University of Capetown

CSC2001F

Computer science

Author: **Talifhani Nemaangani**

Student Number: **NMNTAL002**

OOP DESIGN

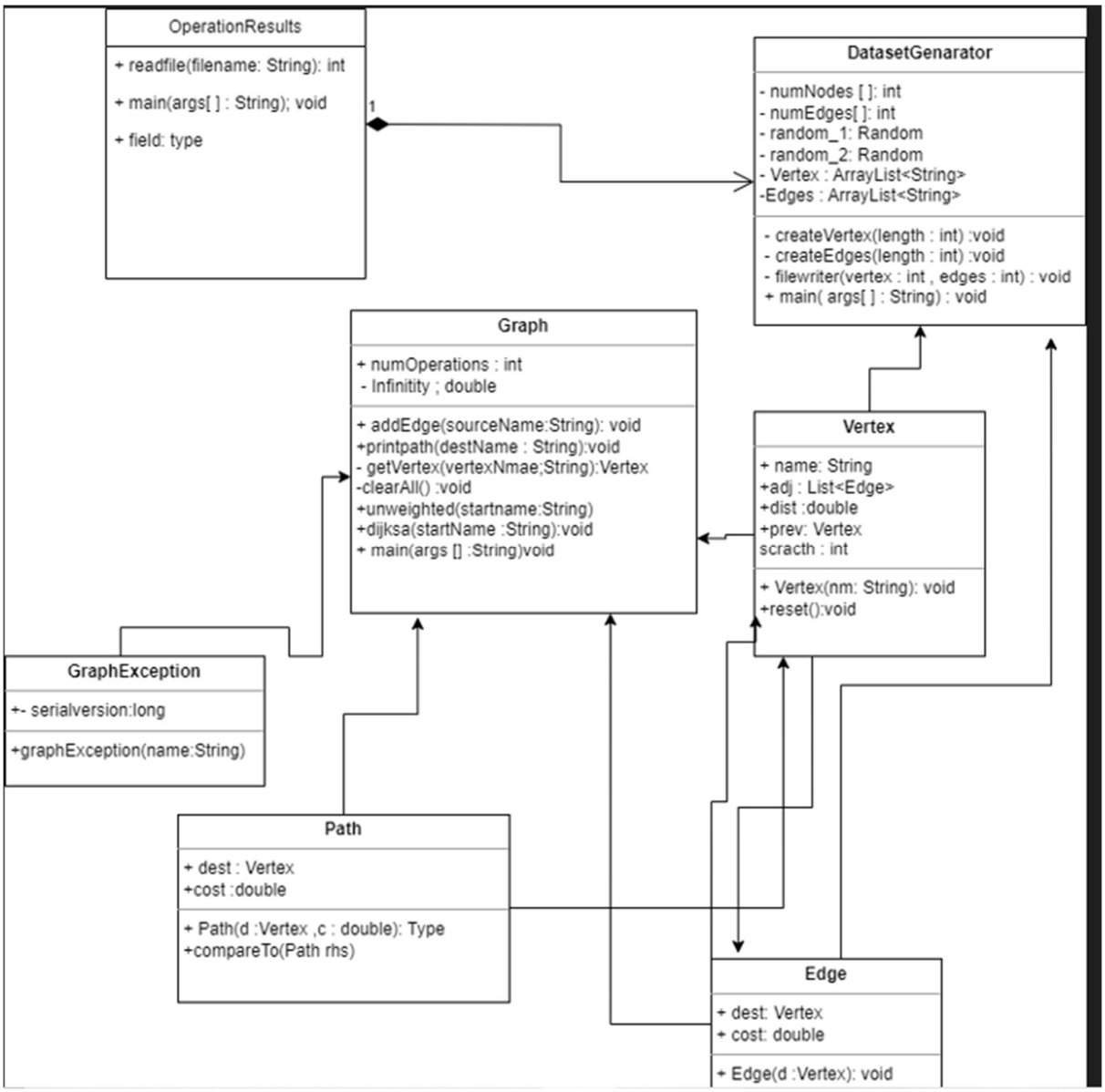
1. **Classes created.**

    - DatasetGenarator.java
    - Operationresults.java
    - Graph.java
    - GraphException.java
    - Edge.java
    - Vertex.java
    - Path.java

2. **Why were the classes created.**

    - The DatasetGenerator.java class is responsible to generate a dataset and store it to a dataset.txt file on disk. The dataset generated is made up of random number of vertices and edges which are associated with random cost/weight. The dataset files are going to be used for the experiment.
    - The OperationResults.java class is responsible for the reading the dataset.txt files stored on disk and loads the graph data from the dataset.txt files into a graph and then perform the Dijkstra's algorithm on the loaded graph to determine the shortest paths. Then it stores the instrumentation results in a .csv file on disk.
    - The remaining five classes are reused from Hussein Solemn which are responsible to implement the Graph structure and perform Dijkstra's algorithm and other related algorithms, and the Graph has few modification to allow instrumentation so that we can be able to perform the experiment.

3. UML
   The UML diagram below show the interactions of the classes


OperationResults
+ readfile(filename: String): int
+ main(args[ ] : String); void
+ field: type
1
DatasetGenarator
- numNodes [ ]: int
- numEdges[ ]: int
- random_1: Random
- random_2: Random
- Vertex : ArrayList<String>
-Edges : ArrayList<String>
- createVertex(length : int) :void
- createEdges(length : int) :void
- filewriter(vertex : int , edges : int) : void
+ main(args[ ] : String) : void
Graph
+ numOperations : int
- Infinitity ; double
+ addEdge(sourceName:String): void
+printpath(destName : String):void
- getVertex(vertexNmae;String):Vertex
-clearAll() :void
+unweighted(startname:String)
+dijksa(startName :String):void
+ main(args [] :String)void
Vertex
+ name: String
+adj : List<Edge>
+dist :double
+prev: Vertex
scracth : int
+ Vertex(nm: String): void
+reset():void
GraphException
+- serialversion:long
+graphException(name:String)
Path
+ dest : Vertex
+cost :double
+ Path(d :Vertex ,c : double): Type
+compareTo(Path rhs)
Edge
+ dest: Vertex
+ cost: double
+ Edge(d :Vertex): void

Aim

The aim of this experiment is to analyse the performance of Dijkstra's Algorithm on randomly generated graphs by measuring the number of vertex-processing and edge-processing operations. The experiment investigates the relationship between the algorithm's performance and the size of the graphs, which is determined by the number of vertices (|V|) and edges (|E|). This analysis will help in understanding the practical implications of the algorithm's time complexity and provide insights into its real-world performance.

Methodology

The methodology for this experiment involves a systematic approach to evaluate the performance of Dijkstra's Algorithm on randomly generated graphs. First, a Java application called DatasetGerator.java is developed to generate random graphs with different values of vertices (|V|) and edges (|E|), while ensuring that the edges are unique and have non-negative integer weights within a specified range. These generated graphs are then saved to disk for validation and future reference. Next, we created another java application called operationResuts.java that loads the saved graphs and runs Dijkstra's Algorithm to determine the shortest paths, using the first vertex in each dataset as the source. The algorithm's performance is analysed by instrumenting the code to count the number of vertex-processing and edge-processing operations whereby this was done by modifying the Graph.java by adding a counter. We then stored the results in a .csv file so that we can use the results to plot the graphs.

Assumptions

In generating the random graphs for this experiment, several assumptions were made to simplify the process and ensure compatibility with Dijkstra's Algorithm. Firstly, I assumed that the edge weights follow a uniform distribution within a specified range, meaning that each possible weight has an equal probability of being assigned to an edge. This approach simplifies the generation process and allows for a more straightforward analysis of the algorithm's performance.

Secondly, the edge weights were assumed to be integers within a range of 1 to 10. By using integer weights, the complexity of the experiment is reduced, and the results can be more easily interpreted. However, the choice of range and the restriction to integers may influence the outcome, and adjustments may be necessary depending on the specific requirements of an experiment or application.

Thirdly, I assumed the absence of negative edge weights in the generated graphs. This is because Dijkstra's Algorithm is not designed to handle negative edge weights correctly, which could lead to incorrect results or infinite loops.

I also assumed that the number of vertices will range from V = {10,20,30,40,50}; and edges will be E= {10,15,20,35,50,65,80,85,90,100} which means that I generated 50 dataset.txt files on disk.

Lastly, while generating random edges, I assumed that duplicate edges and self-loops (edges with the same source and destination) would not be allowed. This ensures that the resulting graphs are simple, undirected graphs, which are suitable for applying Dijkstra's Algorithm.

These assumptions provide a solid foundation for generating random graphs and analysing the performance of Dijkstra's Algorithm. However, it is important to acknowledge that the results of the

experiment may be influenced by these assumptions and may not necessarily generalize to all types of graphs and weight distributions.

Results

Number of Vertices V/S Operations

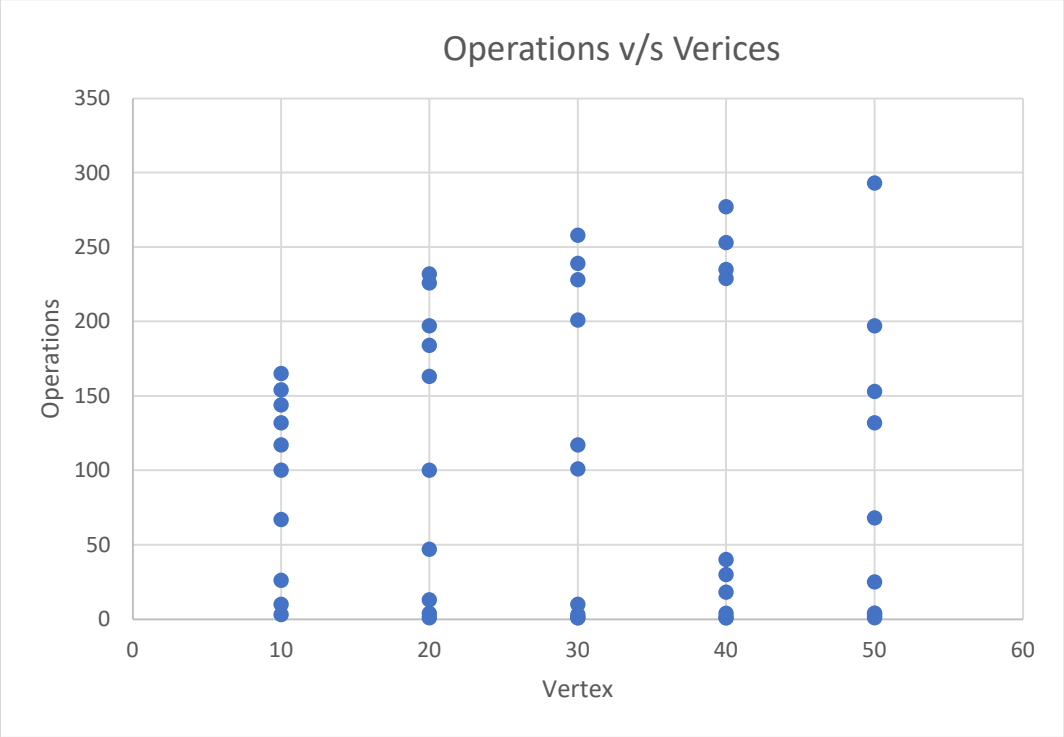


Figure 1

Number of Edges V/S Operation

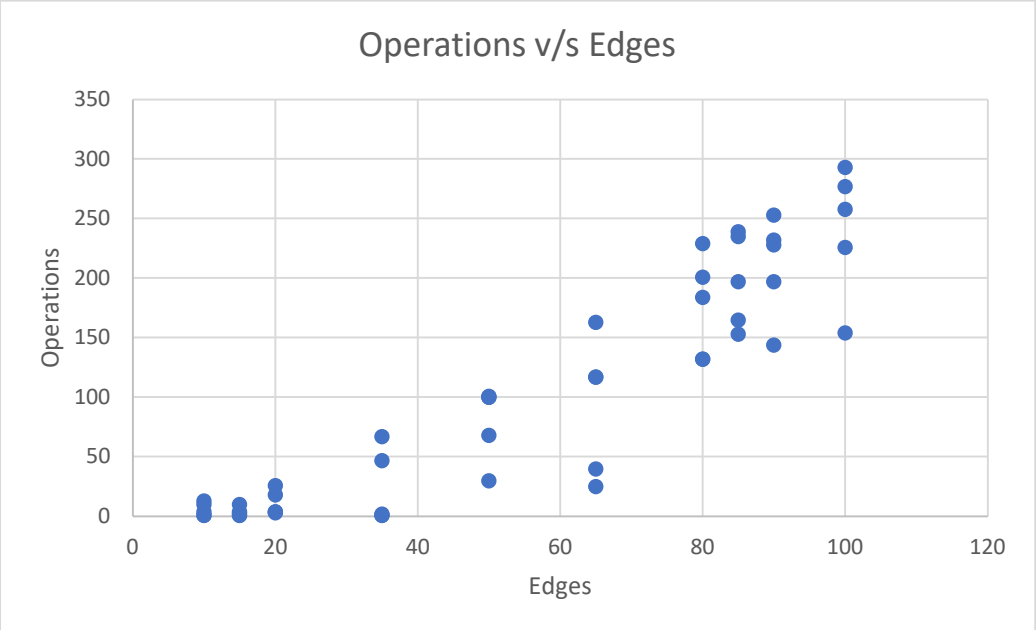


Figure 2

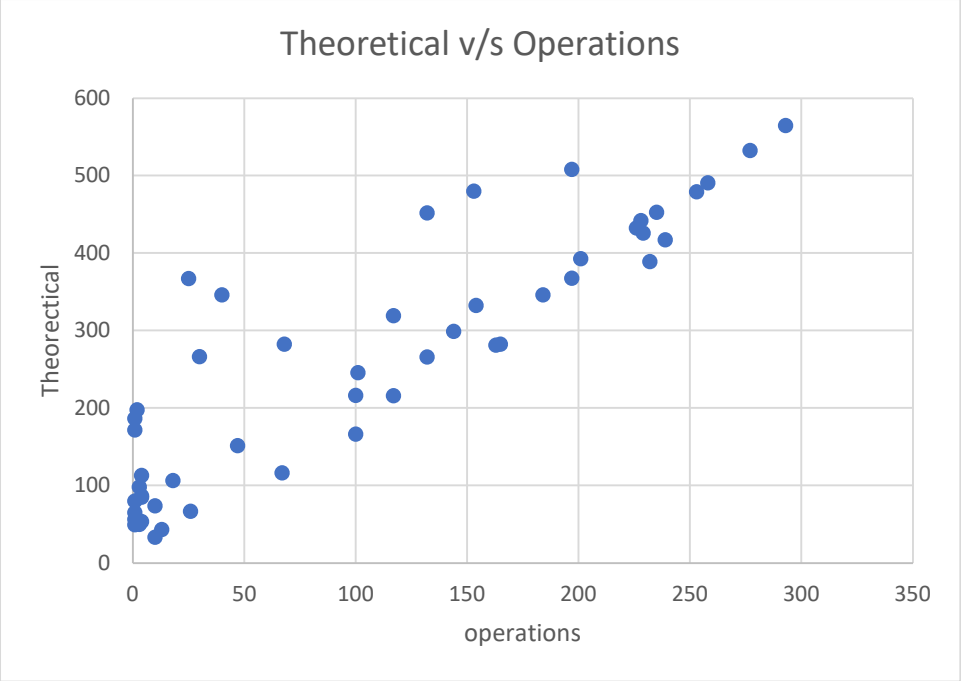


Figure3

Operations vs Theoretical

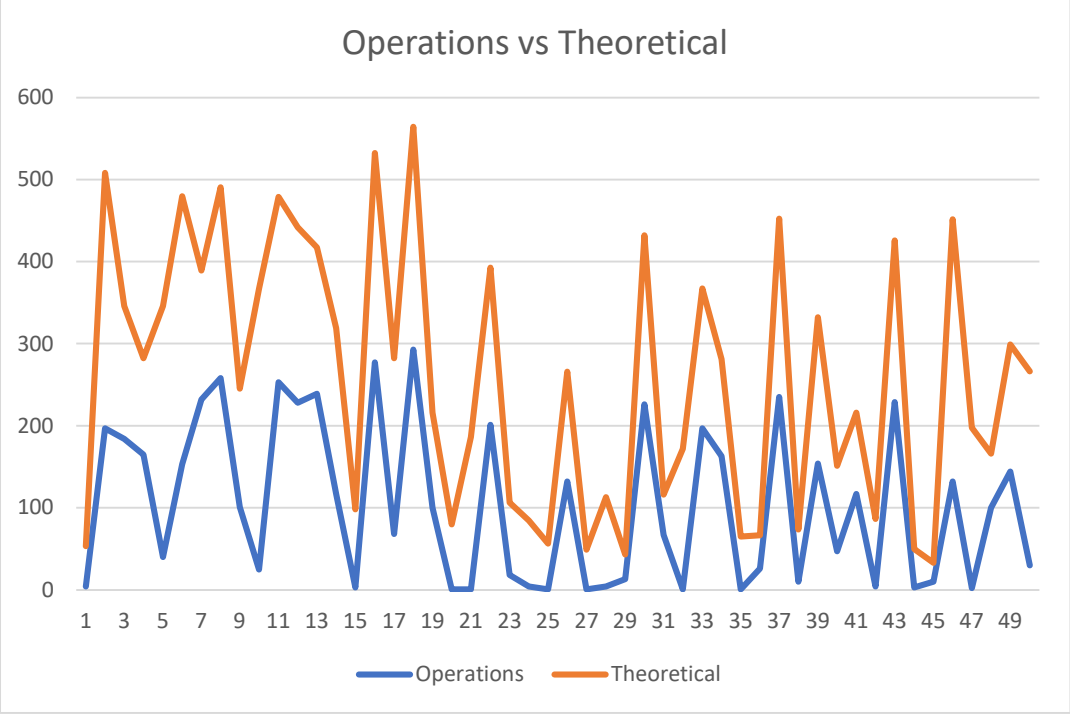


Figure 4

Conclusion

In figure 1, we cannot deduce much between the relationship of the number of vertices and the number of operations as the plot is not increasing linearly. In figure 2 the experiment demonstrates that the performance of the Dijkstra algorithm for shortest path calculation is also significantly affected by the number of edges in a graph. As the number of edges increases, both enqueue and dequeue operations exhibit an increase, leading to a higher computational cost. This correlation emphasizes the importance of optimizing the Dijkstra algorithm for graphs with a higher density of edges to ensure efficient performance.

Additionally, it may be beneficial to explore alternative algorithms that are better suited for specific use cases or graph structures with varying edge densities. These algorithms may offer improved performance for graphs with many edges or varying levels of connectivity. By considering both the number of edges and the computational cost of operations, future research can focus on enhancing the efficiency and applicability of shortest path algorithms in a diverse range of applications.

Figure 3 presents a visual representation of the correlation between the number of operations and the theoretical expectations within the Dijkstra algorithm, showcasing its effectiveness in managing time complexity. When the plotted data points cluster around the theoretical line, it signals the algorithm's proficiency and adherence to the predicted time complexity.

The graph in Figure 4 highlights the interplay between the algorithm's theoretical bounds and its actual operational performance. The theoretical plot acts as a benchmark for the algorithm's optimal performance, which is dictated by the E*log|V| function. In contrast, the operations plot reveals the real-world execution of the algorithm, accounting for the operations required to handle specific inputs.

Examining the graph, it's evident that the operations plot consistently stays at or below the theoretical plot, suggesting that the algorithm is indeed limited by the E*log|V| function. Moreover, the operations plot demonstrates a growth rate that is either equal to or less than that of the theoretical plot, verifying that the algorithm's performance remains within the predetermined boundaries.

To summarize, the information displayed in Figure 4 supports the assertion that the Dijkstra algorithm has a worst-case time complexity of E*log|V|, and its practical performance consistently stays within this threshold. This knowledge is instrumental in evaluating the algorithm's effectiveness and benchmarking it against other algorithms intended for similar applications.

Git log

```
talifhani@talifhani-VirtualBox:~/Desktop/CSC2001F/Assignment2$ git log | (ln=0; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -1
0; echo ...; tail -10)
0: commit f602780db25b08f30fa96575fa8459c7e8c94d15
1: Author: Talifhani Nemaangani <nmntal002@myuct.ac.za>
2: Date: Sat May 6 22:58:39 2023 +0200
3:
4: Done
5:
6: commit 10b0f23e0bd698aa3741a716e2a127f1fb6936b4
7: Author: Talifhani Nemaangani <nmntal002@myuct.ac.za>
8: Date: Sat May 6 14:46:57 2023 +0200
9:
...
25: Author: Talifhani Nemaangani <nmntal002@myuct.ac.za>
26: Date: Fri May 5 20:18:02 2023 +0200
27:
28: Graph experiment done
29:
30: commit edd8b7b723da85bde90e9db02673f29c46757aa3
31: Author: Talifhani Nemaangani <nmntal002@myuct.ac.za>
32: Date: Tue May 2 00:13:37 2023 +0200
33:
34: 1st commit , created repo
```