Theoretical Assignment 3

Gurpreet Singh - 150259 . Nikita Awasthi - 150453

April 2017

Question 1

Pseudocodes

Algorithm 1 Remove smallest element

```
procedure ExtractMin($T$)                ▷ where T is the array containing the Young tableaux
    $min = T[0, 0]$
    $T[0, 0] = \infty$

    RestoreExtract($T, 0, 0$)

    return $min$
end procedure


procedure RestoreExtract($T, i, j$)                ▷ To restore the Young Tableaux property
    $min_i = i$
    $min_j = j$

    if $i < m - 1$ and $T[i, j] > T[i + 1, j]$ then
        $min_i = i + 1$
    end if

    if $j < n - 1$ and $T[i, j] > T[i, j + 1]$ then
        $min_j = j + 1$
    end if

    if $min_i \neq i$ or $min_j \neq j$ then
        Swap($T[i, j], T[min_i, min_j]$)

        RestoreExtract($T, min_i, min_j$)
    end if
end procedure
```

Algorithm 2 Inserting new element in the tableaux

```
procedure INSERT($T, a$)
    if $T[m-1, n-1] < \infty$ then
        return $-1$                                                    ▷ If Tableaux is full
    end if

    $T[m-1, n-1] = a$                    ▷ Inserting element at the last position of the tableaux

    RestoreInsert($T, m-1, n-1$)
end procedure


procedure RESTORE($T, i, j$)        ▷ Recursive procedure that percolates the element up to its position
according to tableaux property
    $max_i = i$
    $max_j = j$

    if $i > 0$ and $T[i, j] < T[i-1, j]$ then
        $max_i = i - 1$
    end if

    if $j > 0$ and $T[i, j] < T[i, j-1]$ then
        $max_j = j - 1$
    end if

    if $max_i \neq i$ or $max_j \neq j$ then
        Swap($T[i, j], T[max_i, max_j]$)

        RestoreInsert($T, max_i, max_j$)
    end if
end procedure
```

Algorithm 3 Searching of an element

```
procedure SEARCH($T, k, row, column$)        ▷ To search for given key k and row an column are current
indices
    if $row >= m$ or $column < 0$ then
        return false
    end if

    if $T[row][column] == k$ then
        return true
    else if $T[row][column] < k$ then
        return SEARCH($T, k, row + 1, column$)
    else
        return SEARCH($T, k, row, column - 1$)
    end if
end procedure
```

Proof of Correctness

Lemma 1: The minimum element of the tableaux is in T[0][0].
Proof: According to the definition of the tableaux, $T[0][0] \leq T[i][0] \leq T[i][j]$. Since this is true for any

general value of i,j, the minimum element of the tableaux is in $T[0][0]$.

Lemma 2: If $T[m-1][n-1] < \infty$, then the tableaux is full.
Proof: If $T[m-1][n-1] < \infty$, then $T[i][j] \leq T[m-1][j] \leq T[m-1][n-1] \leq \infty$. This implies that all elements have a value less than ∞ and hence the tableaux is full.

Removing minimum element

As proved above, the minimum element is found in $T[0][0]$ and can be removed from there. It is replaced by ∞ and therefore the tableaux temporarily loses its property. This is restored using a function similar to $Heapify$.
c
Lemma: The $RESTORE$ function restores the property of an m x n tableaux.

Base Case: Considering a 1 x 1 tableaux. On removing the min element, $T[0][0] = \infty$. Since there is only a single element, tableaux property holds on passing through the $RESTORE$ function.

Induction Hypothesis: Considering that the procedure works for an m x $n-1$ tableaux, it will restore the m x n matrix.
Let us consider a m x n tableaux. Let us consider that the minimum of the neighbours(right and down) for the given element is in the adjacent column. Then we swap the two entries and the size of the matrix to be restored becomes m x $n-1$. According to our induction hypothesis, the $RESTORE$ function, restores the tableaux property for an m x n tableaux. Therefore, the hypothesis holds.

Inserting new element

The new element is inserted in the last position of the tableaux. Now as proved earlier, a $RESTORE-INSERT$ procedure is used which is similar to the $RESTORE$ procedure. The $RESTORE-INSERT$ procedure works similar to the $RESTORE$ procedure except that it percolates the element up and left rather than right and down untill it satisfies the tableaux property. The proof of correctness is same as that for $RESTORE$ procedure except that instead of the minimum of the neighbours, the max of the up and left neighbours is swapped with the current element.

Searching of an element

Lemma: The $SEARCH$ function returns true if the given element is found in the m x n tableaux, otherwise return false.

Base Case: Considering a 1 x 1 matrix, if the key is present at $T[0][0]$, then the procedure would return true else the indices would be out of bounds and procedure would return. So it holds for the base case.

Induction Hypothesis: Let us assume that the $SEARCH$ procedure returns the desired result for a tableaux of size $m-1$ x n and m x $n-1$. Now we check the value of the element at the position $T[0][0$. If it is equal, we are done. Otherwise it recursively iterates into a sub-tableaux which according to our induction hypothesis would output the correct result. In case the counter indices are out of bounds, the procedure returns false.

Time Complexity

Let us consider the $RESTORE$, $RESTORE-INSERT$ and the $SEARCH$ function. The three functions perform some fixed step operations before recursively calling each other such that either the number of rows or the number of columns is decremented by 1.

Considering the $EXTRACT-MIN$ procedure
The removal of the minimum element takes $O(1)$ steps. Depending on which element is the minimum among

the two neighbours(right and down) of the element, either a row or column index is incremented. The number of rows+columns is decremented by 1 each time.
$k = m + n$
$T(k) = O(1) + T(k-1)$
Therefore, time complexity is $O(m + n)$.

Similar is the case for the $INSERT$ procedure where insertion of element takes $O(1)$ steps and there is a slight modification in the $RESTORE - INSERT$ procedure which percolates up. However, the number of rows+columns is decremented by one each time. The recurrence remains the same.
$T(k) = O(1) + T(k-1)$
Time Complexity: $O(m + n)$
Similar to the $RESTORE$ procedure, the $SEARCH$ procedure also has a time complexity of $O(m + n)$.

Question 2

Pseudocode

Algorithm 4 Finding Connected Components and Operating on them individually

```
$count = 0$
$digits\_count = [0] * 10$                                   ▷ An array of 10 '0'

procedure DFS($i, connected\_components$)        ▷ where $i$ is the index of the vertex we are on
    $visited[i] = 1$

    $connected\_components[count] = i$
    $count = count + 1$

    $digits\_count[digits[i]] = digits\_count[digits[i]] + 1$

    for $j$ in NEIGHBOURS($i$) do
        if $!visited[j]$ then
            DFS($j, connected\_components$)
        end if
    end for
end procedure


procedure MAX_NUMBER($n, digits$)
    for $i$ in [0, 1 ... n-1] do
        $connected\_components = [-1] * n$
        DFS($i, connected\_components$)

        SORT($connected\_components, count$)
                                        ▷ To sort elements in a connected component
        $index = 9$
        for $j$ in connected_components do
            while $!digits\_count[index]$ do
                $index = index - 1$
            end while

            $digits[j] = index$
            $digits\_count[index] = digits\_count[index] - 1$
        end for
    end for
end procedure
```

Proof of Correctness

We try to visualize the digits as nodes, and allowed swaps between indices as edges.

Claim: All digits (nodes) in a connected component are inter-changeable.

Proof: Since the allowed swaps are represented as edges an by definition of a connected component, there is a path from a vertex of the connected component to another. If we define moving from one vertex to another as swapping the entries of the nodes, moving along a path would mean swapping the two end points. In a connected component, there is a path from any vertex to another implying that all digits(nodes) in a connected component are interchangeable.

For every digit(node) i, we compute the connected component for the digit using DFS traversal. The idea is to compute the connected component for each node while mantaining the count of each digit in the connected component.

It is clear that the maximum number obtained will be when the highest digit value will be in the smallest indices (MSD). Since we have the connected component for one DFS traversal, i.e. the indices of the digits, we fill the largest digits in the smallest indices first. This is achieved by sorting the indices and filling the maximum digit value first followed by smaller values.

Time Complexity

In the algorithm, for each digit(node) in the graph, we do a DFS traversal followed by sorting the connected component and some operations on the connected component.

The maximum size of a connected component will be n which is the total number of digits(nodes). The sort function to sort the indices in a connected component would therefore take $O(nlogn)$ time (using some sort function like MergeSort or HeapSort). Since the loop is traversing through all nodes (although might not be calling the DFS traversak), there is an additional $O(n)$ time.

Other than this, the DFS traversal for all connected components takes $O(m)$. Hence, the total time will be $O(m + n + nlogn)$ or $O(m + nlogn)$

Question 3

Pseudocode

Algorithm 5 Finding Maximum Path Length

```
$max\_path\_length = 0$                ▷ returns the value of the maximum path length in the graph
procedure MAX_PATH($i$)                ▷ where $i$ is the index of the vertex we are on
    $visited[i] = 1$

    $first\_max = 0, second\_max = 0$

    for $j$ in NEIGHBOURS($i$) do
        if $!visited[j]$ then
            $t =$MAX_PATH(j)
            if $t > first\_max$ then
                $first\_max = t$
            else if $t > second\_max$ then
                $second\_max = t$
            end if
        end if
    end for

    $max\_path\_length = max(max\_path\_length, first\_max + second\_max + 1);$

    return $first\_max + 1$
end procedure
```

Proof of Correctness

The max_path function is essentially the DFS function with a few modifications. So the concepts and explanations are pretty similar to that of DFS algorithm. Also, the similarities justify that the algorithm will cover all vertices for a connected component, and since the given graph is a tree (connected), therefore max_path (DFS) will cover all the vertices.

The algorithm, though not directly, finds out a path for which the maximum length is obtained. In this path, some vertices are used, while some are not. Although we do not directly know which vertex is being used, but we can still say that there are two cases, that the vertex we have currently (in one of the recursive steps) called on is being used in the maximal path, or it is not. The concept we use to find the maximal path is taking precisely these two cases for each vertex.

To achieve this, we find out the lengths of maximal paths using the current vertex's neighbours and adding one to it. For the other case, we just call DFS on the neighbours, and compare their maximal paths lengths with the existing maximal path length found.

Claim: The maximal path length using the current vertex and ignoring the ancestral vertices of the DFS tree is given by $first\_max + second\_max + 1$

Proof: There are 3 cases for this, based on the degree of this vertex -

Case 1: The vertex has 'no' visitable neighbour

If the vertex has only one neighbour, then it is clearly the leaf of the DFS tree, and hence it's length will be one. In this case, the vertex will have no neighbour other than it's ancestor, which is already visited. Hence the variables $first\_max = 0$ as well as $second\_max = 0$. Hence, $first\_max + second\_max + 1 = 1$ which should be true for this case.

Case 2: The vertex has 'one' visitable neighbour
In this case, the maximal path with this vertex is clearly the maximal path length with it's neighbour + 1. Since there is only one neighbour with the maximal path length > 1, only $first_max$will have a value > 1, whereas $second\_max = 0$. Hence, $first\_max + second\_max + 1 = first\_max + 1$ which should be true for this case.

Case 3: The vertex has 'more than one' visitable neighbours
Let us say that the maximal path including each neighbour respectively gives lengths $l_1, l_2 ... l_k$ for some k, then the length of the path we take (including two neighbours) will be $l_{i1} + l_{i2} + 1$. This is clearly maximum when l_{i1} and l_{i2} are the biggest among other lengths. $first\_max$ and $second\_max$ specifically are for this purpose. Hence $first\_max + second\_max + 1$ will give the maximum path length including this vertex.

Hence our previous claim is true. Using this, we can say that if we compare the maximal path length found up until now with the maximal path lengths obtained for every vertex (once including and once not including), we can say that at the end of the algorithm, we will have found the maximal path length.

By visitable, we mean a vertex which has not already been visited *
Case 1 - when we have a leaf vertex of our DFS tree **
Case 2 - when we have any internal two degree vertex or one degree root vertex of our DFS tree ***
Any other vertex will belong to Case 3 ****

Time Complexity

The given graph is undirected, connected and acyclic. By definition, the graph is a tree.

Lemma: A tree has $n - 1$ edges.
Proof (By induction):

Base Case:A tree with one vertex will have no edges. The base case is trivial.

Induction Hypothesis: Assume that the result holds for the case where $|V| < n$. Removing an edge from the tree would disconnect the graph as it is acyclic. The two components are connected and acyclic and therefore trees. T_1 has $n1$ and T_2 has $n2$ vertices. Since the result holds for values less than n, T_1 and T_2 have $n1 - 1$ and $n2 - 1$ edges respectively. If we now add the removed edge, the total number of edges would be $n1 + n2 - 1$. However, we know that $n1 + n2 = n$. Hence, number of edges in T is $n - 1$.

Our algorithm as explained in the proof is a modification of the DFS traversal algorithm and takes only O(1) extra steps per recursive call. It would therefore have a time complexity of $O(E + V)$. However, as proved above the number of edges is $V - 1$. Therefore, time complexity of the algorithm is $O(V)$.

Question 4

Pseudocode

Algorithm 6 To find if a directed graph is a Pseudo-Tree

```
$connected_components = newStack()$                          ▷ Create an Empty Stack

procedure IsRooted($i$)
    $visited[i] = 1$
    $connected\_components.pushback(i)$;

    for $j$ in Neighbours($i$) do
        if $visited[j]$ then
            return false
        else
            if !IsRooted(j) then
                return false
            end if
        end if
    end for

    $rooted[i] = 1$
    return true
end procedure


procedure IsPseudoTree($G$)
    for $i$ in Vertices($G$) do
        if !$rooted[i]$ then
            if !IsRooted(i) then
                return false
            end if

            for !$connected\_components.isEmpty()$ do
                $visited[connected\_components.top()] = 0$
                $connected\_components.pop()$
            end for
        end if
    end for
    return true
end procedure
```

Proof of Correctness

By the definition of a rooted tree, we can have no cycles and no cross-edges in the DFS tree of the subgraph reachable from a vertex. Hence for the graph to be a pseudo-tree, all DFS trees obtained from different subgraphs shall hold the above property.

isRooted Function

This function is quite similar to the DFS function and uses the same concepts. In fact, we are directly applying DFS to recursively check for the subgraph to be a rooted tree.

Claim: If any reachable subgraph of a vertex u in the reachable graph from vertex v does not form a rooted tree, then the reachable subgraph of v does not form a rooted tree.

Proof: This is pretty trivial. If u is in the reachable subgraph of v, then any vertex reachable from u is also reachable from v. Therefore any cycle or cross-edge in the subgraph of u will also be in the subgraph of v.

Claim: If any vertex is reached for the second time, the subgraph it belongs to is not a rooted tree.

Proof: Since we are re-initializing the visited status of every vertex once we complete the whole subgraph, if any vertex is reached again, then it will be from the same subgraph only. Hence, it will form a cycle. This is checked using that vertex's visited status.

The above two checks should be done for all the possible paths in a subgraph, and hence our loop through the neighbours of a vertex in the *isRooted* function is justified.

We also keep a track whether the subgraph obtained from one vertex is rooted or not using the *rooted* list.

isPseudoTree

Our task now is easy, we just need to check each vertex if the subgraph reachable from that vertex is a rooted tree or not. We can achieve this using the isRooted function, however, we do not need to recheck the already checked vertices (checked during the DFS call of some other vertex), so we use the rooted list to avoid repetitions.

Since any vertex that has been visited can be revisited from another vertex without actually forming a cycle or a cross-edge, therefore we need to reinitialize the vertices. This is justified from the while loop over the elements of the *connected_components* stack.

Time Complexity

Since the *isRooted* is just like DFS, we say that the total complexity will be the sum of the in-degrees of the vertices in a connected component. Since this is bounded by $O(E)$, hence *isRooted* is bounded by $O(E)$.

The inner loop to re-initialize the *connected_components* stack is clearly $O(V)$

We are calling the *isRooted* function for every connected component, along with reinitializing the vertices. Hence the loop in the $isPseudoTree$ function will have V iterations. Although the number of times it actually calls *isRooted* function or re-initializes *connected_components* will be less than V, but in the worst case, it is bounded by $O(V)$ only.

Therefore, total time complexity is $O(V(E+V))$