CSCI2100: Regular Exercise Set 13

Prepared by Yufei Tao

Problem 1 (Correctness of Dijkstra) Prove that Dijkstra's algorithm correctly computes all the shortest paths from the source vertex.

Solution. Let s be the source vertex. Recall that the algorithm works by repetitively removing the vertex u from S that has the smallest $dist(u)$. We will prove that, when u is removed, $dist(u)$ equals precisely the shortest path distance—denoted as $spdist(u)$—from s to u.

We will prove the claim by induction on the sequence of vertices removed. This is obviously true for the first vertex removed, which is s itself with $dist(s) = 0$.

Now consider that we are removing vertex u from S, and the claim is true with respect to all the vertices already removed. Consider any shortest path π from s to u. Let v be the predecessor of u on this path. We will prove that v has already been removed. This will complete the proof because when v is removed, we have:

- $spdist(v) = dist(v)$
- Relaxing the edge (v, u) makes $dist(u) = dist(v) + w(u, v) = spdist(v)$.

We will prove that all the vertices on π have been removed (and hence, v as well) at the moment when u is removed. Suppose that this is not true. Let v' be the first vertex (in the direction from s to u) on π that still remains in S. Let p be the predecessor of v' on π. By the inductive assumption, we know that $dist(p) = spdist(p)$ when p was removed. Hence, after relaxing the edge (p, v'), we had $dist(v') = dist(p) + w(p, v') = spdist(v') < dist(u)$. This means that v' should be the next vertex to remove, contradicting that the algorithm has chosen u.

Problem 2. Let S be a set of integer pairs of the form (id, v). We will refer to the first field as the *id* of the pair, and the second as the *key* of the pair. Design a data structure that supports the following operations:

- Insert: add a new pair (id, v) to S (you can assume that S does not already have a pair with the same id).
- Delete: given an integer t, delete the pair (id, v) from S where $t = id$, if such a pair exists.
- DeleteMin: remove from S the pair with the smallest key, and return it. .

Your structure must consume $O(n)$ space, and support all operations in $O(\log n)$ time where $n = |S|$.

Solution. Maintain S in two binary search trees T_1 and T_2, where the pairs are indexed on ids in T_1, and on keys in T_2. We support the three operations as follows:

- Insert: simply insert the new pair (id, v) into both T_1 and T_2.
- Delete: first find the pair with id t in T_1, from which we know the key v of the pair. Now, delete the pair (t, v) from both T_1 and T_2.
- DeleteMin: find the pair with the smallest key v from T_2 (which can be found by continuously descending into left child nodes). Now we have its id t as well. Remove (t, v) from T_1 and T_2.

Problem 3. Describe how to implement the Dijkstra's algorithm on a graph $G = (V, E)$ in $O((|V| + |E|) \cdot \log |V|)$ time.

Solution. Recall that the algorithm maintains (i) a set S of vertices at all times, and (ii) an integer value $dist(v)$ for each vertex $v \in S$. Define P to be the set of $(v, dist(v))$ pairs (one for each $v \in S$). We need the following operations on P:

- Insert: add a pair $(v, dist(v))$ to P.
- DecreaseKey: given a vertex $v \in S$ and an integer $x < dist(v)$, update the pair $(v, dist(v))$ to (v, x) (and thereby, setting $dist(v) = x$ in P).
- DeleteMin: Remove from P the pair $(v, dist(v))$ with the smalelst $dist(v)$.

We can store P in a data structure of Problem 2 which supports all operations in $O(\log |V|)$ time (note: DecreaseKey can be implemented as a Delete followed by an Insert).

In addition to the above structure, we store all the $dist(v)$ values in an array A of length $|V|$, so that using the id of a vertex v, we can find its $dist(v)$ in constant time.

Now we can implement the algorithm as follows. Initially, insert only $(s, 0)$ into P, where s is the source vertex. Also, in A, set all the values to ∞, except the cell of s which equals 0.

Then, we repeat the following until P is empty:

- Perform a DeleteMin to obtain a pair $(v, dist(v))$.
- For every edge (v, u), compare $dist(u)$ to $dist(v) + w(u, v)$. If the latter is smaller, perform a DecreaseKey on vertex u to set $dist(u) = dist(v) + w(u, v)$, and update the cell of u in A with this value as well.

Problem 4. Prove: in a weighted undirected graph $G = (V, E)$ where all the edges have distinct weights, the minimum spanning tree (MST) is unique.

Solution. We will prove that the tree T returned by the Prim's algorithm is the only MST. Set $n = |V|$. Let $e_1, e_2, ..., e_{n-1}$ be the sequence of edges that the algorithm adds to T. Suppose, on the contrary, that there is another MST T'. Let k be the smallest i such that e_i is not in T'.

- Case 1: $k = 1$. This means that e_1, which is the edge with the smallest weight, is not in T'. Add e_1 to T' to create a cycle, and remove from the cycle the edge with the largest weight. This create another spanning tree whose cost is strictly smaller than T' (remember: all the edges are distinct), contradicting the fact that T' is an MST.
- Case 2: $k > 1$. Recall that edges $e_1, e_2, ..., e_{k-1}$ form a tree. Let S be the set of vertices in this tree. Add $e_k = \{u, v\}$ into T' to create a cycle. Suppose $u \in S$; it follows that $v \notin S$. Let us walk on the cycle from v, by going into S, traveling within S, and stopping as soon as we exist S. Let $\{u', v\}$ be the last edge crossed (namely, one of u', v' is in S, while the other one is not). By the way Prim's algorithm runs and the fact that all edges have distinct weights, we know that $\{u, v\}$ has a smaller weight than $\{u', v'\}$. Thus, removing $\{u', v'\}$ from T' gives spanning tree with strictly smaller cost, which creates a contradiction.

Problem 5. Describe how to implement the Prim's algorithm on a graph $G = (V, E)$ in $O((|V| + |E|) \cdot \log |V|)$ time.

Solution. Remember that the algorithm incrementally grows a tree T which at the end becomes the final minimum spanning tree. Let S be the set of vertices that are currently in T. At all times, the algorithm maintains, for every vertex $v \in V \setminus S$, its lightest extension edge *best-ext*(v), and the weight of this edge.

To implement this, we maintain a set P of triples, one for every vertex $u \in V \setminus S$. Specifically, the triple of u has the form (u, v, t), indicating that *best-ext*(u) is the edge $\{u, v\}$ (i.e., $v \in S$), whose weight is t. We need the following operations on P:

- Insert: add a triple (u, v, t) to P.
- DecreaseKey: given a vertex $v' \in S$ and an extension edge $\{u, v'\}$ (i.e., $u \notin S$), this operation does the following. First, fetch the triple (u, v, t). Then, compare t to the weight t' of $\{u, v'\}$. If $t' < t$, update the triple (u, v, t) to (u, v', t'); otherwise, do nothing.
- DeleteMin: Remove from P the triple (u, v, t) with the smallest t.

We can store P in a data structure of Problem 2 which supports all operations in $O(\log |V|)$ time (note: DecreaseKey can be implemented as a Delete followed by an Insert). Besides the above structure, we also store an array A of length $|V|$ to so that we can query in constant time, for any vertex $v \in V$, whether v is in S currently.

Now we can implement the algorithm as follows. Let $\{v_1, v_2\}$ be an edge with the smallest weight in G. The set S contains only v_1 and v_2 at this point. For every vertex $u \in V \setminus S$ where $S = \{v_1, v_2\}$, we check whether u has extension edges to v_1 and v_2. If neither edge exists, insert triple (u, nil, ∞) to P. Otherwise, suppose without loss of generality that $\{u, v_1\}$ is the lighter extension edge of u with weight t; insert a triple (u, v_1, t) into P.

Repeat the following until P is empty:

- Perform a DeleteMin to obtain a triple (u, v, t).
- Recall that u should be added to S, which may need to change the extension edges of some other vertices. To implement this, for every edge (u, u') of u where $u' \notin S$, perform DecreaseKey with u' and $\{u, u'\}$.