COSC 302: Analysis of Algorithms — Spring 2018
Prof. Darren Strash
Colgate University

Problem Set 7 — Minimum Spanning Trees and Single Source Shortest Paths
Due by 4:30pm Friday, March 30, 2018 as a single pdf via Moodle (either generated via LaTeX, or concatenated photos of your work). Late assignments are not accepted.

This is an *individual* assignment: collaboration (such as discussing problems and brainstorming ideas for solving them) on this assignment is highly encouraged, but the work you submit must be your own. Give information only as a tutor would: ask questions so that your classmate is able to figure out the answer for themselves. It is unacceptable to share any artifacts, such as code and/or write-ups for this assignment. If you work with someone in close collaboration, you must mention your collaborator on your assignment.

Suggested practice problems, from CLRS: 23.1-3; 23.1-6; 23.2-4; 23.2-5; 24.1-1

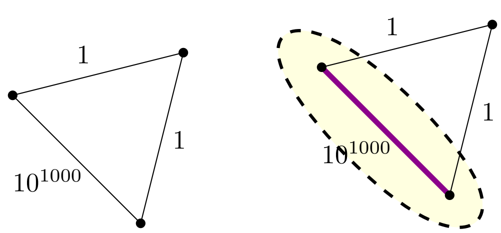
1. Problem 23.1-1 from CLRS.

   Solution:

   Proof. Form a cut $(V \setminus \{v\}, \{v\})$. Note that edge (u, v) is a light edge crossing the cut, and further that this cut respects the set of edges $\emptyset$. Then (u, v) is a safe edge, and is in some MST. $\square$

2. Problem 23.2-8 from CLRS.

   Solution: False, the algorithm does not always correctly compute an MST. Consider the graph depicted in Figure 1, which has an MST of weight 2. If we form cut $\{1, 2\}$ and recursively compute an MST on $\{1, 2\}$ and $\{3\}$ and combine the solution via the light edge of weight 1, then we get a spanning tree with weight $10^{100} + 1$, which is not a spanning tree of minimum weight.



(a) A graph. (b) A bad cut.

Figure 1: A counterexample to the correctness of the supposed divide and conquer MST algorithm.

3. Problem 24.1-6 from CLRS. *Hint: Consider running Bellman-Ford more than once.*

   Solution: First compute the strongly connected components of G. The negative cycle must exist in one of these components. For each component, we will run the Bellman-Ford algorithm $O(V)$ times. For each component, choose an arbitrary start vertex s in each component, and do not follow edges between components. We first conduct one run the of Bellman-Ford algorithm for each component, taking $O(VE)$ time overall. We run one more round of edge relaxation within each component, which we end when one edge is relaxed. We run the remaining calls to Bellman-Ford on the strongly connected component containing this edge. Call this edge (u, v). We compute a negative-weight cycle in time $O(V^2E)$ as follows.

   We show how to build a cycle backwards from (u, v). We run the Bellman-Ford algorithm at most $V - 1$ more times. During each run, we stop when an edge (w, u) entering u is relaxed, and add this edge to a linked list. We then repeat Bellman-Ford with with $u = w$. We mark all vertices that we reach in this way, and when a marked vertex (call it s) is reached, we stop the algorithm. Our negative cycle is the cycle formed beginning with the first edge incident to s in the linked list, to the last edge incident to s.

   We now show that this algorithm finds a negative-weight cycle in G if and only if there is a negative cycle.

   Proof. ($\Rightarrow$): By contraposition. If there is no negative-weight cycle, then the algorithm does not find one. After the first round of Bellman-Ford, the algorithm only relaxes an edge if there is a negative-weight cycle. Therefore, the algorithm will not continue, and will not return a negative-weight cycle if there is no negative-weight cycle.

   ($\Leftarrow$): If there is a negative-weight cycle, then the algorithm finds one. Note that edge (u, v) is not necessarily on a negative-weight cycle. However, there must be some incoming edge (w, u) to u that caused $d[u]$ to decrease before relaxing (u, v) after Bellman-Ford. This logic likewise applies to w after the second iteration of Bellman-Ford. Thus, for edge (u, v) to be relaxed, the algorithm will produce a sequence of edges such that each edge is relaxed. Note that, after V rounds of Bellman-Ford, this path has to contain at least 2 edges with the same vertex s, as the longest simple path has $n - 1$ edges and this procedure will find a path of at most n edges which contains at most n vertices, and therefore this additional edge must contain a vertex already in the path. Since there is a path between the two edges containing s, there is a cycle from s to itself and furthermore, at the moment the Bellman-Ford algorithm finds an outgoing edge from s that is relaxed, then we can relax all edges on the cycle ending at s, and this path therefore is a negative-weight cycle. $\square$

   Note that it is also possible to compute the cycle by following predecessors after running Bellman-Ford once, however the proof is complex.

4. Problem 24.3-4 from CLRS.

   Solution:

   Firstly, there should be exactly one vertex $s \in V$ that is a source. This s is such that $\pi[s]$ =NIL and $d[s] = 0$. Only one such vertex exists, as in a shortest path tree from s, all other vertices v with $\pi[v]$ =NIL are unreachable from s and therefore have $d[v] = \infty$. We find s by iterating through d and π simultaneously to find the unique vertex that has $d[v] = 0$ and $\pi[v] = \infty$. If multiple vertices meet this criteria, we return false.

   We now verify that the predecessor array π forms a tree containing s. For each vertex $v \neq s$ with $d[v] \neq \infty$ we add v and its predecessor (which may be s) to a new graph $G' = (V', E')$ with edge $(v, \pi[v])$. We verify that G' is a tree by checking that is is connected and contains $|V'| - 1$ vertices.

   We now verify that all vertices v with $d[v] = \infty$ are not reachable from s. We run a BFS from s and check that these vertices are white.

   Next, we verify that the tree formed by the predecessor array consists of edges in G. We do this by computing G^T, and checking, for each vertex v, that edge $(\pi[v], v)$ exists in G^T. This is done by iterating over the neighbors of each v. Now we are assured that the tree is the result of search on G from s.

   We now verify distances. We first verify that the distances are consistent with paths through the search tree. For each $v \neq s$ in V, we check that $d[v] = d[\pi[v]] + w(\pi[v], v)$. That is, the edge $(\pi[v], v)$ was presumable relaxed. If this is true for all v then the tree is consistent with running search, and the distances are consistent. However, how do we know it is a shortest path tree?

   For each vertex $v \neq s$ we check if any outgoing edge can be relaxed. We make the following claim.

   Claim 1. *Assuming that distance array d and predecessor array π form a search tree T from s with d values consistent with weights along paths in T, then T is a single source shortest path tree with s as the source if and only if no edge in G can be relaxed.*

   Proof.

   $(\Rightarrow)$: By contrapositive.

   Suppose some edge (u, v) can be relaxed. Then there is a shorter path from s to v through u and T is not a single-source shortest path tree from s.

   $(\Leftarrow)$: By contrapositive.

   Suppose T is not a shortest path tree. We show that there is some edge (u, v) can be relaxed. First, note that some subtree of T (containing s) is a shortest path tree from s since distances in T are consistent with edge weights in G. At the very least the distance from s to itself is correct, and s alone is a subtree of T. Since T is not a shortest path tree, let v be the vertex with smallest value $\delta(s, v)$ such that $d[v] \neq \delta(s, v)$. Then v has an incoming edge that can be relaxed. *Why?* All vertices with distance less than $\delta(s, v)$ are in the shortest path subtree of T, which is consistent with Dijkstra's algorithm. Then the next vertex to be added to the shortest path tree is v, since Dijkstra's algorithm adds vertices to the shortest path tree in order by distance. The distance $\delta(s, v)$ would then be computed by relaxing one of v's incoming edges. Therefore, this edge can be relaxed. $\square$