CSCI 570 - Fall 2021 - HW 4

Due: Sep 23rd

Section 1: Heaps

Reading Assignment: Kleinberg and Tardos, Chapter 2.5.

Problem 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 $\mathcal{O}(\log n)$ time

Do the following:

1. Describe how your data structure will work.
2. Give algorithms that implement the Find-Median() and Insert() functions.

(**Hint:** Read this only if you really need to. Your Data Structure should use a min-heap and a max-heap simultaneously where half of the elements are in the max-heap and the other half are in the min-heap.)

Solution. We use the dn/2e smallest elements to build a max-heap and use the remaining bn/2c elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time $\mathcal{O}(1)$ (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 $\mathcal{O}(\log n)$ 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 $\mathcal{O}(\log n)$ 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 $\mathcal{O}(\log n)$ 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
- Else: return root of min heap as median

■

Problem 2

[10 points] There is a stream of integers that comes continuously to a small server. The job of the server is to keep track of k largest numbers that it has seen so far. The server has the following restrictions:

- It can process only one number from the stream at a time, which means it takes a number from the stream, processes it, finishes with that number and takes the next number from the stream. It cannot take more than one number from the stream at a time due to memory restriction.
- It has enough memory to store up to k integers in a simple data structure (e.g. an array), and some extra memory for computation (like comparison, etc.).
- The time complexity for processing one number must be better than $\mathcal{O}(k)$. Anything that is $\mathcal{O}(k)$ or worse is not acceptable.

Design an algorithm on the server to perform its job with the requirements listed above.

Solution. Use a binary min-heap on the server.

1. Do not wait until k numbers have arrived at the server to build the heap, otherwise you would incur a time complexity of $\mathcal{O}(k)$. Instead, build the heap on-the-fly, i.e. as soon as a number arrives, if the heap is not full, insert the number into the heap and execute Heapify(). The first k numbers are obviously the k largest numbers that the server has seen.

2. When a new number x arrives and the heap is full, compare x to the minimum number r in the heap located at the root, which can be done in $\mathcal{O}(1)$ time. If $x \leq r$, ignore x. Otherwise, run Extract-min() and insert the new number x into the heap and call Heapify() to maintain the structure of the heap.

3. Both Extract-min() and Heapify() can be done in $\mathcal{O}(\log k)$ time. Hence, the overall complexity is $\mathcal{O}(\log k)$.

■

Section 2: MST

Reading Assignment: Kleinberg and Tardos, Chapter 4.5.

Problem 3

[**10 points**] Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution. Proof by contradiction:

1. If T_1, T_2 are Distinct, there must be at least one edge that's in T_1 but not in T_2 (and vice versa) since one MST cannot contain the other. let e_1 be the minimum weight edge among all such edges that are in exactly one but not both.

2. $T_2 + e_1$ contains a cycle, say C. C must contain e_1 since C wasn't present in T_2. Further, the edges in C other than e_1 cannot all be in T_1 since otherwise, T_1 would contain the cycle C. Hence, there's an edge in C, say e_2, which is in T_2, but not in T_1. However, since e_1 has the minimum weight among edges that are in exactly one, but not both, $w(e_1) < w(e_2)$. The strict inequality is because we have all edge weights distinct.

3. Note that $T = T_2 \cup \{e_1\} \backslash \{e_2\}$ is a spanning tree. The total weight of T is smaller than the total weight of T_2, but this is a contradiction, since we have supposed that T_2 is a minimum spanning tree.

■

Problem 4

[**10 points**] Let us say that a graph $G = (V, E)$ is a near tree if it is connected and has at most $n+8$ edges, where $n = |V|$. 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 nine 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 nine 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 $m \leq n + 8$, so this is $\mathcal{O}(n)$.

■

Section 3: Shortest Path

Reading Assignment: Kleinberg and Tardos, Chapter 4.4.

Problem 5

[**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:

1. If all edge weights are unique, then there is a single shortest path between any two nodes in V.
2. If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k.
3. 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.
4. 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.

1. 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.
2. 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).

3. False.

   - 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 $P_1, \ldots, P_k$ 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".

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

■

Problem 6

[**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.

Note: you will receive 10 points if your algorithm is efficient. This means your method must do better than the naive solution, where the weight of each node is set to 0 per time and the Dijkstra's algorithm is applied every time for lowest-cost path searching. You will receive full points (16 points) if your algorithm has the same run time complexity as Dijkstra's algorithm.

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 $\delta(u, v)$. Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \to v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \to v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\mathcal{O}(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

- If we implement Dijkstra's algorithm with Fibonacci heap
  - time complexity of both the Dijkstra's algorithm and the proposed method is $\mathcal{O}(E + V \log V)$.
  - the corresponding time complexity of naive is $\mathcal{O}(E(E + V \log V))$
- If we implement Dijkstra's algorithm with a binary heap
  - the complexity of Dijkstra's algorithm is $\mathcal{O}((E + V) \log V)$, so that the proposed method is $\mathcal{O}(2(E+V) \log V + E)$, since $E = \mathcal{O}(E \log V)$, then it has same time complexity as Dijkstra's algorithm, i.e., $\mathcal{O}((E+V) \log V)$.
  - the corresponding time complexity of naive is $\mathcal{O}(E(E + V) \log V)$

$\blacksquare$