Reflection Essay - Assignment 5

Prasann Viswanathan

Question 1:

1. My efficient solution is based off of the method found in this link https://www.techiedelight.com/change-elements-row-column-j-matrix-0-cell-j-value-
This method uses O(1) auxiliary space (just two flags and iteration variables, 4 variables total) to report the final rook sweep in O(m*n) time. The method is the same as mentioned in the webpage. We use the grid to itself keep track of what columns and rows to set to 1 based on what entries in column 1 and row 1 we change. So the auxiliary space needed is just O(1). As for time complexity, we make one sweep of the array to find the rows and columns to change, and another sweep to change the values. So total steps take 2*m*n steps approximately. Implying a complexity of O(mn). Before this I was setting rows and columns based on whether I saw a 1. This implementation was O(mn(m+n)) but it passed all cases on hackerrank.

2. I don't think it is possible to do better than O(mn) because you would need that many print statements at least to print out each cell value. I think it can be done if additional auxiliary space is allowed. Whilst taking input we mark out in a row array and a column array (O(m+n) space) whether to change the row/column or not
But in printing it would take O(mn) so I don't know how to do better.

Question 2:

Since we are dealing with a Directed Acyclic Graph, by the property of a DAG, we know that a graph is a DAG ⇔ It can be Topologically sorted. This implies there exists a natural partial ordering of the vertices in the given graph.
Suppose for n vertices, without loss of generality the vertices are numbered such that n1 ≥ n2 ≥ … ≥ n_n topologically. This also implies that in a DFS traversal of the graph from n1, we reach n_n in less than or equal to O(n+m) steps.

We will need an additional n length array with all entries set to 0 and the nodes marked by vi as 1
And we will need a k length stack with vk at bottom and v1 at top (path stack)

Proposed solution: Use a DFS stack and perform DFS till the stack is empty.

```
    Start with i=1
    Push v_i on the stack, pop it from path stack and perform DFS till we reach v_{i+1}
        This is done by comparison with the new top of the path stack
        Also, check in O(1) using marked array if a node marked is reached
        If this node is not equal to top of path stack, we have overshot
        In this case, pop current node and go to previous, resuming DFS from there.
        (This will avoid all pathological cases, minimizing double traversal of edges.)
    If v_{i+1} found, make i = i+1, pop all elements and push v_{i+1} in stack as new i
    Else the path doesn't exist, eventually vi will be popped and the stack will be empty
    End loop when v_k is found
```

Time complexity:
The topologically least node takes O(n+m) to be reached from the topologically greatest node. Since existence of a path implies v1 ≥ v2 ≥ … ≥ vk, we know steps taken for
(v1 → v2) + (v2 → v3) … (vk-1 → vk) ≤ Longest path from v1 to vk = O(n+m)

Correctness:
Suppose a path exists from v1 → v2 → … → vk
This implies that at each iteration, when we perform DFS v_i we are sure to find v_i+1. This ensures that if a path exists we will definitely find it
Suppose a path does not exist. This implies that for some vi there is no path to vi+1. In DFS starting from vi, in such a case, we will not find vi+1 and will return to vi, causing it to pop off and us to exit the loop. The termination flag of vk being found will not be set reporting the non-existence of any path.
Therefore,
Existence of a path ⇔ Algorithm returning true.

Now as far as reconstructing the path is concerned, we can start with an empty list at each stack iteration, pushing the node values that make up the path from v_i to v_i+1. There will be a total_path list, to which we will append each v_i to v_i+1 path. This will allow us to reconstruct one of the paths containing v1, …, vk

Question 3:
My algorithm is broken into 3 main steps, all involving DFS
Arbitrarily I choose node 1 as the root of this tree structure and find the number of candy shops in each node as a subtree, rooted at 1, inclusive of whether the start node is a candy shop or not.

For example in a complete 7 node BST, if all nodes are candy shops, the leaves have value 1, their parents have value 3 and the root has value 7. These values for each node are stored in subtree_candy. Operation takes O(V+E) = O(2n) = O(n)

Then I find the aggregate distance of the shops from the root. This again takes O(n), same as the previous case, and the algorithm involves keeping track of depth and adding it recursively if the node is a candy shop.

Lastly, I make use of the fact that moving to a child node, the aggregate distance is incremented by 1 for shops we are going further away from and decreased by 1 for those we are going near

The formula thus is = parent_distance - (subtree shops) + (k-subtree shops)

This is filled up once again using DFS in O(n)

The overall runtime is thus O(n) and works fast enough for our requirement.

Time complexity justified above, addition of 3 parts, each O(n) => O(n) complexity

No. A simple example is a spider web case. Imagine a spider with 8 legs and a candy shop at each leg end, but no shop in the centre. The cost from the centre is 8, however the cost from any leg is 14. Simple counterexample to disprove the statement.