An Evaluation of Performance between Network Routing Algorithms within a Simulated Scalable Environment

Tallon McAbee
Department of Computer Science
University of South Carolina Upstate
Spartanburg, South Carolina, USA
November 28, 2021

tbmcabee@email.uscupstate.edu

ABSTRACT

The internet has become an integral part of society since its emergence as a way for individuals to exchange information between two points across vase distances in an almost instantaneous manner. This influence has brought the utilization of networks into the essential infrastructure of society, allowing individuals to turn what was once a month of traveling by horse to deliver a piece of information into literal seconds of time by utilizing networks of nodes. One of the ways networks transfer data between its nodes are network routing protocols which allow nodes to send data in a more efficient manner utilizing a set of rules and specific routing algorithms. These algorithms allow protocols to find the "shortest path" between nodes on a network, which in turn allows the protocol to choose the most efficient route for the data to pass through. As the shortest path allows the data to reach its desired location by using the lowest number of resources. This methodology also plays into other key infrastructure points of society such as GPS systems, transportation design and other aspects that require finding the shortest path [7]. Which leads to the question behind this research endeavor. Which algorithm, from specific network protocols, will have the best comparative performance within an ever-growing simulated environment of nodes. The idea is to observe how each algorithm reacts to a scalable network of simulated nodes and to compare those metrics to see how the performance of both algorithms change based on specific factors within the simulated environment. The two network protocol algorithms tested will be Dijkstra's algorithm from Link State Protocol and the Bellman Ford algorithm from Distance Vector Protocol. The evaluation metrics that will be utilized to measure these reactions will be as follows: speed of computation (nanoseconds) and accuracy of computation (%). All testing and generation-oriented code will be coded in Java (JDK 16) as it is an object-oriented language with the necessary libraries to build the specific architecture needed for testing. The specific architecture mentioned will be a coded environment separated by functionality into two logical spheres, one for data generation of scalable environments and the other for measuring as well as controlling the parameters behind each of the simulations. These scalable environments will be undirected node-based graphs generated from inputted parameters such as the number of nodes to be generated. The software environment used for developing and running all simulations will be the Java oriented IDE application known as "Eclipse", which will be located within a Windows 10 Pro 64-bit operating system. All simulations within this specific software will also be ran on the same hardware, specifically an Intel i7-10700k processor environment utilizing a set 8GBs of DRAM as primary memory storage. Results obtained from testing seem to support the finding that within simulated, scalable environments, Dijkstra's algorithm would be the most optimal algorithm to utilize. This is based upon its magnitudes faster computations and verified accuracy ratios that show quick and accurate solutions for each of the Test Case's SPPs when compared to the Bellman Ford algorithm's results. Although it should also be mentioned that the Test Cases utilized showed a 100% accuracy ratio across the board for both algorithms during testing, meaning that future experiments should put an emphasis on larger datasets with less simplistic values if they want to see a variety of accuracy ratio data.

Keywords

Nodes, Graphs, Edges, Weights, Scalability, Shortest Path Problem, SPP, Link State Protocol, Distance Vector Protocol, Greedy Algorithm, Dynamic Programming Approach, IDE, Undirected Graph, Directed Graph

1. INTRODUCTION

Networks utilize network routing protocols in other to find the most efficient approaches within an interconnected node environment. Each protocol utilizes it own metrics and rules for finding these approaches, however they all utilize routing algorithms or graph algorithms when finding the "shortest path" within that network. This specific issue of finding the shortest path is defined as the shortest path problem [1], which in its simplest form represents the optimum path within a network that utilizes the least sum of 'weight' when moving across 'edges' [1]. The best form of representation in this case, is to visualize a network as a graph of interconnected points, with each point on the graph representing nodes and the lines or 'edges' connecting these points representing logical connections between each of the nodes. Each logical connection or edge is then designated a quantitative 'weight'. So, what does this weight represent? Well, it could represent factors such as heavy packet traffic, high latency and poor connectivity which would result in a 'heavier' weight therefore making the connection between two nodes logically farther from one another compared to another node with less traffic. In essence, weight serves the same purpose as distance in descripting connections but in a more accurate manner concerning the logic within those connections. For example, visualize two highways of cars going in one direction, where the first highway is full of cars and the second highway only has a few cars flowing through it. In this case, highway one would be designated as the heavier highway as it has a larger influx of cars weighing it down. This concept can then be easily translated into describing weighed edges between nodes, where edges that are full of heavy data traffic have a heavier weight compared to edges that have less traffic. This correlates back to the shortest path problem, where the idea behind routing algorithms is to find the shortest path from one point to another through the combination of the least sum of 'weight' when moving across edges within a network. In summary, routing algorithms treat networks as graphs with weighted edges between points representing network nodes, these algorithms then look for which combination of edges between two specified nodes has the smallest sum of weight. This combination of edges that results in the smallest sum of weight is then designated as the shortest path.

Which leads into the idea central to the purpose of this research project. Which routing algorithm, Dijkstra's algorithm, or the Bellman Ford algorithm, perform comparatively better in a scalable simulated network environment. The specific focus behind the research here is on how each algorithm handles differing sizes of larger environments, which is known as 'scalability'. This relates to real life use cases where the scalability of a network routing algorithm might cause different effects for differing environments. For example, a small business might originally utilize a protocol to handle 25 nodes at most on their network, however after a new business merger the company has grown in size to about 1000 nodes on their network. Performance differences between routing algorithms in these situations can possibly hurt the efficiency of a business when moving data across their network infrastructure. In this case if the specified network routing algorithm within the infrastructure is not suited for larger environments, compared to its counterpart, it could not only lead to performance issues such as high latency, but also lead to the loss of capital for a company through its decrease in throughput due to the previously stated latency issues.

Therefore, this research project will set out to see if there are any discernable differences between the performance of each algorithm within these scalable environments through the comparison of specific evaluation metrics. The specific evaluation metrics compared will be the speed and accuracy of the algorithms within the previously mentioned environment.

2. LITERATURE REVIEW

Due to the importance of both Dijkstra's algorithm and the Bellman Ford algorithm for finding the solution to shortest path problems, there has been a huge effort within the research field to optimize these algorithms further, design hybridized algorithmic solutions for specific use cases and even comparing the performance of each algorithm to the other within simulated environments similar to this research project's efforts.

As recently as June 2021, Mani Parimalam Said Broumi, Kathikeyan Prakask and Selcuk Topal began an intellectual research journey into looking at the utilization of the Bellman Ford algorithm for solving SPPs (Shortest Path Problems) under picture fuzzy environments. In essence the team wished to propose a new algorithm that could utilize TPFNs or (Trapezoidal picture fuzzy numbers) [5] to find SPPs in uncertain numerical situations. Whereas the classical Bellman Ford algorithm required set numerical values in order to find the SPP, the newly proposed algorithm is an updated version of Bellman Ford that integrates the utilization of uncertainty through TPFNs. The results of their research revealed that the new algorithm had superior advantages within practical utilizations compared to its classical counterparts' limitations [5].

Samah W.G Abusalim, Rosziati Ibrahim, Mohd Zainuri Saringat, Sapiee Jamel and Jahari Abdul Wahab go into a very similar research endeavor as this project, where the team compares both Dijkstra's algorithm and the Bellman Ford algorithm on their complexity and performance in terms of shortest path optimization [2]. This in-depth comparison takes into account each algorithm's Big O notation complexity, simulation times for both algorithms in finding single pair shortest path problems and each algorithm's Z notation. From this comparison the team was able to conclude that Dijkstra's algorithm was the faster algorithm for real life applications, as its simulations and Big O notation comparisons had shown that Dijkstra's algorithm seemed to handle a larger body of nodes faster, compared to Bellman Ford's algorithm which showed faster speeds in smaller groups of nodes with an exponential increase in simulation run time, as the body of nodes grew larger [2].

In Yefim Dinitz and Rotem Itzhak's article from the Hournal of Discrete Algorithms, they proposed a hybridized algorithm combining the Bellman Ford algorithm and Dijkstra's algorithm, with the goal of creating an improved algorithm that took advantage of both algorithm's approaches to finding SPPs [4]. Unlike the other presented research literature, this specific article utilizes mathematical proofs and logical analysis to uphold its claims rather than the utilization of simulations or high-level notation comparisons. Throughout this team's research they were able to provide multiple proofs that would uphold the utilization of a hybrid approach by showing how this specific approach could speed up execution through improving running time bounds of the Bellman Ford's algorithm and by utilizing Dijkstra's algorithm within a BFD round to accelerate the approach further.[4].

Within this project's own research approach there are multiple identifiable advantages and disadvantages that relate to its current approach. Such advantages include the centralization of implementation, isolation from outside factors and ease of testing recreation. The centralization of implementation is due to the small physical footprint of the current environment that will be tested throughout this research project. Everything will be limited to a single IDE/hardware environment and will therefore make the current implementation much more centralized than a real-life implementation, which could stretch across multiple factors or pieces of hardware. Isolation from outside factors also play a role within these advantages due to its ability to help keep all testing within a controlled environment. Similar to how a Virtual Machine can run within its own bubble on an OS, this testing environment will be isolated to its own code and can only be affected by its code, therefore outside factors that might normally affect a real implementation of these tests will not impact the testing environment. Recreation of tests is the final role within the advantages of the current approach, as new tests can be ran multiple times and recreated instantly through using the same centralized code, allowing for more simulations, which will increase the accuracy of the dataset by providing more data for comparison. Disadvantages include that the tests will be relative to the testing environment rather than a real implementation scenario and my own coding implementations could lead to possible outliers in data that might affect the datasets. Due to the mass of factors that could be accounted for, these tests cannot include absolutely everything within a real use case, which means that the results of this research project will need to be taken from a relative testing perspective rather than an actual implementation perspective, as there are many other factors that cannot be accounted for within these simulated environments that could affect real implementations of these cases in ways that does not correlate with the tested data. Such as hardware possibly favoring specific routing protocols due to higher technical specifications or other hardware limitations that could not be included within the simulated environment. It also must be considered that my own limited coding scope could affect the simulations as well either due to a coding mistake or implementation mistake from my own faults, and that data in this project should be backed up with other similar studies if utilized as a source.

3. METHODOLOGY

3.1 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm can be best summarized as an optimized solution to the shortest path problem, utilizing a greedy algorithmic approach. This approach is known for its ability to optimize solutions by taking advantage of the most optimal choice at each step in an algorithm or specific process. The greedy algorithm approach is regarded by some as the perfect general solution to optimizing problems [2], however most agree that the approach is not effective in all issues. In this case, Dijkstra's algorithm is regarded as an effective implementation of the greedy algorithm, since it can find the SPP solution when searching for a specific node within a graph, without having to visit other unnecessary nodes within the same graph [2]. This greedy method of finding a node solution gives Dijkstra's algorithm an advantage in computational speed, as it concludes itself once a solution is found and does not revisit nodes after its first visit.

However, this does not exclude the algorithm from disadvantages when comparing it to other solutions. It has been noted that due to the specific way Dijkstra's algorithm is implemented, it can consume a large amount of RAM utilization in larger graph environments. It also does not account for negative weights within a graph's edges, thereby excluding it entirely from graphs that rely on negative weights to represent traffic [2].

At its core, Dijkstra's algorithm begins at an initial node, and then computes the distance between the initial node and every other node until it reaches its destination [3]. For example, if there were a network of 5 nodes the where the user wishes to find the shortest path from node 1 to node 5 which for this case is connected in a linear path.

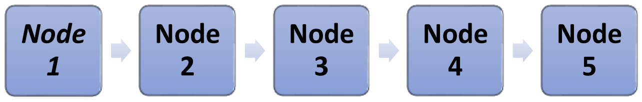

Figure 1: The linear path between nodes 1 and 5. Where node 1 is currently the initial selected node.

Dijkstra's algorithm would set node 1's distance to 0 and store the value within node 1. It would then find the distance from node 1 to node 2 which in this case is the integer 5 and store that distance within node 2.

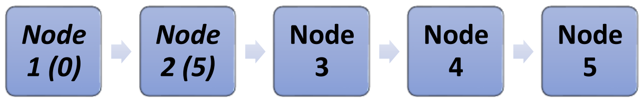

Figure 2: Node 1 within the path currently has a value of 0, whereas the 5 integer value stored in Node 2 is the distance between the two nodes.

Then it would find the distance between node 2 and node 3 which is 10. Where it would then add the value stored within node 2, and the distance value between nodes 2 and 3. It would then store that new computation within node 3 and continue the process until it reached node 5.

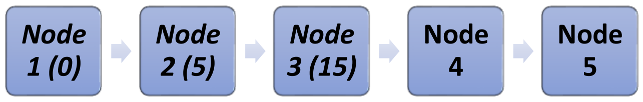

Figure 3: The distance between node 2 and node 3 is 10, therefore the distance stored within node 3 is 15.

This process ends up finding the distances between each node and node 1, storing their individual distances within the node and terminating the sequence once the destination node has been reached.

However, the linear implementation above does not account for graphs with varying connections with different distances and topologies. Therefore, the question arises, what steps does Dijkstra's algorithm take to account for non-linear graphs such as the simulated environment utilized within this project?

3.1.1 Non-Linear Algorithm Steps

1. Initial node J within the graph (G) sets its own distance (D) to an integer value of 0, while setting all other references to the other nodes' distances within G to ∞.

2. Node J will then check the edge weight (E) between itself and its neighbor node K, setting their distance to the following:

$$D_K = D_J + E_{J\ to\ K}$$

3. Once node K's distance has been defined, each of node J's direct neighbor nodes will repeat this step until each neighbor's distance has been set.
4. After each neighbor has been set, the neighbor with the minimum D value will be selected as the next node to visit. Node J will then be marked as visited, and the minimum D value node will be set to the current node. If there are no neighbors left that are unvisited, then a node is selected within the graph that has yet to be visited.
5. The current node C will then repeat steps 2-4 until either the destination node has been reached or all available nodes have been visited.

NOTE: Some implementations of Dijkstra's algorithm compute the distance between initial node J and every node within the graph, however the implementation within this project stops computation once the destination node's distance has been computed.

3.2 THE BELLMAN FORD ALGORITHM

The Bellman Ford algorithm is a versatile and sturdy solution that utilizes the dynamic approach to solve the shortest path problem, this approach specifically utilizes the methodology of separating larger problems into smaller subsections [2]. In this case, the Bellman Ford algorithm separates the larger graph into smaller subsections in order to handle distance calculations one step at a time, decreasing the amount of memory utilization while also increasing the computational time of the process.

Unlike its contemporary, the Bellman Ford algorithm can compute negative edge weights within a graph due to its self-correcting implementation method. Since greedy algorithms find the most optimal solution within that single instance, they do not reconsider the same solution later when other variables might have changed. The bellman ford algorithm does this by revisiting all the distances for multiple iterations, updating new distances if other factors may have changed [2].

An example to help visualize this would be to imagine a directed graph, of 5 nodes numbered 1 to 5. Node 2 will then be set as the initial node during this algorithm example.

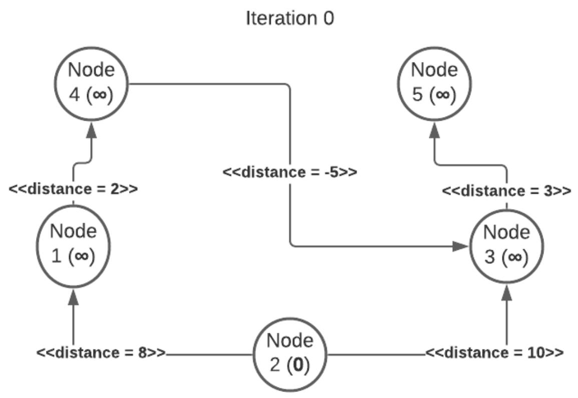

Figure 4: A graph of 5 nodes (1, 2, 3, 4 and 5), that have their initial distances set.

Now in the Bellman Ford algorithm, there are is a specific parameter that has to be calculated before a solution can be found. This parameter is known as the number of iterations to be ran. As stated before, the way the Bellman Ford algorithm handles negative weights is through its utilization of a non-greedy, dynamic programming approach, which means it will revisit all the nodes within a graph multiple times before the computation is over. Therefore, the algorithm must decide how many times it needs to check the graph until the correct solution is found. This value is calculated through the simple equation below:

$$I(Iterations\ to\ be\ ran) = N(Number\ of\ Nodes) - 1$$

In essence, the Bellman Ford algorithm should have found the most optimal solution by the number of nodes value minus 1 times of iterations ran [3].

In this case, since there are currently 5 nodes within the graph, the algorithm will should run 4 times before the most optimal solution has been found. It should also be noted, this is a directed graph, therefore connections will only flow in the direction where it points on the graph [3]. Which means node 3 and transfer data to node 5, however node 5 cannot transfer data to node 3. Iteration 1 is shown below:

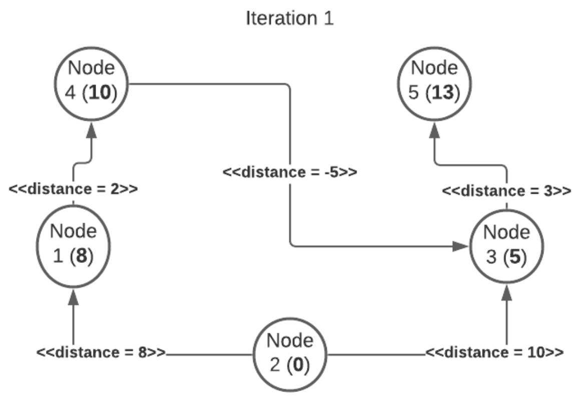

Figure 5: In iteration 1, the distances between each of the nodes and node 2 have been recorded. However, notice the distance stored in node 3?

After iteration 1, the first round of distances has been found between node 2 and its neighbors. However, it seems the distance stored within node 5 is incorrect. As the distance stored within node 3 is only 5 whereas node 5 has a distance of 13. This is the error you would see in using Dijkstra's algorithm to solve a graph with a negative weight. See, initially node 3 had stored the value of 10 within its distance and then utilized that value to calculate node 5's data. However, once the algorithm reached node 4 which has a negative connection to node 3, it updated node 3's value to reflect that connection after the fact. Meaning that node 3 has its correct minimum distance, but node 5's value is based on the original node 3 distance value. This will be correct in iteration 2, where the each of the distances value will be rechecked and updated.

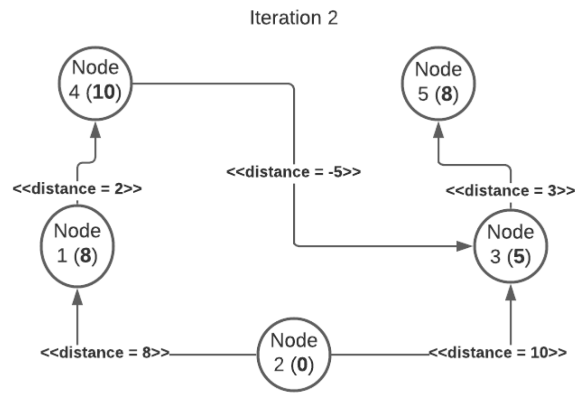

Figure 6: In iteration 2, the algorithm has re-checked each of the distances to update the values stored within the nodes. In this case node 5 now reflects the value stored in node 3.

Within iteration 2, the algorithm checks each value stored in each node and then recalculated the distances. The minimum distance was then stored within each node in order to reflect its true minimum distance. As mentioned earlier, this is the methodology that allows the Bellman Ford algorithm to handle negative weights since old values that might be incorrect can be corrected during other iterations of the algorithm.

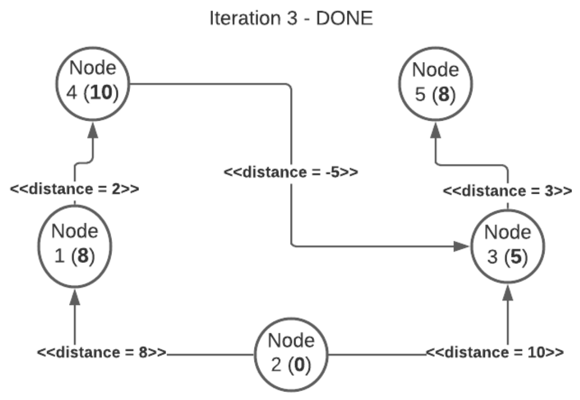

Figure 7: In iteration 3, the algorithm checks the current distance values within each node. No new minimums are found; therefore, no data is updated.

Within iteration 3, the algorithm follows the same steps as mentioned earlier all over again. It checks the values within each of the nodes, setting a new minimum distance value if one is found. However, in this iteration there are no new distances updated. This triggers another rule of the Bellman Ford algorithm to come into play. Which is, if an iteration is ran and no new values have been updated, then the computation stops as the solution has been found. This occurs even if it is on an iteration before iteration n-1. This rule allows the processor to save resource consumption from calculating unnecessary data, since if the values do not change within the nodes, there will be no further changes to reflect within the other nodes. Which means the graph example has found its shortest path problem before it even reached iteration 4.

3.2.1 Algorithm Steps

1. Initial node J within the graph (G) sets its own distance (D) to an integer value of 0, while setting all other references to the other nodes' distances within G to ∞.
2. Before any edge weights are accounted for, the algorithm will calculate the number of iterations to run within this computation through the following formula (I = # of iterations, N = # of nodes within G):

$$I = N - 1$$

3. Node J will then check the edge weight (E) between itself and its neighbor node K, setting their distance to the following:

$$D_K = D_J + E_{J\ to\ K}$$

4. Once node K's distance has been defined, each of node J's direct neighbor nodes will repeat this step until each neighbor's distance has been set.
5. After each neighbor has been set, the neighbor with the minimum D value will be selected as the next node to visit. Node J will then be marked as visited, and the minimum D value node will be set to the current node. If there are no neighbors left that are unvisited, then a node is selected within the graph that has yet to be visited.

6. The current node C will then repeat steps 2-4 until either the destination node has been reached or all available nodes have been visited.
7. Then the algorithm will reset all nodes back to unvisited, restarting from the initial node J and recalculating each D value. This will repeat based upon the earlier calculated value I.

3.3 ARCHITECTURE

The main methodology behind the architecture of this research project, is the logical interaction between data generation and testing of the generated data.

From its initial user facing architecture, the project utilizes a master driver class to allow the user to input specific data parameters that will alter or set specific variables within the tests. In essence, the driver will output specific text in the console to prompt a user to input data for each requested variable. These variables include the number of nodes to be generated (int), the specific algorithm that will be tested (String) and if the user would like to print out the generated graph into the console (Boolean).

The driver is primarily for keeping an easy and centralized method of simulating specific tests during the testing phase of research. Instead of running separate classes with their own drivers or user input interactions, this method of controlling centralized variables allows for convenient simulation control/data harvesting and streamlined developmental debugging.

However, as Figure 8 depicts, the driver is simple the logical transition between the user and the source code's interactions. From here, inputs are feed directly into the Testing Implementation portion of code. Which will then redirect any data generation variables from the driver, such as the integer value for number of nodes to be generated, towards the Data Generation portion of code. This portion will then generate a graph based upon the user's parameters and return it back into the Testing Implementation portion. Which, based upon the String value for algorithm choice, will simulate the selected algorithm utilizing the graph's specific data structure. Once all computations for a simulation have been handled, the Testing Implementation portion will then redirect the recorded evaluation metrics and data concerning the generated graph into a generated txt file to allow for easier and storable data interpretations.

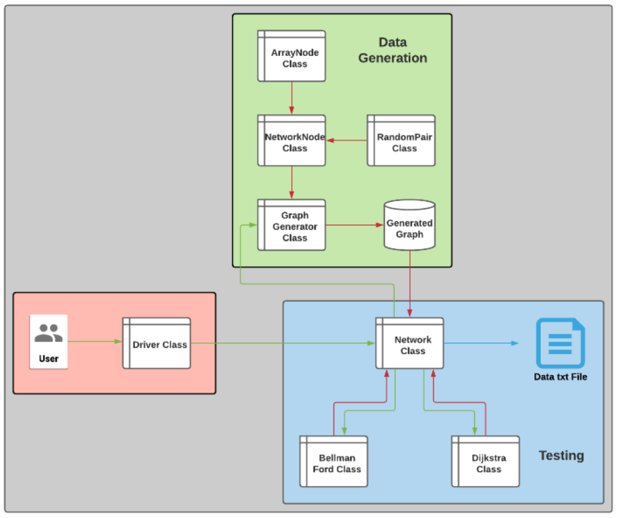

Figure 8: Overall System Architecture Diagram

3.3.1 DATA GENERATION

Data generation is one of the most important portions of this project due its significance within the computational results, as all results are directly reliant on the interpretation of the graph object produced by the data generation code.

Within this logical sphere of the code, the only parameters from an outside source are the relayed input data variables that travel from the driver, into the Network class and finally into the Data Generation sphere's graphGenerator class. This graphGenerator class is the central hub for the graph object's generation, as it handles assembling the specific node objects into a interaction data structure for the Testing Implementation sphere to interact with. The graphGenerator class will then create an ArrayList of NetworkNode objects to represent the graph.

These NetworkNode objects originate from the NetworkNode class which also utilizes an ArrayList of its own to store an object known as an ArrayNode. This class is best described as the logical representation of a node within a network, utilizing its ArrayList as a local routing table which only contains data concerning its neighbors. The NetworkNode class will also handle adding edge connections as well, thereby storing its connections within its local ArrayNode ArrayList data structure. This ArrayNode object is simply a data object utilized for storing neighbor node reference data and specific edge weight values for utilization when reading during the Testing Implementation portion of the code.

After the graph object has been created within the graphGenerator class, it will piece the graph together in a specific order. First, through a linear fashion going one by one through the ArrayList connecting nodes, this guarantees the graph is completely connected and no nodes are left in a disconnected portion of the graph. This means there will be N-1 pre-planned linear connections (With N representing the number of nodes generated within the graph). Once the linear connections have been handled, the graph will then generate a specific number of random connections between each of the nodes, the specific number of random connections would be N // 2 (N divided by 2 and then rounded to the lowest integer). This specific number of random connections allows the program to not reach connection overflow due to each node being restricted to only containing at most four connections per node. Once the random connection generation has complete, the graph object will be ready for Testing Implementation interaction.

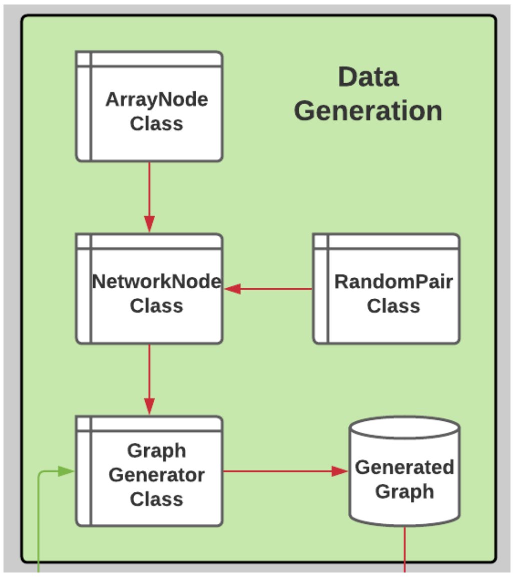

Figure 9: Data Generation Architecture Diagram

3.3.1.1 graphGenerator.java

The graphGenerator java class acts as the central point for graph generation within the program. It specifically generates an ArrayList of NetworkNodes, sets random and linear connections between the NetworkNode objects within the ArrayList in order to create the randomized graph object and handles multiple other functionalities such as checking if random integer numbers meet the specific requirements within the node selection portion of graph generation and a print all node connections function that will send out a complete copy of the generated graph into the IDE's console.

This class contains a constructor, an initialGraph method that sets the linear connections within the graph, a checkRandNumbers method for confirming two selected integer values follow the specific criteria for setting a network connection, the printNodeConnections method for printing out a string value contain structural details of the graph and the class' variable getters/setters.

The class is utilized specifically within the Network class (Testing Implementation) as a method for generating the graph dataset. This class also utilizes the following Java library: java.util.*. Specifically, for random integer value choices during the random connections portion of graph generation.

3.3.1.2 NetworkNode.java

The NetworkNode java class is utilized specifically for storing Node data within the graphGenerator class' ArrayList structure. Although it does not have any dynamic utilization such as graph generation, it does construct the NetworkNode object allowing it to store the node's index number which will be utilized for identifying it within the graph, the current total weight which represents its distance from the initial node (Utilized during testing implementation), the current size of the neighbor array (stored as an integer), the Boolean value indicating whether the node has been visited during an algorithm simulation (Utilized during testing implementation) and a ArrayList of ArrayNode objects for storing the specific NetworkNode's references to its current neighbors within the graph.

The class has an addEdge method which can be utilized to add a new reference within the NetworkNode object's ArrayList which would contain the specific neighbor node's object reference and its edge weight between the NetworkNode object and the current neighbor node object. There is also the checkConnection method which will iterate through the NetworkNode's current neighbor ArrayList to check if a node has already been added to the ArrayList. Finally there is also a toString method that is utilized to list out a NetworkNode object's neighbor ArrayList data into interpretable information for a user and there are the usual variable getters/setters methods.

3.3.1.3 ArrayNode.java

This specific java class is utilized for storing the node reference and edge weight integer value within a node's neighbor ArrayList. It only contains a constructor and its variable getters/setters, allowing it to be utilized for storing specific data concerning the neighbor nodes. Other then that specific utilization within the NetworkNode class, this class is not utilized anywhere else within the environment.

3.3.1.4 randomPair.java

The randomPair java class is utilized for storing only storing two integer values. The conceptual reasoning behind this class, is to allow methods within other classes to return more then one integer value such as during the process of choosing which two nodes will be tested during simulations. It only contains a constructor and its variable getters/setters and is specifically utilized within the following classes: graphGenerator (Data Generation) and Network (Testing Implementation).

3.3.2 TESTING IMPLEMENTATION

Although the Data Generation portion of the project is extremely important due to its possible influence on computational measurements, the Testing Implementation portion is the most dynamic of the logical spheres. As it handles not only the interpretation of measurements, but also the recording of the measurements themselves.

The Testing Implementation portion directly handles the driver's inputted parameters, as its job is to setup the Data Generation's work through the interpretation of the outside parameters. It all begins within the Network class, where the driver directly sends it user input into the class' constructor. This object construction will only create a String reference towards the specific algorithm utilized within this round of testing and the integer value for the number of nodes that were requested within the generated graph. The constructor will generate the graph by calling upon the Data Generation portion and sending the graphGenerator class the integer value for the number of nodes to be generated. Once generation is complete the class has the dataset its needs to compute the requested simulation.

Once the driver then calls upon simulation portion of the Network class, the class will then interpret the previously stored String value in order to decide which algorithm to run within the simulation. Once chosen the class will choose the specific nodes within that simulation to be tested, as well as the number of simulations to be ran. It will then begin recording the amount of time per simulation and recording the overall accuracy ratio of the total number of simulations. During these simulations it will refer to the two algorithm classes Dijkstra and BellmanFord.

Both classes differ based upon their required computational steps, however their testing methodologies remain the same. Each class will construct an object based upon the initial node to start at within the graph and the destination node to reach to within the graph. The classes will then create a reference set (such as an array) to record the distances between each of the nodes and the initial node. The class will then record a start time value, in nano seconds, call upon the specific algorithm's simulation and then record an end time value which will also be in nano seconds. The elapsed time during the simulation is then returned for recording. Accuracy is not handled here as that is handled within the Network class.

Once the simulations have ceased, the evaluation metrics record (computational time and accuracy) will then be recorded into their own references within the Network class. This class will then generate a txt file and write the harvested data as String values into the txt file. Once complete, the file is saved and the program as a whole ends or restarts depending on the driver user interaction.

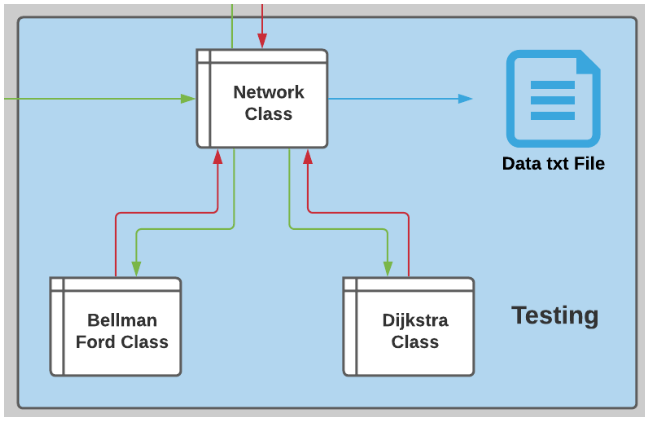

Figure 10: Testing Implementation Architecture Diagram

3.3.2.1 Dijkstra.java

The Dijkstra java class is utilized for running the Dijkstra algorithm's simulation based upon the generated data set and recording the specific computational time that it took to run the simulation. Once called upon, the class will begin constructing a reference to the two nodes that will be utilized within the simulation, specifically the initial node and destination node. These nodes will be pulled from the generated graph based upon the two integer values parameters within the Dijkstra class' constructor. Once both references are created, an integer array is created to represent the amount of distance between each of the nodes and the initial node. This specific array will also represent the exact index of the graph's ArrayList object as well, therefore the index for the node within the graph will also work with its representation within the integer array. Following the initialization of this reference set, where each of the values expect for the initial node's representation are set to the value of -1 (for this project it will logically represent the infinity value), the initial node's distance value is set to 0.

Afterwards, the class utilizes a system command to pull the specific current time in nanoseconds, calls upon the class method algorithmRun and then calls a second pull for the current specific time in nano seconds. After both values have been gathered, the elapsed time is found by subtracting the first nano seconds value from the second value. The class will then print a string value to the system console that describes the shortest distance found between the initial node and the destination node. Once this has occurred and the elapsed time has been found, the class will run through the entire graph resetting each NetworkNode's visited values back to false.

The class contains a constructor for initializing the algorithm run with all the values utilizes during this specific run, a method called checkMinDistance that will return the node index of a NetworkNode's closet neighbor or the next viable node index that the algorithm should visit, the algorithmRun method(s) that recursively handle the Dijkstra algorithm's simulation, and the class' getters/setters.

3.3.2.2 BellmanFord.java

The BellmanFord java class is utilized for handling all Bellman Ford algorithm simulations based upon the generated datasets. This class handles the flow and manipulation of data similarly to Dijkstra's class, with a few exceptions. The main differences being that the BellmanFord class calculates the number of iterations to be run before hand within an integer value, the constructor utilizes an additional loop structure to run multiple iterations and although both of their methods remain the same name wise the algorithmRun methods and the checkMinDistance method is altered to reflect the methodology behind the Bellman Ford algorithm. Both algorithm classes within the Testing Implementation section utilize similar logical structures and data flows except for the before mentioned exceptions.

The class contains a constructor for initializing the algorithm run with all the values utilizes during this specific run, a method called checkMinDistance that will return the node index of a NetworkNode's next viable node index to pursue within the algorithmRun method, the algorithmRun method(s) that recursively handles the Bellman Ford algorithm's simulation, and the class' getters/setters.

3.3.2.3 Network.java

The Network java class is the center point for the Testing Implementation portion of the project. Utilizing a constructor, it creates a Network object that will hold the String value that will reflect which algorithm that is to be simulated, an integer value to reflect the number of nodes that need to be generated and the actual generatorGraph object that is generated within the constructor during the Network call. Afterwards, the simulationRun method can be called within the driver class and begin the computational measurements. The simulationRun method will begin by creating a long array to store long data values, this is utilized to store the nanosecond time values. Afterwards, a conditional will check the current String value within the class to see which algorithm needs to be ran. Either choice will then lead into multiple loop structures. The first loop structure calls upon a randomPair class object to hold two values that are returned from the nodePicker method within the Network class. This loop structure will repeat based upon the number of node pairs that will be tested within each test case. The randomPair object stores the randomly chosen node values that will be utilized within testing, specifically the index of each node as an integer value. Afterwards, both integer values are pull into their own separate integer variables for storage and then a second loop structure is approached. This structure will repeat based upon the number of simulations set for each node pair. Each loop iteration represents a single simulation where the algorithm object is constructed and then the time elapsed for that specific object's time is pulled from the object to be added to a running long variable of total simulation time. Once the second loop structure is completed for the current node pair, an average time is found by dividing the running total of simulation time by the number of simulations ran. This average time value is then stored within the long array. After this action, the first loop repeats again where it chooses another node pair and then repeats the before mentioned computations. After the first loop structure has finished, the simulationRun method will then calculate the accuracy and return all recorded data back to the driver class or into the Network class method utilized for sending data into a txt file. This specific method will take the following data and copy it in a string format into the before mentioned txt file: generated graph structure, computation times for the tested pairs, accuracy ratios for the tested pairs and the specific path found that was utilized to find the shortest path.

This class contains a constructor that generates the graph dataset and stores specific variables utilized within the simulationMethod, a method named simulationMethod that handles the evaluation metrics recording as well as the simulation testing data structures, a method named nodePicker that returns a randomPair object that stores the two random node indexes chosen from the graph that will be utilized within testings and the class' setters/getters.

3.4 REVIEW OF APPROACH

Although research endeavors make up an essential part of society's innovations and theoretical laws, no endeavor is one-hundred percent full proof in its approach towards its research, experimentation, or observations. In this light, it is important for papers to express what currently identified methods or specific factors act as an advantage or disadvantage for a reader when interpreting results. It should be noted that both advantages and disadvantages can change based upon the evolution of a specific industry. Therefore, any disadvantages or advantages listed should be up to scrutiny and interpretation based upon the current environment of an industry.

3.4.1 CURRENT ADVANTAGES

The current identified advantages of this project's approach can be listed as the following: an isolated testing environment, easily re-creatable testing conditions, centralized system of management and easily harvestable data results.

When referring to the isolated environment, this project utilizes a logically separated sphere of code that cannot be easily affected by outside factors. This means, results procured from within that environment are based upon factors that are only handled by the simulations rather then any outside interference. This could be compared to an actual physical testing environment that might see issues such as the following: hardware failure from multiple hardware resources, software architecture conflicts due to differing product manufacturers and even monetary issues from the amount of funding it might take to procure physical equipment.

This approach also utilizes an easily re-creatable testing environment. Meaning factors from within the simulations can be easily altered and recreated through alterations of code rather then physical alterations of hardware which could then require software alterations based upon the factors that have been changed. This advantage allows for a researcher to recreate specific factors or environments through simple alterations in a quick manner, which could lead to a researcher being able to compute more simulations leading to a larger and more cohesive data sets for testing interpretation.

Such re-creatable environments also rely on another advantage of this current approach, which is the centralized system of management. Compared to environments that require tuning and factors changes within multiple devices, this approach allows for the researcher to manage factors from one source which is the code. This allows them to possibly make more changes and test out new cases that would not be possible in a physical environment, such as testing out how Dijkstra's algorithm would handle one million nodes within an environment. So, in this centralized system of management the researcher simply changes the number of nodes to be generated to a value of one million, whereas a researcher in a physical environment would need to physically connect one million nodes which is an impossible endeavor monetary and time wise for a single researcher.

The final advantage to this approach is its easily harvestable data. Lending this ability to the centralized coded environment, data can be feed directly into a specific file or another program for interpretation. Whereas a physical network would run into the possible issues that were outlined before within the isolated environment advantage. This methodology of data harvesting allows for a more reliable and isolated approach to attaining requested data from the simulations.

3.4.2 CURRENT DISADVANTAGES

The current identified disadvantages of this project's approach can be listed as the following: current system does not completely represent a real implementation of a network, possible human coding errors within testing implementations and possible dataset bias from custom generation of datasets.

Although an isolated coded environment does give this project advantages, such as the ones listed under section 3.4.1, it also acts as a double edge sword for the validity of the data attained. This second edge shows itself through the fact that the current tested environment does not consider factors such as ip address management, routing table management and other network related processes that occur in a real environment. Therefore, the results in this test cannot be directly compared to any physical environment counterparts. As other network environmental factors could alter the implementation of the tested algorithms, where it might yield differing results.

This approach also has an issue through the fact that all of its architecture and testing environments are coded by a single programmer. Therefore, there could arise unknown results or issues that might affect the data due to human oversight. This hurts the validity of the data since such issues might lead into data bias towards a specific result, simple from an issue such as an incorrectly implemented algorithmic step in the simulations.

Another issue that is like the previously mentioned disadvantage, is that all datasets are custom generated and not gather from an outside resource. Therefore, there could be unknown data bias due to the custom nature of the researcher generating the structures rather then gathering them from a outside reliable source. This disadvantage also possibly hurt the validity of the results and should taken into consideration when a reader interprets the data.

The final disadvantage that could affect the results the most, is that the datasets might be too limited. Or that the datasets are too simplistic to see a discernable change in data during analysis. This could lead to static results and/or misleading analysis of the data due to its limited scope.

4. IMPLEMENTATION

4.1 HARDWARE ENVIRONMENT

The hardware environment implementation of this project is not extremely extensive due to the nature of the testing being primarily altered and developed within the software environment. However, there is a specific hardware setup being used to act as a control variable when comparing results.

This specific control variable is utilized to not allow any specific hardware component differences between setups alter the data results from each simulation. This specific hardware environment is a custom-built desktop computer, which houses the following components:

- Intel Core I7 10700K x64 Processor (4.5GHz Clock Speed)
- MSI Z490 MPG Gaming Plus Motherboard
- 16GB, Dual Channel, DDR4, G.Skill Ripjaws V Series Ram (3200MHz Clock Speed)
- ASUS TUF Gaming RTX 3060, 12GB VRAM, GDDR6, Graphics Card
- EVGA SuperNova G3 650W, 80+ Gold Efficiency ATX Power Supply

The original use case of this hardware setup was for personal computing; however, it is the most capable machine currently available for utilization within this project. Therefore, it was chosen as the main testing unit for all simulations and data harvesting recorded within the project's results.

4.2 SOFTWARE ENVIRONMENT

The software environment utilized within this project is an isolated coded Java environment, developed in the Eclipse IDE, which utilizes the Java.Util.* library for specific functions that are required for successful program functionality.

The Java environment is coded within the latest Java JDK package, noted as JDK 16. This coding language was chosen due to its efficiency and reliability as a programming language, as well as its extensive utilities library which offers a plethora of tools that could have been used within the project's code. However, only one specific tool within this library was utilized. Specifically, the random class where its use case was primarily for choosing random integers values between 0 and a specific integer variable. This variable changed depending on the use case, however it was utilized within both the graphGenerator and Network classes.

The Eclipse IDE was utilized due to its extensive debugging environment that allow for an in-depth analysis of specific processes. Specifically, for debugging endeavors handling the recursive nature of the algorithm runs and the graph generation. The IDE also allowed another level of isolation from the OS, since all processes were handled within the IDE's virtualized console rather then the hardware environment's system console. This makes running and altering the application much easier during testing, implementation and debugging scenarios.

5. EXPERIMENTAL SETUP

In order to find out which algorithm reacts better within a simulated scalable environment, there needs to be a few definitions explained beforehand. First, the phrase "reacts better" means that the algorithm performs a specific task or combined metric relatively better than the other within a specific environment. In this case, both algorithms are tested based upon their computational speed per simulation and ratio of accuracy per simulation round. This simulation round is defined as the total number of simulations ran per selected node pair. Finally, the simulated scalable environment refers to the generated graph environment and its potential to grow in number of connected nodes. For this experiment, each algorithm will be tested across an ever-growing number of nodes within each graph data set, which is why the phrases reacts better within a simulated scalable environment are paired together. Now that both of those definitions are explained, the experimental setup can also be defined.

In this experimental setup, both algorithms will be feed the same graphs per tested dataset as well as the same selected nodes. When each "test" begins, the dataset graph will be generated, sent into the network class, and then each algorithm will be sending the graph object. Once an algorithm has received this object and begun computation, its evaluation metrics will be measured. Specifically, the start and end times of the algorithm's computation will be taken and utilized to calculate the elapsed time. Then after the simulation round has finished, the elapsed time for each algorithm simulation will be added into a total elapsed time value and divided by number of total simulation rans. This will give the program the average computational speed of the selected node pair. Once that value has been calculated, then the accuracy of the selected node pair will be found. This will be found based upon a ratio of correct SPP solutions to total number of simulations ran. With each value being recorded on a per selected node pair basis.

This specific method of testing will occur for each algorithm within each set test case/dataset. Which is defined in section 5.1.

The evaluation metrics within this experiment are the defined as the following:

- Average computational speed per simulation round (Measured in nano seconds)
- Ratio of correctly selected solutions compared to total number of simulations ran, per simulation round (Measured in % ratio)

5.1 DATASET SETUP

The following datasets are utilized within this experiment to test how each algorithm reacts to an ever-growing environment.

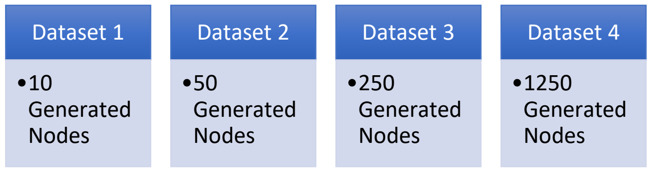

Figure 11: All datasets utilized within the experiment

The datasets follow an exponentially growing trendline, with each dataset (except for the first) increasing to around 500% of the dataset before it.

5.1.1 DATASET GENERATION RULES

Each generated dataset will follow specific rules concerning their generation, this is done to keep the randomized generations to at least a realistic like standard rather than having outlier possibilities within the generated graph data. These outliers could be a set of 1250 nodes having a single node connected to all 1250 nodes or a graph with a whole sphere of nodes not connected to the other side of the graph, leading to issues during testing. Each of the generated rules are listed below:

- Each edge weight between two nodes will be an integer value ranging between 1 and 100.

NOTE: This makes all weights within the graph positive, as Dijkstra's algorithm cannot handle negative weights)

- Each node will have between one and four connections within the graph.
- Each of the node's neighbors will not be redundant. Meaning a node cannot be connected to the same node more than once.
- Each graph will have initially N (number of nodes connections - 1) connections between each of the nodes. This connection will be made in a linear fashion. For example, node 0 to node 1 to node 2 etc.

NOTE: This guarantees that the graph is fully connected and will not leave an isolated sphere or disconnected node after generation.

- Each graph will also have M (number of nodes/2) random connections between the nodes.

5.1.2 SIMULATION SETUP

The simulation setup within the experiment will be centered around each node pair selected. Meaning that all data will be either measured or averaged around each node pair. This is done to condense the data into more interpretable means.

During the testing of a generated data set, three node pairs will be selected per data set. Within each node pair selected there will be a total of fifty simulations computed per pair tested. Within the experiment these number of simulations per node pair will be designated as a simulation round.

1 simulation round = 50 computation runs within a selected node pair

Each simulation round will also have a measured accuracy ratio. That will be based around that specific simulation round rather then individual simulations, as the ratio needs total data to compare to specific data. This accuracy ratio will follow the below formula:

$$A\ (\%) = \frac{C\ (Correct\ number\ of\ simulation\ solutions)}{T\ (Total\ number\ of\ simulations)}$$

A correct solution will be regarded as a solution that is in fact, the shortest path distance from the initial node to the destination node within a simulation round.

5.1.3 DATA HARVASTING

After all computations are complete, they will be written as string values into a txt file to be interpreted and stored for later reference. This data harvesting will be following the specific format within the txt file per simulation round.

Selected Node Pair: Node 1 – Node 2

Algorithm Ran: "Selected Algorithm"

Average Time Elapsed: "time elapsed ns"

Accuracy Ratio: "accuracy ratio %"

Each of simulation round will be separated per algorithm and will have a section as the one listed above. Data will be separated based upon algorithm and node pair selected.

Once data has been successfully harvested into the txt file within the correct string format, each data point will be then be transferred onto a scatter plot graph so that trends and other specific information can be interpreted based upon the data's value.

6. RESULT ANALYSIS

It should be noted that all collected simulation data compared Dijkstra's algorithm and the Bellman Ford Algorithm utilizing the same node pairs within the same graph. For example, simulation 1 results had both algorithms solve the SPP for the same node pair (In this case, say nodes 1 and 5) within the same generated graph. This was done to ensure a better comparison of data during analysis.

6.1 COLLECTED DATA

6.1.1 Dataset 1 (10 Nodes)

Simulation Round #	Dijkstra (ns)	Bellman Ford (ns)
1	33354	52708
2	7022	40780

3	4328	29194
Overall Accuracy (%)	100	100

Figure 12: Computational time and Accuracy ratio for both algorithms for Dataset 1 test case. Test case 1 included a generated graph of 10 nodes.

6.1.2 *Dataset 2 (50 Nodes)*

Simulation Round #	**Dijkstra (ns)**	**Bellman Ford (ns)**
1	21944	942058
2	5970	288506
3	6676	388760
Overall Accuracy (%)	100	100

Figure 13: Computational time and Accuracy ratio for both algorithms for Dataset 2 test case. Test case 2 included a generated graph of 50 nodes.

6.1.3 *Dataset 3 (250 Nodes)*

Simulation Round #	**Dijkstra (ns)**	**Bellman Ford (ns)**
1	107630	5429044
2	381658	3876296
3	42160	3022016
Overall Accuracy (%)	100	100

Figure 14: Computational time and Accuracy ratio for both algorithms for Dataset 3 test case. Test case 3 included a generated graph of 250 nodes.

6.1.4 *Dataset 4 (1250 Nodes)*

Simulation Round #	**Dijkstra (ns)**	**Bellman Ford (ns)**
1	134144	40317232
2	1593050	35659362
3	**2638**	47752048
Overall Accuracy (%)	100	100

Figure 15: Computational time and Accuracy ratio for both algorithms for Dataset 4 test case. Test case 4 included a generated graph of 1250 nodes.

6.2 DIJKSTRA'S ANALYSIS

The results for Dijkstra's algorithm did not follow the expected logarithmic path that was previously predicted to be seen during its tests. This predication is based on Dijkstra's Big O notation time complexity formula where E represent the set of all edges and V represents the set of all vertices which in this case is nodes. The Big O notation is as follows:

$$O(E * \log(V))$$

Based upon that, it was predicted the algorithm would see a logarithmic rise in computation time as the number of nodes increased within the generated graphs. However, this was not the case as shown by Figure 16.

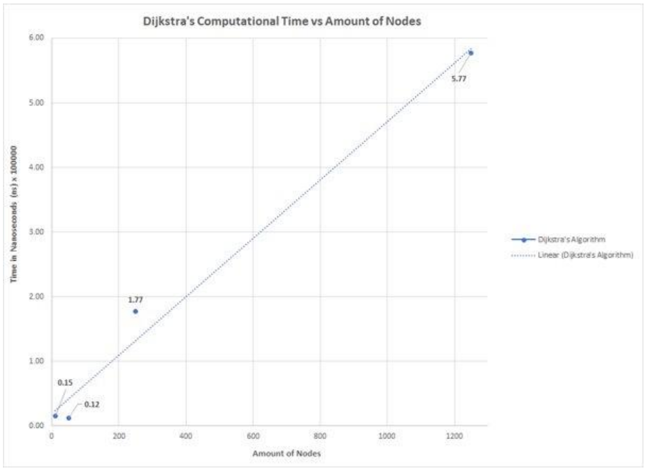

Figure 16: This figure depicts a visual representation of the Dijkstra Algorithm's performance, specifically comparing its computational time versus the number of nodes that were generated within the specific computation.

The figure above depicts a linear growth in computational time as the number of nodes increased from 10 all the way to 1250. The only exception to this linear growth was the average computational speed for Dijkstra's Test Case 2 results, which saw an outlier case in Simulations 2 and 3 which gave the total Test Case average a noticeable drop-in computational time. This outlier event is best explained by Simulations 2 and 3's random node pairs being chosen was extremely close to one another due to either random edge generation or due to the initial linear connection. Another outlier case was found in Test Case 4 Simulation 3 for Dijkstra's algorithm; however, this outlier can be explained through the same means as the earlier found outlier events, and it seemed to not affect the total average of the computation time for the case by an extreme amount. It can also be noted that the accuracy ratio for every test case utilized through Dijkstra's algorithm was found to be 100% accuracy. Although that was not expected, it can be assumed that this might be due to multiple factors to the approach of this experiment. Specifically concerning the test cases and generated graphs.

6.3 BELLMAN FORD ANALYSIS

The results for Bellman Ford algorithm did follow the expected exponential path that was previously predicted. This predication, like Dijkstra's algorithm, was based on the Bellman Ford algorithm's Big O notation time complexity formula where E represent the set of all edges and V represents the set of all vertices which in this case is nodes. The Big O notation is as follows:

$$O(E * V)$$

Therefore, based on the Big O notation formula, it was predicted the algorithm would see an exponential growth in computation time as the number of nodes increased within the generated graphs, which is supported by the visualized data within Figure 17.

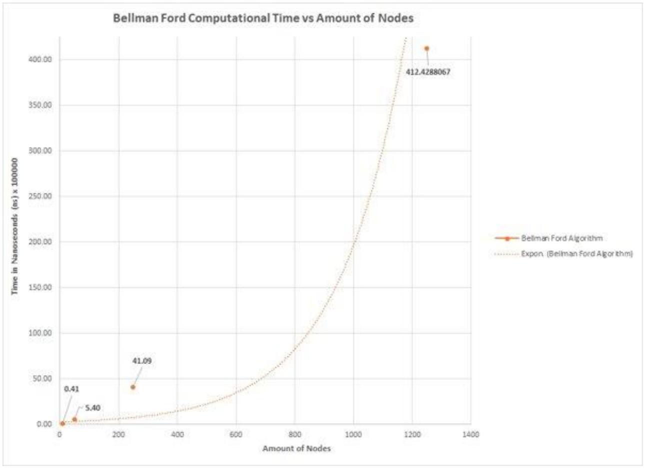

Figure 17: This figure depicts a visual representation of the Bellman Ford Algorithm's performance, specifically comparing its computational time versus the number of nodes that were generated within the specific computation.

Figure 17 shows the computational time within the Bellman Ford algorithm's testing followed an exponential growth in time as the number of nodes grew from 10 to 1250. Unlike Dijkstra's algorithm, no outlier events occurred within the data. It can be noted through that the average Computational time for Test Cases 1 to 3 seemed to be extremely close to one another until Test Case 4 was computed. So much so, that initially it seemed Dijkstra's algorithm might even follow a linear path of progression compared to the final exponential growth rate. The specific nature of this increase comes down to the specific methodology of the Bellman Ford algorithm which will be discussed further in section 6.4. The Bellman Ford algorithm also saw a similar statistic to Dijkstra's algorithm concerning its static nature in accuracy ratio measurements. Where each ratio measured out to be 100% across all the test cases. The reasoning for this static measurement is the exact same as the previous reasoning described within section 6.2, specifically that the initial approach in experiment might be affecting the collected data.

6.4 COMPARISON ANALYSIS

Once results for both algorithms have been obtained there is a clear magnitude in difference between both when comparing their computational time growth within each test case. As described before, Dijkstra's algorithm seemed to follow a linear growth over time whereas the Bellman Ford algorithm followed an exponential growth over time. However, even for the initial Test Case 1 averages the Bellman Ford algorithm saw a much higher time compared to Dijkstra's algorithm. Which can be shown in the table below for comparison:

	Case 1	**Case 2**	**Case 3**	**Case 4**
Dijkstra Average (ns)	14911.3	11530.0	177149.3	576610.7
Bellman Average (ns)	40894.0	539774.7	4109118.7	41242880.7
Difference in Averages (ns)	*25982.7*	*528244.7*	*3931969.4*	*40666270.0*

Figure 18: This table shows the difference between each Test Cases' Average Computation time for each Algorithm.

As shown by Figure 18, the difference in test case average computation times between Dijkstra's algorithm and the Bellman Ford algorithm see a difference at least 2 times Dijkstra's time from test case 1 to a maximum of 70 times. This major increase in computational time shows without a doubt that in conditions where computational speed is a major factor, Dijkstra's algorithm is the clear victor. This can further be shown visually through Figure 19 where the normally linear growth of Dijkstra's algorithm is almost depicted as a static measurement, due to the vase difference in computation time when directly compared to the Bellman Ford algorithm's computational time.

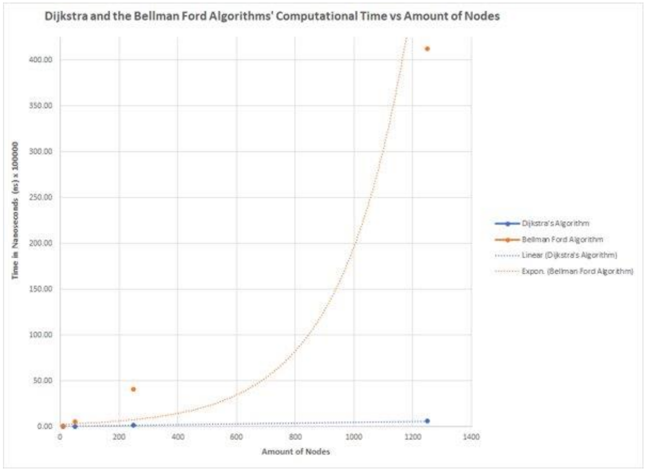

Figure 19: This figure depicts the visual comparison of both Figures 16 and 17's results, to show the specific magnitude of difference between both algorithm's computation time performances.

The magnitudes of difference between both algorithms might come be due to their difference in methodologies when it comes to searching for the SPP solution. Where Dijkstra completely stops its computation once the first most optimal solution has been found, the Bellman Ford algorithm will go another iteration until the most optimal solution has been found or all its iterations have been exhausted. This difference would explain why it takes the Bellman Ford algorithm so much longer to compute a solution, even for smaller test cases as it is looking at the overall picture rather then the most optimal solution's picture. As discussed before in section 3.2 this can be attributed to Bellman Ford's ability to correct the shortest path after each iteration, for situations where the first solution is not the most optimal. This can be seen primarily for graphs with negative weights, that require multiple iterations to find the correct SPP solution since their overall paths can change based on each iterations' calculations. Therefore, based upon the data shown, due to Bellman Ford's dynamic approach that lends credence to focusing on the graph rather than local optimal choices, it is severely slower than Dijkstra's algorithm when a network is scaled up in sized. Meaning, that for scalable networks which primarily focus on speed as a critical functionality, Dijkstra's algorithm is a clear choice for routing its SPPs.

In the next evaluation method tested, accuracy showed a much more mundane set of data points that pointed to an error within this research project's approach that was mentioned in section 3.4.2. The data mentioned can be shown visually below in Figure 20:

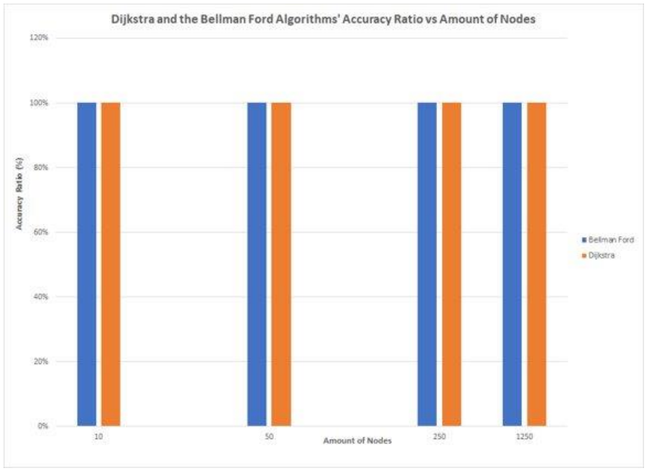

Figure 20: This figure depicts the visual comparison of both algorithm's accuracy ratios based on each test case (1 through 4).

As shown, the accuracy ratio for each Test Case (1 to 4) and each tested algorithm was 100%. Meaning that the measurement was static across all tested cases and algorithms. Which shows that for testing more simplistic graphs that only consider integer weighted edges with no other factors involved, other then the tested algorithms, accuracy ratios might be a redundant measure as they will most likely not vary much or at all. This originates within section 3.4.2, which exposes a possible flaw within the approach of this research endeavor that explains the tested datasets might be too limiting and do not account for enough factors. Possible solutions to this issue would be larger datasets that are greater then 1250 nodes and multiple factors when calculating edge weights such as data traffic per node. However, in the end the data can still be utilized to validate the computational times of each algorithm which lends credence that the data is not completely redundant and can still be utilized to validate other results rather than directly impact the final analysis of the data.

7. CONCLUSION

Based upon the analysis it can be noted that due to the Bellman Ford algorithm's exponential computational time growth when exposed to scalable environments, Dijkstra's algorithm's linear growth as well as its degrees of magnitude lower computational times, would be much better suited for the environment so long as the environment's goal is faster SPP problems solved, and the environment only involves positively weighted edges.

This specific project's main contribution to its research field is its ability to act as a steppingstone for other research endeavors that aim to answer the questions left open and answered by this project. Specifically, it gives credence to the idea that Dijkstra's algorithm's greedy approach provides it an advantage over the Bellman Ford algorithm due to its optimal methodology, thereby giving individuals who need to choose one or the other within their own environment a specific answer that can help fuel their decision on which to utilize to solve their routing problem. The steppingstone portion is due to the possible experiments that could be conducted based upon the resulting data. Such as an experiment that tries to solve the approach issue listed in section 3.4.2, specifically what if larger and more datasets were utilized within testing, would it yield a different result in accuracy and/or the growth rate of either algorithm? Or another experiment, where utilizing the data procured from the initial experiments within this paper, they could compare the Bellman Ford algorithm with a counter part dataset that utilizes negative weights, where they can compare the performance of each within 2 different kinds of environments. Or it could also give data to other looking towards designing new algorithms for specific situations based upon previous design [6], data from this project could help decide which algorithms they base their research on due to its computational speed and accuracy.

Although Dijkstra's algorithm was clearly the winner within this research project for scalable environments, both algorithms also have their own advantages and disadvantages that must be mentioned. From testing, Dijkstra's algorithm excels at computing solutions at a very fast rate, and it can work with more limited data sets since it only needs the local neighbors of each portion it visits when searching for the SPP solution, which might also mean it utilizes less hardware resources as well. The Bellman Ford algorithm excels in environments that have factors that are constantly changing and need to be recalculated [8], as well as environments that have negative edge weights which also need multiple iterations to account for when finding an SPP solution. Where Dijkstra's algorithm falls are the types of environments it can be utilized within due to its optimal approach. This approach limits it to only positively weighted and static environments since it only utilizes a single iteration and only looks at its local most optimal options rather then the entire graph. This means that when utilized within a negatively weighted graph which requires multiple iterations, it cannot be an accurate answer. The Bellman Ford algorithm's largest disadvantage is its computational time, as shown through this research project, as its graphs increase in size its computational time skyrockets into much larger values then its counterpart Dijkstra's algorithm. So, in environments that factor in speed as a critical point, the Bellman Ford algorithm would be a poor choice.

This project takes a unique approach within its approach, specifically that it utilizes its own custom generated graph data structures rather then utilizing a preset structure from an online source or from Java's utility package. This gives the project and other projects a new set of testing infrastructure to build off and utilize for different experiments either related or unrelated to the current topic. This approach allows more freedom within the project to alter environmental factors and algorithm factors when testing datasets due to its custom nature, since those that work directly to develop its infrastructure have firsthand experience within its systems. Allowing them to take advantage of its possible capabilities or identify limitations.

Future implementations of this experiment are possible and could lead to new exciting utilizations of both the Bellman Ford and Dijkstra's algorithms. For this project however, it can be concluded based on section 6, that although both algorithms can provide accurate solutions within environments between the size of 10 and 1250 nodes, Dijkstra's algorithm is a much better choice for scalable environments due to its greedy methodology that allows it to have magnitudes faster computational times.

8. REFERENCES

[1] Muhammad Akram, Amna Habib and Jose Carlos R. Alcantud. 2020. An optimization study based on Dijkstra algorithm for a network with trapezoidal picture fuzzy numbers. *Neural Computing and Applications 33,* (June 2020), 1329-1342. DOI: https://doi.org/10.1007/s00521-020-05034-y

[2] Samah W.G AbuSalim, Rosziati Ibrahim, Mohd Zainuri Saringat, Sapiee Jamel and Jahari Abdul Wahab. 2020. Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization. *IOP Conf. Series: Materials Science and Engineering 917*, (April 2020). DOI: https://doi.org/10.1088/1757-899X/917/1/012077

[3] Maher Helaoui. 2017. Extended Shortest Path Problem - Generalized Dijkstra-Moore and Bellman-Ford Algorithms. *Proceedings of the 6th International Conference on Operations Research and Enterprise Systems 1*, (2017), 306-313. DOI: https://doi.org/10.5220/0006145303060313

[4] Yefim Dinitz and Rotem Itzhak. 2017. Hybrid Bellman-Ford-Dijkstra Algorithm. *Journal of Discrete Algorithms 42,* (January 2017), 35-44. DOI: https://doi.org/10.1016/j.jda.2017.01.00**1**

[5] Mani Parimala, Said Broumi, Karthikeyan Prakash and Selcuk Topal. 2021. Bellman–Ford algorithm for solving shortest path problem of a network under picture fuzzy environment. *Complex & Intelligent Systems 7*, 3 (June 2021). DOI: https://doi.org/10.1007/s40747-021-00430-w

[6] Yuhong Sheng and Yuan Gao. 2016. Shortest Path Problem of Uncertain Random Network. *Computers & Industrial Engineering 99*, 97-105 (Sep 2016). DOI: https://doi.org/10.1016/j.cie.2016.07.011

[7] Alireza Khani and Stephen D. Boyles. 2015. An Exact Algorithm for the Mean-Standard Deviation Shortest Path Problem. *Transportation Research Part B: Methodological 81*, 252-266 (Nov 2015). DOI: https://doi.org/10.1016/j.trb.2015.04.002

[8] Yuanqiu Mo and Lanlin Yu. 2021. A Lyapunov Analysis of the Continuous-Time Adaptive Bellman-Ford Algorithm. *Systems & Control Letters 157*, 105045 (Nov 2021). DOI: https://doi.org/10.1016/j.sysconle.2021.105045