
# CLOSEST ASSOCIATES

COSC 1285 – ALOGORITHM AND ANALYSIS

APRIL 15, 2019

S3692260 DHILIP SURYA RAJENDRAN
S36700757 JAYARAJ ALAGU PONNIAH

# CONTENTS

# INTRODUCTION

Social networks have risen up to be preferred for communication in the past decade. Modelling and mining these social networks have advantages in various fields. In this report we have decided to mine and create a communication graph with nearest neighbours to identify the relationship between users based on the communication patterns.

There relationships vary in different levels. To identify the level, we assign weight to the edges from vertices, by indicating the relationship between different users. The graphs are directed which means the edges have a direction notifying the incoming and outgoing relationships.

We have implemented two data structures, one with an incidence matrix and the other with an adjacency list. Both process outcomes are analyzed carefully to decide which type of process can be more advantageous over the other when building graphs to mine social networks.

# DATA GENERATION AND EXPERIMENTAL SETUP

The following methods have been implemented in our assignment to generate the data used for testing our approaches taken to solve the problem mentioned in the problem statement of the assignment. The classes in the source code are a representative of these methods we need to implement to solve this problem. Below mentioned are the steps undertaken to solve the problem statement sequentially.

- First, we have to load the test data from the assignment specifications in the pdf, as mentioned.
- Next, we implement the command to add the vertices to our file. Next, we add the command to create edges from the vertices. Next, we create a command to calculate the shortest paths from the neighbors.
- Next, we implement the commands to remove the edges and the vertices in a random way, which is not predefined. The next and the final step is to pass the generated file to the evaluation function of the graph as mentioned in the source code section.

Test cases were generated to test the efficiency of each generated graph. These tests generated sufficient evidence to back our conclusions made in the last section of this report.

# ADJACENCY LIST IMPLEMENTATION (TIME COMPLEXITIES CALCULATIONS)

The tasks carried out for Adjacency List are mentioned below along with their theoretical time complexities.

## 1. VERTEX ADDITION

The algorithm operates by adding 0's to the row and column of the adjacency list representing the graph and has a time complexity represented by $C(|V|)$ = Σ 2 = 2(|$V$| − 1 − 0 + 1) = 2|$V$| |$V$|−1 $i$=0

Thus, it is expected that the time for adding a vertex will increase linearly (O(|V|)).

## 2. EDGE ADDITION

The algorithm implemented inserts 1's into 2 positions in the 2D matrix through random access (because the list was declared as a 2D array). Thus, only 2 operations are done, therefore the time complexity is expected to be O (1), constant.

## 3. FINDING VERTEX NEIGHBOURS

In order to obtain a list of neighbours of a vertex, the algorithm finds the index of the vertex by calling indexOf () in the ArrayList and then for that vertex, scans the |V| columns in the 2D matrix (the 1's in the matrix representing the neighbours). The order of growth is (O|V|).

## 4. VERTEX REMOVAL

The algorithm implemented for the vertex removal has the following time complexity: $C(|V|)$= (Σ1|$V$|$i$=1)+ (ΣΣ2|$V$|$j$=1 |$V$|−1$i$=$vertexIndex$)+Σ2|$V$|$i$=1=|$V$|+ (Σ2|$V$||$V$|−1$i$=$vertexIndex$)+ 2|$V$|=3|$V$|+2|$V$|Σ1|$V$|−1$i$=$vertexIndex$

Note that the vertexIndex value can vary from 0 to |V|-1. Since for these tests, vertices were selected randomly, the 2|$V$|(Σ1|$V$|−1$i$=$vertexIndex$) part of the formula can result in 2|V|2, if vertexIndex =0, or 1 if vertexIndex = |V|-1, thus:

$C(|V|)$ $worst$= 3|$V$|+2(|$V$|2) □ O(|V|2)

$C(|V|)$ $best$= 3|$V$|+2|$V$| □ O(|V|).

The selection of vertices to be removed was random, so there is an equally distributed probability that the order of growth will be from |V|to |V|2. In other words, the function grows asymptotically not faster than |V|2 but faster than n or in a more mathematical notation, C(|V|) = Ω(|V|) and C(|V|) =O(|V|2).

### 5. EDGE REMOVAL

The edge removal algorithm simply searches for the two vertices indexes within the vertices array list and then performs another 2 assignment operations to add 0 into the 2 positions in the list representing the edge, further deducting the number of edges. Therefore, the algorithm's efficiency is expected to follow the same asymptotic growth as the searching of the vertices within its array list. It was discussed previously that this algorithm grows linearly (i.e. O(|V|)).

## INCIDENCE MATRIX IMPLEMENTATION (TIME COMPLEXITIES CALCULATIONS)

### 1. VERTEX ADDITION

If you need to copy the existing matrix, you have n2n2 entries to copy, which takes Θ(n2)Θ(n2) steps; if you only need to add the new entries (for example, if you've pre-allocated a large matrix but you're adding the vertices one at a time), you only need to write O(n)O(n) entries each time.

If you did have to copy the matrix every time you grow it, a more time-efficient option would be to start with a small matrix (say, 10×1010×10) and double it every time it fills up. Doing it that way means you need fewer expensive copies, but the potential down-side is that you could waste a lot of space. For example, if your graph ends up having 21 vertices, you'll have allocated a 40×4040×40 matrix, most of which is empty. You could mitigate that by not growing the array so aggressively (say, making it 25% bigger each time it grows, instead of 100%).

### 2. EDGE ADDITION

Given an incidence matrix with row for each vertex and column for each edge, most of the elements of the matrix will be zeroes (there will be at most V nonzero elements on each row). That's called sparse matrix and it can be stored in the memory in some space efficient way, which also allows "skipping" long sequences of zeroes in constant time.

For example, with simple list of lists representation, you have a list of pairs (columnindex, value) ordered by columnindex for each row, one pair for each nonzero element. In our case, just check rows u and v, whether they have an item with same columnindex. As they are both ordered and both have less then V items, it can be done in O(V).

### 3. FINDING VERTEX NEIGHBOURS

This algorithm iterates through every edge and then checks if the edge contains a vertex with the same label as the vertex for which neighbours are being checked. Thus, the time complexity grows asymptotically to Θ(|E|).

## 4. VERTEX REMOVAL

The algorithm for vertex removal has 3 steps:

1. The first step is always removing the edge. We have to go through every edge to determine if vertex is a part of the edge before removing the vertex directly. If an edge has the vertex, then it iterates through the matrix starting from the index where the edge was found to the end.

2. Next we remove the row denoting the vertex and shift the rows a level up.

3. The last part is actual vertex removal, where we remove the vertex from our data.

Thus, the time complexity of the whole process comes up to be $O(|V|^4)$.

## 5. EDGE REMOVAL

This process finds the edge first who has v1 and v2 as part of its vertex list. Then it iterates through the matrix from the starting index of the edge till the end to delete the edge and shift all columns to the left by one place. This is the basic idea of edge removal in graph theory.

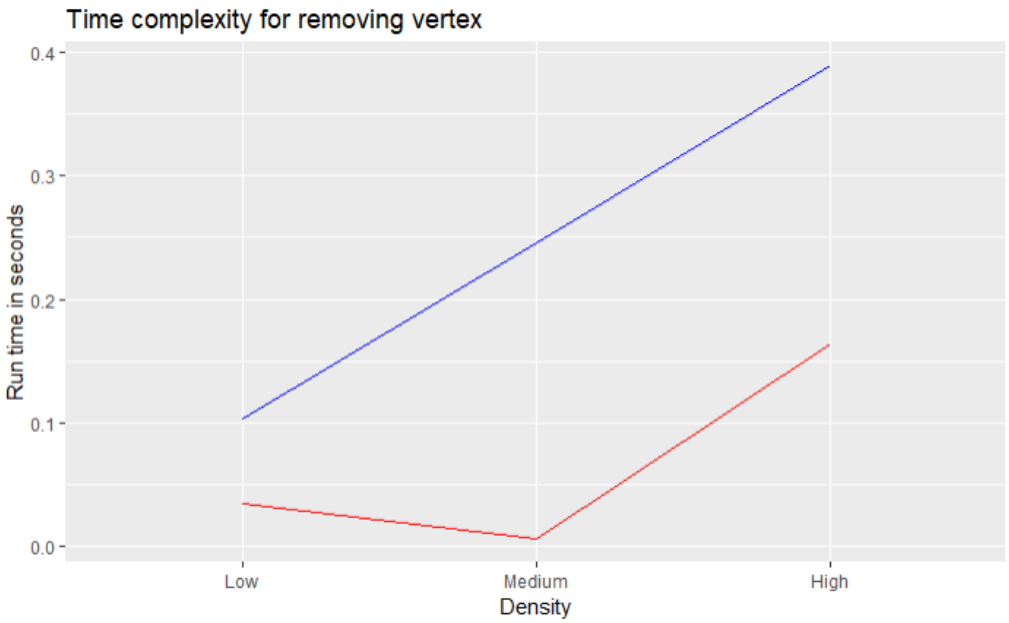
The algorithm implemented had some time efficiency issues. It does not check the existence of the vertices first; in such case the algorithm could terminate earlier (as there would not be an edge if one of the vertices do not exist). At the time the experimental data was run, there was no “break” statement for the main loop once an edge is removed and the matrix is rearranged. Without breaking the repeat, the algorithm continues to unnecessarily iterate through the remaining edges. It was impractical to re-implement the algorithm for testing, as the sample sizes were very large.

In the worst-case scenario, any vertex will be connected to every other vertex. Therefore, the time elapsed grows asymptotically to $O(|V|^3)$.

# EVALUATION OF DATA STRUCTURES

## SHRINKING GRAPHS

Here we perform the vertex and edge removals for the data structure. The reason we are doing this is because in social media networks people unfollow or unfriend the person. So, to represent that we try to remove a vertex. We vary the density and the graph is plotted for the same.




**Graph 1 for Removing vertex**

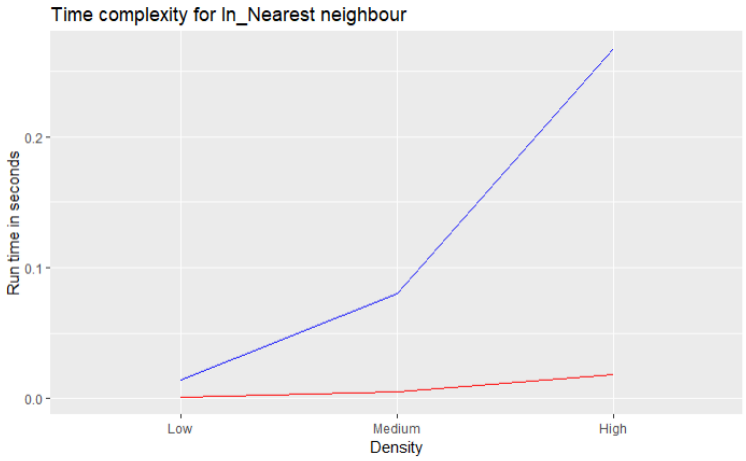
BLUE—Incidence Matrix

RED—ADJACENCY List

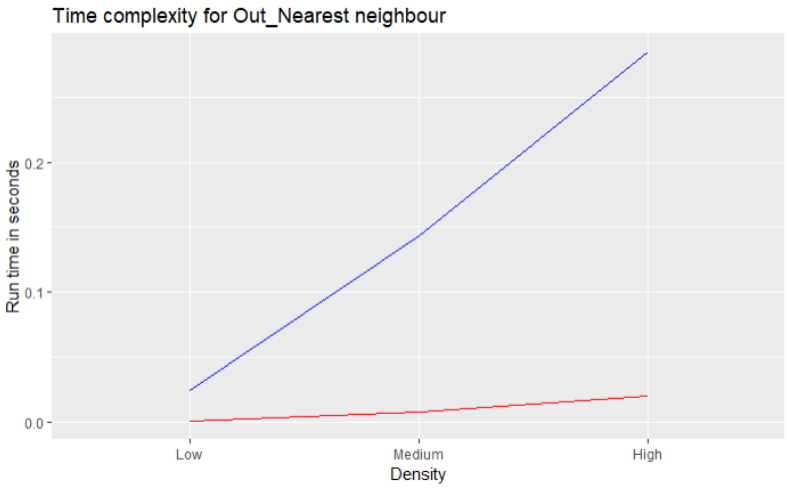
The graph shows us that the time complexity for different density of values. So, on looking at the graph we see that adjacency list has requires less time when you shrink the graph. We can see that for medium density data the time complexity is less. There is a linear increase in the Incidence matrix.

## NEAREST NEIGHBOURS

In this we find the nearest neighbours of each vertex. The value of k gives us the nearby neighbours. We get the absolute values of the input value except for -1 which means we must return all the values which is all the nearest neighbours. The graph here doesn't change, we just test the performance of the nearest neighbourhood implementations.




**Graph 2 for in nearest neighbours**




**Graph 3 for out nearest neighbours**

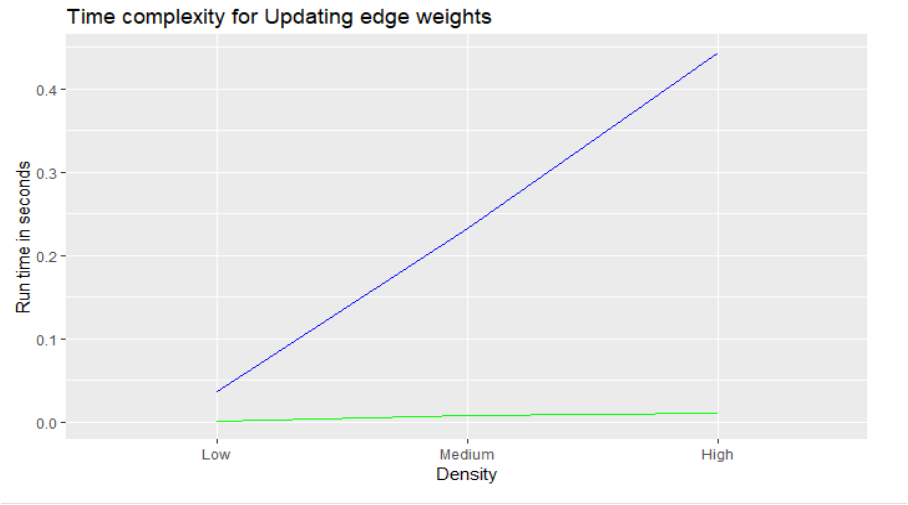
BLUE—Incidence Matrix

RED—ADJACENCY List

When we consider on checking the in and out neighbours we find that there not much change in the graph. So, this one also states that the time complexity of returning the neighbours is least for Adjacency list. As the graph shows the time consumption is very less.

## CHANGING ASSOCIATES

In this case scenario we test association between people when there is a fluctuation between corresponding to the change in edge weights. The changes that are being performed here is changing the edge weights by increasing or decreasing it.




**Graph 4 Updating edge weights**

BLUE—Incidence Matrix

RED—ADJACENCY List

As seen before there is a linear increase for the incidence matrix but when we consider the adjacency list it hardly rises. It stays in the range of less than 0.1 seconds. It remains almost constant for all the types of data (with respect to data density) that we calculated. Therefore, high density data can be use adjacency list data structure when compared to the incidence matrix.

The values of the time complexity are tabulated below. This shows a clear view on the how the value changes with respect to the density variation.

| | adjacent_s1 | Incidence_s1 | adjacent_In_s2 | Incidence_in_s2 | adjacent_out_s2 | Incidence_out_s2 | adjacent_s3 | Incidence_s3 |
|---|---|---|---|---|---|---|---|---|
| Low | 0.034 | 0.1039 | 0.0012 | 0.014 | 0.0011 | 0.0247 | 0.001 | 0.0367 |
| Medium | 0.0063 | 0.246 | 0.0051 | 0.08 | 0.0083 | 0.143 | 0.008 | 0.232 |
| High | 0.1639 | 0.389 | 0.018 | 0.267 | 0.0199 | 0.2847 | 0.011 | 0.443 |

# CONCLUSION

From the calculations done from the above sections and the experiments run we conclude that the adjacency list is more advantageous than the incidence matrix in average cases of complexity, if we consider the time complexities and space complexities of the algorithms.

From the graph 1 it can be seen that the to add or remove a vertex in adjacency list is O(1) and for the incidence matrix which is a square matrix has a time complexity of O(n^2).

From the graphs 2 and 3 the time complexity which is that show the nearest neighbours we traverse the whole matrix which leads to O(n^2). But we just find the links that are attached to the link in the adjacency list.

From the graph 4 the time complexity to update the edge can be changing the values in matrix but in two and traverse the matrix. In case of adjacency list there is no much work to be done. The time complexity is very less when compared.