Review Assignment

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it.

Problem 1: Heap Algorithm Design

Consider a situation where your data is almost sorted. For example you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. We will focus only on the time-stamps.

To simplify this problem assume that each time-stamp is an integer, all time-stamps are different, and for any two time-stamps, the earlier time-stamp corresponds to a smaller integer than the later time-stamp. The time-stamps arrive in a stream that is too large to be kept in memory completely. The time-stamps in the stream are not in their correct order, but you know that every time-stamp (integer) in the stream is **at most hundred positions away** from its correctly sorted position.

Design an algorithm that outputs the time-stamps in the correct order. You can only use a **constant amount of storage**, i.e., the memory used should be independent of the total number of time-stamps processed. State and briefly justify the time and space complexity of your algorithm considering n as total number of time-stamps, and every time-stamp is at most k positions away from its correctly sorted position.

Hint: Use a heap. You can invoke properties of heaps shown in class without re-proving them.

Solution

1. In this problem we know for sure that values can be at most hundred positions away from their correctly sorted position. Hence, we can maintain a heap of (100 + 1) elements.

2. Every time we get a new value we insert it into its correct sorted position in the Heap in logarithmic time and take out the very first element from the heap.

3. In a heap of size 101 we can always guarantee that very first(root) element in the heap is at its correct position.

4. To prove this let us say that at the very beginning our heap is empty. Hence, it is in sorted order. Now we start adding new elements into our heap such that each insertion adds the given element at its sorted position in the heap in logarithmic time. This guarantees that our heap still remains in sorted order. Also, we do not remove any element from the heap until it is full (i.e it has 101 elements in our case).

5. At this point we have 101 elements in our heap and all elements are in sorted order. We know that any particular element can only be at most 100 places away from its correct position. This guarantees that the smallest time-stamp of the stream has to be present in between first 101 elements (its actual position is 0 but it can be displaced up to position 100 but not beyond that). Hence, when our heap reaches the size of 101 we can be sure that the smallest time-stamp is present in the heap. However, all values in our heap are in sorted order hence, it is also true that the smallest time-stamp has to be the very first(root) element in our heap. There can be no other element in the stream which has smaller time-stamp value than the very first element of our heap.

6. Now we remove the very first element of the heap and output it as the first sorted element. Heap size again goes down to 100. We again insert one more element in our heap in logarithmic time and again we can guarantee that the current top of the heap has the 2nd smallest time-stamp value. This procedure can be done in a loop as long as there is a data available in our stream.

7. **Time and Space complexity Analysis:** Since, this algorithm never allows heap to grow beyond k+1 elements, it uses **O(k+1) space** which is constant in general(In our example k = 100). In our algorithm every insert operation on heap takes O(log (k+1)) time (k+1 = size of heap). As our stream contains n elements, we do insert operation for n elements. Hence, total time complexity will be **O(nlog(k+1))**.

Problem 2: Directed Acyclic Graphs (DAGs)

Describe an algorithm to find the longest path in a DAG.

Solution

The algorithm is as follows:

(a) Find a topological ordering of the DAG.

(b) For each node v in the topological ordering, look at its incoming edges and the nodes these edges are coming from. Call this set of nodes S. If w is the maximum length path found among the nodes in S, then set the value of the maximum length path to v as $w + 1$. If node v has no incoming edges, then set the value of the maximum length path to v as 0.

(c) The maximum length path in the graph is the maximum value recorded among all nodes in the graph.

Proof of correctness: The proof of correctness utilises the fact that given a topological sorting of a DAG, the incoming edges to a node v can only be from nodes before v in the queue.
Proof by contradiction: Assume the path returned by the algorithm is not in fact the actual maximum weight path (denoted by P). It's easy to see that the start node of P must always be a node with in-degree 0, because otherwise we could increase the length of the path by including the 0 in-degree node as well.
If our algorithm is incorrect, then it means that an error occured while calculating the maximum weight paths to nodes in the graph G, and let v_1 be the first node for which an error occurred. This means that there exists some path to v_1 that wasn't accounted for. Let S denote the set of nodes that appear before v_1 in the topological sort. Since we followed the topological sort, all nodes in S have correct values for their max weight paths, and all paths from nodes in S to v_1 were accounted for. Thus, the path that the algorithm ignored while calculating the max weight path to v_1 must have come from nodes behind v_1 in the topological sort. But this violates the definition of the topological sort, and is a contradiction.
Runtime: The topological sort takes $\mathcal{O}(n + m)$ time.
The number of comparisons made by the algorithm for a node v is $\mathcal{O}(d(v))$, where $d(v)$ is the in-degree of v. Summing over nodes, you get the runtime for step (b) as $\mathcal{O}(m)$ (since sum of in-degrees is equal to number of edges).
This gives a total runtime of $\mathcal{O}(n + m)$.

Problem 3: Graph Algorithm

Let G be an undirected weighted graph with weight function $w : E \to \{1, 2, 3, ..., k\}$, where k is a small constant. Let u be an arbitrary node in G. Provide an algorithm which computes the length of the shortest paths between u and any other node in the graph in time $O(k(n + m))$.

Solution

For any edge of weight w we create w in between edges of weight 1 in $O(k(n + m))$. After that we can just run BFS in the newly created graph and calculate the value of every shortest path using the layers provided by the algorithm.

Problem 4: Counting Shortest Paths

A number of art museums around the country have been featuring work by an artist named Mark Lombardi (1951-2000), consisting of a set of intricately rendered graphs. Building on a great deal of research, these graphs encode the relationships among people involved in major political scandals over the past several decades: the nodes correspond to participants, and each edge indicates some type of relationship between a pair of participants. And so, if you peer closely enough at the drawings, you can trace out ominous looking paths from a high-ranking U.S. government official to a former business partner to a bank in Switzerland to a shadowy arms dealer.

Such pictures form striking examples of *social networks*, which have nodes representing people and organizations and edges representing relationships of various kinds. And the short paths that abound in these networks have attracted considerable attention recently, as people ponder what they mean. In the case of Mark Lombardi's graphs, they hint at the short set of steps that can carry you from the reputable to the disreputable.

Of course, a single, spurious short path between nodes v and w in such a network may be more coincidental than anything else; a large number of short paths between v and w can be much more convincing. So in addition to the problem of computing a single shortest v-w path in a graph G, social networks researchers have looked at the problem of determining the *number* of shortest v-w paths.

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph $G = (V, E)$, and we identify two nodes v and w in G. Give an algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

Solution

We will solve the more general problem of computing the number of shortest paths from v to *every* other node.

We perform BFS from v, obtaining a set of layers $L_0, L_1, L_2, \ldots$, where $L_0 = \{v\}$. By the definition of BFS, a path from v to a node x is a shortest v-x path if and only if the layer numbers of the nodes on the path increase by exactly one in each step.

We use this observation to compute the number of shortest paths from v to each other node x. Let $S(x)$ denote this number for a node x. For each node x in L_1, we have $S(x) = 1$, since the only shortest-path consists of the single edge from v to x. Now consider a node y in layer L_j for $j > 1$. The shortest v-y paths all have the following form: they are a shortest path to some node x in layer L_{j-1}, and then they take one more step to get to y. Thus $S(y)$ is the sum of $S(x)$ over all nodes x in layer L_{j-1} with an edge to y.

After performing BFS, we can compute all these values in order of the layers; the time spent to compute a given $S(y)$ is at most the degree of y (since at most this many terms figure into the sum from the previous paragraph). Since we have seen that the sum of degrees in a graph is $O(m)$, this gives an overall running time of $O(m + n)$.