CSCI2100: Regular Exercise Set 13

Prepared by Yufei Tao

Problem 1. 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 2. 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:

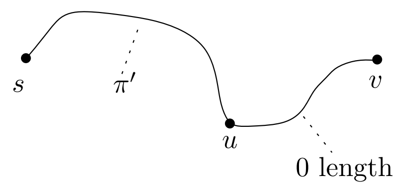
- Perform a DeleteMin to obtain a pair $(v, dist(v))$.

- 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 3*. In the lecture, we proved the correctness of Dijkstra's algorithm in the scenario where all the edges have positive weights. Prove: the algorithm is still correct if we allow edges to take *non-negative* weights (i.e., zero weights are allowed).

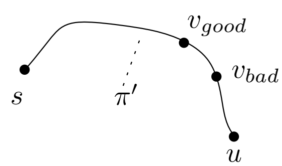
Solution. We argue that, *every time a vertex v is removed from S, we must have $dist(v) = spdist(v)$.* We will do so by induction on the order that the vertices are removed. The base step, which corresponds to removing the source vertex s, is obviously correct. Next, assuming correctness on all the vertices already removed, we will prove the statement on the vertex v removed *next*.

Let π be an arbitrary shortest path from s to v. Identify the last vertex u on π such that $spdist(u) = spdist(v)$. In other words, all the edges on π between u and v have weight 0. Let π' be the prefix of π that ends at u. Note that π' must be a shortest path from s to u.



Claim 1: When v is to be removed from S, all the vertices on π' — except possibly u — must have been removed from S.

Proof of Claim 1: Suppose that the claim is not true. Define v_{bad} as the first vertex on π' that is still in S when v is to be removed from S. Let v_{good} be the vertex right before v_{bad} on π; note that v_{good} definitely exists because v_{bad} cannot be s. By how u is defined, we must have $spdist(v_{bad}) < spdist(u) = spdist(v)$.

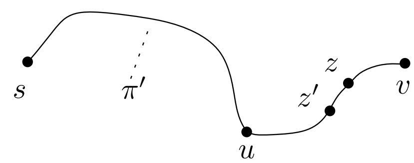


By our inductive assumption, when v_{good} was removed from S, we had $dist(v_{good}) = spdist(v_{good})$. We must have relaxed the edge (v_{good}, v_{bad}), after which we must have

$$\begin{aligned} dist(v_{bad}) &= dist(v_{good}) + w(v_{good}, v_{bad}) \\ &= spdist(v_{good}) + w(v_{good}, v_{bad}) = spdist(v_{bad}). \end{aligned}$$

The value $dist(v_{bad})$ never increases during the algorithm. Hence, when v is to be removed from S, we must have $dist(v_{bad}) = spdist(v_{bad}) < spdist(u) = spdist(v) \leq dist(v)$. But this contradicts the fact that v has the smallest *dist*-value among all the vertices still in S. $\square$

Consider the moment when v is to be removed from S; define z as the first vertex on π that has *not* been removed from S. Note that z is well defined because v itself is still in S at this moment.



Claim 2: When v is to be removed from S, $dist(z) = spdist(z)$.

Proof of Claim 2: Let z' be the vertex right before z on π. Note that z' is well defined because z cannot be earlier than u on π (Claim 1) and z cannot be s.

By our inductive assumption, when z' was removed from S, we had $dist(z') = spdist(z')$. We must have relaxed the edge (z', z), after which we must have

$$dist(z) = dist(z') + w(z', z) = spdist(z') = spdist(z).$$

$\square$

It now follows that, when v is to be removed from S, we have $dist(v) \leq dist(z) = spdist(z) = spdist(v)$. As $dist(v)$ cannot be larger than $spdist(v)$, we must have $dist(v) = spdist(v)$.