**CPSC 331**

**Calder Sloman, 30107102**

**Rezwan Ahmed, 30134609**

**June 10th, 2023**

**CPSC 331 - Assignment #5**

**Graphs**

**Part 1:**

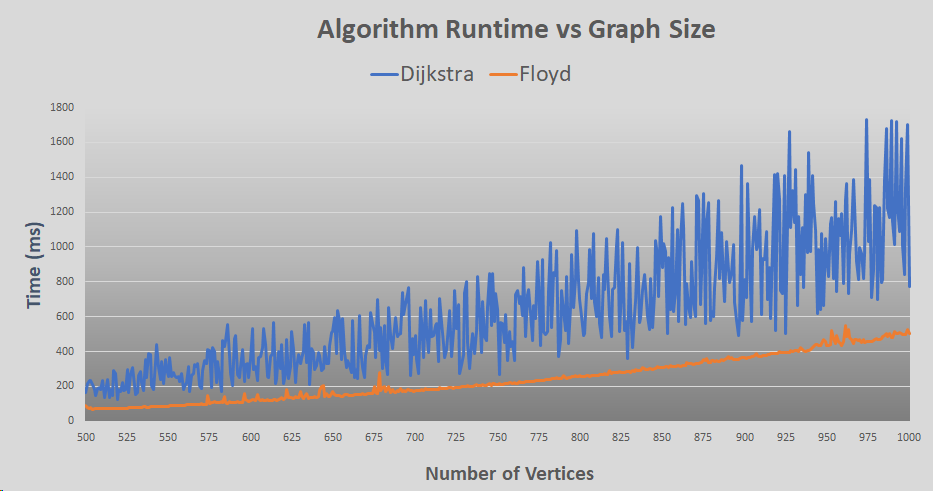


Figure 1

We can observe from Figure 1 that as the number of vertices of a graph increases so does the time required to obtain the All-Pairs-Shortest-Path of that graph. Even though both algorithms show an overall gradual increase in time taken as the number of vertices in the graph increases, Floyd's algorithm has more consistent execution times as the graph size increases. The execution times for Floyd's algorithm exhibit less variation compared to Dijkstra's algorithm, which has much more dramatic fluctuations in runtime.

Furthermore, we can observe that the execution times of Dijkstra's algorithm are consistently higher than those of Floyd's algorithm. One reason that Dijkstra's algorithm generally has higher execution times compared to Floyd's algorithm could be attributed to the fact that Dijkstra's algorithm operates on a single source vertex at a time, whereas Floyd's algorithm considers all pairs of vertices simultaneously.. Dijkstra's algorithm requires multiple iterations to process each vertex individually, resulting in additional computational overhead. In contrast, Floyd's algorithm performs a fixed number of iterations to calculate the shortest paths between all pairs of vertices in a single run. Dijkstra’s algorithm is also not running as efficiently as it could had we implemented a min-heap data structure; this results in a complexity of O(n) in the worst case and could explain the dramatic shifts in the runtime of the algorithm.

While both algorithms experience higher runtimes times with as the number of vertices in the graph increases, Floyd’s algorithm outperforms Dijkstra’s across the range.

**Part 2:**

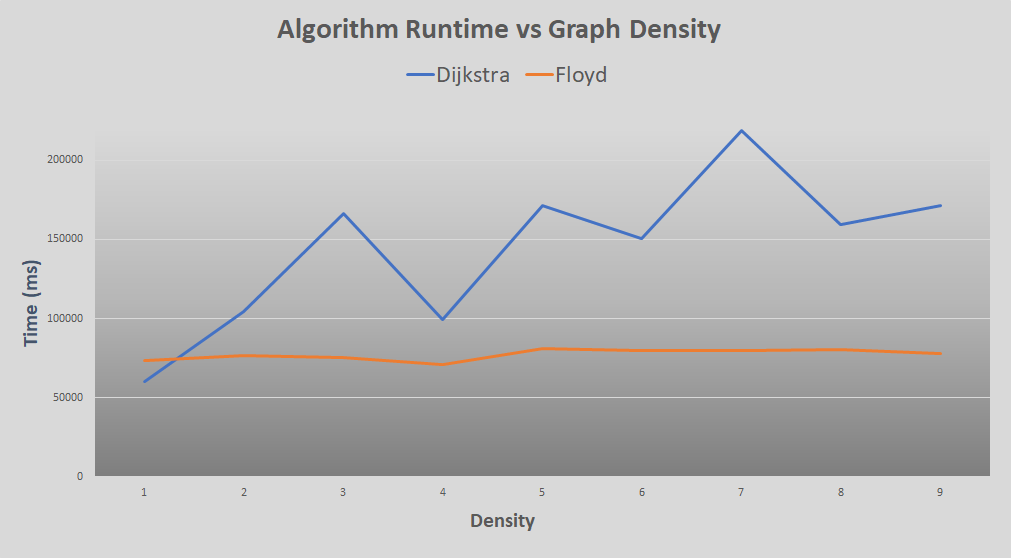


Figure 2

An even more stark difference in the performance of each algorithm emerges when we compare their runtimes on graphs with the same size and varying density. As we can see in Figure 2, the runtime of Floyd’s algorithm remains stable as the density of the graph increases – in fact, the changes in runtime are small enough that they are likely attributable to the background processes occurring on the system executing the algorithm. Dijkstra’s algorithm on the other hand is clearly affected by graph density, and its runtime increases in step with the number of edges in the graph. This is an important test as we can clearly see a difference in the performance of each algorithm as the ratio between edges and vertices changes. Dijkstra’s algorithm performs well when the density is below 1.5 (i.e. a sparse graph), but its performance suffers considerably when that is not the case. Floyd’s algorithm outperforms Dijkstra’s as soon as the density is greater than 1.5.

The reason for this difference in runtime can be explained by looking at the factors that contribute to their complexity. Dijkstra’s algorithm is keenly affected by the number of edges due largely to the fact that we did not implement a min-heap data structure when implementing the “find minimum vertex” part of the algorithm. In the worst case, this means that an additional n executions of a section of code must occur (where n = the number of vertices in the graph). Floyd’s algorithm on the other hand is O(V3) where V is the number of vertices in a graph, meaning that its runtime remains unaffected by a dense graph with many edges.

If I was only interested in calculating the Single-Source-Shortest-Path(SSP) I would choose Dijkstra’s algorithm, as without the minor modification to the code to make it an APSP algorithm, it does just that. Additionally, if I wanted to run an APSP algorithm on a very sparsely populated graph, I would also choose Dijkstra’s algorithm. In all other cases, I would choose Floyd’s algorithm due to its relative simplicity and faster runtime on graphs with higher density.