Homework 4

Jiahao Liu February 4, 2024

1. Design a data structure that has the following properties: Find median takes O(1) time, Insert takes O(lgn) time. (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation).
Describe how your data structure will work and Give algorithms that implement the Find-Median() and Insert() functions.

Just maintain a max heap, a min heap and a counting variable hsize. When hsize is even, the max heap contains the smaller half of the elements, and the min heap contains the larger half. When hsize is odd, the max heap contains one more element than the min heap.

```
Implementation of Find-Median() and Insert() operations:
Find-Median():
    if hsize is 0:, return NULL.
    else if hsize is odd: return the top of the max heap.
    else: return the average of the tops of the max heap and min heap.
Insert(x):
    if hsize is 0: push x into the max heap.
    else if hsize is odd:
        if x is less than the root of the max heap:
            push x into the max heap, pop the top of max heap and push it into the min heap.
        else: push x into the min heap.
    else:
        if x is less than the root of the max heap: push x into the max heap.
        else: push x into the min heap, pop the top of min heap and push it into the max heap.
```

The Find-Median only involves top operation of heaps, which takes O(1).
The Insert involves at most 3 push or pop operations of heaps with size at most n/2, which takes O(lgn).

2. 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.

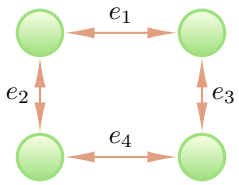
We can just modify the Kruskal's algorithm with an additional label for vertex S to indicate it has been connected in the spanning tree to one other vertex or not.

```
Kruskal(node S, graph G = (V, E)):
Sort the edges in E in non-decreasing order of their costs.
Initialize cnt = 0 to record number of edges in tree, res = 0 to record maintainance cost.
Label node S as unvisited.
Initialize the Union-Find set by let root of each vertex as itself. f[i] = i for all i in V.
For each edge e in the sorted order:
    if one of the endpoints of e is S and S has been visited: continue;
    Find the root node of each endpoint fu = find(e.u), fv = find(e.v);
    if fu is not equal to fv:
        Union(fu, fv); cnt++; res += e.cost;
        if one of the endpoints of e is S: label S as visited.
if cnt is not equal to n - 1: return NULL;
else: return res;
```

We remove S and all its adjacent edges from G to form a new graph G' = (V', E'). Then we can first use BFS to check if G' is connected or not.
If it's not connected, required MST can't be found.
If it's connected, use the Kruskal's algorithm to find the MST of G' and find the minimum cost e from all adjacent edges of S, add e to the MST.

3.(a) 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.

False. We can easily construct a graph with only four nodes and four edges with $e_1 = 1, e_2 = e_3 = e_4 = 2$:



Obviously, the MST of this graph is the tree with edges e_1, e_2, e_3 or e_1, e_2, e_4 or e_1, e_3, e_4. However, a spanning tree with edges e_2, e_3, e_4 is not a MST.

3.(b) 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

False. Different from the case of MST, where we only consider the difference of one corresponding edge weight in two graphs, the shortest path problem consider the sum of all edges' weights in the path, so we can't deduce from $x_1 + x_2 > y_1 + y_2$ to $\sqrt{x_1} + \sqrt{x_2} > \sqrt{y_1} + \sqrt{y_2}$. For example, 1 + 25 > 9 + 16, but $\sqrt{1} + \sqrt{25} < \sqrt{9} + \sqrt{16}$. So the shortest path in G' is not necessarily the shortest path in G.

4. 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.

Instead of minimum distance dist[u] for node u to source s, we can just maintain a bandwidth bw[u] for bandwidth of node u to source s.

```
Dijkstra(node s, graph G = (V, E, w)):
```

Initially, $S = Null, bw[s] = \infty$, $bw[u] = 0$ for all other nodes u.
While $S \neq V$:
    Select a vertex $t \in V - S$ with the $max\{bw[t]\}$ and add t into Set S.
    Look through all the neighbor nodes of t:
        if neighbor nodes $j \in V - S$: $bw[j] = \max(bw[j], \min(bw[t], w(t, j)))$.
bw[u] would be the maximum bandwidth from source s to u, if there are still bw[u] = 0, that means u is not connected to s.

Proof of correctness. The $max\{bw[t]\}$ could make sure node t has maximum bandwidth bw[t]. Proof by contradiction. Suppose there is bigger bandwidth path from s to t, then there must be an edge (s', t') with s' in S and t' in V-S. Because we choose the $max\{bw[t]\}$ node, $bw[t'] \leq bw[t]$ then the path bandwidth from s through (s', t') to t must be smaller or equal to bw[t'], which is smaller or equal to bw[t]. Contradiction. □

5. 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.)
Design an algorithm on the server to perform its job with time complexity better than O(k)

We just need to implement a min heap with array of size k from empty.
When the heap size is less than k, we just push the new number into the heap. O(lgk)
When heap size reaches k and a new number comes, we compare it with the top of the heap. If it is larger, we pop the top and push the new number into the heap. O(lgk)

6. 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 algorithm with same run time complexity as 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.

Apart from the dist[u] in classic Dijkstra' algorithm, we also need to maintain a starDist[u] for each node u to record the lowest cost path from s to u with exactly one edge weight set to zero.

```
Dijkstra(node s, graph G = (V, E, w)):
```

$S = Null, dist[s] = starDist[s] = 0$, $dist[u] = starDist[u] = \infty$ for all other nodes u.
While $S \neq V$:
    Select a vertex $t \in V - S$ with the $\min\{dist[t]\}$ and add t into Set S.
    Look through all the neighbor nodes of t:
        if neighbor nodes $j \in V - S$:
            $dist[j] = \min(dist[j], dist[t] + w(t, j))$;
            if $starDist[j] > \min(dist[t], starDist[t] + w(t, j))$:
                $starDist[j] = \min(dist[t], starDist[t] + w(t, j))$;
                $parent[j] = t$; Record the shortest star path from s to j
Repeat until $S = V$, that's all nodes have find their starDist from s.

We only need to modify the Dijkstra's algorithm to update starDist[u] after we update dist[u], so the time complexity is still O(VlgV + ElgV) = O((V+E)lgV).
We can just run Dijkstra's algorithm to find the shortest path from s to all other nodes. And reverse the graph to find the shortest path from t to all other nodes. O(2mlgn)
Use the above two shortest path results from s and t, we can just iterate through all the edges in G, say (u, v) and find $\min_{e \in E}\{dist[s, u] + 0 + dist[v, t]\}$. O(m)

7. 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.

We can charge each insert operation 2 units of cost. Let's start from empty heap:

Operation	Charge	Actual Cost	Credit
Insert	2	1	1
Insert	2	2	1
Insert	2	1	2
Insert	2	3	1
Insert	2	1	2
Insert	2	2	2
Insert	2	1	3
Insert	2	4	1

Actually, for new node inserted as B_0, it will serve as the root of some tree B_i and be merged into B_{i+1} at most once, so we charge 2 units for each node insert, which is enough to cover the cost of the expensive insert operation. Let's say 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 B_0 into a B_1, every 4^{th} insertion has a merge of 2 B_1 into a B_2 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. The amortized cost is bound by 2, i.e. $\Theta(1)$.