**Question 2. Chaotic Dino**

Divyansh Mathur, 210357

a) **Main idea:** The main idea is to use breadth-first search (BFS) to discover vertices starting from vertex S up to a certain distance x.

During this exploration, all towers within that distance will be added to the queue.

Then, BFS is performed by exiting the queue while checking whether the destination vertex D is reached.

*Algorithm*:

The algorithm includes the following steps:

Initialize a queue named q\_tower and list the source vertex S.

Create a visited table to keep track of visited and marked towers The source vertex is visited (visited[S ] = true).

Initializing a Boolean variable with an initial value is false.

Execute while loop when queue q\_tower is not empty.

Delete the preceding element, denoted t.

Initialize distance changed to 0.

Perform BFS from vertex t to distance x.

For each peak that has an unvisited tower, queue it up and mark it as visited.

If at any time during the BFS journey we reach destination vertex D, set ans to true.

Returns the value of the year.

This algorithm uses BFS to discover towers located within a certain distance from the source vertex and check whether the destination vertex is reachable.

If the destination vertex D is reached during exploration, ans is set to true, indicating a successful path within the given distance.

*Pseudocode:*

bool canSignalReach(int S, int D, int x, vector<vector<int>> graph, vector<bool> towers) {

queue<int> q\_tower; // Initialize a queue to track towers within the distance

vector<bool> visited(number\_of\_cities + 1, false); // Array to track visited cities, assuming city indices start from 1

visited[S] = true; // Mark the source city as visited

bool ans = false; // Initialize the answer as false

q\_tower.push(S); // Start the exploration from the source city

while (!q\_tower.empty()) {

int t = q\_tower.front(); // Get the front of the queue

q\_tower.pop(); // Remove the front element

int distance = 0; // Initialize the distance from the source city

// Create a queue for BFS from the current city

queue<int> q\_t;

q\_t.push(t);

while (distance <= x) {

int size = q\_t.size(); // Get the number of vertices at this distance

while (size > 0) {

int node = q\_t.front(); // Get the front of the BFS queue

q\_t.pop(); // Remove the front element

for (int neighbor : graph[node]) {

if (towers[neighbor] && !visited[neighbor]) {

q\_tower.push(neighbor); // Enqueue towers within the distance

visited[neighbor] = true; // Mark them as visited

if (neighbor == D) {

ans = true; // Destination city is reached

}

}

}

size--;

}

distance++;

}

}

return ans; // Return the answer

}

b.)

**Algorithm: Finding Valid Power using Binary Search**

**Input:** n (the maximum power), canSignalReach(x) function

**Output:** ans (the valid power)

**1.** Initialize right = n-1 and low = 0, as the maximum power can be n-1 in the graph.

**2.** Define ans = n-1.

**3.** While low <= high: a. Calculate mid = (low + high) / 2. b. Check if the power at mid is useful using the canSignalReach function. c. If canSignalReach(mid) returns true: - Update high to mid - 1. - Set ans = mid. d. Otherwise: - Update low to mid + 1.

**4.** Return ans as the valid power.

This algorithm finds the valid power using binary search while ensuring that the meaning remains unchanged.

*PseudoCode*

*function findMinimumPower(S, D, graph, towers):*

*low = 0*

*high = n - 1 // Assuming max power can be n-1 in the graph*

*ans = n - 1*

*while low <= high:*

*mid = (low + high) / 2*

*// Check if the signal can reach D with mid power*

*if canSignalReach(S, D, mid, graph, towers): // this function is implemented in ‘a’ part*

*ans = mid*

*high = mid - 1*

*else:*

*low = mid + 1*

*return ans*