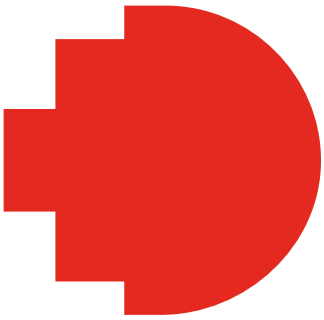

RMIT UNIVERSITY

COSC1285 – Algorithms and Analysis

Assignment 1
Team: #23

Authors:
Marton Marek (s9713615)
Andrew Buttery (s2102826)

19th April 2021

Introduction

Epidemic modelling has taken prominence in the last year due to the worldwide Covid-19 pandemic. Computing models and simulations have formed a critical aspect of the research that underpins how we respond to reduce harm. Many models exist, but a network of close contacts naturally lends itself to a graph-based model with each vertex representing a person and each edge representing a connection.

The objective of this project was to explore three types of graph implementations for modeling a network of contacts and simulating the spread of infection. The three graph implementations were: Adjacency List, Adjacency Matrix and Incidence Matrix. The model used to simulate the spread of infection was the 'Susceptible, Infected and Recovered' (SIR) epidemic model.

The graphs were implemented in Java and based on an existing skeleton code that was provided. Pajek was used to generate a collection of undirected and unweighted graphs with varying vertex counts and degrees. Erdos-Renyi and Scale Free approaches were both used and compared during the analysis. The following three scenarios were explored:

- K-hop Neighborhoods performance ('khop');
- Simulating dynamic contact conditions through the addition and deletion of edges ('dynamic contact'); and
- Simulating dynamic people tracing through the addition and deletion of vertices ('dynamic tracing').

The final aspect of the assignment was to experiment with the SIR epidemic model, exploring the impact of various infection and recovery probabilities on the 'rate of growth' of the infection. In addition, the impacts of graph size (ie. number of vertices and degrees) as well as different initial seed setups were explored for their effects on the outcomes of SIR simulations.

Data and Experiment Setup

Base code setup

The initial approach to the code and setup was to use sound OO design and reusability across the project. All the supporting 'collection' classes (refers to the concepts of the java Collection class and not actually inheriting from it) were built as generic classes to support the entire project. Three main 'collection' classes were built – LinkedList, SuperArray and SuperMatrix. The LinkedList implementation is typical and was kept as 'lean' as possible.
The SuperArray class was built as a generic class with conversions to primitive int array and String array. The underlying data structure for SuperArray is a simple array of length 10. This size doubled when the array was out of room and more items were added. Upon deletion, the array does not re-consolidate its size, instead it keeps a LinkedList of all deleted indexes and uses them for subsequent additions to the array. This eliminated the deletion time costs for SuperArray completely, at the price of an almost negligible time cost for additions (check if the linked list of deleted indexes is empty, if not return first index to use).

The SuperMatrix is a matrix data structure that utilises a SuperArray of SuperArray's. The added complexity of not reconsolidating the array indexes upon deletion in the SuperArray is also inherited into SuperMatrix. This makes the class more difficult to use and understand, with the reward of much better deletion performance.

Data Generation

The DataGeneration class was built to provide the input files and seed files needed to run the simulations. Rather then building this as a separate executable, the DataGeneration methods are encapsulated by AbstractGraph and in turn accessed by commands provided at run time on the command line. The advantage of this is we had access to the loaded graph during the data generation process. This allowed for validation of edges and vertices to ensure we did not have any false values during the simulation.

In addition, a range of Unix bash files were created to execute in loops with variable files names that could individually store different SIR commands and save outputs from those simulations as separate files (which were then consolidated into a single readable csv file by further automated bash scripts). This approach allowed a large number of autonomous executions.

Pajek was used to generate the following graphs to support the simulations in this project:

		Erdos Renyi			Scale Free		
Vertices	Degrees	Average Edges	Average Isolated vertices	Percent isolated	Average Edges	Average Isolated vertices	Percent isolated
500	2	492.67	68.67	13.73%	498.67	252.67	50.53%
500	8	1971.00	0.33	0.07%	2,006.00	97.67	19.53%
500	16	3,993.33	0.00	0.00%	4,005.33	55.67	11.13%
5000	2	5,042.67	670.67	13.41%	4,998.67	2,493.33	49.86%
5000	8	19,899.33	2.67	0.05%	19,995.00	1,008.67	20.17%
5000	16	40,100.33	0.00	0.00%	39,537.33	571.33	11.43%
10000	2	9,969.33	1349	13.49%	10,001.00	5,007.00	50.07%
10000	8	39,996.33	2.00	0.02%	40,318.67	2,019.67	20.19%
10000	16	79,987.33	0.00	0.00%	79,620.33	1,106.33	11.06%

Table 1. Generated graph details used for simulations.

It is notable that the scale free approach generates a much higher percentage of isolated vertices, with even the highest degree still resulting in over 10% of vertices having no incident edges.

It is expected that the difference in the isolated vertices will have a significant effect on the epidemic simulation. For example, if you had a single infected individual to begin with in a 500 vertex Scale Free network with a degree of two there is a ~50% chance that they will not be able to infect anyone else causing the epidemic to burn out immediately.

Graph Size Justification

- **Minimum Graph Size (500 vertices and average 2 degrees):**
  At the conclusion of a few initial tests, we found that running small graphs completed most operations in billionths of seconds. Rather than keeping 10-20 decimal points or using scientific numerical values, we increased the load size to be able to record times that were less than 10 decimal points after zero.
  2 degrees was chosen as the minimum as anything less would either leave a lot of 'rouge' vertices or the graph would almost represent a linked list instead of a social network.
- **Maximum graph size (10,000 vertices and average 16):**
  After initial testing on larger graphs, we found that any graphs larger than 10,000 vertices and 16 degrees were computationally prohibitive for some of our simulations (eg khop).
  This was chosen as it pushed the boundaries of what we could achieve with most simulations. As indicated further in the report, we were able to run most simulations many times with this graph size except for some of the larger khop tests (eg. 5 khops with this size graph was not achievable with the computation power and time restrictions).
- **Average Degrees**
  The numbers for the average degrees are based on 2^1, 2^3, and 2^4. This spread gave us large enough differences between all graph sizes and were computationally acceptable for most operations.

Graph implementations.

Adjacency List.

The adjacency list utilises a 'list' of vertex names, each with a linked list to store edge vertices. In this project, the 'list' was implemented as a HashMap – with the key as a string of the vertex name. The value stored in the HashMap was a Vertex class that held a SIRState and an index to a SuperArray (which held a LinkedList of edges). This implementation meant that adding a vertex and changing a vertex SIRState is a O(1) operation. According to our implementation of SuperArray, deletion operations for adjacency lists are also reduced (see section on SuperArray).

Adjacency Matrix.

The adjacency matrix was implemented using the well knows array of arrays approach. Our implementation of SuperArray was used to store a SuperArray of Boolean values. To indicate an edge existed between two vertices the Boolean value of the corresponding row and column was set to true. No edge was indicated by either a null or false value (we did not initialise all values to false to save on computation time). The adjacency matrix also adhered to the resizing and index deletion mechanisms setup in SuperArray (for both row and column).

Incidence Matrix

The incidence matrix is a list of vertex names (as rows) and edge pairs (as columns). The rows were represented the same way as an adjacency martrix (ie. SuperArray of SuperArrays) storing Boolean values to indicate an edge. However, the columns represented edge pair values. The Edge class was created to store both vertex names and was used as the key to a HashMap key value pair. The value in this HashMap was the index to the column in the incidence matrix. The design decision to make 'edge' the key and not the index value meant more efficient edge addition and deletion operations (due to O(1) access to an edge pair) at the cost of slower vertex deletions.

Scenarios

The three scenarios explored were performance of khop, dynamic contact and dynamic tracing.

The evaluation parameters were:

- Fifty (50) sample test runs across;
- Three (3) graphs with an average of;
- Two (2), eight (8) and sixteen (16) degrees with;
- Five hundred (500), one thousand (5000) and ten thousand (10,000) vertices in; and
- Two (2) different graph generation styles being Erdos-Renyi and Scale-Free.

This resulted in 2,700 test runs for each of the three scenarios.

Timing for each test was determined by taking the average of the 50 sample runs across the three graphs which had the same degree, size and graph generation method. The timing approach used System.nanoTime() method to provide the most accurate timing mechanism. Timing start and finishing points were placed to reflect just the execution time of the operations (khop, etc) without inclusion of any other code. The sample test run value of 50 was settled on as striking a balance between robust sampling versus time of execution based on initial explorations of performance.

Evaluation

The scenario evaluations were performed on the RMIT Unix core servers and as such, we do not have any insight on performance degradations due to other user loads that may have impacted results. Fifty (50) runs of each test were executed and results averaged.

Khop Evaluation

The Khop evaluation consists of identifying 'neighbors' of a given vertex in 'layers'; the immediate layer (ie. Khop 1) consists of all immediately connected vertices. Khop 2 includes all connected vertices for each neighbor and so on. This suggests that the complexity of traversing the graph on average is quadratic based on the average vertex degrees. However, this can be skewed by graphs with high density connections on one side and isolated vertices on the other. This became apparent in our testing and comparison between Erdos-Renyi graphs and Scale-free graphs. The below visualisations show that in one instance (5000v with 16 degrees), the average time to complete the khop's increased significantly (most likely due to a highly one-sided scale free graph). Aside from that single anomaly, the key takeaway is that for khop the average performance across all implementations is significantly worse with the scale free graph. Additional interpretations from the below are that the average time taken for khop for the Erdos-Renyi (mostly visible on the khop 2 test) increases quadratically based on average vertex degree.

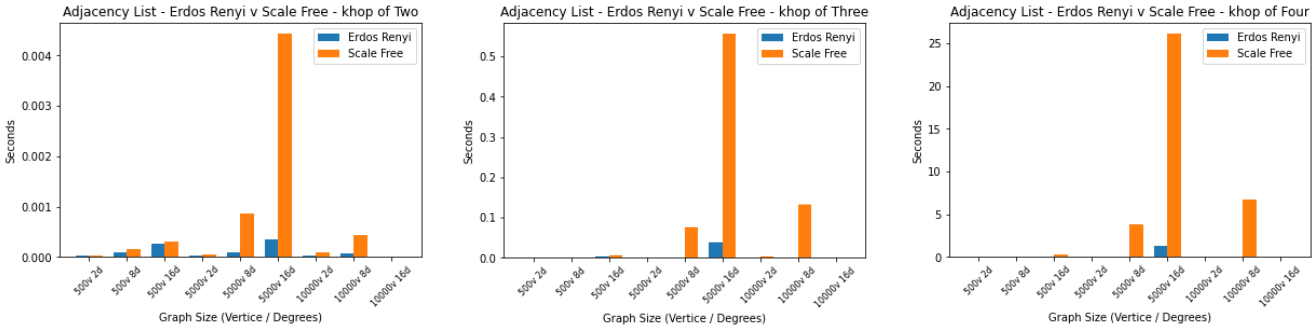

Figure 1. Example results from the Khop two, three and four tests. Comparing Erdos to Scale-free.

For comparison across the different graph implementation types, we decided to use a Khop-2 approach as it could be averaged across a high number of executions. We ran fifty tests and averaged the results.

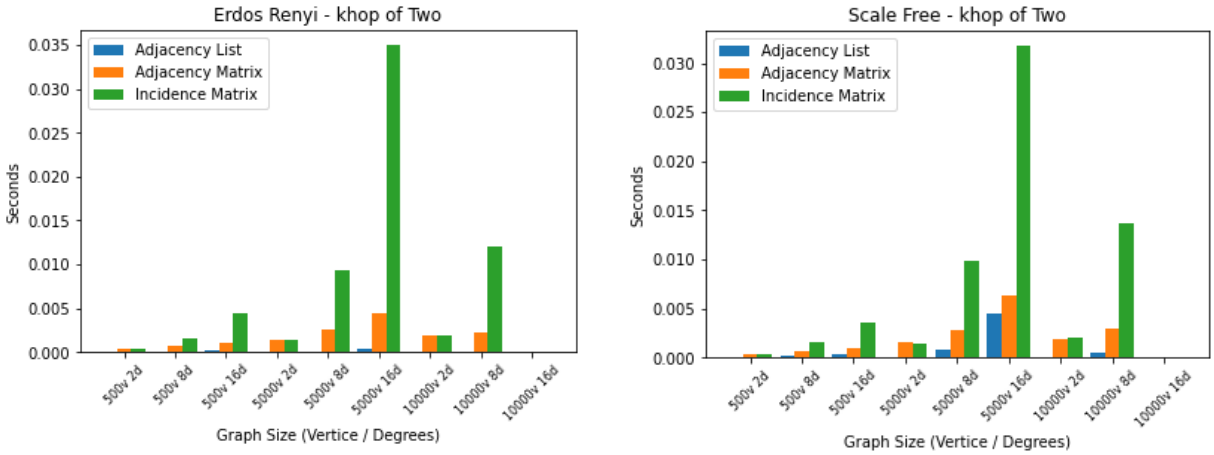

Figure 2. Graph implementation comparison of average time to run Khop-2.

Of the three implementations, adjacency list performed the best overall. The reason for this is that the adjacency list can iterate through its edges quickly and call the recursive function for each layer. In comparison, the adjacency matrix (although great at individual edge lookups) needs to iterate through an array, identify whether the edge exists then lookup the corresponding index value for the target vertex. An additional burden on our specific implementation (as noted in the code) is that we are also running a de-duplicate method on the result of the output of all neighbors. This adds either the computational time of insertion sort (for items less than 100) or quicksort (for larger lists) plus a brute force deletion of duplicates afterwards. The scope of this project did not cater for comparisons between our implementation and an alternative (which is also included in our code).

With incident matrix as the recursion was over the edge HashMap there were a series of linear time searches across the entire edge map to identify the edges of the initial vertex and then each additional vertex triggered another scan of the entire edge map which quickly drove up the number of operations required to complete the khop process.

Dynamic Contact Evaluation

Dynamic contact evaluations involve the addition and deletion of edges between vertices. To conducts these tests, we ran fifty (50) executions of additions and deletions across the three (3) implementation types for comparison.

Edge Additions

The number of vertices and how they are connected has no correlation on the edge addition complexity, so no observable time difference was made between Erdos and Scale-free graphs for edge additions. However, the implementation type has significant differences for edge addition times as summarized by the below visualisation:

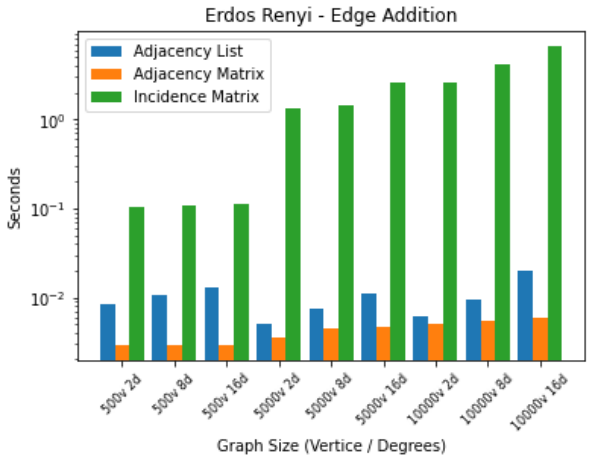

Figure 3. Comparison of graph implementations on edge deletions times.

When it comes to edge additions, the adjacency matrix outperforms the other two implementations by a significant margin. This is due to the adjacency matrix's ability to identify whether an edge exists and update that edge in O(1) constant time. In comparison, the Adjacency list needs to traverse the entire existing edge list to make sure it doesn't add the same edge twice. The adjacency list complexity for identifying an edge is O(average degree of vertices), followed by O(1) constant complexity to add the edge (add the edge to the end of the linked list).

The incidence matrix also needs to traverse through the entire matrix to ensure that the edge required for addition does not already exists. Our design decisions and implementation of the incidence matrix also heavily influenced its performance. Based on our implementation, to check for an edge the incidence matrix needs to traverse an entire row of the matrix and for each edge that exists, it needs to traverse the entire vertex list to identify the edge. This approach resulted in the poor performance of edge additions for the incidence matrix.

Edge Deletions

The graph type (Erdos or Scale-free) did have an influence on the performance of edge deletions as summarised by the below visualisation:

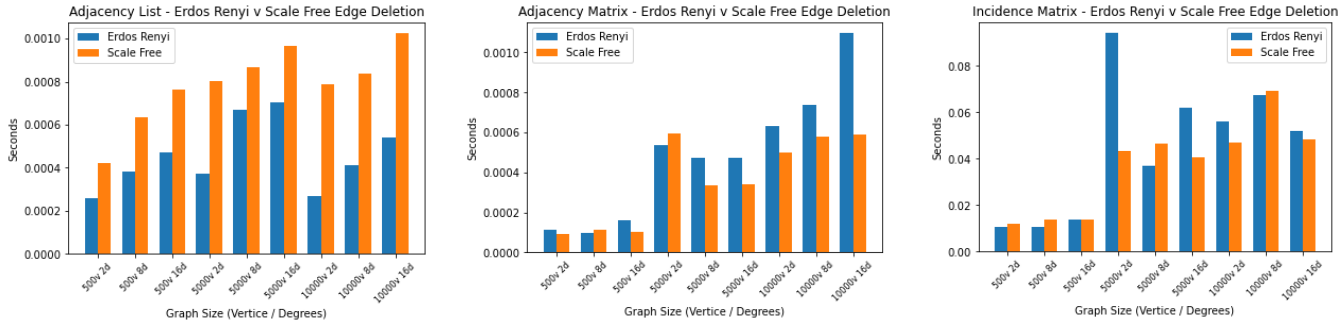

Figure 4. Comparison between graph type and implementation on edge deletions times.

Although there is seemingly some difference in the adjacency matrix timings, it is noted that they are the thousands of seconds and any differences are inconclusive. Similar to edge additions, the adjacency matrix can identify whether an edge exists and remove the edge in O(1) constant time. In comparison, the adjacency list takes O(average degrees of vertices) because it needs to potentially iterate over its entire list of neighbors.

With the incidence matrix, whilst deleting edges is potentially faster than adding one (as we don't need to traverse the entire edge list – with delete we can stop when we find the one we are deleting), it is still the slowest performing of all implementations. Apart from an anomaly with the 5000v/2 degrees graph, the data suggests there is no significant difference between running incidence matrix on scale free or an Erdos graph type.

Dynamic Tracing Evaluation

One of the clearest findings came from the evaluation of the different approaches to vertex addition and deletion.

Vertex Addition

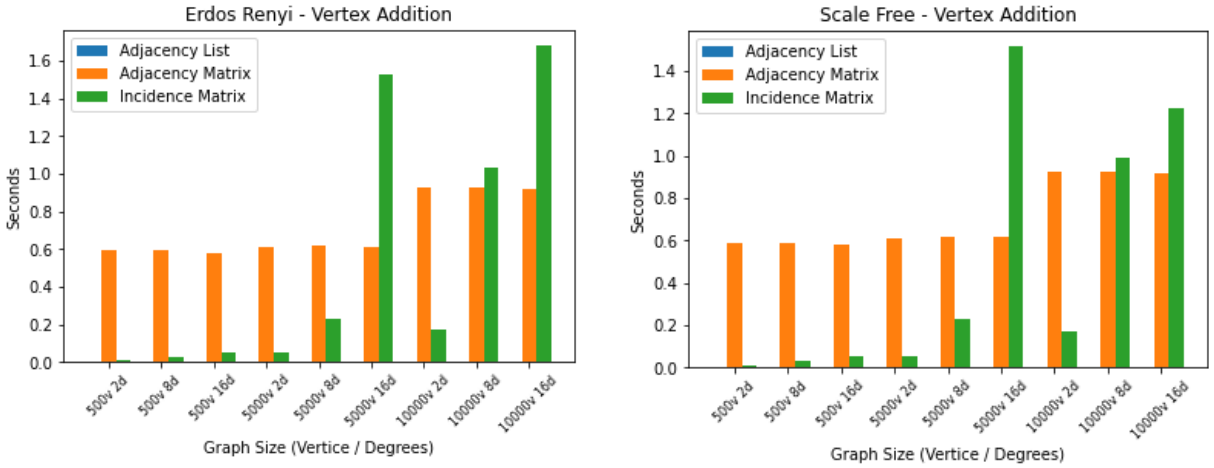

Figure 5. Comparison of graph type and graph implementation for vertex addition average times.

With the vertex add operation, the adjacency list is clearly the best approach as adding a vertex only involves creating an entry in the HashMap. In comparison, the other two implementations need to include a new row to the matrix. In the case of the incident matrix the size of the row is related to the number of edges in the graph and accordingly as the degree increases the size of the array to be allocated increases.

As expected, the generation of the nature of the graph (Erdos or Scale Free) made no difference to the allocation of new vertices.

Vertex Deletion

In the case of vertex deletion, the adjacency matrix outperforms the other two significantly. The main driver of performance when deleting a vertex is associated with the number of edges that are connected to the vertex being deleted. In all cases, the entry is deleted from the vertex HashMap. For the adjacency list the linked list is destroyed and for the adjacency matrix and incident matrix the associated row is set to null. When deleting from an adjacency matrix, deleting the vertex automatically deletes the edge information so the operation is performed in constant time O(1). The adjacency matrix needs to traverse its linked list to find associated edges then traverse those to delete all of the edge information, translating to O(average degrees) time.

The incident matrix has the worst performance as firstly the map of incident edges needs to be search for any vertex pair that matches the deleted vertex and then the corresponding column needs to be updated resulting in a steep rise in time taken as the graphs increase in size and degrees.

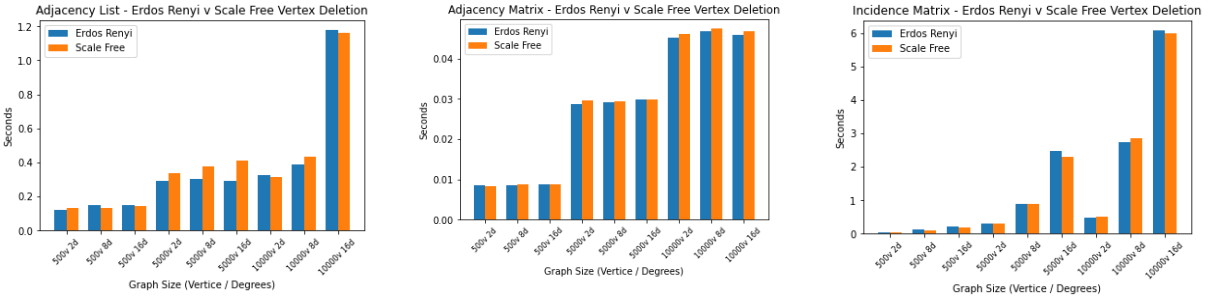

Figure 6. Comparison of graph type and graph implementation on vertex deletions.

Recommendations

Overall, the adjacency list and adjacency matrix perform well under most circumstances.

In a situation where the most common task is to undertake searches of neighborhoods the adjacency list continued to perform better that the other two representations even as the degrees and density of the connected components of the Scale Free generated network increased.

For dynamic contact either adjacency list or adjacency matrix are suitable representation with neither offering a significant performance increase over the other.

For dynamic tracing where there is a lot of change in the number of individuals being traced, especially if they are dropping out of the network, the adjacency matrix is the recommended approach based on its superior vertex deletion performance.

Our tests would suggest that the incident matrix approach never appears to provide acceptable performance except on very small graphs. This may be due to how the representation has been implemented and further exploration of alternative approaches may reveal performance improvements that would create scenarios where it may be a viable, or even preferred, graph representation.

Epidemic Modelling

The five (5) simulations that were executed, tested and analysed are described below. The setup for these simulations were done using the DataGeneration class to create the input for each execution.

For each simulation, we generated separate input data and automated scripts as described below:

To increase the total combinations we could run, a unix bash file was created to run the main program multiple times in a loop, iterating the filenames for the various inputs and using variable file names as the output. The unix bash file would also combine all the output files after each execution, into a consolidated csv file. The combination of variables included in the simulation are listed in the table below:

Graph Types and sizes	Infection and recovery probabilities	Total executions per simulation (A, B, C, D, E)

• 9 Erdos graphs
  ○ 500 (2, 8, 16 degree)
  ○ 5000 (2, 8, 16 degree)
  ○ 10000 (2, 8, 16 degree)
• 9 Scale-free graphs
  ○ 500 (2, 8, 16 degree)
  ○ 5000 (2, 8, 16 degree)
  ○ 10000 (2, 8, 16 degree)
	For each graph, the following probabilities were run:
	(25%-25%), (25%-50%), (25%-75%), (50%-25%), (50%-50%), (50%-75%), (75%-25%), (75%-50%), (75%-75%)
		• 81 Erdos
		• 81 Scale-free

Table 2. Combination of simulations executed for epidemic modelling.

Simulation A, B and C

The objective of simulations A, B and C were to measure the average time taken for SIR simulations based on different graph types (Erdos vs Scale-Free), implementations (adjacency list, adjacency matrix and incidence matrix) and the size of the input graph (number of vertices and average degrees). 162 executions were run using automated bash scripting in Unix for each simulation. The timing results were captured and averaged to provide the below summary table. Simulation A ran with one (1) random infected starting seed. Simulation B ran with three (3) clustered infected seeds (all neighbors) and Simulation C ran with three (3) random seeds (from across the graph). All were timed and averaged across several runs.

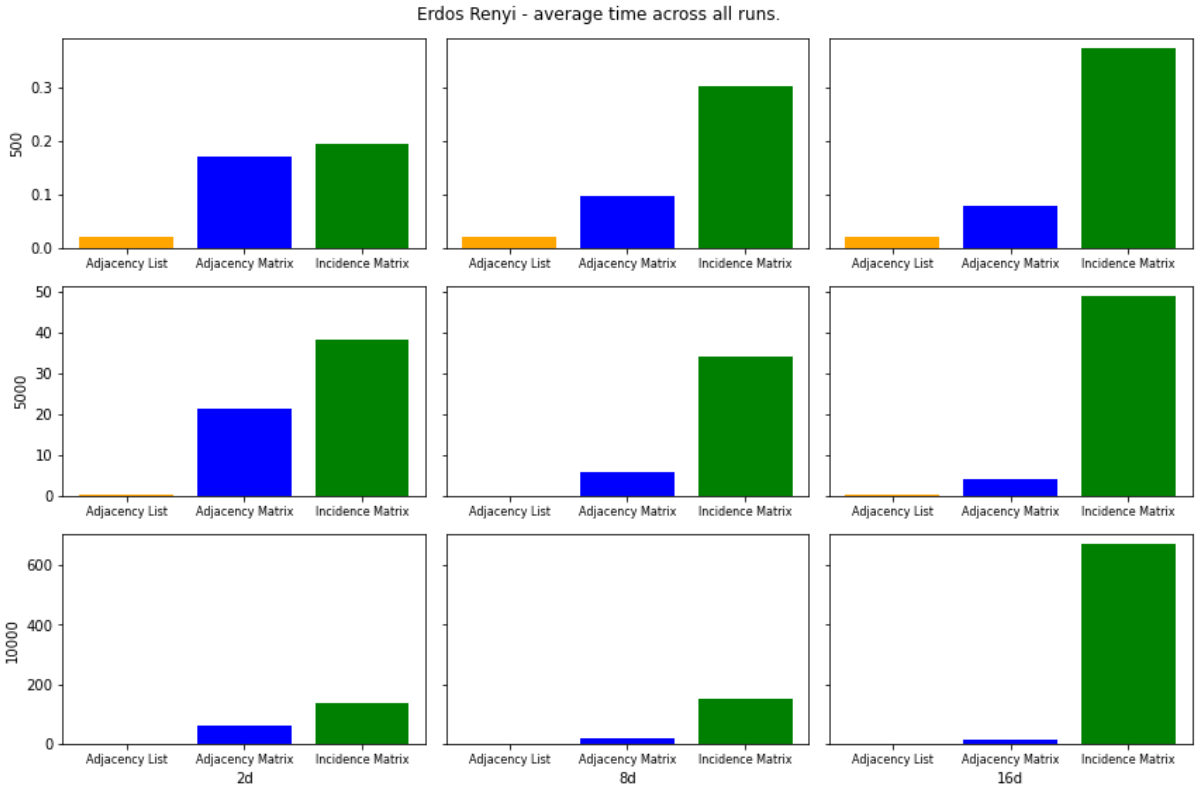

Figure 7. Comparison between the average time for SIR Simulations across graph type and size.

Please note, the scale of the y-axis for each row is based on the average time of the simulation for each graph size. As the average time increased proportionally to the graph size, we needed to change the scale for each row to fit the graphs into the diagram. The Scale-free version of results look almost identical to the visualisation above so it wasn't included to save space.

A key observable result is that the adjacency matrix average time decreases as the average degrees increases. When traversing the entire graph the edge lookup is a key function, so this decrease in average time for the adjacency matrix is related to its ability to perform edge lookups in constant time. In addition, the higher the average degree, the lower the khops needed to traverse the entire graph. However, even with this advantage the adjacency matrix was still inferior to the adjacency list for SIR modelling simulations. Finally, as clearly indicated by the above visualisation, the incidence matrix performed the worst in every test case.

Simulation D and E

Simulations D and E were designed to compare the "rate of infection spread" given the factors of infection probability, recovery probability, graph size (vertices/ degrees) and initial graph creation type (Erdos or Scale Free). The graph implementation for these simulations does not have any influence on the outcome, therefore we chose the adjacency list to run these tests (based on earlier findings that the adjacency list provides the best performance for most operations).

An additional variable we wished to explore for possible impacts on the rate of spread was the initial seed. Specifically, the difference between having a random set of nodes infected that have no common link versus a 'cluster' of infections (ie. every infection is connected as a neighbor).

Simulation D used a clustered approach (ie. all initial seed infections are connected as neighbors) to emulate the 'patient zero' scenario where an infection has begun to spread to close contacts. Simulation E takes a randomised approach and creates an initial seed list of non-connected random vertices. This is to emulate multiple random starting points in the simulation.

The setup of simulation D and E required additional code as it became a requirement to output information about each iteration. The output included the iteration number, graph size, total infections, new infections and newly

recovered totals. This information would provide the base to create visualizations that show the 'rate of spread' based on multiple scenarios. As with the earlier code, an additional command was created in the main program to cater for the creation of seeds as an input file to an altered runIterationSimulation() method.

The following points were observed through the analysis of the given data sets from simulation D:

- High infection rate and low recovery rate (eg. 75% - 25%): would result in a peak in total infections much earlier in the iteration cycle. The recovery would also be slightly slower with a 'tail off'. The early spike and tail off is demonstrated in figure 8. below.

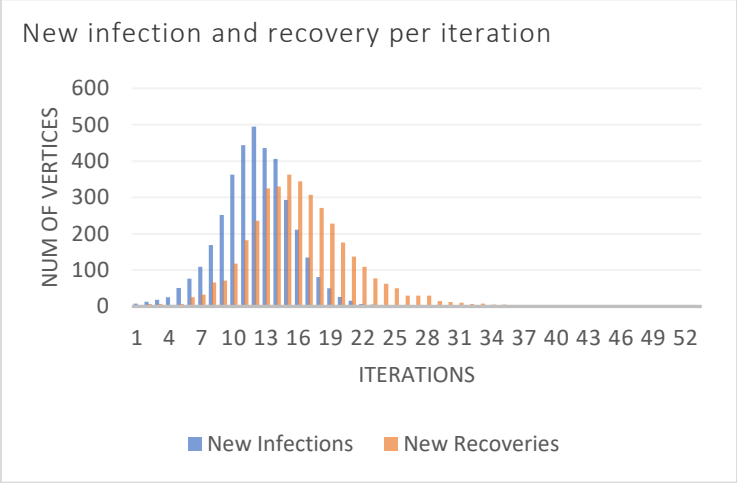

Figure 8. Erdos graph with 5000 vertices and average of 2 degrees. Infection probability 75% and recovery probability 25%.

- Moving the infection and recovery percentages closer together (ie. 50%-50%), results in a significant decline in the initial acceleration of the infection across the network as shown in figure 9. below. The total infection during the entire cycle was also significantly lower.

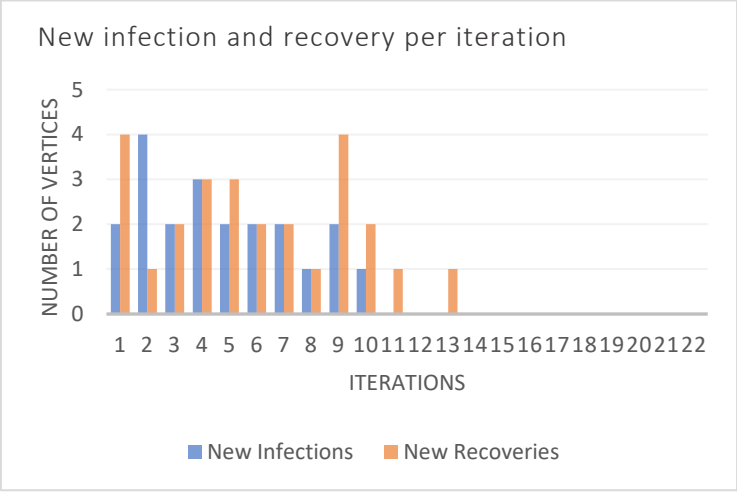

Figure 9. Erdos graph with 5000 vertices and average of 2 degrees. Infection probability 50% and recovery probability 50%.

- Comparing to a graph with higher average degrees, we can see that the higher average degrees have a similar effect to increasing the infection probability. Figure 10. shows a much earlier 'peak' infection with the same probability distribution of figure 9. but with a higher average degrees. In fact, it reaches 100% vertex infection by iteration 6 (good only if you are aiming for herd immunity).

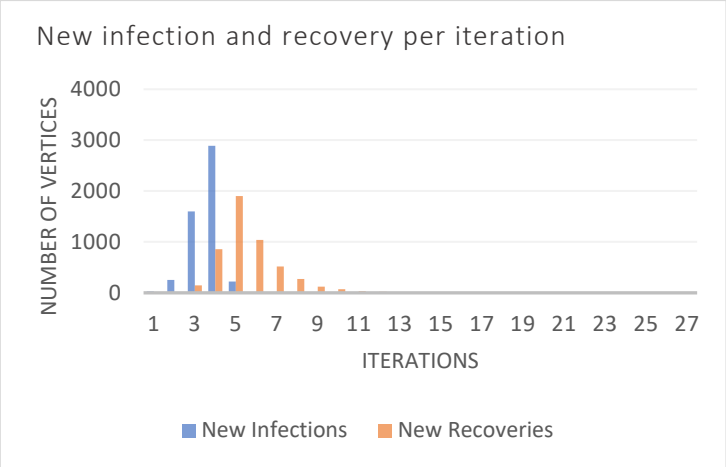

Figure 10. Erdos graph with 5000 vertices and average of 16 degrees. Infection probability 50% and recovery probability 50%.

Erdos vs Scale-Free

Erdos graphs have a more even distribution of edges and more closely reflect the average degrees per vertex. Scale free graphs will have vertices with zero to low number of edges and vertices with a very high number of edges. Due to this variance and unpredictability, scale free graphs can be difficult to analyse for performance. Running many tests and averaging their results will provide increasingly accurate results (if you have the computing power). Although our analysis followed this averaging principle, further iterations are required to provide accurate results. Based on our limited resources for testing, our data showed that scale free graphs have a similar affect to the **increase in vertex degrees**. Running scale free graphs with the same infection and recovery probabilities as the graphs above, all resulted in left skewed bar charts (high infection rate increase early in the iteration cycle).

Clustered vs Random

The results from simulation E were very similar across all variables to simulation D using the Erdos approach. This suggests that the 'clustering' of infections is not as critical to determine rate of spread in evenly distributed networks. Using the scale-free approach showed that clustering made a difference to total infections when the cluster was in the middle of highly connected vertices. When the cluster was isolated, the rate of infections dropped dramatically. This is in line with our hypothesis based on other behaviors observed from the scale-free type graphs.

Conclusion

The major conclusions and key takeaways from our analysis are listed below:

Key takeaways:

- For most operations, the Adjacency List is the best performing graph implementation. Incidence Matrix performs the worst.
- For edge lookups and deletions, the Adjacency Matrix has the best performance.
- For a balanced network simulation and traversal, the Erdos-Renyi approach is more suitable.
- Arguably, in real life people with many friends attract more friends and therefore the scale-free approach (rich-gets-richer) may be more closely aligned with simulating a real network of people.
- If you want to work with the scale free approach you will generally require more computing power (for neighborhood type processes)
- The largest influencer of infection rate of growth (besides a high infection probability and low recovery probability) is the average degrees of vertices of a graph. As the average degrees increases the infection rate rapidly increases even with lower infection probabilities.
- The Adjacency Matrix has a unique property allowing it to decrease the average time to traverse an entire graph as the average degrees increases. This is due to cheap edge operations and the overall decrease in the number of khops needed to traverse the entire graph (with high connection density, more initial neighbors mean less khops).

Bibliography

No external information was used apart from the provided code and conceptual ideas given throughout the course.