Course ESO207: Data Structures and Algorithms
Attempt by Dhruv - 210338
Date November 2023

Problem Statement 1 - All or None

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) (2 points) Given a Depth-First Search (DFS) traversal, can you find such a path? Why or why not?

(b) (2 points) Given a Breadth-First Search (BFS) traversal, can you find such a path? Why or why not?

(c) (4 points) What conditions should the graph meet for you to be able to traverse each road exactly once and return to the starting city?

(d) (7 points) If the graph meets the conditions mentioned in part c, outline the steps you would take to find such a path. (Best Complexity: $O(|E|)$, 3 points for $O(|E|^2)$)

Solution

Part A.

No Becasue -
The dfs traversal give the order along the depth in which we visit the nodes. From a given traversal we cannot infer about structure of graph. Example - A simple cyclic and a simple linear graph can have same dfs traversal but one the two graph follow the required property.

Part B.

No Becasue -
The bfs traversal is a specific order in which we visit the nodes. From a given traversal we cannot infer about structure of graph. Example - A simple cyclic and a simple linear graph can have same bfs traversal but one the two graph follow the required property.

Part C.

Condition 1 : For each vertex $v \in V$ $degree[v]$ should be even, where $degree[v]$ is number of undirected edges connected to vertex v

Condition 2 : All vertices with non zero degree should be part of one component in Graph G.
(Assuming we need to visit all the roads and not cities. If we need to visit all the cities along with roads then all the vertices must form a single component of graph G)

Part D.

Algorithm Analysis:

Verify that the graph satisfies the given conditions. In an undirected graph, all vertices must have an even degree (an even number of edges connected to them).

- If the graph satisfies the conditions, start at any vertex in the graph. We can choose any vertex as the starting point.
- While the current vertex has unexplored edges:
  - Choose one of the unexplored edges from the current vertex.
  - Mark the edge as visited by removing it from the graph (from the current vertex's adjacency list).
  - Move to the adjacent vertex along the chosen edge.
  - Recursively call the Eulerian circuit (dfs) function for the adjacent vertex.
- When the recursive call returns, we have finished exploring all edges from the current vertex. Add the current vertex to the Eulerian circuit path.
- Repeat step 3 until all edges have been traversed.
- The order in which the recursive calls return will give us the Eulerian circuit

I will start the path from vertex 1 and each subsequent vertex in path gives us the road to travel.

Pseudo Code:

Algorithm 1: *dfs(int vertex, int parent, path)*

```
Result: Fills the required path
while $AdjList[vertex].size() > 0$ do
    int child = AdjList[vertex].back();
    AdjList[vertex].pop_back();
    if $child <> parent$ then
        dfs(child, vertex, path);
path.push(vertex);
```

Algorithm 2: *FindPath(AdjList)*

```
Data: AdjList represents the adjacency list of graph
Result: Returns the euler circuit path
$vector < int >$ path;
dfs(1, -1, path);
return path;
```

Time Complexity Analysis:
Each edge is visited twice in the dfs and each dfs call is constant operations hence

$$TimeComplexity = \boldsymbol{\mathcal{O}(E)}$$

Problem Statement 2 - Chaotic Dino

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. IIf 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) (15 points) 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.

(b) (10 points) Write clear pseudo code and explain logic briefly to obtain the minimum power of tower required to send a signal from S to D.

Solution

Part A.
Algorithm Analysis:
The Check algorithm checks whether it is possible to send a signal from the start city (S) to the destination city (D) with a given power parameter (x). It applies bfs for all the cities which contains tower and reachable from S or one of the other reachable tower. If any of the bfs from any reachable tower contains D within distance X we can reach D

Algorithm 1: *Check(s, d, x, g)*

```
Data: g is the adjacency list, s is start city, d is destination city, x is given parameter
Result: Check if x satsify given conditions
queue reachableTowers;
reachableTowers.push(s);
while !reachableTowers.empty() do
    int tower = reachableTowers.front();
    tower.pop();
    queue q;
    q.push(tower);
    dis[] = {0};
    vis[] = {false};
    while !q.empty() do
        int city = q.front();
        q.pop();
        if city == d then
            return true;
        if city contains tower and dis[city] <= x then
            reachableTowers.push(city);
        for child : g[city] do
            if !vis[child] then
                vis[child] = true;
                dis[child] = dis[vertex] + 1;
                q.push(child);
return false;
```

Time Complexity Analysis:

Check: In worst case each city with tower calls *bfs* $\Rightarrow \mathcal{O}((V+E)*Number\ of\ towers) \Rightarrow \mathcal{O}((V+E)*V)$

Time Complexity $\Rightarrow \mathcal{O}((V+E)*V)$

Part B.
Algorithm Analysis:
The MinX algorithm finds the minimum power of the tower required to send a signal from the start city (S) to the destination city (D) in the state. It uses a binary search to efficiently search for the minimum valid power (x). The algorithm iteratively checks whether the current midpoint (mid) of the search range satisfies the conditions using the Check algorithm. If it does, (all $X >= mid$ satifies Check) it updates hi and mn accordingly, narrowing the search space. If it doesn't, it updates lo, shifting the search space to the right.

Pseudo Code:

Algorithm 2: *MinX(s, d, g)*

```
Data: $g$ is the adjacency list, $s$ is start city, $d$ is destination city
Result: Returns minimum value of valid x
int n = $|V|$;
int lo = 0, hi = n;
int mn = n;
while $lo <= hi$ do
    int mid = (hi + lo) / 2;
    if $check(s, d, mid, g)$ then
        hi = mid - 1;
        mn = min(mn, mid);
    else
        lo = mid + 1;
return mn;
```

Time Complexity Analysis:
MinX : Binary search where each iteration of while loop calls *check*

Time Complexity $\Rightarrow \mathcal{O}((V+E)*V*log(V))$

Problem Statement 3 - Room Colors

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 $O((m+n)logn)$ time algorithm to help him find the final color of every room.

Solution

Algorithm Analysis:

This question is similar to multi-increment problem discussed in class. Each node of the complete binary tree (or *arr[i]*) stores the most recent query update number for the leaf nodes in its subtree (a segment of the rooms). To find the final coloring of a room we locate the room in *arr* and travel to its parents and find the parent with most recent (max) query update number. Color correspoding to this query number is the final color of the room.

Pseudo Code: (Assuming n to be power of 2)

Algorithm 1: *update(i, j, query)*

```
Result: Updates the coloring of rooms in range [l, r]
i = i + (n - 1);
j = j + (n - 1);
arr[i] = query;
if $j > i$ then
    arr[j] = query;
    while $(i-1)/2 <> (j-1)/2$ do
        if $i\%2 == 1$ then
            arr[i + 1] = query;
        if $j\%2 == 0$ then
            arr[j - 1] = query;
        i = (i - 1) / 2;
        j = (j - 1) / 2;
```

Algorithm 2: *report(i)*

```
Result: Returns last query update for room i
i = n - 1 + i;
int query = 0;
while $i > 0$ do
    query = max(query, arr[i]);
    i = (i - 1) / 2;
return query;
```

Algorithm 3: *color(n, m, q)*

```
Data: $q$ contains $m$ queries of form $l, r, color$
Result: Returns final coloring of all the rooms
int n, m;
int q[m + 1];
int arr[4n + 5];
for $int\ i = 1; i <= m; i++$ do
    int l, r, color;
    l–;
    r–;
    q[i] = color;
    update(l, r, i);
int coloring[n + 1];
for $int\ i = 0; i < n; i++$ do
    coloring[i] = q[report(i, n)];
return coloring;
```

Time Complexity Analysis:
update(i, j, query) : *arr* represents a complete binary tree. For each iteration of while loop we go to parents of nodes i, j, since max height is $O(log(n))$ and each iteration of while loop have constant operations $\Rightarrow \boldsymbol{\mathcal{O}(logn)}$
report(i) : Similar reasoning as *update operation* on index i $\Rightarrow \boldsymbol{\mathcal{O}(logn)}$
color(n, m, q) : Each iteration of first For loop calls *update* $\Rightarrow \boldsymbol{\mathcal{O}(mlogn)}$, Each iteration of second For loop calls *report* $\Rightarrow \boldsymbol{\mathcal{O}(nlogn)}$, Total $\Rightarrow \boldsymbol{\mathcal{O}(mlogn + nlogn)}$

$$TimeComplexity = \boldsymbol{\mathcal{O}(mlog(n) + nlog(n))}$$

Problem Statement 4 - Fest Fever

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 will 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 does not want to incur his wrath. Given the money M an n queries (updates + requests) in chronological order, give an $O(nlogn)$ time algorithm to that outputs "YES" if a request can be fulfilled and "NO" if it cannot.

Solution

Part A:

Algorithm Analysis:
(Discussed in Class)
Similar algorithm as dynamic range minima problem discussed in class, here we have range report and point updates. Each node of the complete binary tree (or *arr[i]*) stores the sum of values of leaf nodes of its subtree (a segment of the shops). To update the value of a shop we locate the shop in *arr* and travel to its parents and update the value parent node with sum of its two children. Similar reasoning can be used to answer range queries.

Pseudo Code: (Assuming n to be power of 2)

Algorithm 1: *report(i, j)*

```
Result: Returns the sum of prices of the shops in range [i, j]
i = i + (n - 1);
j = j + (n - 1);
int sum = arr[i];
if $j > i$ then
    sum += arr[j];
    while $(i-1)/2 <> (j-1)/2$ do
        if $i\%2 == 1$ then
            sum += arr[i + 1];
        if $j\%2 == 0$ then
            sum += arr[j - 1];
        i = (i - 1) / 2;
        j = (j - 1) / 2;
return sum;
```

Algorithm 2: *update(i, x)*

```
Result: Updates the price of the shop[i] to x
i = n - 1 + i;
arr[i] = x;
i = (i - 1) / 2;
while $i > 0$ do
    arr[i] = arr[2i + 1] + arr[2i + 2]; i = (i - 1) / 2;
```

Algorithm 3: *Request(n, M, price[n], query[n])*

Data: *query* contains n queries of updates and requests, *price* is the initial price of n shops
Result: Returns if a request can be fullfilled

```
int arr[4n + 5];
for int $i = 0; i < n; i++$ do
    arr[i + n - 1] = price[i];
for int $i = n - 2; i >= 0; i--$ do
    arr[i] = arr[2i] + arr[2i + 1];
for int $i = 0; i < n; i++$ do
    int type;
    if $type == 1$ then
        int idx, x;
        idx–;
        update(idx, x);
    if $type == 2$ then
        int l, r;
        l–, r–;
        if $report(l, r) <= M$ then
            $cout << "YES" << endl;$
        else
            $cout << "NO" << endl;$
```

Time Complexity Analysis:
report(i, j) : *arr* represents a complete binary tree. For each iteration of while loop we go to parents of nodes i, j, since max height is $O(log(n))$ and each iteration of while loop have constant operations $\Rightarrow \boldsymbol{\mathcal{O}(logn)}$
update(i, x) : Similar reasoning as *report operation* on index i $\Rightarrow \boldsymbol{\mathcal{O}(logn)}$
Request(n, M, price, query) : Two For loops for the initialization of *arr* $\Rightarrow \boldsymbol{\mathcal{O}(n)}$, Each of the n queries either calls *report or update* $\Rightarrow \boldsymbol{\mathcal{O}(nlogn)}$, Total $\Rightarrow \boldsymbol{\mathcal{O}(nlogn + n)}$

$$TimeComplexity = \boldsymbol{\mathcal{O}(nlog(n))}$$

Problem Statement 5. Edible sequence

(20 points) 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.

Solution

Algorithm Analysis:
In a valid bfs ordering nodes with depth (or level) smaller comes first. Consider two set of nodes, $\mathcal{L}_i = \{v_1, v_2, ..v_n\}$ such that $depth[v] = i \; \forall \; v \in \mathcal{L}_i$ and $\mathcal{L}_{i+1} = \{g_1, g_2, ..g_m\}$ such that $depth[g] = i + 1 \; \forall \; g \in \mathcal{L}_{i+1}$. In a valid bfs order nodes all nodes at same level comes together and v_o comes before g_o. Also let have a bfs ordering with a segment $\{v_1, v_2, ..v_n, g_1, g_2, ..g_m\}$. For this to be a valid bfs ordering children of v_1 comes before that of v_2 i.e $\{g_1, g_2, ..g_{childcount[v_1]}\}$ must have v_1 as their as their parent. Similar argument for $v_2, v_3, ..v_n$.
We keep two pointers, i for iterating over the parents and idx for iterating over the childrens of current node i.e $order[i]$ If any of the node at index idx is not children of node at index i we set $flag$ to false.

Pseudo Code:

```
Algorithm 1: dfs(AdjList, par, vertex, parent)
Data: AdjList is adjacency list, par stores parent of each vertex
Result: Fills par array
par[vertex] = parent;
for child : AdjList[vertex] do
    if child == parent then
        continue;
    dfs(AdjList, par, child, vertex);
```

```
Algorithm 2: ValidBFS(AdjList, order)
Data: AdjList is adjacency list representation of given tree T with n nodes. order is the given bfs traverasal
Result: Check if order is one of many possible bfs traversal of given tree T
AdjList[order[0]].push(0);
par(n + 1);
dfs(AdjList, par, order[0], 0);
i = 0;
idx = 1;
flag = true;
while $i < n$ do
    sz = AdjList[order[i]].size() - 1;
    while $sz > 0$ do
        if par[order[idx]] <> order[i] then
            flag = false;
        idx = idx + 1;
        sz = sz - 1;
    i = i + 1;
return flag;
```

Time Complexity Analysis:

- Let n be number of node in the tree, m (= n - 1) be number of edges in the tree
- **Algorithm 1 :** A simple *dfs* $\Rightarrow \mathcal{O}(n+m) \Rightarrow \mathcal{O}(n)$ (discussed in class)
- **Algorithm 2 :** Calls algorithm 1 $\Rightarrow \mathcal{O}(n)$
  Inner *while* loop runs *sz* times, where *sz* is the number of childrens of node (= order[i]). Outer *while* loop runs over each vertex thus overall both the loops are equivalent to a bfs which visits each node and edges once $\Rightarrow \mathcal{O}(n+m) \Rightarrow \mathcal{O}(n)$

$$\text{Total Time Complexity} = \boldsymbol{\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)}$$