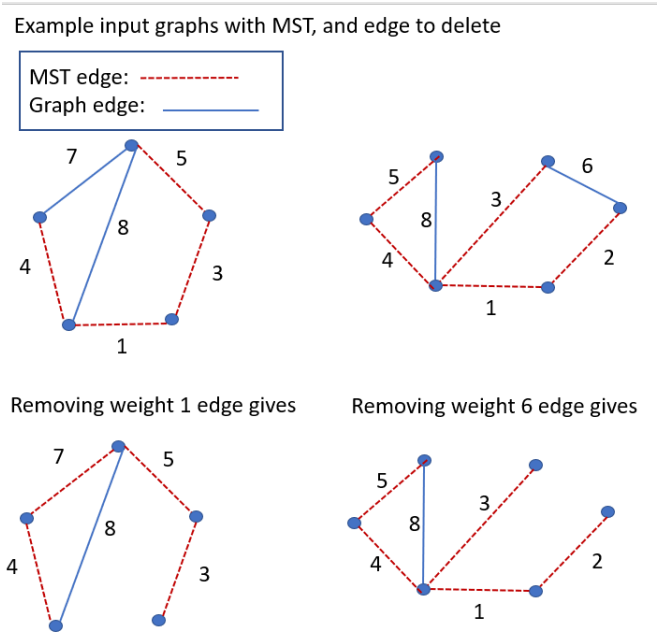
Homework problems to hand in

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'



ANSWER

a) We have two options here, if the edge we want to remove from G isn't in M, then we can just remove it, and the MST stays the same, and the graph just loses the edge.

Otherwise, if the edge is in the MST of G, then when we remove it, we are removing a bonding link between nodes in the MST, which means that M now has two unconnected parts. As such, we need to add another edge that connects them, so we are, at most changing one of the edges in M.

b)

The Algorithm we'd designed would take in G (the original graph), M (the MST of G) and e (the edge to be removed). From this, we'd make an initial check to see if e is in M, we can do this using a method such as $M.indexOf(e)$, if we this method doesn't return a number (ie: the edge is not in M), then we'd return M and finish the method.

If however we have a return value from the method, then we know the MST is affected, so we'll remove the edge from M, and save each of the separate node trees into their own variables, let this be t1 and t2, this would be a simple save, as we'd only really need to save each node and it's connection, just as if we were saving the root node of a tree.

From here begins the hard part, first we'd begin by initializing an empty variable *addingEdge*. In addition, we'd initialize a priority queue of edges called Q, then we'd loop while Q isn't empty getting the smallest edge in Q (de-queuing it), and on each iteration we'd check: if the edge in question is e, then we'd do nothing else this time and keep looping; if the edge is in M, if so we'd do nothing else this time and keep looping. Finally, if none of these are true, then we'd check if the two nodes that are connected by that edge are in different subtrees (ie: one of the nodes is in t1 and the other in t2 or vice-versa).

If this is the case, then we've found the smallest edge that connects the two subtrees t1 and t2, so we'd set *addingEdge* to it. At the end of this loop, we'd be left with *addingEdge* as the smallest edge that isn't yet in M and connects the two subtrees. So we'd just add it to M, and return M.

The algorithm will always work due to it's checks and their order, since we'll never add an edge that is already in the MST, nor will we add the edge we want to remove, and since we are always getting the smallest edge that fits the criteria, we'll always end up with the MST for the new graph.

The runtime of this algorithm will be, at most O(m+n), since it follows the basic looping structure of Dijkstra. As for space-complexity, it'll take O(n), since we need to maintain a queue that is, at most n long, plus an extra variable that is O(1), but can be discarded.

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.

ANSWER

a) Cut property, as from the book, altered to meet MaxST needs: 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.

Cycle property, as from the book, altered to meet MaxST needs: Assume all edge costs are distinct. Let C be any cycle in G, and let edge e = (v,w) be the least expensive edge belonging in C. Then e does not belong to any minimum spanning tree of G.

b) Let us says that there is in fact a path $p1$, that has an edge $e1$, that doesn't belong to MaxST, and that this path is the highest-clearance path between nodes u and v.

If this is true, then, when combining path $p1$, to the MaxST, we'll be adding $e1$ to the MaxST. However, if we do this, it'll create a cycle, as it is adding an alternative path from u to v, from the one in MaxST (which by definition of being a spanning tree already links all possible nodes with exactly 1 path).

Let this cycle be C, then, by the cycle property as written above, the least expensive edge in this cycle does not belong to the MaxST. This however, causes a contradiction. As for $p1$ to be the highest-clearing path between u and v, then its smallest edge must be larger than the smallest edge of any other path.

This means that, even if $e1$ is the smallest edge in $p1$, it'll still be larger than the smallest edge in the path between u and v as set out in the MaxST. Which in turn means that the smallest edge in C is one that is inside the MaxST. This is a direct contradiction of what the MaxST is, and, as such, could never happen.

c) If we use Kruskal's algorithm, yet we sort the edges in reverse order (ie: decreasing order of height-clearence), we'll achieve an algorithm that outputs the maximum spanning tree.

Using the Union-Find datastructure, we receive the graph, and begin by sorting the edges in decreasing order, this will take O(m log n) at most, since it runs exactly the way increase order would. Then we initiate an empty T which will be our MaxST.

From here, we make a union-find datastructure from V (vertices of the graph) to keep track of the connected components as edges are added. Then we begin looping through the edges in sorted order and each time check if the two nodes connected by the edge in question are in unconnected components, and if so add the edge to T and merge the components.

Since we are doing at most 2m Find operations and n-1 Union operations in total, and if we use a pointer-based implementation of Union-Find; we'll have a total runtime of O(m + n log n) for this part of the algorithm.

In the end, we have an algorithm that returns the MaxST in O(m log n), and works since it shares the exact logic of Kruskal's algorithm for the MST, except it orders edges differently.