Comparison of Shortest Path Algorithms

Aleksja Braka - 150160916
Mehmet Yakuphan Bilgiç - 150170703
Sabrina Cara - 150160914

Abstract

In this project, three shortest path algorithms, that are Dijkstra's algorithm, Bellman-Ford algorithm and Johnson's algorithm, are compared in terms of their speed, memory usage and speed performance on different programming languages. Three different sized datasets are used for the experiments. All the experiments are done using macOS and Intel Core i5 CPU running on 2,3 GHz. Finally, results are presented using graphs and comments are made on their efficiency.

Table of Contents

I. Introduction

This final project report is being presented in response to the request of BLG374E instructor. In this project we aim to test 3 different shortest path algorithms using various datasets and different programming languages which will set up different graphs, so as to find which one is most time and spatially efficient in finding the shortest path from a source to a destination. The algorithms will be implemented mainly in C++ programming language as well as Java and Python and run on Intel Core i5 2,3 GHz processor. Shortest path algorithm is the idea behind "six degrees of separation", which states that all the people in the world are connected to one another by at most six social connections [1]. Moreover, it is the root method of revolutionary applications such as Google Maps and popular games like FIFA. These and many more are what make this report worthy of your time and consideration. The forthcoming sections of this report will include scientific and literature background information about the algorithms, detailed explanation of the experiments, methods used to run the experiments, results obtained from running the experiments, a thorough analysis on the results and data obtained and the causes that might have affected these results followed by a final project evaluation as well as concluding statements regarding the conducted experiments.

II. Background

Nowadays, with the rapid spread of technological devices, the problem of network connection among them has become more complex. Thus, the telecommunication field is always in need of more efficient algorithms, which meet the requirements of companies projects' purpose. Furthermore, another field that would benefit from any improvement on the shortest path algorithm is web mapping. Recent development of technology has led people to use internet services like online shopping and navigation on a higher scale. As a result, shipping companies are looking for advancements in their routing method to lower their operation cost. This being said, the shortest path algorithm is constantly providing solutions for any of the emerging problems related to these fields. Hence, any increase on the efficiency of these processes would affect companies and customers equally. The reasons mentioned above have and are currently driving many scientists and people in computer engineering field to conduct studies and write research papers, as it continues to be a very interesting topic in graph theory. One of the research papers published in IEEE 2012 Beijing Conference was written by Lu and Dong, named "Research of shortest path algorithm based on the data structure". In it they discuss about the importance of Dijkstra's shortest path algorithm, and what improvements can be made in order to increase the algorithms spatial and operational efficiency. [2]

Although at the beginning of this course we were considering to make the project on comparison of tree searches, after researching and understanding the crucial importance of these algorithms in real life applications, we decided to follow the same motivation and make a project on comparison of 3 algorithms: Dijkstra's, Bellman-Ford and Johnson's, in terms of speed vs. dataset size, speed vs. programming language and memory storage vs. dataset size. Below you will be able to see a short introduction and tutorial regarding the theoretical concepts of each algorithm in order to ease your comprehension of the upcoming content.

A. Dijkstra's Algorithm

Dijkstra's algorithm is one of the most used shortest path algorithms in computer science. It finds the shortest path from a given node in a graph, called the source node to any other, positive destination

node. The idea behind this algorithm is relatively straightforward. At start, all nodes of the graph belong to the set on uninspected nodes[3, Fig. 1.1].

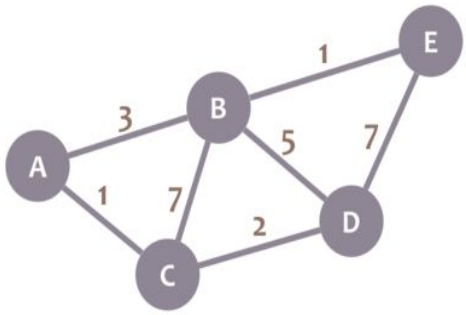


Figure 1.1

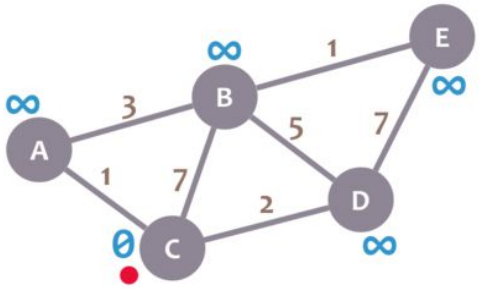


Figure 1.2

After a source node has been chosen, it is then proceeded to measure all distances from that source node to all other adjacent nodes[3, Fig. 1.2, 1.3, 1.4].

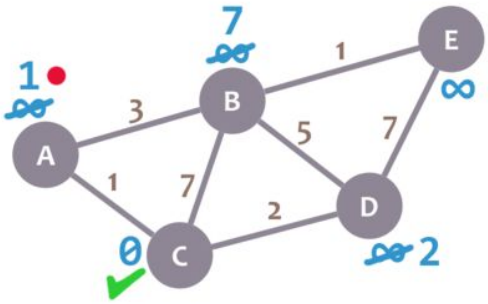


Figure 1.3

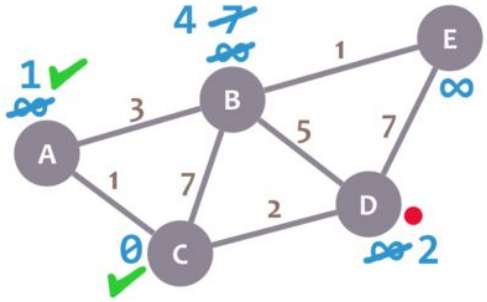


Figure 1.4

When the smallest distance has been evaluated and saved and we have passed to the next, closest node, the previous node is removed from the set of uninspected nodes and the current node becomes the source node [3, Fig. 1.5, 1.6].

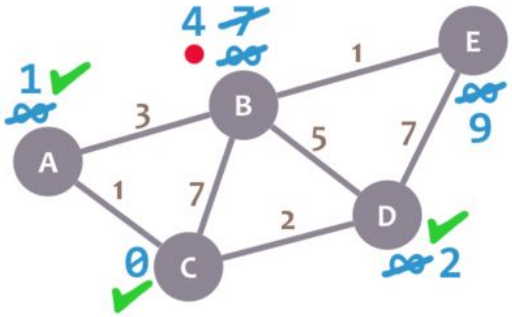


Figure 1.5

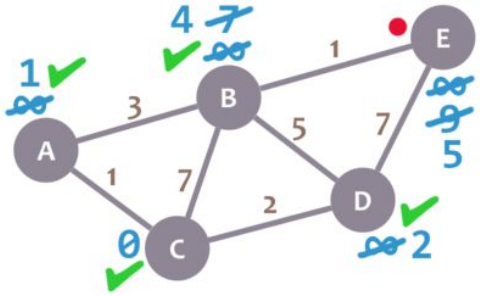


Figure 1.6

This algorithm continues to run until the destination node no longer belongs to the set of uninspected nodes or until all nodes in the graph have been removed from that set[3, Fig. 1.7].After such a condition is achieved, all the saved distances are added and it becomes possible to find the shortest path from source to destination, as well as the total distance. [4]

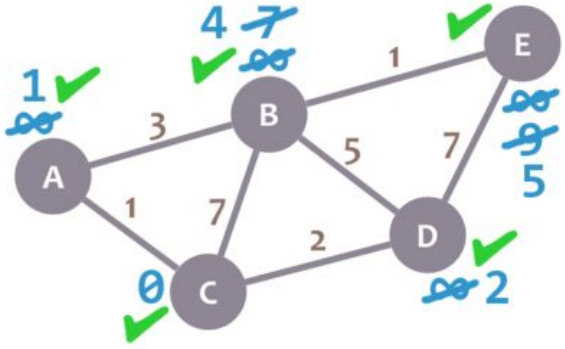


Figure 1.7

With the improvement of Dijkstra's algorithm itself throughout the years, the time complexity and performance have been improved as well. Time complexity of this algorithm depends on the implementation along with the density of the graph it is tested on. It has been stated that for a binary heap implementation, this algorithm has a O(m log n) time complexity, m being the number of nodes and n being the number of edges or paths. On the other hand, an improvement in the implementation, such as Fibonacci heap implementation, has been proved to enhance the time complexity up until O(m+n log n). Similarly, for graphs with high density, binary heap implementation shows to give a O(n2 log n) time, scoring lower when compared to Fibonacci heap implementation , O(n2 +n log n). [5]

```
1:  function Dijkstra(Graph, source):
2:      for each vertex v in Graph:                 // Initialization
3:          dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
4:          previous[v] := undefined                // Previous node in optimal path from source
5:      dist[source] := 0                           // Distance from source to source
6:      Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
7:      while Q is not empty:                       // main loop
8:          u := node in Q with smallest dist[ ]
9:          remove u from Q
10:         for each neighbor v of u:               // where v has not yet been removed from Q.
11:             alt := dist[u] + dist_between(u, v)
12:             if alt < dist[v]                    // Relax (u,v)
13:                 dist[v] := alt
14:                 previous[v] := u
15:     return previous[ ]
```

Figure 1.8 Pseudocode for Dijkstra's algorithm

B. Bellman-Ford Algorithm

The Bellman – Ford algorithm, suggested by Richard Bellman and Lester Ford, Jr. in 1958 and 1956, is one of shortest path algorithms we are examining in this project which implements relaxation approach in to find single source shortest paths on directed graphs.[4] In opposition to Dijkstra's algorithm, Bellman – Ford algorithm does produce accurate outcome when the graph has negative edges. Nonetheless, when a graph which contains negative weighted cycles as input, algorithm cannot produce a correct result. The Bellman – Ford algorithm detects shortest path in the following manner:

1. First of all it initializes distances source to all vertices as infinite and vertices to source 0.
2. After this step algorithm relaxes all edges repeatedly in order to find minimum distance to the next vertex.
3. Lastly, algorithm checks if there is a negative weight cycle, in such case shortest path cannot be found in the graph. [5]

The Bellman – Ford algorithm, in its worst case, runs with the time complexity $O(|V|.|E|)$. Here, V denotes number of vertices and E stands for number of edges of the graph.

Here are the graphical representations of the steps I mentioned above.

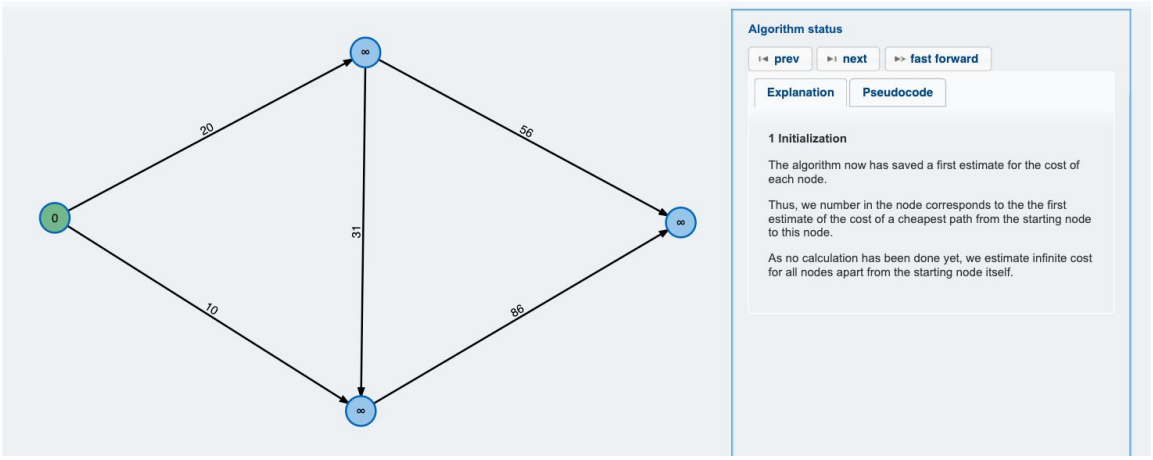


Figure 1.9 (Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].)

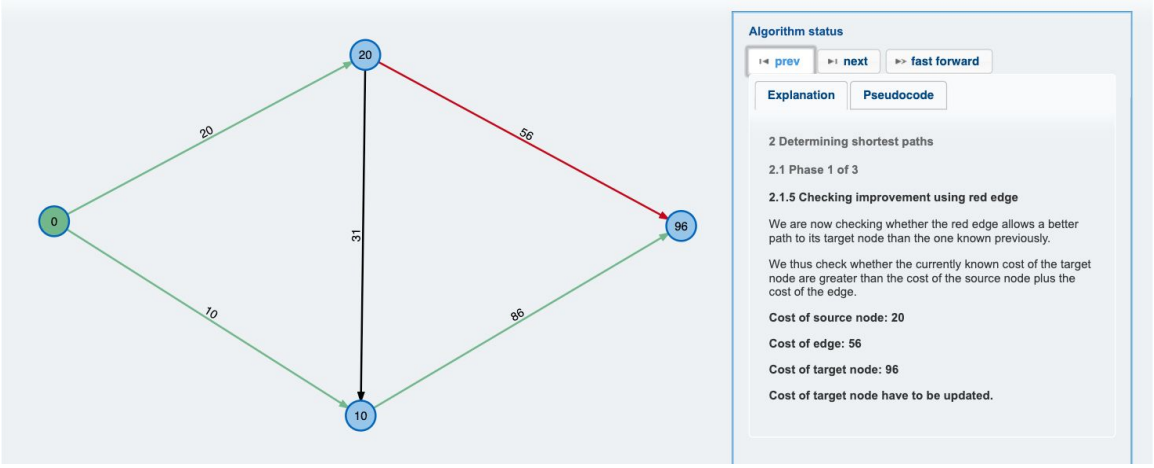


Figure 1.10 (Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].)

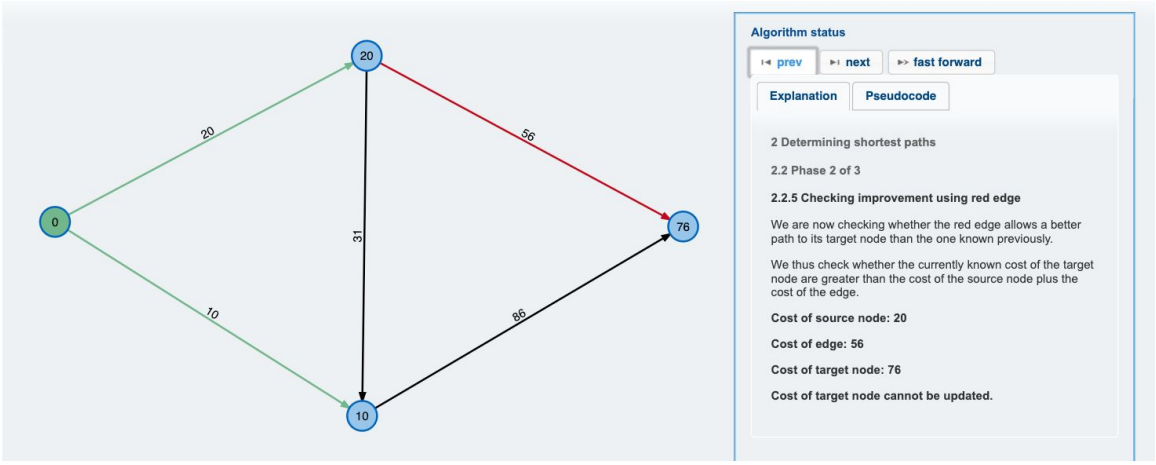


Figure 1.11 (Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].)

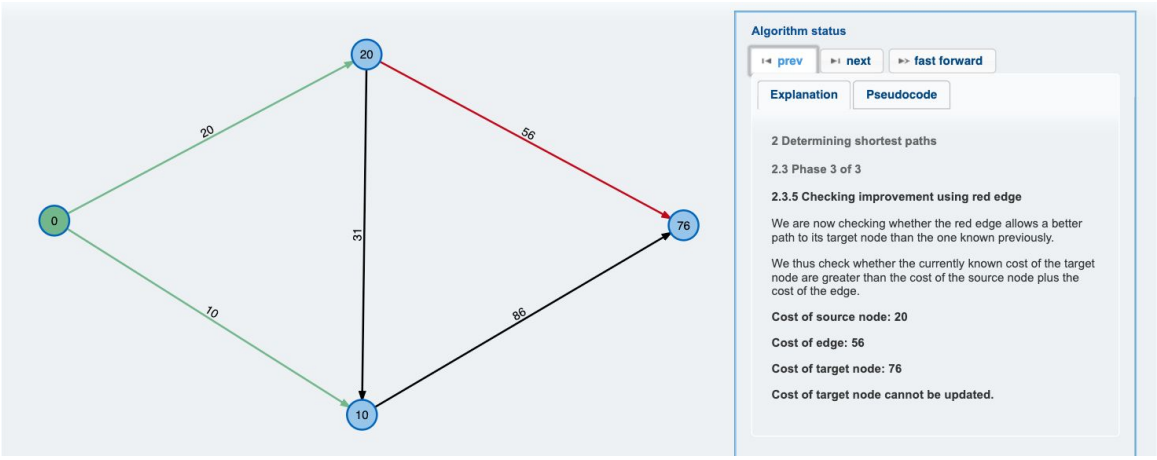


Figure 1.12 (Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].)

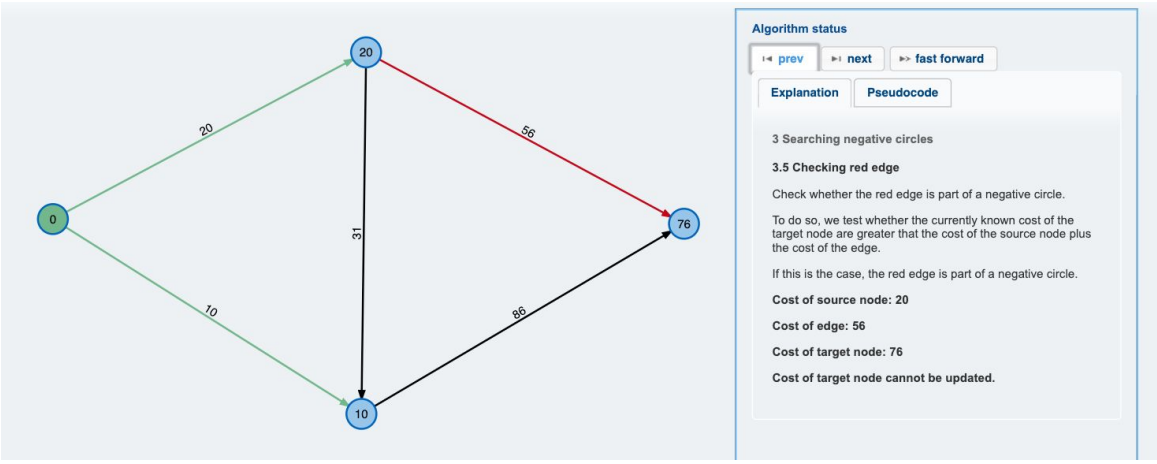


Figure 1.13 (Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].)

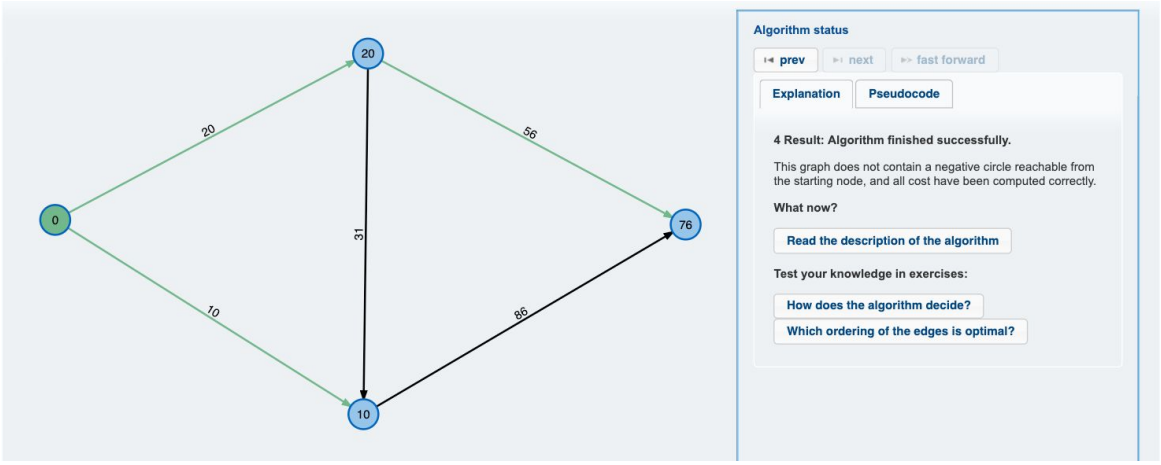


Figure 1.14 (Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].)

C. Johnson's Algorithm

Johnson's algorithm is a short path algorithm that can be run for directed, sparse, or edge weighted graphs. The algorithm works with negative edge situations, but terminates with no output in case of negative weighted cycles. In brief, it is a merge of 2 other algorithms: Bellman Ford and Dijkstra. The first is used to change the existing graph. Then, the second runs in the new resulted graph. As for time complexity, it is a sum of:
Bellman Ford time complexity: $O(|V||E|)$, Dijkstra time complexity: $O(|V|log|V|)$.
So, Johnson's time complexity: $O(|V|^2log|V| + |V||E|)$.
The name of the algorithm is taken by its publisher (1977), Donald B. Johnson [6].
Steps for Johnson's Algorithms execution, as in [7]:

1. Create a node with weight zero called *z*. Connect every given node to the node *z*.
2. Find shortest path weight from *z* to all other points of the graph, one by one. If there is a negative cycle, stop the algorithm.
3. Weight the edges of the given graph again, using the values found in Step 2.
4. Remove the *z* point and use Dijkstra's algorithm to calculate shortest path in the reweighted graph.

As mentioned in [8], idea of Johnson's algorithms was suggested in 1954 and is considered the first official heuristic for the PFSP (Permutation FlowShop Problem). Specifically, it solves the case of 2 machines, or of *m* machines, clustered into 2 "imaginary" machines; the execution complexity of this problem is $O(|n|log|n|)$. In addition, Johnson's algorithm is the root logic under many algorithms of various authors, such as Dudek and Teuton (1964).

For better understanding, a visualization of the algorithm is shown in Fig 1.15. Johnson's algorithm is a conjunction of Dijkstra and Bellman Ford. In such a case, detailed steps are shown in the previous experiments' graphs, too.

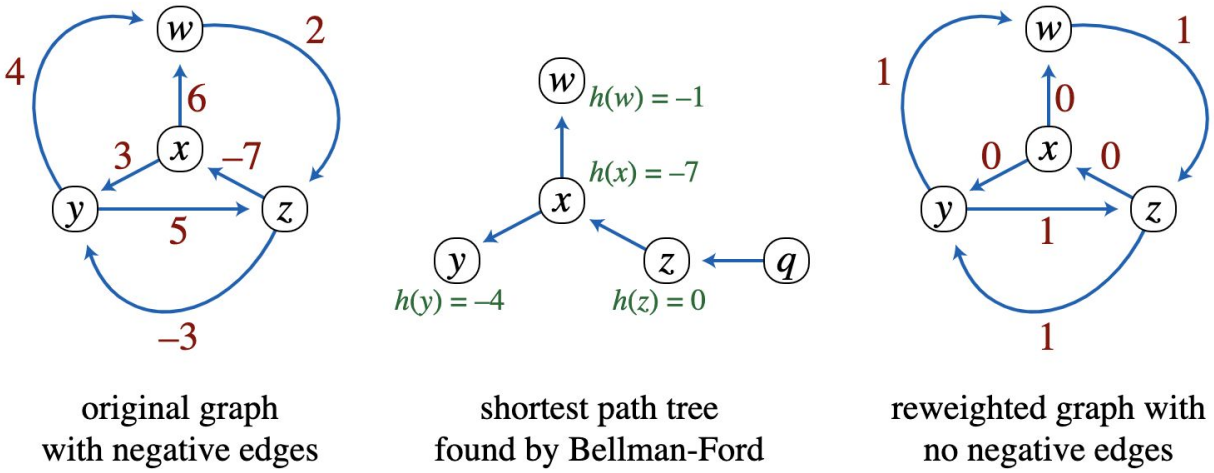


Fig. 1.15 Eppstein, D. (2008). Johnson's algorithm for transforming a shortest path problem in a graph with negative edge weights into an equivalent problem with non-negative weights. [image] Available at: https://upload.wikimedia.org/wikipedia/commons/4/4f/Johnson%27s_algorithm.svg [Accessed 10 May 2019].

III. Methods

In order to conduct each experiment we had to first find or code the algorithms in C++, Java and Python. After doing this, we made a random graph generator program which would later prove to be useful in helping us randomly create graphs from size to large and automatically write them in text files. First two numbers of text files would be respectively number of nodes and edges followed by three columns. First column would contain source node, second column destination node and third column would contain edge or distance between them. Small sized graphs were considered the ones with 3- 40 nodes. Medium sized graphs were generated with 300-500 nodes and large sized graphs were generated with 1000-1500 nodes. After having completed the above tasks, we started the experiments handling.

Results of Small sized files

Dijkstra	Bellman	Johnson	Distance
0.046195	0.04096	0.056259	2
0.04507	0.038447	0.045533	5
0.048837	0.039206	0.061562	11
0.050716	0.039083	0.03719	4
0.04754	0.033938	0.118981	5
0.047096	0.039193	0.074795	18
0.048883	0.039392	0.075016	2
0.060865	0.472431	0.12655	12
0.116137	0.085897	0.153209	13
0.050319	0.04825	0.180204	10
0.054424	0.046557	0.42103	6
0.066642	0.057042	0.535879	9
0.108832	0.054672	0.576762	39
0.065497	0.047106	0.503613	10
0.062125	0.047536	0.487689	16
0.073487	0.063313	0.595599	6
0.094993	0.078972	0.596527	9
0.081811	0.064263	0.596705	39
0.099293	0.079799	0.609179	10
0.070994	0.065677	0.594801	16

Figure 2.1

Results of Large sized files

Dijkstra	Bellman	Johnson	Distance
0.606494	6.40516	435.076	1735
0.672892	6.30438	437.595	1773
0.681953	7.75917	508.362	2101
0.775919	9.58471	616.92	1989
0.748611	11.7858	704.092	1793
0.986435	16.1807	809.734	1971
0.969375	16.2586	956.367	2379
0.903366	17.154	1054.31	65
0.990403	21.054	1189.35	3085
1.106	22.1691	1330.69	1623
0.840481	6.75662	421.46	1885
0.705334	7.20341	485.965	711
0.697871	8.33591	565.033	813
0.760424	11.1091	661.059	2631
1.54828	13.0386	787.472	5728
0.876604	14.6753	909.84	273
0.941267	16.2934	996.095	1125
0.965511	18.3406	1135.28	2605
1.001	20.5735	1237.98	2660
2.15839	24.1693	1375.21	3239

Figure 2.2

Results of Medium sized files

Dijkstra	Bellman	Johnson	Distance
0.458205	1.52554	62.6827	329
0.241788	0.861041	62.9258	589
0.564381	2.13102	67.7662	200
0.335864	1.00459	61.8923	463
0.334343	0.960921	80.0768	1138
0.255369	0.945047	56.8053	275
0.278358	1.0155	62.3073	1116
0.276159	0.966309	64.0035	496
0.383343	0.852939	63.0343	119
0.327711	1.5315	59.2026	1216
0.276024	1.07934	76.4646	441
0.533693	2.24281	79.0939	423
0.261911	0.742265	45.901	212
0.357007	1.38743	79.724	3169
0.29436	1.14904	79.385	93
0.358842	1.43028	82.9969	573
0.369036	1.47319	85.1996	417
0.323575	1.51148	87.8883	901
0.259352	0.682965	42.1754	225
0.260154	0.656942	40.8512	532

Figure 2.3

A.Experiment 1

Experiment 1 was conducted by student Sabrina Cara. It consists in using different sized datasets to test the speed of the three algorithms mentioned above. Datasets were numerous but divided in three main groups: small sized, medium sized and large sized datasets. After achieving the C++ codes of all algorithms, I ran them separately in each of these datasets by entering the name of a graph file as an argument while running the code. This way I could receive the time each of the algorithms took to run. Afterwards, I divided the results in three tables according to the size of the dataset and used them to plot the graphs in Google Sheets online. Result tables can be seen in Figure 2.1, 2.2, 2.3. I ran all the algorithms in the same computer, same programming language and CLion for C++ in order to avoid differences that might arise by using different processors or operating systems.

B. Experiment 2

Experiment 2 was conducted by Mehmet Yakuphan Bilgiç. In this experiment, I aimed to compare the speed of the algorithms on three different programming languages, using fixed size data set. I implemented the algorithms on Java and Python, I used the same code as in Experiment 1 for C++. I made the file reading part same in all 3 different codes in order to use same file format as inputs. I designed the programs in a way that they would write the results to an output file. Also, I wrote a C++ program which creates random graphs in the format I needed for my input files. I used PyCharm Professional for Python3, IntelliJ Idea Community for Java and CLion for C++. After running the codes and getting the result files, I used these values to plot my graphs.

C. Experiment 3

Experiment 3 was computed by Aleksja Braka. The aim of this experiment is to test the mentioned short path algorithms in terms of memory usage (dependent variable). In order to have an independent, clear output and to focus fully on one specification, same device and same programming, same application and language were used again. Algorithms were coded in C++, one independent variable. Dataset, second independent variable in the experiment, was various and in different sizes. Consequently, the comparison was done more accurately, separately for small, medium, large files. I ran the algorithm 3 times for better accuracy. In the end, results were plotted in a graph using MS Excel. I used a line graph for a more clear view of each line.

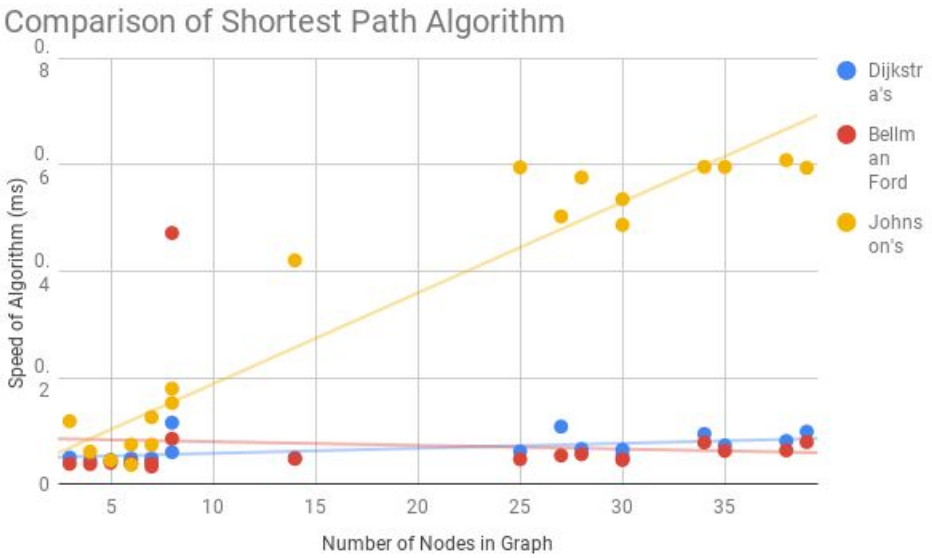
IV.Results

Following the formerly conducted experiments from each team member we obtain some results based on which we plotted our respective graphs and made the appropriate deductions.

1) Experiment 1

As mentioned above, this experiment aimed to compare three different shortest path algorithms in terms of their speed performance. I will be dividing the results into three main categories, based on the size of the graph the algorithms were tested on. They will be better shown in the following graphs as well. These scattered graphs will lead to a better visualization of the relation that exists between the size of the graph, represented by the number of nodes, and the time each algorithm took to run.

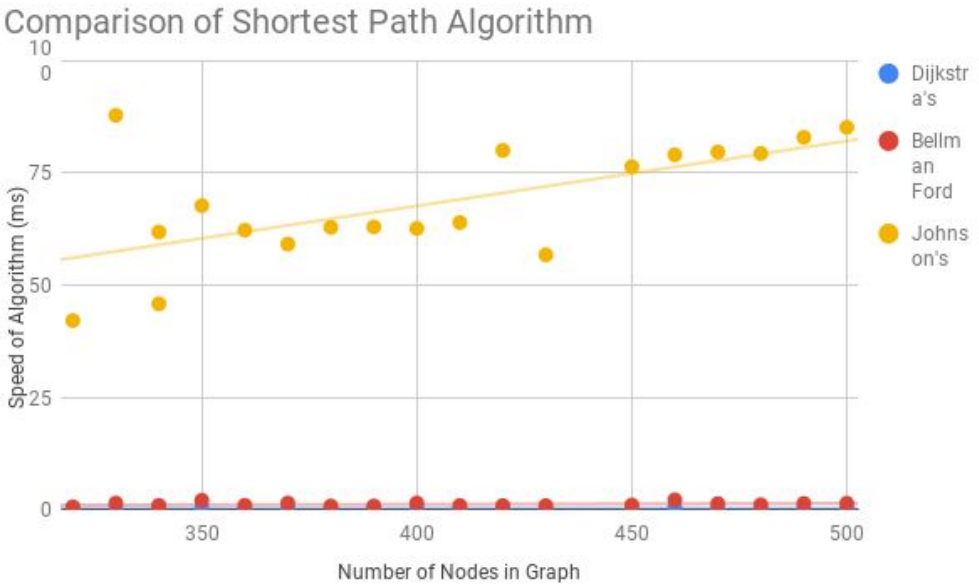
A. In terms of small sized datasets.



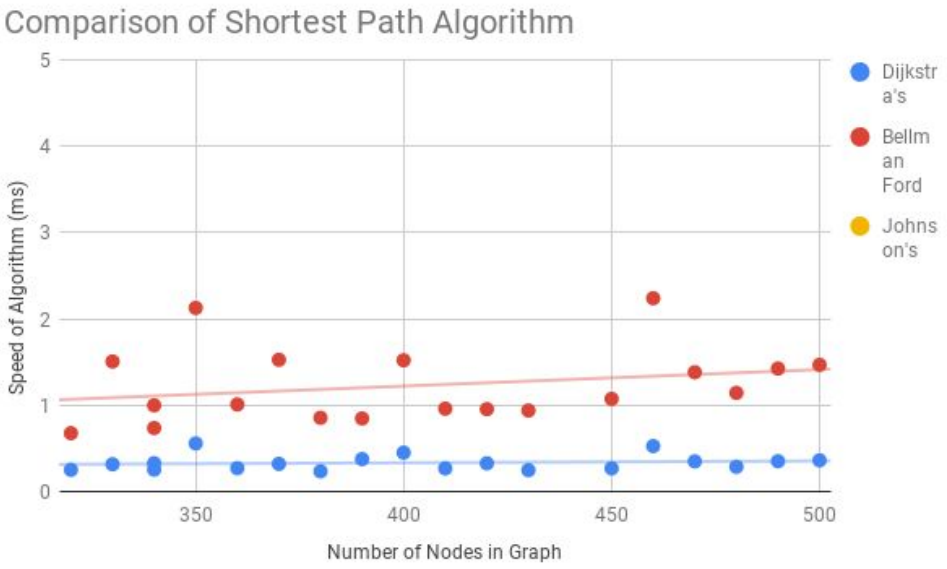
Graph 1.1 Comparison of SH.P.A in small datasets

In small sized datasets Bellman Ford performs faster overall surpassing Dijkstra's speed as the size of the graph increases. Throughout the entire graph(graph 1.1), you will be able to note that Johnson's algorithm took the longest to run.

B. In terms of medium sized datasets.



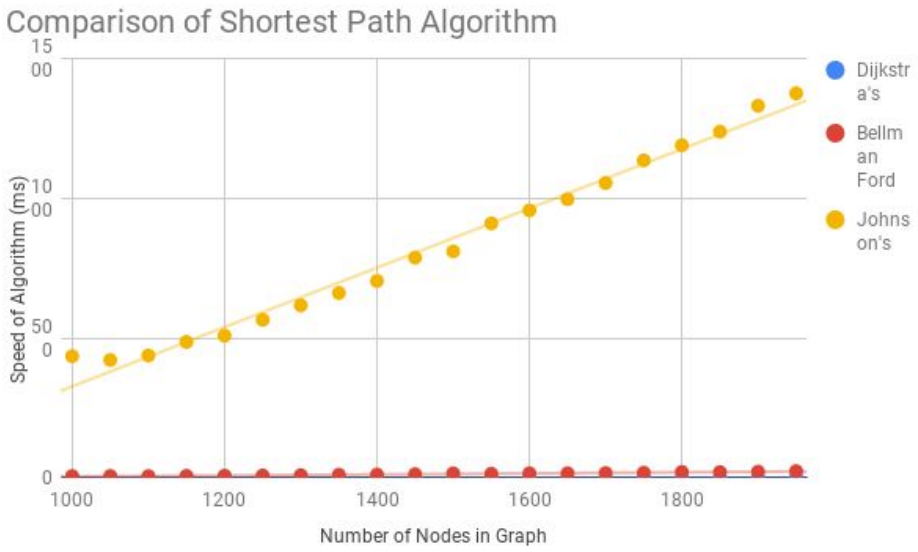
Graph 1.2 Comparison of SH.P.A in medium datasets



Graph 1.3 Comparison of SH.P.A in medium datasets(closer look)

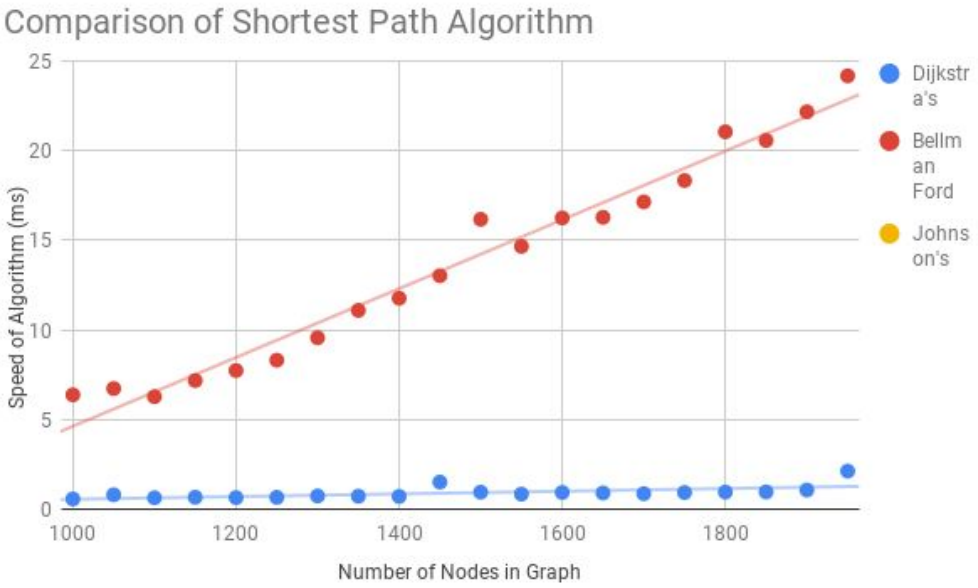
In medium sized datasets, Dijkstra's algorithm is faster with an almost negligible difference to Bellman Ford's. Johnson's algorithm performs again worse in terms of speed(graph 1.2).Graph 1.3 will aid a better visualization of the proximity in performance between Dijkstra's and Bellman Ford's algorithm.

C. In terms of large sized datasets.



Graph 1.4 Comparison of SH.P.A in large datasets

In large sized datasets, again Dijkstra's algorithm is the fastest followed closely by Bellman Ford's algorithm and finalised by Johnson's algorithm, which's speed performance rate remains intact (graph 1.4). Graph 1.5 will aid a better visualization of the proximity in performance between Dijkstra's and Bellman Ford's algorithm.

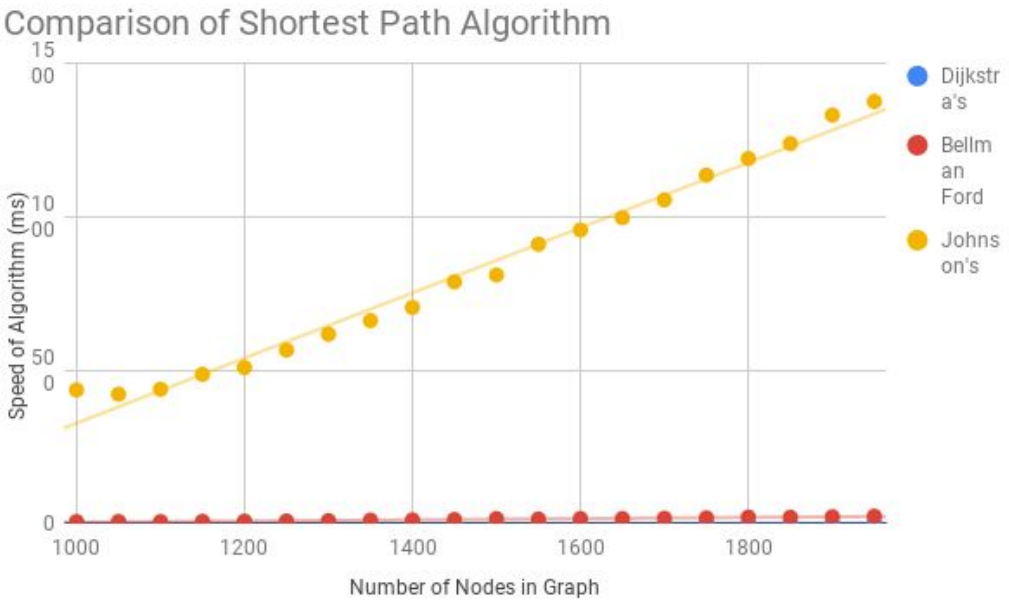


Graph 1.5 Comparison of SH.P.A in large datasets(closer look)

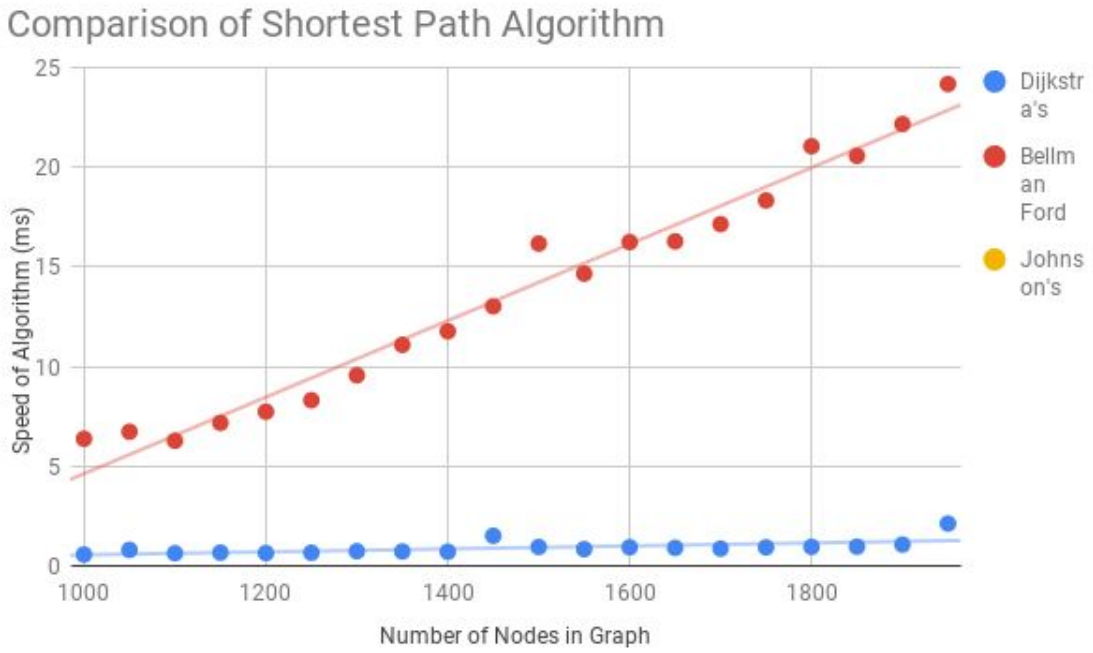
2) Experiment 2

In this experiment, I compared the algorithms on three different languages. Graphs for result of my experiment regarding C++ is same last graph for experiment 1.

A) Results for C++ programming language.

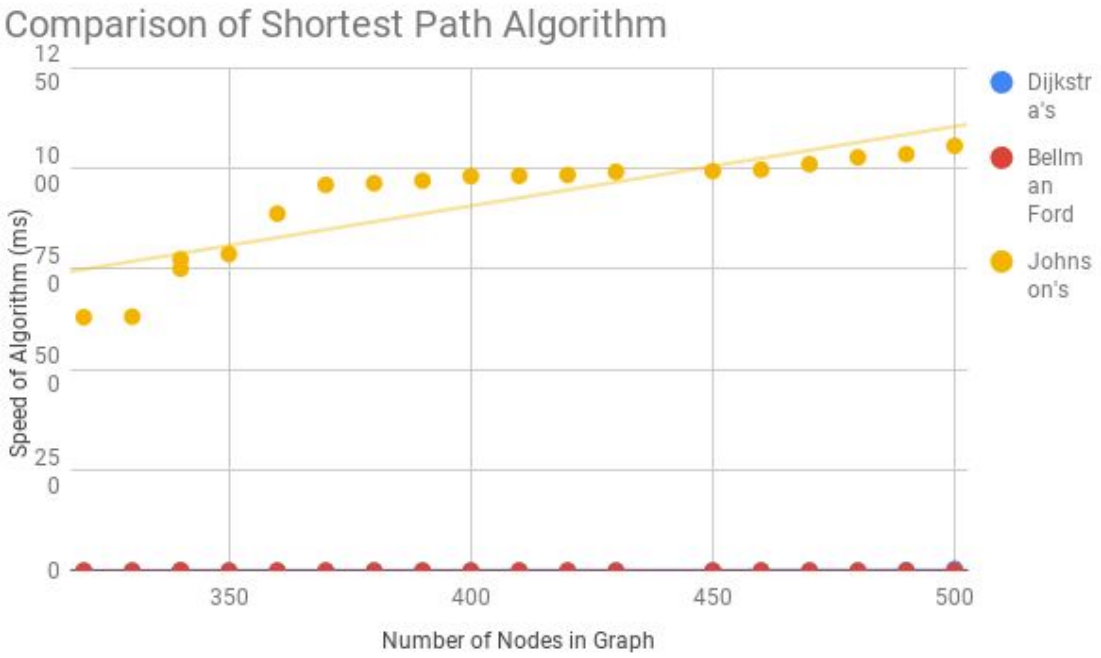


Graph 2.1.1 Performance of Algorithms in C++



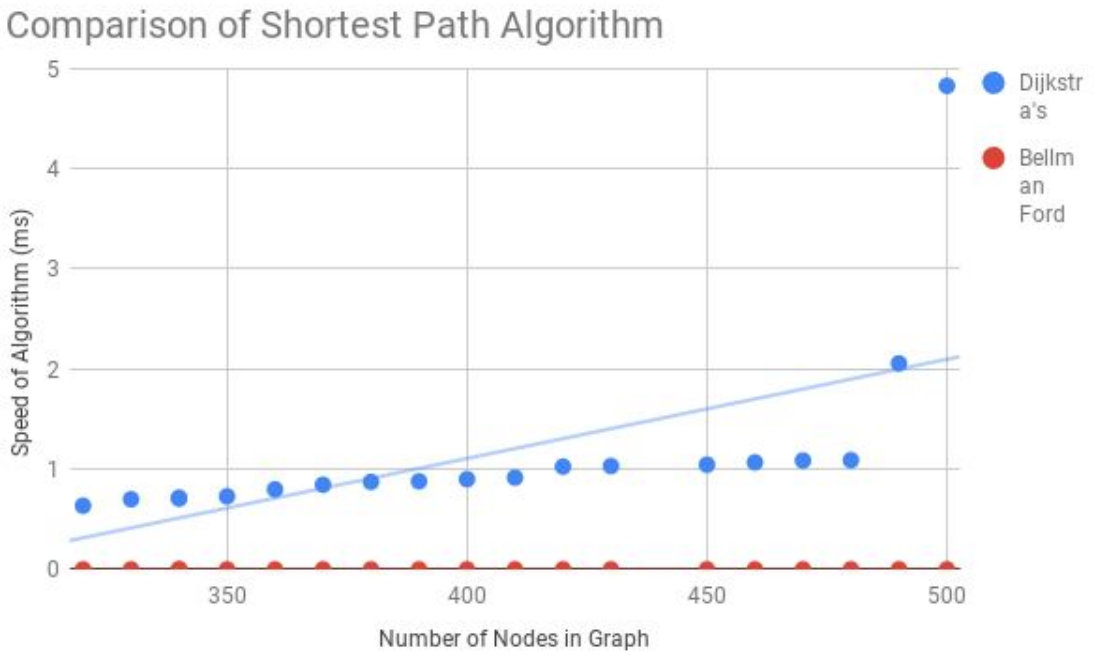
Graph 2.1.2 Performance of Algorithms in C++ (Zoomed in)

B) Results for Java programming language



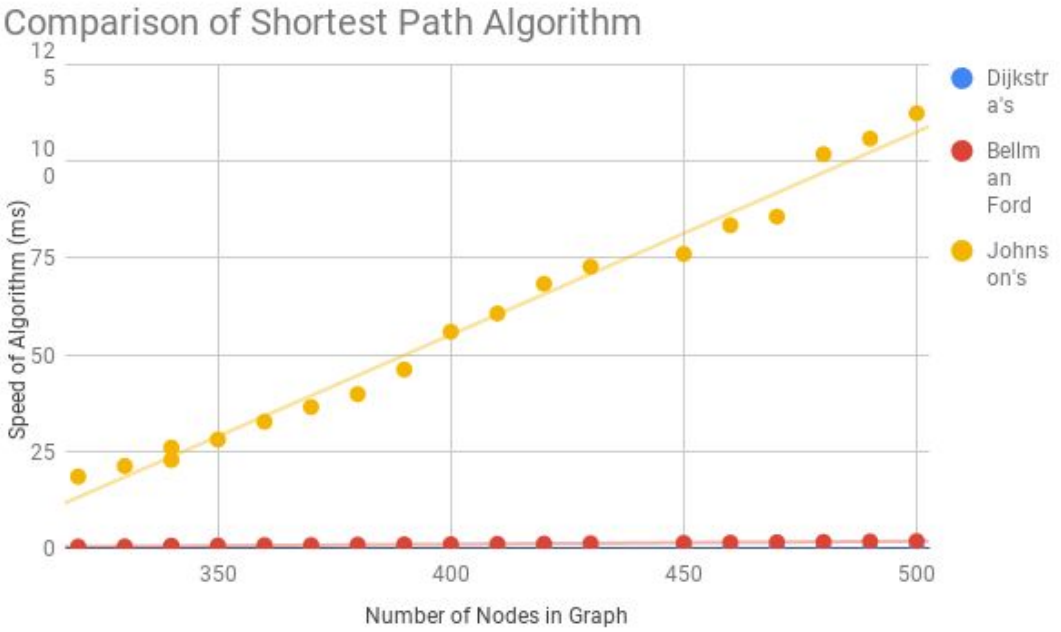
Graph 2.2.1 Performance of Algorithms in Java

Here is the graph zoomed in to Bellman-Ford and Dijkstra in order to see the difference better (Graph 2.2.2).



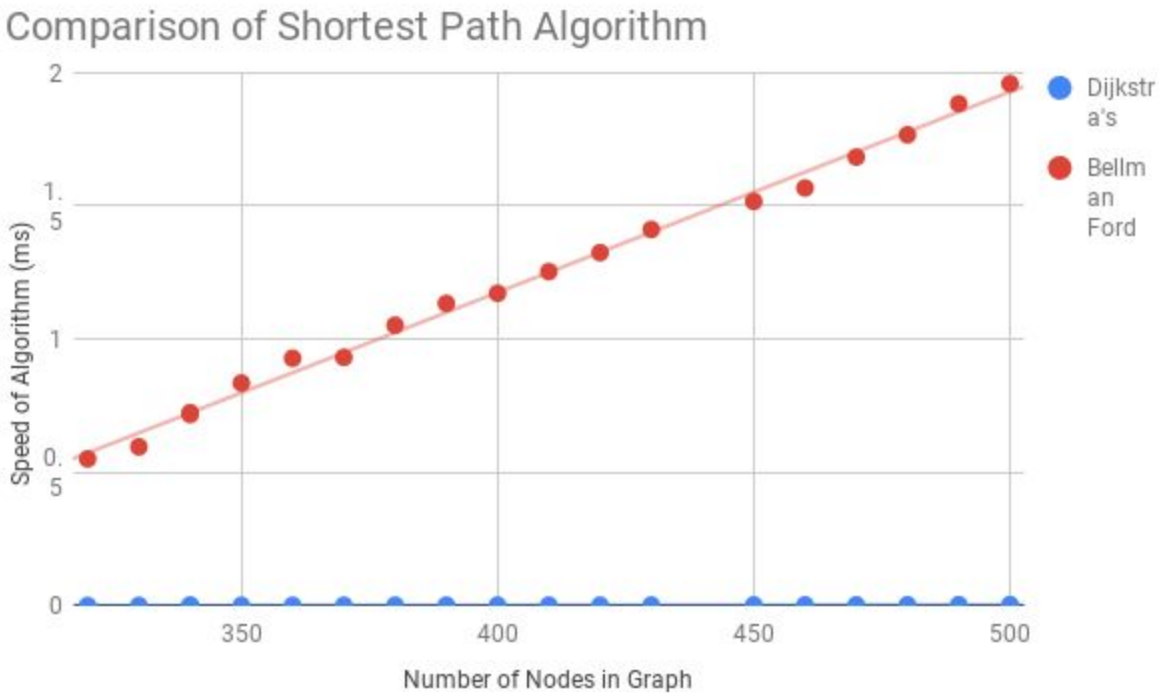
Graph 2.2.2 Performance of Algorithms in Java (Zoomed in)

C) Results for Python programming language



Graph 2.3.1 Performance of Algorithms in Python

Here is the zoomed version again in order to report the difference between lower values on Y-axis (Graph 2.3.2).



Graph 2.3.2 Performance of Algorithms in Python (Zoomed in)

3) Experiment 3

Regarding the comparison of the 3 approaches to finding the shortest path, below is the obtained data while running the algorithms. Whole time is divided into 3 segments where first belongs to Dijkstra's algorithm running time, second to Bellman Ford and third to Johnson.

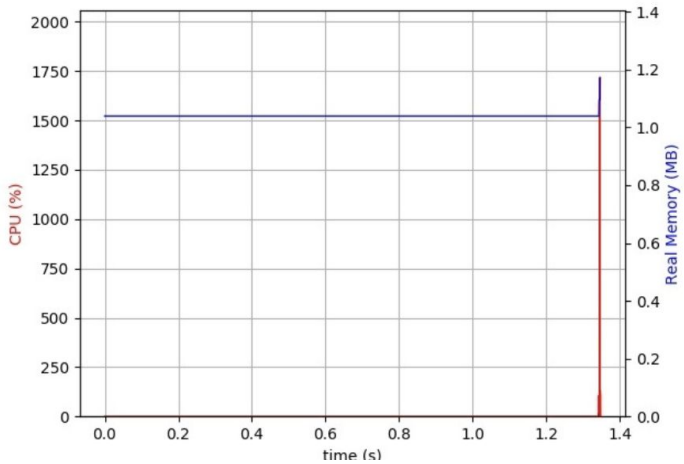


Figure 3.1. Memory usage in small files

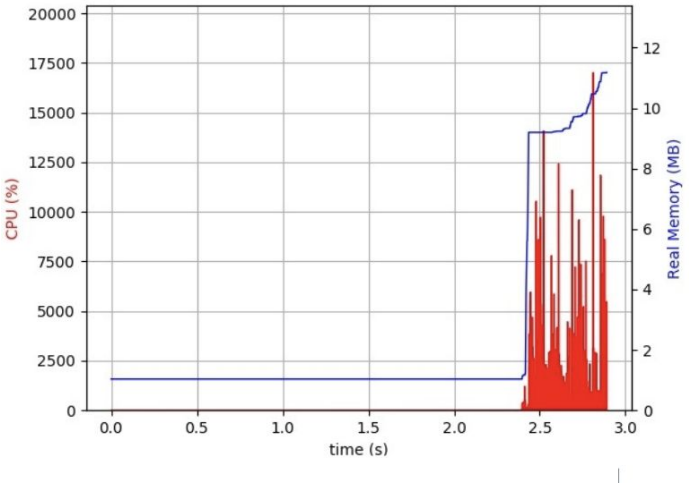


Figure 3.2 Memory usage in medium files

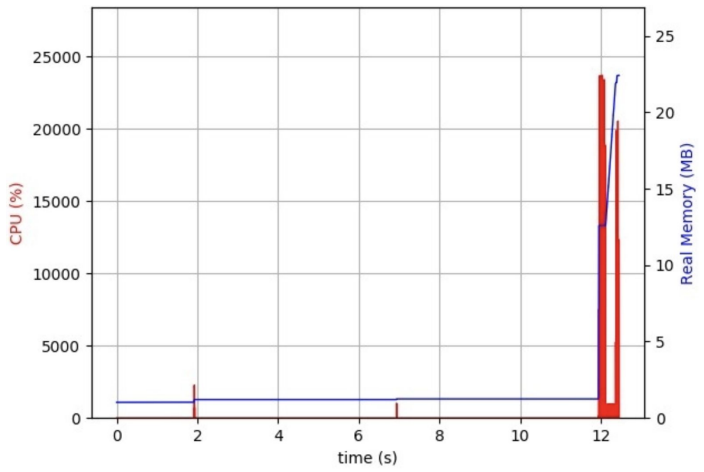
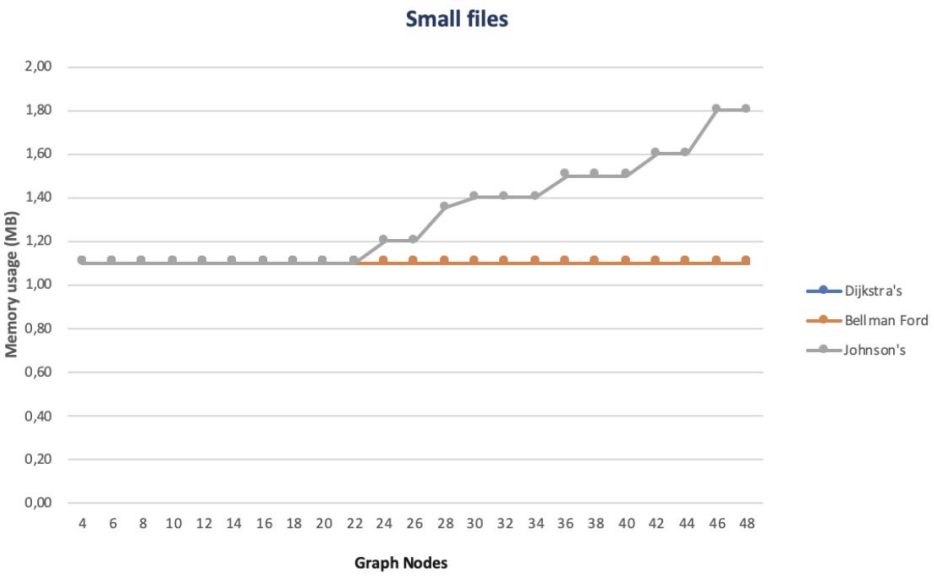


Figure 3.3 Memory usage in large files

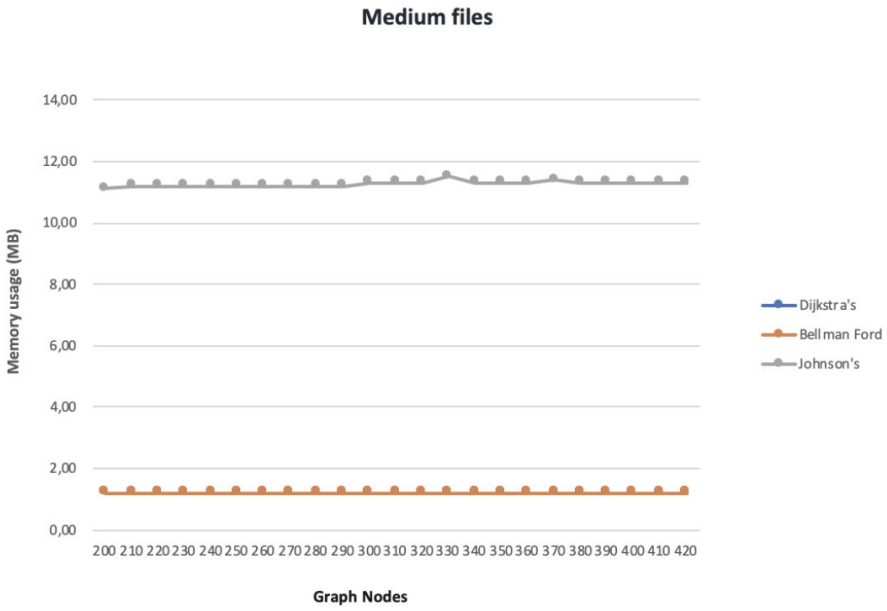
Codes were ran 3 times for reliable results and almost the same graphs were obtained. This data was used and mirrored to line graphs using MS Excel. The results for memory usage for each algorithm in different data sets can be found in Graph 3.1, Graph 3.2, Graph 3.3.

A) Results for small sized data sets



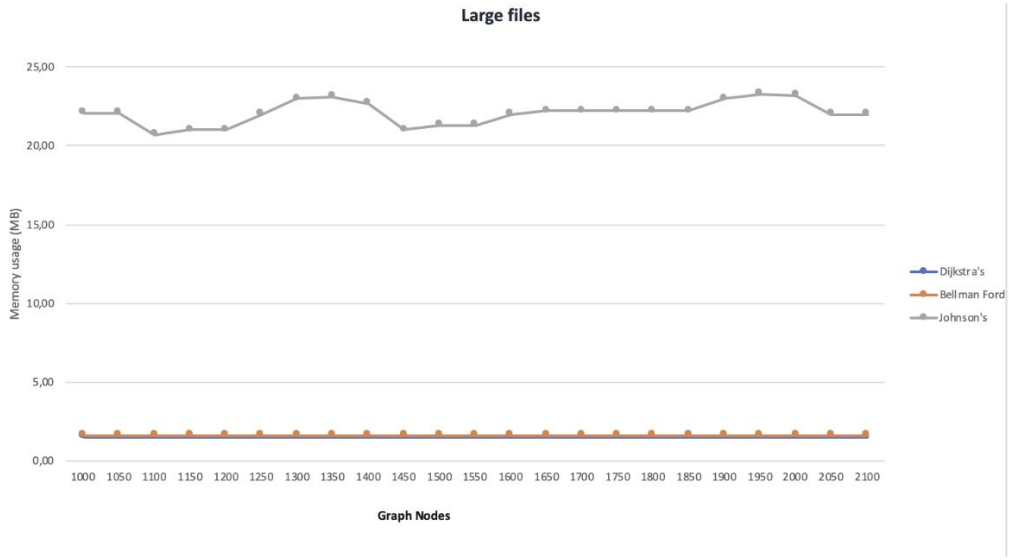
Graph 3.1 Memory usage in small files

B) Results in medium sized data sets



Graph 3.2 Memory usage in medium files

C) Results in large sized data sets



Graph 3.3 Memory usage in large files

V. Analysis

In total, three separate reviews need to be made for each experiment. The experiments set against algorithms in different ways so the main focus of the analysis will be their respective testing aspect. The testing environment is the same for each experiment. Device used is Macbook Pro 2017, 2,3 GHz Intel Core i8, 8GB 2133 MHz DDR3, application used for files coded in C++ is CLion 2019.

1.2., for files coded in Python is PyCharm 2018.3.1, for files coded in JAVA is intelliJ IDEA Community 2019.1.1.

1. Speed Comparison in different Data Sets

We used random data sets, generated by a program we coded ourselves. Data sets are divided into 3 categories: small (0 - 40 nodes), medium (300 – 500 nodes), large (1000 – 2000 nodes).

- In small files, we notice a good performance of Dijkstra's algorithm for a very small number of nodes. However, following the increase in the number of nodes,we can notice a fall on Dijkstra's performance and an increase in Bellman Ford's algorithm performance. Meanwhile, Johnson's algorithm shows to be the slowest, reaching a difference of 6 milliseconds more in the end, this due to the fact that Johnson's implementation contains both Dijkstra's and Bellman Ford's algorithm.
- In medium sized datasets, while running the algorithms we were able to understand from the results retrieved that Dijkstra's algorithm outdid the others in terms of speed. Throughout the overall number of medium graphs tested, it took the lowest to run, followed by a small difference of 1 millisecond by Bellman Ford's algorithm and ended by Johnson's algorithm which, as in small sized graphs, still performed under average.

- In large sized datasets, the difference between Dijkstra's algorithm and Bellman Ford's algorithm remains small although we notice a slightly bigger gap among the two in terms of speed performance. Similarly to other graphs tested previously, Dijkstra's algorithm still performs better showing an almost constant speed rate whilst the number of nodes grows, followed by Bellman Ford's algorithm which's speed reaches up to 24 milliseconds and Johnson's algorithm scoring a much higher speed rate of up to 1400 milliseconds.

In conclusion, Dijkstra's algorithm shows an overall better speed performance, followed by Bellman Ford's algorithm as well as Johnson's which remains slowest whilst taking into consideration different dataset sizes.

2. Speed Comparison in 3 different programming languages

Large data files (1000-2000 nodes) were used to test the algorithms in different languages.

- As far as C++ is concerned, Dijkstra's algorithm is faster. Johnson's algorithm performs worse and with big time difference compared to the others. It can be noticed how Dijkstra's and Bellman Ford's lines are almost horizontal, but Johnson's line has a higher slope.
- Implemented in JAVA, Bellman Ford is the fastest and Johnson again has a worse time performance, even though this time better than in C++.
- Talking about Python, best time performance is achieved by Dijkstra's, following with Bellman Ford's and then Johnson's.

To sum up, Dijkstra's is in general faster regardless of the language and Johnson is the slowest. In addition, Dijkstra's itself has a better performance in Python, Bellman Ford's in JAVA and Johnson's in Python.

3. Memory Usage Comparison in different Data Sets

Random data sets of different sizes, as in Experiment 1, are used. The focus of the comparison in this experiment is memory usage of each algorithm. Experiment was repeated 3 times for reliable results.

- In small sized files, algorithms use same memory until number of nodes reaches 22. Then lines of Dijkstra's and Bellman Ford's continue overlapped (around 0,7 Mb), but Johnson's line rises up with the rise in number of nodes until it reaches 1,8 Mb of memory.
- In medium sized files, Dijkstra's and Bellman Ford's lines again overlap and the memory usage does not differ from the one in small data sets. However, Johnson's algorithm has reached a state of memory usage in range of 11-12 Mb.
- In large sized files, Dijkstra's and Bellman Ford's lines almost overlap, with a slightly better performance of the first. Johnson's algorithm memory usage range varies from 20-25 Mb.

In conclusion, it is obvious that Dijkstra's and Bellman Ford's are more advantageous when considering memory usage. This result is expected as Johnson's is a conjunction of the other two.

All things considered, in terms of time, flexibility of performance in languages and memory usage, Dijkstra's is a better choice and Johnson's does not meet our requirements. Still Johnson's algorithm is the only among the three that can work on negative weighted edges.

VI. Project Evaluation

The finalization of the project was realised in accordance to the initial schedule. We did not face any difficulty in finding a common time to meet and everything was completed as expected. All members have attended the meetings and shared the progression for the Project in front as well as online using Google Documents. The responsibilities can be found in Table 1 and the Schedule of our work for the Project can be found in Table 2.
All the major topics and tasks of our job are mentioned below. Furthermore, we have done our literature review in the background paragraphs.

Tasks	Yakuphan	Sabrina	Aleksja
Coding all the algorithms	R, A	R, A	R, A
Creating datasets	R, A	R, A	R, A
Experiment 1	I, C	R, A	I, C
Experiment 2	R, A	I, C	I, C
Experiment 3	I, C	I, C	R, A
Introduction and Background Introduction	I, C	R, A	I, C
Abstract, Conclusion, Methods Introduction, Results Introduction	R, A	I, C	I, C
Analysis and Project Evaluation	I, C	I, C	R,A

Preparing the presentation	R, A	R, A	R, A
Proofreading	R, A	R, A	R, A
Preparing for presentation	R, A	R, A	R, A

Table 1 – RACI Model Chart

Coding all the algorithms	7 days	01.04.2019 – 07.04.2019
Creating datasets	3 days	10.04.2019 – 12.04.2019
Experiment 1	1 day	20.04.2019 – 20.04.2019
Experiment 2	1 day	24.04.2019 – 24.04.2019
Experiment 3	1 day	28.04.2019 – 28.04.2019
Final Report	7 days	06.05.2019 12.05.2019
Proofreading	2 days	13.05.2019 – 14.05.2019
Writing the presentation	3 days	29.05.2019 – 01.05.2019
Preparing for presentation	1 days	02.05.2019 – 02.05.2019

Table 2 – Project Schedule

VII. Conclusion

In this project, we conducted three experiments which compare performance of 3 shortest path algorithms. Experiments are done successfully and we were able to observe the following outcomes,

- Johnson's algorithm is slowest regardless of dataset and programming language, and uses more memory compared to Bellman-Ford and Dijkstra's algorithm.
- Dijkstra's algorithm is fastest except in Java, where Bellman-Ford takes the lead.
- Bellman-Ford and Dijkstra's algorithm have same memory consumption, with slight differences in large files, where Dijkstra's algorithm uses least.

The project was completed successfully before its due date.

VIII. References

[1] D. Smith, "Proof! Just six degrees of separation between us," *The Guardian,* para. 1, August 3, 2018. [Online]. Available: https://www.theguardian.com/technology/2008/aug/03/internet.email .[Accessed Mar. 3, 2019].

[2] J. Lu, C. Dong, "Research of shortest path algorithm based on the data structure," in IEEE Int. Conf. on Computer Science and Automation Engineering, Beijing, China, 2012, Accessed on: May. 10, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/6269416 (doi:10.1109/ICSESS.2012.6269416)

[3] "Shortest path with Dijkstra's algorithm", 2017, [Online]. Available: https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm [Accessed May. 10, 2019].

[4] K. Gutenschwager, A. Radtke, S. Volker and G. Zeller, "The shortest path: Comparison of different approaches and implementations for the automatic routing of vehicles." in *Winter Simulation Conf.,* 2012, p.3313. Accessed on: Mar. 03, 2019. [Online]. Available at: https://informssim.org/wsc12papers/includes/files/con231.pdf

[5]S. Saunders, "A Comparison of Data Structures for Dijkstra's Single Source Shortest Path Algorithm," University of Canterbury, Nov. 5,1999. Accessed on : Mar. 03, 2019. [Online]. Available: https://www.cosc.canterbury.ac.nz/research/reports/HonsReps/1999/hons_9907

[4] D. Walden, "The BellmanFord Algorithm and "Distributed BellmanFord"" May 2005. Accessed on: Mar. 03, 2019. [Online]. Available: http://www.waldenfamily.com/public/bfhistory.pdf

[5] *Herzog, M. et al (2013). The Bellman-Ford Algorithm Available at: https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford [Accessed 10 May 2019].*

[6] Zhu Wang, Xiao. "The Comparison of Three Algorithms in Shortest Path Issue" *Journal of Physics*, vol. 1087, pp 022011, August 2018 [Online]. Available: https://www.researchgate.net/publication/328036215_The_Comparison_of_Three_Algorithms_in_Shortest_Path_Issue [Accessed on: Mar. 03, 2019].

[7] "Johnson's algorithm," in Wikipedia: the Free Encyclopedia [Online], December 4, 2017. Available: https://en.wikipedia.org/wiki/Johnson%27s_algorithm [March 3, 2019].

[8] Paul E. Black. "Johnson's algorithm." Internet: https://www.nist.gov/dads/HTML/johnsonsAlgorithm.html, Dec. 17, 2004 [Mar. 03, 2019].

[9] R. Ruiz and M.Concepción. "A comprehensive review and evaluation of permutation flowshop heuristics." *European Journal of Operational Research*, vol. 165, pp. 479494, 1 September 2005.