CSCI3160: Regular Exercise Set 8

Prepared by Yufei Tao

Problem 1. Let P be a set of n integer pairs, each of which has the form (id, key). It is guaranteed that no two pairs have the same id (but there may be pairs having the same key). Describe a structure of $O(n)$ space to support each of the following operations in $O(\log n)$ time:

- Insert(i, k): add a pair (i, k) to P if P does not already have a pair with id i;
- DecreaseKey(i, k): if P does not have any pair with id i, this operation has no effects. Otherwise, suppose that the pair is (i, k'); the operation replaces the key k' of the pair with k if $k < k'$;
- DeleteMin: Remove from P the pair with the smalelst key.

Solution. Build two binary search trees (BST). The first one T_1 is created on the id fields of the pairs in P, while the second one T_2 is created on their key fields. The space consumption is clearly $O(n)$. The three operations can be supported as follows:

- Insert(i, k): first check whether the id i already exists in P; this can be done in $O(\log n)$ time using T_1. If not, then insert the pair (i, k) to both T_1 and T_2 in $O(\log n)$ time.
- DecreaseKey(i, k): first check whether the id i already exists. If not, the operation finishes with no more actions. Otherwise, fetch from T_1 the pair (i, k') in P, which takes $O(\log n)$ time. If $k < k'$, remove (i, k') from both T_1 and T_2, and then insert the pair (i, k) into both trees; these deletions and insertions take $O(\log n)$ time in total.
- DeleteMin: Use T_2 to find the pair — say (i, k) — with the smallest key in $O(\log n)$ time. Then, delete the pair from both trees in $O(\log n)$ time.

Problem 2. Describe how to implement 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 1 which supports all operations in $O(\log |V|)$ time.

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 outgoing edge (v, u) of v, 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. Point out the place in the proof that requires the assumption that all the weights are non-negative.

Solution. The proof holds only if $dist(v_{bad}) < dist(u)$ (check the proof in the lecture notes for the meanings of v_{bad} and $dist(u)$). This no longer holds if edges can take negative weights.

Problem 4 (SSSP with Unit Weights). Let us simplify the SSSP problem by requiring that all the edges in the input directed graph $G = (V, E)$ take the *same* weight, which we assume to be 1. Give an algorithm that solves the SSSP problem in $O(|V| + |E|)$ time.

(Remark: you can of course still use Dijkstra's algorithm, but as shown earlier, its complexity is $O((|V| + |E|) \log |V|)$. You mission here is to improve the time complexity to $O(|V| + |E|)$)

Solution. Let s be the source vertex. We perform a breadth first search (BFS) to obtain the so-called *BFS-tree* as follows:

algorithm BFS
1. set $parent(v) = \emptyset$ for every $v \in V$
2. color all vertices in V white
3. initialize an empty queue Q
4. insert s into Q, set $parent(s) = nil$, and color s white
5. **while** Q is not empty
6. remove the first vertex u in Q
7. **for** every outgoing edge (u, v) of u
8. **if** v is black **then**
9. set $parent(v) = u$, insert v into Q, and color v black

T is the tree that is formed by the *parent* function (i.e., the parent node of v in T is $parent(v)$). Note that s is the root of T. The shortest path from s to any vertex v is the only path from s to v in T.

Problem 5*. 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. As in the proof in our lecture notes, we will prove that $dist(v)$ must be $spdist(v)$ when v is to be removed from S. Again 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 that the statement holds on the *next* vertex v to be removed.

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 (i.e., π' is a sequence of edges that is the same as π, except that π' does not grow beyond 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.

This claim can be established using the same argument as in our lecture notes (consider the predecessor of u, which must have been removed, and then discuss what happens when the algorithm relaxed the edge from that predecessor to u).

Now let us focus on the path π'' that is the sequence of edges from u to v on π. Define u' as the first vertex on π'' that has *not* been removed from S. Note that u' is well defined because v itself (which is the last vertex on π'') is still in S at this moment.

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

This claim again can be established using the same argument as in our lecture notes.

It now follows that $dist(v) \leq dist(u') = spdist(u') = spdist(v)$, where the first inequality used the fact that the algorithm is about to remove v from S.