

CO453: Network Design – Winter 2007

Instructor: Chaitanya Swamy

Solutions to Assignment 1

Throughout $G = (V, E)$ with $|V| = n$, $|E| = m$.

Q1: The algorithm we will design will be very similar to Dijkstra’s algorithm. Let s denote the starting point, and $d(v)$ denote the time taken to travel from s to v along the fastest path, i.e., $d(v) = \min_{s \rightsquigarrow v \text{ paths } P} T(P)$, where $T(P)$ is the time taken to travel along path P . Observe that analogous to shortest-path distances, we now have $d(v) \leq f_e(d(u))$ for every edge (u, v) , and this will form the basis of our greedy algorithm. As in Dijkstra’s algorithm, we will maintain a set S of “explored” nodes, the ones for which we have correctly calculated $d(v)$, and we will maintain a label $\ell(v)$ for the nodes not in S , which is our current estimate for the least time taken to reach v from s . We will update $\ell(v)$ based on the above rule but only consider edges (u, v) where $u \in S$, then pick the node $w \notin S$ with smallest $\ell(\cdot)$ value and add it to the set S , and repeat.

- 1) Initialize $S \leftarrow \{s\}$ with $d(s) = 0$. Set $\ell(v) = \infty$ for all $v \notin S$.
Let $v^* = s$. [v^* denotes the last node added to S .]
- 2) While $S \neq V$,
 - (a) For every edge $e = (v^*, v)$ where $v \notin S$, update $\ell(v) \leftarrow \min\{\ell(v), f_e(d(v^*))\}$.
 - (b) Let $w \notin S$ be such that $\ell(w) = \min_{v \notin S} \ell(v)$.
 - (c) Set $S \leftarrow S \cup \{w\}$, $d(w) = \ell(w)$, $v^* = w$.

Here we have presented the improved implementation done in class that yields a better running time. The proof of correctness and running-time analysis also proceed as in Dijkstra’s algorithm. Consider any node w , and the set S just before w is added to it (in step 2(c)). First, note that there is an $s \rightsquigarrow w$ path P_w such that the time taken to travel along P_w is $d(w)$. Note that $d(w) = \ell(w) = \min_{u' \in S: (u', w) \in E} f_{u', w}(d(u'))$, so if $u' \in S$ is the node that attains the minimum in $\ell(w)$, then P_w is obtained by adding the edge (u', w) to the path $P_{u'}$ (which is obtained recursively). (We can easily keep track of these paths in the algorithm.) Now we show that $d(w)$ is indeed the least time taken to reach w along any $s \rightsquigarrow w$ path. Consider any $s \rightsquigarrow w$ path P . Let v be the first node on P that is not in S , and $u \in S$ be the node on P just before v . Let P' be the portion of path P from s to v . Let t_1 and t_2 be the times at which we reach u and v respectively by traveling along path P . In Dijkstra’s algorithm, we used the fact that edge costs are non-negative to argue that the cost of P is at least the cost of P' . Similarly, we will lower bound $T(P)$ here using the given properties of the f_e functions. Let $e_1 = (v, v_1), e_2, \dots, e_k = (v_{k-1}, w)$ be the portion of path P from v to w . Then the time taken to reach w by traveling along P is $T(P) = f_{e_k}(f_{e_{k-1}}(\dots f_{e_2}(f_{e_1}(t_2)) \dots)) \geq t_2$ since $f_e(t) \geq t$ for every edge e . Also, $t_2 = f_{u, v}(t_1) \geq f_{u, v}(d(u))$ since $t_1 \geq d(u)$ and the f_e functions are monotonic. Thus, we have $T(P) \geq f_{u, v}(d(u)) \geq \ell(v) \geq \ell(w)$, where the last two inequalities follow from the definition of the $\ell(\cdot)$ values, and since we choose w to add to the set S . This holds for any path P , which completes the correctness proof.

We call each operation in step 2(a) a **DecreaseKey** operation, and each operation in step 2(b) an **ExtractMin** operation. Treating each call to f_e as a single computational step, over all iterations, there are at most $|E| = m$ **DecreaseKey** operations since we examine an edge e at most once, and n **ExtractMin** operations. Thus the running time is the same as that of Dijkstra’s algorithm — it is

$O(m + n^2) = O(n^2)$ if we simply use an array to store the $\ell(v)$ values, $O(m \log n)$ if we use a data structure called a Priority Queue, and $O(m + n \log n)$ if we use a more sophisticated data structure like Fibonacci Heaps.

Q2(a): First note that T is a tree. It is clearly connected. If T contains a cycle C , then all the edges of C must have been added in a single phase since components in one phase are contracted before going to the next phase. Let $v_0, v_1, \dots, v_k = v_0$ be the nodes on C , and let e_i , $i = 1, \dots, k$ be the edge (v_{i-1}, v_i) . Edge e_1 must have been added due to either v_0 or v_1 , because it is the min-cost edge incident one of these nodes. We may assume without loss of generality that it is added due to v_1 (otherwise rename the nodes), so $c_{e_1} < c_{e_2}$. Then, $e_2 = (v_1, v_2)$ is added because of v_2 , so $c_{e_2} < c_{e_3}$. Continuing this way, since e_i must have been added because of v_i we get that $c_{e_i} < c_{e_{i+1}}$. Thus, $c_{e_1} < c_{e_2} < \dots < c_{e_k} < c_{e_1}$, which is a contradiction.

So T is a tree. The fact that it is a minimum spanning tree (MST) now follows easily from the cut property. Any edge $e = (u, v) \in T$ added because it is the minimum-cost edge incident to u is the minimum-cost edge across the cut $(S_u, V \setminus S_u)$ where S_u is the set of nodes contracted into the (super) node u . This follows since the edges incident to u are precisely the edges of this cut. Hence, T is a subset of the MST, and since it is a tree, it is *the* MST.

(b) This algorithm does not return an MST. Consider a graph with three nodes u, v, w , where the edge (u, v) has cost 1, edge (v, w) has cost 1.5, and edge (w, u) has cost 2. Then if we take the cut $(\{v\}, \{u, w\})$, the algorithm will produce the tree with edges (u, v) , (w, u) with cost 3, but the unique MST consists of the edges (u, v) , (v, w) with cost 2.5.

Given the statement of the algorithm, it is tempting to prove that the algorithm returns an MST using the cut property. But this argument breaks down because a min-cost edge e across some cut in $G[A] = (A, E[A])$ need not be the minimum cost edge across any cut in the original graph G .

(c) We prove the cycle property by contradiction. Suppose that the maximum-cost edge e of some cycle C is included in an MST T . Deleting e from T splits T into two components. Consider the cut $(S, V \setminus S)$ where S comprises the nodes of one of these components. C must cross this cut at least twice, so there is some edge $e' \in C \cap \delta(S)$, $e' \neq e$, so e' is not in T (since $T \cap \delta(S) = \{e\}$). Consider $T' = T \cup \{e'\} \setminus \{e\}$. T' is a tree since it has $n - 1$ edges and is acyclic. Also $c(T') < c(T)$ since $c_{e'} < c_e$, which contradicts the fact that T is an MST.

(d) The proof that T is an MST will follow from the fact that T is a tree and the cycle property. Any edge discarded by the algorithm is the maximum-cost edge in some cycle, and hence, by the cycle property cannot belong to any MST. Thus T is a *superset* of the MST; so if we argue that T itself is a tree, which is the more involved part here, then it follows that T is *the* MST.

T is acyclic. This is almost obvious from the algorithm description, but one small worry is that although, when adding an edge e to T that creates a cycle, the algorithm also deletes an edge e' from a cycle in $T \cup \{e\}$, there could be many cycles in $T \cup \{e\}$ — what is the guarantee that all such cycles are broken by removing e' ? But it is easy to see that if the final tree T contains a cycle, then at the *first* iteration at which the algorithm adds an edge e that creates a cycle, there is a *unique* cycle C in $T \cup \{e\}$, and C gets broken by the deletion of e' . Thus, T can never contain a cycle.

T is also connected. Consider any cut $(S, V \setminus S)$. Since G is connected, $\delta(S) := \{e = (u, v) : |S \cap \{u, v\}| = 1\} \neq \emptyset$. Consider the last iteration at which the algorithm considers an edge $e \in \delta(S)$ for addition. If $T \cap \delta(S) \neq \emptyset$ at this point, then $T \cap \delta(S)$ will continue to be non-empty at any later

iteration. This is because even if an edge $e' \in \delta(S)$ gets deleted (now or later) because it is part of a cycle C created by the addition of an edge, since C must cross the cut $(S, V \setminus S)$ at least twice, there will still be an edge remaining in $T \cap \delta(S)$. If $T \cap \delta(S) = \emptyset$ at this point, then e would be added to T and would be the unique edge across the cut $(S, V \setminus S)$. Since this is the last time we consider an edge of $\delta(S)$ for addition, at any later iteration, there is *at most one* edge of T in $\delta(S)$. This implies that any cycle C that gets created later by the addition of an edge in a subsequent iteration does not cross $\delta(S)$ (since $C \cap \delta(S) = T \cap \delta(S) \subseteq \{e\}$), and hence, e does not get deleted subsequently. In either case, we get that $T \cap \delta(S) \neq \emptyset$, and this is true for every cut, so T is connected.

Q3: With non-distinct edge costs, the algorithms of Prim and Kruskal (as stated in class) are not completely specified since we haven't specified a rule to resolve ties between equal-cost edges. This creates a little ambiguity in the question, namely, do we want to show that there is *some* tie-breaking rule under which Prim and Kruskal return an MST, or that Prim and Kruskal return an MST under *any* tie-breaking rule. We prove the latter stronger statement. We give two proofs.

The first proof uses the hint: we will argue inductively that the set T of edges maintained by Prim or Kruskal is a subset of an MST at all times. The proof will be identical for both Prim and Kruskal (which should not be surprising given that the proofs of correctness of both algorithms rely on the cut property). The base case is $T = \emptyset$, for which the statement is true trivially. Now suppose inductively the statement holds for T , so there exists an MST $T^* \supseteq T$. Suppose the algorithm (Prim or Kruskal) adds an edge e to T . The only property we will use is that e is a minimum-cost edge across some cut $(S, V \setminus S)$ where $T \cap \delta(S) = \emptyset$. Note that this is true of both Prim's and Kruskal's algorithms. If $e \in T^*$, then we are done, since $T^* \supseteq T \cup \{e\}$. Otherwise, we will modify T^* to obtain another MST T' that is superset of $T \cup \{e\}$. This is done via the same edge-exchange argument we used to prove the cut property. $T^* \cup \{e\}$ contains a cycle C , and there must be an edge $e' \in C \cap \delta(S)$, $e' \neq e$ (so $e' \in T^*$). Then $T' = T^* \cup \{e\} \setminus \{e'\}$ is a tree and $c(T) = c(T^*) + c_e - c_{e'} \leq c(T^*)$ since e is a min-cost edge in $\delta(S)$. Also note that $e' \notin T$ since $T \cap \delta(S) = \emptyset$. Thus, T' is an MST containing $T \cup \{e\}$, completing the induction step and the proof.

The second proof is based on the perturbation argument that we sketched in class. The idea is to show that for any tie-breaking rule used by Prim or Kruskal, we can tweak the edge costs by different, extremely small amounts, so that all edge costs become distinct, and such that (a) Prim or Kruskal, when run on the perturbed instance produces exactly the same tree as when run on the original instance (with the given tie-breaking rule); and (b) the unique MST in the perturbed MST is an MST for the original instance. Since we have argued that Prim and Kruskal return the MST when edge costs are distinct, this shows that Prim and Kruskal implemented with any tie-breaking rule produces an MST. Let Δ be the smallest absolute difference between the costs of two trees with different costs, that is, $\Delta = \min_{T, T': c(T) \neq c(T')} |c(T) - c(T')| > 0$. Let $\epsilon = \frac{\Delta}{nm}$. Let T^* be the tree produced by Prim or Kruskal under a given tie-breaking rule. Consider the ordering of the edges by increasing c_e values where ties are broken in favor of edges in T^* . That is, if $c_e = c_{e'}$ and exactly one of e, e' is in T^* , then the edge in T^* comes earlier, otherwise their relative ordering is arbitrary. If e is the i -th edge in this ordering, then we set $\bar{c}_e = c_e + i \cdot \epsilon$. We then have that for any two edges e and e' : (i) if e comes before e' in the ordering then clearly $\bar{c}_e < \bar{c}_{e'}$; (ii) $\bar{c}_e \neq \bar{c}_{e'}$ by (i), since one of e and e' must come before the other in the ordering. From (i) it follows that Prim and Kruskal return the same tree T^* when run on the instance with costs \bar{c} , and from (ii) it follows that T^* is the unique MST in the perturbed instance. Finally, for *any* tree T , we have that $c(T) \leq \bar{c}(T) \leq c(T) + (n-1)m \cdot \epsilon < c(T) + \Delta$ since each edge of T is perturbed by at most $m\epsilon$. Therefore, since $\bar{c}(T^*) < \bar{c}(T)$ for any tree T , we have $c(T^*) < c(T) + \Delta$ implying that $c(T^*) \leq c(T)$.

(by the definition of Δ), and hence, that T^* is an MST for the original instance.

We remark that it may be difficult to compute Δ , but note that the perturbation idea is used only in the *analysis* to prove that Prim and Kruskal produce an MST under any tie-breaking rule.

Q4: We will prove that the MST wrt. edge costs a_e is a tree satisfying all the properties required in part (b). Thus, we can use any MST algorithm, e.g., Prim, Kruskal, Boruvka, to compute it, and this also gives an algorithm for part (a). A key observation is that the outcome of these algorithms (as should be clear from their description) depends *only on the ordering of the edges* (under a tie-breaking rule) and *not* on the actual edge costs. (This is *not true*, for example, of Dijkstra's algorithm for shortest paths — do you see why?) More generally, the (collection of) MST(s) of a graph is completely determined by the relative ordering of the edges. This implies the interesting property that if there are two cost functions c and c' , and there is an ordering of the edges that represents a valid sorted ordering for both c and c' (by increasing values), then there is a tree T that is *simultaneously* an MST for both the c_e costs and the c'_e edge costs; the tree T can be obtained, for example, by running Prim or Kruskal using this sorted order.

Let us call an edge e , i -difficult if $a_e \geq \alpha_i$. Consider the cost function c^i , $i = 1, \dots, k$, where $c_e^i = 1$ if $a_e \geq \alpha_i$, and 0 otherwise. A tree that minimizes the number of i -difficult edges is simply an MST for the the c_i cost function. The first requirement in part (b) is that the desired tree should simultaneously be an MST for each cost function c_i , $i = 1, \dots, k$. Notice that ordering the edges by increasing a_e values is, for every $i = 1, \dots, k$, a valid ordering of the edges by increasing c_e^i values. Thus, by the observation above, an MST wrt. the a_e edge costs is also an MST wrt. the c_e^i edge costs for every $i = 1, \dots, k$.

We now show that the MST T^* wrt. the a_e edge costs also satisfies the second requirement. By the cycle property, if $e \in T^*$ then e is not the max-cost edge in any cycle. Thus for any u, v , if e is the max-altitude edge on the path P_{uv} between u and v in T^* , then $a_e < \max_{e' \in P} a_{e'}$ for any other u - v path P , otherwise e would be the max-cost edge in the (not necessarily simple) cycle $P_{uv} \cup P$.

It is worth noting that there are in fact, many other cost functions c one could consider such that an MST wrt. edge costs c_e would satisfy the first requirement, i.e., it would simultaneously yield an MST for all the cost functions c^i . Any edge costs of the form $f(a_e)$, where f is a monotonic nondecreasing function with $f(\alpha_i) > f(\alpha_i - \epsilon)$ for every $i = 1, \dots, k$, every $\epsilon > 0$ would work. In particular, we can take $c_e = \sum_{i=1}^k c_e^i$, i.e., $c_e = i$ if $a_e \in [\alpha_i, \alpha_{i+1})$. This has the benefit that in Prim's algorithm, the ExtractMin operation can be implemented in $O(k)$ time (DecreaseKey still takes $O(1)$ time) by keeping a different list for each i consisting of all edges with $c_e = i$. Thus, the running time becomes $O(m + nk)$. For part (a) this yields an $O(m)$ time algorithm. The costs c_e also speed up Kruskal's algorithm since sorting the c_e -values can be done in $O(mk)$ time, which is better than $O(m \log m)$ if $k \leq \log m$.