REPORT FOR
ALGORITHM AND ANALYSIS ASSIGNMENT 1

Submitted By:
Rashiv Romio Bhusal (s3511441)
Anto Dominic (s3553172)

ALGORITHM AND ANALYSIS
ASSIGNMENT 1

TABLE OF CONTENTS

1. EXPERIMENTAL SETUP

1.1 Data Scenarios:

We have used the test1, test2 and test3 provided with the initial Assignment files to debug and correct the code for various logic's including show vertex, show edges, show neighbours and shortest path.

For the report we had created a generator to test graphs with different density. To increase and decrease the density we increased and decreased the number of edges in the graph. The number of edges that we generated were 199800 (Low Density, approx. 20%) , 699300 (Medium Density, approx. 50%) and 699300 (High Density , approx. 70%). The performance has affected as the data size had been increased. The value that is generated during the addition, removal and shortest path calculation is being changed as the size got increased.

Scenario 1:

Additions of Vertexes:

Size of Vertex		Adjacency Matrix	Adjacency List
1000	L(20%)	49.123764178	2.813513176
1000	M(50%)	50.672032374	2.9584982
1000	H(70%)	50.26879	6.3425967

Additions of Edges:

Size of Edges		Adjacency Matrix	Adjacency List
199800	L(20%)	26.9735597	489.710181798
499500	M(50%)	52.0008164	3525.19038
699300	H(70%)	64.2539459958	4247.6925

Scenario 2:

Neighbors and Shortest Path:

Shortest Path:

Size		Adjacency Matrix	Adjacency List
50	L(20%)	58.7178963	326.222583
50	M(50%)	50.3428039	644.669398

50	H(70%)	55.3362	378.7726608

Neighbors:

Size		Adjacency Matrix	Adjacency List
50	L(20%)	5.780785	4.24805796
50	M(50%)	6.5466389778	5.6715404
50	H(70%)	6.50147	4.34913

Scenario 3:
Removals of Vertexes:

Size		Adjacency Matrix	Adjacency List
100	L(20%)	1.9757936	40.8191429
100	M(50%)	1.765359	175.575379
100	H(70%)	1.894169	58.8324358

Removals of Edges:

Size		Adjacency Matrix	Adjacency List
100	L(20%)	0.0956899	0.963057198
100	M(50%)	0.017850376	1.2611254
100	H(70%)	0.030524578	0.9131

In scenario 1, 2 and 3 we have used low ,medium and high density of graph in 20%,50%,70% ratios. The results also shows the difference of addition as well as the removal of the number of vertices, edges and the shortest path during the calculation time.

1.2 Generation of the Scenarios:

The java code written in the DataGenerator.java was programmed in such a way that user could generate a test.in file by providing the values (a, b, c , d , e ,f,g) where ‘a’ is the maximum range of number to be used in vertex numbering, ‘b’ is the total number of vertex to be added, ‘c’ is the total number of edges to add, ‘d’ is the total number of vertex to be deleted, ‘e’ is the total number of edges to delete, ‘f’ is to add the total number of neighbouring vertexes to calculate

and ‘g’ is the total number of shortest path to generate. Then from the run configuration we will put the data size that we want to generate and this will populate the test.in file in the folder which can be tested using the python script.

```
ALGORITHM DataGenerator
//INPUT:  add vertex,add edge,remove vertex , remove edge, shortest path, neighbours
//OUTPUT: test1.in with add vertex, add edge, remove vertex ,remove edge, shortest path,neighbours
        1: for i form 1 till number of adding
        Generate random words from the fixed string set.
        Add in list Append “AV”for adding vertex
        Add in list Append “AE”for adding edges in front of the number and print in test.in file
        2: for i form 1 till number of remove .
        Get from the list , append “RV”to remove vertex
        Get from the list ,append “RE” to remove edges in front of the number and print in the test.in
        3: for i form 1 till number of shortest path□
        Get from the list, append “S ” in front of number and print in test.in file
        4: for i form 1 till number of the neighbours
        Get from the list, append “N” in front of the number and print in the test.in file
```

1.3 Timing:

To get the timing of each operation performed on data structures having different data sets, we have used the start time using the System.nanoTime() method and the end time is recorded after the completion of the operations. The calculation that have been used for the calculation of each operation timing is

Time= ((double)(endtime-starttime)/Math.Pow(10,9))

```
Example: Time measurement

Starttime=System.nanoTime();
If (tokens.length ==2)
{
Graph.addVertex(tokens[1]);
}
else
{
System.err.println(lineNum + “: incorrect number of tokens.”);
}
```

```
endtime =System.nanoTime();
addvertextime +=((double)(endtime-starttime)/Math.pow(10,9));
```

We have recorded the time in milliseconds.
Each of the data file generated is tested on every data structure, in other words each data file is tested 5 times. We have generated 6 data files that are tested upon all of the data structures. We have performed 30 tests using time operation method.

Example of the data generated for adding the vertex	
Size -1000	Adding the Vertex
1st	2.18021
2nd	3.10475
3rd	2.50217899
4th	11.86856900
5th	12.05727599
Total time average:	6.34259679

1.4 Fixed Sets:
For the generation of the data randomly we have taken numbers from 0 to x_value, where x is the maximum range of the number to be included in the generation. In our case we used x_value as 1000, ie we will only generate numbers vertices from 0 to 1000.

We calculated the no of edges using the formula : ((n^2 - n) * percentage_density) where n is the number of vertices.

Example: The below values are common for all tests:

Range: 1000
No of vertices : 1000
No of vertices Removals : 100
No of edges Removal : 100
No of Neighbours to be added : 50
No of Shortest path : 50

No of Edges is different for test4, test5 and tes6 in the order :

a) High Density : 699300 edges (test4.in)
b) Medium Density : 499500 edges (test5.in)
c) Low Density : 199800 Edges (test6.in)

1.5 Generated Data Labels

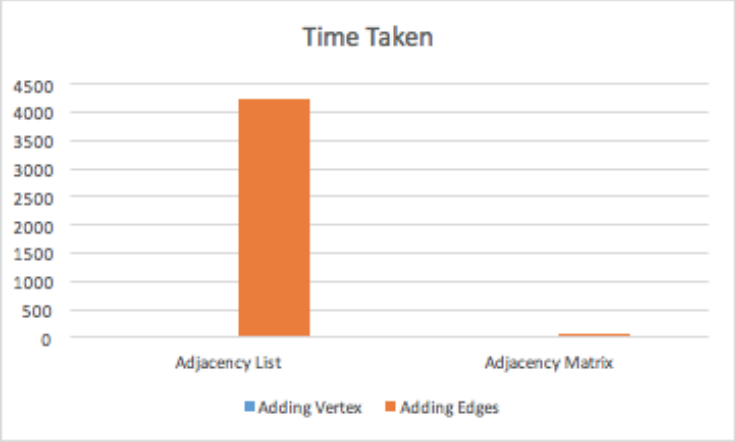
The size of the experimental data is 25000 of string data. We have randomly selected the data for the operation of the addition , remove and shortest path of the graph . We have chosen 3 different scenario which will differ from each other.

2.1 SCENARIO 1 (Growing Friendship Graph)

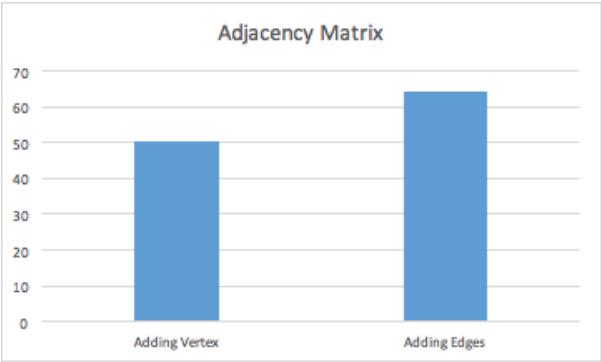
Additions

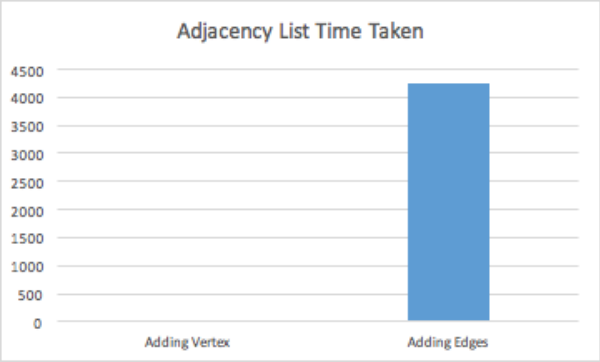
	Time Taken	
	Adjacency List	Adjacency Matrix
Adding Vertex	6.3425967	50.26879
Adding Edges	4247.6925	64.2539459958

Fig :-Performance of data structures in Case1



From the graph above it is clear that Adjacency Matrix is way faster overall compared to adjacency List. The graph below shows the difference of each Adjacency matrix and Adjacency List on basis of Adding vertex and adding edges.





From the experiment it was found out that adding vertex takes less time in Adjacency List compared to Adjacency Matrix and the average time difference between both were only 45 milliseconds. The reason for this can be because in Adjacency matrix we are using to Hashmaps to store the key-> object and object -> key data.
For adding edges adjacency matrix takes less time compared to adjacency List. This is because we are only using arrays which is 0's and 1's in adjacency matrix while we are using data structure for adjacency list. The difference between the time taken between adding edges were huge around a difference of 4000 millisecond.

2.1.2 SCENARIO 2: Neighborhoods and shortest paths

	Time Taken	
	Adjacency List	Adjacency Matrix
Shortest Path	378.7726608	55.3362
Neighbors	4.34913	6.50147

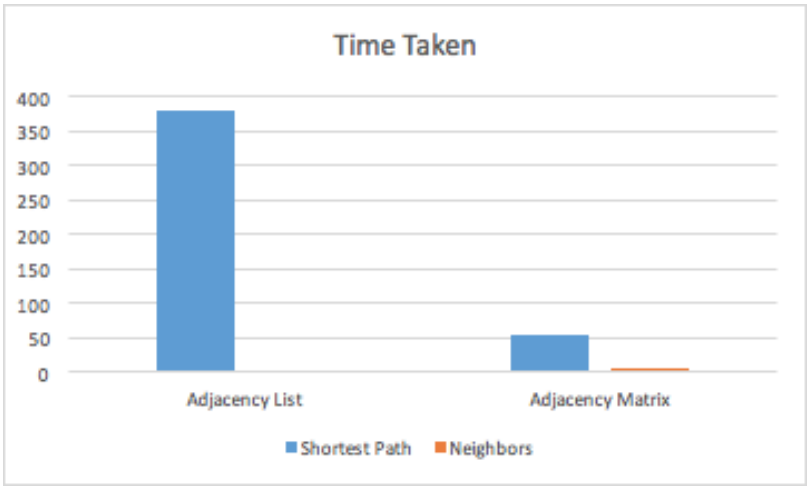
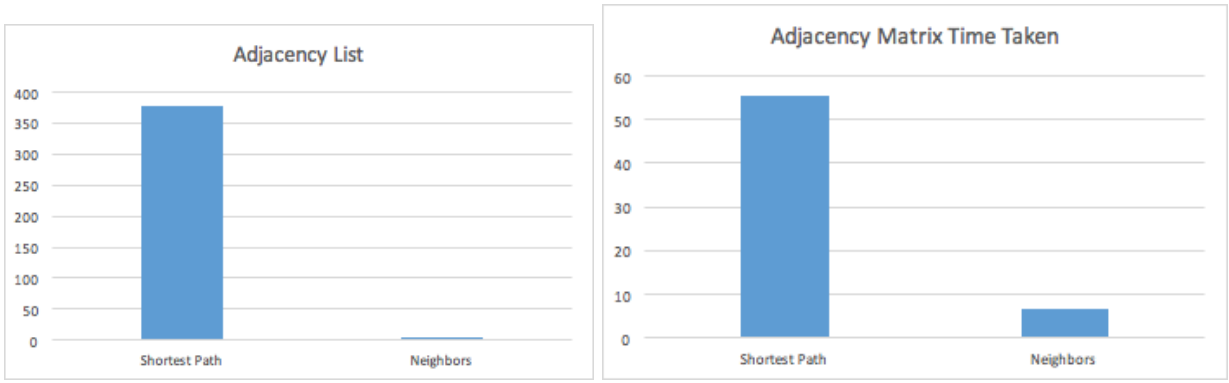


Figure:2 Performance of data structures in Case2

From the overall time taken between Adjacency List and Adjacency Matrix for the shortest Path testing and displaying Neighbors testing it is clear that Adjacency Matrix is again the winner in terms of speed.



From figure 2.a, we can see that for calculating the neighbors both Adjacency List and Adjacency Matrix is taking almost equal time. While for calculating the shortest path Adjacency Matrix is much faster by 300 millisecond on average.

Shortest Path:

We have used the Dijkstra algorithm to calculate the shortest path between the start node and any other node in a graph. This algorithm will continuously calculate the shortest path beginning from a start point, and does not included the longer paths when the update is made. The following steps is used :

1. Initialization of the nodes with infinite distance.
2. Marking of the distance as temporary for all other nodes and permanent to the starting node.
3. The start node is set to active.
4. The temporary distances of the neighbors of the active node is calculated by summing the distance with the weights of the node set to 1(undirected graph) for all.
5. If the distance measured is smaller than that of the current one then the distance is updated and the current node is set as the head node which is termed as Dijkstra's central idea.
6. The node is set with the temporary distance as minimum and the distance is marked as permanent.
7. The step from 4 to 7 is repeated until any nodes are left with the permanent distance. But, the existed neighbor still have the distance as temporary.

2.1.3 SCENARIO 3 :Shrinking friendship graph (Removals)

	Time Taken	
	Adjacency List	Adjacency Matrix
Removing Vertex	58.8324358	1.894169
Removing Edges	0.9131	0.030524578

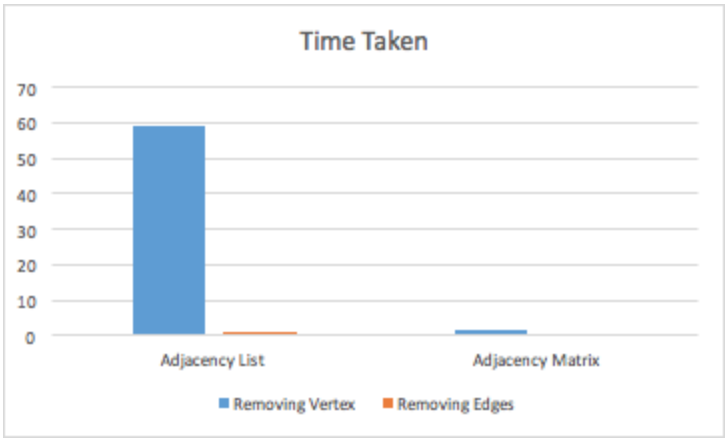
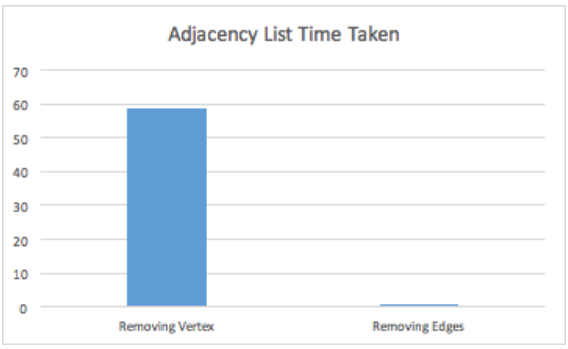
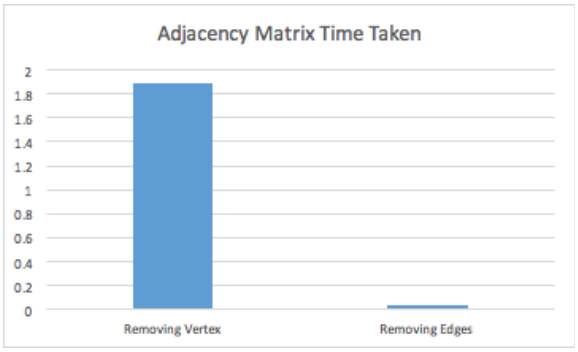


Figure:3 Performance of data structures in Scenario3

For removing the vertex and removing the edges , the adjacency matrix is taking less time compared to adjacency list. Where it is noted that Removing Edges takes almost equal time in both Adjacency matrix and Adjacency List , While removing vertex is taking a huge amount of time in Adjacency List by around 58 millisecond.





Adjacency lists

The most convenient way to list the vertex in the adjacency list is to be in increasing order.
In an undirected graph, vertex jj is in vertex ii's adjacency list if and only if ii is in jj's adjacency list. Adjacent list allows us to store graph in more compact form, than adjacency matrix, but the difference decreasing as a graph becomes denser. Adjacent list allows to get the list of adjacent vertices in O(1) time, for some algorithms.Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix. It requires, on the average, O(|E| / |V|) time, which may result in cubical complexity for dense graphs to add all edges.Adjacent list doesn't allow us to make an efficient implementation, if dynamically change of vertices number is required. Adding new vertex can be done in O(V) , but removal results in O(E) complexity.

Adjacency matrices

For a graph with |V|∣V∣ vertices, an adjacency matrix is a |V| \times |V|∣V∣×∣V∣ matrix of 0s and 1s, where the entry in row ii and column jj is 1 if and only if the edge (i,j)(i,j)is in the graph. With an adjacency matrix, we can find out whether an edge is present in constant time, by just looking up the corresponding entry in the matrix. Adjacency matrix is very convenient to work with. Add (remove) an edge can be done in O(1) time, the same time is required to check, if there is an edge between two vertices. Also it is very simple to program and in all our graph tutorials we are going to work with this kind of representation. Adjacency matrix consumes huge amount of memory for storing big graphs. All graphs can be divided into two categories, *sparse* and *dense* graphs. Sparse ones contain not much edges (number of edges is much less, that square of number of vertices, $|E| << |V|^2$). On the other hand, dense graphs contain number of edges comparable with square of number of vertices. Adjacency matrix is optimal for dense graphs, but for sparse ones it is extra.

3. CONCLUSION:

From the experiment it's clear that overall adjacency matrix performs better in almost every case that we have experimented. However, Adjacency list also had a good perform while adding vertexes ,removing edges or showing the neighbors. Due to the space requirement of adjacency matrix its is much optimal for a dense graph. Nevertheless, in the adjacency list the number of vertices are changed more efficiently in a structured way thus it can be a good solution for a sparse graphs.

4.REFERENCES:

- **http://www.algolist.net/Data_structures/Graph/Internal_representation**
- **http://www.boost.org/doc/libs/1_61_0/libs/graph/doc/adjacency_list.html**
- **http://www.boost.org/doc/libs/1_61_0/libs/graph/doc/adjacency_matrix.html**
- **http://algs4.cs.princeton.edu/41graph/**
- **lecture note**