CSC2001F Assignment 5
ERSAND012

Andrew Erasmus

Design and Implementation of OOP and data structures:

The classes that I implemented for this experiment were the DataGenerator.java, Edge.java, Graph.java, GraphException.java, GraphExperiment.java, Path.java, Vertex.java classes.

The Vertex.java and Edge.java classes were implemented to represent Vertices and Edges in a graph data structure. An object of Vertex.java represents a node in the data structure with the name, adjacent vertices, cost to get to that node and the previous node in the shortest path algorithm as attributes. An object from Edge.java represents the path between 2 nodes or Vertex objects and contains the destination vertex in the edge and the cost of the edge as attributes as well. By using instances of these classes in the Graph.java class, a graph data structure could be generated and used for the experiment. This experiment is performed, and results saved to a CSV file in this class by running Dijkstra's shortest path algorithm and counting the number of operations during the course of the algorithm on a graph.

The DataGenerator.java class was used to generate 25 data sets for the experiment in the form "NodeXXX NodeYYY Z". This represented the edges from any NodeXXX to another NodeYYY with a weight of Z. This class took in user input for the filename, the number of vertices and the number of edges required by the user. This was used to generate nodes with numbers between 1 and the number of vertices entered (inclusive) to generate random nodes. These random nodes were combines with a random weight between 1 and 10 (inclusive) to generate the number of edges required by the user (which is represented in the number of lines in the file). Overall, this process was used to create 25 random datasets following user requirements with no duplicate edges or edges from and to the same node. This class was used to create the 25 dataset text files used by the GraphExperiment.java class. The GraphExperiment.java class was implemented so that Dijkstra's algorithm could be run on all 25 graphs from the dataset text files through the Graph.java class. This class does this by iterating and passing in different datasets to Graph.java, where Graph.java runs Dijkstra's algorithm to calculate the shortest path between all nodes. This algorithm does this by making use of a priority queue and the Path.java class to represent items in the priority queue. The Graph.java class then counts the number of operations while running Dijkstra's algorithm and writes the results after each dataset is processed to a results.csv file to compare the actual performance versus the theoretical performance of Dijkstra's algorithm. This therefore will perform the experiment to compare the performance of Dijkstra's shortest paths algorithm with the theoretical performance bounds.

Experiment Description:

Goal:

The goal of this experiment is to compare the performance of Dijkstra's shortest paths algorithm with the theoretical performance bounds by making use of a written program. This will be done by creating graphs in the Graph.java class from 25 different datasets with different amounts of Vertices and Edges, by running Dijkstra's algorithm on the graphs generated and recording the number of vertex, edge, and priority queue operations.

Execution:

Datasets are generated prior to the execution of the experiment. This is done by randomly generating vertices and edges for various vertex and edge combinations (with random weights between 1 and 10 for each edge). The number of vertices used for each group of 5 datasets include 10, 20, 30, 40 and 50 vertices. The number of edges is then chosen randomly and incremented for each dataset (but in the same proportion for each group of 5 datasets). For example, for 10 vertices, the number of edges used for the graphs include 20, 35, 50, 65 and 80 edges, and for 20 vertices, the number of edges include 80, 140, 200, 260 and 320 (which remains in the same proportion as the 10 vertex graph's edges). The number of edges chosen and restricted within the theoretical bound of $|E|<|V|^2$.

These vertices and edges are randomly generated with no duplicate edges and no edges from and to the same Vertex. Once generated, this data is then saved to a text file in the data directory of the project and labelled as a dataset. After this, the experiment could be run across all the datasets using the GraphExperiment.java class. This class iterates 25 times by calling the Graph.java main method with each different dataset as a parameter during each iteration. A graph data structure is then generated based on the data read in from the dataset and Dijkstra's algorithm is then run on the Graph generated. From this, while the shortest path to each Node is calculated, the number of vertex, edge and priority queue operations are counted with instrumentation. When the algorithm is completed for a graph/dataset, the results are then saved (appended) to a CSV file named "results.csv" in the data directory for future analysis.

Design decisions:

Decisions made for this assignment include saving 25 different datasets with the random vertex and edge combinations (as seen in the table of results below whereby each line was obtained by processing a dataset file). The format for these datasets is "NodeXXX NodeYYY Z" as described above so that the graph generating process in Graph.java could create the graph. I also chose to save the results of the experiment in a CSV file to allow for efficient analysis as described the "Creativity" below. Furthermore, I used the GraphExperiment.java class to firstly, append a title to the "results.csv" file in the form seen in the table below, and secondly, to iterate and run the Graph.java class (and hence Dijkstra's algorithm and instrumentation) for all 25 datasets from the GraphExperiment.java class, whereby a new dataset is passed in during each iteration.

To execute the experiment, I chose to generate datasets in a separate DataGenerator.java class, save these datasets in the data directory. I decided then to iterate over each dataset file as described above and chose to then read in these files in the Graph.java class. From this, I chose to use instrumentation to count operations in the "Dijkstra" method in Graph class and then save these results by appending them to the "results.csv" file in the main method of the same class after Dijkstra's algorithm was run on each iteration.

Results:

The following results represent the results of Dijkstra's shortest path algorithm being run on graphs generated from 25 datasets with different Vertex and Edge values compared to theoretical performance bounds.

*Note: Vertex values include 10, 20, 30, 40 and 50 while edge values include 20, 35, 50, 65 and 80 for 10 vertices and increase proportionally to their corresponding number of vertices (while remaining in the theoretical bound of |E|<|V|^2).

Table1: Table to show the performance of Dijkstra's shortest paths algorithm on 25 datasets, recorded in the results.csv file.

NumV	NumE	vCount	eCount	pqCount	Total Ops	**|E|log|V|**
10	20	10	20	14	44	66
10	35	10	35	24	69	116
10	50	10	50	34	94	166
10	65	10	65	18	93	216
10	80	10	80	39	129	266
20	80	20	80	74	174	346
20	140	20	140	126	286	605
20	200	20	200	124	344	864
20	260	20	260	169	449	1124
20	320	20	320	181	521	1383
30	180	30	180	243	453	883
30	315	30	315	295	640	1546
30	450	30	450	319	799	2208
30	585	30	585	335	950	2871
30	720	30	720	371	1121	3533
40	320	40	320	339	699	1703
40	560	40	560	437	1037	2980
40	800	40	800	505	1345	4258
40	1080	40	1080	515	1635	5748
40	1280	40	1280	517	1837	6812
50	500	50	500	649	1199	2822
50	875	50	875	649	1574	4938
50	1250	50	1250	765	2065	7055
50	1625	50	1625	735	2410	9171
50	2000	50	2000	703	2753	11288

Figure 1.

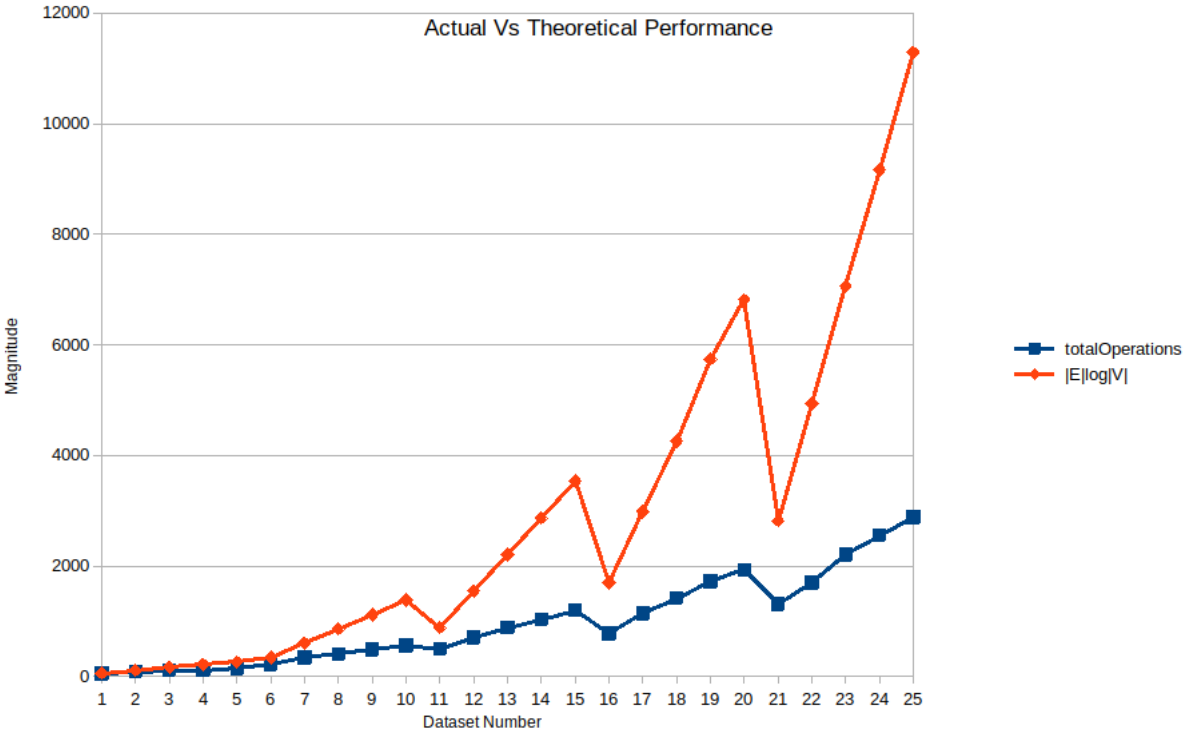


Figure 2.

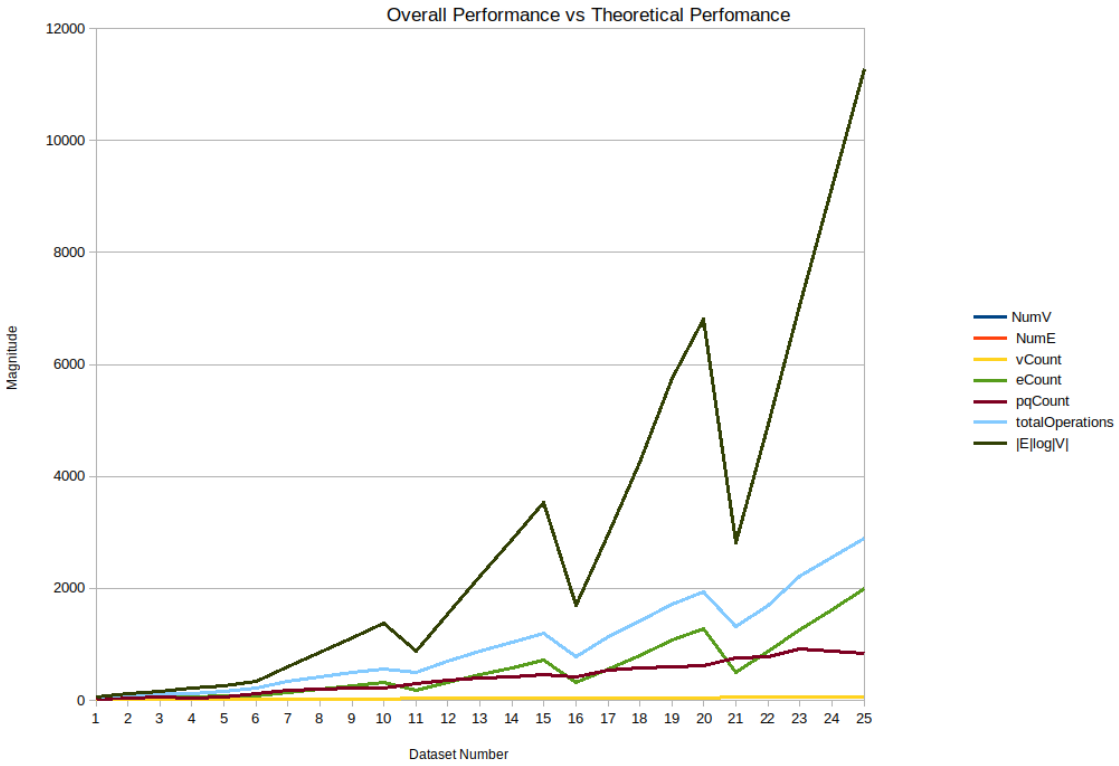
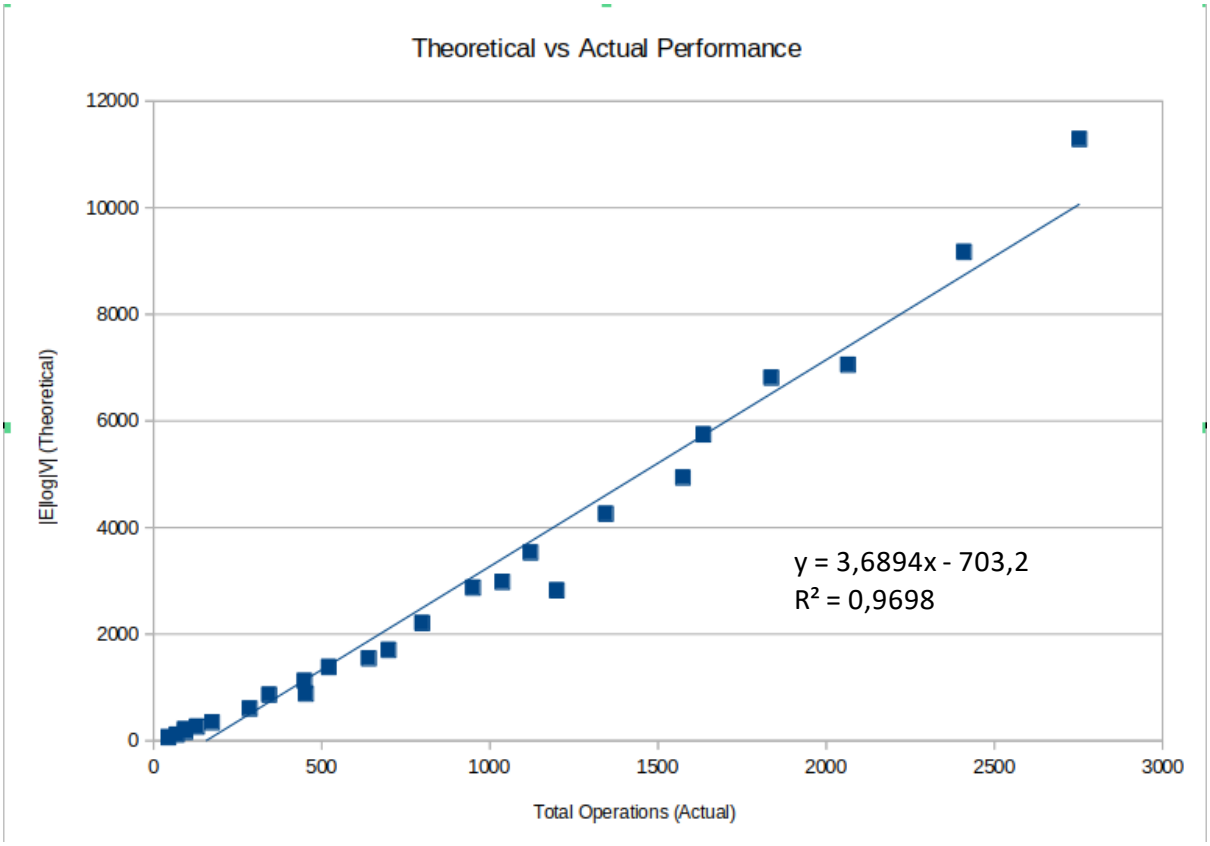


Figure 3.



Discussion of Results:

The above table and graphs represent the results of the experiment comparing actual vs theoretical performance of Dijkstra's shortest path algorithm based on standard priority queue implementation.

The table shows the number of operations recorded through instrumentation corresponding to different combinations of vertices and edge quantities. This table shows the number of total comparisons (or the performance of the algorithm) next to the theoretical performance bound based on the |E|log|V| formula (the performance bound for Dijkstra's algorithm based on the course content) with the same number of vertices and edges. The purpose of this table is to represent the data in an understandable manner and is used for the following graph representations.

Total Operations vs Theoretical Performance (|E|log|V|):

Figure 1. represents a line graph of Actual recorded performance vs the Theoretical performance bound of Dijkstra's shortest paths algorithm across all 25 datasets. This graph shows that the actual performance measured by the number of operations during processing shows that for any combination of vertices and edges. As seen in the graph representation, the actual performance remains under the O(|E|log|V|) performance bound which represents the worst case for the algorithm. This signifies that the algorithm runs with a faster performance on average (through less operations) than the |E|log|V| bound for all the datasets tested and highlights that Dijkstra's algorithm performs in a way that is expected when instrumentation is performed for each graph. Therefore, the actual performance of Dijkstra's algorithm is better than the theoretical performance.

Furthermore, Figure 2, which represents all the data within the above table, indicates that all the operations recorded while recording the algorithm fall under the theoretical performance bound which further highlights the above point that Dijkstra's algorithm is working as expected and is more efficient on average than the theoretical performance complexity.

Moreover, Figure 3 represents the relationship between the Actual performance vs the Theoretical performance of the algorithm (represented by the number of operations counted during the algorithm's execution). This graph represents a strong correlation between the actual vs theoretical performance (while the actual performance is still under the theoretical performance bond). This is seen as the data points are grouped closely around the line of best fit in the diagram. Furthermore, this is seen as this graph has a correlation coefficient of $R^2 = 0.9698$ which represents very strong positive correlation. Therefore, this indicates that Dijkstra's algorithm runs as expected seen in the theoretical performance bound of $|E|\log|V|$. This shows that as the size of edges and vertices increase with each dataset, the actual and theoretical performance scale proportionally and hence, Dijkstra's algorithm works as expected within the bound of $O(|E|\log|V|)$.

Creativity:

HashMap implemented to store random edges generated:

During this assignment I used a HashMap to save the edges that had been randomly generated to see if all new edges generated are unique before adding to the dataset. This was creative because the HashMap finds the result of the comparison in constant time and is very efficient when searching through large amounts of edges. If the match was found then the new edge would not be added to the file and if there was no match from this comparison, the edge was successfully added to the file. This was beyond basic assignment requirements because edge duplicates could be found in simpler and yet slower ways, was not described in depth in terms of Java implementation for the assignment and a HashMap is a very efficient way to complete this task.

CSV Files used to save data for efficient results analysis:

For the experiment, I stored the results of the performance in a CSV file instead of a text file. I did this by editing the code necessary to write data to a file. This was creative and goes past the scope of the assignment because the results were merely required to be stored in a text file and the use of a CSV file allows the data to be plugged directly into Microsoft Excel or Libre Office Calc to be analysed and displayed efficiently and with formatting. This allowed time to be saved when analysing and displaying results as it required less manual work to transfer the results into a meaningful data table or mathematical graph, thus being a creative way to improve efficiency for the experiment.

Looping experiment process to get results:

In the experiment I looped through all 25 datasets that I created for the experiment by passing the different dataset file names as arguments for the Graph.java main method in the GraphExperiment.java class. This looping allowed the program to loop through all edges in all datasets, create graphs, run Dijkstra's algorithm and count and record results for all datasets in the experiment. This was creative as it increased the experiment's efficiency by allowing it to be run on all datasets in one systematic process and record results accordingly.

Use absolute file paths to run experiment and organise data and results efficiently:

In the experiment, I used absolute file paths when reading in datasets for Dijkstra's algorithm, and when writing results to a results.csv file (both contained in the data directory). This was so that files

were organised effectively at all times during the experiment and so that files could be accessed efficiently at all points during the experiment. This was creative because the experiment merely requires results to be recorded and not organised in real time, only after the processes are run. However, in this case, the use of absolute file paths goes past this scope so that files are effectively and efficiently organized and accessed during the entire experiment.

Git Log:

andrew@andrew-ubuntu:~/uct-assignments/assignment5/graph-experiment$ git log | (ln=0; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; tail -10)

0: commit d818ecec7eef46633fcaacd242a01b634a0b42f2

1: Author: andrew-erasmus <andrew.r.eras@gmail.com>

2: Date: Thu May 4 19:27:06 2023 +0200

3:

4: Complete version of experiment

5:

6: commit f95b65586a80ff721cbf85048a25503adeb63316

7: Author: andrew-erasmus <andrew.r.eras@gmail.com>

8: Date: Thu May 4 19:14:51 2023 +0200

9:

...

109: Author: andrew-erasmus <andrew.r.eras@gmail.com>

110: Date: Tue Apr 25 12:11:15 2023 +0200

111:

112: Create DataGenerator.java to generate data and add it to a file

113:

114: commit 2c7e8e85488d1137ca875eee813220d04870713d

115: Author: andrew-erasmus <andrew.r.eras@gmail.com>

116: Date: Tue Apr 25 11:28:38 2023 +0200

117:

118: Add the graph files and bin, src and doc repos to project