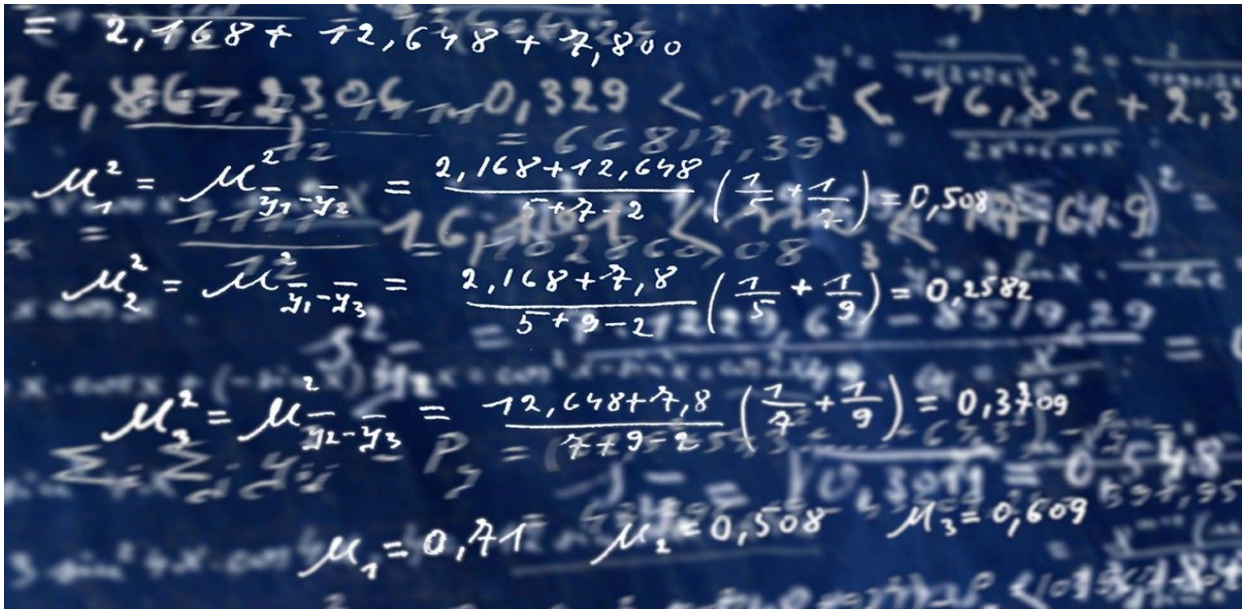
ALGORITHMS AND ANALYSIS

REPORT - Task B

DUY VO - S3723069
DON PHUU - S3716808



Introduction

Over the course of time, algorithms have been used widely in everyday situations, whether it is in the database - retrieving data within a database, or solving a simple problem such as sorting a set of numbers in ascending or descending order. Scientists and mathematicians continue to discover new algorithms, more efficient to cater their complex problems. This leads to more unique algorithms, as new algorithms are implemented. With new algorithms arising from the surface, the algorithm gets better to suit the complex problem.

This report will cover the time complexities of an algorithm, and review the adjacency and incidence matrix that was implemented in part A of the assignment.

Aim:

- Testing the provided association graph with our implemented methods from the adjacency list and incidence matrix classes.
- Testing the time complexities of the three scenarios of the data set
    - Removal of vertex / edges
    - K - nearest neighbours
    - Updating weight edges
- How does the density of the graphs affects the results (Low , Medium , High)

Time Complexities

Pre Knowledge

Before we discuss the basis of the report, we must discuss the importance of time complexities. In computing, time complexities involves the time that is required to compute an algorithm. Time complexities are calculated by the amount of operations in the algorithm by the input size. Therefore, algorithms will be calculated by a **constant factor.**

Big O Notation

The way that the time complexity of an algorithm can be calculated is using big O notation. In Computer Science, big O notation is the way to describe how an algorithm performs based on their operations and it's complexity.

Different algorithms will have different time complexities because some algorithms may contain more operations than the other. An example of this is iterating through an array to obtain the element. Here, the time complexity of the algorithm is O(N), as it has a constant performance of n.

```
boolean findWeight(int value)
{
    for(int i = 0; i < weight; i++)
    {
        if( weight[i] == value )
        {
            return true;
        }
    }
}
```

A single for loop represents a time complexity of O(N)

Whereas, accessing a nested for loop will have a constant factor of $O(N^2)$, because iterating through a nested for loop will square the amount of operations.

```
for (int i = 0; i < numOfHead; i++) {
    for (int j = 0; j < adjList[i].getNumEdges(); j++) {
        if (adjList[i].getEdgeArray()[j].getTarget().equals(vertLabel)) {
            myEdge.add(adjList[i].getEdgeArray()[j]);
        }
    }
}
```

A nested for loop will have a time complexity of $O(N^2)$

Different methods within a class will have different time complexities. For instance, in an adjacency list and an incidence matrix, its methods (removal of vertex/edges, retrieving k nearest neighbours and updating weight edges) will contain different time complexities because of the way their data is stored.

This will be discussed further below.

Adjacency List

An adjacency list is a data structure that has a node that contains a array of it's connecting edges. In terms of its structure, it is possible to have a node that contains no edges and thus displaying a node with no lists of edges.

Removal of Vertex/Edges

When removing a vertex, it is compulsory that all the edges linked to that vertex must also be removed. Hence, the time complexity of this operation (using big O notation) is denoted as

O ((time to find vertex) + (time to find edge))

Retrieving K-Nearest Neighbours

When retrieving a vertex's k-nearest neighbours, it depends on the the k value. Furthermore, we must iterate through all the nodes as well as iterating through all their edges to find the highest neighbours for the desired node. The time complexity for this can be denoted as

O ((time to find vertex) + (time to find edge) + (time to find desired node))

Updating Weight Edges

When updating the weight edge of a source node to a target node, we must iterate through our adjacency list to find both the source node and target node, at which we will be able to update their edge weight. In Big O Notation, this can be denoted as

O ((time to find source node) + (time to find target node))

Incidence Matrix

The incidence Matrix, unlike the adjacency list is based on a series of arrays, these arrays in turn supply the graph data in the form of a symmetric matrix which is occupied by edge weights.

Removal of Vertex / Edges, Updating Weight Edges, Retrieving K-Nearest Neighbours

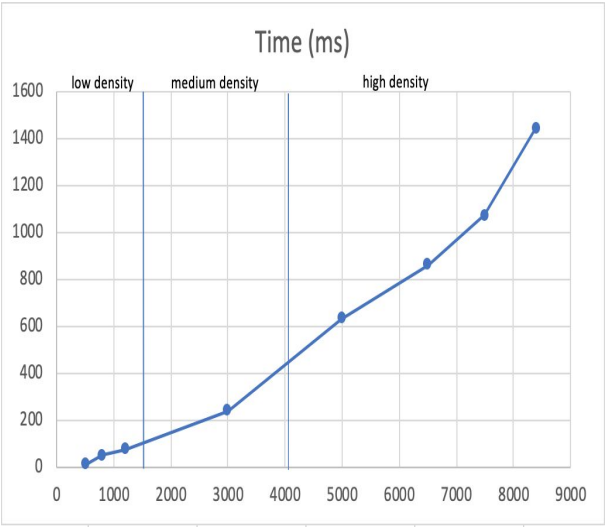
Follow the removal of Vertex, comes the removal of edges. In an Incidence Matrix, the removal of a vertex requires the iterations through the rows and columns in order to remove all related edges to that vertex. Once related edges are removed, objects in the matrix must be refactored accordingly. Hence, the expected trend for the average case time complexity would be (in Big O Notation)

O (time to find desired vertex (column) * time to find all desired edges (row))

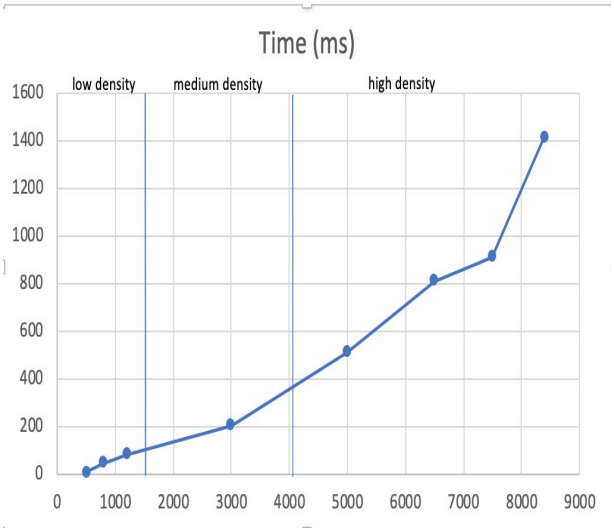
This time complexity is also applicable for the other majority of the methods are well since they all must undergo a row and column search of the vertex/edges (or iteration) in order to perform their individual tasks. The updating edges method and retrieving neighbour methods may slightly differ in time complexity considering minor essential operations.

Adjacency List (X AXIS -> DENSITY, Y AXIS -> TIME(MS))

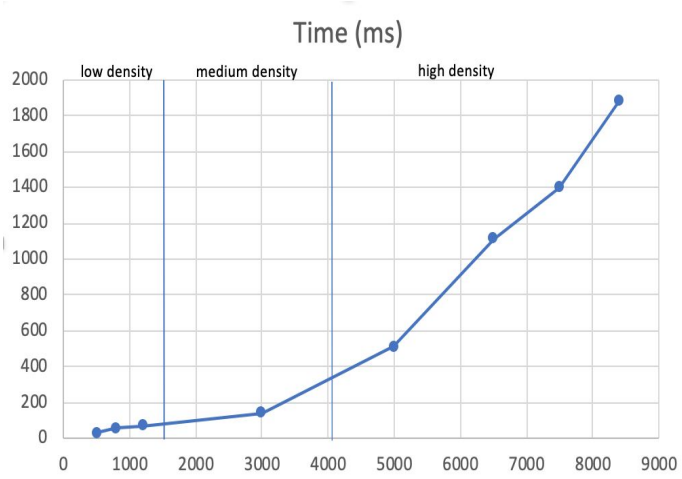
Removing Vertex and Edges



Updating weight from source to target



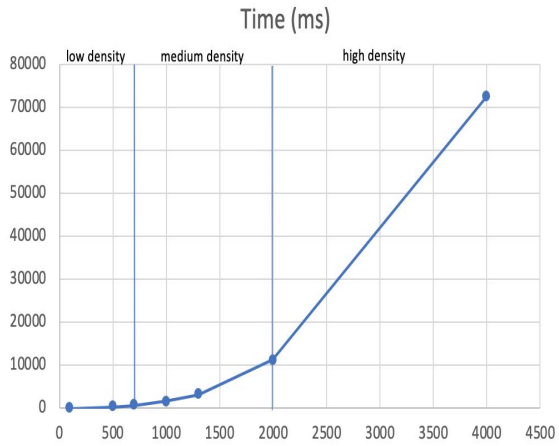
K Neighbour



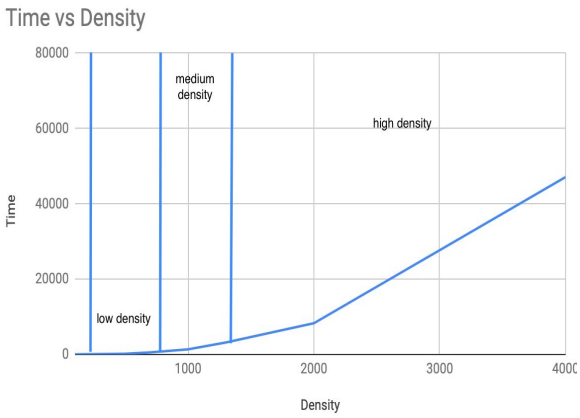
Overall = O (n) **or** (n iterations till node and execute);

Incidence Matrix:

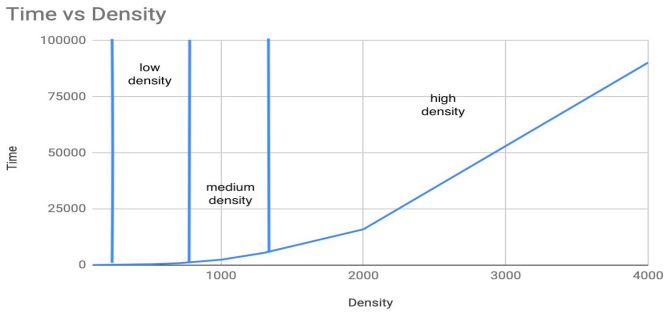
Removing Vertex / Edge



Updating Edge Weight



Finding Nearest Neighbours (In only)



Overall = O(n^2) **or** (n iterations to find column * n iterations to find row)

Discussion

How does the density of the data affect the time complexity of an algorithm?

Throughout the density tests we have ran, both graphing algorithms have illustrated the general trend that as the density of the objects occupied in the list increase, the time it takes for vertices and edges to be accessed increases. This is fairly rational as we iterate through more objects the more essential operations and comparisons the program must perform, hence increasing the time of execution for algorithm.

Method Execution and Time ...

The two algorithms also suggest that finding the nearest neighbours takes the longest out of the three methods. For this we can assume that since the we use an inbuilt sorting algorithms ; in this case Collections.sort(), which is also overridden. This sorting algorithm influences the time taken to be executed because it causes the inNearestNeighbour() method to iterate through the arrays, lists or nodes in both classes. The extra iteration results in a lengthy addition of time in comparison to the other methods.

Updating the edge was the deemed the quickest for the incidence matrix. In comparison to the remove vertex, it would be assumed to be quicker as, we are on finding one distinct edge which on requires one 2D array iteration, once the edge is found it is updated. Remove vertex for the incidence matrix requires a couple more array iterations in order to dynamically adjust the 2D array after all the related edges and the vertex itself is removed.

On the other hand for the adjacency list, update edge and remove vertex are quite similar in time. The speculation for this result is that since both methods follow the logic : find the node is found, execute desired action (remove/update) the execution times will be fairly similar as only an essential operation is performed.

So.. Incidence Matrix or Adjacency List ?

Overall our implementations illustrated that the Adjacency was much more efficient that in executing methods in comparison to the Incidence matrix. Due to the nodes in the Adjacency list, the lack of iterations proved to perform more rapidly than a 2D array . The extent of efficiency however can be questioned as it is far too drastic (on average 40x more efficient) considering the time complexity of O(n) vs O(n^2). This may of be caused by flaws in the implementational logic or language restrictions which had to be followed, with that being send, we were able to obtain a general trend for the average case time complexity.