# Extended Essay

# Computer Science

## Topic: Comparing greedy (Dijkstra’s algorithm) and dynamic (Bellman Ford’s algorithm) algorithms for solving shortest path optimization problem given a graph with positive edges

**Research Question:** How does the efficiency of Dijkstra algorithm compare to that of Bellman-Ford’s, for shortest path optimization in terms of execution time as the number of nodes in a graph changes?

Word Count: 3997 words

## Table of Contents

## 1. INTRODUCTION

In the current digital world, technology is often used to solve problems of many types. Problems can have many solutions, but optimization problems focus on searching for the best solution out of many possible solutions.[1] There are a multitude of real-life applications to solving optimization problems, for example it is used in GPS systems like google maps, network routing and recommendations on social networks. One way technology solves optimization problems is by using algorithms. Common methods used to formulate these algorithms are based on the greedy method and the dynamic programming method. The greedy method works by taking the most optimum next move, without inspecting its future outcomes. This means it can sometimes produce substandard solutions, but as it is based on a straightforward concept, it is easy to implement and generally takes less time to execute. Dynamic programming is based on finding and executing every possible solution and choosing the best one. This means it usually requires more memory and time to execute than the greedy method as it may perform more calculations than the greedy method, but its solution is typically confirmed.[2] When I was exposed to both these concepts, I wanted to test them on graph theory as it has numerous real-world applications. It is used in building communication networks, road networks and more. The most used greedy algorithm to solve graph theory related problems is the Dijkstra's algorithm and the most used dynamic algorithm to solve graph theory related problems is the Bellman Ford algorithm. My exploration will be concentrated on comparing both Bellman Ford algorithm and Dijkstra's algorithm by

---

[1] Black, Paul, "optimization problem", xlinux, January 6, 2021, https://xlinux.nist.gov/dads/HTML/optimization.html
[2] Coderucks, "Dynamic Programming Vs Greedy Algorithm", coderucks, January 24, 2021, "https://codecrucks.com/dynamic-programming-vs-greedy-algorithm/

identifying how a change in the number of nodes affects the execution time of both algorithms. In addition, when used in the real world, the algorithms are rarely used with negative edge weights. Furthermore, both algorithms have the possibility of giving an incorrect answer when used with negative edge weights. For the above reasons, the exploration will only focus on positive edge weights and how increasing the nodes in a graph with positive edge weights will affect the time complexity of both Dijkstra's and Bellman Ford's algorithm.

## 2. BACKGROUND INFORMATION: [3]

### 2.1 - Graphs

A graph is a type of data structure that stores information in the form of nodes and edges. [4] Edges show how each of the nodes are related (i.e. they connect 2 nodes and may have a value), and nodes can contain data. Graph 'G' can be represented as follows (Refer to Figure 1):

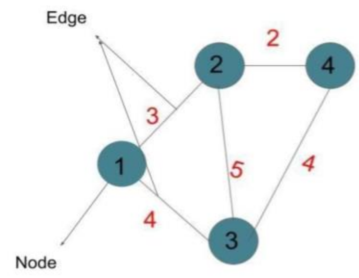



**Figure 1:** Edges and Nodes

$$G : (V, E)$$

G is the name of the graph. The set of vertices is shown by V and set of edges is shown by E. [5]

Graphs can either be directed or undirected, and weighted or unweighted. Directed graphs are graphs in which the edge points to a particular direction. This means that when we traverse between two nodes, we can only move in one direction but not the opposite. Undirected graphs are graphs where we can traverse in both directions.[6] Weighted graphs are graphs whose edges have specific values (also known

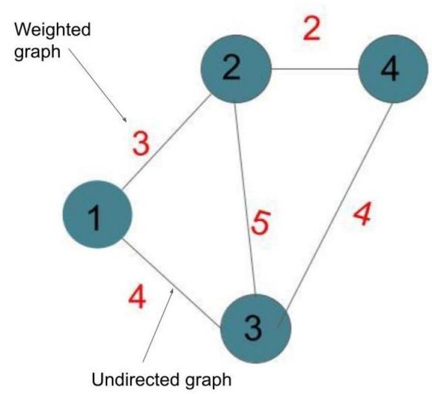
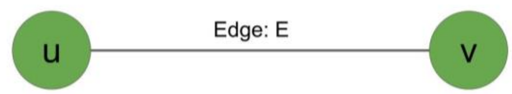



**Figure 2:** Undirected and Weighted graph

[3] Samah W.G. AbuSalim et al 2020 IOP Conf. Ser.: Mater. Sci. Eng. 917 012077
[4] Bhatta, Ranjit,"Graph Data Structure", programiz, May 6 2022, https://www.programiz.com/dsa/graph
[5] Sun, Timothy, "Graphs", Columbia, May 6 2022, http://www.columbia.edu/~cs2035/courses/ieor6614.S11/graph.pdf
[6] Little, Jack, "Directed and Undirected Graphs", mathworks/MATLAB, May 7 2022, https://www.mathworks.com/help/matlab/math/directed-and-undirected-graphs.html

as weights) and unweighted graphs are graphs whose edges do not have values.[7] Shortest path optimization problems apply to only weighted graphs as they calculate the distance between two nodes based on the "weights" of the edges. In the following exploration I will be using only undirected (Notice that the arrows do not point in a specific direction in Figure 2) and weighted graphs (notice the numbers on the edges in Figure 2). To represent a single edge of a graph, we use the following:

$$E : (u, v)$$




u is the first vertex and v is the second vertex, E is the edge connecting both the first and the second edge.

**Figure 3:** Edge Representation

Note that in my exploration, (u,v) equals (v,u) for all edges as in an undirected graph, when you traverse in any direction, the weight remains the same.[8] Weighted edges can be represented as follows:

$$E: (u, v) = w \ OR \ w(u, v)$$

"E : (u,v)" represents an edge, and w represents the weight of that edge, in the case of figure 4, w(u,v) = 5.

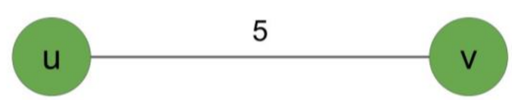



**Figure 4:** Representation of weighted graphs

The concept of a source node is very important in shortest path optimization problems. A source node is the starting node, or the node we want to calculate the distance from. For example, in Figure 4, if we would like to find the distance from u to v, u is the source node as we start taking the distance from u. Additionally in Figure 1, if we would like to find the distance from 1 to 4, 1 would be the source node.

[7] McQuain, "Data Structures And Algorithms", VirginiaTech, May 7 2022, https://courses.cs.vt.edu/~cs3114/Fall10/Notes/T22.WeightedGraphs.pdf

Similarly, another important concept is the destination node. It's the node we want to calculate the shortest path to. If we take the same example, 4 would be the destination node.

Now that we have seen how a graph can be represented on paper, how can we represent it on screen? There are multiple methods for representing a graph on a screen, but the one I will be using is the adjacency matrix. It uses a 2D array of size V x V (V being the number of vertices in the graph). [9] Given a 2D array named 'A':

$$A[i][j] = w$$

$$Edge\ from\ vertex\ i\ to\ vertex\ j\ is\ equal\ to\ weight\ w$$

In this equation, i represents the first vertex, and j represents the second vertex, if an edge exists between them, we get a weight 'w' as the outcome, which corresponds to the weight of the edge connect vertex i and vertex j (Note, for undirected graph A[i][j] = A[j][i]).

### 2.2 - Relaxation

The relaxation technique is used in both Dijkstra's algorithm and Bellman Ford's algorithm, so it may be useful to explain what relaxation is. Though relaxation on its own cannot produce solutions, it is often implemented in various algorithms in different ways to help find solutions.

Relaxation works on updating estimates of the shortest path to each node. In both algorithms, the first step is to initialise the "estimates" of all nodes as infinity except the

[9] Wormald, Nicholas, "Graphs", University of Western Australia, May 22, 2022, https://users.monash.edu/~lloyd/tildeAlgDS/Graph/#:~:text=The%20adjacency%20matrix%20of%20a,infinity%22%2C%20indicates%20this%20fact.&text=Adjacency%20Matrix%20of%20Weighted%20Directed%20Graph.&text=Adjacency%20Matrix%20of%20Weighted%20Undirected%20Graph.

source node. Now we can introduce the relaxation statement and explain it using an example.

Relaxation states (Note that d represents distance which corresponds to the current estimate on the node) - [10]

$$if\ d[v] > d[u] + w(u,v)$$

$$then\ d[v] = d[u] + w(u,v)$$

The statement says that if the distance of vertex v is greater than the distance of vertex u + the weight of the edge u to v, then the distance of v is equal to the distance of u + the weight of the edge u to v. Let's use an example to better understand this. Suppose we have a graph as in figure 5, with u as the source vertex. The source vertex always has a distance estimate of 0 since we start from the source (Estimates are shown above the node typically). This is because we do not need to traverse through any edges to reach the source. Vertex v's estimate is updated to infinity. To find the shortest path to v based on the relaxation technique, if the distance of v (∞) is greater than the distance of u (0) + the w(u,v) (8), (∞ is greater than 0+8 = 8, so the next statement is executed), the distance of v is updated to d[u] + w(u,v) which is equal to 8.

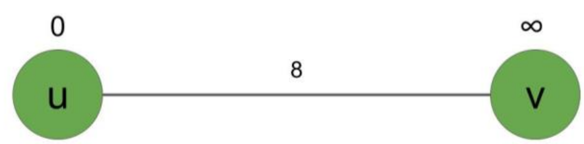



**Figure 5:** Relaxation example

In the end, v's estimate of infinity is updated to 8, which is the new estimate of the shortest path. Note that if the distance of v was instead smaller than the distance of u + w(u,v), the estimate on the node would remain the same.

[10] Jaiswal, Sonoo, "Relaxation", javaTpoint, May 23 2022, https://www.javatpoint.com/relaxation#:~:text=The%20single%20%2D%20source%20shortest%20paths,equivalent%20the%20shortest%20%2D%20path%20weight.

### 2.3 - Dijkstra's Algorithm

Dijkstra's algorithm was created according to the greedy programming technique. This means the best condition is chosen at each step, without considering its future consequences.[11] Let's understand this algorithm on a small graph. In figure 6, we will try to find the distance between vertex 1 and vertex 2. We observe that vertex 1 is the source vertex as its distance is estimated to 0. The other two vertices' estimates are updated to infinity. Now we must update the estimates according to the direct edge paths from 1. 1 has a direct edge to vertex 2 and vertex 3. This means we must relax (use relaxation technique) the nodes from 1 to 2 and 1 to 3. This updates the estimate of 2 to 8 and 3 to 5. Once the estimates are updated, this means vertex 1 is fully "explored" (this is because we have updated estimates according to all direct edges of vertex 1), and hence according to Dijkstra's algorithm 0 is 1s shortest path. In the next step according to Dijkstra's algorithm, we must take the next node with the lowest estimate. In this case, 3 has the lowest estimate, so we start from 3. Since 1s estimate is already verified, we don't have to relax 1, but we have to relax vertex 2 as its estimate is not verified. Upon relaxing vertex 2, its estimate is updated to 7. Now since all the edges from 3 have been discovered, 3 is now "explored" and its shortest path is 5. Since only one node is left (when only one node is left, it is automatically explored), the graph automatically becomes explored, which means the shortest path from 1 to 2 according to Dijkstra's algorithm is 7.

| Graph 1 | Graph 2 | Graph 3 |
| --- | --- | --- |

[11] Jain, Sandeep,"Dijkstra's Shortest Path Algorithm",GeeksForGeeks, May 25 2022, https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

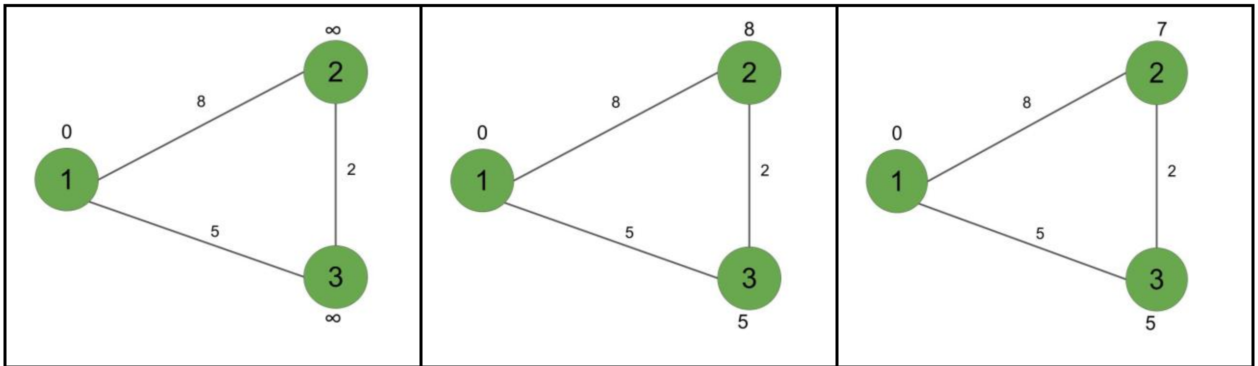



**Figure 6:** Dijkstra's Algorithm

The same rules can be applied on a larger graph with more vertices and edges. We would just have to relax the vertices more.

## 2.4 - Bellman Ford's Algorithm

Bellman Ford's algorithm was created according to the dynamic programming technique. Bellman Ford's algorithm says that you must find all solutions and choose the best solution. We can find all the solutions by relaxing all the edges multiple times.[12] We must relax all the edges V-1 times, where V is the number of vertices. This will ensure all solutions are found. Let's test this example on the graph in figure 7. Note that 1 is taken as the source vertex and 2 is the vertex to reach.

First let's make a list of all the edges in the graph -

$$(1,2),(2,1),(2,3),(3,2),(1,3),(3,1)$$

Now we must relax all these edges 3-1 = 2 times (as there are 3 vertices). When we relax (1, 2) we get the weight on 2 as 5, when we relax (2, 1) the weight of 1 remains the same, when we relax (2, 3) the weight of 3 changes to 6, when we relax (3, 2) the weight of 2 remains the same, when we relax (1, 3) the weight of 3 changes to 3, and when we relax

[12] Bhatta, Ranjit, "Bellman Ford's Algorithm", Programiz, July 9 2022, https://www.programiz.com/dsa/bellman-ford-algorithm

(3, 1) the weight of 1 remains the same. Now we have completed relaxing all the nodes once, we must repeat this process for one last time since there are 3 vertices. The outcomes are shown in figure 8.

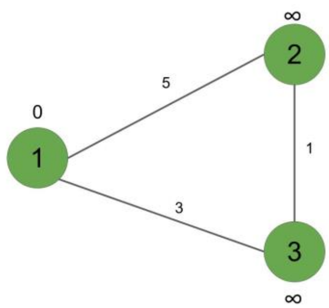
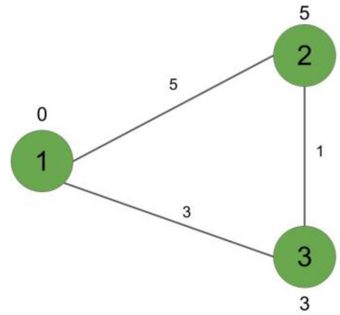
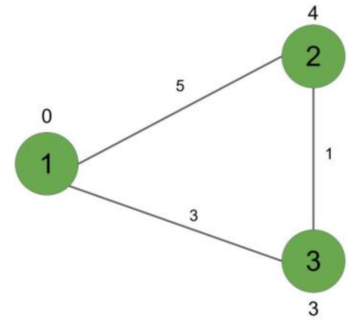
| Graph 1 – Initial graph | Graph 2 – First relaxation | Graph 3 – Second relaxation |
|---|---|---|
|  |  |  |

**Figure 7:** Bellman Ford's Algorithm

Now we have found the shortest path to vertex 2 from vertex 1, which according to Bellman Ford's algorithm is 4. The same steps would have to be followed even for a larger graph.

### 2.5 – Number Of Edges

As the number of nodes increases, there are more edges that could be created in a given graph. When each node is connected to every other node in a graph system, it is known as a complete graph. Complete graph has the maximum number of edges for a graph with a given number of nodes. For my exploration, I will not be using complete graphs, but the maximum number of edges that could be created for a given number of nodes is important. It can be calculated with the formula (where n is the number of nodes) – [13]

$$Max\ number\ of\ edges\ that\ could\ be\ created = \frac{n(n-1)}{2}$$

[13] Pandey, Avinash, "Complete Graph", d3gt, July 15 2022, https://d3gt.com/unit.html?complete-graph

## 3. METHODOLOGY:

### 3.1 - Variables

**Independent Variable – *Number of Nodes*** - To find the faster algorithm of the two, number of nodes is taken as the independent variable as in a graph, changing the number of nodes has a high effect on execution time. Number of nodes is also taken such that there is a varying change in the increase in the number of nodes. This allows us to examine the effect of differing rates of change in the number of nodes for both algorithms.

**Dependent Variable** - ***Execution Time*** - Execution time was measured using the function ".nanoTime()" in Java. Each data set was run multiple times (10 times) in 2 different IDEs to reduce errors related to inefficiencies in IDE. The first execution was deleted from each IDE execution as the time taken for the Java virtual machine to boot may make the program slower. The time is started exactly once the array is inputted into the method. The methods for each algorithm were written to ensure maximum efficiency. Unnecessary data copies were avoided, and string buffer was used rather than multiple "system.out" to ensure computation is focused only on the execution of the algorithms. The time taken by both algorithms was outputted only once towards the end.

**Control Variable** - ***CPU and RAM Usage*** - The same device was used for executing all algorithms. All applications other than the necessary applications and the IDE were kept closed to ensure maximum usage of CPU and RAM. Both IDEs were used evenly along with several executions. This ensures that fluctuations in use of system resources are eliminated along with IDE related overheads.

### 3.2 – Matrices (2D Arrays) Used

To represent the graphs in a 2D Array, a program was created to generate a matrix such that the graph created is unweighted. Additionally, since using a complete graph would favour Dijkstra's algorithm (as more nodes means much more relaxation for Bellman Ford's algorithm), the program was given a 50% probability to make an edge at any possible node to node connection. Lastly, care was taken to ensure that the graphs were not disconnected (a graph in which there is no connection between any two nodes), if the graph generated was disconnected, a recursive algorithm was created to generate another graph until a connected graph was created. Lastly, the source and destination node were randomized, but were ensured that they were not the same node.

### 3.3 – Experimental Procedure

Nodes from the following values were taken –

| Number of Nodes | Maximum Number Of Edges Possible |
|---|---|
| 10 | 45 |
| 13 | 78 |
| 16 | 120 |
| 19 | 171 |
| 22 | 231 |
| 25 | 300 |
| 30 | 435 |
| 35 | 595 |
| 40 | 780 |
| 45 | 990 |
| 50 | 1,225 |

| | |
|---|---|
| 60 | 1,770 |
| 80 | 3,160 |
| 100 | 4,950 |
| 200 | 19,900 |
| 300 | 44,850 |
| 400 | 79,800 |
| 500 | 124,750 |
| 600 | 179,700 |
| 800 | 319,600 |
| 1000 | 499,500 |

**Table 1: Number of Nodes and Maximum Number Of Edges**

Each number of nodes was generated 22 different connected graphs. The 22 graphs were spread equally among 2 different IDE's and executed with both Dijkstra's and Bellman Ford's algorithm. The first two executions in each IDE were not considered. The execution times outputted by each algorithm were noted.

## 4. HYPOTHESIS:

As the number of nodes increases, the time of execution for the Bellman Ford algorithm increases at a faster rate than Dijkstra's algorithm. The time taken by the Bellman Ford algorithm will be larger than Dijkstra's algorithm for all data sets, but Bellman Ford's algorithm will have more stable results.

## 5. THE EXPERIMENTAL RESULTS

### 5.1 - Dijkstra's Algorithm

#### 5.1.1 – Tabular Data

The table below shows the processed results of running Dijkstra's algorithm on a varying number of nodes.

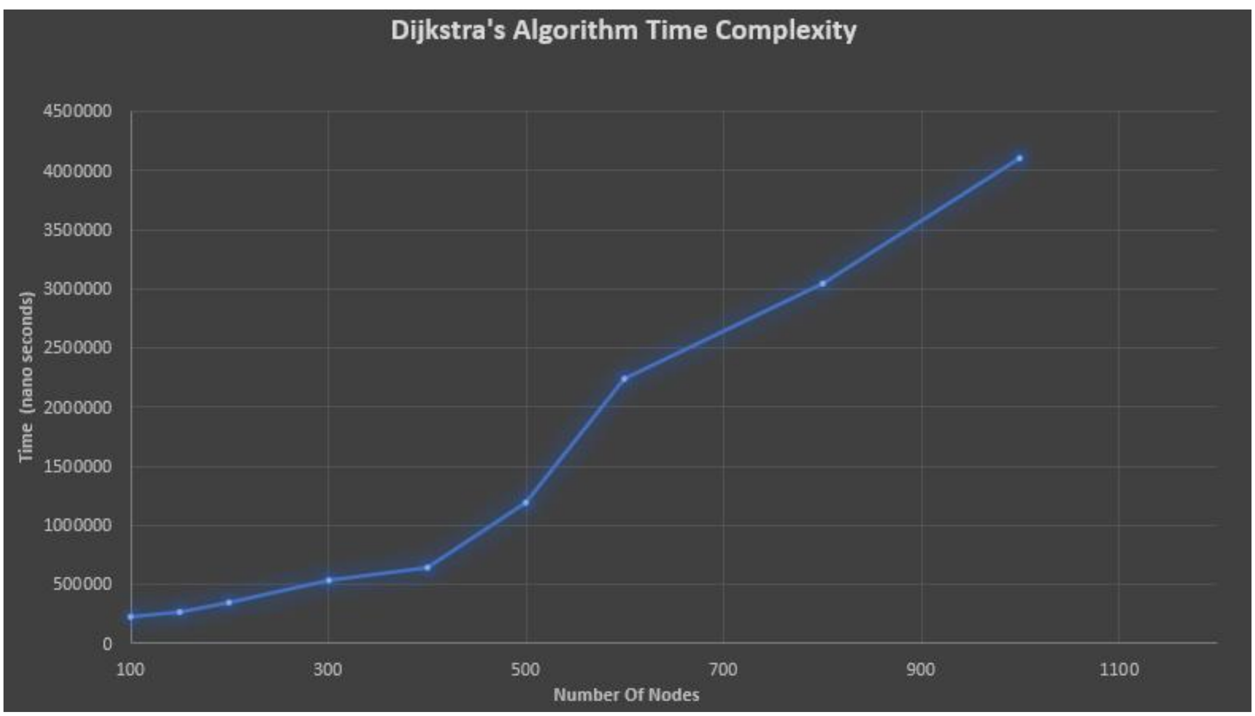
| Number Of Nodes | Average Time Taken (seconds) | Relative Standard Deviation (Percentage) |
|---|---|---|
| 10 | 19,253 | 44.93411 |
| 13 | 22,678 | 36.6638 |
| 16 | 34,852 | 65.8444 |
| 19 | 35,505 | 66.01341 |
| 22 | 40,615 | 33.42257 |
| 25 | 51,357 | 40.27035 |
| 30 | 52,510 | 51.47192 |
| 35 | 89,021 | 30.52561 |
| 40 | 99,694 | 63.56813 |
| 45 | 115,526 | 67.61167 |
| 50 | 138,226 | 69.71442 |
| 60 | 147,016 | 77.00876 |
| 80 | 141,515 | 67.17754 |
| 100 | 227,001 | 83.56837 |
| 150 | 271,200 | 52.41632 |
| 200 | 341,438 | 211.8987 |

| | | |
|---|---|---|
| 300 | 529,384 | 43.94714 |
| 400 | 642,989 | 54.44302 |
| 500 | 1,189,500 | 196.0965 |
| 600 | 2,243,100 | 64.78164 |
| 800 | 3,049,115 | 149.4159 |
| 1000 | 4,107,673 | 76.89447 |

**Table 2: Dijkstra's Algorithm**

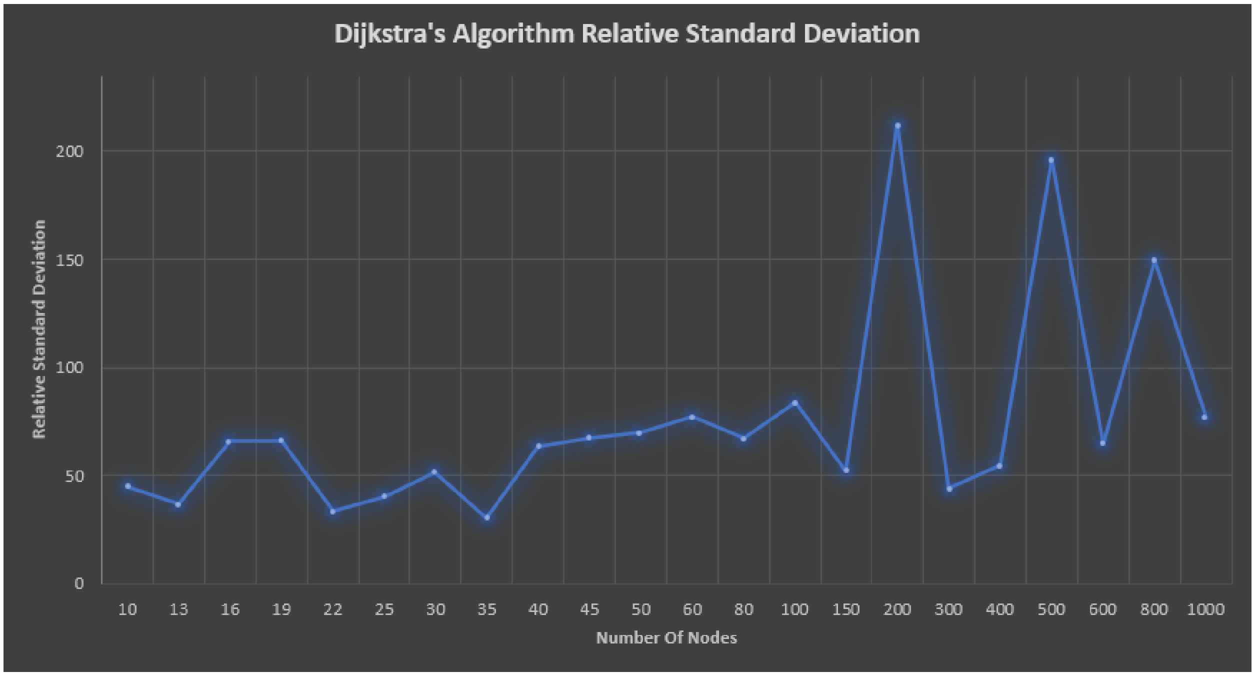
### 5.1.2 – Graphical Data

The following graph shows the number of nodes against time taken for Dijkstra's algorithm to complete finding the shortest path. Note that the graphs start from 100 nodes as including the other values would result in data points being too close to each other (as their values are small compared to the large scale).




**Graph 1: Dijkstra's Algorithm Time Complexity**

The following graph displays the relative standard deviation of each of the number of nodes, when executing Dijkstra's algorithm.




**Graph 2: Dijkstra's Algorithm Relative Standard Deviation**

### 5.1.3 – Data Analysis

We observe that as the number of nodes increases, the time taken by the algorithm also increases. This is likely because the algorithm now must traverse more nodes than before and relax more edges than before. We also observe that the relative standard deviation value is high. This is because the number of edges were randomized. Hence the possible combinations to reach the destination from the source may fluctuate greatly. Additionally, since the algorithm works by choosing the best possible "next solution", the number of times this must be done can vary greatly depending on the number of nodes. This may bring a broad range of execution times, which means that this algorithm takes an uneven amount of time when calculating the distance between 2 nodes.

### 5.2 – Bellman Ford's Algorithm

#### 5.2.1 – Tabular Data

The table below shows the processed results of running the Bellman Ford's algorithm on a varying number of nodes.

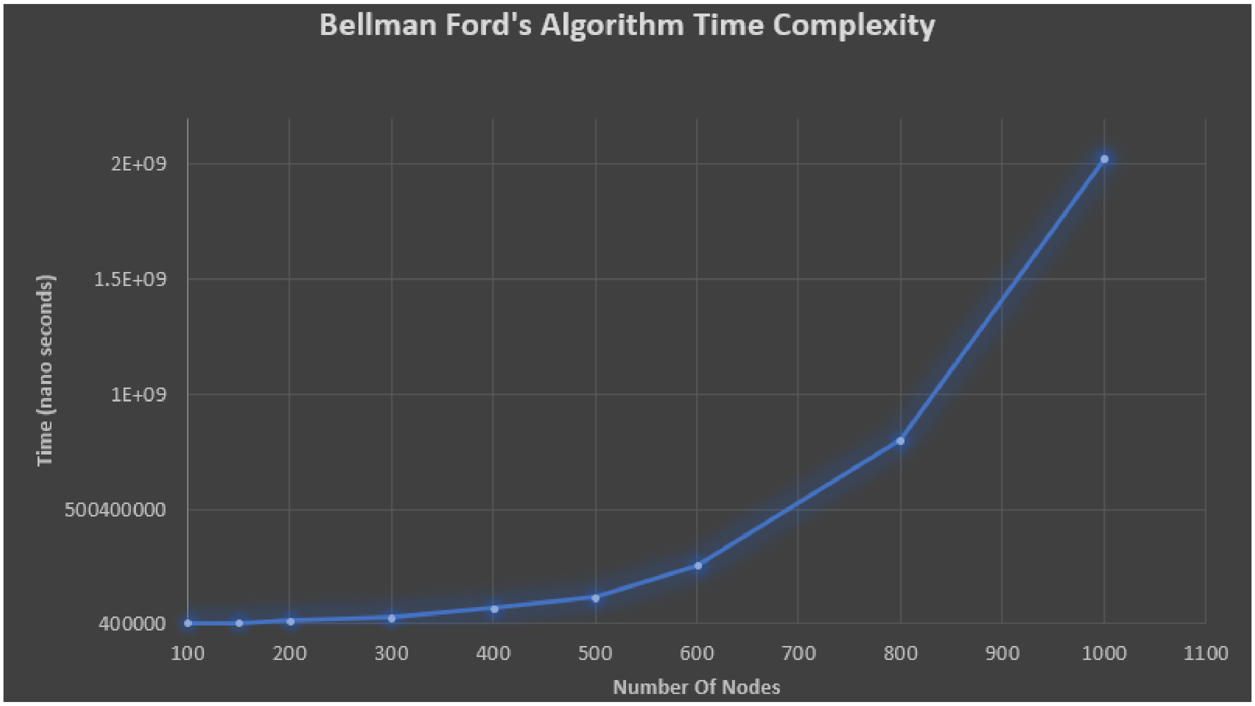
| **Number Of Nodes** | **Average Time Taken (seconds)** | **Relative Standard Deviation** |
|---|---|---|
| 10 | 35,800 | 20.99313 |
| 13 | 62,742 | 23.5367 |
| 16 | 103,693 | 32.55637 |
| 19 | 141,573 | 47.94875 |
| 22 | 156,831 | 47.44195 |
| 25 | 173,728 | 69.93443 |
| 30 | 241,873 | 64.25956 |
| 35 | 259,147 | 89.29785 |
| 40 | 287,510 | 71.53663 |
| 45 | 298,305 | 18.93875 |
| 50 | 476,047 | 37.08386 |
| 60 | 584,622 | 37.56087 |
| 80 | 3,451,197 | 227.7628 |
| 100 | 4,010,942 | 130.2899 |
| 150 | 5,238,189 | 184.4197 |
| 200 | 12,858,721 | 48.7599 |
| 300 | 29,544,352 | 44.11812 |

| | | |
|---|---|---|
| 400 | 67,634,368 | 11.73972 |
| 500 | 117,937,500 | 11.1074 |
| 600 | 256,789,463 | 8.223952 |
| 800 | 799,591,105 | 11.59528 |
| 1000 | 2,024,936,721 | 12.65869 |

**Table 3: Bellman Ford's Algorithm**

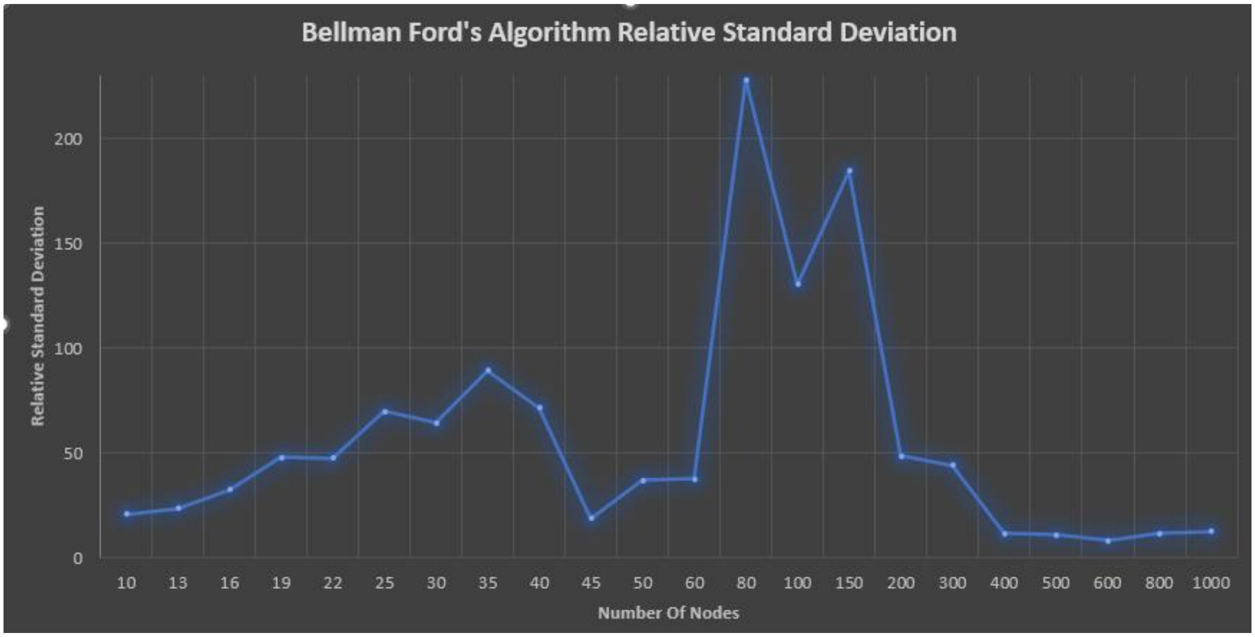
### 5.2.2 – Graphical Data

The following graph shows the number of nodes against time taken for Bellman Ford's algorithm to complete finding the shortest path. The graph starts from the value of 100 nodes for the same reason as Dijkstra's algorithm.




**Graph 3: Bellman Ford's Algorithm Time Complexity**

The following graph displays the relative standard deviation of each of the number of nodes, when executing Bellman Ford's Algorithm.




**Graph 4: Bellman Ford's Algorithm Relative Standard Deviation**

### 5.2.3 – Data Analysis

As the number of nodes increases, the time taken by the algorithm also increases. This happens because when the number of nodes increases, the number of edges that could be created also increases. This leads to a higher number of edges being created as the graph generator has a 50% probability to create an edge for every given connection. This means the algorithm has to relax more edges when the number of nodes increases, and hence longer time taken as the number of nodes increases. Though it takes long time to execute the algorithm, the deviation of values is low as the algorithm always relaxes all nodes of the graph. This means that though the number of edges is randomized, the algorithm already tries out all possible combinations. So even a change in the number of edges does not drastically increase the number of edges that need to be relaxed. Also note that for a given number of nodes, the number of times the edges must be relaxed remains the same as the number of vertices remains the same. This could lead to a lower deviation value.

## 5.3 – Comparative Analysis

### 5.3.1 – Tabular Data

The table below shows the results of running the Dijkstra's algorithm and Bellman Ford's algorithm on a varying number of nodes.

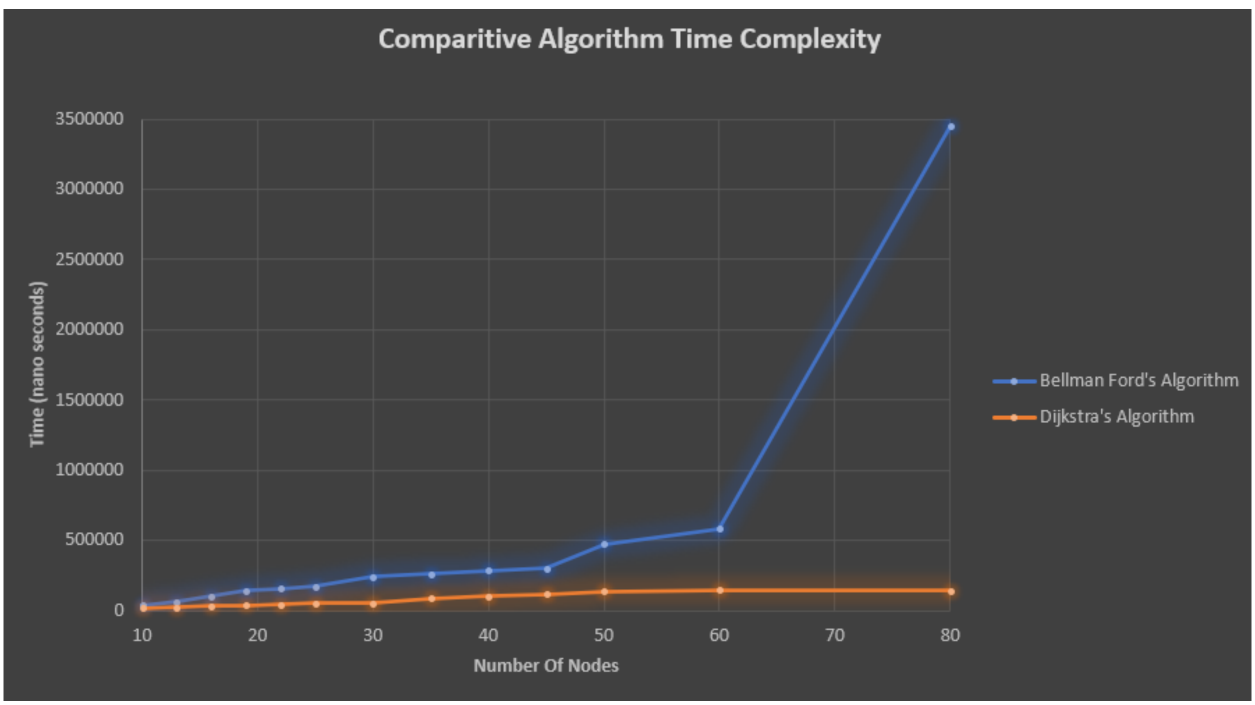
| Dijkstra's Algorithm | | | Bellman Ford's Algorithm | | |
|---|---|---|---|---|---|
| **Number Of Nodes** | **Average Time Taken (seconds)** | **Relative Standard Deviation** | **Number Of Nodes** | **Average Time Taken (seconds)** | **Relative Standard Deviation** |
| 10 | 19,253 | 44.93411 | 10 | 35,800 | 20.99313 |
| 13 | 22,678 | 36.6638 | 13 | 62,742 | 23.5367 |
| 16 | 34,852 | 65.8444 | 16 | 103,693 | 32.55637 |
| 19 | 35,505 | 66.01341 | 19 | 141,573 | 47.94875 |
| 22 | 40,615 | 33.42257 | 22 | 156,831 | 47.44195 |
| 25 | 51,357 | 40.27035 | 25 | 173,728 | 69.93443 |
| 30 | 52,510 | 51.47192 | 30 | 241,873 | 64.25956 |
| 35 | 89,021 | 30.52561 | 35 | 259,147 | 89.29785 |
| 40 | 99,694 | 63.56813 | 40 | 287,510 | 71.53663 |
| 45 | 115,526 | 67.61167 | 45 | 298,305 | 18.93875 |
| 50 | 138,226 | 69.71442 | 50 | 476,047 | 37.08386 |
| 60 | 147,016 | 77.00876 | 60 | 584,622 | 37.56087 |
| 80 | 141,515 | 67.17754 | 80 | 3,451,197 | 227.7628 |
| 100 | 227,001 | 83.56837 | 100 | 4,010,942 | 130.2899 |
| 150 | 271,200 | 52.41632 | 150 | 5,238,189 | 184.4197 |

| | | | | | |
|---|---|---|---|---|---|
| 200 | 341,438 | 211.8987 | 200 | 12,858,721 | 48.7599 |
| 300 | 529,384 | 43.94714 | 300 | 29,544,352 | 44.11812 |
| 400 | 642,989 | 54.44302 | 400 | 67,634,368 | 11.73972 |
| 500 | 1,189,500 | 196.0965 | 500 | 117,937,500 | 11.1074 |
| 600 | 2,243,100 | 64.78164 | 600 | 256,789,463 | 8.223952 |
| 800 | 3,049,115 | 149.4159 | 800 | 799,591,105 | 11.59528 |
| 1000 | 4,107,673 | 76.89447 | 1000 | 2,024,936,721 | 12.65869 |

**Table 4: Comparative Data**

### 5.3.2 – Graphical Data

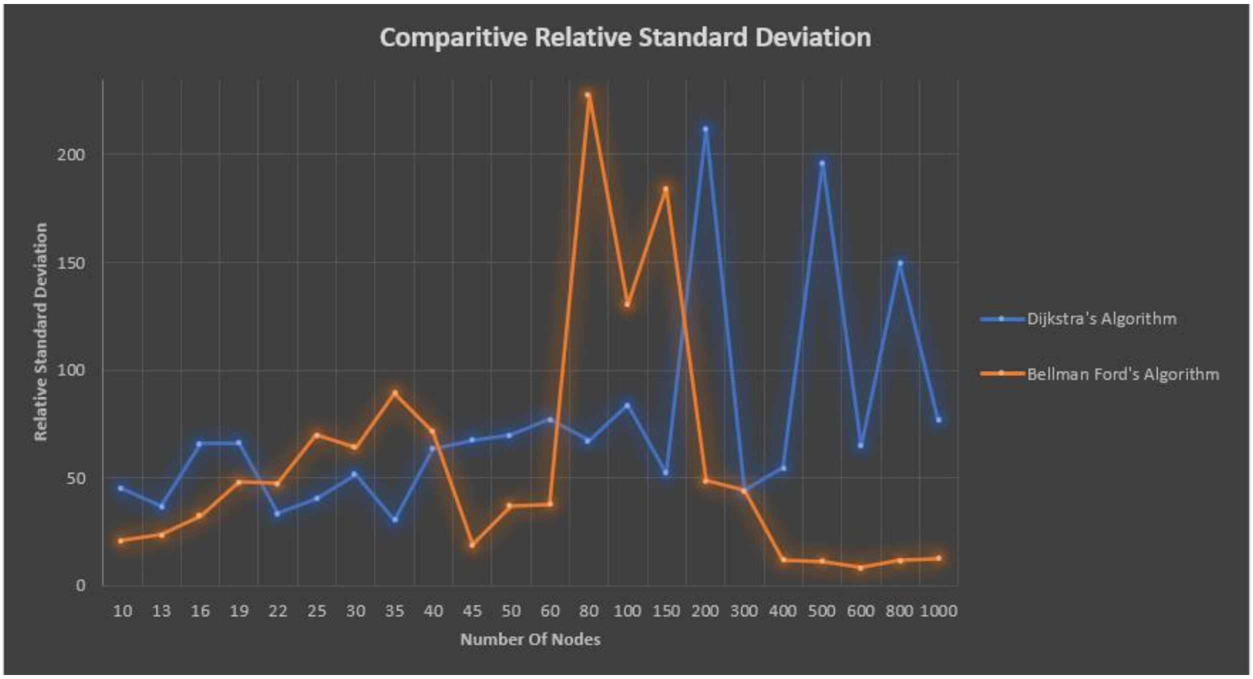
The following graph compares the number of nodes against time taken for Dijkstra's algorithm and Bellman ford's algorithm to complete finding the shortest path.




**Graph 5: Comparative Algorithm Time Complexity**

Note that the above graph only contains values from nodes 10 to 80 due to the large scale and large differences in execution time of both algorithms.

The following graph compares the number of nodes against the relative standard deviation for both the graphs.




**Graph 6: Comparative Relative Standard Deviation**

### 5.3.3 – Analysis

We see that for all number of nodes, Dijkstra's algorithm executes faster than Bellman Ford's. This is likely because Dijkstra's algorithm, being based on the greedy approach does not need to try all possible combinations in the graph. Bellman Ford's algorithm on the other hand is based on the dynamic approach and must try all possible combinations in the graph. Additionally, Dijkstra's algorithm will have to relax the nodes a lesser number of times compared to Bellman Ford's algorithm; hence it can be executed faster. The same is reflected in the graph. When looking at the relative standard deviation, we observe that Dijkstra’s algorithm on average has a higher relative standard deviation. This is because Bellman Ford’s algorithm always relaxes all edges when searching for the shortest path between two nodes in a graph. Hence in all cases, all edges of the graph are relaxed in Bellman Ford’s algorithm. In the case of Dijkstra’s algorithm, it only relaxes edges that need to be relaxed to obtain the shortest path for the given source and the destination. So, if there are many combinations to reach the destination, then the time taken will be higher, but if not, it will be lesser. This means that though Dijkstra’s algorithm is faster compared to Bellman Ford’s, Dijkstra’s algorithm may give drastically different execution times for different types of graphs. Lastly, we see that Bellman Ford’s algorithm slows down much faster than Dijkstra’s algorithm. This is seen through the larger slope of the graph (especially prominent from nodes 60 to 80).

## 6. EVALUATION OF HYPOTHESIS

The hypothesis stands true as it can clearly be seen that Dijkstra's algorithm is faster for all number of nodes. Additionally, by analysing the slope of the graphs, we see that Bellman Ford's algorithm gets slower at a faster rate than Dijkstra's algorithm. We also observe that due to the lower relative standard deviation of Bellman Ford's algorithm, it has a much more stable execution time.

## 7. EVALUATION

**Strengths** -

- Methods for both algorithms were written to ensure that computation is focused on the algorithms.
- Inclusion of both mean and relative standard deviation analysis allows us to effectively compare both algorithms mathematically.
- Exploration has considered control variables and ensures that the results remain unaffected by system resources and IDEs.
- Can verify if the shortest path execution is correct by comparing the results of both algorithms for each execution.
- Automated undirected graph creating system ensures that there is no human error related to graph creation. Ensuring that the graph is connected makes sure that both algorithms execute without errors.
- Use of multiple graphs for same number of nodes ensures that the algorithms are tested in multiple environments, hence enhancing the diversity of data. Data input is free from human bias.
- Inputs of both algorithms were in the same data structure. This ensures that the effect of data structures in both algorithms remains consistent.

**Limitations –**

- Number of edges has not been considered.
- The exploration does not consider negative edge weights.
- Space complexity of the algorithms is not considered in this exploration.

**Future Scope –**

Improvements to my original exploration can be made by considering the number of edge weights along with the execution times of the graphs, rather than only the number of nodes. Furthermore, the probability of creating an edge, could be changed to see the effect of edges on time complexity of both algorithms. Negative edge weights have not been considered in this exploration. Though there aren't many uses for negative edge weights, it would be interesting to consider how both algorithms perform when inputted negative edge weights. Another interesting consideration would be to use Dial's implementation of Dijkstra's algorithm on smaller edge weights and see how this affects the time complexity and execution time of Dijkstra's algorithm. Additionally, we could compare the efficiency of different dynamic and heuristic approaches used to solve the travelling salesmen problem. Lastly, we could also compare how AI driven algorithms, such as the A* algorithm, compares to both the greedy and dynamic approach.

## 8. CONCLUSION

In this paper, the effects of changing the number of vertices is considered for both Dijkstra's and Bellman Ford's algorithm in terms of execution time. They were analysed and compared based on mathematical observations. Additionally, logical explanations were made to explain the results from the experiment.

From the results, we see that the greedy programming technique (Dijkstra's algorithm), is much faster than the dynamic programming technique (Bellman Ford's algorithm), for any number of nodes.

Though the speed of Dijkstra's algorithm is faster than that of Bellman Ford's, we observe that there is a large deviation in the values of Dijkstra's algorithm. This may mean that the speed of the algorithm comes at the cost of higher inefficiencies when tested with graphs of different types. Bellman Ford's algorithm on the other hand has a lower deviation on average and this means that it can deal with many types of graphs in a consistent manner.

Since the edge weights are randomly picked, the effects of edge weights on both the algorithms cannot be confirmed, but as the number of nodes consistently increases, it can be confirmed that as the number of nodes increases, this increases the chance of creating an edge, both of which slow down both algorithms. Dijkstra's algorithm's time complexity is not affected as much as that of Bellman Ford's algorithm when the number of nodes increases. This may also be a defining factor when deciding which algorithm to use.

I hope this paper will prove useful to people by providing interesting perspectives on both algorithms. I hope it helps people who work on, and plan to implement solutions for the shortest path optimization problem.

## 9. BIBLIOGRAPHY


Coderucks, “Dynamic Programming Vs Greedy Algorithm”, coderucks, January 24, 2021, ”https://codecrucks.com/dynamic-programming-vs-greedy-algorithm/

Jack Little, “Directed and Undirected Graphs”, mathworks/MATLAB, May 7 2022, https://www.mathworks.com/help/matlab/math/directed-and-undirected-graphs.html

McQuain, “Data Structures And Algorithms”, VirginiaTech, May 7 2022, https://courses.cs.vt.edu/~cs3114/Fall10/Notes/T22.WeightedGraphs.pdf

Nicholas Wormald, “Graphs”, University of Western Australia, May 22, 2022, https://users.monash.edu/~lloyd/tildeAlgDS/Graph/#:~:text=The%20adjacency%20matrix%20of%20a,infinity%22%2C%20indicates%20this%20fact.&text=Adjacency%20Matrix%20of%20Weighted%20Directed%20Graph.&text=Adjacency%20Matrix%20of%20Weighted%20Undirected%20Graph.

Paul Black, “optimization problem”, xlinux, January 6, 2021, https://xlinux.nist.gov/dads/HTML/optimization.html

Ranjit Bhatta,”Graph Data Structure”, programiz, May 6 2022, https://www.programiz.com/dsa/graph

Ranjit Bhatta, “Bellman Ford’s Algorithm”, Programiz, July 9 2022, https://www.programiz.com/dsa/bellman-ford-algorithm

Sandeep Jain, “Dijkstra’s Shortest Path Algorithm”,GeeksForGeeks, May 25 2022, https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Sonoo Jaiswal, “Relaxation”, javaTpoint, May 23 2022, https://www.javatpoint.com/relaxation#:~:text=The%20single%20%2D%20source%20shortest%20paths,equivalent%20the%20shortest%20%2D%20path%20weight.

Timothy Sun, “Graphs”, Columbia, May 6 2022, http://www.columbia.edu/~cs2035/courses/ieor6614.S11/graph.pdf

Samah W.G. AbuSalim et al 2020 IOP Conf. Ser.: Mater. Sci. Eng. 917 012077

# Appendices

## Appendix A: Code Used For Data Collection

### A1: Bellman Ford's Algorithm

```
import java.io.*;
import java.util.Scanner;

// Class the computes the shortest distance using Bellman Ford
class ShortestPathBF {

    int[][] m_edges;
    StringBuilder m_outBuf;
    int m_numNodes;
    int m_numEdges;

    ShortestPathBF(int[][] edges, int num_nodes) {
    m_edges = edges;
    m_outBuf = new StringBuilder();
    m_numEdges = edges.length;
    m_numNodes = num_nodes;
    }

    void findShortestPath (int src)
    {

    // Reset the output buffer to print text
    m_outBuf.setLength(0);
    long start = System.nanoTime();

    // Initialize distance of all vertices as INFINITE except for source
    // which iz zero

    int dist[] = new int[m_numNodes];

    for (int i = 0; i < m_numNodes; i++)
        dist[i] = Integer.MAX_VALUE;

    dist[src] = 0;

    // Relax all edges numNodes-1 times.
    for (int count=1; count < m_numNodes; count++) {
```

```
        for (int e = 0; e < m_numEdges; e++) {

        int from = m_edges[e][0];
        int to = m_edges[e][1];
        int d = m_edges[e][2];

        // If the from node in the edge is at INFINITY distance, skip
        if(dist[from] == Integer.MAX_VALUE) continue;

        // If the distance to node can be relaxed, relax it
        if ( dist[from]+d < dist[to] ) dist[to] = dist[from] + d;
        }
    }

    // If we are still able to relax the edges, this might mean there are
    // negative cyles in the graph

    for (int e = 0; e < m_numEdges; e++) {

        int from = m_edges[e][0];
        int to = m_edges[e][1];
        int d = m_edges[e][2];

        // If the from node in the edge is at INFINITY distance, skip
        if(dist[from] == Integer.MAX_VALUE) continue;
        if (dist[from] +  d < dist[to])
        m_outBuf.append("Graph contains negative"
                   +" weight cycle\n");
    }

    m_outBuf.append("Vertex Distance from Source " + src + " is:\n");
    m_outBuf.append("Execution time [BF] : " + String.valueOf(System.nanoTime()-
start) + " nano seconds\n");

    // for (int i = 0; i < m_numNodes; i++)
    //     m_outBuf.append(src + " --> " + i + " is " + dist[i] + "\n");
Commented Code

    System.out.println(m_outBuf);
    }



    public static void main(String[] args)
```

```
    {
    // Enter the graph same way that we are entering for Dijkstra and lets build
edgers ourselves

  int graph[][] = new int[][] { { 0, 4, 0, 5, 0, 1, 0, 8, 0 },
                                { 4, 0, 8, 0, 6, 0, 10, 11, 328 },
                                { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                { 192, 0, 7, 0, 9, 14, 0, 6, 0 },
                                { 0, 22, 0, 9, 0, 10, 0, 3, 0 },
                                { 0, 1, 4, 14, 10, 0, 2, 0, 0 },
                                { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                { 8, 11, 0, 8, 0, 0, 1, 0, 7 },
                                { 0, 0, 2, 0, 5, 7, 6, 7, 0 } };

    //  No. of edges is equivalent to no. of non-zero elements in the matrix
    int num_nodes = graph.length;
    int num_edges = 0;

    for (int i = 0; i < num_nodes; i++) {
        for (int j=0; j < graph[0].length; j++) {
              if (graph[i][j] != 0) num_edges++;
        }
    }

    // Lets build edges from the graph.
    int edges[][] = new int[num_edges][3];

    int e = 0;
    for (int i = 0; i < num_nodes; i++) {
        for (int j = 0; j < graph[0].length; j++) {

        if (graph[i][j] != 0) {
            edges[e][0] = i;            // from node
            edges[e][1] = j;            // to node
            edges[e][2] = graph[i][j];  // distance
            e++;
        }
        }
    }


    ShortestPathBF spb = new ShortestPathBF(edges, num_nodes);

    Scanner sc = new Scanner(System.in);
    String s;
```

```
    s = sc.nextLine();
    spb.findShortestPath(8);
    s = sc.nextLine();
    spb.findShortestPath(8);
    s = sc.nextLine();
    spb.findShortestPath(8);
    s = sc.nextLine();
    spb.findShortestPath(8);
    s = sc.nextLine();
    spb.findShortestPath(8);

    // Every edge has three values (u, v, w) where
    // the edge is from vertex u to v. And weight
    // of the edge is w.
    // int graph[][] = { { 0, 1, -1 }, { 0, 2, 4 },
    //                   { 1, 2, 3 }, { 1, 3, 2 },
    //                   { 1, 4, 2 cd }, { 3, 2, 5 },
    //                   { 3, 1, 1 }, { 4, 3, -3 } };

    //    BellmanFord(graph, V, E, 0);

    }
}
```

Code modified from geeks for geeks.

## A2: Dijkstra’s Algorithm

```
import java.io.*;
import java.util.Scanner;

// Class that computes shortest path using Dijkstra's algorithm
class ShortestPathDijkstra {

    int[][] m_graph;
    StringBuilder m_outBuf;

    ShortestPathDijkstra(int[][] graph) {
    m_graph = graph;
    m_outBuf = new StringBuilder();
    }

    // Find the next minDstance node thats not yet processed
```

```
    int minDistance(int dist[], Boolean isProcessed[])
    {
        // Initialize min value
        int min_dist = Integer.MAX_VALUE;
    int min_index = -1;

        for (int v = 0; v < dist.length; v++) {
            if (!isProcessed[v] && dist[v] <= min_dist) {
                min_dist = dist[v];
                min_index = v;
            }
    }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printComputation(int src, int dst, int dist[], int n,
              Boolean isProcessed[], boolean allNodesProcessed)
    {
        m_outBuf.append("Vertex Distance from Source "+ src + " is:\n" );

        for (int i = 0; i < dist.length; i++) {

        String isMinFound = " ";
        if (isProcessed[i]) isMinFound = "*";

        if ( !allNodesProcessed && (i == dst)) {
        m_outBuf.append(src + " --> " + i + isMinFound + " is " + dist[i] + " <==
***\n");
        } else {
        m_outBuf.append(src + " --> " + i + isMinFound + " is " + dist[i] +
"\n");
        }
    }
    }

    // Function that implements Dijkstra's single source shortest path
    void findShortestPath (int src, int dst)
    {

    m_outBuf.setLength(0);
    long start = System.nanoTime();

    int num_nodes = m_graph.length;
```

```
        int dist[] = new int[num_nodes];  // shorted computed distance from src
to i as of now
    Boolean isNodeProcessed[] = new Boolean[num_nodes];

        // Initialize all distances from src as INFINITE (MAX_VALUE)  and
isNodeProcessed[] as false
        for (int i = 0; i < num_nodes; i++) {
            dist[i] = Integer.MAX_VALUE;
            isNodeProcessed[i] = false;
        }

        // Distance of source vertex from itself is always 0
        dist[src] = 0;

        // Find shortest path for all vertices
        for (int count = 0; count < num_nodes-1; count++) {

            // Pick the minimum distance vertex from src thats not processed yet
        int min_dist = Integer.MAX_VALUE;
        int minNode = -1;

        for (int v = 0; v < dist.length; v++) {
        if (isNodeProcessed[v]) continue;
        if (dist[v] <= min_dist) {
            min_dist = dist[v];
            minNode = v;
        }
        }

        // If minNOde is destination we are done..

        if (minNode == dst) {

        // We found the min staince to destination node
        m_outBuf.append("Yay !! we found the shortest path from " + src + " to "
+ dst + ": " + dist[dst] + "\n");
        m_outBuf.append("Execution time: " + String.valueOf(System.nanoTime()-
start) + " nano seconds\n");
        }

            // Mark the picked node as processed
            isNodeProcessed[minNode] = true;

            // Update dist of adjacent nodes from src
            for (int v = 0; v < num_nodes; v++) {
```

```
                // Update dist[v] only if is not in isNodeProcessed, there is an
                // edge from minNode to v, and total weight of path from src to
                // v through minNode is smaller than current value of dist[v]

        if (isNodeProcessed[v]) continue;
        if (dist[minNode] == Integer.MAX_VALUE) continue;
        if (m_graph[minNode][v] == 0) continue;

                if (dist[minNode] + m_graph[minNode][v] < dist[v])
                    dist[v] = dist[minNode] + m_graph[minNode][v];
        }

        }

    m_outBuf.append("Execution time [Dijkstra] : " +
String.valueOf(System.nanoTime()-start) + " nano seconds\n");
    // for (int i = 0; i < dist.length; i++)
    //     m_outBuf.append(src + " --> " + i + " is " + dist[i] +
"\n");   commented code
    System.out.println(m_outBuf);
    }


    public static void main(String[] args)
    {
    // Graph with N nodes is represented in NxN array for simplicity
        int graph[][] = new int[][] { { 0, 4, 0, 5, 0, 1, 0, 8, 0 },
                                    { 4, 0, 8, 0, 6, 0, 10, 11, 328 },
                                    { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                    { 192, 0, 7, 0, 9, 14, 0, 6, 0 },
                                    { 0, 22, 0, 9, 0, 10, 0, 3, 0 },
                                    { 0, 1, 4, 14, 10, 0, 2, 0, 0 },
                                    { 0, 0, 0, 0,10, 2, 0, 1, 6 },
                                    { 8, 11, 0, 8, 0, 0, 1, 0, 7 },
                                    { 0, 0, 2, 0, 5, 7, 6, 7, 0 } };

        ShortestPathDijkstra spd = new ShortestPathDijkstra(graph);

    Scanner sc = new Scanner(System.in);
    String s;

    s = sc.nextLine();
        spd.findShortestPath(8, 1);
    s = sc.nextLine();
```

```
    spd.findShortestPath(8, 1);
    s = sc.nextLine();
    spd.findShortestPath(8, 1);
    s = sc.nextLine();
        spd.findShortestPath(8, 1);
    s = sc.nextLine();
    spd.findShortestPath(8, 1);

    }
}
```

## A3: Graph Generator Algorithm

```
import java.io.*;
import java.util.Arrays;
import java.util.Scanner;
import java.util.Random;
import ShortestPathBF;
import ShortestPathDijkstra;

class Graphgenerator {

public static int[][] graphG(int N) //N is the number of vertices
{
    int temp;
    int[][] Graph = new int[N][N];
    Random number = new Random();
    for(int i =0; i < N; i++){
        for(int j = 0; j < N; j++){
            Graph[i][j] = -1;
        }
    }
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            if(i == j){
                Graph[i][j] = 0;
                continue;
            }
            else if(Graph[i][j] != -1)
            {
                continue;
            }
            if(number.nextInt(2) == 0){
```

```
                Graph[i][j] = 0;
                Graph[j][i] = 0;
                continue;
            }
              else{
                if(Graph[i][j] == -1 && Graph[j][i] == -1){
                    temp = 1+number.nextInt(999);
                    Graph[i][j] = temp;
                    Graph[j][i] = temp;
                }
            }
        }
    }
    return Graph;
}
    public static void main(String[] args){
        Random number = new Random();
        int size = 20;
        int S = 1+number.nextInt(size); // calculates the source
        int D = 1+number.nextInt(size); // calculates the destination
        int[][] X = graphG(size);
        // for (int i = 0; i < X.length; i++) { //this equals to the row in our
matrix.
        //     for (int j = 0; j < X[i].length; j++) { //this equals to the
column in each row.
        //        System.out.print(X[i][j] + " ");
        //     }
        //     System.out.println(); //change line on console as row comes to end
in the matrix.
        //  }
         ShortestPathDijkstra y = new ShortestPathDijkstra(X);
         int num_nodes = X.length;
    int num_edges = 0;

    for (int i = 0; i < num_nodes; i++) {
        for (int j=0; j < num_nodes; j++) {
              if (X[i][j] != 0) num_edges++;
        }
    }

    // Lets build edges from the graph.
    int edges[][] = new int[num_edges][3];

    int e = 0;
    for (int i = 0; i < num_nodes; i++) {
```

```
        for (int j = 0; j < num_nodes; j++) {

        if (X[i][j] != 0) {
            edges[e][0] = i;            // from node
            edges[e][1] = j;            // to node
            edges[e][2] = X[i][j];  // distance
            e++;
        }
        }
    }


         ShortestPathBF spb = new ShortestPathBF(edges, num_nodes);
         for(int i=0; i<21; i++){       // computes 20 trials per execution
            S = 1+number.nextInt(size); // calculates the source
            D = 1+number.nextInt(size); // calculates the destination
            X = graphG(size);
            y.findShortestPath(S, D);
            spb.findShortestPath(S);
         }
    }
}
```

Code modified from geeks for geeks.

## Appendix B: Raw Data Collection

### B1: Dijkstra's Algorithm

(Small font was used as the numbers were extremely large)

| IDE 1: Visual Studio Code (version 1.70) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of Nodes** | **Execution Time (nanoseconds)** | | | | | | | | | |
| 10 | 13300 | 10900 | 20300 | 13700 | 14600 | 21900 | 45700 | 18300 | 33400 | 15400 |
| 13 | 26300 | 30300 | 15900 | 40400 | 22100 | 25000 | 11800 | 29900 | 16000 | 14000 |
| 16 | 13300 | 25800 | 28100 | 40700 | 28600 | 36200 | 53000 | 20500 | 9900 | 32300 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 19 | 27400 | 12400 | 39100 | 28000 | 14000 | 56700 | 20100 | 49500 | 50600 | 21400 |
| 22 | 23000 | 10000 | 36600 | 34100 | 31300 | 40100 | 34400 | 43100 | 39500 | 41200 |
| 25 | 72800 | 31700 | 47600 | 17200 | 80700 | 40400 | 17700 | 90400 | 50800 | 50400 |
| 30 | 23600 | 60700 | 18300 | 108600 | 46400 | 26600 | 61100 | 45600 | 16600 | 79800 |
| 35 | 92500 | 118200 | 24500 | 84400 | 89000 | 103400 | 71100 | 102000 | 84900 | 124100 |
| 40 | 154000 | 112900 | 95200 | 90900 | 49600 | 86100 | 79700 | 120000 | 105800 | 135100 |
| 45 | 120100 | 134300 | 101400 | 26400 | 141300 | 49200 | 108700 | 349700 | 81400 | 61600 |
| 50 | 61500 | 159600 | 181800 | 74300 | 99100 | 191900 | 23100 | 168300 | 43700 | 124200 |
| 60 | 295400 | 88600 | 207700 | 51800 | 203100 | 144600 | 135900 | 37900 | 180300 | 24700 |
| 80 | 326000 | 69700 | 296400 | 295600 | 203100 | 144600 | 135900 | 37900 | 180300 | 24700 |
| 100 | 428200 | 815700 | 68230 | 231400 | 128900 | 502100 | 145800 | 186200 | 157100 | 125100 |
| 150 | 309400 | 184200 | 360600 | 136600 | 187800 | 360900 | 177100 | 22700 | 666000 | 326300 |
| 200 | 2740300 | 2267700 | 821900 | 418300 | 263300 | 409600 | 359800 | 226500 | 148700 | 169300 |
| 300 | 357500 | 771700 | 479600 | 129900 | 295200 | 497500 | 108800 | 378900 | 561600 | 85000 |
| 400 | 979700 | 621700 | 517900 | 1207300 | 1460400 | 756400 | 540000 | 269000 | 718400 | 123600 |
| 500 | 1037600 | 20747900 | 1346900 | 16157500 | 2175900 | 1350400 | 339000 | 929600 | 609900 | 1496400 |
| 600 | 2195000 | 595500 | 1959400 | 645600 | 364900 | 261700 | 528000 | 606600 | 575900 | 1631800 |
| 800 | 4508000 | 3068500 | 28925900 | 3544900 | 2338900 | 3354000 | 4619200 | 436100 | 3446600 | 1145500 |
| 1000 | 5900500 | 2180000 | 8844500 | 238500 | 107100 | 758500 | 4600300 | 4491700 | 872900 | 1422700 |

| IDE 2: Eclipse (version 2022-06 R) |
|---|

| Number of Nodes | Execution Time (nanoseconds) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 16900 | 14400 | 24000 | 24200 | 24000 | 8400 | 14900 | 14000 | 33400 | 14500 |
| 13 | 15400 | 16000 | 32000 | 14500 | 19600 | 22000 | 17600 | 23400 | 16000 | 38100 |
| 16 | 32600 | 41400 | 31900 | 57700 | 24300 | 27200 | 16500 | 34400 | 25100 | 115300 |
| 19 | 22400 | 31900 | 10600 | 38800 | 29900 | 33400 | 54000 | 24400 | 19200 | 113200 |
| 22 | 40200 | 56900 | 67400 | 54400 | 42800 | 28400 | 60300 | 42900 | 34000 | 41200 |
| 25 | 40400 | 42200 | 31500 | 72000 | 36200 | 60200 | 50000 | 63700 | 64000 | 66300 |
| 30 | 75800 | 42300 | 28100 | 93700 | 61500 | 68400 | 89100 | 42500 | 61000 | 23800 |
| 35 | 114100 | 75100 | 97300 | 111300 | 46900 | 81500 | 132100 | 88000 | 52800 | 112300 |
| 40 | 134200 | 36800 | 114600 | 79900 | 74200 | 322100 | 49800 | 36400 | 44500 | 106600 |
| 45 | 61400 | 38500 | 142300 | 265400 | 107000 | 149900 | 67700 | 100400 | 51500 | 98200 |
| 50 | 123200 | 184300 | 463800 | 111600 | 91900 | 104800 | 126300 | 51600 | 68700 | 67900 |
| 60 | 183800 | 189600 | 136000 | 66000 | 26900 | 63300 | 45400 | 49700 | 17100 | 61000 |
| 80 | 29700 | 123500 | 82200 | 37500 | 98900 | 127700 | 63300 | 80600 | 88000 | 270900 |
| 100 | 125400 | 74500 | 423800 | 128200 | 198800 | 37000 | 159900 | 188400 | 123900 | 189800 |
| 150 | 326100 | 432300 | 329900 | 229300 | 144000 | 232400 | 144000 | 232400 | 403700 | 273200 |
| 200 | 169300 | 152800 | 410800 | 139300 | 148900 | 593900 | 59400 | 203200 | 321700 | 204900 |
| 300 | 98000 | 563500 | 316600 | 54500 | 230000 | 43400 | 433000 | 764100 | 402800 | 97700 |
| 400 | 103300 | 44600 | 621900 | 767400 | 564700 | 149700 | 680000 | 689900 | 758500 | 745700 |
| 500 | 1381400 | 991400 | 871400 | 656700 | 405100 | 1365200 | 266500 | 1236900 | 458300 | 1576300 |
| 600 | 1432100 | 2453000 | 2298300 | 2031000 | 1195800 | 2012900 | 355500 | 514300 | 1488300 | 889000 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 800 | 1325500 | 2906500 | 2773100 | 374800 | 851800 | 2382500 | 3858400 | 3383100 | 2621600 | 3506400 |
| 1000 | 1122700 | 3946100 | 5256400 | 5044700 | 177200 | 4927600 | 1434900 | 1434600 | 2252200 | 3142800 |

B2: Bellman Ford's Algorithm

(Small font was used as the numbers were extremely large)

| IDE 1: Visual Studio Code (version 1.70) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of Nodes** | **Execution Time (nanoseconds)** | | | | | | | | | |
| 10 | 36000 | 28000 | 34500 | 24500 | 36200 | 41500 | 59200 | 35000 | 38800 | 34600 |
| 13 | 61000 | 72900 | 49900 | 53700 | 41200 | 100600 | 57700 | 50100 | 66600 | 44300 |
| 16 | 120700 | 67400 | 65400 | 65600 | 108700 | 81500 | 129300 | 103600 | 92500 | 98900 |
| 19 | 112900 | 118100 | 118900 | 111800 | 117200 | 149200 | 107500 | 152300 | 172200 | 107400 |
| 22 | 182400 | 168000 | 166900 | 191500 | 187200 | 272900 | 282700 | 174800 | 259700 | 203500 |
| 25 | 273700 | 259500 | 256700 | 265400 | 239700 | 307300 | 331800 | 149800 | 107000 | 80800 |
| 30 | 435000 | 439100 | 571800 | 181000 | 120300 | 94600 | 462000 | 571300 | 513200 | 442400 |
| 35 | 664400 | 772100 | 170600 | 133600 | 157900 | 195000 | 135100 | 107700 | 138000 | 281400 |
| 40 | 931000 | 293600 | 162000 | 132700 | 177200 | 250800 | 180900 | 222200 | 155900 | 163500 |
| 45 | 380300 | 231400 | 262700 | 263900 | 263000 | 347200 | 395000 | 287000 | 274900 | 351700 |
| 50 | 357200 | 764400 | 498600 | 335500 | 315000 | 300500 | 355000 | 562500 | 679800 | 440900 |
| 60 | 804000 | 580600 | 745900 | 527600 | 643800 | 700300 | 763100 | 1421400 | 840700 | 955600 |
| 80 | 1080700 | 1900100 | 1874300 | 1362000 | 2083700 | 1629400 | 1193000 | 9016100 | 1232800 | 2188200 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 2075300 | 18699700 | 205560 | 3466700 | 2162600 | 3514300 | 1832800 | 821000 | 1709700 | 954200 |
| 150 | 309400 | 184200 | 360600 | 136600 | 187800 | 360900 | 177100 | 22700 | 666000 | 326300 |
| 200 | 31286200 | 27689900 | 17495000 | 11618100 | 13454500 | 9718800 | 11399200 | 12216900 | 13003000 | 10270000 |
| 300 | 76050500 | 30492800 | 53020900 | 22648300 | 26189200 | 24503500 | 22737600 | 25030800 | 24816600 | 26033500 |
| 400 | 54824300 | 62584800 | 84945200 | 73603700 | 72374000 | 65990400 | 67882200 | 64014800 | 64716100 | 61557400 |
| 500 | 157741700 | 102236200 | 108568000 | 102379200 | 113677200 | 106203800 | 111136600 | 104532400 | 111603800 | 115837200 |
| 600 | 236330600 | 239946000 | 241099000 | 241107000 | 241315800 | 249153400 | 249259500 | 231633100 | 233536700 | 262235900 |
| 800 | 572574300 | 570297400 | 806017300 | 828918500 | 814160900 | 864612600 | 818976500 | 809288200 | 819130600 | 823684500 |
| 1000 | 1840821500 | 2838474700 | 2515692700 | 2268271100 | 1956530700 | 1915711400 | 1947348600 | 1906181400 | 1896393600 | 1922388500 |

| IDE 2: Eclipse (version 2022-06 R) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of Nodes** | **Execution Time (nanoseconds)** | | | | | | | | | |
| 10 | 27900 | 31200 | 27400 | 43000 | 35100 | 34900 | 34600 | 40800 | 38800 | 37000 |
| 13 | 67000 | 77300 | 75800 | 62100 | 67600 | 63100 | 70700 | 71600 | 38900 | 61300 |
| 16 | 153400 | 202100 | 92500 | 117200 | 72400 | 78700 | 117200 | 118600 | 89600 | 121400 |
| 19 | 112300 | 165100 | 185500 | 105900 | 145100 | 207900 | 131300 | 155600 | 403700 | 102200 |
| 22 | 62300 | 78300 | 126300 | 61100 | 49100 | 45200 | 101600 | 166700 | 199600 | 52900 |
| 25 | 49100 | 125700 | 59500 | 66800 | 95850 | 40800 | 56800 | 60000 | 461300 | 62400 |
| 30 | 500400 | 203600 | 174500 | 146500 | 107500 | 171200 | 124100 | 99300 | 104900 | 401200 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 35 | 162400 | 340600 | 113000 | 170200 | 100800 | 148600 | 109900 | 177600 | 838200 | 165100 |
| 40 | 240500 | 227800 | 240700 | 165600 | 262500 | 209700 | 131600 | 203400 | 244000 | 239300 |
| 45 | 249900 | 284100 | 232700 | 356400 | 314600 | 358500 | 328600 | 298300 | 187600 | 231400 |
| 50 | 489000 | 561100 | 288900 | 490300 | 982400 | 434000 | 409900 | 420100 | 359800 | 464200 |
| 60 | 651400 | 746800 | 296500 | 610600 | 311900 | 346600 | 217100 | 521400 | 259300 | 633100 |
| 80 | 1883900 | 1416800 | 997800 | 1008100 | 1162000 | 1654500 | 2302300 | 1193800 | 41028400 | 1723800 |
| 100 | 1553100 | 1474900 | 1432600 | 9363900 | 786000 | 1634700 | 9363900 | 884400 | 3637400 | 1413500 |
| 150 | 2969100 | 2506300 | 3168800 | 3450700 | 2391500 | 3450700 | 2391500 | 4386300 | 3504800 | 3265100 |
| 200 | 8961800 | 8151900 | 9349800 | 9249000 | 9947000 | 10659900 | 8064300 | 9800100 | 11180900 | 9761200 |
| 300 | 25461000 | 25302700 | 26437700 | 24688400 | 23781000 | 26202100 | 24708600 | 28178500 | 25059000 | 25461000 |
| 400 | 64722200 | 60846700 | 60580300 | 65921000 | 64425800 | 60608100 | 79263300 | 76501200 | 79691500 | 63829100 |
| 500 | 114777700 | 120778200 | 118889700 | 123494500 | 115506100 | 130383800 | 127762200 | 129367800 | 125936400 | 124811700 |
| 600 | 265795900 | 264480800 | 258685000 | 262127700 | 295681900 | 314958600 | 257980400 | 265480400 | 268192100 | 243710900 |
| 800 | 868981300 | 884850200 | 971469400 | 784434800 | 777151700 | 775221300 | 800073700 | 822220800 | 780167000 | 881256100 |
| 1000 | 1897219200 | 1993033400 | 1899000400 | 2153442600 | 1924845600 | 1891192500 | 1920363900 | 1894614200 | 1892271700 | 1782781500 |