ESO207 Theoretical Assignment 3 Submission

Naman Kumar Jaiswal
220687
namankj22@iitk.ac.in

Fall Semester 2023

Question 1: All or None

For parts A, B the required tour is a Euler path. For C, D the required tour is a Euler Curcuit.
Eulerian Path is a path in a graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path that starts and ends on the same vertex.

Part A:

We can not find a Euler path from a DFS traversal.
Though - if we are given that such a path exists (if not we will need to calculate degrees of vertices) - we can use DFS traversal to identify cycles and use back tracking to find an euler path for the part of graph - we get after removing the cycles - and then adding the cycles in the path to get a longer Euler path for the complete graph.
But the question asks that given a DFS traversal can we find an Euler Path? The answer is NO as we need to perform DFS traversal with a lot of modifications and use backtracking to find loops and go along it before bumping into an edge where we do not have anywhere to go.

Part B:

Given a BFS traversal, we cannot find an Euler path for a graph G.
For an Euler path when we traverse an edge - we are not allowed to go back but rather we have to go further down the depth. The core principle of BFS is that after a node K is traversed we are going to traverse the next node from the parent of K. This is fundamentally different from the thought process we must employ for finding an Euler path.

Part C:

The conditions for the Undirected path to possess an Eulerian Circuit is

1. The graph must be connected – This is obvious as we have to visit each city at least once.
2. The graph must only contain vertices with even degrees – For non start/end vertices whenever we reach the node we must also leave it - even parity of its degree – For start/end node - we have to have a starting edge and an ending edge - even degree.

Part D:

We need to find an $\mathcal{O}(E)$ algorithm to find a Eulerian Circuit in the graph.

Thinking Process:

Since we are given that a Eulerian Cycle exists for the graph every node has even, whenever we start a traversal we can not end on another node with even degree. We can only stop when we reach the starting edge once again.

Logic:

1. First we will start traversing the graph and insert the edge as a part of the circuit only when there are no more edges reachable from it.
2. We will push each node we traverse into a stack. We will backtrack the stack only when we reach the dead end till we get to a node from where we have a unvisited edge.
3. The temporary path is stored in a temp vector while the original path is inserted only in the stack.
4. We will pop the stack and print it one by one.

Principle for Optimization:

1. Use of backtracking to only traverse the graph a single time.
2. When we reach an endpoint, the rest of the graph is a Cycle which can be superimposed on the current path to get the final Euler Cycle.

Pseudocode:

1. Euler Path Function:

```
// Adj is the adjacency list,

EulerCycle(adj[]) { if (n = 0) {return;} // empty graph

    // The Current Tour we are taking is the Tour vector
vector<int> Tour; Tour.push_back(0);

// list to store final EulerCycle
vector<int> EulerCycle;

//Tour.size() goes less than 0 only when we traverse the complete graph.

while (Tour.size() > 0) {Vertex ← Tour[Tour.size() - 1]; //latest vertex

    if (adj[vertex].size() > 0) {// Not a deadend.
    //Push the new vertex to the stack
     Tour.push_back(vertex); }

    else { //deadend reached - supposed to be the end of the Euler Cycle
    //Backtrack and push this portion of path in the circuit
    circuit.push_back(Tour.back()); Tour.pop_back();}}
```

2. Printing the euler path:

```
while (circuit.size() > 0) {
    cout >> circuit[circuit.size - 1]; circuit.pop_back();}
```

Question 2: Chaotic Dino

Part A:

We need to find an algorithm that will return whether a node dest is reachable from a source src if every src is allowed to traverse only dept x. There are more nodes that has the possibility to become a node if and only if it has been reached by another source

Thinking Process:

Since we have to traverse all the layers of depth x from the source, we need to BFS traversal. Also, since we need to redo the source action whenever we find a tower, we are going to push it to a queue and then pop it later.

Logic:

1. First we BFS traverse the given graph from source till the depth x. When we reach a tower we push it to a queue. When the BFS is over, we pop a src and redo this step.
2. We will break only when the destination is reached or the queue gets empty.

Principle for Optimization:

1. Using BFS to perform layer-based traversal in linear time.

Pseudocode:

1. BFS traversal:

```
//BFS will return true if it reaches the destination
//x is the power of Signal

BFS(s){ dist[s] ← x; q.push(s);
    while (!q.empty()) {v ← q.pop(); if (dist[v] = 1) {break;}
    for (auto u : adj[v]) {
    if (Tower[u]) {Queue.push_back(u);}
    if (u = dest) {return TRUE;}
    if (dist[u]) {dist[u] = dist[v] - 1 > dist[u] ? dist[v] - 1 : dist[u] ;}
    if (!dist[u]) { dist[u] ← dist[v] - 1; q.push(u);}
    }} return FALSE;}
```

2. Tower Analysis:

```
vector<int> dist[n], Tower[n], Queue[n], q[n]; vector<vector> adj[];

//Tower vector stores whether a node is a tower or not
//dist stores the power it will be getting

Destruction(x, src, dest) {while(!Queue.empty()){
    k ← Queue.pop_front();
    if (BFS(k)) {return TRUE;}} return FALSE;}}
```

Part B:

We have to find the minimum value of x for which Destruction (x, src, dest) is TRUE.

Thinking Process:

We can use binary search on a range of 1 to n - 1 for the maxima element.

Logic:

(a) If Destruction (i, src, dest) is TRUE Destruction (i - 1, src, dest) is FALSE, i is the answer.

(b) if Destruction (1, src, dest) is TRUE, 1 is the answer

(c) If Destruction (n - 1, src, dest) is False, n is the answer

Principle for Optimization:

(a) Using Binary Search.

Pseudocode:

(a) Binary Search:

```
BinarySearch(src, dest){ if (Destruction (1, src, dest)){return 1;}
    if (Destruction (n - 1, src, dest)){return 1;}
    l ← 1, r ← n - 1;
    while ( l <> r ) { mid ← ( l + r ) / 2;
    if ( Destruction (mid, src, dest) && !Destruction (mid - 1, src, dest)
    ) {return mid;}
    if ( Destruction (mid, src, dest) && Destruction (mid - 1, src, dest))
     {r ← mid - 1;}
    if ( !Destruction (mid, src, dest) && !Destruction (mid - 1, src, dest
    )) {l ← mid + 1;}
    }
    }
```

Question 3: Room Colors

We have to find a data structure to store the room numbers and the color allotted to them. But once again we can not store it in just a simple array as it will burst the time complexity bound.

Thinking Process:

The color bombing operation is a segmented operation - we can store the colors for a room in a segment tree. We have m update operations and n fetch color operations under the time of $\mathcal{O}((n+m)\log(n))$, hence each operation in $\mathcal{O}(\log(n))$ time.

Logic:

(a) We will first populate the segmented tree by assigning the n leaf nodes as the rooms and the rest nodes as the latest color bombing that happened to them. We will also populate a 2D table that stores a color bombing and the corresponding color.

(b) For the Update operation t - (l, r, c) we will add the time-color (t, c) pair to the 2D table and the (l, r) segment as the union of a few pre-set segments in the segment tree and update the time in the segment tree.

(c) For Fetch Color operation (room) we are going to travel the ancestry of the room in the segment tree and return the time of the latest color bombing of the room. Then we will return the corresponding color from the table.

Principle of Optimization:

(a) Flexibility of update in $\mathcal{O}(\log(n))$ in Segment trees as well as $\mathcal{O}(\log(n))$ nature of fetch time of a room in a Segment Tree which stores time.

Pseudocode:

(a) Population of the Segment Tree (array-based implementation) and the 2D time-color map:

```
// Segment trees are complete binary trees.
// Extra cells are initialized to 0 - which is an unrealistic value for a
    room and also will not interfere with our interested portion.

t = pow(2, ceil[log(n)]);

Array SegmentTree[2*t] = 0; //initialize with 0

//For uncolored rooms
vector<pair<int, string>> ColorMap;
ColorMap.push(0, "No Color");

//Room Numbers from 1 to n -- leaf nodes
//The nodes represent the latest time they were being color-bombed
```

(b) Update Operation

```
Update (l, r, c, m) { l ← n + l; r ← n + r; SegmentTree[l] ← m;
pair p(m, c); ColorMap.push_back(p);
if (r > l){
     SegmentTree[r] = m;
     while(floor(l/2) <> floor(r/2)){if(l%2 = 0){ SegmentTree[l+1] ← m; }
        if(r%2 = 1){SegmentTree[r-1] ← m;}
        l ← floor(l/2); r ← floor(r/2);}}}
```

(c) Fetch Color Operation:

```
FetchColor (i) {temp ← n + i; latest ← 0;
    do { latest ← SegmentTree[ temp ] > latest ? SegmentTree[ temp ] : latest; temp ← floor (temp / 2) ;} while ( temp <> 1)}
    return ColorMap[latest].Second;
}
```

Question 4: Fest Fever

We need a data structure to store the sweet prices which is flexible to update and can return the sum operation of a continuous interval.

Thinking Process:

Since we need an algorithm that can update a specific sweet price and return the sum of a segment of shops in at most order of log n time, we can use segment trees to store them.

Logic:

(a) First we will populate the segment tree array (size 2n) with last n leaf elements being the prices of sweets, while other nodes storing the sum of all it's leaf elements.

(b) For requests of (l, r) form, dividing the given segment as a union of pre-set segments and adding them up directly from the segment tree.

(c) For the requests of the form (i, x) we will find the net increment Δ and add it to the leaf element and its ancestry.

Principle for Optimization:

(a) Flexibility of update in $\mathcal{O}(\log(n))$ in Segment trees as well as $\mathcal{O}(\log(n))$ nature of sum operation of a segment in a Segment Tree which stores sum.

Pseudocode:

(a) Population of the Segment Tree (array-based implementation):

```
// Segment trees are complete binary trees.
// Extra cells are initialised to 0 - which is an unrealistic value for a
    price and also will not interfere with our interested portion.

t = pow(2, ceil[log(n)]);

Array SegmentTree[2*t] = 0; //initialize with 0

//Sweet prices -- leaf nodes
for (int i ← 0; i < n; i++) { cin >> SegmentTree[n + i];}

//Sum of prices -- non-leaf nodes
for (int i ← n - 1; i > 0; i--)
{ SegmentTree[i] = SegmentTree[2 * i] + SegmentTree[2 * i + 1];}
```

(b) Update Operation:

```
Update (i, x) {
inc ← SegmentTree[n + i] - x; temp ← n + i;

do {SegmentTree[temp] += inc; temp ← floor(temp / 2);} while (temp <> 1)
} // updates the whole ancestry of i
```

(c) Sum Operation:

```
Sum (l, r) {l ← n + l; r ← n + r; Sum ←  SegmentTree[l];
if (r > l){
    total_price += SegmentTree[r];
    while (floor(l / 2) <> floor(r / 2){
        if( l % 2 = 0 ) {Sum += SegmentTree[l+1];}
        if( r % 2 = 0 ) {Sum +=  SegmentTree[r-1];}
        l ← floor(l / 2); r ← floor(r / 2);}
return Sum;}
```

(d) Main:

```
// We have already populated the Segment tree

for (int i ← 0; i < queries; i++){
    if ( query_type = update )  {cin >> i >> x; Update(i, x);}
    if ( query_type = Sum )  {cin >> l >> r;
    if (Sum (l, r) <= money} {cout >> "YES";}
    else { cout >> "NO"; }}}
```

Question 5: Edible sequence

We are given the tree with its nodes numbered. We have to identify whether a permutation of these n nodes are in a valid BFS order.

Thinking Process:

We are given that the sequence is of length n means that we can reduce our problem to three conditions - the sequence must start with the root - nowhere in the sequence can a node of level i come before a node of level less than i - the children of a node must come before the children of another node in the same level.

Logic:

(a) First we will apply BFS on the tree and populate the parent and dist array.
(b) Then we will start checking the sequence for conditions. Condition 1 is straightforward.
(c) Then we will use two iterators - one at the parent level and the other at the children level to check for conditions 2 and 3.

Principle for Optimization:

(a) BFS for achieving a level-based structure providing for upcoming analysis.
(b) The fact that we are given a sequence containing n nodes and we can skip a lot of unnecessary conditions.

Pseudocode:

(a) Populating the parent and distance array:

```
Array parent[n] = -1, distance[n] = -1;
// initialise them both with - 1 to represent an unvisited node

BFS(adj[], s, parent[], dist[]){
    dist[s] ← 0; q.push(s);
    while (!q.empty()) {u ← q.pop();
    for (neighbor ← adj[u] -> next -> vertex_number; neighbor <> nullptr;
     neighbor ← neighbor -> next) {
        int vertex ← neighbor -> vertex_number;
        if (!visited[vertex]) { visited[vertex] ← true; dist[vertex] ←
    dist[u] + 1;
        q.push(vertex);
        parent[vertex] ← u;}
        }}}

// We are assuming adjacency list representation
```

(b) Parsing the given sequence s and check for any condition not being true:

```
//Assuming the sequence is an array and s is the root
Check(sequence[], dist[], parent[]){
    if (sequence[0] <> s) {return FALSE;}
    i ← 0, j ← 1;
    while (j < n) {if (parent[sequence(j)] <> sequence(i)) {return FALSE;}
        while (parent[sequence[j]] = sequence[i]) {i++;}
        j++;}
        return TRUE;}

// TRUE is returned for a edible sequence
// We have combined condition 2 with 3 an checked them simultaneously
// This may miss some errors in the initial parse for certain cases but
    due to the length of the sequence being n - it can be guaranteed that
    somewhere down the parse an error will be forced to be detected
// We are not required to return at which point the sequence failed to
    become BFS
```