CSCI 570 - Fall 2016 - HW4 with solutions

1. 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 O(1) time
- Extract-Median takes O(log n) time
- Insert takes O(log n) time
- Delete takes O(log n) time

Do the following:

(a) Describe how your data structure will work.
(b) Give algorithms that implement the Extract-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 $\lceil n/2 \rceil$ 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,

(a) Compare x to the current median (root of max-heap).

(b) If x < median, we insert x into the max-heap. Otherwise, we insert x into the min-heap. This takes O(log n) time in the worst case.

(c) If size(maxHeap) > size(minHeap)+1, then we call Extract-Max() on max-heap and insert the extracted value into the min-heap. This takes O(log n) time in the worst case.

(d) Also, if size(minHeap) >size(maxHeap), we call Extract-Min() on min-heap and insert the extracted value in to the max-heap. This takes O(log n) time in the worst case.

Extract-Median() algorithm:

Run ExtractMax() on max-heap. If after extraction, size(maxHeap) < size(minHeap) then execute Extract-Min() on the min-heap and insert the extracted value into the max-heap. Again worst case time is O(log n).

2. 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:

(a) 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.

(b) 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.*).

(c) The time complexity for processing *one number* must be better than $\Theta(k)$. Anything that is $\Theta(k)$ 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.

(a) Do not wait until k numbers have arrived at the server to build the heap, otherwise you would incur a time complexity of $O(k)$. 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.

(b) 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 $O(1)$ 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.

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

3. Consider the following modification to Dijkstra's algorithm for single source shortest paths to make it applicable to directed graphs with negative edge lengths. If the minimum edge length in the graph is $-w < 0$, then add $w + 1$ to each edge length thereby making all the edge lengths positive. Now apply Dijkstra's algorithm starting from the source s and output the shortest paths to every other vertex. Does this modification work? Either prove that it correctly finds the shortest path starting from s to every vertex or give a counter example where it fails.

Solution 1: No, the modification does not work. Consider the following directed graph with edge set $\{(a, b), (b, c), (a, c)\}$ all of whose edge weights are -1. The shortest path from a to c is $\{(a, b), (b, c)\}$ with path cost -2. In this case, $w = 1$ and if $w + 1 = 2$ is added to every edge length before running Dijkstra's algorithm starting from a, then the algorithm would output (a, c) as the shortest path from a to c which is incorrect.

Solution 2: Another way to reason why the modification does not work is that if there were a directed cycle in the graph (reachable from the chosen source) whose total weight is negative, then the shortest path is not well defined since you could traverse the negative cycle multiple times and make the length arbitrarily small. Under the modification, all edge lengths are positive and hence clearly the shortest paths in the original graph are not preserved.

4. You are given a weighted directed graph G = (V, E, w) and the shortest path distances $\delta(s, u)$ from a source vertex s to every other vertex in G. However, you are not given $\pi(u)$ (the predecessor pointers). With this information, give an algorithm to find a shortest path from s to a given vertex t in $O(|V| + |E|)$ time.

Solution 1:

(i) Starting from node t, go through the incoming edges to t one at a time to check if the condition $\delta(s, t) = w(u, t) + \delta(s, u)$ is satisfied for some $(u, t) \in E$.

(ii) There must be at least one node u satisfying the above condition, and this node u must be a predecessor of node t.

(iii) Repeat this check recursively, *i.e.* repeat the first step assuming node u was the destination instead of node t to get the predecessor to node u, until node s is reached.

Complexity Analysis: Since each directed edge is checked at most once, the complexity is $O(|V| + |E|)$. Note that constructing the adjacent list of G^{rev} in order to facilitate access to the incoming edges is needed, which is still bounded by $O(|V| + |E|)$ complexity.

Solution 2: Run a modified version of Dijkstra's algorithm without using the priority queue. Specifically,

```
(a) Set $S = \{s\}$
(b) While $t \notin S$
    i. For each node u satisfying $u \notin S$, $(v,u) \in E$ and $v \in S$, check if the condition
       $\delta(s,u) = w(v, u) + \delta(s, v)$ is true.
    ii. If true, add u to S, and v is the predecessor of u.
(c) EndWhile
```

Complexity Analysis: Since each directed edge is checked at most once, the complexity is $O(|V| + |E|)$.

5. We are given a directed graph G = (V, E) on which each edge $(u, v) \in E$ has an associated value r(u, v), which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret r(u, v) as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between any two given vertices.

Solution: Let P$*$ be the most reliable path connecting a given node s to another given node t. Since each edge fails independently of all other edges, the probability of the path P$*$ not failing equals $\prod_{(u,v)\in P*} r(u,v)$. Using the monotonicity of log($\cdot$) function, finding P$*$ is equivalent to maximizing $\log\left(\prod_{(u,v)\in P} r(u,v)\right) = \sum_{(u,v)\in P} \log r(u,v)$, or equivalently, minimizing $\sum_{(u,v)\in P} -\log r(u,v)$ over all paths P connecting s to t. Consider the following algorithm:

(a) Assign the edge weight $w(u, v) = -\log r(u, v)$ to each edge $(u, v) \in E$, with appropriate representation for edge weight of infinity.

(b) Run Dijkstra's shortest path algorithm and return the path found from node s to node t as the path P$*$.

Since $r(u,v) \in [0,1]$, all weights w(u,v) are non-negative and hence Dijkstra's algorithm gives the correct shortest path, and thus also the most reliable path connecting node s to node t.

6. Consider two positively weighted graphs $G = (V, E, w)$ and $G' = (V, E, w')$ with the same vertices V

and edges E such that, for any edge $e \in E$, we have $w'(e) = w^2(e)$.

(a) Prove or disprove: For any two vertices $u, v \in V$, any shortest path between u and v in G' is also a shortest path in G.

(b) Prove or disprove: If T is a minimum spanning tree of G, then it must also be a minimum spanning tree for G'.

Solution:

(a) The claim is false. Consider the following counterexample. Let G and G' be undirected graphs with V = {a,b,c} and E = {(a,b),(b,c),(c,a)}. Define the weight function w by w(a,b) = 6,w(b,c) = 4, w(c, a) = 3 and consider the shortest path between nodes a and b. The edge (a, b) constitutes the shortest path in graph G, whereas the edge set {(a, c), (c, b)} constitutes the shortest path in graph G'.

(b) The claim is true. Consider an execution of Prim's algorithm on G that led to the tree T. To see that the same execution sequence is valid on the graph G' we use the monotonicity of the $(\cdot)^2$ function and proceed by induction. With the same starting node s in both cases, squaring the edge weights does not change the minimum weight edge and hence the first edge added to the partial tree is the same for both graphs G and G'. Now assume that the same set of edges have been added to the partial tree up to stage k on both graphs. So the edges to consider at stage k + 1, *i.e.* the edges that do not form a cycle, are exactly the same for both graphs. Among these candidate edges, squaring the edge weights does not change the minimum weight edge so that addition of the same edge is valid for both G and G′. Hence, the partial trees are the same for both graphs after stage k + 1 and the proof is complete by mathematical induction.

7. Solve Kleinberg and Tardos, Chapter 4, Exercise 8.

Solution: Suppose by way of contradiction that T and $\hat{T}$ are two distinct minimum spanning trees for the graph. Since T and $\hat{T}$ have the same number of edges but are not equal, there exists an edge $e \in T$ such that $e \notin \hat{T}$. Adding e to $\hat{T}$ results in a cycle C. As edge weights are distinct, take e' to be the uniquely most expensive edge in C. By the cycle property, no minimum spanning tree can contain e', contradicting the fact that the edge e' is in at least one of the trees T or $\hat{T}$.

8. Solve Kleinberg and Tardos, Chapter 4, Exercise 21.

Solution: To do this, we apply the cycle property nine times, *i.e.* we perform BFS until we find a cycle in the graph G and then we delete the heaviest edge on this cycle. We have now reduced the number of edges in G by one, while keeping G connected and not changing the identity of the minimum spanning tree (again by the cycle property). If we do this 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 thus it has to be the minimum spanning tree of G. The running time of each iteration is O(m + n) for the BFS and subsequent check of the cycle to find the heaviest edge. Since $m \leq n + 8$ and there are a total of nine iterations, total running time is O(n).

9. Solve Kleinberg and Tardos, Chapter 4, Exercise 22.

Solution: Consider the following counterexample. Let G = (V, E) be a weighted graph with vertex set V = {v1, v2, v3, v4} and edges (v1, v2), (v2, v3), (v3, v4), (v4, v1) of cost 2 each and an edge (v1, v3) of cost 1. Then every edge belongs to some minimum spanning tree, but a spanning tree consisting of three edges of cost 2 would not be minimum.

10. Assume that you are given a graph G and a minimum spanning tree T of G. A new edge e is added to G (without introducing any other vertices) to create a new graph $\bar{G}$· Design an algorithm that given G, T and e, finds a minimum spanning for $\bar{G}$. Your algorithm should run in O(n) time.

Solution: We add the edge e to the minimum spanning tree T and this creates a unique simple cycle C. Let e_{max} be an edge of maximum weight in the cycle C. Then, removing this edge would give us a spanning tree $\bar{T}$ = T {e} \ {e_{max}} since we would have |V | − 1 edges and no cycles. To prove that $\bar{T}$ is a minimum spanning tree for the new graph $\bar{G}$, we consider two distinct cases.

(a) If w(e) = w(e_{max}), then by the cycle property, T is a minimum spanning tree of $\bar{G}$. Since, in this case, $\bar{T}$ has the same weight at T , $\bar{T}$ is a minimum spanning tree of $\bar{G}$ as well.
(b) If w(e) < w(e_{max}) then w = w($\bar{T}$) + w(e) − w(e_{max}) < w(T). Assuming that $\bar{T}$ is not a minimum spanning tree of $\bar{G}$, there must exist $\bar{T}_{opt}$ such that w($\bar{T}_{opt}$) < w($\bar{T}$) . Note that edge e must be contained in $\bar{T}_{opt}$, otherwise $\bar{T}_{opt}$ would be a spanning tree of G of weight less than the minimum spanning tree T. Removing edge e from $\bar{T}_{opt}$ gives two disjoint connected components (call them A and B). Since T is a spanning tree, there exists a unique edge $e^{'} \in$ T connecting A and B. By construction, $e^{'} \in$ C. We construct the spanning tree $T' = \bar{T}_{opt} \cup \{e'\} \setminus \{e\}$ which is contained in G by construction. We have,

$$\begin{aligned} W(\bar{T}_{opt} \cup \{e'\} \setminus \{e\}) &= w(\bar{T}_{opt}) + w(\{e'\}) - w(e) \\ &< w(\bar{T}) + w(\{e'\}) - w(e) \\ &= w(T) + w(e) - w(e_{max}) + w(e') - w(e) \\ &= w(T) + w(e') - w(e_{max}) \end{aligned}$$

Since $w(e') \leq w(e_{max})$, we have $w(T^{'}) < w(T)$ leading to a contradiction as T is a minimum spanning tree of G and $T^{'}$ is contained in G. Thus, $\bar{T}$ is indeed a minimum spanning tree of $\bar{G}$.

Running Time: All we need to compute is the edge e_{max}. Construct the unique path (call it P) in T that connects the two ends of e. This can be accomplished using BFS in O(n) time. The cycle C is path P concatenated with edge e and we can compute e_{max} as the maximum weighted edge in C.