Homework 4

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):
    a. • Find median takes O(1) time
    b. • Insert takes 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.

   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) (for the case of even n, median is the average of the middle two elements).

   Initialize size of maxheap (maxlen) and size of minheap (minlen) to 0. We want to maintain maxlen = minlen in case of even elements and maxlen = minlen + 1 in case of odd elements.

   Insert() algorithm: For a new element x
   • Compare x to the current root of max-heap (current median).
   • If x < median, we insert x into the max-heap. Increase maxlen by 1. Otherwise, we insert x into the min-heap, and increase minlen by 1. This takes O(log n) time in the worst case.
   • Now, if size(maxHeap) > size(minHeap)+1, then we call Extract-Max() on max heap, and decrease maxlen by 1 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 • Else: return root of min heap as median

Rubric
• 3 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.
• 2 pt: Correct Time Complexity

2. [20 points] A network of n servers under your supervision is modeled as an undirected graph G = (V, E) where a vertex in the graph corresponds to a server in the network and an edge models a link between the two servers corresponding to its incident vertices. Assume G is connected. Each edge is labeled with a positive integer that represents the cost of maintaining the link it models. Further, there is one server (call its corresponding vertex as S) that is not reliable and likely to fail. Due to a budget cut, you decide to remove a subset of the links while still ensuring connectivity. That is, you decide to remove a subset of E so that the remaining graph is a spanning tree. Further, to ensure that the failure of S does not affect the rest of the network, you also require that S is connected to exactly one other vertex in the remaining graph. Design an algorithm that given G and the edge costs efficiently decides if it is possible to remove a subset of E, such that the remaining graph is a spanning tree where S is connected to exactly one other vertex and (if possible) finds a solution that minimizes the sum of maintenance costs of the remaining edges.

Solution

We remove *S* and all its adjacent edges from G to form *G′* . We can first check in G is connected using BFS - if not, the required MST cannot be found. If it is connected, we run Prim's (or any other MST algorithm) to find the MST of *G′*. Then, among all edges adjacent to *S*, we find the one with the minimum maintenance cost and connect *S* to the MST using this edge.

Rubric

• 5 pt: provides a way to check whether such an MST is possible

• 10 pt: finds MST for a graph after removing all edges to/from S

• 5 pt: states least cost edge from S is added to MST.

3. [15 points] Prove or disprove the following:

- *T* is a spanning tree on an undirected graph *G* = (*V*, *E*). Edge costs in *G* are NOT guaranteed to be unique. If every edge in *T* belongs to SOME minimum cost spanning trees in *G*, then *T* is itself a minimum cost spanning tree.

- 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 $w'(e) = w(e)^2$ For any two vertices u, v in V, any shortest path between u and v in $G'$is also a shortest path in G.

Solution

1. False. Counter example: G is a Triangle abc.

ab = 2, bc = 2 , ac = 1
T = ab, bc
MST1 = ab, ac
MST2 = ac, bc

T is a spanning tree.
Ab is in MST1
Bc is in MST2
Thus, the given property holds for T, however, T is not a MST.

Rubric

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

2. False. Assume we have two paths in G, one with weights 2 and 2 and another one with weight 3. The first one is shorter in G' while the second one is shorter in G.

Rubric

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

4. [15 points] A new startup FastRoute wants to route information along a path in a communication network, represented as a graph. Each vertex represents a router and each edge a wire between routers. 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

Data structure change: we'll use a max heap instead of a min heap used in Dijkstra's priority queue

Initialization of the heap: Initially all nodes will have a distance (bandwidth) of zero from *s*, except the starting point *s* which will have a bandwidth of ∞ to itself.

Change in relaxation step. Based on the definition of a path's bandwidth, the bandwidth of a path from *s* to *u* through *u*'s neighbor *v* will be min(*d*(*v*), *weight*(*v*, *u*)), because of the definition of bandwidth of a path. Then, in the relaxation step, we will be changing:

$d(u) = \min(d(u), d(v) + weight(v, u))$

with

$d(u) = \max(d(u), \min(d(v), weight(v, u))$

Rubric

• 2 points for max heap

• 3 points for initialization

• 10 points for relaxation modification

5. [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 *O*(*k*). Anything that is *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. Initially, 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 (assume insert() executes Heapify() internally).

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 $O(1)$ time. If $x \le r$, ignore x. Otherwise, run Extract-min() and insert the new number x into the heap (can assume Heapify() is performed internally to maintain the structure of the heap.)

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

Rubric

- 2 points - states min-heap as the desired data structure
- 3 points - correct method for inserting when heap is not full
- 4 points - correct method for inserting when heap is full
- 1 point - correct time complexity

6. [15 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 (15 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 v, and its length by $\delta(u, v)$. Now, try setting each edge to zero. For each candidate edge $(u, v) \in E$, consider the path $s \to u \to v \to t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$ (both of these values are already computed from the two dijkstra calls above). Find the edge for which this length is minimized and set it to zero; the corresponding path $s \to u \to v \to t$ is the desired path. 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. Thus the total running time is the same as that of Dijkstra:

Rubrics:

• 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 have the same complexity as Dijkstra's algorithm

• 0 pt if the naive approach or any other approach with larger complexity than naive is proposed

• 3 pt for time complexity analysis.

7. [10 points] When constructing a binomial heap of size n using n insert operations, what is the amortized cost of the insert operation? Find using the accounting method.
Solution:
We make sure that at any point, we have a credit of >= 0. Insert involves creating a new order-0 tree (i.e. with just one element) (which is constant time) and then merging if there is another order-0 tree (i.e. merge 2 T0 into a T1) and so on.
Let's say inserting a tree with a single new element is assigned an amortized cost of 2. The actual cost of the i^{th} insertion is 1 + #merges(i). This means that for every insert we get a credit of 1 - #merges(i). Now, we observe that every 2^{nd} insertion has a merge of 2 T0 into a T1, every 4^{th} insertion has a merge of 2 T1 into a T2 and so on. Thus upto the first n insertions, the total #merges = $\lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \ldots < n$. Thus we can see that the total credit accumulated by this point is n - #merges > 0 as required. Thus, the amortized cost is bound by 2, i.e. it is $\Theta(1)$.

Rubrics:

• 3 pt - Correctly assigns amortized cost of 2 to each insert

• 3 pt - Correctly mentions actual cost to be 1 to each T0-creation, and additional 1 for each potential merge

• 3 pt - correctly shows credit >= 0 for n insertions (arbitrary n)

• 1 pt - concludes the amortized cost to be $\Theta(1)$