**Design**   
This OO program creates sets of nodes and vertices desired by the user to create a graph that is positive weighted, acyclic, directed and dense. The program then uses Dijkstra's algorithm to evaluate the shortest path between the nodes; the program iterates this process to achieve average values for the amount of computation done for graphs of various sizes.

**Classes**

DataSet

This class prompts user to enter the amount of desired vertices and edges to create a graph. It then generates a text file with edges to be inserted in a graph

GraphExperiment

This class reads the edges of the text file generated in the DataSet class and inserts it within the graph. Once the edges have been inserted, Dijkstra’s algorithm is then run between 2 randomly selected nodes. This process is repeated 10 times and the average amount of nodes, edges and time taken to execute Dijkstra’s algorithm is then printed to the screen.

Path

Represents an entry in the priority queue for Dijkstra's algorithm.

Vertex

Represents a vertex in the graph.

GraphException

Used to signal violations of preconditions for various shortest path algorithms.

Graph

Creates a graph

**Goal of Experiment**

The goal of this experiment is to create a graph and programmatically compare the performance of Dijkstra's shortest paths algorithm with the theoretical performance bounds using the aforementioned graph.

**Experiment Design**

1. Using the DataSet class I created Graphs of various amount of vertices {10,20,30,50,100,200) for each of amount of vertices I then created various amount of edges {20,35,50,65,80} for each amount of vertices(PICTURE) .
2. In the GraphExperiment class the edges are inserted into a graph.(PICTURE)
3. Once the edges are inserted into the graph, Djikstra’s algorithm is then used to find the shortest path between two random vertices within the graph. (PICTURE)
4. The amount vertices and edges processed when finding the shortest path between the two random vertices are recorded as vCount and eCount.
5. The time taken for Djikstra’s algorithm to find the shortest path taken is recorded as execution Time.
6. Steps 2-5 are repeated 20 times and the average of vCount,eCount and execution Time is then recorded.
7. Repeat step 2-6 for each amount of vertices and for each of these vertex amounts with various amount of edges described in step 1.

## **Dijkstra Algorithm Time Complexity**

Implementing Dijkstra's algorithm using an adjacency matrix representation has an O(V2) time complexity.

Because a VxV matrix is needed to store the representation of the graph, the space complexity of the algorithm's adjacency matrix representation of the graph is also O(V2). The algorithm will additionally employ a second array of V length to keep track of each vertex's state, although overall space complexity will still be O(V2).

Using an adjacency list representation of the graph and a min-heap to store the unvisited vertices, the time complexity of Dijkstra's algorithm can be decreased to O((V+E)logV), where V is the number of vertices and E is the number of edges in the graph.

With this implementation, it takes O(V+E) time to visit each vertex, and O(logV) time to analyze all of a vertex's neighbors.

Time for visiting all vertices = O(V+E)

Time required for processing one vertex= O(logV)

**Results**

Figure 1: Graph comparing theoretical bound of average number of vertices processed to average number of vertices processed in the experiment

Figure 2: Graph comparing theoretical bound of average number of edges processed to average number of edges processed in the experiment

Figure 3: Graph comparing theoretical bound of average time taken to run Dijkstra’s algorithm

to average time taken to run Dijkstra’s algorithm in the experiment

**Discussion of results**

From the Dijkstra’s algorithm Time Complexity section we see that the amount of vertices processed required to find the shortest path is O(logV) and this is confirmed in the results of the experiment. In Figure 1 the average number of vertices processed in the experiment does not exceed the theoretical bound.

From the Dijkstra’s algorithm Time Complexity section we see that the amount of edges processed required to find the shortest path is O(logV) and this is confirmed in the results of the experiment. In Figure 2 the average number of edges processed in the experiment does not exceed the theoretical bound.

Dijkstra’s algorithm Time Complexity section we see that the time required for processing the shortest path is O(logV) and this is confirmed in the results of the experiment. In Figure 3 the time taken to calculate the shortest path using Dijkstra’s algorithm has logarithmically as the number of Vertices and edges increase confirming that the time complexity of Dijkstra’s algorithm is O(logV).

**Creativity**

One major point of creativity added to the experiment was to not only measure the number of vertices and edges processed but to measure the time Dijkstra’s algorithm took to calculate the shortest path. Using the time sampling package in Java the execution is calculated to the nanosecond. While the measure of the how many vertices and edges processed does provide insight into the time complexity of the Dijkstra’s algorithm, the formal definition of time complexity of an algorithm is “The time complexity of an algorithm is the amount of time the algorithm takes to solve a problem expressed as a function of the problems size” going by this definition measuring the time taken to execute Dijkstra’s algorithm for graphs of different sizes provides a more accurate measure of the time complexity of Dijkstra’s algorithm.

To get the most comprehensive results once a graph with a set number of vertices and edges was created I ran Dijkstra’s algorithm on two randomly selected nodes to get a better idea of the average and this process was iteratively repeated 20 times to provide more comprehensive results

.