July 4, 2024

Chayan Kumawat 220309 **Theoritical Assignment 3(ESO207)**

Question 1

(a)

The path described in the question can indeed be discovered using a Depth-First Search (DFS) traversal technique.This is made possible because, during each recursive call, the edge we add becomes connected to the preceding edge in the path. DFS traverses the vertices in a depth-first manner, making it well-suited to identify the specified path. Moreover, DFS inherently returns to previously visited vertices and initiates new paths, aligning with the requirements of the problem. With suitable modifications, DFS can effectively be employed to find a path that visits each city exactly once, utilizes each road exactly once, and potentially returns to the starting point

(b)

The path described in the question cannot be discovered through a Breadth-First Search (BFS) traversal. This is because BFS explores vertices in a fundamentally different manner compared to the requirements of the problem. we can't guarantee presence of a non-BFS tree edge while moving within the same level.In BFS, the focus is primarily on exploring vertices at a uniform distance from the initial vertex, whereas the problem at hand is concerned with visiting the next vertex directly connected to the current vertex. Therefore, the BFS traversal approach does not align with the specific conditions of this problem, making it unsuitable for finding the desired path.

(c)

To ensure that you can traverse every road once and return to the starting city in an undirected graph representing a network of cities and roads, two key conditions must be met:

1. Connectivity: The graph must be connected, allowing a path between any pair of cities.

2. Even Degree: Each city must have an even number of connected roads, ensuring a successful traversal of every road while returning to the initial city. These conditions stem from Euler's theorem, indicating that an undirected graph has an Eulerian circuit if all its vertices have even degrees and the graph remains connected.

(d)

Discription of the Algorithm :
In this algorithm, given the graph G represented as an adjacency list and a starting vertex v, we aim to construct a circuit that traverses all edges once and returns to the starting vertex, if possible. The algorithm begins by initializing two data structures: an empty stack called 'circuit' to maintain the current path and an empty list named 'path' to store the final Eulerian circuit. We push the starting vertex v onto the 'circuit' stack to initiate the traversal.

Within the main loop, the algorithm continuously explores the graph. It examines the current vertex, $curr_v$, located at the top of the 'circuit' stack. If there are remaining adjacent edges from $curr_v$, the algorithm identifies the next vertex, $next_v$, by selecting the last adjacent vertex. It then removes the edge between $curr_v$ and $next_v$ from the graph and records $curr_v$ in the 'path' list. Subsequently, $next_v$ is pushed onto the 'circuit' stack, allowing the traversal to proceed from there. In cases where there are no more adjacent edges, $curr_v$ is added to the 'path' list, indicating a backtrack.

This process continues until the 'circuit' stack becomes empty. After completing the traversal, the 'path' list is reversed to determine the Eulerian circuit, capturing the order in which the circuit starts and concludes at the initial vertex v.

```
findEulerianCircuit( graph,start_vertex) {
    stack<int> circuit;
    array<int> path;

```

```cpp
    circuit.push(start_vertex);

while (!circuit.empty()) {
    int current_vertex = circuit.top();

    if (!graph[current_vertex].empty()) {
        int next_vertex = graph[current_vertex].back();
        path.push_back(current_vertex);
        circuit.push(next_vertex);
        graph[current_vertex].pop_back();
    } else {
        circuit.pop();
        path.push_back(current_vertex);
    }
}
reverse(path);
return path;

}
```

Question 2

a)

Discription of the Algorithm The provided Algorithm checks if it's possible to send a signal from a source city S to a destination city D while destroying cities with dormant dinosaurs using towers of adjustable power. The algorithm initializes data structures and a queue for a breadth-first search (BFS). It processes towers and their properties, then performs BFS to calculate distances and merge connected components. If the power (x) of the towers is set correctly and the graph meets certain conditions, it ensures that S and D belong to the same connected component in the Disjoint Set Union (DSU) data structure. If they do, it returns true, indicating that it's possible to send a signal from S to D, destroying cities in the process. Otherwise, it returns false.

```
bool check(G,CityWithtowers, x, S,D) {
    int N = G.size();
    //we have the DSU data structure
    DSU dsu(N);

    // Initialize arrays and data structures.
    array dist(N) <- infinity;
    array par(N) <- -1;
    array visited(N) <- false;
    queue(parameter1, parameter2, parameter3) Q;

    // Add towers to the queue and initialize their
   properties.
    for (int)i form 0 to Citywithtower.size() {
        int tower = Citywithtowers[i];
        dist[tower] = 0;
        par[tower] = tower;
        visited[tower] = true;
        Q.push({tower, 0, tower});
    }

    //If the D is not a tower city then add it to the queue
    if (!visited[D]) {
        par[D] = D;
        visited[D] = true;
        dist[D] = 0;
        Q.push({D, 0, D});
    }

    while (!Q.empty()) {
        int node, distance, root;
        (node, distance, root) = Q.front();
        Q.pop();

        for (int neighbor : G[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                dist[neighbor] = distance + 1;
                par[neighbor] = root;
                Q.push({neighbor, distance + 1, root});
            } else {
                if (dist[neighbor] + distance + 1 <= x) {
                    dsu.merge(root, par[neighbor]);
                }
            }
        }
    }

    // Check if S and D belong to the same connected
   component in the DSU.
    return (dsu.find(S) == dsu.find(D));
}
```

(B)

Discription of the Algorithm : This algorithm uses binary search to find the minimum tower power required to send a signal from source city S to destination city D. It iteratively narrows down the power range by adjusting ‘x‘ and checks if it’s possible to establish a connection using the original algorithm. The minimum power is found when a valid connection is established.

```
int findMinimumPower(G, towers, S,D) {
    int low = 0, high = HighestpowerUpperbound;

    while (low < high) {
        int mid = (low + high) / 2;
        bool possible = check(G, towers, mid, S, D);
        if (possible) {
            high = mid;  // If it's possible with the
    current power, reduce the upper bound.
        } else {
            low = mid + 1;  // Otherwise, increase the
    lower bound.
```

```
        }
    }
    return low;  // The minimum power required is in '
   low'.
}
```

.

Question 3

Discription of the Algorithm :
To determine the final colors of rooms after multiple bombings in a hostel, create a perfect binary tree as an array with room colors and indices. Process bombings chronologically, updating the tree's color and index information within specified ranges. Then, recursively traverse the tree to find the most recent color for each room and store the results in a final-color array. This efficient algorithm runs in O((m + n) log n) time

```
 Node <- struct { color ,  step }

void applyColorBomb(left , right , newColor , Node* nodes ,
    step) {
    leafNodes = static_cast(pow(2, ceil(log2(right + 1))
    );
    left = leafNodes - 2 + left;
    right = leafNodes - 2 + right;

    while ((left - 1) / 2 != (right - 1) / 2) {
        nodes[left].color = newColor;
        nodes[left].step = step;
        nodes[right].color = newColor;
        nodes[right].step = step;

        if (left % 2 == 1) {
            nodes[left + 1].color = newColor;
            nodes[left + 1].step = step;
        }

        if (right % 2 == 0) {
            nodes[right - 1].color = newColor;
            nodes[right - 1].step = step;
        }

        left = (left - 1) / 2;
        right = (right - 1) / 2;
    }
}

```

```
void fillFinalColors(nodes, i, finalColors, currentColor, currentIndex, leafNodes) {
    if (i > leafNodes - 2 && i < leafNodes - 1 + n) {
        if (currentIndex > nodes[i].step) {
            currentColor = nodes[i].color;
            currentIndex = nodes[i].step;
        }
        finalColors[i - leafNodes + 2] = currentColor;
        return;
    } else if (i <= leafNodes - 2) {
        if (currentIndex > nodes[i].step) {
            currentColor = nodes[i].color;
            currentIndex = nodes[i].step;
        }
        fillFinalColors(nodes, 2 * i + 1, finalColors, currentColor, currentIndex, leafNodes);
        fillFinalColors(nodes, 2 * i + 2, finalColors, currentColor, currentIndex, leafNodes);
    } else
        return;
}

findFinalColors(n, m) {
    leafNodes <- pow(2, ceil(log2(n));
    size = 2 * leafNodes - 1;
    nodes = new Node[size];

    for (step = 0; step < m; step++) {
        left, right, newColor;
        applyColorBomb(left, right, newColor, nodes, step);
    }

    finalColors = new[n];
    fillFinalColors(nodes, 0, finalColors, 0, 1, leafNodes);

    delete[] nodes;

    return finalColors;
}
```

Question 4

Algorithmic Discription : The algorithm efficiently handles Shantanu's budget and his brother's requests during a festival. It uses a binary tree structure, where prices of sweets are stored in the leaves, and each parent node maintains the sum of its children. Updates are made by modifying the relevant prices and propagating the changes to the root, while requests are fulfilled by calculating the total price of sweets within the specified range by traversing towards the root.

```
void updatePrice(sweetIndex, newPrice, priceArray) {
    leafIndex = sweetIndex + priceArray.size() / 2 - 1;
    priceDifference = newPrice - priceArray[leafIndex];

    while (leafIndex != 0) {
        priceArray[leafIndex] += priceDifference;
        leafIndex = (leafIndex - 1) / 2;
    }
    priceArray[0] += priceDifference;
}

bool canAfford(leftSweetIndex, rightSweetIndex,
    priceArray, budget) {
    sum = 0;
    leafIndexLeft = leftSweetIndex + priceArray.size()/2
     - 1;
    leafIndexRight = rightSweetIndex + priceArray.size()
    /2 - 1;

    sum += priceArray[leafIndexLeft];
    if (leafIndexRight > leafIndexLeft) {
        sum += priceArray[leafIndexRight];
    }

    while ((leafIndexLeft - 1) / 2 != (leafIndexRight -
    1) / 2) {
        if (leafIndexLeft % 2 == 1) {
            sum += priceArray[leafIndexLeft + 1];
        }
        if (leafIndexRight % 2 == 0) {
            sum += priceArray[leafIndexRight - 1];
        }
        leafIndexLeft = (leafIndexLeft - 1) / 2;
        leafIndexRight = (leafIndexRight - 1) / 2;
    }

    return (sum <= budget);
}
```

Question 5

To ascertain whether a given sequence conforms to a Breadth-First Search (BFS) order for a tree, we undertake a BFS traversal of the tree, collecting essential information for each vertex: its level (distance from the root) and its parent. Subsequently, we conduct two critical checks:

1. **Level Consistency:** We verify that the levels of the vertices in the sequence are arranged in a non-decreasing order. This validation ensures that the traversal progresses from the root towards deeper levels, aligning with the expected behavior of BFS.
2. **Child Exploration Order:** As we transition from one level to the next in the sequence, we confirm that we explore the children in the same order as their parent level. For instance, if we traverse from left to right at one level, we uphold this order when inspecting the children at the subsequent level. This ensures that children of a parent enqueued first are visited prior to children of a parent enqueued later. This adherence to order preserves the fundamental principles of BFS traversal.

Performing a Breadth-First Search (BFS) on a tree takes O(V + E) time, where V represents the number of vertices and E represents the number of edges. For any given tree, this simplifies to O(n) time, as we visit and enqueue each vertex exactly once. Additionally, we traverse the sequence of n entries exactly once, which also takes O(n) time. Therefore, the overall time complexity of the algorithm is O(n).

```python

def isBFSOrder(sequence, root):

    queue = []
    level = {}
    visited = {}
    parent = {}
    pos = {}
    flag = True

    level[root] = 0
    visited[root] = True
    parent[root] = -1

    queue.append(root)

    while queue:
        curr_node = queue.pop(0)
        visited[curr_node] = True

    for child in curr_node.children:
        if child not in visited:
            visited[child] = True
            parent[child] = curr_node
            level[child] = level[curr_node] + 1
            queue.append(child)

    if sequence[0] != root:
        flag = False

    pos[sequence[0]] = 0

for i in 1 to sequence.length():
    pos[sequence[i]] = i

    if level[sequence[i]] < level[sequence[i - 1]] or
   pos[parent[sequence[i]]] < pos[parent[sequence[i -
   1]]]):
        flag = False
        break

return flag
```