COMS21103: Problem set 3

2015/2016

Remark: Most problem sets for the next few weeks will contain at least one starred problem, which is more challenging. If any of the problems seem unclear, please post a question on the Blackboard discussion board.

1. You are given a weighted graph G with integer edge weights between 1 and 5. Argue that you can solve the (single source) shortest paths problem on G using Breadth First Search in $O(|V| + |E|)$ time.

   Answer (sketch): Replace every edge (u, v) with weight w with a chain of w unweighted (equivalently weight one) edges. More specifically in the case that $w = 3$ we replace (u, v) with two new vertices x_1 and x_2 and three edges (u, x_1), (x_1, x_2) and (x_2, v). Let $G' = (V', E')$ be the transformed graph (after replacing each edge with a chain). Let $|E'|$ be the number of edges in G' and $|V'|$ be the number of vertices. Each edge in G becomes at most 5 edges in G' and introduces at most 4 new vertices. We have that $|E'| \leq 5|E|$ and $|V'| \leq |V| + 4|E|$. The key observation is that the length of the shortest paths between any two vertices in the original graph G, is the same as in the transformed graph G'. We then run BFS on G' (using the desired source). BFS solves the (single source) shortest paths problem on the graph G' in $O(|V'| + |E'|)$ time. However we have that $O(|V'| + |E'|) = O(|V| + |E|)$. Also observe that we can convert G into G' in $O(|V'| + |E'|) = O(|V| + |E|)$ time.

2. Consider the following algorithm takes as input an unweighed, undirected, *possibly unconnected* graph G. What does it do? What is its time complexity?

   ```
   MYSTERY(G) :
   All vertices are initially unmarked
   x = 0
   For each vertex v,
       If v is unmarked
           TRAVERSE(v) (any vertices marked by TRAVERSE remain marked)
           x = x + 1
   Return x
   ```

   Answer (sketch): This is essentially the TRAVERSEALL operation from JE chapter 18 (the reading for the BFS/DFS lecture). It counts the number of components in the graph (notice that we did not assume that the input was connected). It runs in $O(|V| + |E|)$ time. The very first call to TRAVERSE(v) marks all the vertices connected to v and visits each edge in the connected component. As any vertex connected to v is now marked, those edges/vertices will never be seen by any call to TRAVERSE(v) again. By repeating this argument it follows that TRAVERSE is called once per connected component. The stated time complexity then follows.

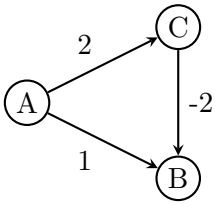
3. Prove the claim made in lecture that if a Binary heap is stored in an array, the operations Parent, Left and Right work correctly as defined.

   Answer (sketch): We first prove the claim that Left(i) $= 2i$. Consider the j-th node in the k-th level of the tree (where the root is at level 0, node 0). This is the $((2^k - 1) + j)$-th element in the

array (in particular the root is the 0-th element in the array). So for this node $i = ((2^k - 1) + j)$. Its left child is the $(2j)$-th node in level $k+1$ of the tree. This is element $(2^{k+1} - 1) + (2j)$ in the array. As required, $(2^{k+1} - 1) + (2j) = 2((2^k - 1) + j) = 2i$. As the node in the tree was arbitrary, this completes the proof. The proofs for Right and Parent follow almost immediately from the claim that $\text{Left}(i) = 2i$.

4. Give an example of a graph G with at least one negative-weight edge such that Dijkstra's algorithm does not correctly output a shortest path in G. Explicitly identify where the property of having only non-negative weight edges was used in the proof of correctness of Dijkstra's algorithm.

   Answer (sketch): Consider the following example.



   The point in the proof which uses this condition is the claim that for vertices y, v on a shortest path where y appears before v, $\delta(s, y) \leq \delta(s, v)$. Check that this is not true in this example.

5. Imagine we want to "fix" a graph with negative-weight edges to make Dijkstra's algorithm work, and do this by adding some large constant to the weight of each edge to make all the weights positive. Give an example which shows that this approach does not work.

   Answer (sketch): Try the example from the last part. In particular observe that in the graph above the shortest path from A to B goes via C. However if we add a 100 to each weight, the shortest path from A to B is now not via C. The path ABC has weight 200 while the path AB has weight 101. The problem with adding a constant to each edge weight is that paths which contain more edges have more weight added to them than paths with fewer edges.

6. Imagine that we want to modify our Binary Heap to support a new IncreaseKey operation (as well as Insert, DecreaseKey and ExtractMin). The IncreaseKey(x, k) operation is defined in the natural way to increase the key of element x so that $x.key = k$ (assuming $k > x.key$). How would you modify your Binary Heap to support IncreaseKey in $O(\log n)$ time?

   Answer (sketch): The 'cheat' answer is that you can add support by using the existing operations to perform IncreaseKey(x, k) as follows: First perform DecreaseKey$(x, -\infty)$ and then perform ExtractMin. You have now removed the item x. Now perform Insert(x, k) to put the item back in with the reduced key. This runs in $O(\log n)$ time because it performs a constant number of operations on the queue, each in $O(\log n)$ time. Unfortunately, in practice this is a poor approach because the new operation we have added has much larger constants than the other operations (it does one of each). A more sensible direct approach follows from considering the behaviour of the DeleteMin operation after it deletes the minimum. This is in-effect an IncreaseKey operation.

7. ($\star$) Imagine we would like to implement Dijkstra's algorithm using a restricted priority queue which does not support the DecreaseKey operation (so it only supports Insert and ExtractMin). Modify Dijkstra's algorithm to work using such a queue. What is the time complexity of your modified algorithm?

8. ($\star$) Design a data structure which supports the operations Insert(x) and ExtractRandom(). The Insert(x) operation inserts an element x. The ExtractRandom operation should extract a (uniformly chosen) random item from the data structure. This can be seen as a random implementation of the 'bag' abstract data structure seen in lectures. Both operations should run in worst-case $O(1)$ time. You are allowed to assume that you have of use of an O(1) time random number generator.