15-351 HW4 Solutions

October 19, 2020

1. The degree-constrained minimum spanning tree problem is this: you are given an undirected graph $G = (V, E)$ with positive weights on the edges, and you are also given an integer k. You must find a minimum weight spanning tree T of G (as we saw in class) with the extra restriction that no vertex has more than k edges adjacent to it in T. That is, if $k = 3$ then the left MST is allowed, but the right MST is not. This problem is hard, and no polynomial-time algorithm is known for it.

Explain how to apply A* to solve this problem. You don't need to explain the A* algorithm; just explain how you would use an A* "library" to solve the problem. Briefly explain why your algorithm will find an optimal solution. You do not need to show anything about the running time.

Solution: In the state graph, every state is a spanning tree of a subset of nodes that satisfies the extra restriction. Denote the subset of nodes by $S \subseteq V$, and the set of unsaturated nodes in S by S'. Some simple and admissible heuristics are

(a) the length of the smallest edge connecting a node in S and a node in $V \backslash S$.

(b) the length of the smallest edge connecting a node in S' and a node in $V \backslash S$.

(c) the total length of the smallest edge incident to each node in $V \backslash S$.

(d) the length of the MST of the graph where we merge nodes in S into a supernode.

(e) the length of the MST of the graph where we remove all saturated nodes (i.e., $S \backslash S'$) and merge the rest of nodes in S into a supernode.

Below are wrong heuristics:

(a) any heuristics that use the coordinates or Euclidean distances, because we don't know the coordinates and the distance may not satisfy the triangular inequality.

(b) the total length of two or more edges connecting S and $V \backslash S$, because there might be exactly one such edge, and the other selected edges might have large weights.

(c) the length of the shortest path that starts from and ends at S to S and doesn't pass any other nodes in S, because this forms a cycle and thus may have a larger weight.

2. Give an $O(|E|)$-time algorithm to check whether a given tree T is a shortest-path tree of an undirected, connected graph $G = (V, E)$ with positive weights $d(u, v)$ on each edge.

Hint: check whether the edges that are not in T are correctly excluded.

Solution: We find the distances from the source to any nodes using BFS/DFS, and denote it by $\text{dist}(v)$ for node v. We then enumerate every edge $e = (u, v)$ and check if $|\text{dist}(u) - \text{dist}(v)| \leq w(e)$ holds. We output YES if this holds for all edges, and NO otherwise.

The runtime is $O(|E|)$, because we spend $O(1)$ time on every node and edge.

Correctness: 1) If this tree is not a shortest-path tree, among all shortest paths not included in the tree, let P be the one with the smallest number of edges. And then, the last edge in P, denoted by $e = (u, v)$,

is not in the tree. Because P is a shorter path from s to v and because the s-u segment in P is shortest, we know $\text{dist}(v) - \text{dist}(u) > w(e)$ and therefore we will output NO. 2) If we output NO, then we find an edge $e = (u, v)$ such that $d(u) - d(v) > w(e)$ w.l.o.g., which means we find a shorter path to u via e, and therefore this tree is not a shortest-path tree.

3. The skip list introduced in lectures supports several expected $O(\log n)$-time operations, such as `find`, `insert`, and `delete`. Please modify the skip list to support another operation `index` which takes an integer number k as input and outputs the k-th smallest elements in the skip list. The expected runtime should also be $O(\log n)$ (assuming k is a constant). Show how to handle this query, and how to modify other operations if necessary.

Hint: Try to store an extra piece of information at every node and update it when necessary.

Solution: For each node (including the leftmost $-\infty$ and excluding the rightmost $+\infty$), we keep track of the number of integers/elements between it and the node to the right, and denote this by size. The size includes the current node and excludes the next node to the right. E.g., the size of every node at the lowest level is one.

Index. Assume we want to find the kth smallest element. At node u, we go down if $\text{size}(u) \geq k$. Otherwise, we decrease k by $\text{size}(u)$ and go right. We repeat this until $k = 1$.

Deletion. Assume we delete element x. Let u be any node from which we go down. If the next node of u is a node of x, we increase $\text{size}(u)$ by $\text{size}(x) - 1$, and update the pointer of u. Otherwise, we decrease $\text{size}(u)$ by one.

Insertion. Assume we're going to insert element x. As we go right and down to find the position for x, we keep track of the index of visited elements. Specifically, the index of the leftmost $-\infty$ is 0. Every time we go down, the index is the same; every time we go right from node u to node v, the v's index is equal to u's index plus $\text{size}(u)$. After we create nodes for element x, assume the height is h. Now let u be any node from which we go down and denote by δ the difference between the indices of u and x. If u's height is greater than h, we increase u's size by 1. Otherwise, we set $\text{size}(h) = \text{size}(u) + 1 - \delta$ and set $\text{size}(u) = \delta$.

The runtime analysis is similar to the one introduced in lectures. We don't visit extra nodes and only spend $O(1)$ time at every visited node, so the overall runtime is unaffected.

The skip list can be thought of as a generic rooted tree where every node can have any number of children. The root is the leftmost $-\infty$ at the highest level. The first child of each node is the node below itself. And the next sibling is the node to the right, if that node is not the first child of some other node. So, the size stored at each node is actually the number of its children.

4. Design an $O(\log n)$-time algorithm that takes a real number a and a positive integer n as input and finds a^n. All operations you can use is addition, subtraction, multiplication, division, and rounding a real number up/down to an integer, all of which takes $O(1)$ time. You are not allowed to use the logarithm or exponentiation, because these two operators are not defined on all groups.

Hint: The result a^n might be very large or small, but you do not need to worry about it. Also, for any $m \in \{1, \ldots, n-1\}$, we have $a^n = a^{n-m}a^m$.

Solution: **Answer 1** Let $f(a, n) := a^n$. The base case is $f(a, 0) = 1$. We use recurrence on the second argument:

$$f(a, n) = \begin{cases} f(a, n/2)^2 & n \text{ is even} \\ f(a, (n-1)/2)^2 \times a & n \text{ is odd} \end{cases}$$

In either case, we perform $O(1)$ operations and reduce the size (n) to half. So the runtime is $O(\log n)$.

Wrong Answer $f(a, n) = f(a, \lfloor n/2 \rfloor) \times f(a, \lceil n/2 \rceil)$, because the runtime is $O(n)$.

Answer 2 We use the binary representation of n: let $k_1 < k_2 < \dots k_m$ be integers such that $n = \sum_{i=1}^{m} 2^{k_i}$. In other words, the k_i-th bit is 1. Then we find a^{2^i} for any $i \le k_m$, using the same recurrence. Finally, we calculate $a^n = \prod_{i=1}^{m} a^{2^{k_i}}$.

Answer 3 A better implementation that takes $O(1)$ space is by iteration and based on the binary representation of n.

(a) Set a temporary variable $b = a$ and a cumulative variable $o = 0$

(b) Repeat until $n = 0$

i. If n is odd, multiply o by b

ii. Update b to b^2

iii. Update n to $\lfloor n/2 \rfloor$

(c) output o

5. You want to draw a two-dimensional skyline like the following: You are given a list of the building x-coordinates and their heights:

$$(l_1, h_1, r_1), (l_2, h_2, r_2), \dots, (l_n, h_n, r_n)$$

This list will be sorted in increasing order of left-most x-coordinate. Design an $O(n \log n)$ algorithm to produce the skyline line to draw in the format:

$$x_1, y_1, x_2, y_2, x_3 \dots$$

meaning that at x_1 we draw a building at height y_1 until x_2, at which point we draw a line up or down to high y_2 and then continue horizontally until x_3 and so on. Note that for this problem, you need only *sketch* the main ideas without providing detailed discussion for algorithm correctness.

Solution: Answer 1

Use divide and conquer. The base case is the skyline of one building which is trivial. Then, to draw the skyline of n buildings, we 1) divide these building into two parts evenly and recursively call the subroutine to draw the skylines of the two halves, and 2) merge the two skylines in a merge sort-like manner in $O(n)$ time. The details about merging are beyond the sketch of the main idea.

Let $f(n)$ be the runtime of drawing the skyline of n buildings. And then the recursion is

$$f(n) = 2f(n/2) + O(n)$$

and therefore $f(n) = O(n \log n)$.

This is correct because the height is the maximum. By induction, we can prove the height of every location is always the height of the highest building covering this location during recursion.

Answer 2

We sweep from left to right, and maintain a heap. In the heap, every element is a building, and the building with the largest height is at the top.

When we sweep from the left to right,

(a) Every time we reach the left-most point of a building, we insert it into the heap.

(b) When we reach the right-most point of the currently highest building, we delete the top until the top's right-most point is to the right of the current x-coordinate, i.e., we find the highest building that has not expired. (It also correct to remove the building that is not the top when we reach its right-most point, if we sort the right-most points or use another heap to efficiently find the building with the smallest right-most point to be removed.)

(c) If the height of the building at the top changes, we output the current x-coordinate and the new height.

The heap can be substituted with a variety of data structures, including skip lists and balanced binary search trees, as long as the data structures supports $O(\log n)$-time finding minimum/maximum, insertion, and deletion. If we use skip list or balanced binary tree, where the elements are sorted, we can discard any building whose right-most point and height are both smaller than those of another building. If the rightmost points are also guaranteed to be non-decreasing, we can replace the heap/BST with a queue and the overall runtime is $O(n)$.

We insert and delete every building at most once and each time the runtime is $O(\log n)$. So the overall runtime is $O(n \log n)$.

1) The building at the top of this heap is always valid. 2) All currently valid buildings are in the heap, although some expired buildings might also be in the heap. These guarantee the heights are correct. The x-coordinates are correct because height changes only at the boundaries of some building.

This problem is included in LeetCode and its ID is 218. You can implement your algorithm and submit it for online judgement, if you want.