ESO207 THEORETICAL ASSIGNMENT 3

Aditi Khandelia

4th November 2023

1) Picture a playful adventure in a land of cities and roads! Meet Comren, a curious explorer ready to roam this exciting world. The cities are dots on a map, and the roads are the lines connecting them. This can be represented as an undirected graph representing a network of cities connected by roads. Comren won't settle for less, it's no fun to repeat or miss any road, all or none. Your job? Travel to every city once, using each road exactly once, and finally return to where you started, if at all possible.

(a)Given a Depth-First Search (DFS) traversal, can you find such a path? Why or why not?
Ans.
If given a sequence of vertices produced by DFS, it is not possible to find the path mentioned in the question.This is because consider a tree, as well as a another graph taht has the same edges as the previous graph along with an edge between the leaf and the root,they will produce the same DFS sequence. However, clearly the required path will not exist in the first graph since it will contain no cycles, barring us from revisiting the first vertex.

However, we can find the required path using a modified version of depth-first traversal. During depth first traversal, upon reaching a current vertex v, we visit all its unvisited neighbours.Here, instead of visiting an univisted neighbour, we will use an unvisited edge. Whilst traversing, all the vertices can be stored temporarily. If suppose all the vertices have even degree (which is a necessary condition for the existence of the path as that mentioned in the question), we can only get stuck at the starting vertex.If suppose we have arrived at the starting vertex without having explored all the paths, we can backtrack and add the vertices with no unvisited edges to the permanent path while also popping them from the temporary path. Once we reach an ancestor that has an unexplored edge, we can again perform DFS. This process is continued till the stack storing the temporary path empties, signifying that all the paths have been explored and that we have reached the vertex we started from. Since DFS is the traversal technique that allows us to explore as far as possible, till there are unvisited edges left, and can also be used to backtrack to the closest ancestor with an unvisited edge, it can be utilised to find the required path.

(b) Given a Breadth-First Search (BFS) traversal, can you find such a path? Why or why not?
Ans.
Neither a sequence of vertices produced by BFS traversal nor BFS traversal technique can be used to find the path as described in the question. This is because BFS traverses vertices in the order of their increasing distance from the starting vertex. Thus, suppose a vertex at distance d is visited and another vertex is present at a distance of d. BFS traversal reuses the edge used previously to backtrack to the parent and then visit the other vertex.Thus, edges will be traversed more than once in BFS traversal.

(c) What conditions should the graph meet for you to be able to traverse each road exactly once and return to the starting city?
Ans.
The condition for such a path to exist is that all the vertices in the graph should have an even degree and that all the vertices belong to the same connected component.
First we will prove that the all vertices must have even degree:
Suppose a particular vertex has an odd degree.
Case 1: This is the starting vertex. Suppose the no. of edges in incident on this vertex is 2n+1 for some integer n. Suppose the tour begins with an edge a. Clearly the next edge to be used that is incident on this vertex will be used to return to this vertex from some other vertex. Thus, after the usage of two edges, the current vertex is the starting vertex. So, after using 2n edges, we'll end up at the starting vertex. The next edge can only be used to leave this vertex, implying that there will be no edge through which we can return to the stratin g vertex.This is clearly not allowed in the problem.

Case 2:This is an intermediate vertex. Again, suppose the no. of incident edges are 2n+1 for some integer n. Since the tour doesn't begin at this vertex, the first edge incident on this vertex to be used will be used to enter this vertex. Clearly, the next edge will be used to depart this vertex. Thus, after using two edges, we end up at a vertex different than the current vertex. So, if we use the last edge, it will only be to enter the current vertex. According to the question, the tour must end at the starting vertex. However, in this case there is no edge to depart the current vertex. Hence, this is not possible.

Case 3: Vertex is not present in the path. This is not allowed by the constraints of the problem.

Thus, any of the three cases is not possible, implying that we can only have vertices of even degree.

All vertices should be connected since the question requires that the explorer must visit all cities. Thus, if the tour begins in the connected component C_1, there will be no path to any city in the connected component C_2, thus barring the explorer from visiting some cities.

(d)If the graph meets the conditions mentioned in part c, outline the steps you would take to find such a path.
Ans.
The required approach to find the path mentioned in the problem involves using the unvisited edges of a particular vertex till we reach a vertex that has no edges left to be explored, and then backtracking to a vertex that has unvisited edges left to be explored to explore those.
The answer can be modelled using two stacks. One will store the temporary path and the other will store the final path.
Steps to be taken:

1. Begin with the starting vertex. Push it in the stack storing the temporary path.
2. Go to a neighbouring vertex through an unvisited edge. This edge is now deleted from the graph and the vertex is added to the stack of temporary path and marked visited.
3. Continue the process till we reach a vertex that has already been visited. Now, begin to check the topmost element of the stack. If it has an unvisited edge, continue with step 2. Otherwise pop this vertex and place it in the stack storing the permanent path.
4. Continue popping till we find a vertex that has an unvisited edge. Then, continue with step 2.
5. Repeat the process till the starting vertex is added to at the top of the stack and all the edges have been explored.

Suppose the graph is given in the form of an adjacency list G(V,E).Algorithm to find the path:

```
FindPath(G,v)
{
    temppath=createemptystack();
    permpath=createemptystack();
    push(temppath,v);
    while(!isempty(temppath))
    {
        q=top(temppath);
        if q has no unvisited edges: pop(temppath); push(permpath,q);
        else:
        {
            for unvisited edge (u,v): push(temppath,u));
        }
    } while(!isempty(permpath)) cout¡¡pop(permpath)¡¡endl;
}
```

2.We have chaotic Dinosaurs dormant(which can be activated by a signal) across all cities in a state. The state has several cities connected via roads, which can be represented as an undirected graph. Some of these cities have towers. Each tower has the same power, say x. Towers can send signal to cities whose shortest distance from the tower is at most x. A tower is activated iff the city where the tower is situated receives a signal from another tower.
If the city receives a signal, the dormant dinosaurs destroy the city. You, an evil mastermind, have a list of all cities with a tower and a map of the state. Your source city is S. You want to destroy the destination city D.
Note: Source city S has a tower, and you can only activate this tower. However, you can also configure the power of every tower, x.

(a)Write clear pseudo code and explain logic briefly to check whether the power of x for all towers will be able to send a signal from S to D.
Ans.
The code uses Breadth First Search Traversal since this travel technique can be used to find the shortest distance between the root node and any vertex that is connected to it. The logic of the code is that we need to find all the towers that lie at a distance less than x from an activated tower, to activate those as well. This can be done by enqueuing those vertices which are at a distance of less than x from an activated tower, and then updating the distances of its neighbours accordingly.

1. The algorithm begins with the source whose distance is set to 0 and it is enqueued into an empty queue.
2. The program runs till either we reach the destination vertex or till the queue becomes empty, in which case the given x can not be used to reach the destination vertex.
3. During each iteration of the loop, we perform dequeue to obntain the vertex v. If the vertex v is at a distance >= x, it is removed from the queue without performing any operations on it.
4. If suppose distance(v)<x, then we explore all the neighbours of v. If the neighbour is a tower, we first check if the neighbour is the destination tower d. If yes, we return true and the program terminates. Otherwise, since now distance of the tower<=distance(v)+1<x+1, thus the tower gets activated. Its distance is set to 0.
5. If the neighbour is not a tower, its distance is updated and it is queued.

Suppose the graph is represented by G(V,E), the source is given by the vertex S and the destination by the vertex D.Pseudocode for the algorithm:

```
IsAllowedX(G,tower,S,D,x,n)
{
    distance[n];
    for(int i=0;i<n;i++) distance[i]=∞;
    distance[S]=0;
    Q=createemptyqueue();
    while(!isempty(Q))
    {
        v=dequeue(Q);
        if(distance(v)<x)
        {
            for neighbour w of v:
            {
                if(tower(w) and w==D) return true;
                else if(tower(w))
                {
                  distance(w)=0;
                  enqueue(w);
                }
                else
                {
                  if(distance(w)>=(distance(v)+1) distance(w)>=distance(v)+1;
                  enqueue(w);
                }
            }
        }
    }
    return false;
}
```

In the worst case, BFS is carried out for each vertex. Thus, the time complexity of the algorithm is O(|V|(|E|+|V|)).

(b)Write clear pseudo code and explain logic briefly to obtain the minimum power of tower required to send a signal from S to D.
Ans.
This part of the program makes use of binary search. Suppose a certain integer x can destroy the city D. Then, for every x'>x, the city D will get destroyed.This can be proven as follows:Since the locations of towers remain constant, so do the distances between them. If D gets destroyed when the power is x, the tower which causes the destruction of D is present at a distance <= x from D. Now, distance of the tower <= x' since x'>x.Thus, D gets destroyed if the tower is

activated.
When we consider the tower, it was activated by another tower at a distance<=x.As proven above, we can prove that the tower is activated even if the power is x'. This way we can keep moving from one tower to the previous till we reach S. Thus, if a certain integer x can destroy the city D then, for every x'>x, the city D will get destroyed.

Now, the maximum distance between any two nodes for a connected graph of n nodes can be (n-1). Thus, x lies in the range 0 to (n-1).

```
FindMinX(G,tower,S,D,n)
{
    min=0;
    max=n-1;
    while(low!=high)
    {
        mid=(min+max)/2;
        if(IsAllowedX(G,tower,S,D,mid,n)) max=mid;
        else min=mid+1;
    }
    return min;
}
```

Binary search has time complexity O(log |V|) while the function that checks if the given x is allowed is O(|V|(|E|+|V|)). Thus, the time complexity of the algorithm is O(|V|(|E|+|V|)log |V|).

3.Shantanu and his friends are very excited to visit their home during the midsem break, being the rowdy bunch that they are they decided to color bomb their hostel before leaving. Upon leaving he realizes that he no longer knows what color the rooms are, and since he is the hall president it is very important to know what the final color of every room is, Shantanu must figure out the colors on his way home, he remembers who fired what shots and in what order, but since he lives in a big hostel with a lot of rooms, he is unable to perform the requisite calculations on his own. Luckily he finds you as his co-passenger on the ride home.
There are n rooms in the hostel, you are given m bombings in chronological order and each bombing is of the form (l,r,c) which means all rooms from l to r where 1 <= l < r <= n were bombed with color c. Give an O((m + n) log n) time algorithm to help him find the final color of every room.

Ans.
This problem can be solved by the usage of a segmented tree. Here, the array of leaves will contain the room numbers in chronological order. Additionally, all the nodes will have two fields, one storing the serial number of colour c, and the other storing the serial number of the bombing.

Suppose there is a bombing of the form (l,r,c). Then the colour field of the leaf node of room l is updated to be the colour c.Additionally, let this be the k^{th} bombing. Then the field storing the serial number of the bombing is updated to be k. If this node is the left child of the parent, same updates are made for its right sibling. Similarly, the colour field of the node of room r is updated to c and the serial number to k. If it is the right child of its parent, same updates are made to its left sibling, provided that that room lies in the range [l,r]. We keep continuing this process as we move up towards the parents and stop when the nodes on both sides have the same parent.

To report the final colour of the room r, we use two variables col and num. Traversal begins from the current node, where col is set to be the colour label of the node and num is the serial number of the bombing.We then move upwards till we reach the root of the tree, and at each level we compare num with the serial number of the ancestor node. If it is greater, col is updated to the colour label of the ancestor.

This structure can be implemented using an array of structs.
The structure is defined as:

```
struct node
{
    char colour;
    int snum;
}
```

Based on this the tree is made as:

```
MakeTree(n)
{
    N=2^{ceil(log(n)};
    struct node tree[N];
    for(int i=0;i<N;i++) tree[i].snum=0;
    return tree;
}
```

Suppose the total no of rooms is n. Initially, the serial number for all nodes is set to be 0.

```
Bombing(tree,l,r,c,n,k)
{
    N=2^{ceil(log(n))};
    l=(N-2)+l;
    r=(N-2)+r;
    tree[l].colour=c;
    tree[l].snum=k;
    tree[r].colour=c;
    tree[r].snum=k;
    while(floor((l-1)/2)!=floor((r-1)/2))
    {
        if(l mod 2==1) tree[l+1].colour=c;tree[l+1].snum=k;
        if(r mod 2==0) tree[r-1].colour=c;tree[r-1].snum=k;
        l=floor((l-1)/2);
        r=floor((r-1)/2);
    } return tree;
}
```

The segmented tree has 2*N-1 nodes. During each iteration, we move to the ancestor of the current node, and this process continues till the ancestors of the left and right current nodes become the same, which in the worst case is the root.Additionally, some instructions that take constant time are carried out during each iteration.
Thus, the time complexity is O(log(2*N-1))=O(log(N)=O(log($2^{ceil(log(n))}$))=O(log(n)).
m bombings will take O(mlog n) time.

```
RoomColour(tree,l,n)
{
    N=$2^{ceil(log(n)}$;
    l=(N-2)+l;
    col=tree[l].colour;
    num=tree[l].snum;
    l=floor((l-1)/2);
    while(i > 0)
    {
        if(tree[l].snum>num) num=tree[l].snum; col=tree[l].colour;
        l=floor((l-1)/2);
    }
    return col;
}
```

As in the previous code, this code will also move up each level till we reach the root of the tree. In each iteration of nthe loop, some instructions that take constant time are carried out. Thus, time complexity=O(log(2*N-1))=O(log(N)=O(log($2^{ceil(log(n))}$)))=O(log(n)).

The main program :

```
GiveColours(n,m)
{
    tree=MakeTree(n);
    for(int i=0;i<m;i++)
    {
        cin>>l>>r>>c;
        tree=Bombing(tree,l,r,c,n,i+1);
    }
    for(int i=0;i<n;i++) cout<<RoomColour(tree,i+1,n)<<endl;
}
```

4.While he is home on vacation, Shantanu takes his little brother, Anuj, to visit the ongoing festival fair, his brother insists on eating from some consecutive set of sweets from a stall, Shantanu was aware some- thing like this would happen and had therefore asked the prices of all the sweets beforehand. The prices may change as the fest progresses but being the Bacchan of Shastri Nagar, he'll get to know as soon as a price changes.

Shantanu has some money M,when his brother makes a request of the form (l,r) where 1 <= l < r <= n, he checks if he has enough money, if so he fulfills his request, also, note that as soon as a request is fulfilled, Shantanu is returned all his money since the vendor doesn't want to incur his wrath.

Given the money M an n queries (updates + requests) in chronological order, give an O(n log n) time algorithm to that outputs "YES" if a request can be fulfilled and "NO" if it cannot.

Ans.
This problem can be solved using a segmented tree. Here, the leaf nodes will store the prices of each sweet. Some additional leaf nodes are added to make the lower level a power of 2. The prices here are set to be 0. Then, as we move upwards, the nodes will store the sum of the values of their children.
The following code will return the price of all sweets in the range [l,r]:

```
Price(T,l,r,M)
{
    N=$2^{ceil(log(n))}$;
    price=0;
    l=(N-2)+l;
    r=(N-2)+r;
    price+=T[l];
    price+=T[r];
    while(floor((l-1)/2)!=floor((r-1)/2))
    {
        if(l mod 2==1) price+=T[l+1];
        if(r mod 2==0) price+=T[r-1];
        l=floor((l-1)/2);
        r=floor((r-1)/2);
    }
    if(price<M) cout<<"YES"<<endl;
    else cout<<"NO"<<endl;
}
```

The segmented tree will have 2*N-1 nodes. Each iteration of the loop moves up one level and performs some operations on that level that take constant time. Thus, time complexity of answering each request=O(no. of levels)=O(log(2*N-1))=O(log(N)=O(log($2^{ceil(log(n))}$))=O(log(n)).

During update operation, we will have to update the price of the node storing that particular sweet, as well as the values of each ancestor since the value of their child is changing.

```
UpdatePrice(T,i,x)
{
    N=$2^{ceil(log(n))}$;
    l=(N-2)+l;
    T[l]=x;
    l=floor((l-1)/2);
    while(l>0)
    {
        T[l]=T[2*l+1]+T[2*l+2];
        l=floor((l-1)/2);
    }
}
```

Again, each iteration of the loop moves one level up in the tree. During each iteration some commands are executed that take constant time. Thus, the time complexity fopr each update operation=O(no. of levels)=O(log(2*N-1))=O(log(N)=O(log($2^{ceil(log(n))}$))=O(log(n)).

The combined program for answering all queries:

```
AnsQuery(T,M)
{
    for(int i=0;i<n;i++)
    {
        int op;
        int a,b;
        cin>>op>>a>>b;
        if(op==1) Price(a,b,M);
        else UpdatePrice(a,b);
    }
}
```

Here, the tree T is the segmented tree. If the initial prices are given by the array P[n], the tree is constructed by:

```
ConstructT
{
    N=$2^{ceil(log(n))}$;
    T[2*N-1];
    for(int i=0;i<2*N-1;i++) T[i]=0;
    for(int i=N-1;i<=N-2+n;i++) T[i]=P[i-(N-1)];
    for(int i=N-2;i>=0;i–) T[i]=T[2*i+1]+T[2*i+2];
}
```

5.It is time for Shantanu to return home after the midsem break, since he was a little sad leaving home, his mother packed fresh home grown apples for the ride back to campus.
Shantanu, being the nerd he is has numbered all the apples growing on his mother's apple tree and made a tree structure using the apples as nodes. He finds you on his ride back and tells you all about his tree. Now you're both getting hungry but the sequence of apples is edible iff you eat them in a valid BFS order.
Given the tree T with n apples, and a sequence of n apples, give an O(n) time algorithm to help Shantanu find if the sequence is edible or not.

Ans.
For a sequence to be considered a valid BFS traversal, the nodes must be sequenced in the order of increasing distance from the source node. Additionally, when two nodes are present at the same distance from the root, the node whose parent was traversed first must be traversed first as well.Now, since the graph is already a tree, each element will have a fixed parent in the BFS traversal beginning from a particular node.Suppose this were not true and any node has two or more than two parents.Now, in a BFS traversal, the edge can be to the same level, the previous level or the following level.Thus, both the parents lie at the same level, the level above the node in consideration,implying that one is not the ancestor of the other. Since both these nodes have a path to the root as well as a path to their common child, we get a cycle which is not possible in a tree.
Thus, given the sequence in the form of an array S containing n elements, we can carry out BFS traversal beginning from the node S[0] to calculate the distance of each node from S[0] as well as their parent. This can then be used to check if the given sequence is a valid BFS traversal.

Initially visited for all nodes is set to false.Code to initialise parent array and distance array:

```
BFS(G,S[0])
{

    Q=createemptyqueue();
    visited[S[0]]=true;
    distance[S[0]]=0;
    parent[S[0]]=S[0];
    while(notempty(Q))
    {

        v=dequeue(Q);
        for (neighbour u of v)
        {
                if (visited(u)=false)
                {
                    distance(u)=distance(v)+1;
                    visited(u)=true;
                    parent(u)=v;
                }
            }
        }
}
```

Now, BFS traversal has time complexity O(n+m). Since no. of edges in a tree with vertices n is n-1, BFS traversal has time complexity O(n+n-1)=O(n).

Next, we will traverse the sequence S to and check that all the vertices are being traversed, and in the order of non-decreasing distance, and that the node whose parent was traversed first gets traversed first as well. For this, we will keep another array called serailnum which will be analogous to DFN, but in the context of BFS. We first create visitseq which is set to false for all nodes.

```
IsValid(visited,parent,distance,S,n)
{
    currdistance=0;
    currparentserialnum=0;
    serialn=0;
    serialnum[S[0]]=0;
    for(int i=0;i<n;i++)
        if(visitedq[S[i]]) return false;
        else if(distance[S[i]]<currdistance or distance[S[i]]-currdistance>1)
        return false;
        else if(distance[S[i]]==currdistance)
            if(serialnum[parent[S[i]]]==currparentserialnum)
                serialnum[S[i]]=serialn;
                serialn++;
                visitedq[S[i]]=true;
            else if(serialnum[parent[S[i]]]>currparentserialnum)
                serialnum[S[i]]=serialn;
                serialn++;
                visitedq[S[i]]=true;
                currparentserialnum=serialnum[parent[S[i]]];
            else return false;
    return true;
}
```