CSCI 270 - Spring 2023 - HW 4

Due: February 8, 2023

Graded Problems

1. [10 points] Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):

   - Find median takes $\mathcal{O}(1)$ time
   - Insert takes $\mathcal{O}(logn)$ time

   Do the following:

   (a) Describe how your data structure will work.

   (b) Give algorithms that implement the Find-Median() and Insert() functions.

   Solution We use the $\lceil n/2 \rceil$ smallest elements to build a max-heap and use the remaining $\lfloor n/2 \rfloor$ elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time O(1) (we assume the case of even n, median is n/2-th element when elements are sorted in increasing order).

   Insert() algorithm: For a new element x

   - Initialize len of maxheap (maxlen) and len of minheap (minlen) to 0.
   - Compare x to the current root of max-heap.
   - If x $<$ median, we insert x into the max-heap.Maintain length of maxheap say maxlen, and every time you insert element into maxheap increase maxlen by 1. Otherwise, we insert x into the min-heap, and increase length of minheap say minlen by 1. This takes O(log n) time in the worst case.
   - If size(maxHeap) $>$ size(minHeap)+1, then we call Extract-Max() on max-heap, and decrease maxlen by 1 of and insert the extracted value into the min-heap and increase minlen by 1. This takes O(log n) time in the worst case.

- Also, if size(minHeap)>size(maxHeap), we call Extract-Min() on min-heap, decrease minlen by 1 and insert the extracted value into the max-heap and increase maxlen by 1. This takes O(log n) time in the worst case.

Find-Median() algorithm:

- If (maxlen+minlen) is even: return (sum of roots of max heap and min heap)/2 as median
- Else if maxlen>minlen: return root of max heap as median
- return root of min heap as median

Rubric

- 2 pt: Using max heap and min heap to store first half and second half of elements.
- 3 pt: Comparing element to the root and proper if conditions for inserting in appropriate heap
- 2 pt: Proper if conditions for returning the median.
- 3 pt: Correct Time Complexity

2. [10 points] Let us say that a graph $G = (V, E)$ is a near tree if it is connected and has most $n + k$ edges, where $n = |V|$ and k is a **constant**. Give an algorithm with running time $\mathcal{O}(n)$ that takes a near tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all edge costs are distinct.

Solution

1. To do this, we apply the Cycle Property $k + 1$ times. That is, we perform BFS until we find a cycle in the graph G, and then we delete the heaviest edge on this cycle.

2. We have now reduced the number of edges in G by one while keeping G connected and (by the Cycle Property) not changing the identity of the minimum spanning tree.

3. If we do this a total of $k + 1$ times, we will have a connected graph H with $n - 1$ edges and the same minimum spanning tree as G. But H is a tree, and so in fact it is the minimum spanning tree.

4. The running time of each iteration is $\mathcal{O}(m + n)$ for the BFS and subsequent check of the cycle to find the heaviest edge; here $m \leq n + k$, so this is $\mathcal{O}(n)$.

Rubric

- 3 pt: Applying cycle property $k + 1$ times.
- 2 pt: Figuring out finding LCA to find heaviest edge

- 2 pt: Figuring out that we will have to delete the heaviest edge on this cycle
- 3 pt: Correct Time Complexity

3. [14 points] A new startup FastRoute wants to route information along a path in a communication network, represented as a graph. Each vertex and each edge represent a router and a wire between routers respectively. The wires are weighted by the maximum bandwidth they can support. FastRoute comes to you and asks you to develop an algorithm to find the path with maximum bandwidth from any source s to any destination t. As you would expect, the bandwidth of a path is the minimum of the bandwidths of the edges on that path; the minimum edge is the bottleneck. Explain how to modify Dijkstra's algorithm to do this.

   Solution

   We modify the Dijkstra's Algorithm described in p138 in the following way (Diffs are highlighted.). Notice that ℓ_e is a bandwidth of an edge here.

```
Modified Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
  For each $u \in S$, we store the maximum bandwidth $d(u)$
Initially $S = \{s\}$ and $d(s) = \infty$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
    $d'(v) = \max_{e=(u,v):u \in S} \min(d(u), \ell_e)$ is as large as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
```

   (The proof is not required in this problem but you should master it.) The following is the proof but large part of the proof comes from the textbook p139. (Diffs are highlighted.)

   Proof: We prove this by induction on the size of S. The case $|S| = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = \infty$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; we now grow S to size $k+1$ by adding the node v. Let (u, v) be the final edge on our $s - v$ path P_v.

   By induction hypothesis, P_u is the $s - u$ path with maximum bandwidth for each $u \in S$. Now consider any other $s - v$ path P; we wish to show that cannot have larger bandwidth than P_v. In order to reach v, this path P must leave the set S somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

P cannot have larger-bandwidth than P_v because it has already lesser or equal bandwidth than P_v by the time it has left the set S. Indeed, in iteration $k+1$, Dijkstra's Algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favor of adding v. This means that there is no path from s to y through x that has larger-bandwidth than P_v. But the subpath of P up to y is such a path, and so this subpath has at least as small-bandwidth as P_v. Since $\min(d(u), \ell_e)$ cannot make the total bandwidth larger, the full path P has at least as small-bandwidth as P_v as well. This is complete proof.

Runtime is the same as original Dijkstra's algorithm.

Rubric

- 4 points for the part of $d(s) = \infty$
- 10 points for the part of $d'(v) = \max_{e=(u,v):u\in S} \min(d(u), \ell_e)$ is as large as possible

4. [20 points] Given a connected graph $G = (V, E)$ with positive edge weights. In V, s and t are two nodes for shortest path computation, prove or disprove with explanations:

   (a) If all edge weights are unique, then there is a single shortest path between any two nodes in V .

   (b) If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k.

   (c) If the weight of some edge e decreases by k, then the shortest path cost between s and t will decrease by at most k.

   (d) If each edge's weight is replaced by its square, i.e., w to w^2, then the shortest path between s and t will be the same as before but with different costs.

Solution

(a) False. Counter example: (s, a) with weight 1, (a, t) with weight 2 and (s, t) with weight 3. There are two shortest path from s to t though the edge weights are unique.

Rubric

- 2 pt: Correct T/F claim
- 3 pt: Provides a correct counterexample as explanation

(b) False. Counter example: suppose the shortest path s $\to$ t consist of two edges, each with cost 1, and there is also an edge e = (s, t) in G with cost(e)=3. If now we increase the cost of each edge by 2, e will become the shortest path (with the total cost of 5).

Rubric

- 2 pt: Correct T/F claim
- 3 pt: Provides a correct counterexample as explanation

(c) False (or True if you do not allow repeated vertices in a path).

Note: Since a path is not allowed to repeated vertices in the definitions of some textbooks, we accept solutions that answer True with the explanation in the first bullet.

- Only true when we have the assumption that after decreasing all edge weights are still positive (however we don't have this in the problem). For any two nodes s, t, assume that $P_1, ..., P_k$ are all the paths from s to t. If e belongs to P_i then the path cost decrease by k, otherwise the path cost unchanged. Hence all paths from s to t will decrease by at most k. As shortest path is among them, then the shortest path cost will decrease by at most k.
- When 1) there is cycle in the graph, 2) and there is a path from s to t that goes through that cycle, 3) and after decreasing, the sum of edge weights of that cycle becomes negative, then the shortest path from s to t will go over the cycle for infinite times, ending up with infinite path cost, hence not "decrease by at most k".

Rubric

- 2 pt: Correct T/F claim
- 3 pt: Correct explanation

(d) False. Counter example: 1) suppose the original graph G composed of V = A, B, C, D and E : (A $\rightarrow$ B) = 100, (A $\rightarrow$ C) = 51, (B $\rightarrow$ D) = 1, (C $\rightarrow$ D) = 51, then the shortest path from A to D is A $\rightarrow$ B $\rightarrow$ D with length 101. 2) After squaring this path length become $100^2 + 1^2 = 10001$. However, A $\rightarrow$ C $\rightarrow$ D has path length $51^2 + 51^2 = 5202 < 10001$ Thus A $\rightarrow$ C $\rightarrow$ D become the new shortest path from A to D.

Rubric

- 2 pt: Correct T/F claim
- 3 pt: Provides a correct counterexample as explanation

5. [16 points] Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which allows you to change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Propose an efficient method based on Dijkstra's algorithm to find a lowest-cost path from node s to node t, given that you may set one edge weight to zero.

   Solution Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by u $\rightsquigarrow$ v, and its length by δ (u, v). Now, try setting each edge to zero. For each edge (u, v) $\in$ E, consider the path $s \to u \rightsquigarrow v \to t$. If we set w(u, v) to zero, the path length is δ(s, u) + δ(v, t). Find the edge for which this length is minimized and set it to zero; the corresponding path $s \to u \rightsquigarrow v \to t$ is the desired path.

   Time complexity: The algorithm requires two invocations of Dijkstra, and an additional $O(|E|)$ time to iterate through the edges and find the optimal edge to take for free, which is asymptotically less than Dijkstra. Thus the total running time is the same as that of Dijkstra. (For time complexity analysis, any implementation for Dijkstra's, such as with Binary heap or with Fibonacci heap works just fine.)

   Rubric

   For the algorithm:

   - 12 pt if the approach is of the same complexity as Dijkstra's algorithm
   - 4 pt if the approach is more efficient than naive but not has the same complexity as Dijkstra's algorithm
   - 0 pt if the naive approach or any other approach with larger complexity than naive is proposed.

   4 pt for time complexity analysis.