CT3532 Database Systems 2

Assignment 3: Graphs

Name: Maxwell Maia   Student ID: 21236277

---

1. A way of representing the graph in a relational database.

Strategy – Create two tables: one for nodes (players) one for edges (distances).

In every game, graphs will be generated every few minutes possibly. This leads to a lot of graphs per game. To assist in keeping our database organized we will store one Nodes table and one Edges table per game.

There will be many samples within a game. These can be uniquely discerned using the timestamp attribute.

This solution will create edges for each player between teammates and the opposing team.

The graph is undirected.

Note: the naming schema of the table could be done in any way. This proposed solution says that the all the games in a league are numbered from game number 1 to the final game. The team names are included in the name of the table for clarity.

<u>The Nodes table:</u>

Nodes_Game_1_Liverpool_ManCity

playerId	playerName	teamId
11	Messi	1
63	Ronaldo	3

playerId - Unique ID for each player.

playerName - Name of the player.

teamId - ID for the team to which the player belongs.

Additional fields about the player may be included here.

A foreign key to the player ID will be included in the Edge's table. This makes it easy to query information about the players in the graph.

The Edges table:

Edges_Game_1_Liverpool_ManCity

edgeId	player1Id	player2Id	distance	timestamp
1	11	63	21.40	1

Let's say that a graph is made every minute in a game starting from minute 0.

Each graph will have multiple edges in the Edges table, edges are identifiable by their edge ID.

Note: The graph is undirected, so when inserting into the table, the algorithm that inserts the edges iterates through the players of only 1 team, finding the edges between that player and everyone else in the game. The algorithm will skip creating edges of players with themselves and also skip creating edges with players that already have an edge with them. This keeps the graph undirected and saves storage space and computational effort.

Player 1 and 2 ID are foreign keys to the Nodes table.

Distance is a real number measure of distance between players in metres.

Timestamp is the time that the edge was recorded.

This structure allows for flexibility and efficiency in handling both player information and dynamic graph data. The timestamp in the edges table enables tracking changes over time.

2. Representing the data in a data structure.

For the edges of each timestamp:

Use an adjacency matrix to represent the graph.

An adjacency matrix is a 2D array where each element (i, j) represents the presence or absence of an edge between nodes i and j. This is a weighted graph so the matrix stores the distance between nodes instead of just 0 or 1.

This data structure is chosen because it allows for time efficient representation and manipulation of dense graphs. Our graph is dense because every player in the game could have an edge with every other player in the game. This means that an adjacency matrix is a memory efficient data structure for storing the edge information. Furthermore, an adjacency matrix allows for fast edge queries: for example: checking for the existence of an edge between two players can be done in constant time O(1). This is because you can directly access the matrix entry corresponding to the two players.

Here is an example of one of these adjacency matrices. This example represents the first samples of Game_1. This data comes from the tuples in the table "Edges_Game_1_Liverpool_ManCity" where timestamp = 1.

The entry at position (i, j), in the matrix represents the connection (edge) between player i and player j. Player ID's along the top and left of the matrix for a visual representation.

	1	2	3	4	5	6	7	8	9	1	11	56	57	58	59	60	61	62	63	64	65	66
1																						
2																						
3																						
4																						
5																						
6																						
7																						
8																						
9																						
10																						
11																			21.40			
56																						
57																						
58																						
59																						
60																						
61																						
62																						
63																						
64																						
65																						
66																						

Note that the graph is undirected and so the matrix is symmetric so only half of the graph could be stored for efficiency.

There should be values at every cell in the upper half of the matrix indicating the distance of the edges (choosing to not store the lower half of symmetrical matrix). The only edge in the edge table above is represented in the matrix above.

<u>For nodes:</u>

Player information is stored in a separate list where each index corresponds to a node. This makes it easy to retrieve player details by index.

3. An algorithm to measure the similarity of two graphs.

The graph is weighted, so a sensible approach to calculating similarity would be to use the Cosine Similarity algorithm.

The algorithm leverages the idea of representing each graph as a vector in a high-dimensional space, where each dimension corresponds to an edge in the graph, and the value along that dimension is the weight of the edge.

By calculating the cosine of the angle between these vectors, you obtain a measure of similarity that takes into account both the presence and magnitudes (weights) of edges in the graphs.

4. If the distance between two players is less* than k, keep the edge, else discard the edge.

*Correction made by lecturer, according to fellow student.

4.1. Pseudo code to calculate the degree of each node given a graph.

The following pseudo code iterates over all unique pairs of nodes, checks the distance in the adjacency matrix, and increments the degree of both nodes if the distance is less than k. The result is an array containing the degree of each node.

```
function initialize_degree_array(length):
    # Create an array of length initialized with zeros
    degree = new Array(length)
    for i = 0 to length - 1:
        degree[i] = 0
    return degree

function calculate_degree_with_constraint(nodes, adjacency_matrix, k):
    # Initialize an array to store the degree of each node
    degree = initialize_degree_array(nodes.length)

    # Iterate over all edges in the adjacency matrix
    for i = 0 to nodes.length - 2:
        for j = i + 1 to nodes.length - 1:
            distance = adjacency_matrix[i][j]
            # Check the constraint: if the distance is less than k, keep the edge
            if distance < k:
              degree[i] += 1
              degree[j] += 1

    return degree
```

4.2. Pseudo code to determine which node/nodes is/are on the most paths.

The following pseudo code iterates over all unique pairs of nodes, checks the distance in the adjacency matrix, and counts the edge if the distance is less than k. Then, it identifies the node/nodes with the highest count, indicating that they are on the most paths and returns them.

```
function initialize_node_count_dict(length):
  # Create an empty dictionary to store the count of each node
  node_count = new Dictionary()

  # Initialize counts for each node to zero
  for i = 0 to length - 1:
    node_count[i] = 0

  return node_count

function nodes_on_most_paths_with_constraint(degree, adjacency_matrix, k):
  # Initialize a dictionary to store the count of each node
  node_count = initialize_node_count_dict(degree.length)

  # Iterate over all edges in the adjacency matrix
  for i = 0 to degree.length - 2:
    for j = i + 1 to degree.length - 1:
      distance = adjacency_matrix[i][j]
```

```
        # Check the constraint: if the distance is less than k, count the edge
        if distance < k:
            node_count[i] += 1
            node_count[j] += 1

# Find the node(s) with the highest count
max_count = 0
for each count in node_count.values():
    if count > max_count:
        max_count = count

# Initialize a list to store nodes on most paths
nodes_on_most_paths = empty list

# Iterate over each node in node_count
for each node, count in node_count:
    if count == max_count:
        add node to nodes_on_most_paths

return nodes_on_most_paths
```