CSF222 ASSIGNMENT

-Prof Sundaresan Raman

GROUP 27

Member 1: Sanjeev Mallick
2021A7PS2217P
Member 2: Garvit Singhal
2021A7PS2226P
Member 3: Harpreet Singh Anand
2021A7PS2416P

DOCUMENTATION

Ques 1)

A station is a junction if it is directly connected to 4 or more different stations. For each station 'i', we calculated the number of stations connected to it (by traversing through the ith row of the adjacency matrix) and stored it in a variable directConnections.
After traversing the ith row of the adjacency matrix if directConnections is greater than or equal to 4 then ith station is a junction.
To keep the count of junctions we initialized a integer variable noOfJunctions to 0, and increased it by 1 each time we found a station to be a junction.
At the end we returned noOfJunctions.

Ques 2)

The approach was to use the concept of Euler Path and Euler Circuit
Euler's Theorem:
1. If a graph has more than 2 vertices of odd degree then it has no Euler paths.
2. If a graph is connected and has 0 or exactly 2 vertices of odd degree, then it has at least one Euler path
3. If a graph is connected and has 0 vertices of odd degree, then it has at least one Euler circuit.

As the function returns a bool value we followed the below approach:

(a)Tour should end at the same station(Euler Circuit):
Step 1-We created an array degree[noOfStations] to store each station’s degree.
Step 2-For each station ‘i’ we calculated its degree and stored it in degree[i].
Euler circuit exists only when all the stations have even degree.
Step 3-So we traversed the ‘degree’ array if we found a station with even degree we returned false(that means tour is not possible).
If there is no station with degree odd then we return true(that means tous is possible).

(b)Tour should end at different station:
Step 1-We created an array degree[noOfStations] to store each station’s degree.
Step 2-For each station ‘i’ we calculated its degree and stored it in degree[i].
For such a tour there should be exactly 2 stations of degree odd and rest all should have degree even.
Step 3-So we traversed the ‘degree’ array and kept a count of the stations with odd degree in variable.
Step 4- Such a tour is possible only when the variable is 2.

Ques 3)
First Part-Warshall’s Algorithm
Step 1- We allocate memory for the closure matrix closure to store the transitive closure of the graph.A nested loop initializes the closure matrix to

initially match the adjacency matrix of the given graph. This step ensures that the direct connections between vertices are accounted for in the closure matrix.

Step 2- We update the closure matrix based on the transitive nature of connections in the graph by iterating through all the vertices k the algorithm checks if there's an indirect connection between the vertices i and j through vertex k. If such a path exists, the closure matrix is updated accordingly (closure[i][j] = closure[i][j] || (closure[i][k] && closure[k][j]);

Step 3- The function now returns the computed closure matrix.

Second Part - Finding impossible pairs

Step 1- We use the function for warshall's algorithm created above to calculate the transitive closure of the given graph

Step 2- Now we iterate through all pairs of vertices. For each pair of vertices (i,j) (such that i<j) we check if there is a path from i to j and j to i. If there's no such path, we increment the impossible_pairs variable.

Step3- We return impossible_pairs as the no. of impossible pairs in the given graph **.**

Ques 4)

The approach was to use the concept of cut edges in graph theory. A cut edge is an edge which when removed from a graph creates more components than previously in the graph.This function identifies the vital train tracks which correspond to the cut edges of the graph and calculates the number of such tracks.

Step 1- Firstly we initialize the variable ‘vital_tracks’ to 0. This will store the number of vital train tracks for the given graph.
Step 2- Check if there is a track between the current pair of cities (g->adj[i][j] == 1). If there is a track, temporarily remove it by setting g->adj[i][j] and g->adj[j][i] to 0]
Step 3 -Call the warshall function to compute the transitive closure of the graph with the temporary removal of the track. The warshall function returns a 2D matrix ‘new_closure’ representing the transitive closure.
Step 4- Restore the temporarily removed track by setting g->adj[i][j] and g->adj[j][i] back to 1 and then check if the transitive closure matrix indicates that there is no path from i to j and from j to i in the graph.
Step 5- If both conditions are true, it means removing the track would make it impossible to travel between the corresponding cities using any combination of train tracks. In this case, we increment the vital_tracks count.
Step 6- We return the final count of vital train tracks after iterating over all pairs of cities and checking the vitality of each track.

Ques 5)

We used the concepts of Bipartite graphs(analogous to colouring the graph with 2 colours such that no 2 adjacent nodes are of same colour). The question is analogous to colouring the graph with 2 colours red or blue in a bipartite manner.

Step 1-Initialize the upgrade array with -1(means no colour).
upgrade[i] : gives the colour of the ith city(-1 or 0 or 1).

Step 2-Maintain a queue(of array) using front and rear pointers to store the cities.

Step 3- Initially push any one of the cities into the queue and give it a colour(either 0 or 1).

Step 4- For each city in the queue, enqueue their adjacent city only when their adjacent city's corresponding upgrade value is -1.

Step 5-For each city 'curr_city' in the queue ,for each of their adjacent city there are 3 possibilities :

a) if(upgrade[adj_city == -1]) then enqueue the adj_city and colour it with the colour other than that of the curr_city.

b) else if(upgrade[curr_city] == upgrade[adj_city]) then return upgrade array with all entries as -1 because 2 adjacent nodes can't be of the same colour.

c) else if(upgrade[curr_city] != upgrade[adj_city]) then we won't do anything because this already means that adjacent nodes are of different colour.

Step 6- We return the updated upgrade[] array.

Ques 6)

As our graph is undirected, to find the shortest path between any 2 stations(city_x and city_y) we use the concept of breadth first search traversal.

The implementation of the bfs traversal is as follows:

Step 1- We made 2 arrays distance and visited.

distance[]: to store the distance ith city from city_x .

visited[]: to keep an track of the cities we have already visited.

Step 2-Another array of integers(we used it as a queue) is maintained to enqueue the new cities.

We maintained 2 pointers to front and rear , to keep an track of the current city and the last city enqueued respectively

Step 3-We enqueue all the adjacent cities(not previously visited) of the city pointed by the front pointer(current city on which we are on) and update the distance array.Each time we enqueue a city we increase the rear pointer.

Step 4-After this we moved the front pointer an index ahead and repeat the step till front equals rear.

Step 5-If city_y is enqueued then we return the distance[city_y] else we return -1.

Ques 7) The approach we used is that for each city we first calculated the shortest distance to every other city and added all the distances .Now whichever city has the least sum is the capital city.

Step 1- Initialize 2 variables min_sum and capital to keep the note of the min sum and capital city respectively.

Step 2- For every station ‘i’ initialize a variable sumOfDistances=0. Now traverse all the other cities and find the shortest distance from city ‘i’ by the distance(Graph *g, int city_x, int city_y) function in the QUES 6.

Step 3-Now compare the ‘sumOfDistance’ with the ‘min_sum’. If (sumOfDistance’ < min_sum) then update the ‘min_sum’ to ‘sumOfDistance’ and ‘capital’ to ‘i’.

Step 4-Return capital as answer after traversing all the stations.

Ques 8) In this question, we employ concepts related to the Hamiltonian path problem. A Hamiltonian path is a type of path that visits each vertex of a graph exactly once, and it's similar to what the Maharaja Express tour seeks—a path that visits each city exactly once before returning to the starting city. We use the DFS(Depth First Search) algorithm to explore the path that the maharaja express can take.

Helper Function(maharaja_express_tour)

Step 1) The function explores the graph using a recursive depth-first search approach to determine if a Maharaja Express tour is possible from a specific source city. visited array keeps track of visited cities.

Step 2) For a given current_city,we check all adjacent cities using the adjacency matrix (g->adj[current_city][i] == 1). It ensures that the algorithm doesn't revisit the previous city (i != previous_city) and doesn’t immediately return to the source city (i != source).

Step 3) If an unvisited adjacent city meets the criteria, the function makes a recursive call to explore the tour from that city.

Base Cases: If the adjacent city is the source city and it's not the previous_city, it means a tour is possible, so it returns true. If a city has already been visited, it skips exploring that city further.Hence the function returns true if a valid Maharaja Express tour is found, otherwise false.

Second Function(maharaja_express)

Step 1) We allocate memory for the visited array to keep track of visited cities.

Step 2) We initialize the visited array to mark all cities as unvisited.

Step 3) Call the maharaja_express_tour function to check if a Maharaja Express tour is possible starting from the given source city and store the output of this function(boolean) in a boolean variable .

Step 4) Free the memory allocated for the visited array and return true if a Maharaja Express tour is possible from the specified source city, otherwise false.