Algorithms and Analysis
(Joshua Besei – s3839264 & Jacqueline Ann Co Lim – s3753072)

Data and Experimental Setup

Graph Generation

For the implementation of epidemic modelling, we had generated both Erdos-Renyi (random) and Scale-free graphs with three different sizes and average degrees for each type. For each size and average degree, we need to generate three graphs. The size of a graph is determined by the number of vertices present in the graph. Since we need to evaluate the performance of some scenarios depending on the different sizes of the initial graphs, we decided to have 100 vertices as the LOW size, 1000 vertices as the MEDIUM size, and 5000 vertices as the HIGH size. We initially had 10,000 vertices as the high size, however, it took too long to read and generate the data, thus we changed it to 5000 vertices instead.

In terms of the average vertex degrees, we had implemented different sets for Erdos-Renyi and Scale-free graphs. The average vertex degrees tested is only within the range from 1 to 6, since we want to limit the number of edges in the graphs. If we would have a very large average vertex degree, then it would take so much time generating all the data considering that it will have a huge number of edges. In addition, we ensure that the three different average degrees for each type of graph are not too close from each other to see significant differences in the data results. Hence, for Erdos-Renyi and Scale-free, we chose 1 as the LOW average degree, 3 as the MEDIUM average degree, and 6 as the HIGH average degree.

We chose Erdos Renyi over Scale-free graphs for the data scenarios implementation. This is because Erdos Renyi graphs are more randomised and have evenly distributed vertex degrees regardless of the graph size, as compared to scale-free. Hence, when we would add or delete edges, it would have a fair amount for each graph. Furthermore, since the number of edges generated for Erdos Renyi graphs are more consistent, the data results are more likely to be accurate and easier to evaluate.

Data Scenarios

For the data scenarios, the number of operations to be executed is dependent on the size of the graph (number of vertices). In fact, we had decided to test 10% of the graph size. For the LOW-sized graph, we had evaluated 10 operations, as it has 100 vertices. Likewise, for the MEDIUM-sized and HIGH-sized graph, 100 and 500 operations were executed, as it has 1000 and 5000 vertices, respectively. We chose 10%, as usually in statistics, the standard sample size is 10% of the population.

For the k-Hop Neighbourhood scenario, we had only used one graph size, which is medium, since we were only evaluating its performance based on the different average degrees and the values of k. While for the other two scenarios, particularly Dynamic Contact Conditions and Dynamic

People Tracing, we evaluated the performance based on the different sizes and average degrees of the graphs.

Data Generation

Our data generation had a multiple of steps, being the data generator needed to take an input from a graph and load this graph into a Linked List using the util library for the generation to check duplicate commands such as AE/AV as these commands cannot have a duplicate that was already created in the initial graph. We relied on the code from the RMIT Covid Modelling class to assist us with loading this graph but instead of adding them to a new graph we instead use it get information to create the data generation.

The data generation takes in parameters regarding which command to generate, and if it is a KN command a supplied int alongside it called. It also takes a parameter for sample size.

All Pseudocode has been placed in the appendices under relevant headings to relate too.

- **K-Hop Neighbourhoods**

For K-Hop Neighbours before we begin the data generation, we check if the argument read in was KN and a valid K-Hop int was read as the second argument. We can then set the generation start range to 1, and set the end range to the length of the total vertices or the graph that was read in. We then begin the generation of K-Hop Neighbours. As K-Hop Neighbours and AV and DV share the same pseudocode for the generation for test data we will move the pseudocode under its own heading after the explanation of AV and DV.

- **Dynamic People Tracing (Vertex Addition and Deletion)**

For Vertex Addition and Deletion, a similar approach was taken with K-Hop Neighbours however the only thing which differs is the command argument. Once the command argument has been set, we can set the start and end of the range. Deletion takes the K-Hop approach however Addition takes the approach in getting values only outside the range of the total vertices the initial graph read in contains. This means the start of the range becomes the total vertices plus one and the end of the range becomes total vertices times 2. This gives us an extensive range that will not give us a duplicate since the initial graph contains these AV commands already. We then can set the sample for both addition and deletion by generating the test data using the sample size argument and do a similar looping approach.

- **Dynamic Contact Conditions (Edge additions and deletions)**

For Edge Addition and Deletion, the approach we decided to take is different from the rest as the print statement requires two randomized numbers rather than one. For addition we need to be able to check for duplicates and for deletion we need to be able to check for initial additions to delete.

- **Explanation for 'Get the generation test data using sample size argument' Pseudocode**

The generation portion of the code is used to generate values. The confusing part regarding the code is the get next int however this can be explained logically. The data generator constructor creates a variable called mRandGen which is a new reference using the system's time as the

current seed for the creation of the variable. Get next int is a method for the Random class and is called using a parameter var plus one. As such we use this method to generate random numbers which do not produce duplicates and are within range of the sample size for us not to receive randomly generated integers that fall below this size.

Fixed Sets

- **K-Hop Neighbourhoods**

For K-Hop Neighbourhoods we decided to gather a range of randomly generated values between the start of range equalling to one and end of range equalling to the total amount of vertices from the initial graph loaded. We did it like this as we want to randomly generate a value someone between the initial graph as we do not want any randomly generated values outside the range, hence we have no invalid values that fail. We also want to check for duplicates so we can have a test which is completely random and no two share the same value.

- **Dynamic Contact Conditions (Edge additions and deletions)**

For edge addition the same principle applies when generating a range similar with K-Hop Neighbourhoods however the only difference is that the generation required two values rather than a single one. Even so, this required us to check for duplicates as well since we did not want any duplicate edges already in the initial graph and we wanted new edges to be added. For edge deletion we only had the set that was given to us by the initial loaded graph, hence the only valid deletions were ones that were loaded so no generation or random numbers were needed rather we had to randomly select a position in the loaded graph to use for deletion.

- **Dynamic People Tracing (Vertex Addition and Deletion)**

For the deletion of a vertex the K-Hop principle is more or less identical apart from the outputted print command. However, for the addition of a vertex is the only command which gets values that are out or range. The fixed sets we determined that would be valid were the start of range would be the total amount of vertices plus one and the end of range would be the total amount of vertices times two. This meant for an initial graph of 100, the valid options would be 101 to 200. As we chose to get 10% for sample size this left us with plenty of random choices to choose from, especially for initial graphs we used where there were 5000 vertices.

Approach Used for Measuring Time Results

For the measuring of time results we modified the main run class and implemented two variables for each command, one which receives the current runtime when read and once the command completes another variable which gets the current runtime. This means we have an initial (start time) and a final (end time) to receive an execution time of final minus initial. We can print this out to get the execution time. However, we went that one step further and attempted to receive the average execution time for the test ran, hence the last printed execution time was the average of all executions completed during the test. We stored the final minus initial in a variable and continued to add to this variable, while also counting each iteration, and printed the variable

divide iteration. This meant the final printed value was the average execution time of all commands ran in the test.

Evaluation of the Data Structures

Scenario 1 k-hop Neighbourhoods:

For this scenario, we had evaluated the performance of k-hop neighbourhood implementations as the average vertex degree and the number of hops (value of k) of the initial graph varies for each data structures. As illustrated in Figure 1.1, Figure 1.2, and Figure 1.3, the graph with the HIGHEST k-value and HIGHEST average vertex degree took the longest time to execute the k-Hop neighbourhood implementation, while the ones with the LOWEST k-value runs the fastest. This simply implies that both average vertex degree and the value of k have an impact on the performance of the k-Hop neighbourhood implementation. In fact, in the implementation of k-hop for three data structures, it needs to find which edge is connected to the vertex to determine its k-hop neighbours. Thus, the greater number of edges to check, the longer time it takes to execute. In addition, the k-value can also affect the performance, as for the implementation, the number of iterations is dependent on the value of k. For instance, if the value of k is 5, then the implementation will loop at least 5 times until all the neighbours are stored in the array. In short, what our k-Hop implementation does is that it goes 1-hop from one to another for each iteration.

For Figure 1.4 and Figure 1.5, we had calculated the average execution time to compare the performance of the three data structures based on the different average degrees and different k-values. Based on Figure 1.4 and Figure 1.5, we can determine the efficiency of the data structures as the average vertex degree and k-values increases. In fact, for adjacency list, it took approximately 0.022844275 seconds when performing the k-Hop implementation for the graph with the HIGHEST average vertex degree and HIGHEST k-value. For adjacency matrix, it took approximately 0.015399055 seconds to execute the same type of graph. Whereas for incidence matrix, it took approximately 0.098297696 seconds. Hence, based on these results, the fastest is the adjacency list, followed by the adjacency matrix, and the slowest is incidence matrix.

However, the execution time of the three data structures may seem to have slight differences from each other. This is because the implementation for k-Hop is quite similar among all the three data structures. On the other hand, the curve line in Figure 1.4 and Figure 1.5 is different for all data structures. For both adjacency list and adjacency matrix, the time complexity for k-Hop implementation is linear, which is O(n), since it only needs to iterate for the number of hops. Whereas the incidence matrix iterates both number of hops and loop the edges, which are the columns, to determine the neighbours of the vertex. Hence, for the incidence matrix, the results form a curve line, which can be a quadratic. This may suggest that the time complexity of the incidence matrix is more expensive, as compared to the other two data structures when executing the k-Hop implementations.

Scenario 2 Dynamic Contact Conditions:

Edge Addition:

For the Dynamic Contact Conditions, we had evaluated the performance of edge addition operations as the average degree and size (number of vertices) of the initial graph are different. Based on Figure 2.1, for adjacency list, the graph with the LOWEST size (number of vertices) and the LOWEST average vertex degree has the shortest execution time when adding edges. In contrast, the graph with the HIGHEST size (number of vertices) and the HIGHEST average vertex degree has the longest execution time. As such the add edge for incidence matrix (Figure 2.3) and adjacency matrix (Figure 2.2) have identical results as the adjacency list. In fact, for all the three data structures, the execution time for add edge increases, as the average degree and size of the graph increases. This is because the program, which all data structures are being implemented, checks whether the vertices, particularly source vertex and target vertex, that are connected by the edge exist before adding the edge. In addition, it also checks whether the edge already exists in the graph.

However, despite having the same trend, the amount of time varies for each data structure. For adjacency list, the graph with the HIGHEST size and HIGHEST degree took approximately 0.009496 seconds. For adjacency matrix, the graph with the HIGHEST size and HIGHEST degree took approximately 0.014495033 seconds, which is close to adjacency list. For incidence matrix, the graph with the HIGHEST size and HGHEST degree took approximately 185.3634 seconds, which has an extremely huge difference from the adjacency list and adjacency matrix. Overall, we can conclude that the adjacency list and adjacency matrix have roughly the same execution time, while the incidence matrix takes a longer time to perform the edge addition operations. This data results can be explained with the known theoretical time complexities.

As illustrated in Figure 2.4, we had calculated the average execution time for each data structure as the average vertex degrees are varied, while in Figure 2.5, it also shows the average time execution of the data structures as the size (number of vertices) increases. Based on Figure 2.4 and Figure 2.5, the performance for both adjacency list and adjacency matrix is constant. This implies that the time complexity of adjacency list and adjacency matrix for ADDING EDGES is O(1). In fact, this is reasonable since when adding an edge for the adjacency list, it is only adding the target vertex to the source vertex's LinkedList, as well as adding the source vertex to the target vertex's LinkedList. Likewise, for the adjacency matrix, it is only setting the bit of the source vertex and target vertex to 1. Hence, both data structures do not need to perform iterations when adding edge. On the other hand, for incidence matrix, it forms a curve as the size and average degree increases. This may suggest that the running time of incidence matrix is slow. Since an incidence matrix is made up of vertices as its rows and edges as its columns, then the time complexity is more likely to be O(VxE). This is because the incidence matrix must copy and iterate all the existing elements (both vertices and edges), then add a new column for the new edge. Thus, it is expensive to perform add edges on incidence matrix, as compared to adjacency list and adjacency matrix.

Edge Deletion:

Aside from edge addition, we also evaluated the performance of the delete edge operations as the average vertex degree and size are varied. As similar with the edge addition, the trend seems to be identical among the data structures. As the initial graph size grows, the execution time per edge deletion operation increases, as there are more vertices, as well as higher sizes contains more edges. With the graph degree, the execution time varies in that a high degree takes an extraordinary amount of time in comparison to a low degree graph. In fact, the execution time increases when there is a larger initial graph size or a larger average vertex degree, as there are more edges within a graph to check which exist prior to deletion. The larger the list, the longer the implementation is required to search and remove.

Based on Figure 2.6, for adjacency list, the graph with the HIGHEST size and HIGHEST average vertex degree took approximately 0.0000223 seconds. Based on Figure 2.7, for adjacency matrix, the graph with the HIGHEST size and HIGHEST average vertex degree took approximately 0.0000317 seconds. Based on Figure 2.8, for incidence matrix, the graph with the HIGHEST size and HIGHEST average vertex degree took approximately 2.67240595 seconds. Overall, similar as the situation for edge addition operations, incidence matrix runs the slowest when deleting an edge, as compared to both adjacency list and adjacency matrix.

As shown in Figure 2.9 and Figure 2.10, the three data structures had been evaluated based on the average execution time as the average vertex degree and size increases. The data results are somewhat similar with the edge addition. For both adjacency list and adjacency matrix, it shows a constant function, which may imply that the time complexity for both data structures is O(1). This is plausible for adjacency matrix since it is merely setting the bit of source vertex and target vertex to 0. However, the time complexity for adjacency list should be O(V) when deleting an edge since it needs to iterate and search for the vertices (particularly both source vertex and target vertex) that are connected to the edge before removing them from each other's LinkedList. Our implementation for the adjacency list might have an impact to the results, as we merely get the index positions of the vertices (source vertex and target vertex), then remove them from each other's LinkedList. Hence, no iteration was executed for our implementation, resulting in to have O(1) as its time complexity for adjacency list. On the other hand, as mentioned previously, incidence matrix runs the slowest. This is because it needs to copy all the existing elements while excluding the column of the edge to be deleted. Thus, as similar with edge addition, the time complexity of incidence matrix when deleting an edge is O(VxE).

Scenario 3 Dynamic People Tracing:

Vertex Addition:

For Dynamic People Tracing, we had evaluated the performance of vertex addition operations as the average degree and size of the initial graph varies. As explicitly shown in Figure 3.1, 3.2, and 3.3, the execution time for each data structure grows larger, as the size and average degree

increases. This simply means that it takes a longer time to execute the vertex addition operations when there are a greater number of vertices in the initial graph. The size of the graph has an impact on the execution time of the vertex addition operations since the program checks whether the vertex already exists or not before adding it to the data structure. In addition, we can notice that the execution time difference among the different average degrees is huge for the HIGH-sized graph, as compared to the LOW-sized and MEDIUM-sized ones. This may suggest that the number of edges can also affect the performance of the vertex addition operation, especially for adjacency matrix and incidence matrix.

We can determine which data structure runs the slowest and the fastest through the data we got for the graph with the HIGHEST size and HIGHEST average vertex degree. For adjacency list, the graph with the HIGHEST size and HIGHEST average vertex degree took approximately 0.000262 seconds, while for adjacency matrix, it took approximately 0.109058935 seconds. For incidence matrix, it took approximately 0.396090769 seconds. If we are going to rank the three data structure based on its efficiency on adding a vertex, the fastest is adjacency list, followed by the adjacency matrix, and the slowest is the incidence matrix. In Figure 3.4, it shows a summary of how the different average vertex degrees have effect on the performance of the three data structures, while in Figure 3.5, it displays the average execution time of the three data structures as the size of the initial graph increases.

Based on Figure 3.4, both adjacency list and adjacency matrix performed constantly regardless of the average degree, whereas the execution time for incidence matrix became slower as the average degree increases from 1 to 3. This may mean that the number of edges does not impact the execution time for both adjacency list and adjacency matrix when adding a new vertex. In fact, for adjacency list, we would only iterate the vertices and not the edges when performing the vertex addition. Likewise, for adjacency matrix, we would only loop the vertices as well, hence the number of edges does not really affect both data structures when adding a new vertex. Unlike the implementation for incidence matrix, which is made up of vertices as its row and edges as its column, we still need to check if there are edges exist and add a new row to every column for the newly added vertex.

Based on Figure 3.5, we evaluated the average execution time of vertex addition operation for the three data structures, as the size (number of vertices) of the initial graph increases. Since we are performing vertex addition, the number of vertices can affect the execution time for some of the data structures, especially for matrices. For adjacency list, as shown in Figure 3.5, the execution time remains constant despite that the size of the graph is increasing, thus the time complexity for adjacency list is O(1). This is probably because the adjacency list only adds a new vertex as the last element of the row. For our implementation, we only add the new vertex to the array of vertices, then create a new LinkedList for it to store the edges that are associated with it. Unlike in the adjacency matrix and incidence matrix, it needs to iterate both rows and columns to add the new vertex. Since adjacency matrix is made up of vertices for both rows and

columns, it needs to iterate n x n, and create a new row and column for the newly added vertex. Hence, the time complexity for adjacency matrix when adding a vertex is O(V^2), while for incidence matrix, it needs to iterate the rows which consists of vertices and add a new row of 0's to every column for the newly added vertex, thus the time complexity of vertex addition for incidence matrix is O(VxE).

Vertex Deletion:
As similar with the results for vertex addition, the execution time of vertex deletion operations also increases, as the average degree and size of the initial graph increases, as shown in Figure 3.6, 3.7, and 3.8. This is because for our implementation of all the data structures, it needs to check whether the vertex exists in the graph before performing the vertex deletion operation.

Similar with the vertex addition, we can merely compare the performance of the three data structure by evaluating the time operated to delete a vertex for the graph with the HIGHEST size and HIGHEST average vertex degree. For adjacency list, it took approximately 0.001021632 seconds, while for adjacency matrix, it took approximately 0.091893074 seconds. Whereas for the incidence matrix, it took approximately 0.417685303 seconds. Therefore, according to these data results, the fastest is the adjacency list, followed by adjacency matrix, and the slowest is incidence matrix. For Figure 3.9 and 3.10, we had compared the average execution time of the three data structures based on the different average vertex degrees and sizes, respectively.

For Figure 3.9, similar with the results from the vertex addition, the average execution time for both adjacency list and adjacency matrix is constant as the average degree varies. This may suggest that the number of edges does not have any influence on the performance of both adjacency list and adjacency matrix when deleting a vertex. For adjacency matrix, it only needs to iterate the vertices when deleting a vertex. We may think that there are edges associated with the vertex, but for adjacency matrix, once the rows and columns of the vertex are deleted, then all the edges that are connected to that vertex will also be deleted since it is just sets of bits of 0's and 1's. Whereas for adjacency list, the edges should be involved when deleting the vertex, but our implementation might have affected the results. In fact, for adjacency list, when deleting a vertex, it needs to iterate all the vertices and remove the vertex from the other vertices' LinkedList as well. However, for our implementation, we merely get the position of the vertex to be removed, then delete the vertex from the others' LinkedList, which is still odd, as our implementation also performed iterations, but the results turn out to be constant for adjacency list. For incidence matrix, it is reasonable that the execution time increases as the number of edges increases, since it needs to iterate the columns and delete the edges that are connected to the vertex to be deleted.

For Figure 3.10, we had evaluated the average time performance of vertex deletion for the three data structures as the size of the initial graph increases. For adjacency list, the execution time remains constant, hence it is suggesting that the time complexity is O(1). However, it should be

O(V+E) since when deleting a vertex from the adjacency list, it should iterate all the vertices and remove the vertex to be deleted. As what was mentioned in the previous section, our implementation might have affected the results, as we merely get the index position of the vertex to be deleted, hence it performed faster than expected. For adjacency matrix and incidence matrix, this is reasonable, as for adjacency matrix, it needs to iterate both rows and columns while excluding the row and column of the vertex to be deleted, hence it is iterating n x n. Thus, as shown in the Figure 3.10, it has a curve-like for adjacency matrix, which we can assume that it is forming a quadratic curve, and the time complexity is O(VxV). Moreover, for the incidence matrix, it is also forming a curve, which may also be quadratic. This is because the incidence matrix must iterate both rows and columns as well. It loops the rows to remove the vertex to be deleted, as well as loop the columns to remove the edges that are associated with the deleted vertex.

Summary of the Analysis

Overall, with the range of tests completed we acquired a good understanding of our own implementation and the overall effect a graph degree and size can have in the effectiveness and efficiency of reading and writing data. For adding an edge and removing, adjacency list and adjacency matrix were by far the more efficient in comparison to incidence matrix. However, in terms of memory allocation, adjacency list would be a clear winner as the adjacency matrix requires much more memory to store its matrix while the linked list implementation for adjacency matrix proved to be quite efficient. Also, adjacency list would prove to win slightly since the matrix already contains the edge's pre-set to zero and only requires flipping this bit, while adjacency list requires us to add the vertex to the linked list. Even so they are very close in terms of speed. With vertex addition and deletion adjacency list proves to be faster since it only requires an addition of the vertex in the array while adjacency list must not only add the vertex but set the newly added row and newly added column to zero to represent no edges for this new vertex. For K-Hop we only tested average degree and not size, so we could not compare total memory required however adjacency list ended up being the fastest as however adjacency matrix came very close. The reason adjacency list was faster was mainly due to our implementation of all the methods in each class coordinating with each other to produce an accurate result. In conclusion we recommend using adjacency list as the data structure due to the overall time execution of per instruction and the memory efficiency of the data structure is quite exceptional in comparison to the matrixes implemented.

SIR Evaluation

Most parameters that were tested seem to have an overall fast spread however the fastest would be an infection probability which is high. In our testing these parameters that spread quickly were the ones that has an infection probability of 0.75 and 1. Thus, among all the 18 different parameter combinations (graph type, infection probability, and recovery probability), then the one that has the fastest infection spread is the scale-free graph with an infection probability of 1 and recovery probability of 0.5 (as shown in Figure 4.10). Likewise, this combination also

generated the most extensive number of infected vertices among all the tests. In addition, having an infection probability of 0.5 was also relatively fast since most tests after around 7 days the susceptible would drop close to its all-time low.

Even though we anticipated that a lower probability would produce a smaller number of infected, this is not necessarily true. The reasoning behind this is quite simple in that a lower infection probability such as 0.25 can infect neighbours just as fast since the infection rate is exponential. This is especially the case if each vertex has contact with multiple other vertices. Hence, the infection spreads faster in the scale-free graphs than the Erdos-Renyi (random) ones. Since we tested our parameters against a graph of 1000 vertices with an edge degree of 6, each vertex has an infection probability of 0.25. This means with a degree of 6 on average a vertex can infect 1.5 vertices (which is 25% of the total vertices). In a real-world situation this would not stop coronavirus since if each vertex would infect 25% of the population, coronavirus would continue to gain traction. Rather decreasing the probability is one step to help solve coronavirus, and a solution would be quarantining, which many countries have implemented. Looking to the edge degree, we only tested a degree of 6 however with a degree lower, this would mean each vertex has less contact with other vertices. Having less contact can stop spreading the virus through human contact, and if the degree were only lowered to 3 with the same probability a single vertex would infect 0.75 vertices. This would help prevent the spread of coronavirus from continuing. In a real-world situation this can be seen in the form of masks, social distancing and limiting the amount of people in public venues. Overall, our testing proved to show a situation when government intervention was very little however a degree of 3 with a lower probability is more likely today with action taken to prevent the spread of coronavirus.

As part of our analysis, we evaluated the average time performance of the three data structures when operating the SIR model with the random graph that has 1000 vertices and an average degree of 6. As illustrated in Figure 4.19, adjacency list runs the fastest that only took 0.094125 seconds, followed by the adjacency list which took 0.102684 seconds, and the slowest one is the incidence matrix which took 0.360471 seconds. All the average timings have been obtained after executing 9 tests (SIR commands) for each data structure. As we can notice, the average timing is quite similar with the k-Hop Neighbourhood, since for our implementation of the SIR model, it calls the k-HopNeighbours() and toggleVertexState() functions from the selected data structure (i.e., adjlist, adjmat, incmat). The SIR model uses the k-Hop Neighbourhood implementation to determine the close contacts, particularly the neighbours of the infected vertices. Since the implementation for toggle vertex state is the same for all the three data structures, it does not affect the difference of the time performance, hence it may suggest that the results are mainly affected by the k-Hop Neighbourhood implementation.

References

Pseudocode - Computer Science Wiki. (2016). Retrieved April 19, 2021, from Computersciencewiki.org website: https://computersciencewiki.org/index.php/Pseudocode

Appendix

- **Pseudocodes**
    - **K-Hop Neighbourhoods**

High-Level Pseudocode for setting up K-Hop Neighbours generation and printing.

START

Determine if K-Hop Argument is less than 0

Print error message

Otherwise continue

Set generation start of range to 1

Set generation end of range to total vertices

Get the generation test data using sample size argument

Set var to zero

While var is less than length of sample size argument

Print the samples value alongside the command argument and K-Hop argument

Increment var

End while

END

- **Dynamic People Tracing (Vertex Addition and Deletion)**

High-Level Pseudocode for Vertex Addition and Deletion

(GOTO X) will mean that there is a difference, and it will be stated below.

START

    GOTO ADDITION / DELETION

    Get the generation test data using sample size argument

    Set var to zero

    While var is less than length of sample size argument

    Print the samples value alongside the command argument

    Increment var

    End while

END

ADDITION

    Set generation start of range to total vertices plus one

    Set generation end of range to total vertices times two

DELETION

    Set generation start of range to 1

    Set generation end of range to total vertices

END

- **High-Level Pseudocode for 'Get the generation test data using sample size argument'**

START

Obtain sample size

Determine population size by end of range minus start of range plus one

Set samples array to length of sample size

Set var to zero

While var is less than sample size

Set samples position var to var plus start of range

Increment var

End while

Set var to sample size

While var is less than population size

Set value to get next int using var plus one

Determine if value is less than sample size

Set samples position value to var plus start of range

Increment var

End while

Return samples

END

- **Dynamic Contact Conditions (Edge additions and deletions)**

High-Level Pseudocode for setting up Edge Addition

START

Set count to zero

While count is not equal to sample size

Set var array to get new edge

Determine if var is not equal to null

Print the edge array positions one and two alongside its command

Increment count if var is not equal to null

End while

END

High-Level Pseudocode for setting up Edge Deletion

START

Set var to zero

While var is less than sample size

Set edge array to get existing edge

Print edge array positions one and two alongside command

Increment var

End while

END

High-Level Pseudocode for 'Get new edge'

START

Set A and Set B to next int using end of range minus start of range plus one

Set C array to null

Determine if A is not equal to B and A is not equal to zero and B is not equal to zero

Set D array to new string which contains A and B in position respectively

Determine if D is check if exists

Add C to linked list of edges

Return C

END

High-Level Pseudocode for ‘Check if exists’

START

Obtain D

Set check to false

Set var to zero

While var is less than the length of linked list

Set edge array to linked list get position var

Determine if edge position zero is equal to D position zero and edge position one is equal to D position one OR edge position zero is equal to D position one and edge position one is equal to D position zero

Set check to true

Escape while loop early

End while

Return check

END

High-Level Pseudocode for ‘Get existing edge’

START

Shuffle linked list using collections shuffle from collections class

Set edge array to linked list pop

Return edge

END

- **Graphs**
  - **Scenario 1 k-hop Neighbourhoods**

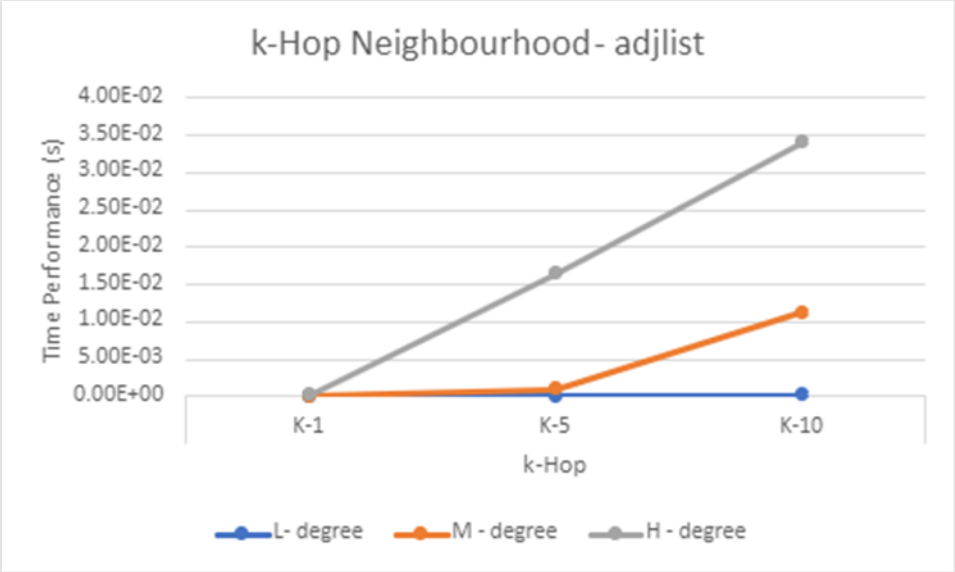

Figure 1.1 k-Hop Neighbourhood for Adjacency List (Time Performance vs k-value per Average Degree)

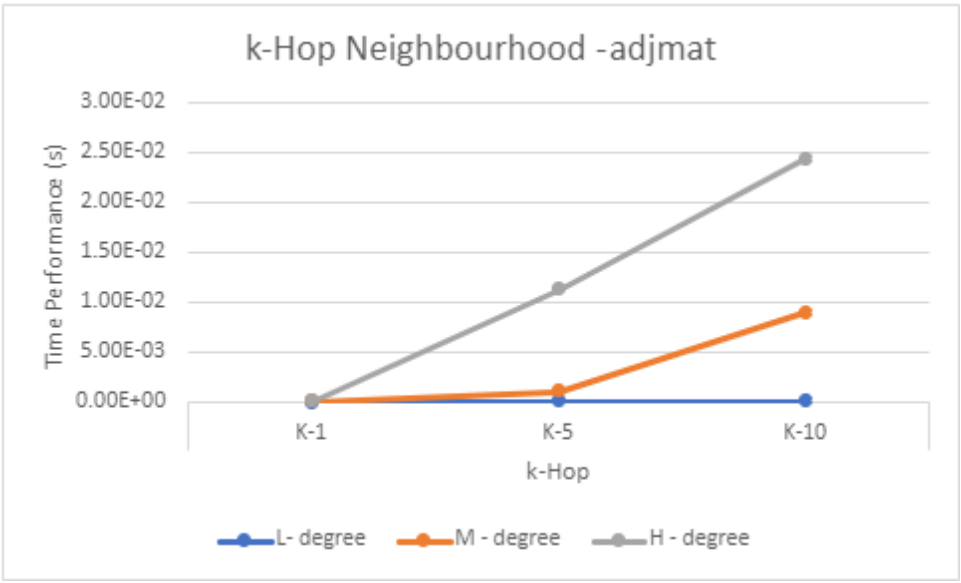

Figure 1.2 k-Hop Neighbourhood for Adjacency Matrix (Time Performance vs k-value per Average Degree)

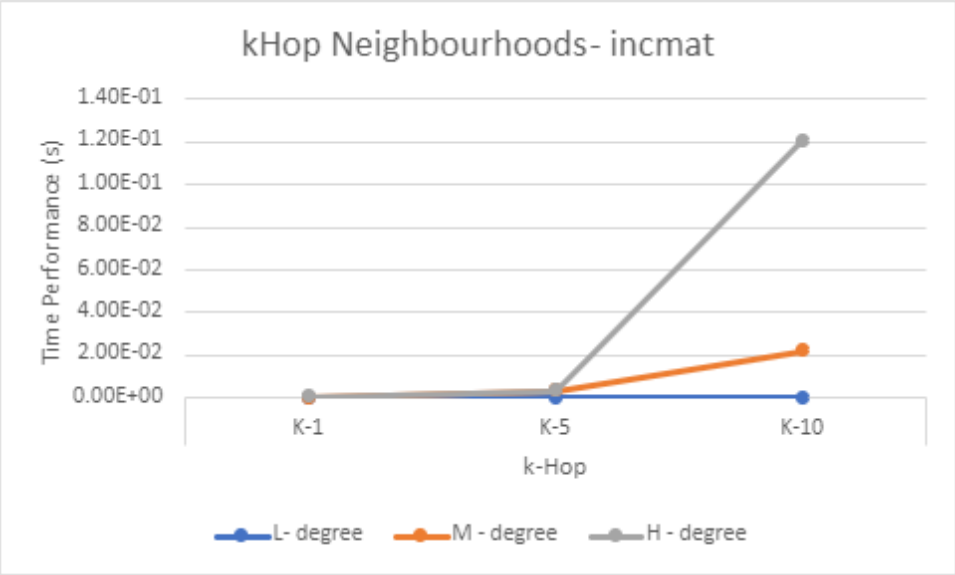

Figure 1.3 k-Hop Neighbourhood for Incidence Matrix (Time Performance vs k-value per Average Degree)

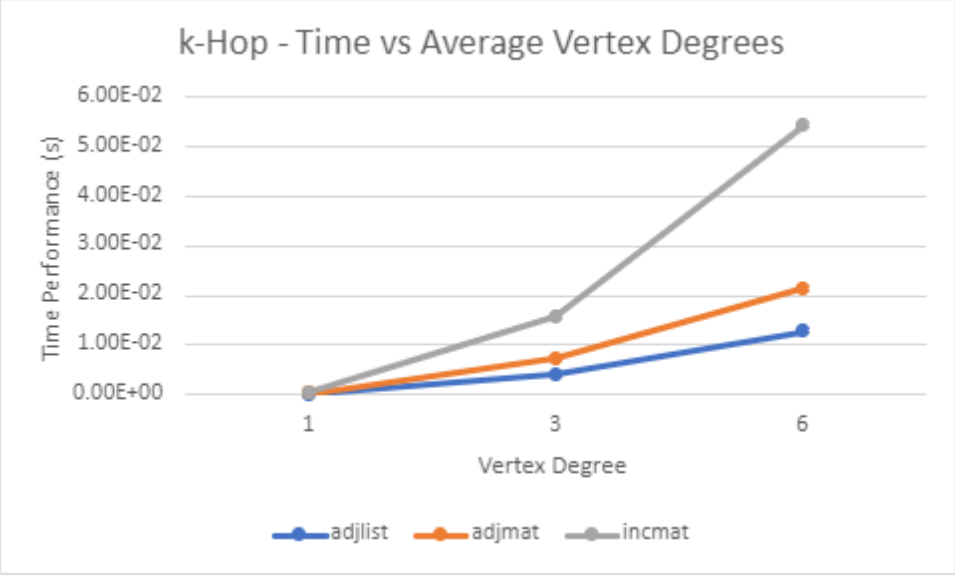

Figure 1.4 Comparison Between the Three Data Structures for k-Hop Neighbourhood (Time Performance vs Average Degree)

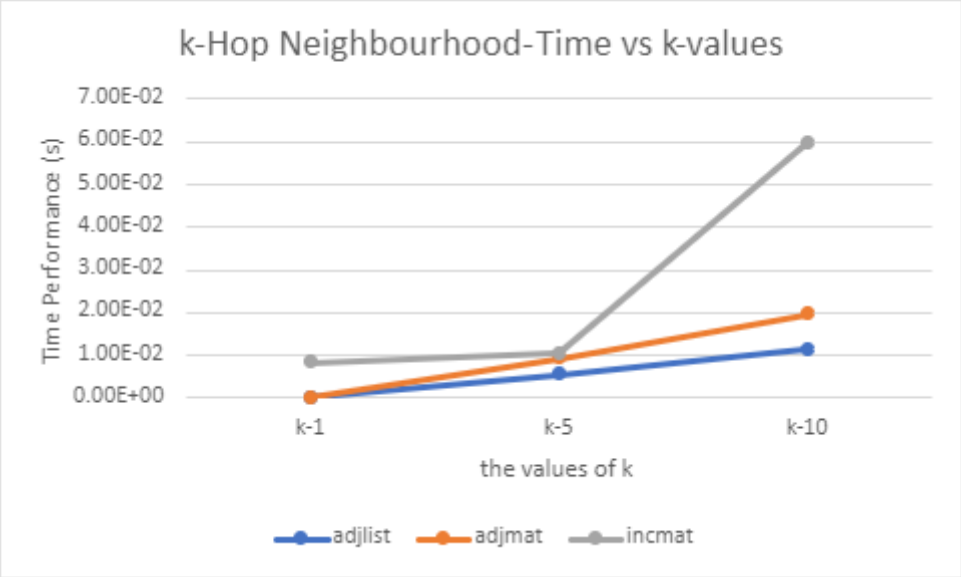

Figure 1.5 Comparison Between the Three Data Structures for k-Hop Neighbourhood (Time Performance vs k-value)

- **Scenario 2 Dynamic Contact Conditions**

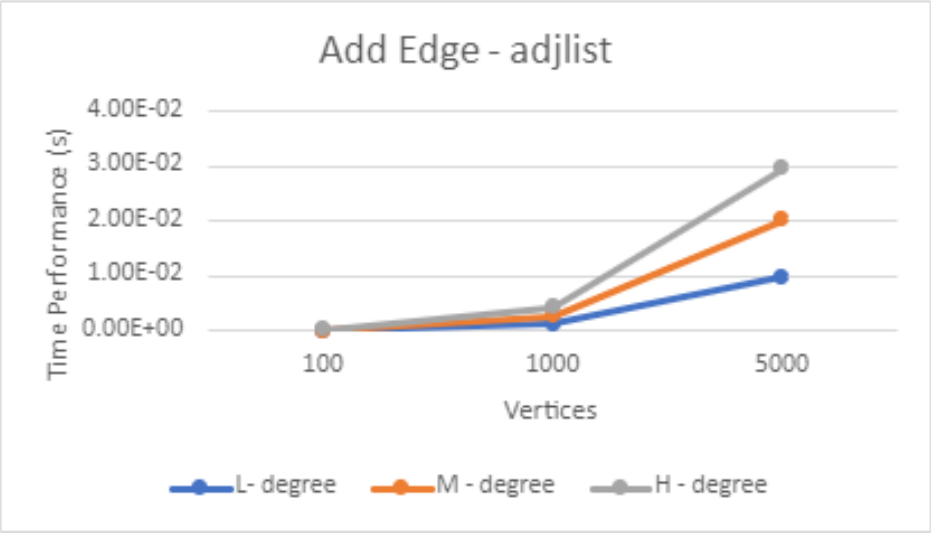

Figure 2.1 Edge Addition for Adjacency List (Time Performance vs Graph Size per Average Degree)

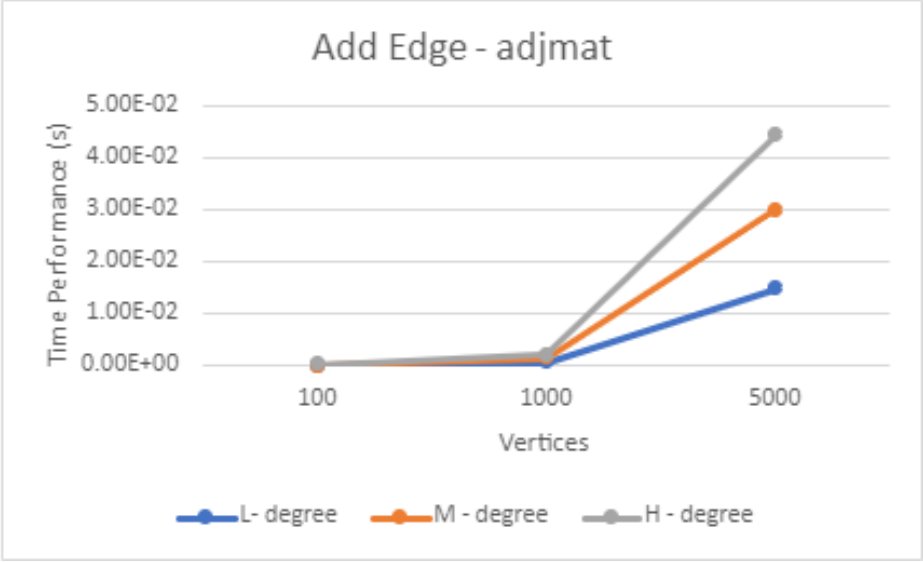

Figure 2.2 Edge Addition for Adjacency Matrix (Time Performance vs Graph Size per Average Degree)

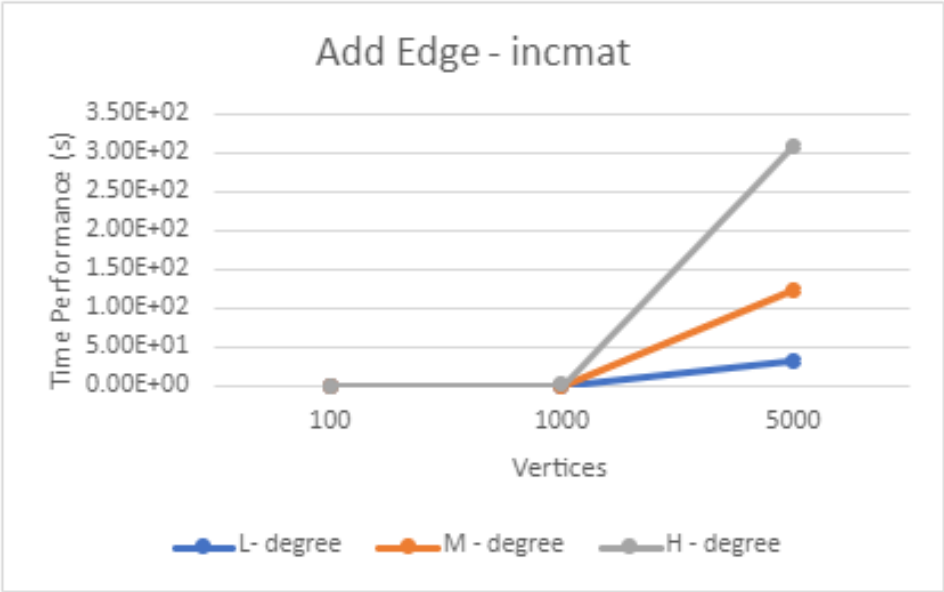

Figure 2.3 Edge Addition for Incidence Matrix (Time Performance vs Graph Size per Average Degree)

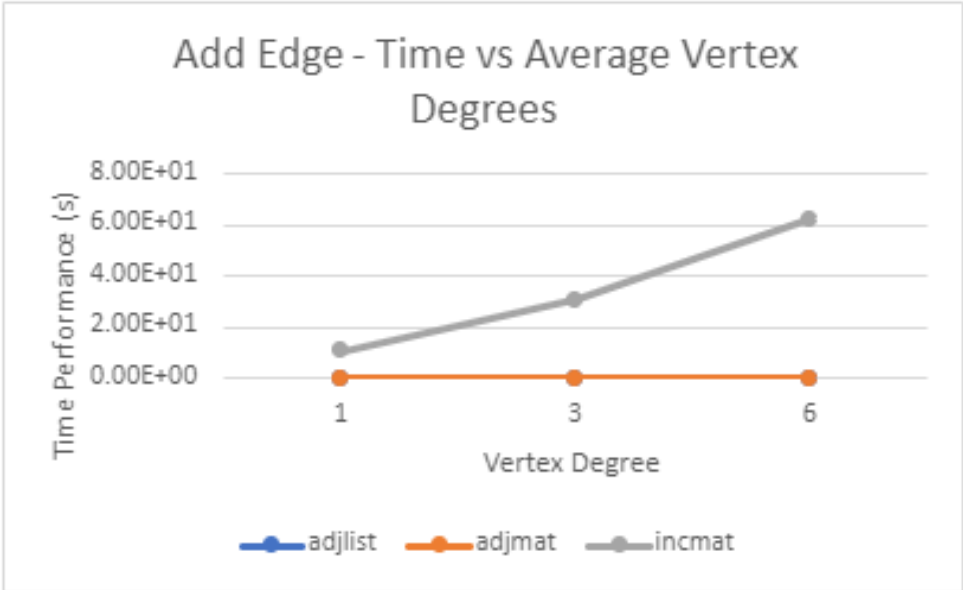

Figure 2.4 Comparison Between the Three Data Structures for Edge Addition (Time Performance vs Average Vertex Degree)

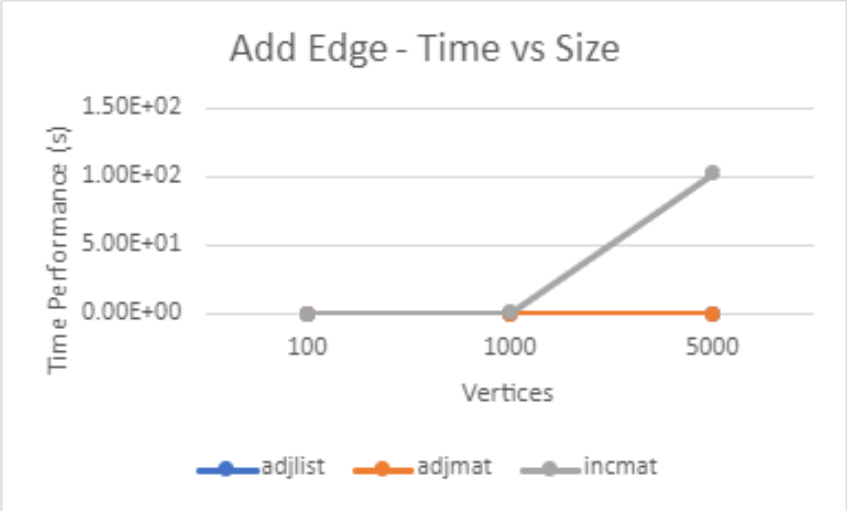

Figure 2.5 Comparison Between the Three Data Structures for Edge Addition (Time Performance vs Graph Size)

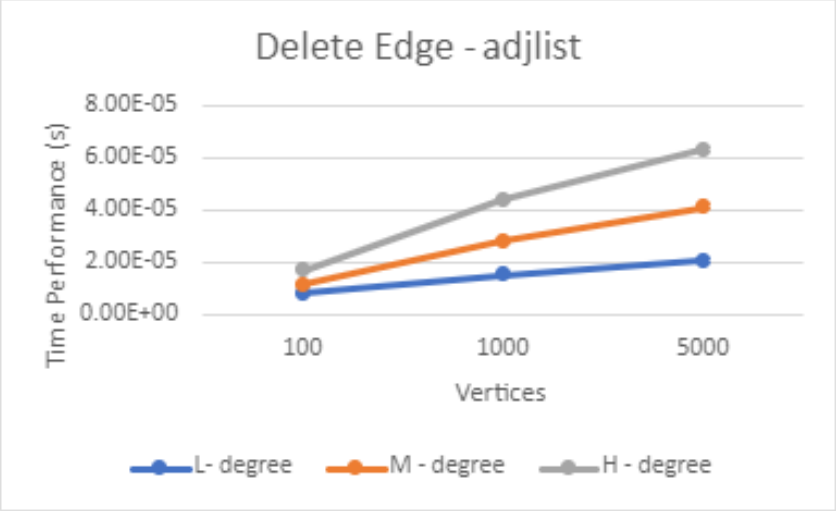

Figure 2.6 Edge Deletion for Adjacency List (Time Performance vs Graph Size per Average Degree)

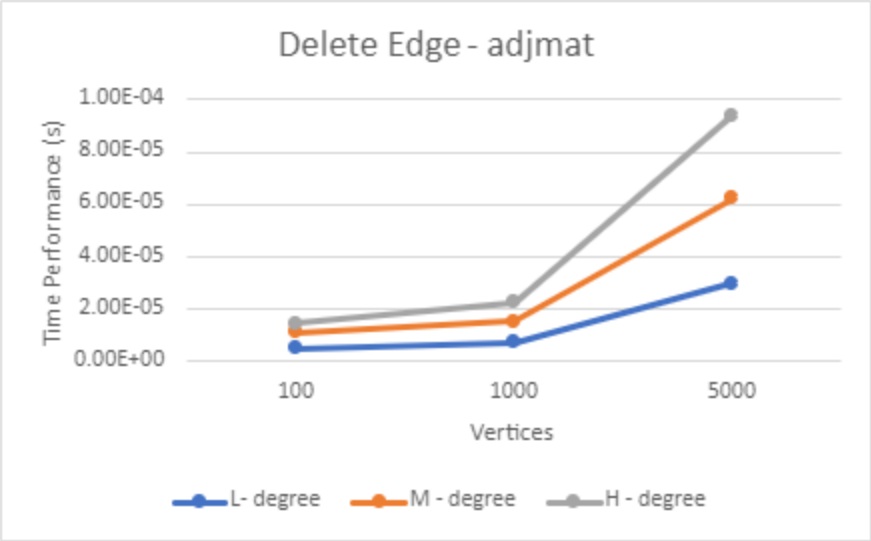

Figure 2.7 Edge Deletion for Adjacency Matrix (Time Performance vs Graph Size per Average Degree)

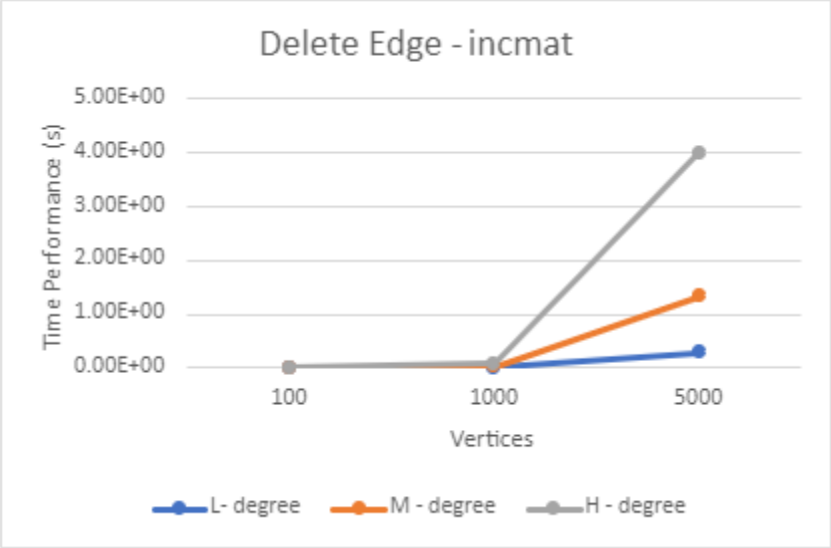

Figure 2.8 Edge Deletion for Incidence Matrix (Time Performance vs Graph Size per Average Degree)

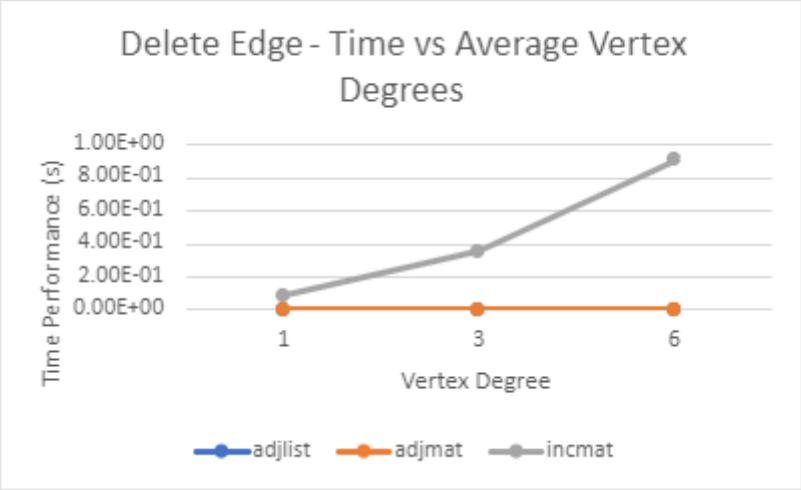

Figure 2.9 Comparison Between the Three Data Structures for Edge Deletion (Time Performance vs Average Vertex Degree)

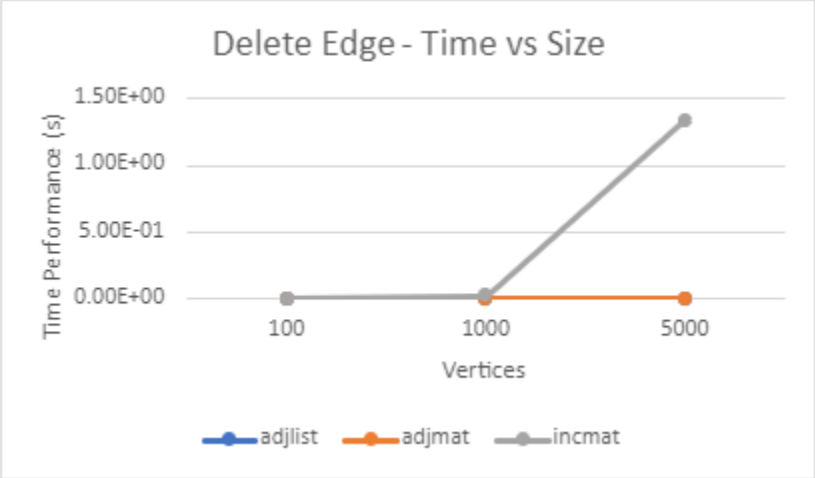

Figure 2.10 Comparison Between the Three Data Structures for Edge Deletion (Time Performance vs Graph Size)

- **Scenario 3 Dynamic People Tracing**

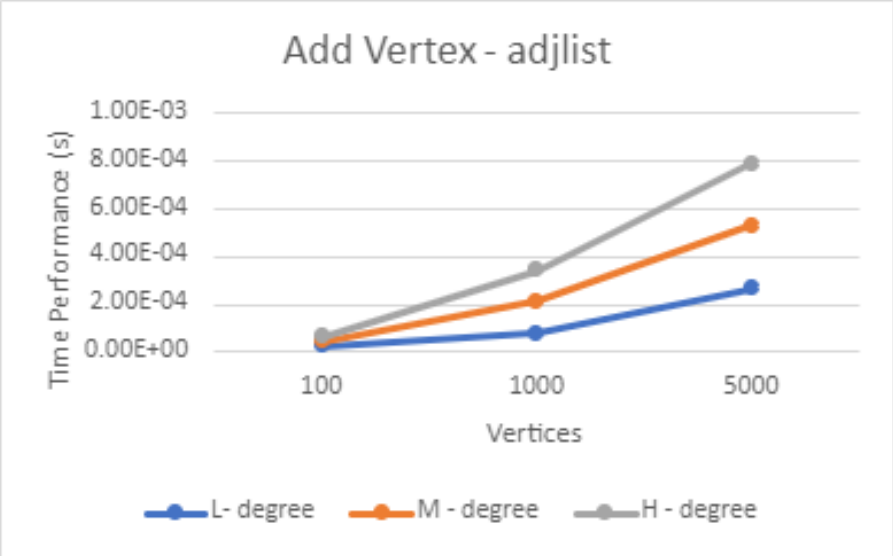

Figure 3.1 Vertex Addition for Adjacency List (Time Performance vs Graph Size per Average Degree)

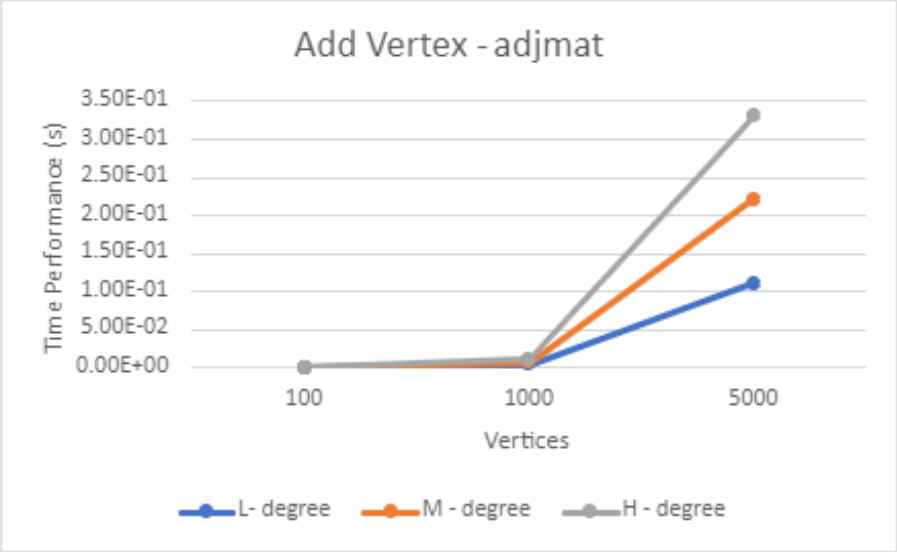

Figure 3.2 Vertex Addition for Adjacency Matrix (Time Performance vs Graph Size per Average Degree)

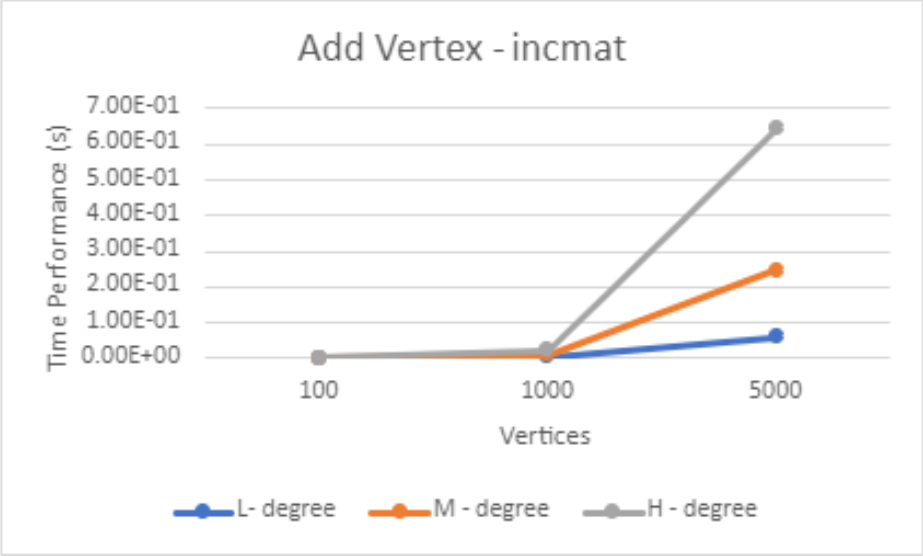

Figure 3.3 Vertex Addition for Incidence Matrix (Time Performance vs Graph Size per Average Degree)

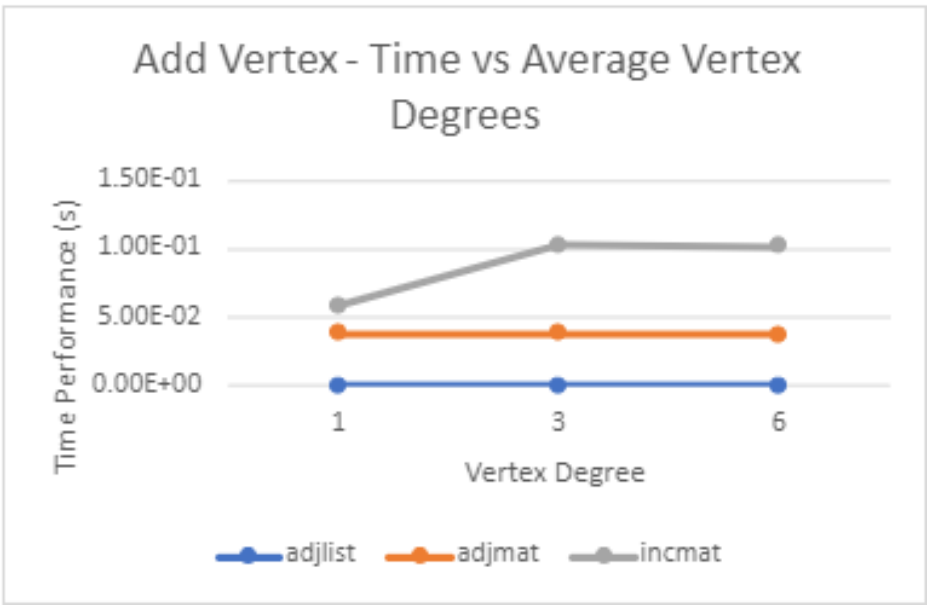

Figure 3.4 Comparison Between the Three Data Structures for Vertex Addition (Time Performance vs Average Vertex Degree)

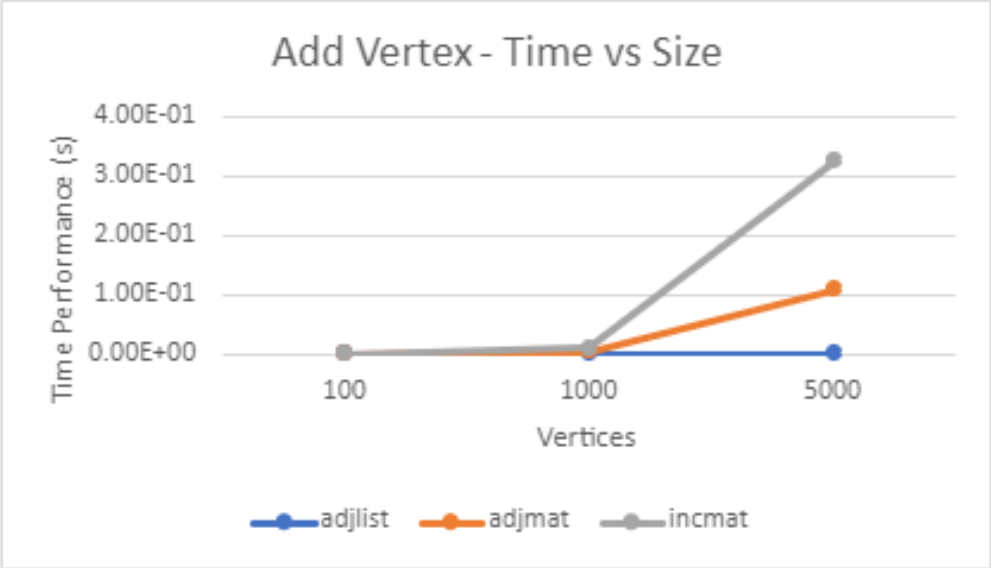

Figure 3.5 Comparison Between the Three Data Structures for Vertex Addition (Time Performance vs Graph Size)

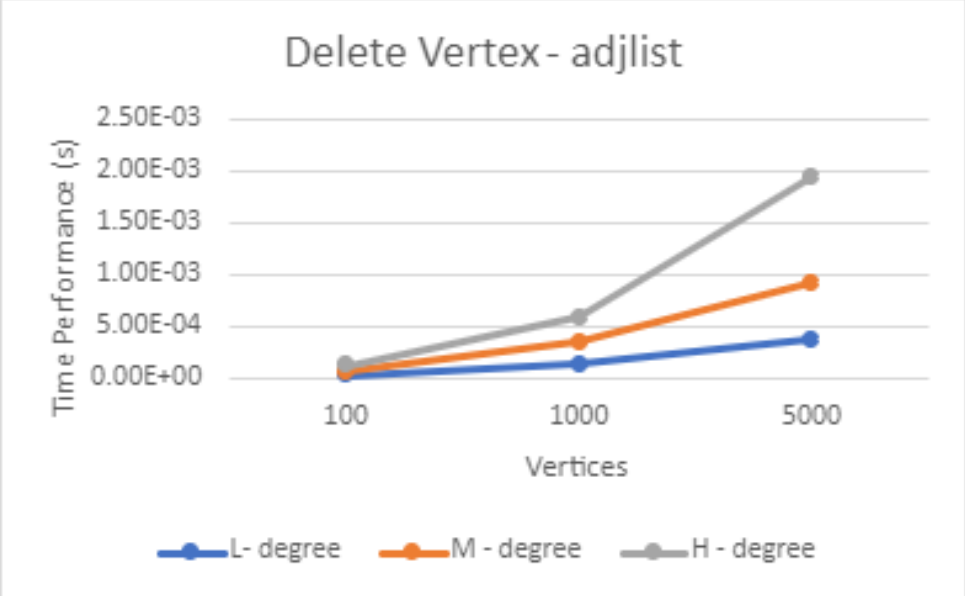

Figure 3.6 Vertex Deletion for Adjacency List (Time Performance vs Graph Size per Average Degree)

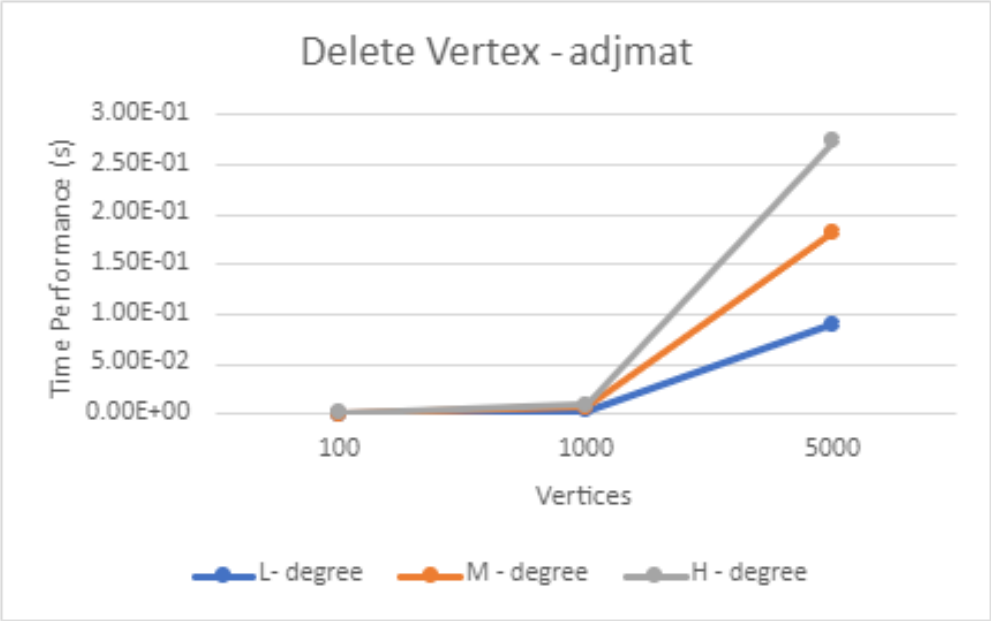

Figure 3.7 Vertex Deletion for Adjacency Matrix (Time Performance vs Graph Size per Average Degree)

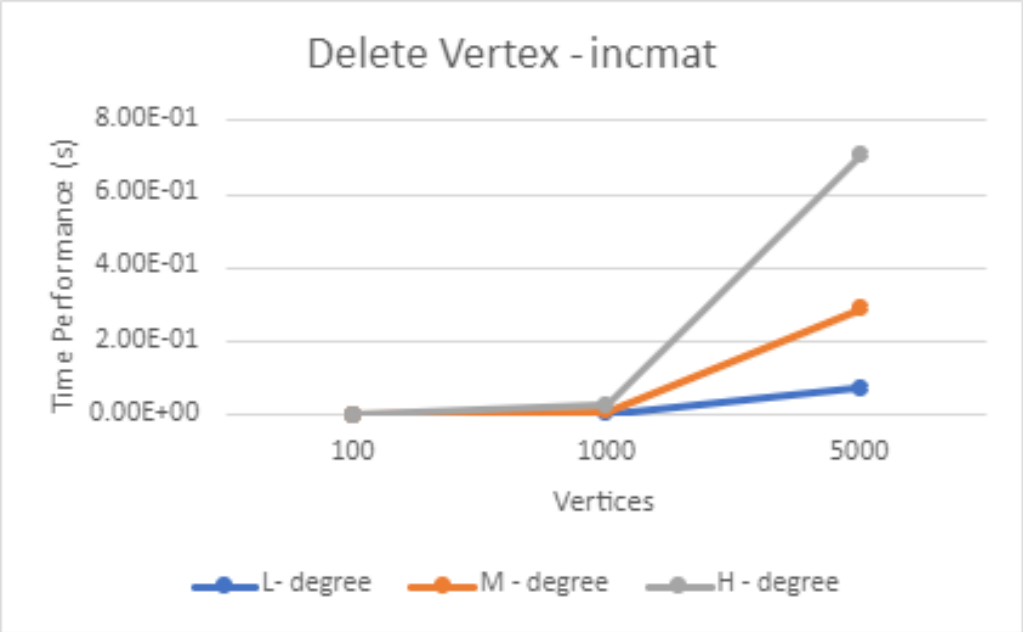

Figure 3.8 Vertex Deletion for Incidence Matrix (Time Performance vs Graph Size per Average Degree)

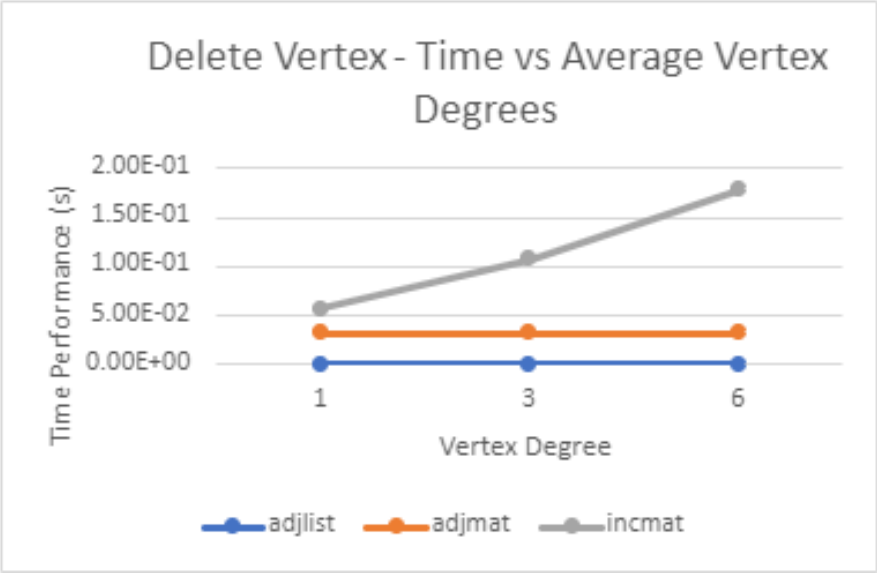

Figure 3.9 Comparison Between the Three Data Structures for Vertex Deletion (Time Performance vs Average Vertex Degree)

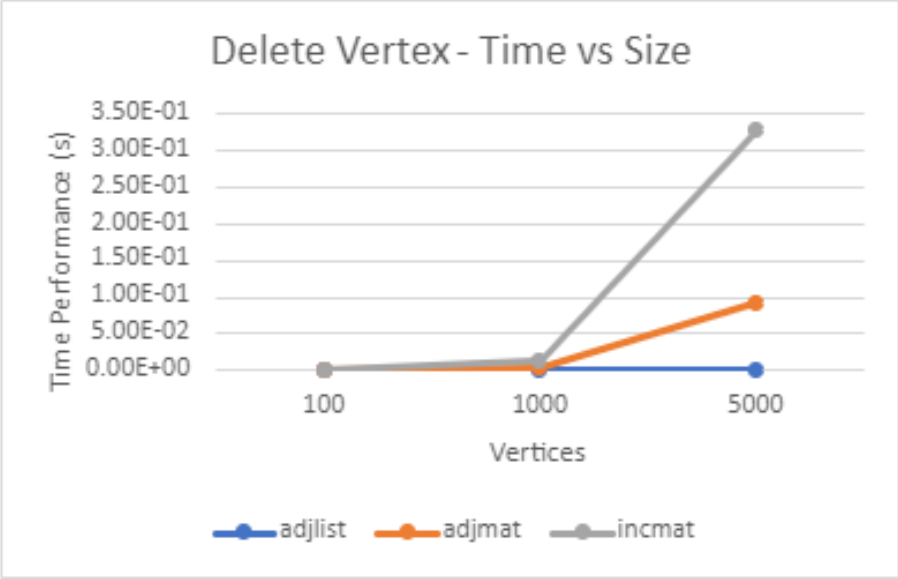

Figure 3.10 Comparison Between the Three Data Structures for Vertex Deletion (Time Performance vs Graph Size)

- **SIR Graphs**

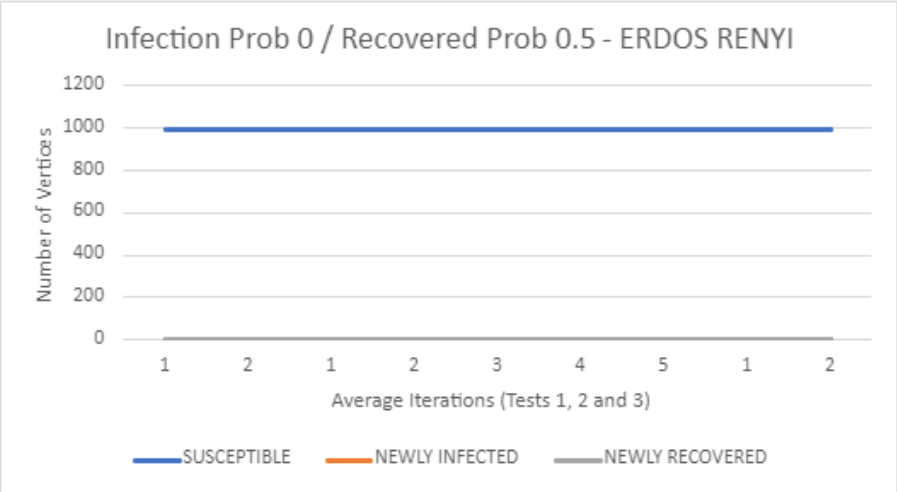

Figure 4.1 Erdos Renyi (Random) Graph with an Infection Probability of 0 and Recover Probability of 0.5

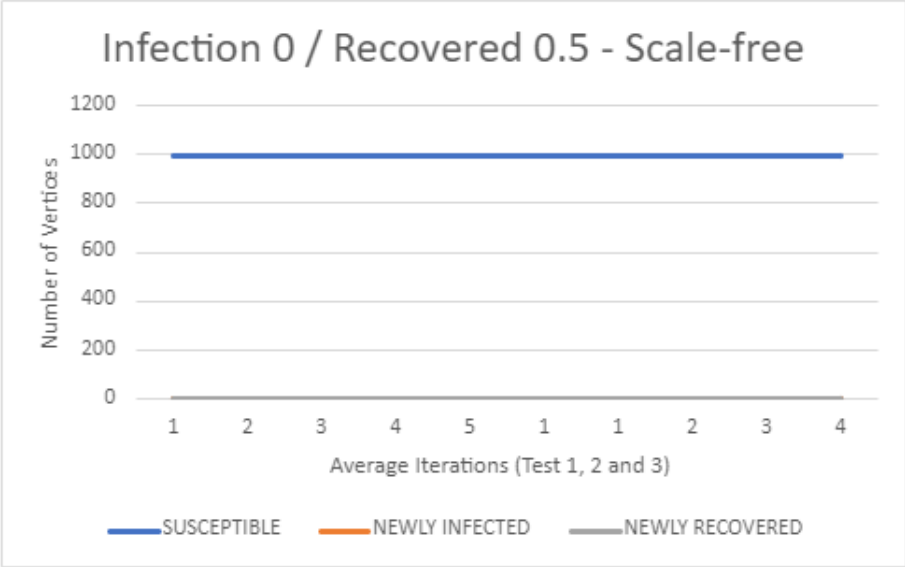

Figure 4.2 Scale-free Graph with an Infection Probability of 0 and Recover Probability of 0.5

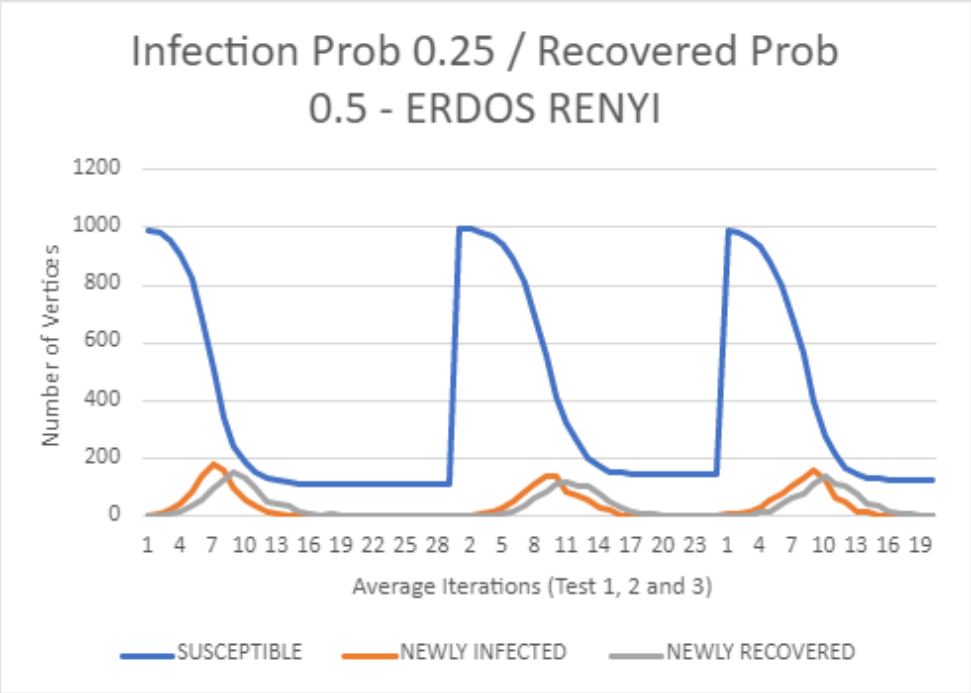

Figure 4.3 Erdos Renyi (Random) Graph with an Infection Probability of 0.25 and Recover Probability of 0.5

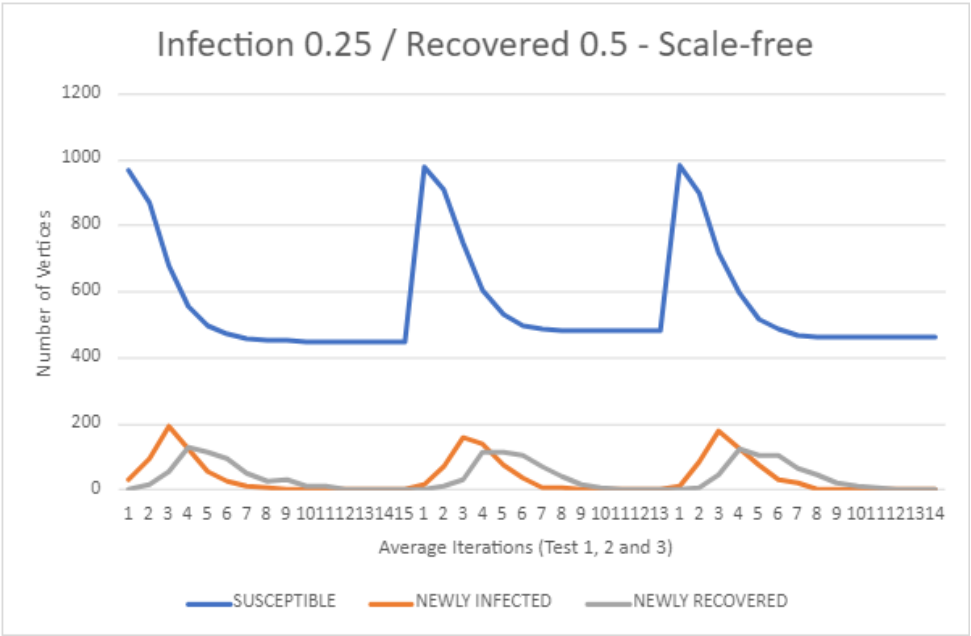

Figure 4.4 Scale-free Graph with an Infection Probability of 0.25 and Recover Probability of 0.5

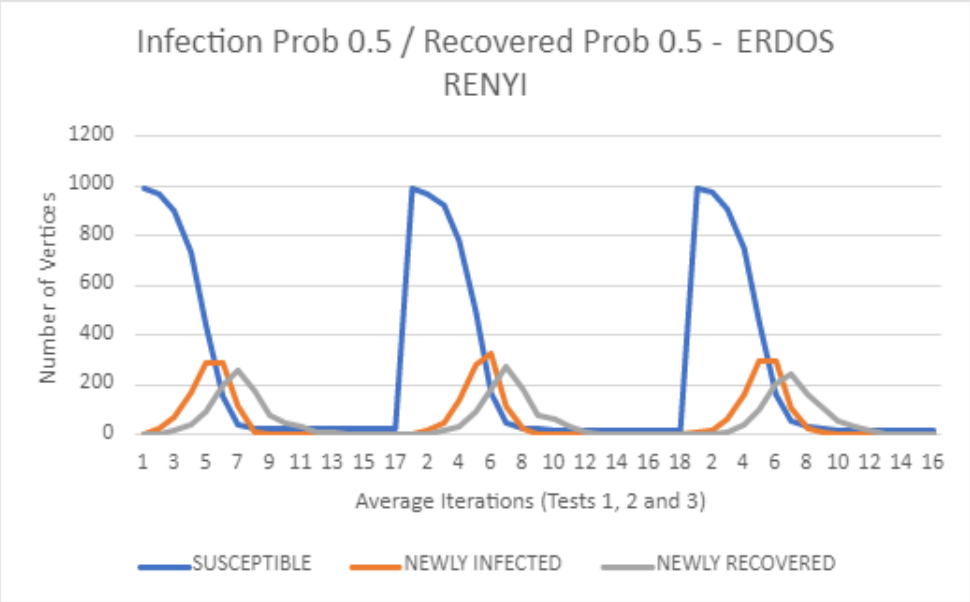

Figure 4.5 Erdos Renyi (Random) Graph with an Infection Probability of 0.5 and Recover Probability of 0.5

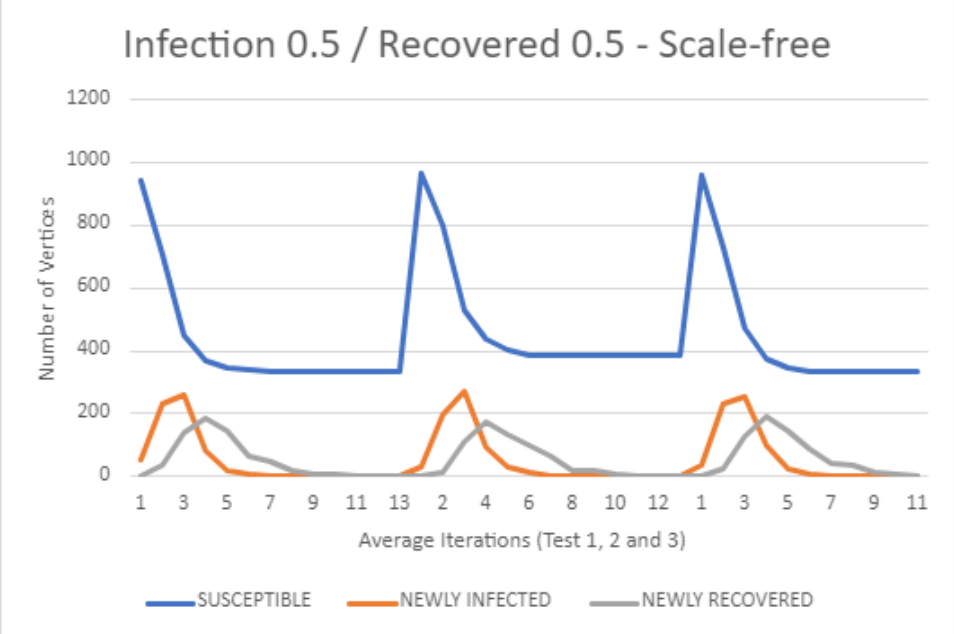

Figure 4.6 Scale-free Graph with an Infection Probability of 0.5 and Recover Probability of 0.5

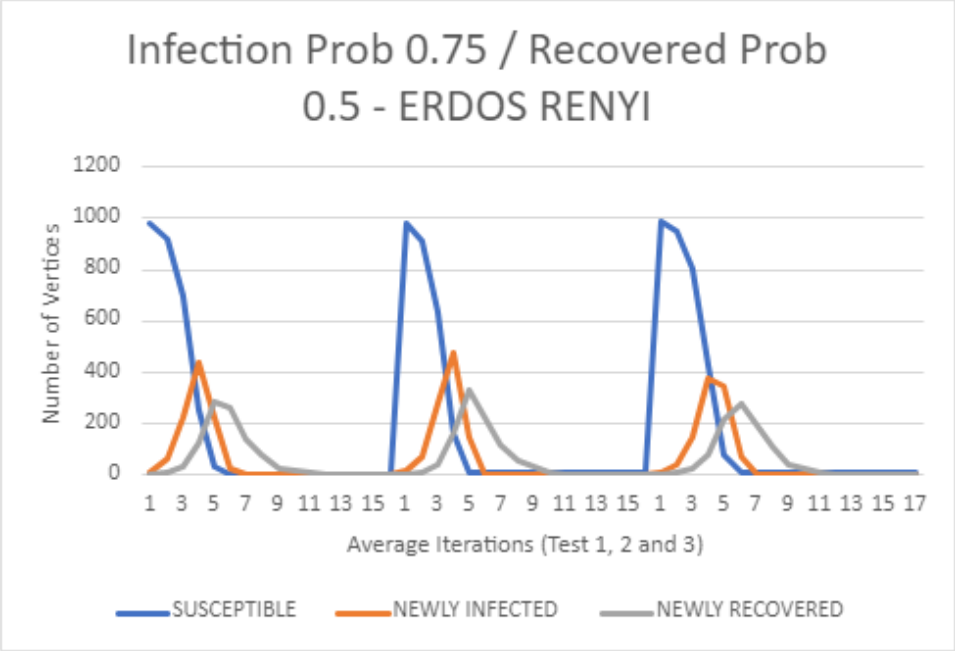

Figure 4.7 Erdos Renyi (Random) Graph with an Infection Probability of 0.75 and Recover Probability of 0.5

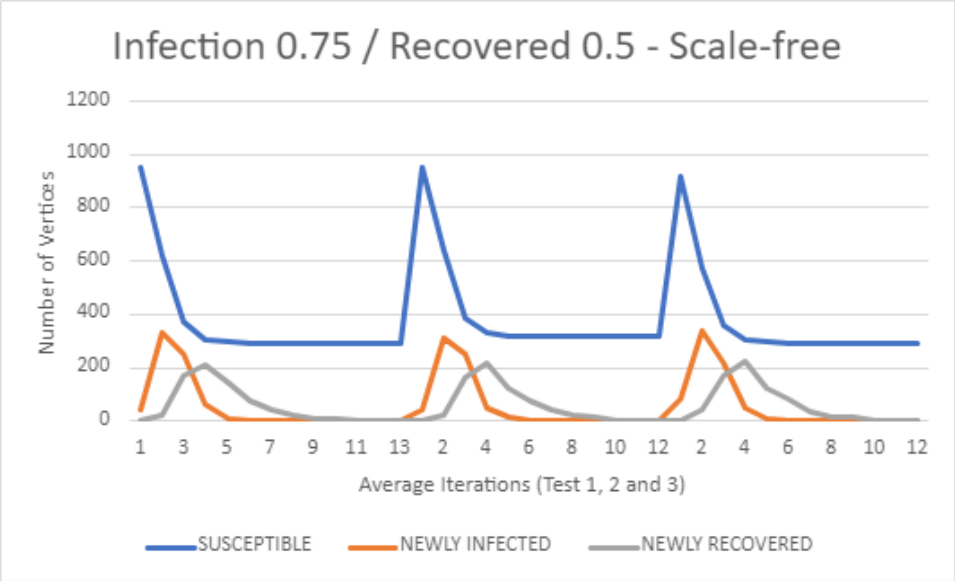

Figure 4.8 Scale-free Graph with an Infection Probability of 0.75 and Recover Probability of 0.5

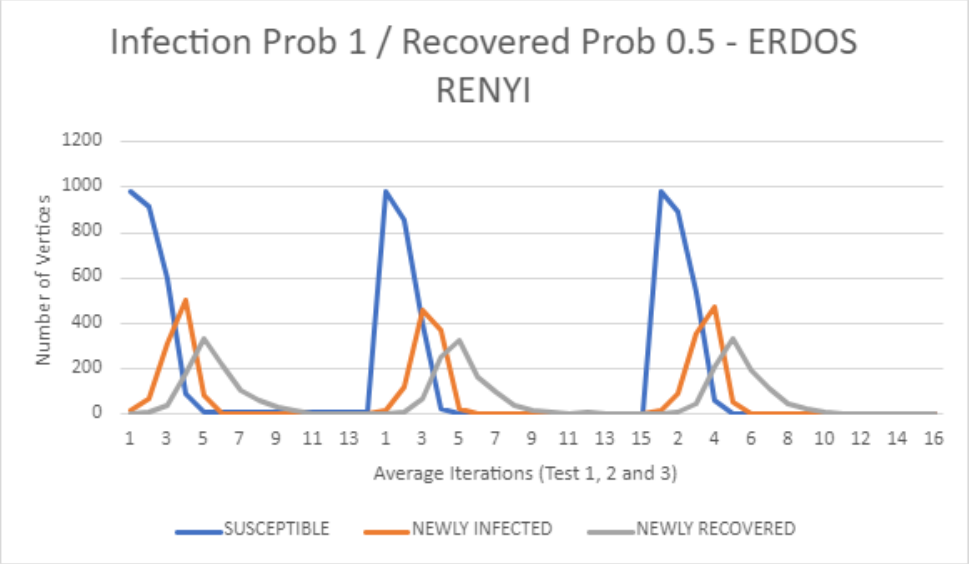

Figure 4.9 Erdos Renyi (Random) Graph with an Infection Probability of 1 and Recover Probability of 0.5

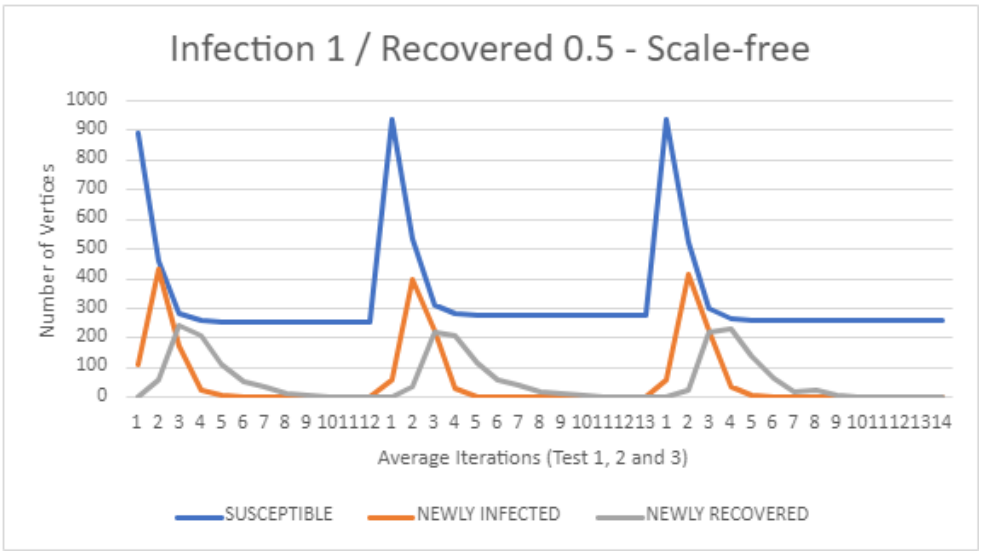

Figure 4.10 Scale-free Graph with an Infection Probability of 1 and Recover Probability of 0.5

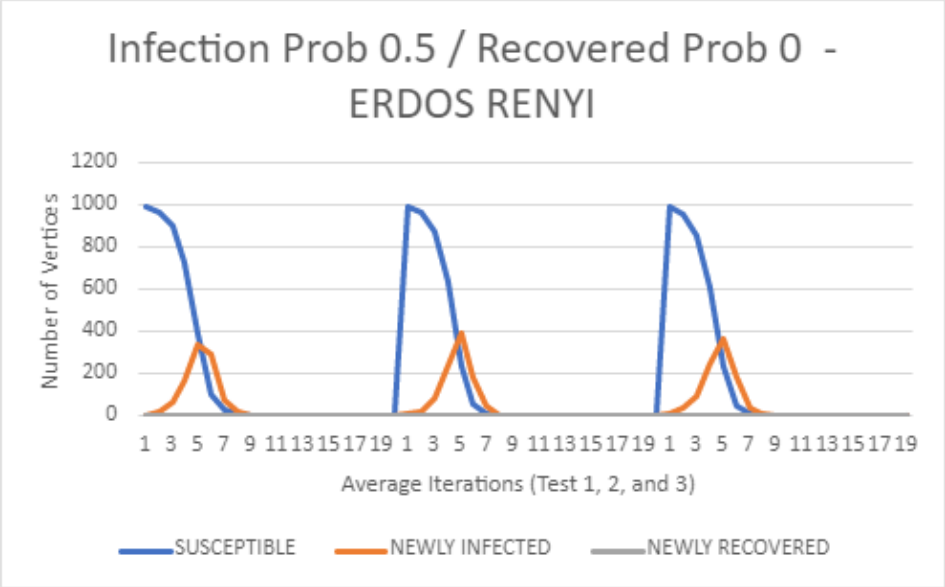

Figure 4.11 Erdos Renyi (Random) Graph with an Infection Probability of 0.5 and Recover Probability of 0

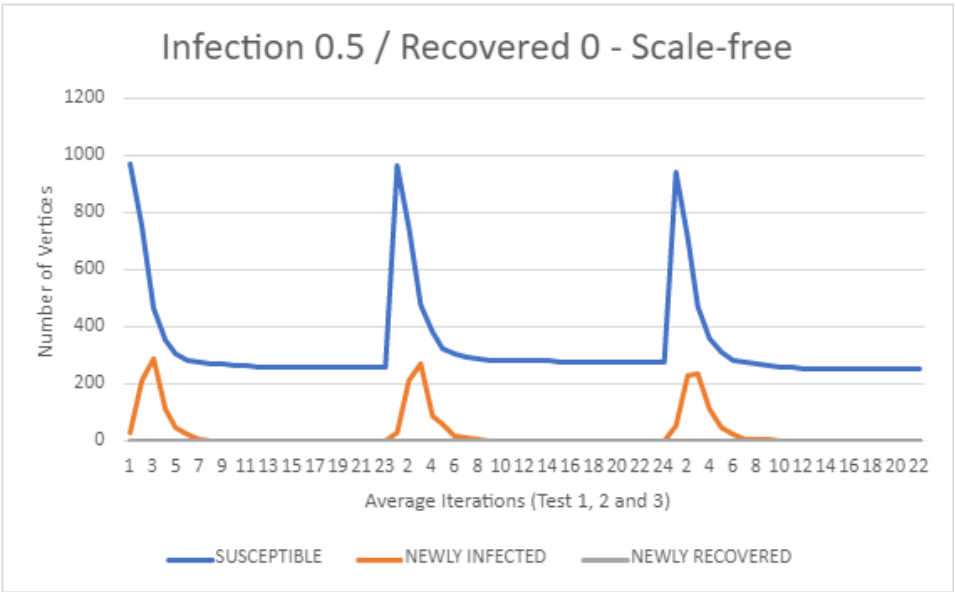

Figure 4.12 Scale-free Graph with an Infection Probability of 0.5 and Recover Probability of 0

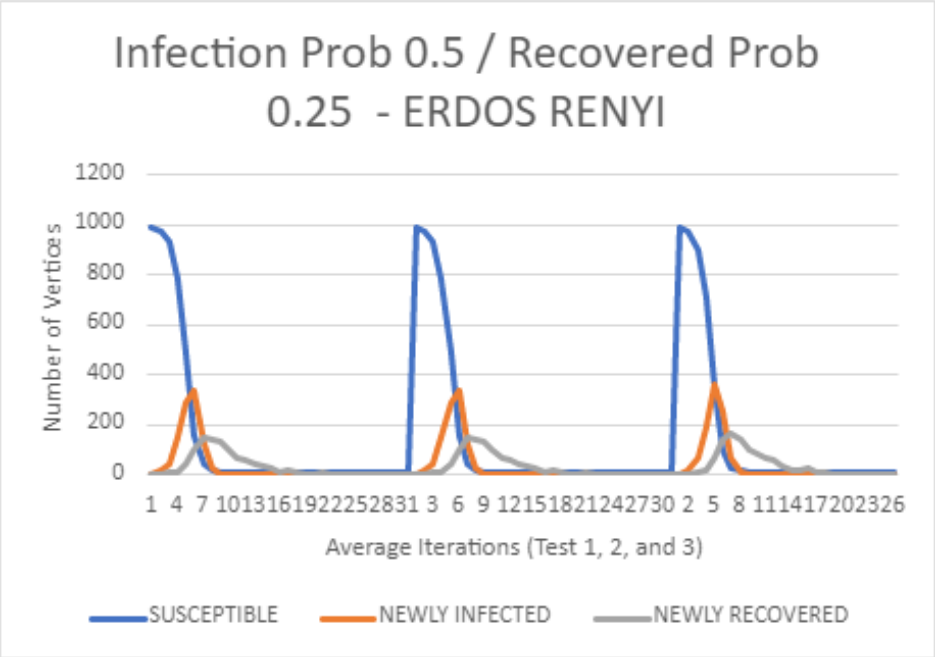

Figure 4.13 Erdos Renyi (Random) Graph with an Infection Probability of 0.5 and Recover Probability of 0.25

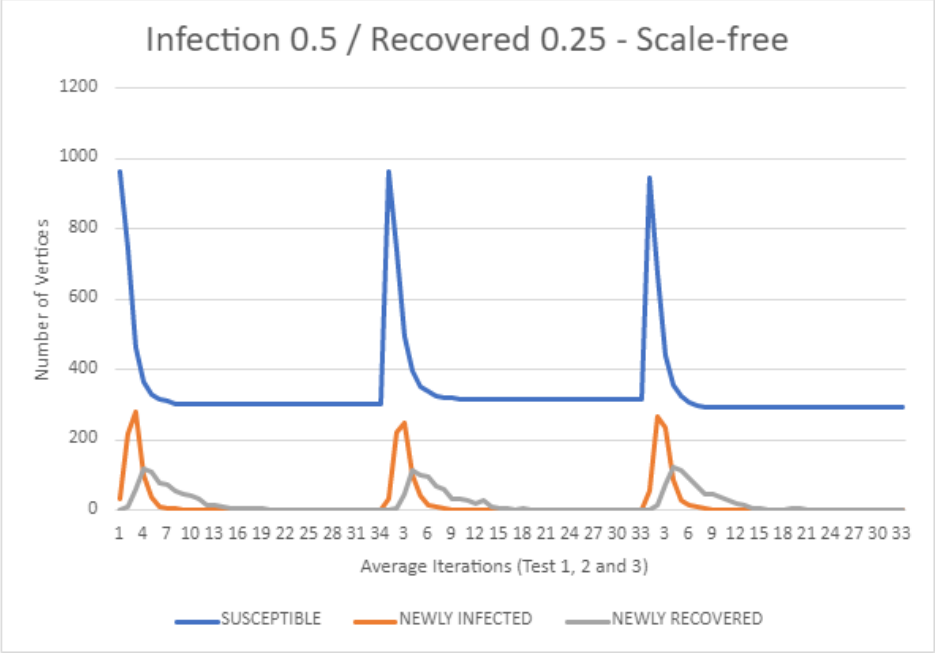

Figure 4.14 Scale-free Graph with an Infection Probability of 0.5 and Recover Probability of 0.25

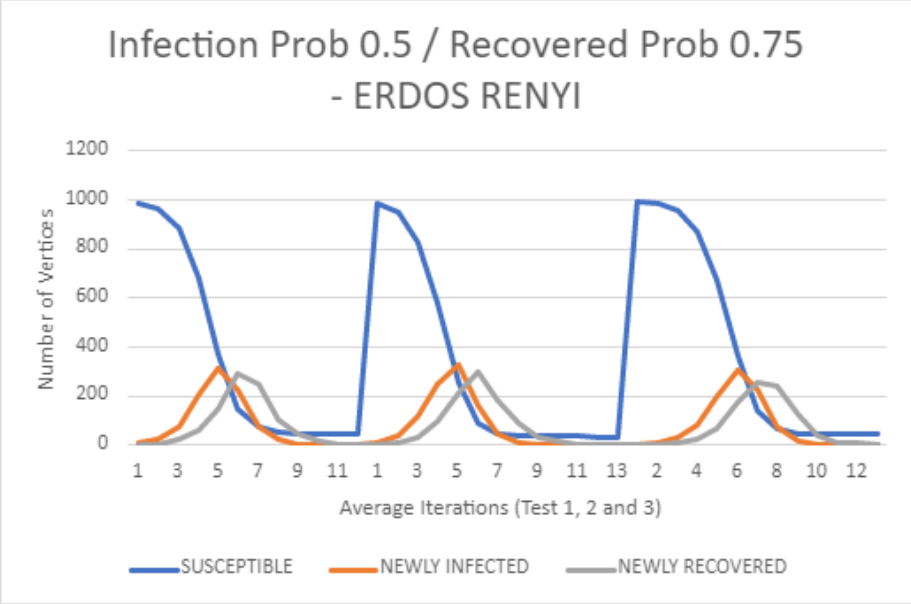

Figure 4.15 Erdos Renyi (Random) Graph with an Infection Probability of 0.5 and Recover Probability of 0.75

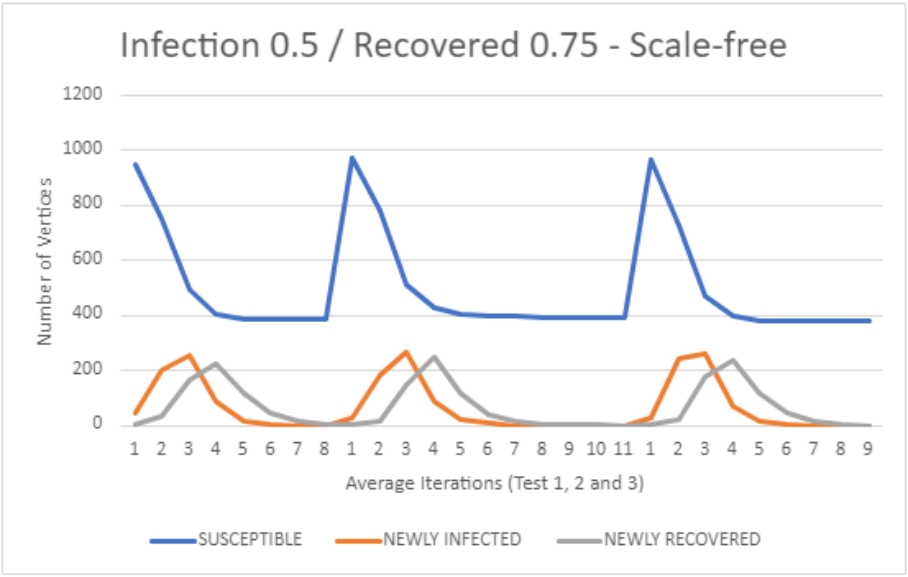

Figure 4.16 Scale-free Graph with an Infection Probability of 0.5 and Recover Probability of 0.75

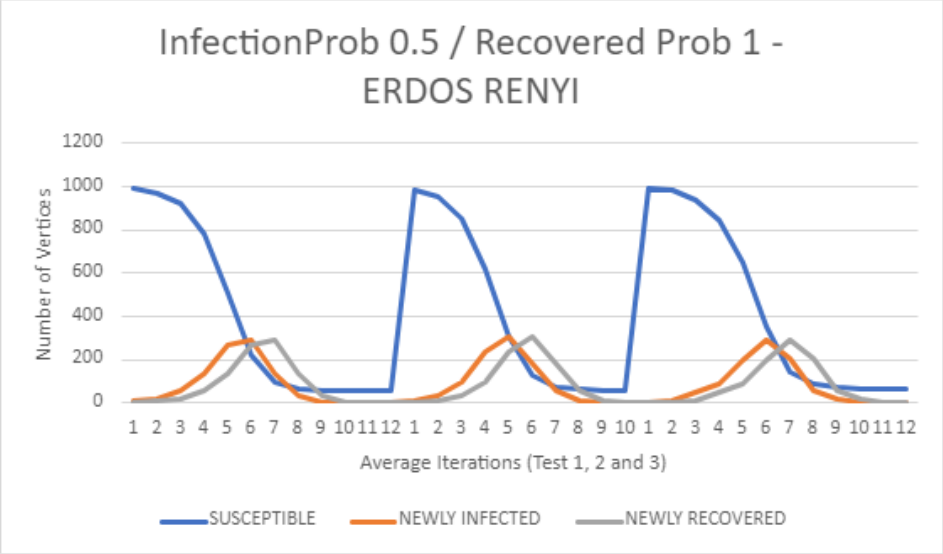

Figure 4.17 Erdos Renyi (Random) Graph with an Infection Probability of 0.5 and Recover Probability of 1

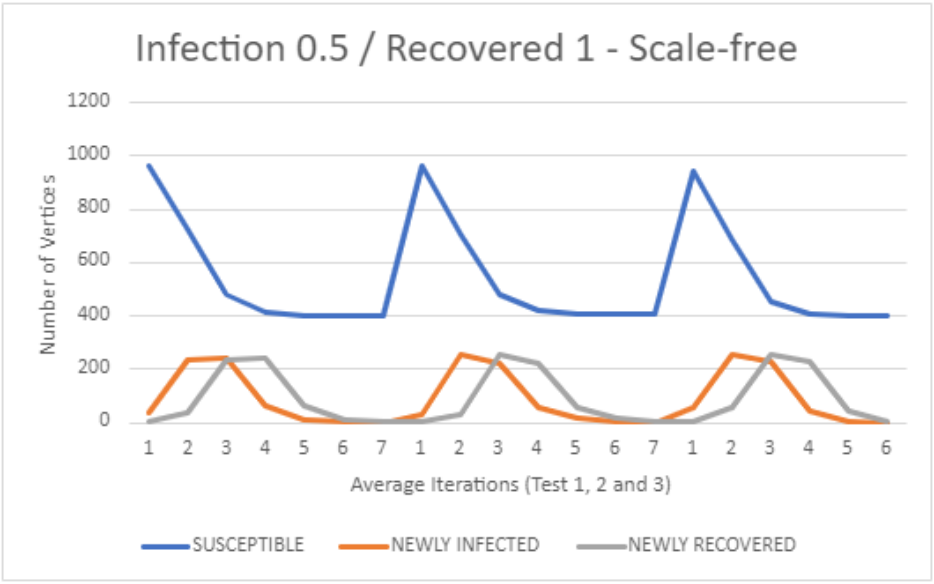

Figure 4.18 Scale-free Graph with an Infection Probability of 0.5 and Recover Probability of 1

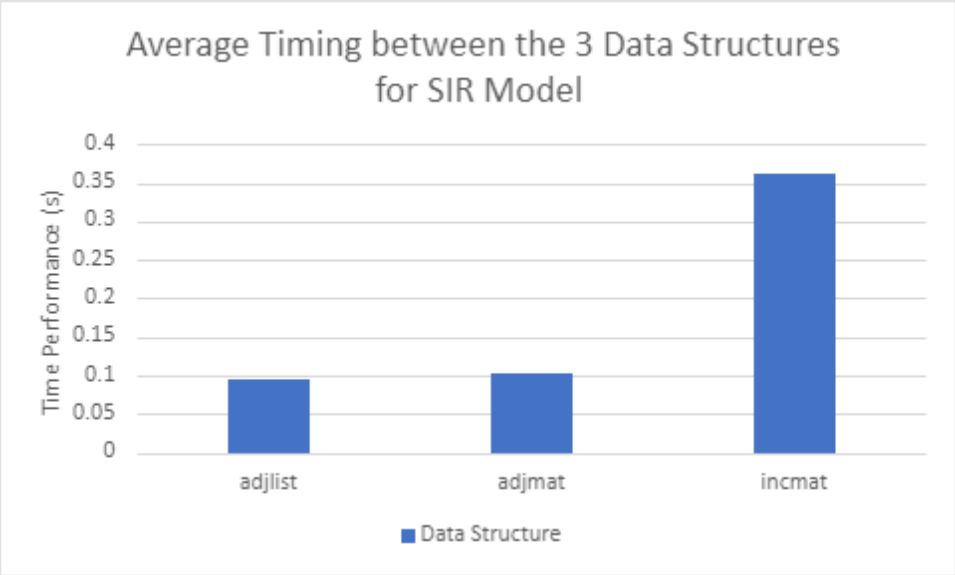

Figure 4.19 Average Timing Between Adjacency List, Adjacency Matrix, and Incidence Matrix for SIR Model Implementation