Theoretical Assignment 3

Pragati Agrawal (220779)

October 2023

Pragati Agrawal: 220779 (Question 1)

We want to find an eulerian cycle in the graph G.

(a) No, we cannot find such a path using a given DFS traversal of the graph. This is because in DFS traversal we create a DFS tree. So in the traversal, we do not cover those edges, whose vertex has already been marked visited due to some other edge. Therefore, the back edges are left out in the DFS traversal, but for an eulerian cycle, we want to cover every edge.

(b) No, we cannot find such a path using a given BFS traversal of the graph. This is because in BFS traversal of G will create a BFS tree. Since in a tree, we do not include the cycle-creating non-tree edges, if G has an eulerian cycle, then some edges must be left out in the BFS tree. Therefore, using a BFS traversal, we cannot cover all the edges.

(c) The condition required for such a path to exist is that the graph must be connected and that the degree of every vertex in the graph must be even. This is necessary to be able to return to the starting point, and ensure that no edge would be covered twice.

(d) If the graph has an eulerian cycle, we will keep pushing its vertices in a stack, and delete that edge, until we reach some vertex having degree=0. After this we will pop out the stack until we reach some vertex whose degree is non-zero. Then we run the same algorithm again. The algorithm is as :

- Start from any vertex v of $G = (V, E)$. Create an empty stack S.
- Push v in the stack.
- Use a loop that terminates when stack becomes empty.
- Make a temporary variable x that stores the value currently at the top of the stack.
- If $degree(x) \neq 0$, we take some edge $e = (x, y)$ in the graph, remove this edge, and push the vertex y into the stack. We can use hash maps to remove the edge e in constant time.
- If $degree(x)$ is 0, this means x has no more edges to go to. At this stage, we will pop the stack until we reach some vertex whose degree $\neq 0$. The values being popped, will be stored in a list/printed (in the same order) as the answer.

The pseudo-code for the algorithm is given below:

```
FindDegree(G){
    deg[n+1]={0};
    for(i=0;i<m;i++){
        a=edge[i].first;
        b=edge[i].second;
        deg[a]++;
        deg[b]++;
    }
}

FindEulerianCycle(G){
    CreateEmptyStack S;
    List tour;
    Push (v,S);
    while(Not isEmpty(S)){
        int x=Top(S);
        if(degree(x) != 0){
            int y=neighbour(x);
            delete edges (x,y) and (y,x); //using hash table
            deg[x]--;
            deg[y]--;
            push(y,S);
        }
        else{
            Insert(x,tour);  //or print it directly to stdout.
            pop(S);
        }
    }
    return List;
}
```

Time Complexity Analysis:
The first function $FindDegree()$ clearly works in $O(m)$ time.
Now, for the second function $FindEulerianCycle()$, in each iteration of the loop, we are deleting edges from the graph, until we have no more edges in that cycle, and then we print it. Now, if some other connected cycle is left to be traversed, that connecting vertex will start the tour in that cycle, since its degree won't be 0. The stack has no more elements to be pushed when all edges are deleted from the graph. Then we pop out all the remaining elements of the stack.
Since each edge can be deleted only once, the algorithm is bound to terminate push operations after $O(m)$ steps, where m is the number of edges in the graph. Now, since we will cover each edge exactly once, but the vertices can be visited multiple times, in the worst case, the stack will have $m + 1$ values in it. This is because after the starting vertex, each edge corresponds to one more vertex. Hence, the pop operations, can take $O(m)$ time in the worst case. So the function will have maximum $2m + 1$ iterations.
Therefore, the overall time complexity of the algorithm is $O(m) + O(m) = O(m)$.

Pragati Agrawal: 220779 (Question 2)

(a) We want to destroy the destination city D from the source city S, through a series of towers, each of power x. For this, we will start a BFS traversal from S, and go enqueue all towers within the range x in some queue since all these will be activated from S. We will also keep another array called dest[ ] for all the cities, and we would update it if the city gets destroyed. So, we will dequeue one tower at a time, perform BFS and again enqueue the remaining towers in the queue. We will continue this until either we at some point enqueue D, or the queue becomes empty. If dest[D]=1 after the check ends, then we could destroy D from S.
The complete **algorithm** is given below:

- Create an empty array dest[ ] initialised with 0.
- Create an empty queue Q. Enqueue S in it.
- Create a loop that ends either if Q becomes empty.
- Create a variable v containing the top of the queue, $v = Front(Q)$. Dequeue Q.
- Initialise the distances of all nodes to be infinity , and visited to be 0. Start BFS from v, and enqueue all towers which lie at a distance $\leq x$ from v into Q. Destroy all cities that come in the path.

The **pseudo code** for the modified BFS algorithm is as:

```
new_BFS(G,v,x,Q,Distance,dest){
    CreateEmptyQueue q;  //queue for BFS
    int Visited[n]={0};
    Distance[v]=0;
    Visited[v]=1;
    Enqueue(v,q);

    while(Not IsEmptyQueue(q)){
        p=Front(q);
        Dequeue(q);
        for (each neighbour w of p){
            if(Visited(w)==0){
                Distance(w)=Distance(p)+1;
                Visited[w]=1;
                Enqueue(w,q);
                if(Distance[w]<=x and dest[w]==0){
                    if(w is a tower){
                        Enqueue(w,Q);
                    }
                    dest[w]=1;
                }
            }
        }
    }
}
```

The **pseudo code** for the check algorithm is as:

```
int Check_power(graph G, source S, destination D, power x){
     int Distance[n]={0};
     int dest[n]={0};
     CreateEmptyQueue(Q);   //The queue of towers
     Enqeue (S,Q);
     int flag=0;
     while(Not isEmpty(Q)){
         int v=Front(Q);
         if(v==D) {
             flag=1;
             break;
         }
         Dequeue(Q);
         new_BFS(G,v,x,Q,Distance,dest);   //for each tower in Q
     }
     if (dest[D]==1){
         flag=1;
     }
     return flag;
 }
```

Time Complexity Analysis:
For each city that can be destroyed, we will update the dest[ ] array. So if at any point D is within the radius x of any tower, then the function changes its value at that infex, returns true, otherwise false. In the while loop, we perform one dequeue operation in each iteration. Each tower can enqueue at most once in the queue Q. If the graph has $O(n)$ towers, then for each tower, it will take $O(m+n)$ time for the BFS. Hence time complexity of the algorithm will be $O(towers*(m+n)) \leq O(n*(m+n))$.

(b) Let the graph has n vertices and m edges. We know that the maximum possible distance of D from S can be $n-1$, when S and D are linearly apart by all other vertices.
So, if $x = 0$, we know that a signal cannot be sent. And if $x = n-1$, surely a signal will be sent from S to D directly, no matter how the graph is. Now, to find the minimum value of x, we will apply something similar to binary search. This will work correctly because of the correctness of binary search algorithm.
The **algorithm** is given below:

- Make three variables: left, right and mid.
- Initialise left=0, right=n-1.
- In a loop, assign mid to be the average of left and right (assuming integer division).
- Now, use the above *Check_power* function to check if $x = mid$ works or not.
- If it does, update right to be mid-1. Otherwise update left to be mid+1.
- Keep doing until left¡right.
- The value stored in mid will be the minimum value of x.

The **pseudo-code** for the above is given below:

```
int Find_minimum_power(){
    int left, right, mid;
    left=0;
    right=n-1;
    while(l<=r){
        mid=(left+right)/2;
        int temp=Check_power(G,S,D,mid);
        if(temp==1){
            right=mid-1;
        }
        else{
            left=mid+1;
        }
    }
    return mid;
}
```

Time Complexity Analysis: Since binary search works in $O(logn)$ time, the time taken by this algorithm will be $O(logn * (n) * (n + m))$.

Pragati Agrawal: 220779 (Question 3)

We can solve the problem using complete binary trees. Given n rooms which are bombed with colours, we will construct a complete binary tree in $O(n)$ time that will have the last n nodes as the initial colours $\{c_1, c_2, c_3, ..., c_n\}$. WLOG, let $c_i \in \{1, 2, 3, ..., n\}$ of the rooms, and the internal nodes changed colour of the subtree rooted at that node due to a bombing.
The complete **algorithm for building the data structure** is given below:

- If n is not a power of 2, take the next highest power of 2, i.e. $n \leq 2^p$, then let $t = 2^p$. Now create an array of size $s = (2t - 1)$. Let the array be arr.
- Populate the array with the initial colours of the rooms starting from the index $arr[s-t]$, upto $arr[s - t + n - 1]$. If $n \neq t$, populate the remaining last elements, i.e. from $arr[s - t + n]$ to $arr[s - 1]$ with the dummy value 0.
- Now, populate all the internal nodes of the tree with the dummy value 0 initially.

The **pseudo code** for creating the data structure is given below:

```
    //n is the number of rooms
    //rooms[n] is an array containing the initial colours of the rooms
    /pseudo-code to create the data structure
CreateDataStr(n,rooms[]){
    int t=1;
    while(t<n){
        t=t*2;
    }
    int s=2*t-1;
    int arr[s];
    int k=1;
    for(int i=s-t;i<s-t+n;i++){
        arr[i]=rooms[k++];
    }
    for(int i=s-t+n;i<s;i++){
        arr[i]=0;
    }
    for(int i=s-t-1;i>=0;i--){
        arr[i]=0;
    }
}
int pref[s]={0};
```

Also, we will create one more array: $pref[]$ of size s, that will store the chronological order of that bombing. Initially the array is given the dummy value 0, denoting no bombing. Now, after creating the complete binary tree structure, we will perform the bombings. Let the m bombings be indexed from 1 to m.
To update the colour of any room, the algorithm is similar to the multi-increment problem as taught to us in lectures:

- For any bombing (l, r, c), let its index be a. So a starts from 1 and the colour for this bombing is c.

- Now, make two pointers, maybe the left pointer will be i and the right pointer be j.
- $i = s - t + l - 1$ and $j = s - t + r - 1$.
- Change $arr[i]$ to c and $pref[i] = a$.
- If $(j > i)$ change $arr[j]$ to c, and let $pref[j] = a$.
- While the parents of i and j are not the same,
  - if i is the left child of its parent, i.e. its index is odd, change $arr[i+1]$ to c and $pref[i+1] = a$.
  - if j is the right child of its parent, i.e. its index is even, change $arr[j-1]$ to c and $pref[j-1] = a$.
  - update i and j to be their parents.
- increment a and move to the next bombing.

The **pseudo-code** for the above algorithm is:

```
    Update_colours(l,r,c,a){
        int i=s-t+l-1;
        int j=s-t+r-1;
        arr[i]=c;
        pref[i]=a;
        if(j>i){
            arr[j]=c;
            pref[j]=a;
            while((i-1)/2 != (j-1)/2){
                if(i%2==1) {arr[i+1]=c; pref[i+1]=a;}
                if(j%2==0) {arr[j-1]=c; pref[j-1]=a;}
                i=(i-1)/2;
                j=(j-1)/2;
            }
        }
        a++;
    }
int a=1;
for(m in bombings){
    Update_colours(m.left,m.right,m.color,a);
    a++;
}
```

Time Complexity Analysis of the above algorithm:
This entire function $Update\_colours()$ works in $O(logn)$ time, as in each iteration of the while loop, we are moving one level up, and there are at most $O(logn)$ levels since we have a complete binary tree, which has height $h = O(logn)$. Rest are constant operations. Now, for m bombings, it would take $O(mlogn)$ time.
Now, to find the final colour of each room, we will use an algorithm similar to $report(i)$ algorithm as taught to us in lectures. The **algorithm** is as:

- We create an empty array $final\_col[]$.
- For each of the n leaf nodes, we have some colour corresponding to it in the array arr, and some $pref$. Now, we will move from this node to its parent, all the way up to the root. In the path, if at any node, the preference of that node is greater than the preference of the previous node, we update the colour of the corresponding leaf node in the new array $final\_col$ at the correct index.

The **pseudo-code** for the above algorithm is:

```
Find_col(){
    int final_col[n+1]={0};
    for(int i=s-t,k=1;i<s-t+n;i++,k++){
        final_col[k]=arr[i];
        while(i>0){
            int parent=(i-1)/2;
            if(pref[i]<pref[parent]){
                final_col[k]=arr[parent];
            }
            i=(i-1)/2;
        }
    }
}
```

Time Complexity Analysis:
This algorithm takes $O(logn)$ time for each node, as in each iteration of the while loop, we update i to be its $parent$, i.e. we are moving one level up, and there are at most $O(logn)$ levels. Rest are constant operations. Now, for n rooms, it would take $O(nlogn)$ time.
Now, the array $final\_col$ gives the final colours of all the rooms, which can be printed from $\{1, 2, 3, ..., n\}$ in $O(n)$ time.
So, the total time complexity of the algorithm is:
$O(mlogn) + O(nlogn) + k * O(n) = O(mlogn + nlogn) = O((m + n)logn)$.

Pragati Agrawal: 220779 (Question 4)

We can solve the problem using complete binary trees. Given total n sweets from which requests are made, we will construct a complete binary tree in $O(n)$ time that will have the last n nodes as the prices of the sweets, and the internal nodes containing the sum of the prices of their children. The complete **algorithm for building the data structure** is given below:

- If n is not a power of 2, take the next highest power of 2, i.e. $n \leq 2^p$, then let $t = 2^p$. Now create an array of size $s = (2t - 1)$. Let the array be arr.
- Populate the array with the prices of the sweets starting from the index $arr[s - t]$, upto $arr[s - t + n - 1]$. If $n \neq t$, populate the remaining last elements, i.e. from $arr[s - t + n]$ to $arr[s - 1]$ with the dummy value 0.
- Now, populate the internal nodes of the tree with the sum of the values of their children. If the index of the child is x, then the index of its parent will be $\lfloor \frac{x-1}{2} \rfloor$. So, in another loop, start from the end of the array, and add values in pairs, and store them in their parent node. Do this upto the root node, i.e. upto $arr[0]$.

The **pseudo code** for creating the data structure is given below:

```
    //n is the number of sweets
    //sweets[n] is an array containing the prices of the sweets
    //pseudo code to create the data structure
CreateDataStr(n,sweets[]){
    int t=1;
    while(t<n){
        t=t*2;
    }
    int s=2*t-1;
    int arr[s];
    int k=1;
    for(int i=s-t;i<s-t+n;i++){
        arr[i]=sweets[k++];
    }
    for(int i=s-t+n;i<s;i++){
        arr[i]=0;
    }
    for(int i=s-1;i>0;i--){
        int parent = (i-1)/2;
        arr[parent]=arr[i]+arr[i-1];
        i--;
    }
}
```

Now, after creating the complete binary tree structure, we will perform the update and request queries.

To update the price of any sweet, the algorithm is:

- Let i be the index in the sweets[] of the sweet whose price has changed. So, its index the array arr[] would be $idx = s - t + i - 1$, where $i \in \{1, n\}$, i.e. if the price of the first sweet changes, $i = 1$.

- Let the new price be x, and the old price be y. So the change in its price $z = x - y$.
- Perform this change in price for that particular sweet, and then for all its ancestors, to maintain the tree structure.

The **pseudo-code** for the update operation is as:

```
//update operation
UpdateTree(i,x,arr){
    int idx=s-t+i-1;
    int y=arr[idx];
    int z=x-y;
    while(idx>=0){
        arr[idx]+=z;
        idx=(idx-1)/2;
    }
}
```

Time Complexity Analysis of the above algorithm:
This algorithm would take $O(logn)$ time for each update operation, as in each update, we move one level up in the complete binary tree, whose height is $h = O(logn)$. Therefore. for n update queries, it will take $O(nlogn)$ time.

Now, for the request queries, we will find the sum of the prices of the sweets from sweets[l] to sweets[r]. To do it in some finite steps, the **algorithm** is:

- Make two pointers, maybe the left pointer will be i and the right pointer be j.
- $i = s - t + l - 1$ and $j = s - t + r - 1$.
- Make a new variable $sum = 0$ initially.
- Add $arr[i]$ to sum.
- If $(j > i)$ add $arr[j]$ to sum.
- While the parents of i and j are not the same,
  - if i is the left child of its parent, i.e. its index is odd, add $sibling(i)$ to sum.
  - if j is the right child of its parent, i.e. its index is even, add $sibling(j)$ to sum.
  - update i and j to be their parents.

The **pseudo code** for the algorithm is:

```
    //sum stores the total cost of a request of the form (l,r), both included.
Find_cost(l,r){
    int i=s-t+l-1;
    int j=s-t+r-1;
    int sum=0;
    sum+=arr[i];
    if(j>i){
        sum+=arr[j];
```

```
        while((i-1)/2 != (j-1)/2){
            if(i%2==1) sum+=arr[i+1];
            if(j%2==0) sum+=arr[j-1];
            i=(i-1)/2;
            j=(j-1)/2;
        }
    }
    return sum;
}
```

Time Complexity Analysis of the above algorithm:
This algorithm would take $O(logn)$ time for each request, as in each iteration of the while loop, we are moving one level up, and there are atmost $O(logn)$ levels. Rest are constant operations.
Therefore the final pseudo code for the problem is:

```
    input n, sweets[];
    CreateDataStr(n,sweets[]);
    for( i=0;i<n;i++){
        input query q;

        if (q is (1,i,x)){
            UpdateTree(i,x);
        }
        if(q is (2,l,r)){
            int cost=Find_cost(l,r);
            if(cost>M) cout<<"NO"<<endl;
            else cout<<"YES"<<endl;
        }
    }
```

Complete Time Complexity Analysis:
The data structure is built in $O(n)$ time. Since both the $UpdateTree()$ and $Find\_cost()$ functions take $O(logn)$ time, for n queries, the algorithm would take $O(nlogn)$ time. Hence the time complexity of the algorithm is $O(nlogn)$.

Pragati Agrawal: 220779 (Question 5)

We are given a tree T having n nodes, and another sequence seq of n apples. Let seq be indexed from 1 to n. If the sequence seq is a valid BFS traversal of the tree T, then it must satisfy the following two conditions:

- For every i, $index(parent(i)) < index(i)$.
- If $index(i) < index(j)$, then $index(parent(i)) \leq index(parent(j))$.

To check for the first condition, we can apply BFS traversal of T, taking the first vertex, $seq[1]$ as the root vertex/starting vertex for the BFS traversal. This will create a *dist* and a *parent* array for the vertices. Now, if we create a distance array of the sequence from the above-created array *dist*, the values in the array must be non-decreasing, and if it increases, it must increase by unity. This is because, from that node, all its children at any given distance will always come together in the array, just their order can be different in different sequences. So we can check this new distance array to be non-decreasing or increase by 1 if seq is a valid BFS traversal.
The **algorithm** for this check is:

- Apply BFS traversal from the $seq[1]$ vertex. This will create the arrays parent[] and distance[]. As per clarification mail,
- Now, create a new empty array dist_seq[ ].
- For each value in the sequence, update its corresponding distance from the distance[ ] array into the array dist_seq[ ].
- Check if the array has an increment of atmost 1.

The **pseudo-code** for the algorithm is:

```
    //T is the given tree having n vertices ={1,2,3,....,n}
    //seq is the given sequence indexed from 1 to n.
    //We perform BFS from seq[1] vertex
    //two arrays parent[] and dist[] is created
 BFS(G,seq[1],parent[],dist[]);
    int flag=1;
    if (seq[1]!=1){
        flag=0;
        cout<<"INVALID";
    }
    int dist_seq[n+1];
    for(int i=1;i<=n;i++){
        dist_seq[i]=dist[seq[i]];
    }

    for(int i=1;i<n;i++){
        if(dist_seq[i+1]<dist_seq[i] or dist_seq[i+1]-dist_seq[i]>1){
            flag=0;
            cout<<"INVALID";
        }
    }
```

Time Complexity Analysis for the above algorithm: Since the above check involves a single traversal of the array *dist_seq* of size n, therefore, it takes $O(n)$ time to check.

Now, if $flag$ is 1 after the second for loop, this means that the parent of any node is before its appearance in the sequence. Now, to check the second condition, we will use the *parent* array. We will create a new array *parent_seq*[] containing the parents of the elements of the array. Now, the parent_seq must be a subsequence of the given sequence, in order. The parent of the root, i.e. $Parent(seq[1]) = seq[1]$
If $i < j$ and $parent[i] \neq parent[j]$, then if $parent[i] = seq[a]$ and $parnet[j] = seq[b]$, then $a < b$.
The **algorithm** for the above condition is:

- Consider two pointers: x in *seq* and y in *parent_seq*, and initialise them both to 1.
- Now, to check for all elements in *parent_seq*, use a loop with terminating condition when pointer y reaches the end of the array, i.e. $y \leq n$.
- Also put the condition that if the array *seq* is exhausted, come out of the loop, i.e. $x \leq n$.
- While ($seq[x] == parent\_seq[y]$), increment the pointer y.
- As we come out of this loop, increment the pointer x.
- Now, after the termination of the loop, two conditions may arise:
  - if y reached the end, i.e. $y > n$ and x did not reach the end, i.e. $x < n$, then the parent array is well in order. So it is a valid BFS traversal.
  - if x reached the end, i.e. $x > n$ and y dis not reach the end, i.e. $y < n$, we reached the end of the *seq* array, which are the leaf nodes to find the parent, which is not possible. Hence *seq* is not a valid BFS traversal.

The **pseudo code** for the above algorithm is as:

```
if(flag==1){
    int parent_seq[n+1];
    for(int i=1;i<=n;i++){
        parent_seq[i]=parent[seq[i]];
    }
    int x=1;
    int y=1;
    while(x<=n && y<=n){
        while (seq[x]==parent_seq[y]){
            y++;
        }
        x++;
    }
    if (y>n && x<n){
        cout<<"VALID";
    }
    else if(x>n && y<n){
        cout<<"INVALID";
    }
}
```

Time Complexity Analysis:
The above algorithm is also an $O(n)$ time check. This is because, in each iteration of the inner while loop, the pointer y is incremented. And if the control flow doesn't enter the inner while loop, then the pointer x is incremented. So, in each iteration of the outer while loop x and sometimes y too is incremented. And we exit the loop when either x or y reach the end. Hence it is an $O(n)$ time check. If the sequence seq is a valid BFS traversal, it must pass the above two tests. Both are individually $O(n)$ time tests. So the overall time complexity of the algorithm is $O(n)$.