Closest Associates Analysis Report

by Jing Li (s3676458) and Dili Wu (s3649332)

Data Generation

A dedicated data generator class (`GraphGenerator.java`) was created to facilitate graph generation and scenario evaluation. With the number of vertices N supplied, the generator creates a list of integers with 1 to N as the elements, and then shuffles the elements. This is the list of all vertices the graph has.

To grow the graph to the desired density, the generator will first add all the elements from the vertex list to the graph (Adjacency List or Incidence Matrix), and then add edges to the graph until the intended density is reached.

Graph Density:

Graph density can be defined as the number of edges over the number of vertices squared:

$$\frac{M}{N^2}$$

where M is the number of edges and N is the number of vertices. Also, for directed graphs, the **maximum number of edges** can be expressed as $N(N-1)$, as each vertex can form an edge with all other $(N-1)$ vertices. Since this project is to research social networks where each edge is a relationship between two people. Self-loop doesn't make much sense and therefore ignored from calculating total number of edges. The maximum density for a graph is hence:

$$\frac{N(N-1)}{N^2}$$

We then know that the range of the density is $[0, 1]$. Density is 0 when there is no edge at all, and density is 1 when the number of vertices approaches infinity and all the vertices form edges with all other vertices.

Then for the graph generator, it will calculates the maximum number of edges Mm given the number of vertices, and then multiply it with the desired density (a decimal number between 0 and 1) to get the number of edges the graph should have. For example, a graph with 10 vertices can have at most 90 edges ($10 \times (10-1)$), and to form a graph with a density of 0.5, there should be 45 edges (90×0.5).

The generator loops through the list of vertices and for each vertex, it will form edges with all other vertices in the same vertex list. The weight for each edge is a randomly generated integer between 1 and 100. Every time a new edge is added, a counter will increase by 1. The loops end when the counter reaches the desired number of edges. This will grow a graph to the desired density. The following pseudo code is a simple illustration of the generation process:

```
numOfAddedEdges -> 0;
numOfEdges -> MaxNumEdges * Density;
outer loop:
    for i = 0; i < vertexList.length; i++ {
        srcVert -> vertexList[i];
        inner loop:
            for j -> 0; j < vertexList.length; j++) {
                if (numOfAddedEdges == numOfEdges) break outer loop;
                if (j == i) continue inner loop;
                tarVert -> vertexList[j];
                weight -> Random(1, 100);
                graph.addEdge(srcVert, tarVert, weight);
                numOfAddedEdges++;
            }
    }
```

Experiments

For this project, we want to test the performance of the two graph data structures, **adjacency list** and **incidence matrix**, under three scenarios with varied densities. To create the graphs to test, we decided to have **1,000** vertices and then vary the density by adding different numbers of edges.

To vary the density, we defined three types of densities: **High** (0.3), **Medium** (0.1), and **Low** (0.05). Although the theoretical range of the density is [0, 1], due to the limitations of our computing power, too high a density would cause the experiments to run a long time or even face `Out of Memory` exception, since 1,000 vertices can have a maximum of whopping 999,000 edges. Therefore, we arrived at those three numbers as they are not extremely high to cause memory or time issues, and they are spread out and large enough to create significant experimental results.

Scenarios:

The three scenarios are **Shrinking Graph**, **Nearest Neighbors**, and **Changing Associations**. For each of the scenarios, we tested the two graph structures with the three different densities for 5 iterations and took the average running time of those 5 iterations to do the analysis. Below are the descriptions of the three scenarios:

Shrinking Graph

For each of the two data structures with a given density, we run the following operations and measure the running time in milliseconds:

- Remove all the vertices
- Remove all the edges

Note: because both of the above operations destroy the graph structure, we create a new same graph before each of the operations.

Nearest Neighbors

For each of the two data structures with a given density, we run the following operations and measure the running time in milliseconds:

- Get all in-neighbors for all the vertices
- Get 5 nearest in-neighbors for all the vertices
- Get 999 nearest in-neighbors for all the vertices
- Get all out-neighbors for all the vertices
- Get 5 nearest out-neighbors for all the vertices
- Get 999 nearest out-neighbors for all the vertices

Changing Associations

For each of the two data structures with a given density, we run the following operations and measure the running time in milliseconds:

- Update all the edges with new randomly generated weights

Note: the new weights are all randomly generated integers between 1 and 100, when the data is large enough like in our case, we can observe both increases and decreases to the original weights.

Running Experiments:

For each iteration of the above scenarios, each data structure generates 9 results per density. We have run the scenarios 5 times and calculated the average of those 5 iterations to get a more accurate overall picture of the performances.

Analysis of Results

All the results below are running time in milliseconds.

Scenario: Shrinking Graph

Remove Vertices Average Running Time:

	Adjacency List	Incidence Matrix
High Density	423.13	3,632.49
Medium Density	28.12	412.57
Low Density	6.99	98.73

Remove Edges Average Running Time:

	Adjacency List	Incidence Matrix
High Density	0.2	0.39
Medium Density	0.03	0.11
Low Density	0.02	0.07

From the above tables, we can see that adjacency list outperforms incidence matrix on every level, regardless of density. However, even though adjacency list still runs faster in terms of removing edges, but the gap is not as wide as that in removing vertices. Also, both data structures took significantly longer to remove vertices than edges. The results are expected, as when removing a vertex, the graph has to remove all the edges that involve the removed vertex. For adjacency list, it has to loop through all the linked list to find the edge and remove the node within the list; and for incidence matrix, it also has to loop through the columns of the matrix. However, because the structure, incidence matrix can have a really huge column list and looping through that list would take much longer than looping through the vertex list of the adjacency list. This explains the huge differences presented above.

As the density changes from High to Low, the time spent also decreases, because when density decreases, the number of edges to delete also decreases.

Scenario: Nearest Neighbors

In-Neighbor Operations

Get All In-Neighbors Average Running Time:

	Adjacency List	Incidence Matrix
High Density	1,665.35	40,373.28
Medium Density	344.46	11,819.36
Low Density	141.1	5,870.88

Get 5-Nearest In-Neighbors Average Running Time:

	Adjacency List	Incidence Matrix
High Density	1,605.53	36,579.51
Medium Density	350.95	11,944.09
Low Density	146.94	5,734.62

Get 999-Nearest In-Neighbors Average Running Time:

	Adjacency List	Incidence Matrix
High Density	1,593.38	37,019.26
Medium Density	289.83	11,939.92
Low Density	151.26	5,687.89

From the above results for in-neighbor operations, we can see that they both take a long time, more so for the incidence matrix. For adjacency list, to find the in-neighbors of a vertex, the graph has to go through edge lists of all the vertices and find the edge whose target vertex matches the vertex in question. For the incidence matrix, similarly, the graph also has to go through the entire edge map to find all the neighbors of the vertex in question, then the graph also needs to find the weight for that edge by accessing the 2-D array. Although both structures are inefficient for finding the in-neighbors, incidence matrix took over 30 times longer. The reason behind it might have something to do with the space complexity as for incidence matrix, storing the huge edge list of the incidence matrix takes up a big chunk of memory of the computer, looping through such a big list might slow down the entire operation.

Comparing all three in-neighbor operations, the results are very similar, leading us to think that sorting the neighbors doesn't take a long time, but getting them does. This makes sense as getting the neighbors has to loop through a huge number of edges ($O(E)$, where E is the number of edges), but the resultant list of neighbors can only have a maximum size of 999, which is the number of vertices minus 1. And these operations correlate well with the density, because the density determines the number of edges the graph can have.

Out-Neighbor Operations

Get All Out-Neighbors Average Running Time:

	Adjacency List	Incidence Matrix
High Density	11.4	38,574.86
Medium Density	1.75	12,619.65
Low Density	0.8	6,623.56

Get 5-Nearest Out-Neighbors Average Running Time:

	Adjacency List	Incidence Matrix
High Density	57.26	39,498.02
Medium Density	9.93	12,088.65
Low Density	4.95	5,872.09

Get 999-Nearest Out-Neighbors Average Running Time:

	Adjacency List	Incidence Matrix
High Density	35.14	37,208.86
Medium Density	9.8	12,417.03
Low Density	5.27	5,680.68

For out-neighbor operations, it's a totally different story for adjacency list. Adjacency list can easily returns all the out-neighbors a vertex has, and it takes $O(1)$ constant time to get the list of all neighbors. However, due to the nature of the linked list, we still need to go through all the nodes to extract the weights, that's why there is still a significant time difference among different densities. But overall, adjacency list is very efficient at getting out-neighbors.

As to incidence matrix, the algorithm is almost exactly the same as getting in-neighbors, therefore the results are almost the same as those from in-neighbor operations.

Scenario: Changing Associations

	Adjacency List	Incidence Matrix
High Density	0.25	0.32
Medium Density	0.03	0.03
Low Density	0.01	0.03

From the above results, we can see that when it comes to changing associations or updating edge weights, both data structures perform almost the same. For the adjacency list, it takes almost $O(1)$ time to find the linked list of the source vertex of the edge, and then finding the target vertex of the edge takes $O(e)$, where e is the number of elements in that linked list, if e is pretty small, then the time spent there is trivial. As for the incidence matrix, accessing the weight in the 2-D array is also $O(1)$ as array allows for random access when you know the index. Getting the indexes from the vertex map is fast, but getting the indexes from the edge map, although still pretty instant, might take

a bit longer as the size of the edge index map can be huge and the worst case performance can be $O(logN)$, where N is the number of edges.

As density increases, the number of edges also increases, therefore requiring more time to update the edge weights. However, the difference between Medium density and Low density are small and negligible. We think it's due to the density values we chose for Medium (0.1) and Low (0.05) are very close and if we were to make the difference wider, the time difference would become more significant.

Besides, whether it be increasing or decreasing the weights, the results will stay the same, as the underlying algorithm is the same.

Based on the analysis, the results above are reasonable and expected.

Recommendations

From the experiments and analyses, we can see that adjacency list outperforms incidence matrix in every way. Not only does adjacency list run faster under different scenarios, it also saves space. The space complexity for adjacency list is $O(V + E)$, where V and E are the number of vertices and number of edges, respectively, while it's $O(V \times E)$ for incidence matrix. Adjacency is especially good when the density is low, but still better than incidence matrix, when the density is high, because for high density, E can become V^2. Although adjacency list has its disadvantage when getting in-neighbors, it's still better than incidence matrix. In social networks research or applications, if you're more interested in knowing who one's connecting to, then adjacency list would be a great choice for data structure. Also, in reality, a person's social network, if represented in graph, is usually of very low density, because a person can't possibly know all the other people. Therefore, we would recommend adjacency list as the underlying data structure. However, adjacency list is by no means the best data structure out there. Before choosing a data structure for the problem, one should always consider all the scenarios and the time/space complexities of the data structure.