ESO207A

Theoretical Assignment 3

DEEPANSHU BANSAL (150219)
MUKUL CHATURVEDI (150430)

Q1

Pseudocode

```
EXTRACT-MIN(Y)
    x←Y [1, 1]
    Y [1, 1]←∞
    STABILISE(Y, 1, 1)
    return x


STABILISE(Y, i, j)
    Smallest i←i
    Smallest j←j
    if i + 1 ≤ m and Y [i, j] > Y [i + 1, j]
    then smallest i←i + 1 smallest j←j
    if j + 1 ≤ n and Y smallest i ,smallest j ] > Y [i, j + 1]
    then smallest i←i smallest j←j + 1
    if smallest i  !=i  or smallest j  != j
     then exchange Y [i, j] ↔ Y [smallest i, smallest j ]
    STABILISE(Y, smallest i , smallest j )
```

```
INSERT(Y, k)
    DECREASE-KEY(Y, m, n, k)
DECREASE-KEY(Y, i, j, k)
     if Y [i, j] ≤ k
          then return error
     Y [i, j]⟵k
     threshold⟵∞
    largest_i⟵i
    largest_j⟵j
    while(i > 1 or j > 1) and Y [i, j] < threshold
          do exchange Y [i, j] ↔ Y [largest_i , largest_j ]
           i⟵largest_i
          j⟵largest_j
          if i − 1 ≥ 1 and Y [i, j] < Y [i − 1, j]
           then largest_i⟵i − 1 largest_j⟵j
          if j − 1 ≥ 1 and Y [largest_i , largest_j] < Y [i, j − 1]
          then largest_i⟵i
          largest_j⟵j − 1
          threshold⟵Y [largest_i, largest_j ]


Search(Y,k)
i⟵1 and j⟵m
while(Y[i,j] != k and i<n+1 and j>0)
    If Y[i,j]<k  Then i⟵i+1
    If Y[i,j]>k  Then j⟵j−1
Return i , j
```

Proof of Correctness

For EXTRACT-MIN function

Consider Y [i, j] for any i and j. By the property of a Young tableau, Y [1, 1] ≤ Y [i, 1] ≤ Y [i, j]. Therefore, Y [1, 1] is the minimum because the above is true for every i and j. EXTRACT-MIN function returns Y[1,1] and makes the Young Tableau according to the rules by calling the function STABILISE.

We should note that in every Young tableau for any index (i,j) , Y [i + 1..m, j..n] is a Young tableau and Y [i..m, j + 1..n] is a Young tableau. In STABILISE function we compare current index , right index and down index of Young Tableau and the smallest one in them takes the position of current.If current index is smallest then we come out of the function as Y [i + 1..m, j..n] and Y [i..m, j + 1..n] are already Young tableau . If right index is smaller then we exchange current and right now don't know if Y [i..m, j+1..n] is proper Young tableau so we call STABILISE in that part. Similarly we do when down index is smallest thus in the end if (i,j) reach (n,m) we stop thus giving us valid Young tableau.

For Insert function

First we insert number to be inserted (let k) at Y[n,m] if Y[n,m] = ∞ otherwise error. Now we call DECREASE-KEY function.

We should note that in every Young tableau for any index (i,j) , Y [1..i-1, 1..j] is a Young tableau and Y [1..i, 1..j-1] is a Young tableau. In DECREASE-KEY function we compare current index , left index and up index of Young Tableau and the largest one in them takes the position of current.If current index is largest then we come out of the function as Y [1..i-1, 1..j] and Y [1..i, 1..j-1] are already Young tableau . If right index is smaller then we exchange current and right now don't know if Y [1..i-1, 1..j] is proper Young tableau so we call DECREASE-KEY in that part. Similarly we do when up index is largest thus in the end if (i,j) reach (n,m) we stop thus giving us valid Young tableau.

For Search function

For this function we start from i=1 and j=m. Now if Y[i,j] = k (number to be searched) then we return from function. Else if Y[i,j] < k then i = i+1 because Y[i+1,j] is greater than Y[i,j] so we now again iterate. Else if Y[i,j] > k then j = j+1 because Y[i,j-1] is smaller than Y[i,j] so we now again iterate till we get Y[i,j] = k or get i = n and j=1.

Time Complexity

For EXTRACT-MIN function

Obviously, the running time of STABILISE is O(m + n) because STABILISE exchanges Y [i, j] with either Y [i + 1, j] or Y [i, j + 1] and recursively STABILISE the tableau at the corresponding entry. Therefore, with every recursive step, i + j is incremented by 1. However, i + j cannot be more than m + n and this gives the time bound of O(m + n). More precisely, we can say that STABILISE a tableau of size m × n will recursively YOUNGIFY a tableau of size (m − 1) × n or of size m × (n − 1).

Therefore, if we let p = m + n to be the size of our problem: T(p) = O(1) + T(p − 1) The solution of the above recurrence is T(p) = O(p) = ***O(m + n)****.*

For Insert function

In one iteration of while loop function takes constant number of operation and i or j decreases by 1. Sum of i and j at the starting is m+n but by the end , it's more than 2 so we take atmost m+n iterations of loop . Thus, time complexity of while loop is O(m+n).

*T(m,n)=c*O(m+n), where c is a constant.*

Thus, time complexity of while loop is ***O(m+n)****.*

For Search function

In one iteration of while loop function takes constant number of operation and i increases by 1 or j decreases by 1. In the while loop i can increase maximum n times and j can be decreased by m times so while loop can iterate maximum m+n times and there are constant number of steps in each iteration.Thus, time complexity of while loop is O(m+n).

*T(m,n)=c*O(m+n), where c is a constant.*

Thus, time complexity of while loop is ***O(m+n).***

Q2

Pseudocode

```
BFS(G,s,data[],index[])
CreateEmptyQueue(Q);
Enqueue(Q,s); dist(s) ← 0 ; Visited(s) ← true ;
data[dist(s)] ← s.value;
index[dist(s)] ← s.label;
while(Q is notEmpty())
        x ← Dequeue(Q) ;
        for each neighbour v of x
                if(Dist(v)= ∞ ) then
                        Dist(v) ← Dist(x) +1 ;
                        data[dist[v]] ← v.value;
                        index[dist[v]] ← v.label;
                        Visited(v) ← true
                        Enqueue(Q,v) ;
Largest_number(N)
{
→ Take input as a string N a  n-digit  number;
→ N[0] is most significant  and N[n-1] is least significant ;
→ Create a Graph G with n vertices labeled from 1 to n ;
→ Each node of graph has two elements one as label and one value.
→ Value will store the digit at that position in number and label is the position of digit in number.

for each of m-pairs (i,j)
        addEdge(G,i,j);
BFS_TRAVERSAL(G)
{
for each vertex  x
        Visited(x) ← false;
        Dist(x) ← ∞ ;
for each vertex v of V
        if(Visited(v) = false ) then
                Initialize two empty arrays data[] and index[] ;
                BFS(G,v,data[],index[]) ;
                → Sort data and index array in descending order;
                → Mergesort(data[])  in descending order;
                → Mergesort(index[]) in ascending order;
                i ← 0 ;
                while(not traversed through whole data array)
                        N[index[i]] ← data[i] ;
                        i++ ;
}
return N ;
}
```

Proof of Correctness

The Largest_number(N) function gives us the correct output as first of all the G has only those as connected components which can be swapped as once the program enters into BFS(G,v) inside BFS_TRAVERSAL(G) it will traverse all the nodes which are connected (Proof discussed in class of BFS) ie. all positions in numbers which can be swapped. Thus since it covers all its positions (Proof as BFS visits all the reachable nodes) thus we gets the labels as well as the digits of these positions in two arrays. Now once we get arrays we sort data array in descending order and index array in ascending order thus place the highest digit number in lowest position as lower positions are more significant. This will hold for all connected components and at last we get correct result.

Time Complexity

Here |V| = n , |E| = m, hence

T(BFS) = O(n+m)

And we can have at most n nodes in connected component hence

T(Mergesort) = O(nlogn)

*T(BFS_TRAVERSAL) = T(BFS) + 2*T(Mergesort) + c*

T(BFS_TRAVERSAL) = O(n+m) + O(nlogn) + c

T(Largest_number) = T(BFS_TRAVERSAL) +d

T(Largest_number) = O(n+m) + O(nlogn) + f

T(Largest_number) = O(m+nlogn)

Hence time complexity of algorithm is **O(m + nlogn).**

Q3

Pseudocode

```
pair<int,int>BFS(G,s)
CreateEmptyQueue(Q);
Enqueue(Q,s); Dist(s) ← 0 ; Visited(s) ← true ;
pair<int,int > max;
while(Q is notEmpty())
    x ← Dequeue(Q) ;
    for each neighbour v of x
        if(Dist(v)= ∞ ) then
            Dist(v) ← Dist(x) +1 ;
            Visited(v) ← true;
            Enqueue(Q,v) ;
            if(Dist(v)>max) then
                max.first ← Dist(v);
                max.second ← v;
Return max;
```

```
Max_Dist(G)
→ lets say s is any node and initialize required variables
→ (First_Node,dist1) ← BFS(G,s);
→ (Second_Node,result) ← BFS(G,First_Node);
→ return result;
```

Proof of Correctness

A acyclic,undirected and connected graph is a tree. The largest distance between two nodes of tree are those indices which are two most farthest from root of tree(selected randomly) . We first select any node s in G and apply BFS in G with s as root. Now,let say we get v as the node which is farthest from root . Now this node v must be one of the end point of the longest path (max distance) as it isn't then if any other node is let's say w then dist(v) >= dist(w) +1 which is clearly a contradiction. Now we have to find the second farthest node x from root .

We apply BFS from v and the farthest from v in the BFS is actually node x . Thus we get to the farthest distance this way. Lets say u be the farthest common ancestor of v and x from the root.

Now during BFS(v) we must encounter u and since we are finding max distance it will divert its path from root towards the node x. Thus this ensures that we get correct result.

Time Complexity

*T(Max_Dist) = 2*T(BFS) + c*

T(BFS) = O(|V|+|E|) .

Now as this graph is an undirected, connected, acyclic graph which means a tree and for a tree

We have |E|=|V-1|

Thus, T(BFS) = O(2|V|-1) =* ***O(|V|)****.*

*T(Max_Dist) = 2*T(BFS fn.)+c = O(|V|).*

Thus time complexity of algorithm is *O(|V|)* .

Q4

Pseudocode

```
int DFS(v)
Visited(s) ← true ; DFN[v] ← dfn++ ; flag ← 0;
for each neighbour w of v
    if(Visited(w) = false ) then
        Parent(w) ← v ;
        DFS(w);
    Else if(Parent(v) = w) then
        flag = 1;
        return flag;

Pseudo_tree(G) {
for each vertex  x
    Visited(x) ← false;
dfn ← 0 ;
for each vertex v of V
    if(Visited(v) = false ) then
        result ← DFS(v) ;
    → if(result=1 ) then tree is pseudo_tree otherwise not
}
```

Proof of Correctness

If for all u ∈ V , the subgraph induced by the set of vertices reachable from u, forms a rooted tree that means that there is no cycle in the tree. As for a tree |E| = |V| - 1 and if tree is not rooted then it will violate this condition and hence it must have a cycle if graph is not a pseudo tree.

Hence to show whether a directed graph G = (V, E) is pseudo-tree or not can be decided if it has a cycle then it is not a pseudo-tree. Otherwise it will be a pseudo-tree.

Now in DFS function if the program enters the else part of the for loop that would mean that it is going to visit a vertex which has already been visited and also not a parent of vertex from where it is coming that means there is a back edge from this vertex which means a cycle hence tree is not a pseudo tree.

Time Complexity

Here time complexity is only of DFS-Traversal which is **O(|*V*| + |*E*|*)*** *as one vertex is visited only one time as also discussed in the class.*

Hence time complexity of algorithm is **O(|V|+|E|).**