Assignment 2 - CT3532 Database Systems 2

Cathal Lawlor - 21325456 | Liam Holland - 21386331

1. **Suggest a way of representing this graph in a relational database.**

We believe the best way to represent a graph in a relational database would be to have a table containing tuples representing the edges of the team's graph at a certain snapshot. Each tuple has 4 attributes, the snapshot number, node_1, node _2, distance. We believe it would be best to use the player id of each player for the node numbers here.

The following constraints should also be in place:

- Each edge is undirected, where there must be only one entry per edge.
- node _1 must always smaller than node_2

With these constraints on the database, as each player is entered, the number of edges to enter for the next player's node reduces by one, reducing the space requirement overall for the database. I.e., when you store 1->2 you negate the need to store 2->1

If you had 3 nodes, for example, it would look like the following:

Snapshot No.	Node_1	Node_2	Distance
1	1	2	5
1	1	3	13
1	2	3	10

We only need to store three tuples to represent the entire graph.

2. **Suggest a suitable means to represent the data in a data structure.**

To store the data in a data structure, an adjacency matrix of the players could be used, where the distance is the value in the matrix of two players. This allows simple and rapid look ups where each datapoint is at a known index, with no calculation required. It is well suited as we know it is a fixed size. Even if the system needed to expand to account for substitutions, all you would have to do is store a different player id in the tuples relating to a certain snapshot.

Sample: if you look at column 2, and match it to row 5, it means player 2 is 3 units away from player 5.

PlayerID	1	2	3	4	5	6	7	8	9	10	11
1	0	5	56	23	40						
2	5	0	12	5	3						
3	56	12	0	7	10						
4	23	5	7	0	14						
5	40	3	10	14	0						
6						etc					
7											
8											
9											
10											
11											

The only disadvantage here is that only half of the matrix is necessary to store all of the data. As you can see above, the values will repeat each other on each side of the diagonals (which are 0). There are two options here: fill in both sides and have redundant data or leave half of the matrix empty and have unused space. This problem arises because the graph is undirected, so the edge has the same value in both directions. Still, the advantages of using an adjacency matrix outweigh the disadvantages, especially here, where the 2D array is only 11x11.

3. **Suggest an algorithm to measure the similarity of two graphs.**

Again, we can make use of adjacency matrices here. Comparing the two matrices constructed on different snapshots by subtracting graph_2, from graph_1 will give us a matrix of the "difference" between the graphs. The percentage difference between the graphs can then be found by dividing the absolute value of each entry in the matrix by the maximum possible distance a player could move in one snapshot. At the most basic level, this would just be the distance from one corner of the pitch to the diagonal opposite corner, which is the value we decided to use for our algorithm.

This is not perfectly accurate, as if the player is located in the middle of the pitch at one snapshot, the maximum distance they could move is actually the pitch diagonal divided by 2. However, we decided not to consider exact player position for this assignment, but we are still satisfied that you will obtain a satisfactory result on how similar the two graphs are; if the similarity is 0%, it would be because the graphs could not be more different, no matter what.

Finding the average percentage in the matrix will give us the percentage difference between the two graphs, so to find the percentage similarity, we simply subtract the percentage from 100%.

The equation will look like this:

$$1 - \frac{\sum_{i=0}^{11} \sum_{j=0}^{11} |(G_1[i][j]) - (G_2[i][j])\,|}{121 \times \max\ \_distance}$$

Where G_1 and G_2 are the two adjacency matrices representing the graphs.

It is also important to note that the algorithm only has to run over half of the matrix, as both halves are identical.

4. **Consider the following constraint; if the distance between two players is less than k, keep the edge, else discard the edge.**
   - Given sample data, write code/pseudo code to calculate the degree of each node for any snapshot (a given timestamp).
   - Given sample data, write code/pseudo code to determine which node(s) is on the most paths?

Part 1

The first part of this question is relatively simple. To measure the degree of each node, given an adjacency matrix where the operation to remove edges with a value less than k has already been completed, we can simply iterate over each row of the matrix and count the number of values that are greater than 0. This is assuming that when performing the operation to remove edges, you did it for the entire matrix, not just the top half, for efficiency. If only the top half of the array was operated on, you would have to increment the degree not only for the node relating to the row you are on, but also for the node of the column you are on. This is because you would need to count the edges connected to that node which are not recorded further down in the matrix.

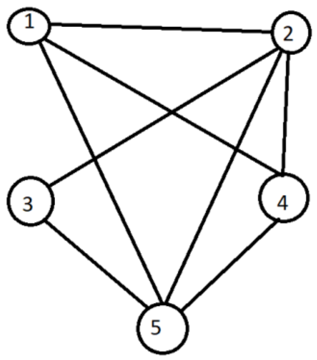
For the code implementation of this, we used python to create a sample matrix, where we took the value of *k* to be 3, resulting in values in the matrix of 0 (no edge), 1 or 2 (edge). Both implementations of the algorithm are below, written in Python:

This is the sample data we are using for this question:

```
#adjacency matrix for a graph of size 4, where k = 3
graph = [
    [0, 2, 0, 1, 1],
    [2, 0, 1, 1, 1],
    [0, 1, 0, 0, 1],
    [1, 1, 0, 0, 1],
    [1, 1, 1, 1, 0]
]
```

We opted to use a 5x5 matrix for the sake of simplicity in testing our code. The code will scale up to an 11x11 matrix with no changes.

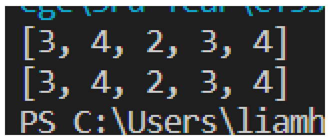
Here is a sketch of what that graph looks like, with numbered vertices, the weights of the edges are excluded, as they are not relevant to the operations required in this question: ---->



The code to find the degree of each node is as follows:

```
#array for the degree of each node
degrees_all = [0, 0, 0, 0, 0]

#for to top half of the rows
for i in range(0, len(graph)):
    #for the second half of row i
    for j in range(0, len(graph)):
        #if there is an edge between those two nodes
        if graph[i][j] > 0:
            degrees_all[i] += 1

degrees_half = [0, 0, 0, 0, 0]

#for each row
for i in range(0, len(graph)):
    #for the second half of row i
    for j in range(i, len(graph)):
        #if there is an edge between those two nodes
        if graph[i][j] > 0:
            degrees_half[i] += 1
            degrees_half[j] += 1

print(degrees_all)
print(degrees_half)
```

This code will return the degree of each node, which will work for both implementations of finding which edges to keep and discard:



```
[3, 4, 2, 3, 4]
[3, 4, 2, 3, 4]
PS C:\Users\liamh
```

Part 2

For the second part of this question, we need to count the number of times each node appears on a path. In order to do this, we will need to traverse every path in the graph.

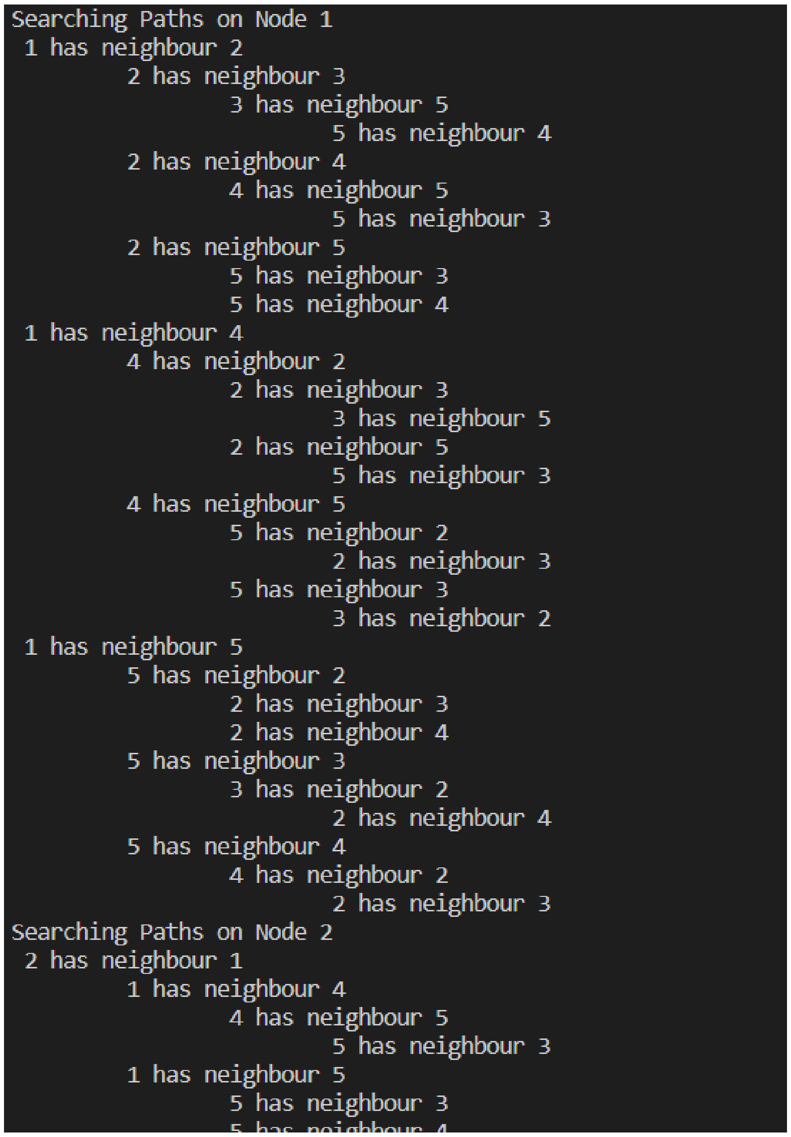
All we do is use a depth-first search algorithm on each node, travelling to each node connected to that node as long as it has not already been visited. By doing this, we will end up traversing every path in the graph. We can then simply employ a frequency array to allow us to find the most frequently visited node in the graph.

The disadvantage of this approach it that it is necessary to traverse the entire matrix. While this is not a major issue due to the fact that the matrix will only ever be 11x11, it requires that the operation to discard edges if they are greater than *k* be done on the entire matrix, not just half of it. It is also an expensive operation to find all of the paths in the graph, however, there are few ways in which to find all of the paths in a graph faster than simply traversing every single one. Additionally, we believe the fact the matrix will never exceed a size of 11x11 means the operation should never take too long.

In Python, that would look like the following (using the sample data included above):

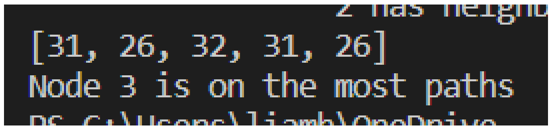
```python
#find the number of times each node was visited by on a path

#frequency array for the number of times each node is visited
#initialised to -1 because otherwise the algorithm will count the initial starting node for
the paths each node appears on
times_visited = [-1, -1, -1, -1, -1]

#depth-first search on the graph
def search(node, visited_nodes, numtabs):
    visited_nodes.add(node) #node has been visited
    times_visited[node] += 1    #node was visited again

    #for each neighbour node
    for i in range(0, len(graph)):
        if graph[node][i] > 0 and i not in visited_nodes:
            tabs = '\t' * numtabs
            print(f"{tabs} {node + 1} has neighbour {i + 1}")
            search(i, visited_nodes.copy(), numtabs + 1)

#search each node in the top half of the rows in the matrix
for i in range(0, len(graph)):
    print(f"Searching Paths on Node {i + 1}")
    search(i, set(), 0)

print(times_visited)
print(f"Node {times_visited.index(max(times_visited)) + 1} is on the most paths")
```

The output is as follows:



```
Searching Paths on Node 1
 1 has neighbour 2
         2 has neighbour 3
                 3 has neighbour 5
                         5 has neighbour 4
         2 has neighbour 4
                 4 has neighbour 5
                         5 has neighbour 3
         2 has neighbour 5
                 5 has neighbour 3
                 5 has neighbour 4
 1 has neighbour 4
         4 has neighbour 2
                 2 has neighbour 3
                         3 has neighbour 5
                 2 has neighbour 5
                         5 has neighbour 3
         4 has neighbour 5
                 5 has neighbour 2
                         2 has neighbour 3
                 5 has neighbour 3
                         3 has neighbour 2
 1 has neighbour 5
         5 has neighbour 2
                 2 has neighbour 3
                 2 has neighbour 4
         5 has neighbour 3
                 3 has neighbour 2
                         2 has neighbour 4
         5 has neighbour 4
                 4 has neighbour 2
                         2 has neighbour 3
Searching Paths on Node 2
 2 has neighbour 1
         1 has neighbour 4
                 4 has neighbour 5
                         5 has neighbour 3
         1 has neighbour 5
                 5 has neighbour 3
                 5 has neighbour 4
```

Etc. until the end result is obtained:



```
                 2 has neighb
 [31, 26, 32, 31, 26]
 Node 3 is on the most paths
 PS C:\Users\liamb\OneDrive
```