Algorithms & Analysis – Assignment 1: Closest Associates

1. Introduction & Assignment Overview

With social media as an example, people communicate to other people residing within the network creating a connection. People within the network can then communicate to many other people within the network and those people also communicate with more different people creating a web of connections. This web of connections or network can be represented as a directed graph and presented within a data structure. These data structures will interpret the people as vertices and their connections as edges, and their overall structure can vary.

Using skeleton code provided, the assignment is to implement two data structures for a network and analyse their overall performance in terms of time complexity when performing tasks for three different scenarios. Then judge which is the better data structure

The data structures that needed to be implemented were an adjacency list and incidence matrix. These data structures are intended to store directed graphs where the direction of the edge matters and has weight. They should also be designed to enable the adding or vertices, adding of edges, updating of edges, removing vertices and removing edges. In addition, these data structures should also be capable of returning certain values regarding the data it holds. These returns are: the weight of a given edge, all the vertices the given vertex connects towards, all vertices that connect to the given vertex, all vertices and all edges.

Adjacency list data structures store the edges in a table, rows being vertices and columns being edges with their connected to vertex and the connections weight. For clarity a table was created as an example on how the adjacency list stores the graph. Each row holding a unique vertex, and with each connection that the unique vertex connects to, a column is used holding the vertex it connects towards and the weight towards it. The size the column in each row can vary depending on the amount of edges it holds, as shown by the blank points in the table.

Vertex	Edge 1	Edge 2
A	B, 3	-
B	A, 1	C, 8
C	A,2	-

Visual representation of how an adjacency list stores data

Vertex	A->B	B->A	B->C	C->A
A	3	-1	0	2
B	-3	1	8	0
C	0	0	-8	-2

Visual representation of how an incidence matrix stores data

An incidence matrix stores data in a table too, rows being unique vertices, and each column being a unique edge. Each point only holds the value the vertex has for the edge. If the vertex is the connector it holds the positive weight, but if it's the connected it holds the negative weight, and if it is neither it holds 0. This ensures all points in the table hold a value, regardless of whether the row vertex is involved with the particular edge.

2. Data Generation & Experiment Setup

2.1. Experiment Setup

To test the implementations time complexity an experiment was designed and performed that attempts to mimic the pseudo randomness of real-world network. Tests were conducted multiple times on three graph densities and was done over three unique scenario tests for each adjacency list and incidence matrix data structures.

2.2. Data Generation

Data for the test was created, using implemented code that generates a .csv file with similar format to the assocGraph.csv file provided. This code uses an argument where the user passes in a desired density, and the code returns a .csv file of edges and weights.

The overall data generation code is intended to mimic the size and randomness of a real-world network. Firstly, a large random number of vertices is chosen between 1900 and 2000 to generate. A large but small relative random number is used to simulate the size and small randomness of real-world networks. An edge amount is then calculated using a given formula and the passed in desired density and number of vertices. The program then proceeds to create unique edges of pseudo random weights of max 25 until the amount of edges to reach a given density has been reached.

Pseudocode for data generation:

```
Input: density
Output: .csv file
Vertices[] = a random amount between 1900 and 2000 of random vertices
Edges = of size density * vertices amount all squared
WHILE amount of edges not reached
    Source vertex = next value in vertices
    Amount of edges = random number of edges for the vertex to have
    WHILE amount of edges not reached
        Target vertex = random value in vertices that has not been chosen for target
        APPEND edges with source vertex, target vertex, random weight
    ENDWHILE
ENDWHILE
WRITE all edges into the .csv file
```

Data for this particular experiment was created using densities of 0.001, 0.0025 and 0.005. This low density is due to the data structures being unable to hold such a large number of edges for the number of vertices being used to mimic real-world networks.

In relation to the overall test structure, this code was used to create 27 graphs. This is due to the three test scenarios using three densities and each density using three graphs to test on and get the average time for a given test. This average is needed due to the fluctuation in processing speeds for a CPU. The decision to create a new graph for each test was to find the averages as it is possible that one generated graph is simple in difficulty.

2.3. Test Scenario Generation

The scenarios that were required to be tested were for shrinking graphs, finding nearest neighbours and changing associations. Each of these scenarios test the length of time it takes both data structures to complete a necessary operations for the scenario. Shrinking graphs is intended to test the removals of vertices and edges within the data structure. Finding nearest neighbours are designed to test the speed of which the data structure searches for all neighbours a vertex is connects to and is connected to. And the change in association is to test the speed of which the data structure is able to locate an edge and change its weight.

In relation to the design of the experiment a new test needs to be generated for each test. This is due a new graph being used for each test with the reason previously stated. And as each graph has a unique set of vertices, a new test needs to be created for that unique set of vertices.

Generation of the unique tests for a given graph and scenario was accomplished with an implementation of a test generation program. This program takes in the scenario number and graph to create a test for, then generates a .in file similar structure to the tests.in files provided with lists of operations that call the relevant operations of size 150 to test the data structures.

Pseudocode for removal:

```
Input: graph
Output: test.in file
Operations[] = array to hold operations
Source vertices[] = array of source vertices in graph
Target vertices[] = array of target vertices in graph
WHILE amount of operations not reached
    Remove = random index for vertex or edge that has not been removed
    IF removing edge
        APPEND edge with operation to remove the edge
    ELSEIF removing vertex
        APPEND edge with operation to remove vertex
ENDWHILE
WRITE all operations to .in file
```

For the removal scenario, the program generates operations that remove edges then vertices. The program collects all edges in the graph, it then picks a random edge then generates a string operation that should remove the edge. The program then gets a list of unique source vertices then randomly picks a vertex and writes the relevant operation to remove it.

Pseudocode for nearest neighbours:

```
Input: graph
Output: test.in file
Unique source vertices[] = list of all unique source vertices
Unique target vertices[] = list of all unique target vertices
Operations = list of operations
WHILE amount of operation is not reached for in
        Vertex = random unique value in vertex array
        IF vertex has not been selected already
                APPEND operation for in neighbours with -1 and vertex
        ENDIF
ENDWHILE
WHILE amount of operation is not reached for out
        Vertex = random unique value in vertex array
        IF vertex has not been selected already
                APPEND operation for out neighbours with -1 and vertex
        ENDIF
ENDWHILE
WRITE all operations to .in file
```

For the nearest neighbours scenario, half the operations generated will be of random in neighbours for a vertex, and the other half being of random out neighbours for a vertex. For out neighbours the program selects a vertex from the unique source vertex array. If the selected vertex has not been selected to be used for an out operation already then it appends to the operation the vertex and -1 to test as a worst case. The same process is done for the in neighbours, but instead the unique target vertices array is used to get values from. It can be seen that unique source and target vertices are stored, and this is to ensure no vertex takes priority over another if they have more edges stored in the graph.

The final scenario is the association changes scenario which is performed by updating random edges to a new weight. This code is performed by reading each source and target vertex columns in the graphs into an array with the indexes being used to pair the source and target vertex into edges. A random edge is then selected, and the update operation is performed by using the edge and selecting a new pseudorandom number.

Pseudocode for changing associates:
Input: graph
Output: test.in file
Source vertices[] = array of all source vertices
Target vertices[] = array of all target vertices
Operations = list of operations
WHILE amount of operations is not reached
    Vertex = random index to get pair of source and target vertices
    APPEND to operations the relevant source and target vertex with random weight
ENDWHILE
WRITE all operations to .in file

2.4. Conducting the test

Using the GraphEval file provided as a base code, a new program was created derived from the GraphEval code in order to test the implementation. The new GraphEvalTaskB code used the code that read from the input file using "-f" in order to input the graph data into the data structure, then read from the scanner using the test file as input and recording all the operations into an array to be tested later. This is all read and stored in the programs memory to avoid factors like the latency in reading from an external file when timing the operations.

Once all the mentioned preparations were completed, the test was conducted with the use of the System.nanoTime() function which returned the current systems time in nano seconds to time the test. The use of nano time ensures for precise time recordings from the start to end of the test. Before the loop began to iterate through the operations, the start time was recorded, and once the program finished looping through the operations the end time was recorded. This start and end time was used to calculate the overall time it took to run all the operations in the test file, concluding the test for that graph. This test was run for all pairs of graphs and tests generated and all results were stored in excel file Scenario Test Results.

3. Evaluation of Data Structures

3.1. Removals scenario

Evaluating the theoretical time complexities for both data structures in the implementations, it was expected that the adjacency list would take longer to complete the removal tests. Theoretically, removing edges from the adjacency lists would give a time complexity of O(k), k being the number of edges in the vertex, and the incidence matrix would have a time complexity of O(1) as it is able to locate the edge immediately. When removing of vertices, both structures have time complexity O(E) due to their requirement to remove all edges where the vertex is being connected to. Accounted separately, removing edges and vertices, the adjacency

list would be considered faster due to the time complexity of removing edges being faster. But as they are accounted together, both have a time complexity of O(E) as it is the largest time complexity between both removal of edges and vertex operations hence, they should not necessarily have the same test duration but the same rate of change over densities.

Referring to the results from the tests, the results support these expectations that the data structures would have similar rates of changes. Seeing as the rate of change between the lowest to highest tested density in the adjacency list averages to 50% while the incidence matrix is 53%, it shows that they do change at a roughly similar rate. Having this similar rate of change shows that it has a similar if not the same time complexity. Results also show that the adjacency matrix is considerably faster overall at the tested densities, which would probably also be a result of unforeseen and unaccounted for implementation code.

Adjacency List				Incidence Matrix			
Scenario 1: Removals				Scenario 1: Removals			
Density	**0.001**	**0.0025**	**0.005**	**Density**	**0.001**	**0.0025**	**0.005**
Trial 1	0.0213948	0.03059	0.060333	**Trial 1**	0.44378	0.68242	0.90224
Trial 2	0.0266301	0.031388	0.043813	**Trial 2**	0.382588	0.552973	0.778154
Trial 3	0.0222147	0.032712	0.055212	**Trial 3**	0.211702	0.51403	0.851607
Avg Time	0.0234132	0.031563	0.053119	**Avg Time**	0.33269	0.583141	0.844

3.2. Nearest neighbours scenario

Reviewing the implementations, it is expected that the incidence matrix would be quicker in terms of searching for nearest in and out neighbours. Within the adjacency list and incidence matrix, the time complexity for returning all in/out neighbours of a given vertex was O(V) due to the need to iterate through each vertex. For the Incidence matrix it is expected that the time complexity for in/out neighbours would be O(E) as it needs to iterate through each edge too to find all neighbours. With this in mind, knowing that the data generated contained a fairly large number of vertices but has a larger changing number of edges as the density scales, the adjacency list should have a higher test duration due to the large amount of vertices, but the incidence matrix should have a larger rate of change due to the increase in edges.

Looking towards the results, the hypothesis that the adjacency list has a higher duration, but slower rate of change proves true. In the results the adjacency list starts at an average of 12.59 seconds for the given density, but the incidence matrix starts at an average of 0.05 seconds. Although, looking at the increase of time over the densities the adjacency list barely increased with 5% meanwhile the incidence matrix increases by 290%. By ignoring the magnitude of time, it can be seen that as densities grow, the time of the incidence matrix grows much larger than that of the adjacency list. This scaling henceforth proves the incidence matrix has a larger time complexity but starts at a lower time duration.

Adjacency List				Incidence Matrix			
Scenario 2: Neighbours				Scenario 2: Neighbours			
Density	**0.001**	**0.0025**	**0.005**	**Density**	**0.001**	**0.0025**	**0.005**
Trial 1	12.721524	13.41532	13.58525	**Trial 1**	0.05443	0.143111	0.191856
Trial 2	12.774858	13.50095	13.35025	**Trial 2**	0.046095	0.110249	0.167679
Trial 3	12.286923	13.14611	14.11984	**Trial 3**	0.04447	0.125836	0.202904
Avg Time	12.594435	13.35413	13.68511	**Avg Time**	0.048332	0.126399	0.18748

3.3. Changing associations scenario

From the implementation it is predicted that the adjacency list should have an overall lower magnitude in time, although the time complexity while not necessarily higher than that of the incidence matrix, will scale faster with the given tests due to the data used. The implementation of the adjacency list is able to locate the given target vertex immediately, although must iterate through each edge contained within that row. This gives the adjacency list a time complexity of O(k) with k being the amount of edges within the vertex. Meanwhile the incidence matrix after every update, needs to iterate over each row vertex within the matrix to update their values hence giving a time complexity of 0(V). Seeing this, the results would be similar to scenario two, with the incidence matrix having larger overall magnitude of time but scaling slower due to the density changing the number of edges but not vertices. Meanwhile, as an increase in edges results in a possibly, but necessarily higher amount of edges in a given vertex, the adjacency list would scale faster than the incidence matrix over an increasing density.

With the results of the scenario test, the hypothesis given that the adjacency list being faster in terms of time, scales faster with increasing density that that of the incidence matrix was proven correct. While the average time at density 0.005 for adjacency matrix is 0.0009 seconds, the incidence matrix is 0.0665 hence proving that the incidence matrix would have a higher test duration due to large number of vertices testes. Although calculating the change in time from the lowest to highest density tests, the test duration for adjacency list only increased by 67% while the incidence matrix only increases by 14%. With this information it is clear that the time complexity of the adjacency list is higher but starts at a lower time duration.

Adjacency List				Incidence Matrix			
Scenario 3: Update Weights				Scenario 3: Update Weights			
Density	**0.001**	**0.0025**	**0.005**	**Density**	**0.001**	**0.0025**	**0.005**
Trial 1	5.64E-04	6.25E-04	8.79E-04	**Trial 1**	0.05761	0.061814	0.064999
Trial 2	5.93E-04	6.67E-04	9.77E-04	**Trial 2**	0.057924	0.063287	0.067259
Trial 3	5.09E-04	6.23E-04	9.09E-04	**Trial 3**	0.059576	0.063507	0.067077
Avg Time	0.0005552	0.000639	0.000922	**Avg Time**	0.05837	0.062869	0.066445

3.4. General overview of tests

Each scenario displayed overall varying results from other scenarios, displaying the positives and negatives of picking and incidence matrix or adjacency list.

Overall, depending on how the data is going to be used and the amount of data being held, either adjacency list or incidence matrix can be used. If overall speed is desired when constantly changing and manipulating the data in a graph, then an adjacency list is better to be used due to the low test durations when updating and removing vertices and edges. Although it must also be noted that as the amount of edges for a vertex increases to a considerable amount, the adjacency list would be slower as the time complexity is higher than the incidence matrix.

If the use of the data is merely to read and obtain data, the incidence matrix would be faster. This is due to the lower test times when testing for nearest neighbours, but the incidence matrix for this use would also become slower over a larger graph with many more vertices as the time complexity is also higher than that of the adjacency list.

4. Recommendations & Analysis Overview

Due to the background of the entire assignment to analyse and use data structures to hold data for networks, particularly large social media networks, it is presumed that the number of nodes and henceforth vertices would be incredibly large. It is recommended that the adjacency list be used due to the large graph size to be used for social media.

This choice is due to out of the three scenarios tested, the adjacency list had a better overall performance for larger graphs. For the nearest neighbours scenario, the adjacency proved to be slower, but this was only the case for smaller densities, which the adjacency list would not be used due to the graph being for social media. For removals, the time complexities were expected to be the same and was proven, although it is also shown that the adjacency list starts at a lower time duration hence making it the better option for lower and higher densities.

The only negative test for the adjacency matrix was when updating associate weights. Despite the adjacency list starting at a lower test time duration, it was clear that it had a significantly higher time complexity supporting the hypothesis and making updating associate weights when handling a larger number of nodes.

As the positive scenarios outweigh the negative scenarios regarding the adjacency list at larger densities, it would be the preferred choice. Although, if it is found that a large amount of operations to change associations is being used by the social media platform, then it can be argued that the incidence matrix can be the better choice as the data structure.