Performance Evaluation between the Bellman Ford Algorithm and Dijkstra's Algorithm within a Scalable Simulated Environment

-

Final Presentation

UNIVERSITY OF SOUTH CAROLINA UPSTATE – DR. ZHONG
– U599 SENIOR SEMINAR – TALLON MCABEE

Introduction

- Every day we use networks and routing protocols through our smart devices, gaming setups and desktop devices in order to transfer information across vase distances. The efficiency within these interchanges of data heavily impact the performance and experience a user will have, examples include:
    - **Remote gaming (Such as Xcloud, Google Stadia, etc)**
    - **Latency when accessing data on a remote server**
    - **Mobile Wifi Networks**
    - **Etc**
- In order to find the most efficent/shortest path in between two nodes within these networks, network protocols utilize different routing algorithms that take in specific factors in order to calculate the shortest distance between two nodes on a network. In this case, we will be looking at two very popular routing protocol algorithms: Dijkstra's Algorithm and the Bellman Ford Algorithm.
- My project is to **compare the performance metrics of Dijkstra's algorithm and the Bellman Ford algorithm witin a scalable network/graph of nodes in a simulated environment** in order to test their performance in an ever-growing network of nodes with a variety of distances in between each node. This simulation environment will include a custom generated graph dataset, in which each algorithm will be tested, and tested data will be recorded into a txt file for interpretation.

Dijkstra's Algorithm Summary

Dijkstra's Algorithm is a greedy algorithm that can find the shortest path between a specified starting vertex or node within a graph to every other node within that graph.

What is a greedy algorithm?

- A greedy algorithm is an algorithm that makes the most optimal choice at each step of the process in order to find the most optimal solution
- **I.E** Choosing the shortest distance between the node and next node during each algorithmic step

This algorithm can be utilized within a graph of vertices with positively weighted edges between each vertex.

- Dijkstra's algorithm DOES NOT work within a graph that includes negatively weighted edges.

Time Complexity:

- If represented by a graph G(V, E) where E = set of all Edges and V = set of all Vertices within the graph G, the Big O notation representation of Dijkstra's algorithm would be ***O(E log(V))***

Dijkstra's Algorithm Steps

STEP 1: Set initial node's distance to 0 and set all other nodes within the graph's distance to infinity

STEP 2: Create a set containing a reference to all the nodes within the graph for tracking visited nodes, when the algorithm begins this set will be empty

STEP 3: Now check the distance between each of the current node or initial node's neighbors (If you are at the beginning of the algorithm) and consider their distances from the **initial node**. If the distance is lower than their original distance from the initial node, update the node's distance with the new value.

STEP 4: Once each neighbor has been checked, the current node should be marked as visited within the set.

STEP 5: Choose the node with the smallest distance as the next "current node" and repeat steps 3 to 4.

- This next node should be unvisited.
- You continue this algorithm until the desired node has been found!

Bellman Ford Algorithm Summary

The Bellman Ford Algorithm is an algorithm that utilizes the Dynamic Programming approach in order to find the shortest path between a specified starting vertex or node within a graph to every other node within that graph.

What is the Dynamic Programming approach?

- This approach utilizes the methodology of separating the larger problem or code into smaller subsections, where they are memorized and can be reused for similar or the same problems. This approach works to optimize the solution by utilizing the past results of smaller subsections within the computation of larger solutions.
- **I.E** The computation of Fibonacci numbers

This algorithm CAN be utilized within a graph of negatively weighted edges

Time Complexity:

- If represented by a graph G(V, E) where E = set of all Edges and V = set of all Vertices within the graph G, the Big O notation representation of the Bellman Ford algorithm would be ***O(V*E)***

Bellman Ford Algorithm Steps

STEP 1: Set initial node's distance to 0 and set all other nodes within the graph's distance to infinity

STEP 2: Now check the distance between each of the current node or initial node's neighbors (If you are at the beginning of the algorithm) and consider their distances from the **initial node**. If the distance is lower than their original distance from the initial node, update the node's distance with the new value.

STEP 3: Choose the node with the smallest distance as the next "current node" and repeat **STEPS 2 to 3** until all nodes within the graph have been visited!

STEP 4: Restart **STEP 2** at the initial node (Keeping the same distance values) and repeat **STEPS 2 to 3** for n-1 iterations. (n = # of nodes within the graph) OR until you reach an iteration where none of the graph's node distance values are updated.

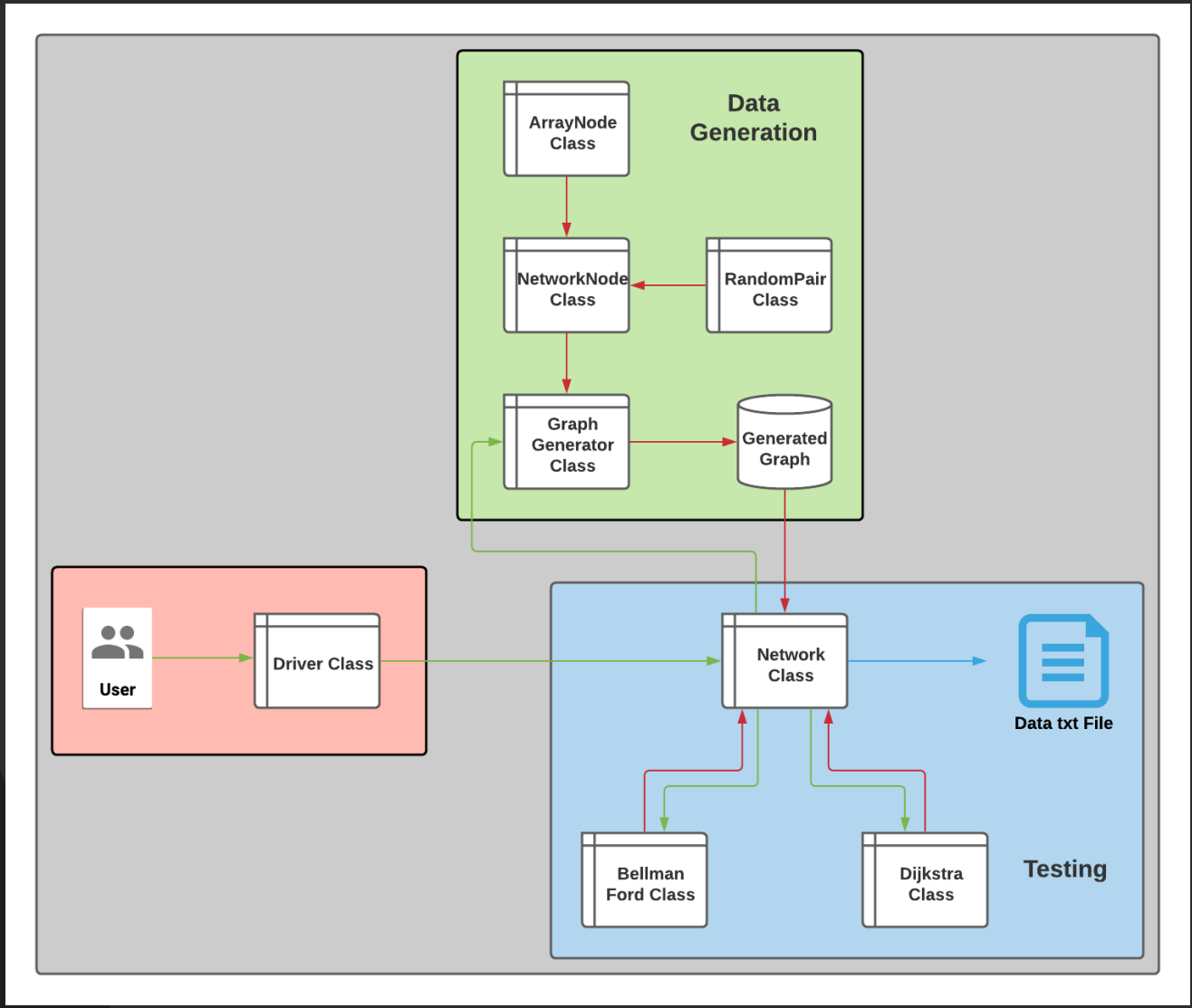
Current Advantages of Approach

1. Isolated environment compared to a physical network where outside factors will be less of an issue during testing/implementation (**I.E** other factors such as hardware failure, software conflicts between two pieces of hardware, **ETC**).
2. Easy recreation of specific testing environments (Makes reoccurring testing much easier, allowing for more possible simulation runs).
3. Centralized system of management (**I.E** makes factors and specific variables much easier to change due to the centralized nature of the testing environment).
4. Easier method of data harvesting from tests (Data will be instantly feed into a separate file straight from simulation).

Current Disadvantages of Approach

1. System does not include all possible factors of a physical network implementation might face (**I.E** conflicting product architectures, IP address management, routing table management, network collisions, **ETC**).
2. Possible coding errors that may arise from human error that could alter testing implementations.
3. Possible Dataset bias from generating my own Data that might affect the data being feed into the testing implementations.
4. Datasets might too limiting and do not account for enough factors which might lead to static results.

System Architecture Diagram



Implementation Setup

- Primary Programming Language: Java 16
- Software and Hardware Testing environments
  - Software environment:
    - **OS:** Windows 10 Pro, Version 10.0.19043 Build 19043, x64
    - **IDE:** Eclipse Version 2021-06 (4.20.0)
    - **Java Libraries Utilized:** java.util.*
  - Hardware Environment:
    - **Processor:** (Intel) Core I7 10700k (Overclocked to 4.5GHz)
    - **Motherboard:** (Intel) MSI MPG GAMING PLUS, Z490, LGA 1200
    - **RAM:** G.SKILL Ripjaws V Series, 16GB (Dual-Channel) DDR4, (Clocked to 3200MHz)
    - **Graphics Card:** ASUS, GeForce RTX 3060 TUF Gaming, 12GB GDDR6, Dual-Fan
    - **Power Supply:** EVGA SuperNOVA, 650W, G3, 80+ GOLD Efficiency

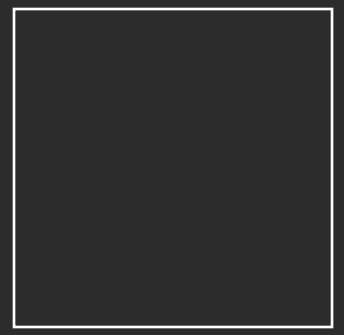
Experimental Dataset

Test Case 1	Test Case 2	Test Case 3	Test Case 4
Graph of 10 Nodes	Graph of 50 Nodes	Graph of 250 Nodes	Graph of 1250 Nodes

Generated Dataset Rules:

- Each edge weight in-between 2 nodes will have a weight in between 1-100 (Stored as an INT value)
- Each node with have in-between 1-4 connections to another node, with 1-2 connections matching the node next to it within the index (Node 6 is guaranteed to be connected to nodes 5 and 7)
- The graph is initially generated to have a linear connection across the entire graph that includes *number of nodes* connections (Node 0 – Node 1 – Node 2 ….). This is meant to keep the graph completely connected with no chance of disconnected nodes or isolated loop structures.
- After initial generation, the graph will then add another (*number of nodes/2*) extra random connections between randomly chosen nodes
  - These extra connections **do not include**:
    - Redundant Connections (Node 0 – Node 0)
    - Repeating Connections (Node 1 – Node 5 and Node 1 – Node 5)

Evaluation Metrics

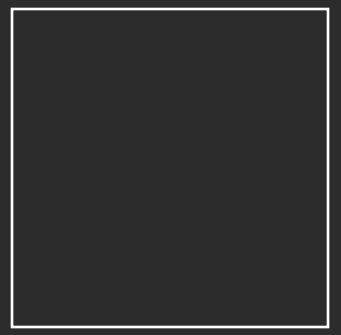


Computational Speed:

Measured in Nanoseconds (10^-9 seconds)

Depicted measurements will be based on total simulation round averages

- Test Case Average = 3 Simulation Round Average = 150 Total Separate Simulations Average



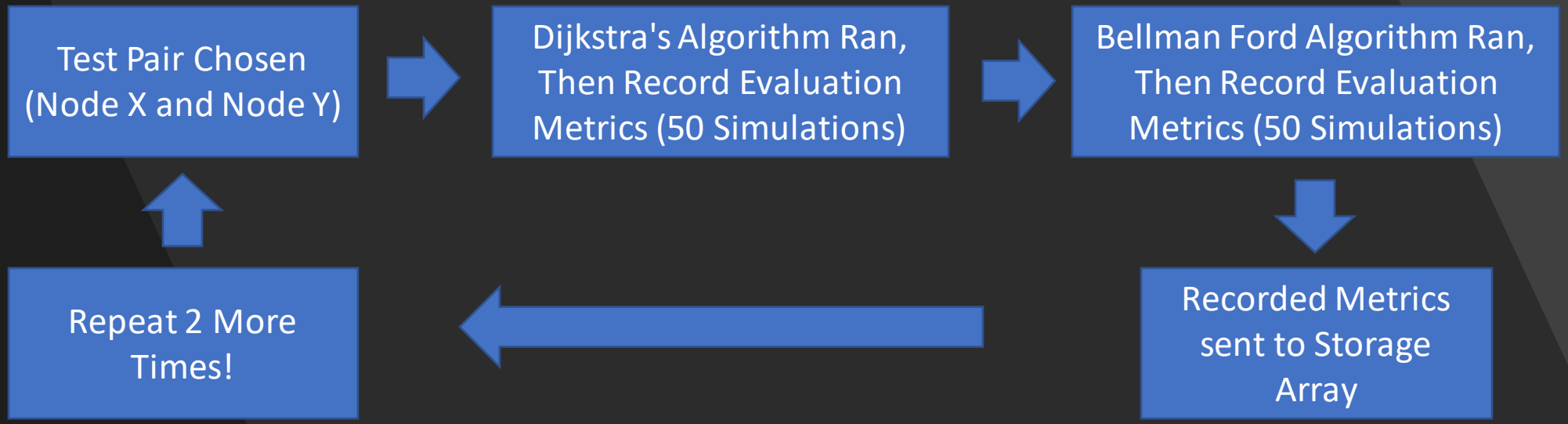
Computational Accuracy:

Measured in % ratio

Ratio: *Number of Correct Runs/Number of Total Runs*

- This ratio will be on a Test Case by Test Case Basis

Experimental Setup



The experiment will be setup in the following steps:

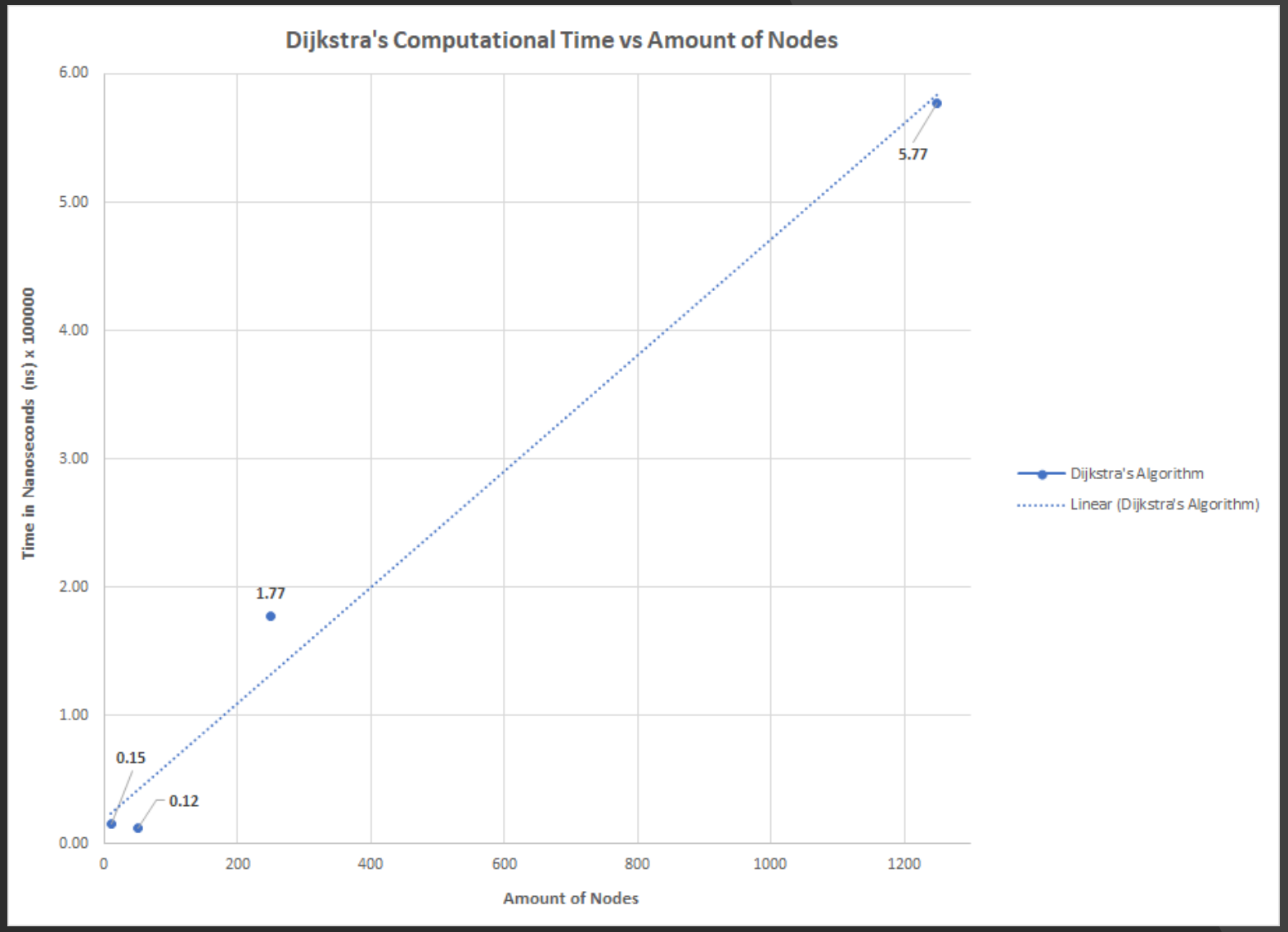
1. **2 Node pairs will be chosen at random** from the generated graph
2. E**ach algorithm will be running separately utilizing the same node pair and graph**, once complete both algorithm's evaluation metrics will be recorded. **Each algorithm will run 50 simulations** which is considered in total to be **1 simulation round**.
3. Total metrics will then be recorded into its corresponding array/storage structure.
4. Afterwards, the same steps will be **repeated 2 more times for the specific Test Case.**
5. **Then the entire process will be repeated for each Test Case**

Overall Dataset Collection

Dijkstra:	Test Case 1	Test Case 2	Test Case 3	Test Case 4
	33354	21944	107630	134144
	7022	5970	381658	1593050
	4358	6676	42160	2638
Averages:	14911.33	11530.00	177149.33	576610.67
Accuracy:	100%	100%	100%	100%
Bellman:	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**
	52708	942058	5429044	40317232
	40780	288506	3876296	35659362
	29194	388760	3022016	47752048
Averages:	40894.00	539774.67	4109118.67	41242880.67
Accuracy:	100%	100%	100%	100%
Nodes:	**10**	**50**	**250**	**1250**

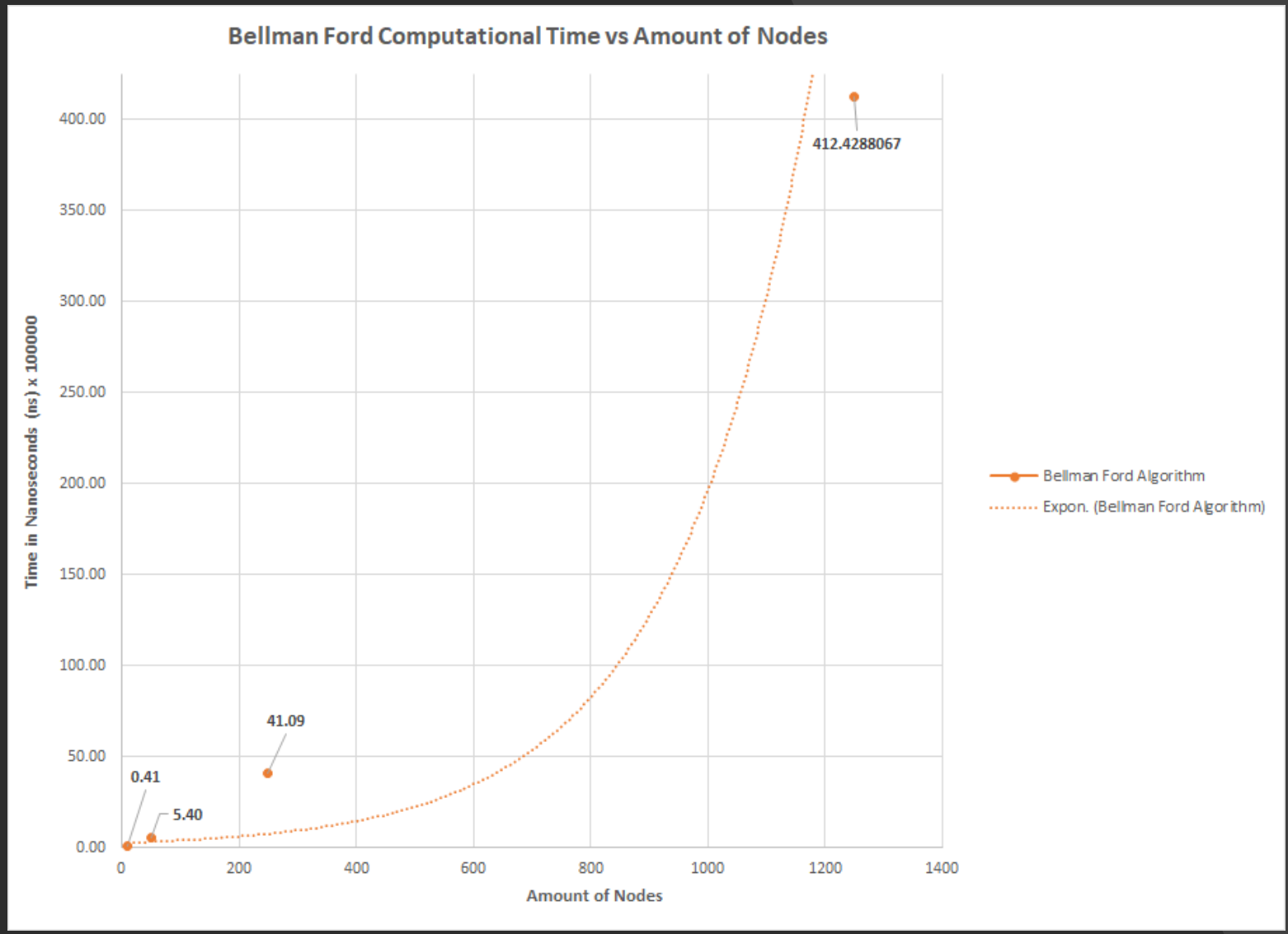
Results Analysis: Dijkstra's Algorithm

- There seems to be **Linear growth** in time taken to compute the shortest path as the number of nodes grew.
- A small dip in computational time for test case 2, however this might be due to outlier event data. This would require more interpretation of the generated graph to find out.
- Also, I expected to see a logarithmic growth in computational speed, however this breaking of expectations might be due to not enough test cases being utilized.



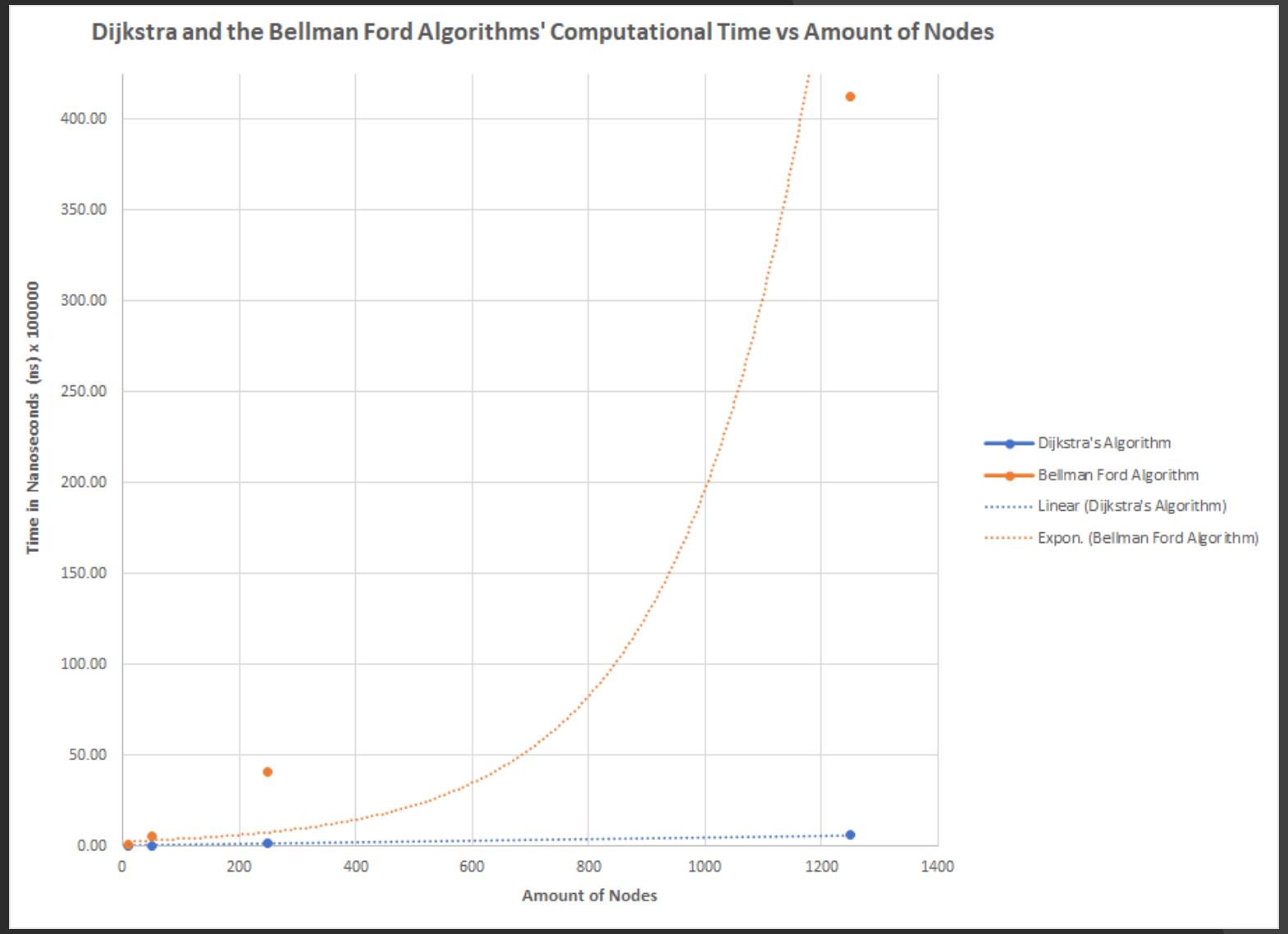
Results Analysis: Bellman Ford Algorithm

- **Exponential growth** in time taken to compute the shortest path as the number of nodes grew
- Test Cases 1-3's Computational Speed was closer grouped towards one another until Test Case 4's exponential increase in time.
- This exponential growth did match my initial expectations, as I expected the computational time to increase dramatically with test case growth due to Bellman's methodology of recording every distance within the graph.



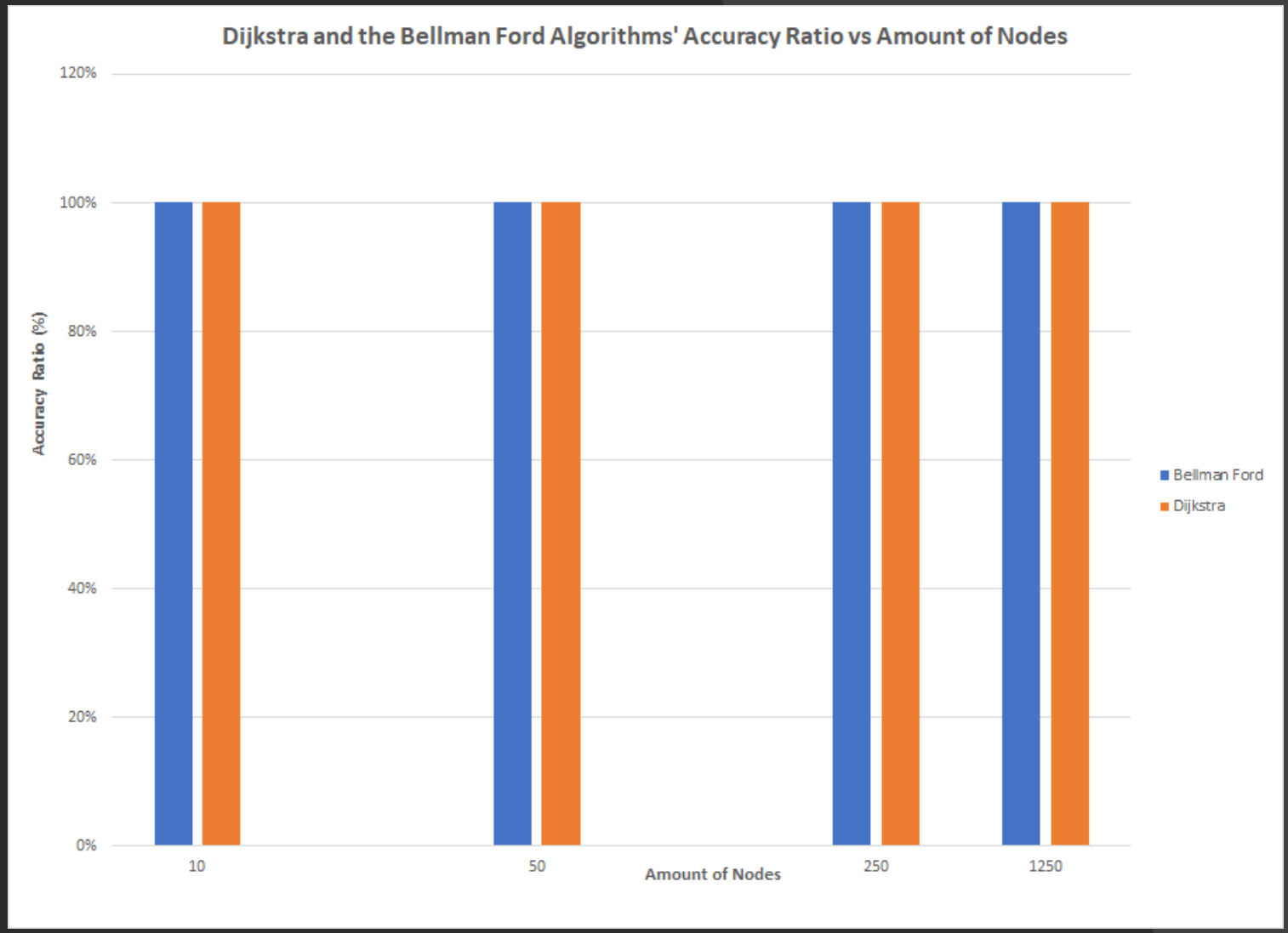
Results Analysis: Comparison of Algorithms

- Comparison of Data Points:
  - **Dijkstra-** Linear Growth
  - **Bellman Ford-** Exponential Growth
  - Both algorithms have shown to have increasing computational times as test cases grew, which was expected, however the Bellman Ford algorithm's time was magnitudes larger than Dijkstra's algorithm even before test case 4.



Accuracy Metric

- 100% accuracy across the board, but why?
  - Only dealing with one factor, edge weights.
  - Test Cases might be too small/easy
- Can this data still be used for interpretation?
- Yes! This accuracy metric stills shows that both algorithms are finding correct SPP Solutions every time, which confirms our computational time can be used for valid comparison.
- With each of the accuracy values being 100%, it just means that comparison between each algorithm's accuracy is **redundant**.



Conclusions

- Based upon the data collected we can come to conclude:
  - That in a scalable positive edge weight environment Dijkstra's algorithm can compute solutions at a faster speed then the Bellman Ford algorithm due to its greedy nature.
  - In graphs smaller than or equal to ~1250 nodes that only compute integer based positive edge weights, accuracy is negligible to measure or in some cases redundant due to the simplicity of the factors involved
  - In the end, I believe the computational speed difference was due to a base line different approach in methodlogy between Dijkstra's algorithm and the Bellman Ford algorithm. Since Bellman Ford records all the graph's interconnected distances and Dijkstra's algorithm only records the distances of those it sees as most optimal.

- What future experiments should be conducted based upon this data?
  - Test both algorithms in larger environments, graphs larger then 1250 nodes.
  - Include more than 1 factor within the graphs for edge weight consideration, such as data traffic between nodes.
  - Compare the computational speed of the Bellman Ford algorithm within a negatively weighted environment to the its performance within a positively weighted environment.

Code DEMO and Walkthrough

References

I. 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

II. 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

III. 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

IV. 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**

V. 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

VI. 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

VII. 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

VIII. 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