CLOSEST ASSOCIATES

By

Dac Trung Nguyen (s3748402)

COSC 2123 /1285 Algorithms and Analysis

TASK B: EVALUATE DATA STRUCTURES

This report seeks to evaluate two implemented representations of the Weighted, Directed Graph in Task A: Adjacency List and Incidence Matrix

DATA GENERATION

```
file = open("test.txt", "w")

import random

L_PROBABILITY = 5 # connect to 1 out of every 5 other vertices
M_PROBABILITY = 3 # connect to 1 out of every 3 other vertices
H_PROBABILITY = 1 # connect all other vertices

probability = H_PROBABILITY

size = 400

count = 0

# add vertexes
for vertex in range(1, size):
    file.write("AV " + str(vertex) + "\n")

# add edges
for srcVertex in range(1, size):
    for tarVertex in range(srcVertex, size):
        if (srcVertex != tarVertex):
            canConnect = (random.randint(1,probability) % probability == 0)
            if (canConnect):
                file.write("AE " + str(srcVertex) + " " + str(tarVertex) +
" " + str(random.randint(1,9)) + "\n")
                count += 1

print("Added " + str(count) + " edges.")

file.close()
```

The above is a Python Script to generate data at random. The output of the script will give vertices and edges to remove, find neighborhood and update edge weights.

EXPERIMENT SETUP

I. The Data Generator

The above code written in Python generates a graph with fixed size of 400 vertices. To change the density, I have chosen Low Probability to be that only 1 out of every 5 vertices are connected to each other. Medium Probability to be that only 1 out of every 3 vertices are connected to each other. And High Probability to be that every vertex are connected to each other.

To ensure randomness, I have also used randint() function of the random module in Python 3. The edges are generated such that a random number of 1 to the "probability" chosen is divisible by the "probability". If it is divisible, the pair of vertices will connect, forming an edge.

II. Measuring Time performance

We use Java System.nanoTime() to measure the time it takes to 1) remove vertices and edges, 2) implement nearest in and out neighbors and 3) update edge weights.

Each of the relevant methods in Java are to print out the duration when each method is run, i.e. we start with a starting time at the beginning of the method and end with an ending time at the end of the method. For example, the code for updateWeightEdge () method is written as follows:

```
public void updateWeightEdge(String srcLabel, String tarLabel, int weight)
{
    long startTime = System.nanoTime();
    Node currentNode = vertMap.get(srcLabel);
    if (weight != 0) {
        while (currentNode != null) {
            if (currentNode.getTarLabel().equals(tarLabel)) {
                currentNode.setWeight(weight);
                break;
            }
            currentNode = currentNode.getNextNode();
        }
    }
    else {
        if (currentNode != null) {
            if (currentNode.getTarLabel().equals(tarLabel)) {
                vertMap.put(srcLabel, currentNode.getNextNode());
            }
        }
        while (currentNode != null) {
            Node nextNode = currentNode.getNextNode();
            if (nextNode != null) {
                if (nextNode.getTarLabel().equals(tarLabel)) {
                    currentNode.setNextNode(nextNode.getNextNode());
                }
            }
            currentNode = currentNode.getNextNode();
        }
    }
```

```
    long endTime = System.nanoTime();
    long duration = (endTime - startTime);
    System.out.println("Executed in " + duration + " nano seconds.");
} // end of updateWeightEdge()
```

The same is set up for vertex removal method, update edge weights method and nearest neighbor methods.

III. Notes

A. Scenario 1

Each vertex removal method is done on the same vertex which is chosen to be 1, i.e. the command is RV 1. And then the time measured is taken.

Each edge removal method is done on the same edge which is chosen to be 1 and 2, i.e. the command is U 1 2 0, which means edge 1 and 2 should have 0 weight. And then the time measured is taken.

B. Scenario 2

Each in-nearest neighbor method is done on the same vertex which is chosen to be 1, and for all of the nearest in neighbors of 1, i.e. the command is IN -1 1

Each out-nearest neighbor method is done on the same vertex which is chosen to be 1, and for all of the nearest out neighbors of 1, i.e. the command is ON -1 1

C. Scenario 3

Weight update is done on, for example, edge between 1 and 14, to increase it from weighted 7 to weighted 11. The command is U 1 14 11.

Weight update is done on, for example, edge between 1 and 14, to decrease it from weighted 11 to weighted 7. The command is U 1 14 7.

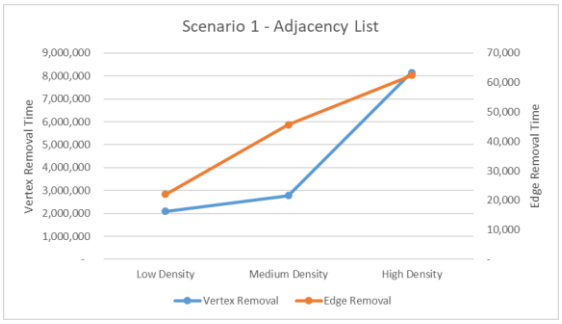
IV. The results

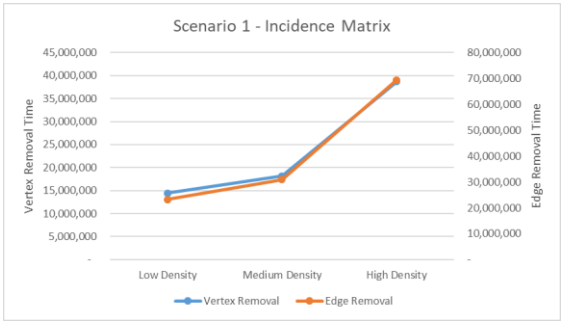
Each scenario is repeated 10 times and the results in nano seconds are average as follows:

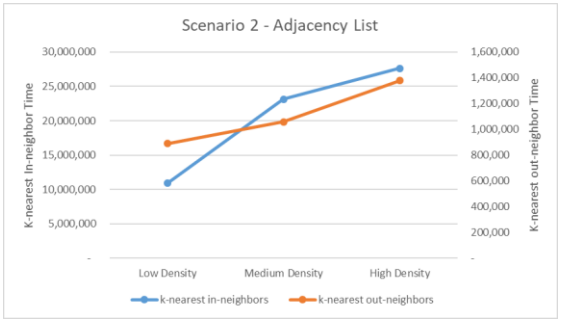
Adjacency List	**Scenario 1**		**Scenario 2**		**Scenario 3**	
Time (nano seconds)	Vertex Removal	Edge Removal	k-nearest in-neighbors	k-nearest out-neighbors	Increase weight	Decrease weight
Low Density	2,098,241	22,045	10,893,901	889,843	12,730	4,968
Medium Density	2,773,851	45,641	23,128,155	1,059,056	12,109	3,105
High Density	8,148,304	62,407	27,630,463	1,377,611	16,145	4,036

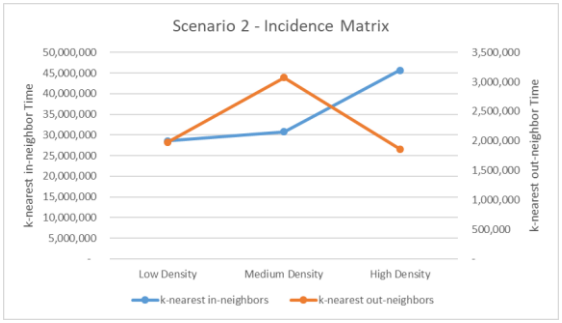
Incidence Matrix	Scenario 1		Scenario 2		Scenario 3	
Time (nano seconds)	Vertex Removal	Edge Removal	k-nearest in-neighbors	k-nearest out-neighbors	Increase weight	Decrease weight
Low Density	14,455,446	23,235,582	28,587,680	1,979,637	592,711	14,592
Medium Density	18,178,132	30,953,557	30,758,884	3,073,467	858,484	31,048
High Density	38,695,128	69,445,940	45,636,276	1,861,653	871,835	15,524

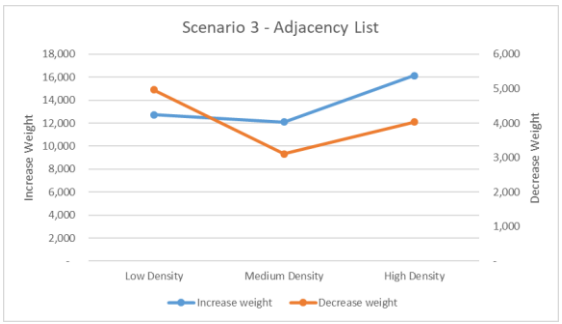
The two tables above show the average time performance of each scenarios in each level of density, for both graph representations.











V. Observations

Difference (IncMat – AdjList)	**Scenario 1**		**Scenario 2**		**Scenario 3**	
Time (nano seconds)	Vertex Removal	Edge Removal	k-nearest in-neighbors	k-nearest out-neighbors	Increase weight	Decrease weight
Low Density	12,357,205	23,213,537	17,693,779	1,089,794	579,981	9,624
Medium Density	15,404,281	30,907,916	7,630,729	2,014,411	846,375	27,943
High Density	30,546,824	69,383,533	18,005,813	484,042	855,690	11,488

When comparing directly between Time performance between the two representations, it is clear that Adjacency List outperforms Incidence Matrix in all operations.

The matrix is usually fast in looking up if a particular edge exists. However, it is slow to iterate over all edges and therefore, slow in adding/deleting a node. Time complexity is $O(n^2)$. Adjacency list however, is fast in iterating over all edges because it can access neighbor nodes directly. The adjacency list is also faster to adding / deleting a node at O(1). Therefore in our results, we see that removal operation (scenario 1) and searching neighbors (scenario 2), the matrix cannot perform as fast as the Adjacency List.

The adjacency list takes more time looking up a particular edge at O(i) with i is the number of neighboring nodes. Since each list can be as long as the degree of the vertex, at worst-case scenario (unordered list), it must go through all neighboring nodes. Therefore, with denser graphs, we can see from our results above that adjacency list shows signs of an **accelerating cost** in time performance. In contrast, the matrix shows even better performance at higher density.

On space complexity, the matrix is likely to take up more space. As it is an N by N array, it takes $O(n^2)$ space complexity. When the graph has few nodes (sparse), there would be inefficient number of edges. For adjacency list, we only create new node when it is needed, thus, this results in lower memory usage.

VI. Recommendation

The adjacency list is fast in iterating over all edges, fast in removing and searching neighbors. And it is better performance for sparse graphs. This makes the adjacency list more suitable for real-world problems which mostly produce sparse graphs. At denser graphs, the matrix could be a better choice. However, social network problems such as Facebook and Twitter rarely produce dense graphs like such.

The matrix is better if it is used for transportation where each node location may not need constant update. It is obvious that the matrix uses excessive space. If the graph is sparse, this could be a waste of memory.