Experiment for performance of Dijkstra's shortest paths algorithm.

The goal of this assignment is to programmatically compare the performance of Dijkstra's shortest paths algorithm (as per the course slides) with the theoretical performance bounds.
The design consists of several classes that work together to implement a graph data structure and provide related functionalities. Here's an overview of the classes and their interactions:

1. `Vertex`: This class represents a node in the graph. It provides the basic functionality required to represent a vertex in a graph and initialise its attributes.

2. `Edge`: This class represents an edge in the graph. It has a cost and connects two vertices. It provides methods to get information about the edge, such as its cost and the vertices it connects.

3. `Path`: This class represents a path between two vertices in the graph. It consists of a list of edges and provides methods to get information about the path, such as its cost and the vertices it connects.

4. `Graph`: This class represents the graph itself. It is used to evaluate the shortest paths in a weighted graph using Dijksta's algorithm. It contains methods to add edges to the graph, print the shortest path from source vertex to a destination vertex, and perform Dijkstra's algorithm with added instrumentation to count the number of times of vertex, edge, and priority queue processing operations.

5. `GraphException`: This class represents an exception that can occur when performing operations on the graph, such as adding an edge that connects non-existing vertices.

6. `DataGenerator`: This class is responsible for generating random graph data for testing purposes. It provides methods to generate graphs with a specified number of vertices and edges, as well as methods to write the generated data to files. I have created datasets that represents the graph that will be inserted into the Graph class and do Dijkstra experiment. The dataset is randomly generated and has source vertex that has no incoming edges and connected to other edges either directly or indirectly. No same vertices on one edge and no duplicate edges.

7. `GraphExperiment`: This class is the entry point of the program. It creates a `Graph` object, adds the generated data to the graph, and performs Dijkstra's algorithm on the graph to find the shortest paths. It then outputs vertices, edges, vertices count, edges count and priority queue count for each generated dataset.

The `Vertex`, `Edge`, and `Path` classes are used by the `Graph` class to represent the graph data, while the `GraphException` class is used to handle exceptions that can occur during graph operations. The `DataGenerator` class is used by the `GraphExperiment` class to add generated graph data to 'Graph' class, and the `Graph` class is used by the `GraphExperiment` class to perform Dijkstra's algorithm and output vertices, edges, vertices count, edges count and priority queue count (operations) for each generated dataset.

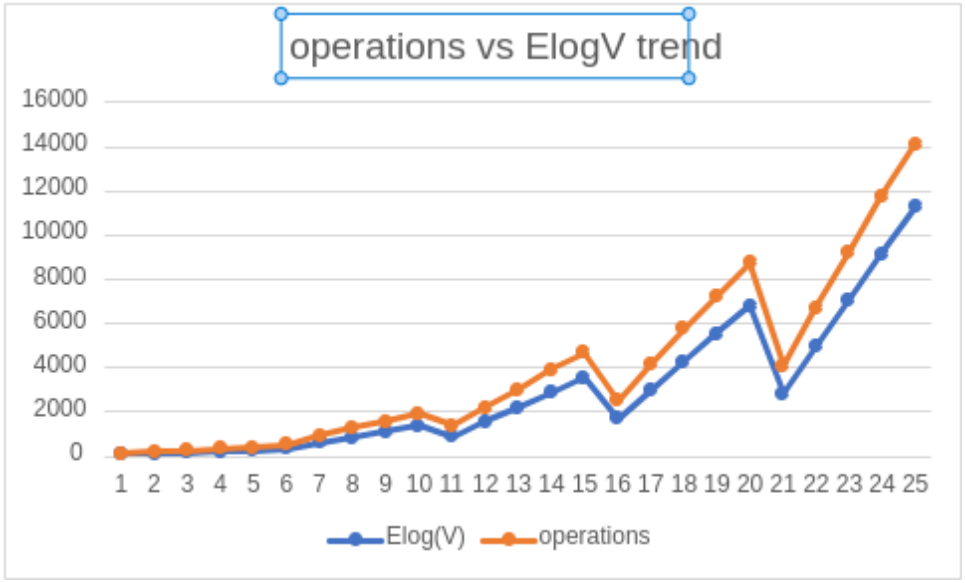
Experiment:

I have modified the Graph class by inserting vCount, eCount, pqCount(vertex, edge, priority queue count whenever vertex, edge, priority is processed in dijstra() method) . PqCount calculates the logarithm base 2 of the current size of the priority queue operations, which is proportional to the number of levels of the priority queue, and it gives a more accurate estimation of dijkstra's algorithm performance. I inserted each dataset generated into the graph and did disjktras algorithm on "Node000"(source vertex) and the output is (vertices, edges, vCount, eCount, pqCount).

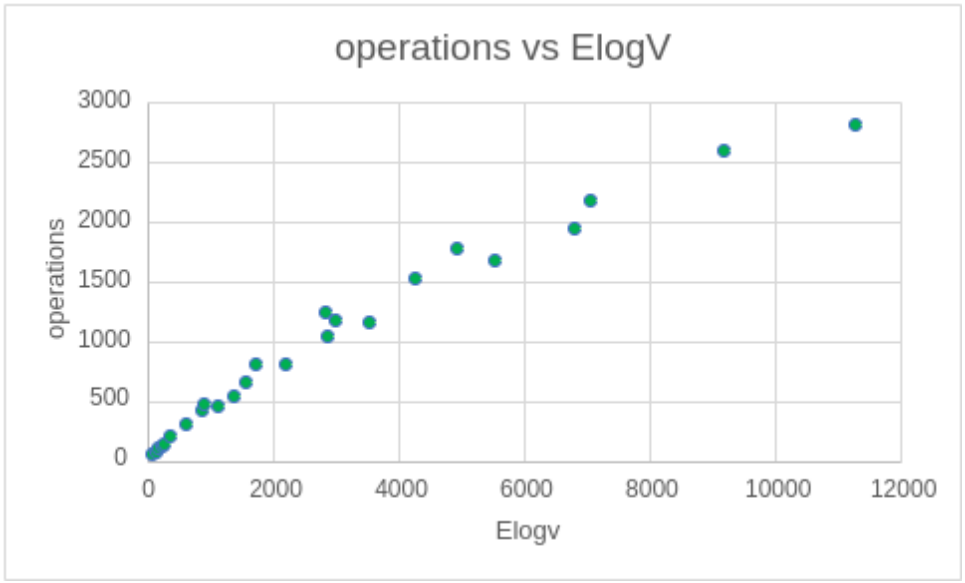
Below is a table from GraphExperiment output (vertices, edges, vertices count, edges count and priority queue count) for each generated dataset. With operations derived from (vCount+eCount+pqCount) and ElogV (complexity for Dijkstra algorithm) derived from (Edges*$\log_2$(Vertices))

vertices	edges	vCount	eCount	pqCount	Elog(V)	operations
10	20	10	20	24	66.43856	54
10	35	10	35	33	116.2675	78
10	50	10	50	41	166.0964	101
10	65	10	65	48	215.9253	123
10	80	10	80	52	265.7542	142
20	80	20	80	102	345.7542	202
20	140	20	140	151	605.0699	311
20	200	20	200	198	864.3856	418
20	260	20	260	176	1123.701	456
20	320	20	320	193	1383.017	533
30	180	30	180	270	883.2403	480
30	315	30	315	314	1545.671	659
30	450	30	450	324	2208.101	804
30	585	30	585	424	2870.531	1039
30	720	30	720	408	3532.961	1158
40	320	40	320	454	1703.017	814
40	560	40	560	578	2980.28	1178
40	800	40	800	689	4257.542	1529
40	1040	40	1040	603	5534.805	1683
40	1280	40	1280	625	6812.068	1945
50	500	50	500	700	2821.928	1250
50	875	50	875	858	4938.374	1783
50	1250	50	1250	873	7054.82	2173
50	1625	50	1625	928	9171.266	2603
50	2000	50	2000	762	11287.71	2812

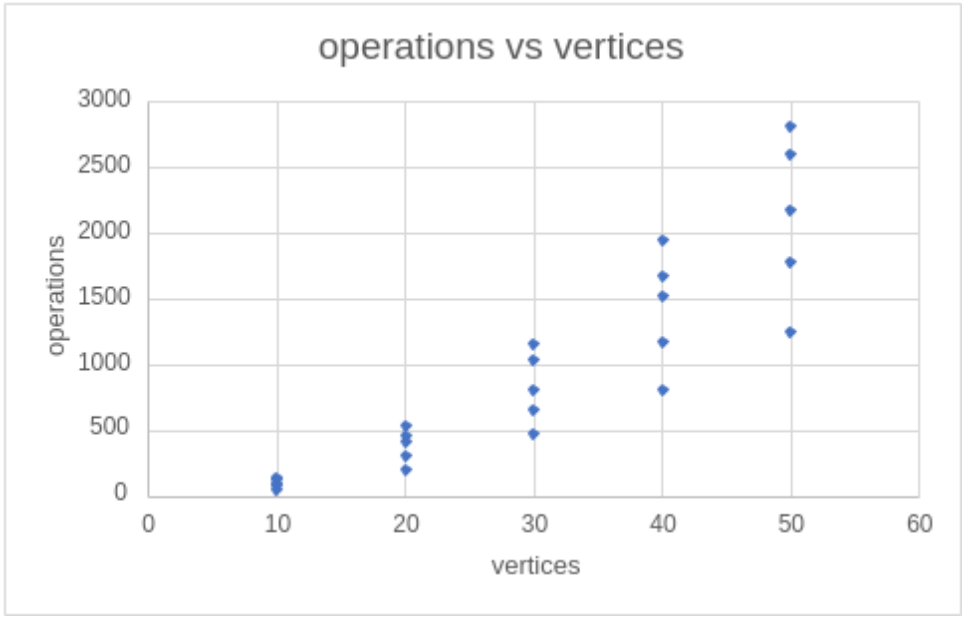
Below is a trend of operations vs ElogV which provides information about performance of Dijkstra's algorithm and how it relates to the size and structure of the graph being processed over time.



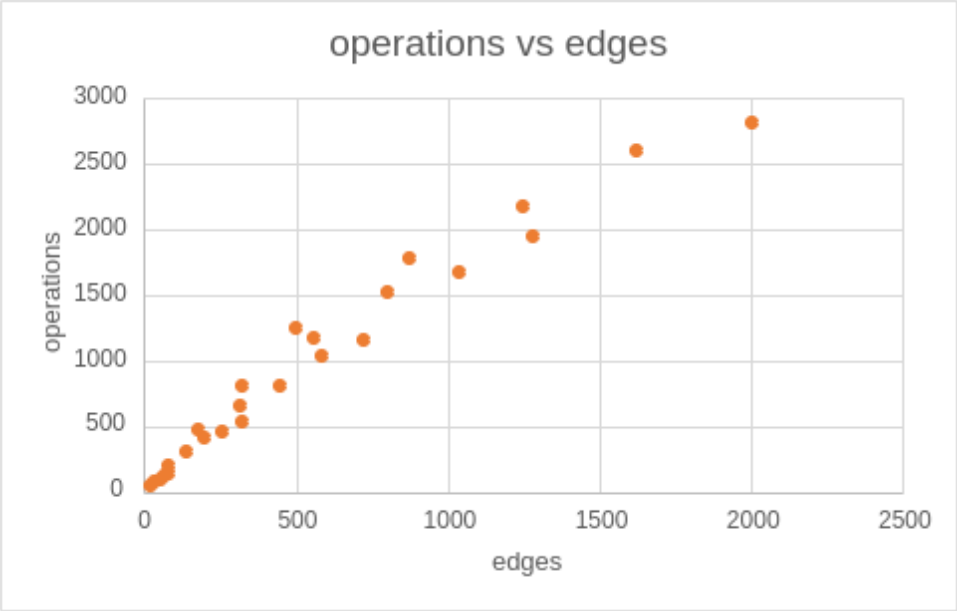
Below is a scatter plot of operations vs ElogV showing performance of Dijkstra's algorithm and how it relates to the size and structure of the graph being processed.



The scatter plot below shows operations vs vertices which is how the algorithm processes each vertex.



The scatter plot below shows operations vs edges which is how the algorithm processes each edge.



Results:

Based on the results of the experiments for Dijkstra's algorithm with different graph sizes (number of vertices and edges), it is clear that the number of operations required to run the algorithm increases as the size of the graph increases. This is expected as Dijkstra's algorithm is a shortest path algorithm that involves processing each vertex and edge in the graph.

The experiment results show that the number of operations increases roughly logarithmically with the number of vertices and edges. This is expected as Dijkstra's algorithm has a time complexity of O(Elog V), where E is the number of edges and V is the number of vertices. Therefore, the number of operations should increase roughly logarithmically with the number of vertices and edges.

The experiment results also show that the number of priority queue operations (pqCount) increases with the number of edges and is higher than the number of edges for small graphs but becomes lower than the number of edges for larger graphs. This is because the priority queue is used to keep track of the vertices to be processed in Dijkstra's algorithm, and the number of vertices in the priority queue can be smaller than the number of edges in the graph. As the graph size increases, the priority queue becomes more efficient at keeping track of the vertices to be processed.

In conclusion, the experiment results confirm that Dijkstra's algorithm has a time complexity that increases logarithmically with the size of the graph, and that the number of priority queue operations increases with the number of edges in the graph.

Git:

```
0: commit 9e587a6caff04f89fa4c4285e7c0d8298960cdf9
1: Author: Rector Ratsaka <rtsrec001@myuct.ac.za>
2: Date: Thu May 4 01:02:58 2023 +0200
3:
4: vCount,eCount and pqCount added on Grapg class for intrumentation
5:
6: commit 89918b901ff834f38697676a198912b6892396fc
7: Author: Rector Ratsaka <rtsrec001@myuct.ac.za>
8: Date: Wed Apr 26 13:22:35 2023 +0200
9:
...
13: Author: Rector Ratsaka <rtsrec001@sl-dual-282.cs.uct.ac.za>
14: Date: Mon Apr 24 14:16:29 2023 +0200
15:
16: implementation of GraphGenerator class with 5 saved(data) text files
17:
18: commit 166dd87ac7b71a13d5c49fbc87794637ca80b518
19: Author: Rector Ratsaka <rtsrec001@myuct.ac.za>
20: Date: Fri Apr 21 02:37:52 2023 +0200
21:
22: Initial commit
rector@rector-VivoBook-ASUS:~/Documents/Assignment5$
```