# Analysis of computation time for Dijkstra's Algorithm on randomly generated networks

Erik Aho

## Introduction

### Abstract

Dijkstra's algorithm, written by Edsger Dijkstra in 1956, is a search algorithm that finds the shortest path between nodes in a network.

In this context, a 'node', or 'vertex', is a point in a graph/network which is connected to other nodes by 'paths', or 'edges'. Starting from a chosen node, Dijkstra's algorithm is able to quickly find the shortest path to all nodes in the network.

It was one of the earliest shortest path algorithms, and while fast, more optimized variants now exist for certain use cases. Often these optimizations occur by cutting down on the size of the subgraph which must be searched using heuristic functions (see A*), allowing the algorithm to handle negatively weighted paths (Bellman-Ford, Johnson's), or by introducing priority queues (Fibonacci Heap, or BFS for ordered trees). That said, Dijkstra's algorithm is still implemented in many fields, and is the basis for many programs today, from network routing protocols (such as IS-IS and OSPF) to apps which allow you to find the quickest way to get home from work on the train.

The purpose of this project was two-fold; to construct a script in Python that is capable of generating random networks with a specified number of nodes and paths with random weights, and to examine how time to compute the shortest path from one node to all others varies with the complexity of the network.

### Example

Here I will give a small example of how Dijkstra's algorithm functions.

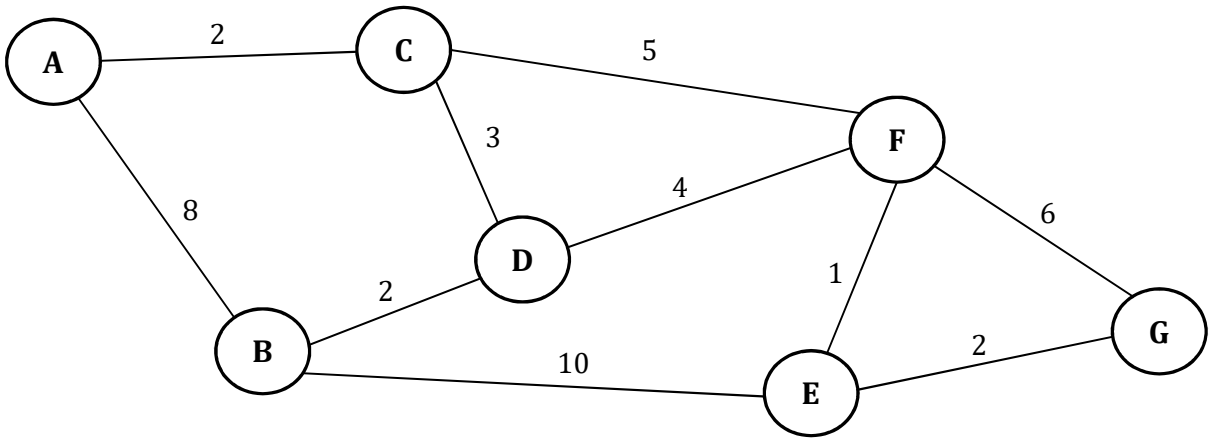


Figure 1: an example of a network, where the lettered nodes (A-G) are connected by paths with the indicated weights.

In Fig 1 above, we can see 7 nodes (lettered A-G) connected via paths with weights between 1 and 10 arbitrary units. Starting from node A, this is how Dijkstra's algorithm works.

- See which nodes are accessible from A
- Update accessible nodes (B,C) with distances from A.
    - AC has distance 2, AB has distance 8
- Select to 'visit' next closest node from A
    - Here, C is closest
- See which nodes are accessible from C that have not been visited yet (D,F)
- Update shortest distances from A
    - ACD has distance 5, ACF has distance 7
- Select to 'visit' next closest node from A
    - Here, D is closest
- See which nodes are accessible from D that have not been visited yet (B,F)
- Update shortest distances from A
    - ACDB has distance 7, which is shorter than previous shortest path to B (AB=8)
        - Update shortest path to B from A
    - ACDF has distance 9, which is greater than previous shortest path to F (ACF= 7)
        - Shortest path to F is not updated

The algorithm continues in such a way, visiting the next closest node, updating shortest paths to unvisited nodes, and so on, until either the chosen target node is visited (meaning there are no possible shorter routes to get to it) or all nodes have been visited, in which case a shortest path tree has been formed from the previous network.

Dijkstra's algorithm runs in time

$$O(|E| + |V|log|V|) \quad (1)$$

meaning we should see the run time being directly proportional to E, as well as directly proportional to V logV, where E is the number of edges and V is the number of vertices.

# Method

Python was used to write a script that generates a network with X nodes and Y paths between the nodes, where X and Y could be controlled independently. A second python script was written (*FullScriptSingleRun.py* in the repository) for the purpose of showing how the script works step by step, with each stage being shown as an output in the terminal.

*RunTimer.py* is the main script used to generate the data. Starting at 10 nodes in the network, it generates a random network with 10 nodes and 9 paths (the minimum possible count of paths for the network to be connected), then times how long it takes Dijkstra's algorithm to run on this random network. This is done 2500 times for 10 nodes and 9 paths, with each run's timing being saved. This is then repeated for each possible count of paths for 10 nodes (between n-1 and nC2 paths). This process is then repeated for 11 to 30 node networks.

The above choices were made in an attempt to balance between collecting enough data to average out the randomness inherent on timing a protocol like this on a computer, and trying to avoid the script taking 10 weeks to complete. With the above parameters and on my current hardware (CPU: i7-7700HQ @ 2.8 GHz, Memory: 16 GB DDR4) the script took ~ 2 days to run.

After collecting the data, the following figures were created using the PyPlot library in the *graphing.py* script.

External Libraries used: NumPy, PyPlot, Dill

# Discussion of results

## Single vertex count correlation

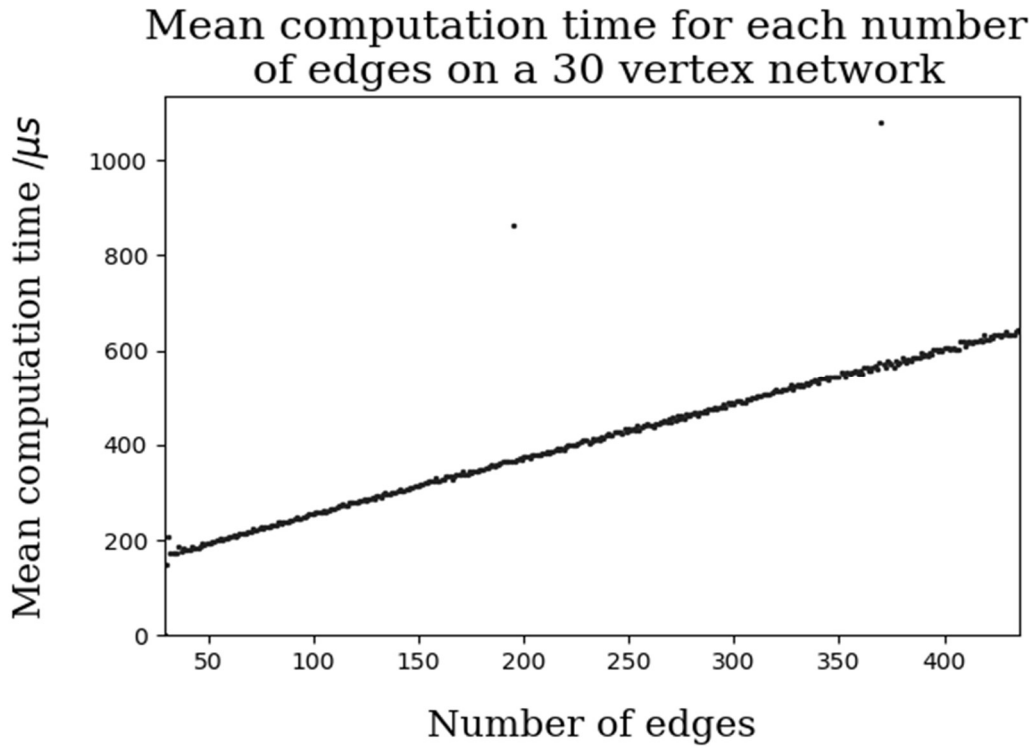


Figure 2: A plot showing the mean computational time for all 2500 runs for each number of edges in a 30 vertex network.

In figure 2 (above), we can see a very strong and obvious linear correlation between the number of edges and the run time of Dijkstra's algorithm ($R^2 > 0.99$). Three main areas have significant deviation from this trend; the first few edge counts, and the timings for edge counts of 195 and 370 edges.

The first few edge counts are likely to have more anomalous timings due to how this script generates random networks. The edges are randomly assigned between pairs of nodes, and so it is quite likely that for smaller number of edges in the network, the network will not be connected (meaning that you can't get to all nodes from node A). The algorithm handles this by simply not recording timing data for non-connected networks, and so for the smaller edge counts, there are fewer recordings, leading to greater variance in the mean. Figure 3 (right) shows how many non-connected plots occurred for each of the number of edges.

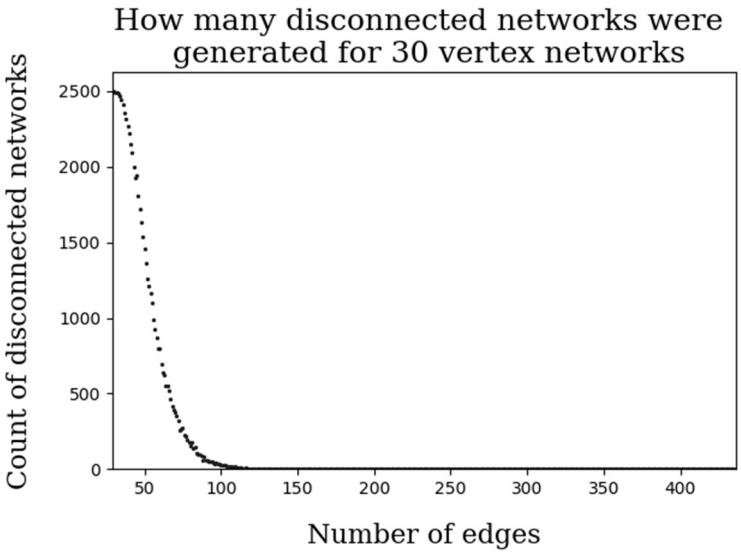


Figure 3: A plot showing how many networks which were not connected were generated for 30 vertex networks

Now for the more interesting points. The spike at 195 edges is quite unusual. All of the path counts surrounding it agree very strongly with the linear trend, but it alone does not. This seems to be an erroneous result, as does the result at 370.

Let us look more in detail at the results which are being averaged out to provide the points in figure 2 to try to understand these results better. Below is Figure 4, in which we show the timing results of all 30 vertex, 190 edge networks. These points were averaged out to make the single point at 190 edges in Figure 2.

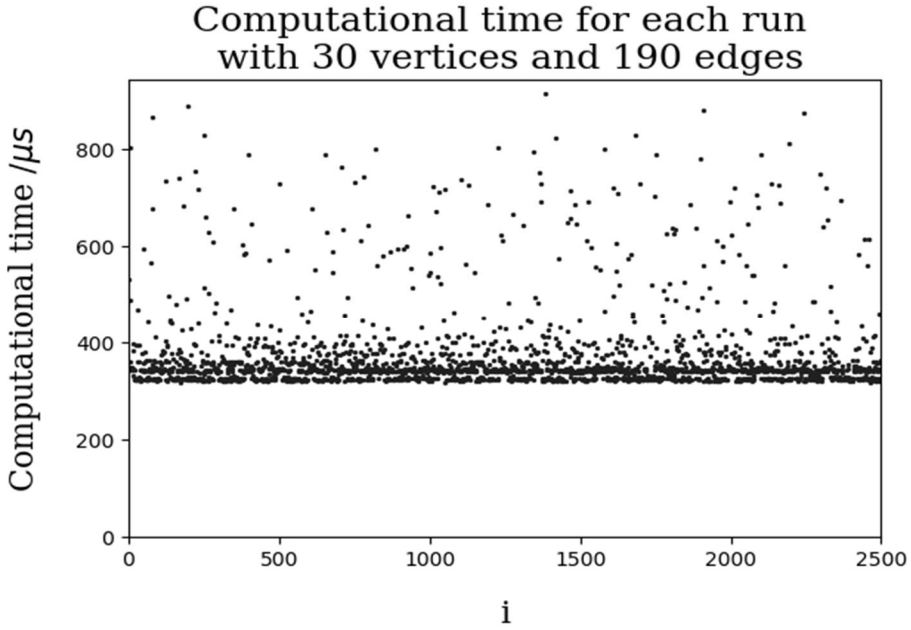


Figure 4: A plot to show how long each run with 30 vertices and 190 edges took to compute.

As seen, the vast majority of points are closely grouped together between around 350 and 400 μs, with even the slowest timings (~ 900 μs) being no greater than about 3x larger than the smallest.

A similar plot can be seen for all of the points which lie on the line, with a fairly small deviation between the largest and smallest timings for any given number of paths. Compare this plot to the one seen below in Figure 5 and 6, which shows the same plot with the data from the 195 edge networks

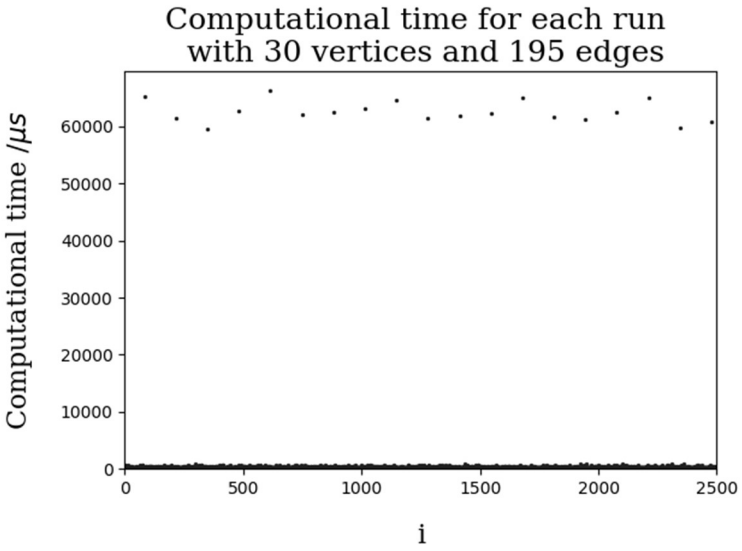


Figure 5: A plot showing computational time for all 30 vertices, 195 edge runs

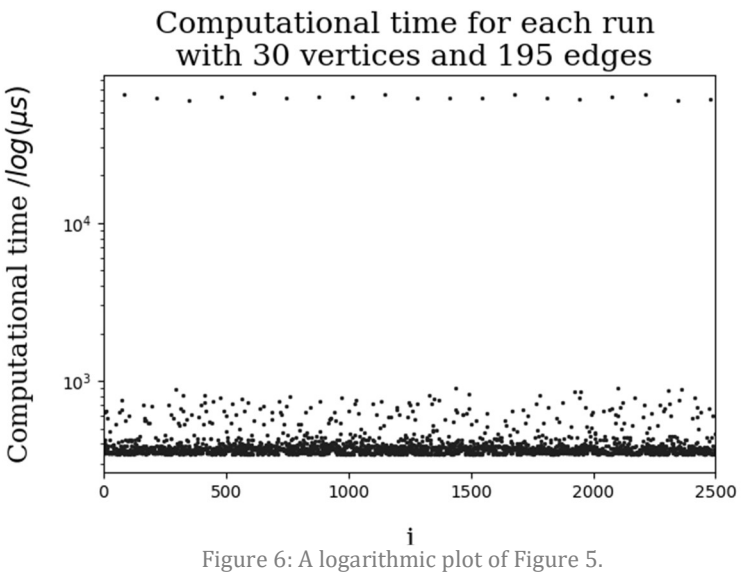


Figure 6: A logarithmic plot of Figure 5.

As seen in figure 5, there are a small number of runs (19) which took ~ 150x longer than the other runs. In Figure 6, it is visible that other than those anomalous 19 runs, all of the remaining runs fit a similar pattern to that we've seen for the 190 edge runs. Most are grouped between 350 and 500 μs, with none going above 1000 μs except the 19 anomalies. A very similar pattern occurs for the 370 edge runs. The highly ordered nature of these anomalies may indicate some sort of array sizing or garbage collection issue, which needs to be looked into further.

Having understood the two anomalous results as bugs, and the initial noise to be due to the high proportion on non-connected networks generated which were randomly generated, we can conclude that there is a very strong and obvious linear correlation between the number of edges in a network and the time Dijkstra's algorithm takes to compute.

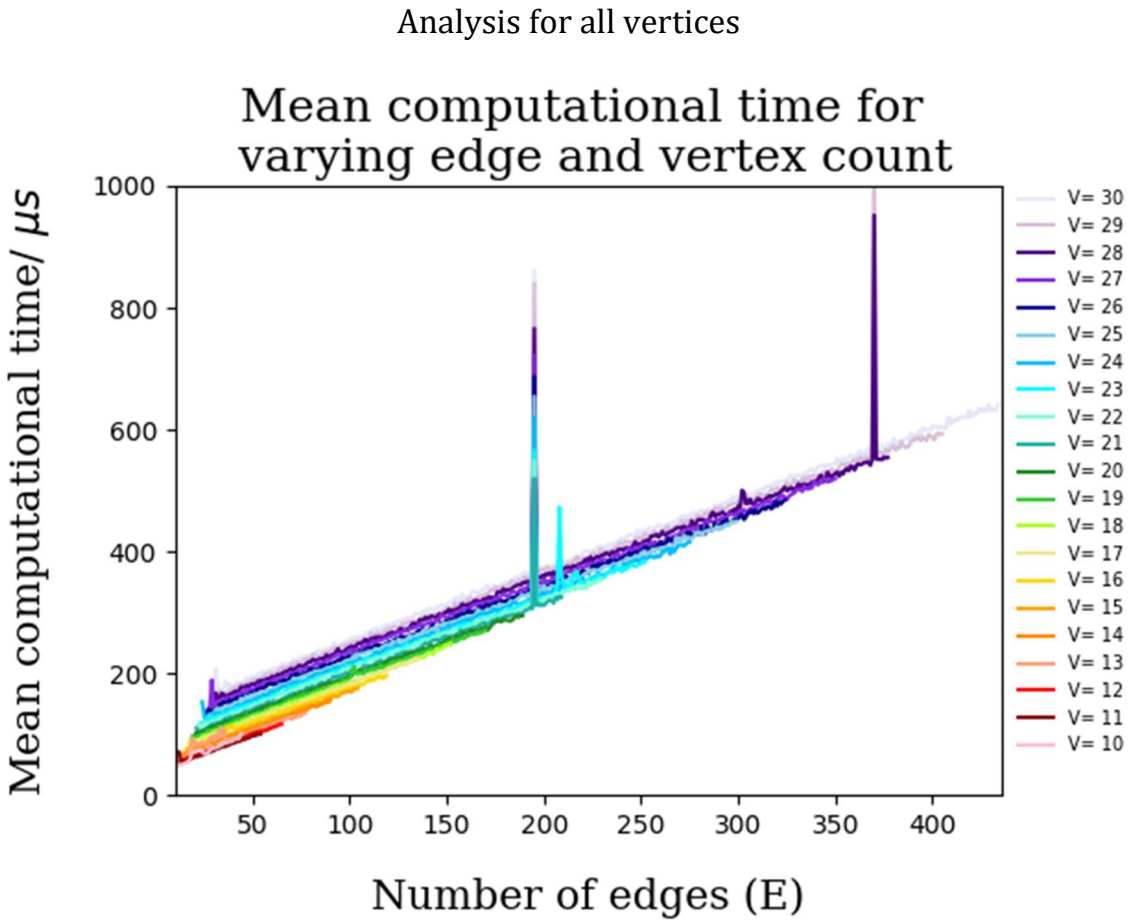


Figure 7: A plot showing how vertex count and edge count affect computational time for Dijkstra's algorithm

Above is Figure 7. Due to the anomalous spikes, it is hard to see much detail at this scale, but it interesting to see that the bugs at 195 and 370 nodes occurs for all vertex counts. This is very useful and interesting, as it could allow us to understand the nature of the bug more. The distributions of the timings for each of these anomalous results look very similar to those in Figures 5 and 6. If we excluded the anomalously large values which occur at both 195 and 370 nodes, we can obtain Figure 8 as seen overleaf.

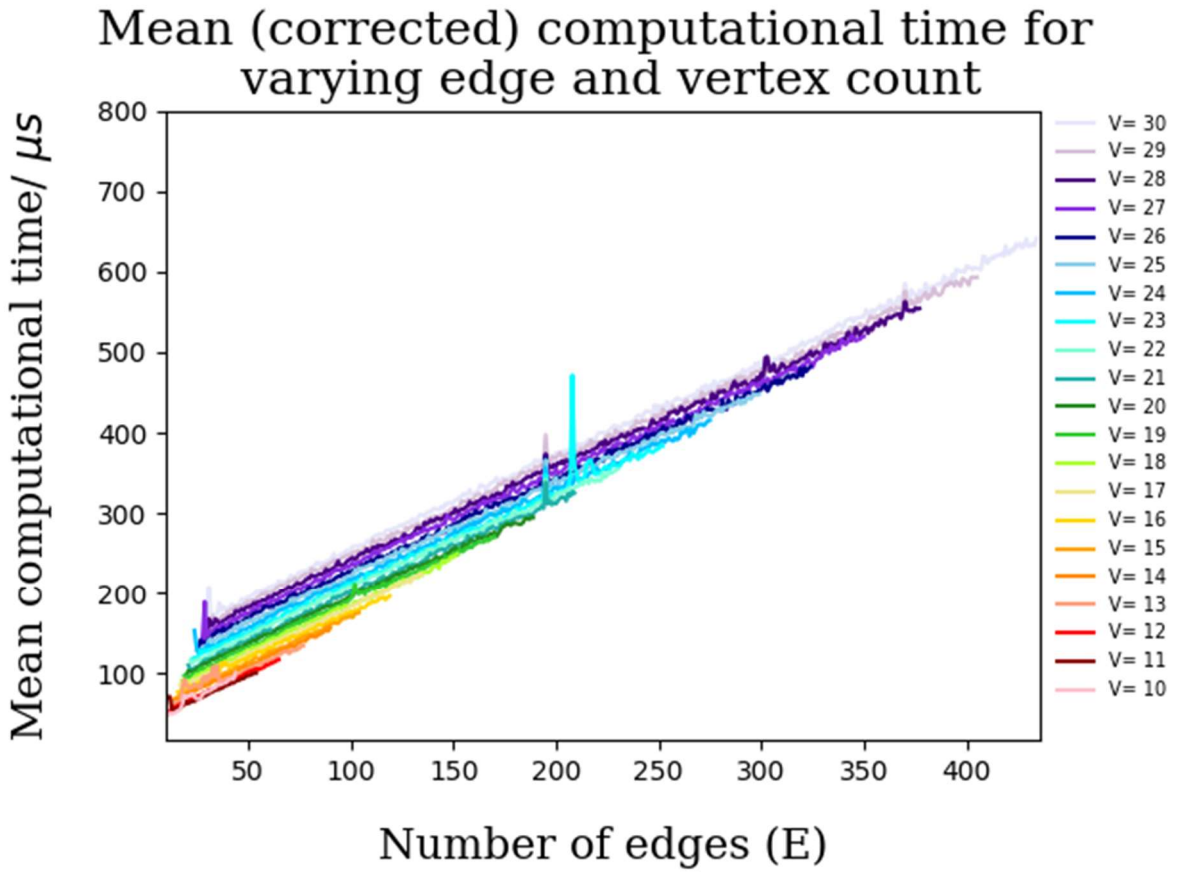


Figure 8: A plot showing how vertex count and edge count affect computational time for Dijkstra's algorithm, with anomalous results excluded from readings where E = 195 or 370.

As we can see above, even excluding the anomalies, the average timings took longer than the surrounding values at E = 195 and E = 370, indicating that these calculations were still, even excluding the results ~150 times greater than the others, harder on average to compute. This could well be a bug with the hardware or firmware being used, and running the script on a different computer may well lead to these anomalies not being present.

Looking at Figure 8, we can see a strong linear correlation exists for each vertex count with the number of edges, as expected from the worst case performance (Equation 1). However, at this scale, it is much harder to judge if at a single path count, the number of vertices fits the expected proportionality of V logV. This is explored more in the next section. A version of Figure 8 which focuses only on E < 190 is available in the annex.

The other most significant anomaly is that seen for V = 23 at E = 208 (the large light blue spike in the middle of the graph). Overleaf we can see the results which were averaged out to get to this value:

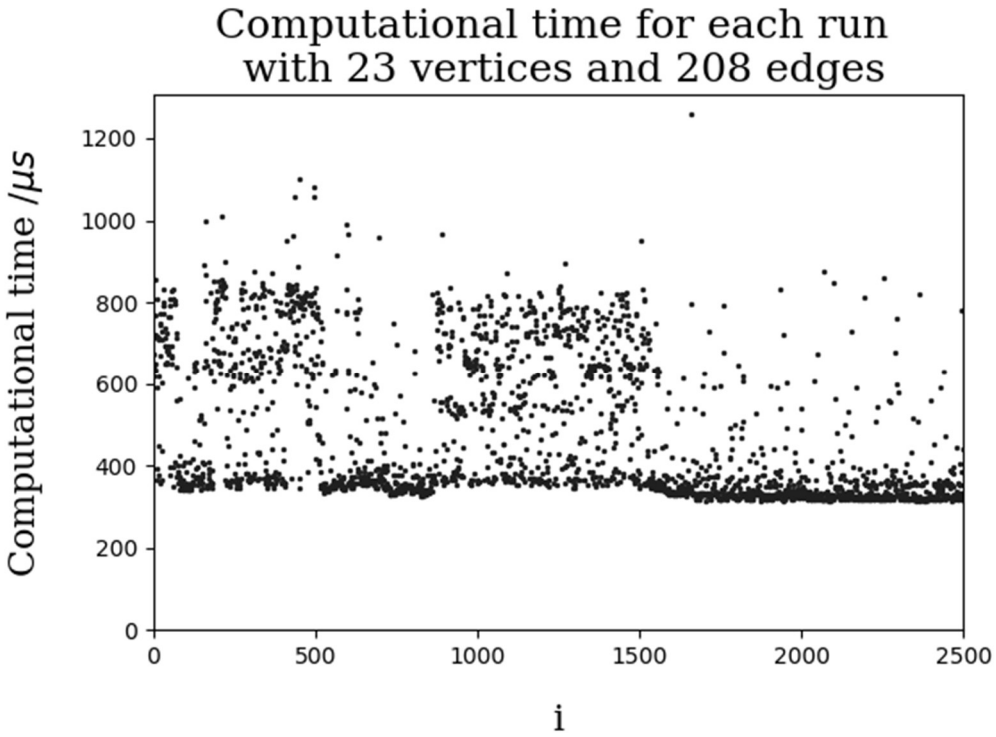


Figure 9: A plot showing the timings of each run with 23 vertices and 208 edges.

As seen when comparing Figure 9 to Figure 4, the first 2000 results are much more random, with no easily visible best run time (no flat line on the bottom) and many more timings in the range of 1.5 to 2 times the lowest timings. This noise seems to dissipate after i = 1700 or so, and the expected distribution returns. Because this large spike is a one-off anomaly, which has a much more chaotic spread than the errors at E = 195 and 370, it would seem this error was caused by external factors to the program which may have temporarily slowed down the processor.

The script was run on my personal laptop over a period of ~ 2 days. I was careful to not interfere with it while it was running, but it is still possible some sort of automated background process ran during this time, such as a virus scan or an update. If one of these were to run, the CPU would become more strained, leaving less processing power available to run the algorithm. This fits Figure 9, as the more chaotic spread could have been caused by a varying processor utilisation during the runtime of some automated program.

## Checking performance with varying vertex count

To check the correlation between vertex count and run time, it must be isolated as a variable. This limits the range of data, as the edge count is not independent of vertex count (e.g. there aren't any networks with 30 vertices and 20 edges, or 10 vertices and 300 edges). To minimize the impact of this, and maximize the available data, a path count of 90 was chosen to be examined, as most vertex counts have a network at this path count (excluding V = 10-13), and it is not small enough that there would be significant errors due to lack of connected networks for the higher vertex count networks. Overleaf, in Figures 10 and 11, mean runtime is plotted against vertex count or V logV to show which is better correlated, and if the expected trend of V logV was observed in this experiment.

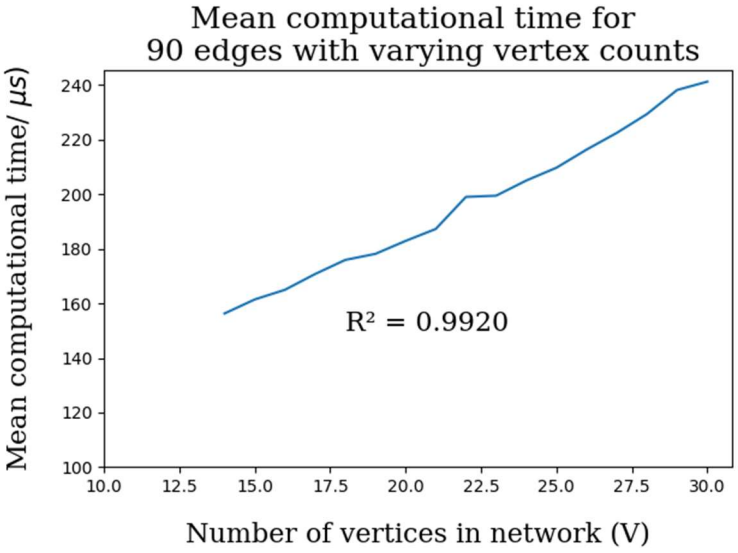


Figure 10: A plot showing how computation time correlates with V

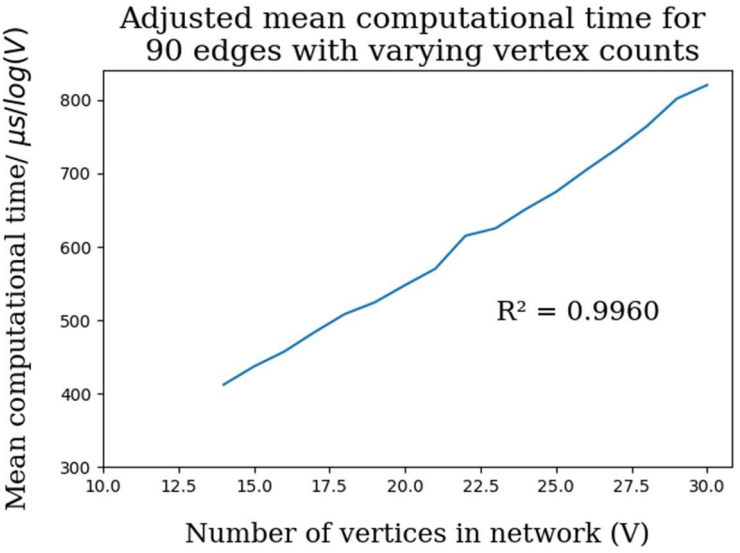


Figure 11: A plot showing how computation time / log V correlates with V

As seen above, both the normal linear correlation (Figure 10) and the V logV correlation (Figure 11) have very strong agreement with the results ($R^2$ of 0.992 and 0.996 respectively). This makes it very hard to positively identify that a run time of V logV instead of just V was observed. This mainly is due to the small range of vertices which could be tested. Improvements to this aspect of the experiment are discussed in the conclusion section.

# Conclusion

Overall, this experiment successfully demonstrated that Dijkstra's algorithm runs with time O(E) for a constant number of vertices, but with varying vertex count, this experiment was not able to discern between a run time of O(V) and the expected O(V logV) , as both trends agreed with the obtained data with $R^2 > 0.99$. The decision to run the algorithm only between V = 10 and 30 was made in an effort to balance the amount and accuracy of collected data and the time the program would take to run.

In future, if an experiment were to be designed with the aim of positively identifying the expected O(E+ V logV) run time, it would make sense to test both independently in order to save time. One script could generate data for V = 10 to 50 for all edge counts, to test that the run time is proportional to the number of edges, and a second script could run V = 40 to 200 with a fixed edge count (e.g 800) to test if the run time is proportional to V logV.

The anomalous results at E = 195 and 370 are still not well understood, and could be investigated further using a profiler which shows where time is being spent on the script.

To reduce the randomness seen at the start of Figure 2 due to the increased number of non-connected networks generated, the script could be altered to repeat the protocol until 2500 connected networks are generated. However, this would cause a drastic increase in the time taken to run the script, as networks are very rarely connected when generated randomly with the minimum possible edge count, and so the script would have to run thousands of times to get a single valid network. Alternatively, the script could be rewritten such that only connected graphs are generated by limiting the way edges are assigned. This would affect the 'randomness' of the results overall and may cause biased timings, as the networks may tend to become more ordered as a result.

Despite these problems, overall, I believe this project has been a success in the two main aims: The script was able to generate random networks with any chosen number of vertices and edges, and the timing data was able to show some insight into how the run time of Dijkstra's algorithm varies with the nature of the network.

# Annex

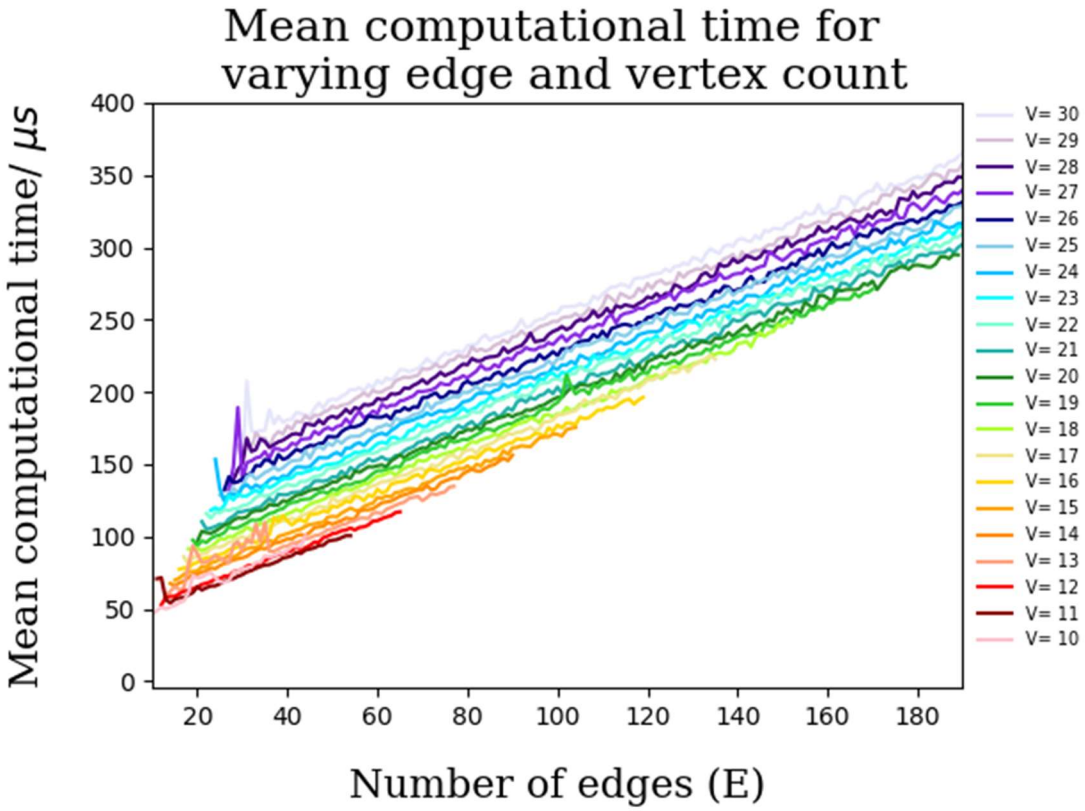


Figure 12: A version of Figure 8 which shows only the results with E < 190

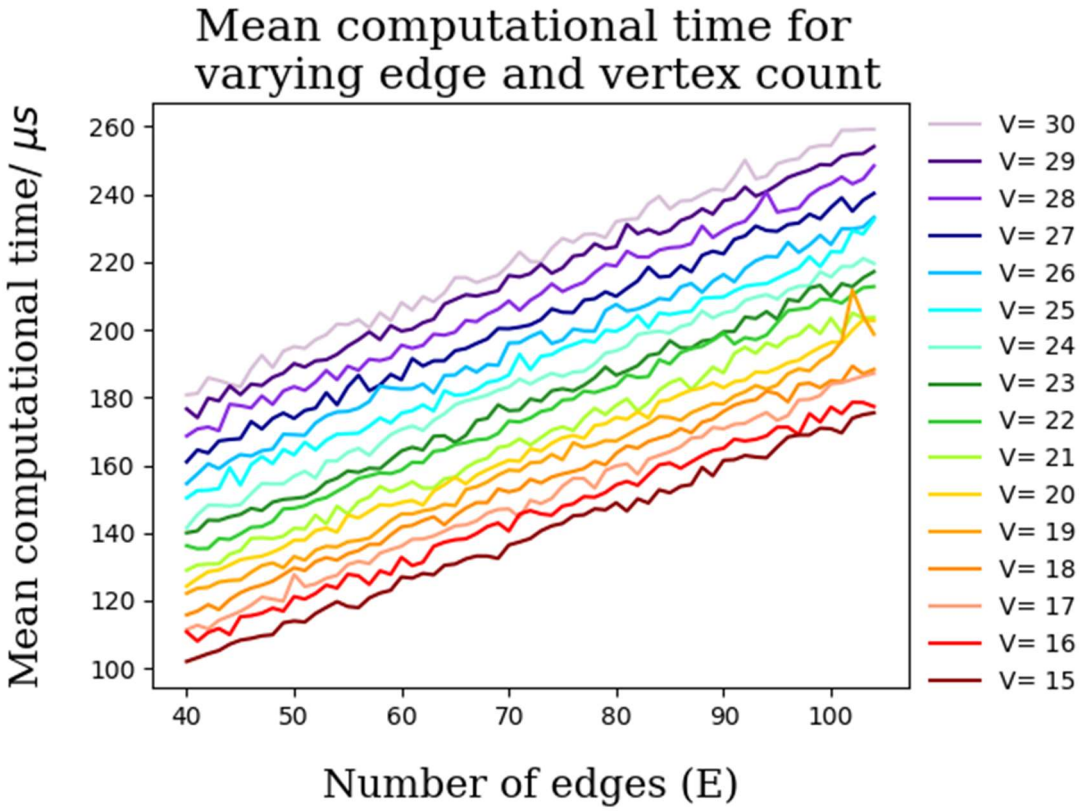


Figure 13: A plot isolating V from 15 to 30 between E = 40 to 100 to show the linear trends in more detail.