COL106 Practice Problems Solutions

Viraj Agashe

November 2021

1 AVL Trees

1.1 Merging AVL Trees

We are given two AVL Trees T_1 and T_2 of height h_1 and h_2 respectively. Additionally we know that all keys in T_1 are smaller than all keys in T_2. Notation: $ht(v)$ denotes height of subtree rooted at v.

Let us assume that $h_1 \geq h_2$. The general idea we will use is that we find a sub-tree of height *close* to h_2 in the tree T_1 and attach that subtree and T_2 to some common node. After that we can balance this AVL tree in $O(1)$ time using rotations. The algorithm is as follows:

1. First, perform AVL-delete on T_2 and delete the smallest element (say x) in it, i.e. the element which we get by following all left-links. This will take us $O(h_2)$ time. Assume that the height of the tree T_2 after deletion is h'.
2. Next, find an element $v \in T_1$ using only right-child links such that $ht(v) = h'$ or $h' + 1$. Note that at any right child link the height may change by at most 2, so we will always find such a node. Also store the parent of this node (say v'). This step is $O(h_1)$ if h_2 is small compared to h_1.
3. Now create a tree with the root as x, left subtree as the subtree rooted at v and the right subtree as the entire tree T_2 (after deleting x). Since height of subtree rooted at v is h' or $h' + 1$, this tree is balanced. Call this tree T^*.
4. Finally, connect this tree T^* with a right-link to the parent v' of v. The height of T^* is one greater than $ht(v)$. Balance factor of v' may remain same or change by 1, which can be fixed by an $O(1)$ double rotation.

The running time of the algorithm is $O(h_1 + h_2)$.

1.2 Splitting an AVL Tree

We are given an AVL Tree T and element $x \in T$. We want to split T into T_1 and T_2 such that $T_1 < x < T_2$. Let the height of T be h.

1.2.1 $O(h^2)$ algorithm

1. Start from the root and use BST-find(x) with a tweak. Suppose at any node v_i we take a right child link, store the subtree T_i rooted at the left child of v_i, as well as the vertex v_i.
2. In the worst case when x is the rightmost element of the tree we will have stored trees of heights $h - 1, h - 2, ...1$ as well as $O(h)$ nodes.
3. Use the merge AVL tree algorithm as described in Q1 to merge all of these AVL trees, and finally insert these $O(h)$ nodes one by one as well. Both steps are $O(h^2)$.

1.2.2 $O(h)$ algorithm

Unable to think of an $O(h)$ algorithm so far.

2 Graph Traversals

2.1 Bipartite Graphs

We wish to check whether any given graph has an odd-length cycle using BFS.

2.1.1 Simple Case

First, given a point v on a cycle, we want to check whether it is of odd length or not.

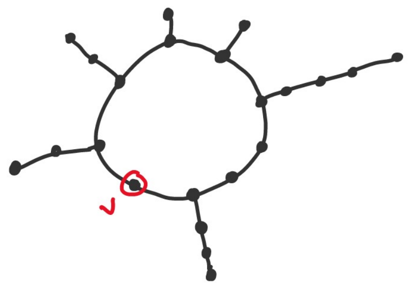

Figure 1: Graph of special form

Note that we only visit a vertex twice if it is part of a cycle. Using this, we follow the BFS algorithm for calculating distance discussed in class with a small change. If `w.visited = 1` for any neighbour of x then we compare `w.distance` with `x.distance+1`. If they are the same, then we have an even length cycle, else the cycle is of odd length. This is because in an even length cycle, while performing BFS, the node where the clash occurs will be at an equal distance from x no matter how we visit it. In an odd length cycle, the distances will be different in a clockwise and anticlockwise manner.

2.1.2 General Case

In the general case to detect an odd length cycle, the algorithm is very similar. If there is an odd length cycle in the graph, then while performing BFS for some point in the cycle we will obtain different distances from the vertex v we started the BFS from. So in $O(n + m)$ time we can detect an odd-length cycle in any graph.

2.2 Chess Board

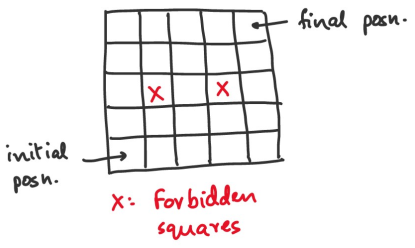

Figure 2: Chess Board

In an $N \times N$ chess board, we want to find the minimum number of steps needed for a knight to reach the final position from the initial position given that some squares are forbidden.

We can represent each square of a chess board as a vertex. Two vertices have an edge between them if a knight can move between them. Add an attribute `forbidden` to all the vertices and set it to `true` for the forbidden squares. Now perform normal BFS on the graph to compute the shortest distance. If we encounter any forbidden vertex, we can simply skip it.

Time Complexity: To create the adjacency lists, it will take us $O(N^2)$ time. The DFS step will take us $O(N^2+m)$ steps. If N is large, degree of each vertex will be roughly 8. So number of edges is also $O(N^2)$. Thus the overall time complexity of this solution is $O(N^2)$.

3 Sorting

3.1 Number of Inversions

Given an array $A = [a_1, a_2, ...a_n]$ we wish to find the size of the set,

$$\mathcal{S}_A = \{(i,j) \mid a_i > a_j, i < j\}$$

We want to do this in $O(nlogn)$ time. (This is called the set of inversions)

3.1.1 Solution 1: Using Merge-Sort

In merge-sort we split the array A into two halves A_1 and A_2, recursively call merge-sort on both and then merge them to create a sorted array. Here, we can modify mergesort to also return the number of inversions in that array. Suppose merge-sort(A_1) and merge-sort(A_2) correctly return sorted arrays A_1 and A_2 as well as the number of inversions n_1 and n_2 of those two arrays.

While merging the arrays to create the array A, suppose we maintain an integer n, initialized to $n = n_1 + n_2$. We choose the *next* minimum of A_1, A_2 and keep inserting it into A. If the next minimum is chosen from A_2 we increment n by 1.

For example, suppose $A_1 = [1,2,7]$ and $A_2 = [4,5,6]$. Then first we insert $1,2$ into A. The next minimum element is 4 from A_2, so we increment n by 1. Similarly for $5,6$ we increment n by 1. So in addition to the inversions in A_1 and A_2 themselves, there are 3 inversions.

3.1.2 Other Solutions

Solutions using other sorting algorithms/other methods here.

3.2 QSortInPlace Stability

We wish to verify whether `QSortInPlace(A,i,j)` as discussed in class is a stable sorting algorithm. We claim it is **unstable** in general. Consider the array $A = [1,2,6^*,6,3,4]$ for example. Suppose we choose the pivot element as 6^*. Then by the invariant of the algorithm, on using `pivotedRearrange`, $A[i,k-1] \leq 6$ and $A[k+1,j] > 6$. But this means that 6 now comes before 6^* in the sorted array, which implies it is unstable.