<u>Compare and contrast the efficiency of the Dijkstra shortest path algorithm and the Shortest Path Faster Algorithm (SPFA) on randomly generated bi-directional weighted graphs at various input sizes.</u>

To what extent does the efficiency of Dijkstra's Algorithm and Shortest Path Algorithm differentiate on randomly generated unweighted bi-directional graphs?

A Computer Science Extended Essay

3662 Words

Table of Contents:

1: Introduction

Graph theory is the study of mathematical structures that model pairwise relationships between variables, known as graphs. Each graph is composed of a set of vertices which are connected by another set of edges (Weisstein, 2021). The concept of graphs was first proposed by famous mathematician Leonhard Euler in his "Seven Bridges of Königsberg" paper (Paoletti, 2011). Euler observed that the city of Königsberg, in Prussia had seven bridges which connected the two large islands of Kneiphof and Lomse to the each other and the mainland. In his publication, Euler proposed that there was no way to walk through the city by crossing each bridge exactly one time (Paoletti, 2011). His work marked the beginning of the branch of mathematics known as graph theory.

Graphs come in shapes and forms, such as Tree Graphs, where each pair of vertices is connected by exactly one edge, or weighted graphs, where each edge has specific value that has to be expended to travel across. However, the two most simple and common type of graphs are directed and undirected graphs. The edges within a directed graph denote that the path can only be crossed in a specified direction and an undirected graph contains edges that are bidirectional.

In the present, the study of graph theory also has many real world applications. An example of which would be studying the spread of infectious diseases such as the Coronavirus in the field of epidemiology or criminal networks in criminology (Levenkova, 2014). The Dijkstra's Shortest Path Algorithm, which will be investigated in this paper has also been implemented as the primary algorithm for finding the shortest path in Google Maps (Crovari, 2019). New research is constantly being conducted on the development of faster and more efficient algorithms that can traverse graphs.

Even though there are many different problems involving graph theory, one of the most common and important issue to this day would be the “single source shortest path problem”. The problem requires an algorithm to find the shortest path between a given source node, and every other node within the graph (Chapuis & Andonov, 2017). In order for an algorithm to accomplish this, it must be provided a list of all the edges between nodes as well as the source node where the query must be conducted. There are currently two contenders for the most efficient single source shortest path algorithm, namely the Dijkstra algorithm (Richards, 1972), first proposed by computer scientist Edsger W. Dijkstra in 1956 and the Shortest Path Faster Algorithm (SPFA), an improvement to the Bellman-Ford Algorithm made by Chinese researcher FanDing Duan (玉人, 2010).

This investigation aims to compare and contrast the efficiency of both graph theory algorithms by comparing their time complexity at various input data sizes and record the performance of each algorithm over a series of a hundred trials.

1.1: Research Question

To what extent is the Shortest Path Faster Algorithm more efficient than Dijkstra’s Shortest Path Algorithm at various number of nodes and edges within randomly generated directed graphs with weighted edges.

1.2: Hypothesis

The SPFA algorithm is predicted to have a better average run time on randomly generated graphs because it utilizes an array to keep track of nodes that is currently in the queue to avoid repeating nodes. Therefore, this would reduce the number of nodes within a queue at any given time and improve the time complexity on randomly generated graphs. Hence, it is hypothesized that the Shortest Path Faster Algorithm will have a fast average run time at all data sizes compared to Dijkstra's Single Source Shortest Path Algorithm.

2: Background Information

2.1: Algorithmic Efficiency

To measure the algorithmic efficiency of both methods, the time and space complexities must be taken into consideration. However, as most modern-day computers have at least 4 Gigabytes of internal Randomly Accessed Memory (RAM), the issue of memory allocation will not be a factor considered in this investigation. Therefore, the primary dependent variable of this paper would be the time complexity of the two algorithms. The time complexity of most algorithms is divided into three parts, the best case, average case and worst cast time complexity. This indicates the theoretical upper and lower bound of the algorithm's run time as well as the average time taken on randomly generated data sets.

The time complexity of an algorithm can be denoted by the "Big O Notation" in the form of O(N). The number within the brackets following the capital 'O' denotes the theoretical time complexity of the algorithm based on the size of the input, which is generally represented by the capital letter 'N'. In the case of both SPFA and Dijkstra's algorithm, the time complexity is not only dependent only on one input value but two. The run time of both of algorithms are dependent on both the number of nodes and edges within a graph. Hence the time complexity can be given as O(MN) where N is the number of nodes and M is the number of edges. Most of the time, the time complexity of an algorithm can be determined by the analyzing the number of operations that the computer needs to conduct. Take the example of a linear searching algorithm designed to find an element in an array. If the array contains 'N' elements, then the time complexity of the algorithm would be O(N), because in the worst-case scenario, the algorithm will have to search through all N elements before it can find the desired result. On the other hand, if the desired element is the first element in the array, then it can be found in O(1) time which is also the best-case time complexity of the algorithm. The average time of an algorithm, however, is much harder to determine, as different input could result in different average time complexities. But assuming all the input data is random, then the average time complexity of the algorithm should be O(N/2) as on average, the algorithm will have to traverse half of the entire array.

2.2: Dijkstra's Algorithm

2.2.1: Algorithm Explanation

Dijkstra's shortest path algorithm employs the use of Breadth First Search (BFS) to determine the distance between a source node and every other node within a graph. Dijkstra's algorithm works by utilizing a first in first out data structure, known as a Queue, which keeps track of all the nodes that is being processed. While the queue is not empty, the top node in the queue is popped out and denoted as the current node. After this has been completed, the algorithm checks for all the nodes the current node is connected to and adds them all to the queue. This process is repeated until every node in the graph has been updated and the shortest distance between the source node and every other node in the graph can be determined (Gregg, 2018).

2.2.2: Time Complexity

The theoretical time complexity of the algorithm is $O(V^2)$ where V is the number of vertices within the graph. However, by implementing a priority queue instead of a normal queue, the time complexity can be optimized to $O(E\log(V) + V)$ where E is the number of edges. This algorithm can be further optimized to $O\big(E + V\log(V)\big)$ through the use of a combination of Priority Queue and the Fibonacci heap data structure. For the purpose of this investigation, the Priority Queue Dijkstra will be tested.

2.2.3: Implementation

The Dijkstra algorithm presented above can be implemented as shown in the program below:

```java
class Graph {
    class Edge implements Comparable<Edge>{
        int node;
        int weight;
        Edge(int node, int weight) {
            this.node = node;
            this.weight = weight;
        }

        public int compareTo(Edge o) {
            return this.weight - o.weight;
        }
    }
    int N;
    ArrayList<LinkedList<Edge>> adj = new ArrayList<>();
    boolean[] visited;
    int[] distance;
    Graph(int N) {
        this.N = N;
        visited = new boolean[N];
        distance = new int[N];
        for (int i = 0; i < N; i++) {
            adj.add(new LinkedList<>());
        }
    }
    void add(int source, int dest, int weight) {
        adj.get(source).add(new Edge(dest,weight));
        adj.get(dest).add(new Edge(source,weight));
    }
    void dijkstra(int root) {
        PriorityQueue<Edge> q = new PriorityQueue<>();

        Arrays.fill(distance,Integer.MAX_VALUE);
        distance[root] = 0;
        q.add(new Edge(root,0));

        while (!q.isEmpty()) {
            int a = q.remove().node;
            if (visited[a]) continue;
            visited[a] = true;

            for (Edge e: adj.get(a)) {
                int b = e.node;
                int w = e.weight;
                if (distance[a] +w < distance[b]) {
                    distance[b] = distance[a] + w;
                    q.add(new Edge(b,distance[b]));
                }
            }
        }
    }
}
```

2.3: Shortest Path Faster Algorithm (SPFA)

2.3.1: Algorithm Explanation

The Shortest Path Faster Algorithm is an improvement to the Bellman-Ford shortest path algorithm and involves the use of lazy propagation to “relax” vertices. Similar to Dijkstra, SPFA also finds the shortest distance between a single source node and every other node in the graph. Furthermore, SPFA also maintains a queue to keep track of all the nodes that are going to be visited. However, it also has an array to check if an element is already in the queue. If the element is already in the queue, it will not be added (玉人, 2010). This reduces the number of nodes that must be checked and is able to improve the time complexity of the algorithm.

2.3.2: Time Complexity

The theoretical worst case time complexity would be $O(V \times E)$ (玉人, 2010). Even though not enough research has been done to confirm this, experimental data suggests that the average time complexity of the algorithm on randomly generated graphs is near $O(E)$ (玉人, 2010). Furthermore, SPFA is always preferred over the original Bellman-Ford algorithm because it is significantly faster on randomly generated graphs and has the same worst case time complexity.

2.3.3: Implementation

The SPFA algorithm presented above can be implemented as shown in the program below:

```java
int V = sc.nextInt(), E = sc.nextInt();
ArrayList<ArrayList<Edge>> adj = new ArrayList<ArrayList<Edge>>();
for (int i = 0; i <= V; i++) {
    adj.add(new ArrayList<Edge>());
}
int dis[] = new int[V + 1];
Arrays.fill(dis, 0x3f3f3f3f);
boolean inQ[] = new boolean[V + 1];
Arrays.fill(inQ, false);

for (int i = 1; i <= E; i++) {
    int u = sc.nextInt(), v = sc.nextInt(), w = sc.nextInt();
    adj.get(u).add(new Edge(v, w));
    adj.get(v).add(new Edge(u, w));
}

LinkedList<Integer> q = new LinkedList<Integer>();
dis[1] = 0;
inQ[1] = true;
q.add(1);

while (!q.isEmpty()) {
    int u = q.poll();
    inQ[u] = false;
```

```java
        for (Edge e : adj.get(u)) {
            if (dis[u] + e.w < dis[e.v]) {
                dis[e.v] = dis[u] + e.w;
                if(inQ[e.v] == false) {
                    q.add(e.v);
                    inQ[e.v] = true;
                }
            }
        }
    }
```

2.4: Data Set Constraints

The processing power of computers are measured in Million Instructions per Second (MIPS), which denotes the number of operations the computer can compute each second. As this value is different for every processor, a simple program can be written to calculate the MIPS.

The following simple Java program determines the MIPS of a given processor by calculating the starting time and stopping until one second has completed. The program calls the built in Java function, "System.nanoTime()" which returns the current time in nanoseconds (ns). Therefore, while the difference between the current time and starting time is less than 109 ns, or exactly one second, the counter will increment by one. After the time period has concluded, the number of instructions done by the processor will be printed out.

```java
long startTime = System.nanoTime(), count = 0;
while(System.nanoTime() - startTime < 1e9) {
        count++;
}
System.out.println(count);
```

After conducting exactly one hundred trails on the average instructions per second, the following results were recorded. Only the first ten trials are shown.

Table 1: Million Instructions per Second of a I5-6600K 3.50GHz CPU

Trials	Million Instructions per Second
1	0.044981391
2	0.046862592
3	0.046891600
4	0.046870524
5	0.046744232
6	0.046890223
7	0.046893581
8	0.046886984
9	0.046890159
10	0.046808171
Average	0.0466719457

As shown in the primary data above, the processor used for this experiment can execute around 0.0467 million instructions each second, which is around 4.6672 × 107 individual operations. Therefore, to determine a visible difference in the run time of the two algorithms, each will be tested with rigorous amounts of data. Based on the theoretical time complexity mentioned in the previous section, both algorithms will be tested with data set of at most 106 vertices and edges to ensure that there will be visible differences in the run time. However, as many real-world applications of this algorithm does not require such large data sets, both algorithms will also be tested at 105 and 104 vertices and edges.

3: Experimental Methodology

The main source of data used for this investigation comes from primary sources of data. Both Dijkstra and SPFA were programmed in Java and were fed random locally generated data at the specified constraints. Primary sources of data were chosen for this experiment because SPFA has limited secondary and tertiary data and no comprehensive research has been conducted to verify its average time complexity. Furthermore, primary data collection allows for the manipulation of the independent variables involved in the algorithm. However, this experimental methodology also comes with its limitations. The scope of this investigation was severely limited as all the code used for data collection was executed on a home desktop with mediocre computational power, hence larger data sets could not be tested.

3.1: Experimental Procedure

The experimental procedure for both the Dijkstra's algorithm and SPFA are relatively similar. Both programs start off by generating the test data and printing it off to a ".txt"file. Then the data is scanned in my the program and the timer begins. The program will either execute Dijkstra's algorithm or SPFA and when the program has stopped running, the run time will be printed out. This procedure is clearly demonstrated in the flow chart below.

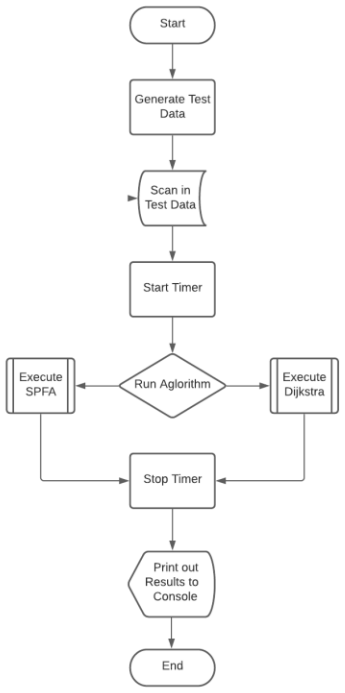


Figure 1: Visualization of Experimental Procedure

The procedure can also be demonstrated using Pseudo code as shown below.

```
int t = readInteger();
for(int i = 0; i < t; i++) {
    generateTestData();
    int v = readInteger();
    int e = readInteger();
    for(int i = 0; i < e; i++) {
        int start = readInteger();
        int end = readInteger();
        int dist = readInteger();
        makeGraph();
    }
    long timeStart = System.nanoTime();
    // Run Code
    long timeEnd = System.nanoTime();
    System.out.println(timeEnd - timeStart);
}
```

3.2: Measurement of Data

To measure the time complexity of both algorithms at varying sizes, Java's System.nanoTime() can be implemented. The total run time can be determined by finding the difference in the initial value recorded before the program was run and the final value.

```java
 long start = System.nanoTime();
// Execute Code
long end = System.nanoTime();
System.out.println("Total Run Time in Nanoseconds is: " + (end-start))
```

3.3: Experimental Accuracy

The theoretical experimental accuracy of the System.nanoTime function should be $\pm 1\text{ns}$. However, this appears to not be the case as repeated trials of the functions indicated that the rightmost three digits of the returned value is always zero. This results in the experimental accuracy of the function to be $\pm 1\mu\text{s}$. As this is around 0.001% of the experimental results, the uncertainty values can be neglected.

3.4: Variables

3.4.1: Independent Variable

The independent variable of this investigation is the data constraints of the datasets that is being inputted into the algorithms. The amount of input size will directly correlate to the time complexity.

3.4.2: Dependent Variable

The dependent variable of this investigation is the time complexity of the algorithm measured in seconds. The run time will be directly affected by changes in the input size, making it the dependent variable.

3.4.3: Controlled Variable

1. Processor:

The same processor must be used to compute both the programs as different processing units can compute different number of instructions per second.

2. Programming Language:

The programming language used to code and execute the algorithms must also be kept the same as different programming languages take different amount of time to compile and execute code.

3. Measurement of Time:

To ensure that the time taken is measured correctly, the time must be measured after the data has been generated and inputted to ensure that the collected data has high accuracy

4: Experimental Results

4.1: Tabular Representation of Experimental Data

The following table displays the performance of both the Dijkstra's Algorithm and Shortest Path Faster Algorithm at various data sizes from 106 to 104 data values. Furthermore, the final two graphs show the correlation between input data size and time taken of both programs.

For each data set, only the first ten out of one fifty trials were shown and the average time for the trials were shown on the final line of each table. Furthermore, a large number of decimal places were used to maintain high accuracy in the recorded results, as the difference between a selective number of trails were fairly minuscule.

Table 1: Time Complexity of Dijkstra and SPFA at 1, 000, 000 Data Values

Trials	Shortest Path Faster Algorithm (s)	Dijkstra's Algorithm (s)
1	2.49675	2.20985
2	2.08050	3.35393
3	1.69652	3.07306
4	1.93442	2.46844
5	2.00193	2.50254
6	1.94521	2.36200
7	1.55685	2.39478
8	1.52814	2.49976
9	1.96018	2.15172
10	1.87326	2.13480
Average	1.90738	2.51509

Table 2: Time Complexity of Dijkstra and SPFA at 100, 000 Data Values

Trials	Shortest Path Faster Algorithm (s)	Dijkstra's Algorithm (s)
1	0.26428	0.25972
2	0.20451	0.19840
3	0.14483	0.17919
4	0.17780	0.18377
5	0.12722	0.24069
6	0.18332	0.15402
7	0.16731	0.20284
8	0.16175	0.24611
9	0.15707	0.15519
10	0.11558	0.17455
Avg	0.17037	0.19945

Table 3: Time Complexity of Dijkstra and SPFA at 10, 000 Data Values

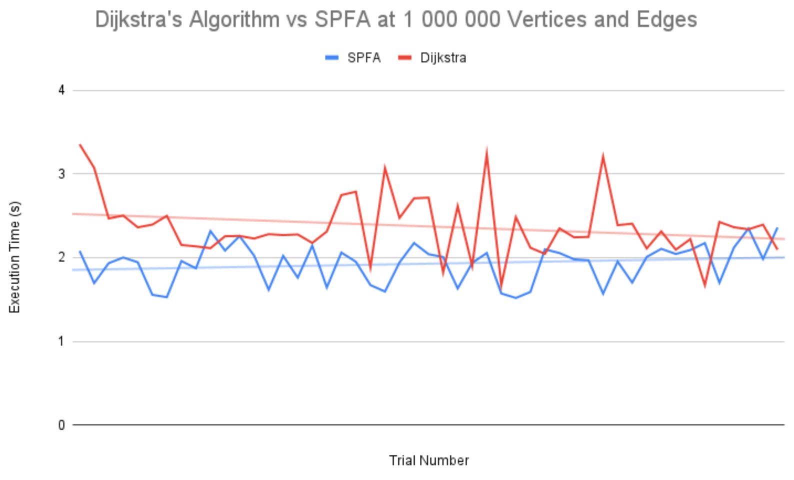
Trials	Shortest Path Faster Algorithm (s)	Dijkstra's Algorithm (s)
1	0.05457	0.05463
2	0.03146	0.03753
3	0.01957	0.01984
4	0.02277	0.01818

5	0.01906	0.01786
6	0.02073	0.02173
7	0.02041	0.02026
8	0.01885	0.0197
9	0.01494	0.01512
10	0.01541	0.01494
Avg	0.02378	0.02398

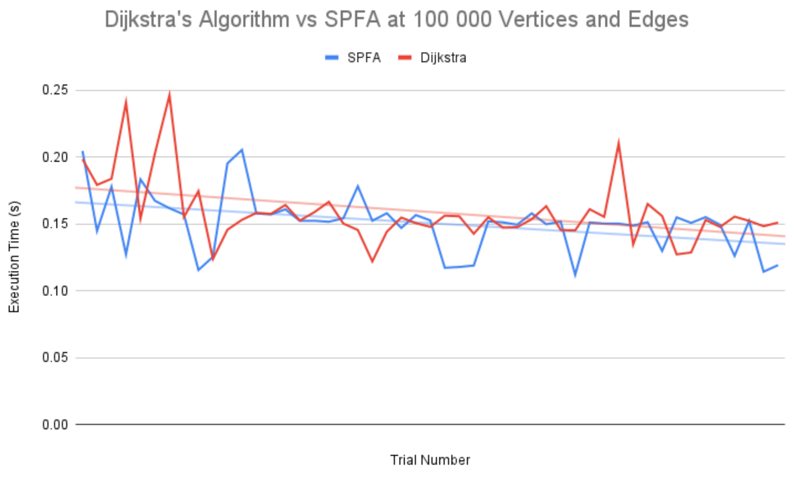
4.2: Graphical Representation of Data

In order to better understand the trends in performance, the collected experimental data has been converted into a double line graph. The red line denotes the performance of the Dijkstra' Algorithm and the blue line represents SPFA. The x-axis of the graph is the trial number and has no units, while the y-axis is the amount of time taken to complete execution of the program and is measured in seconds.

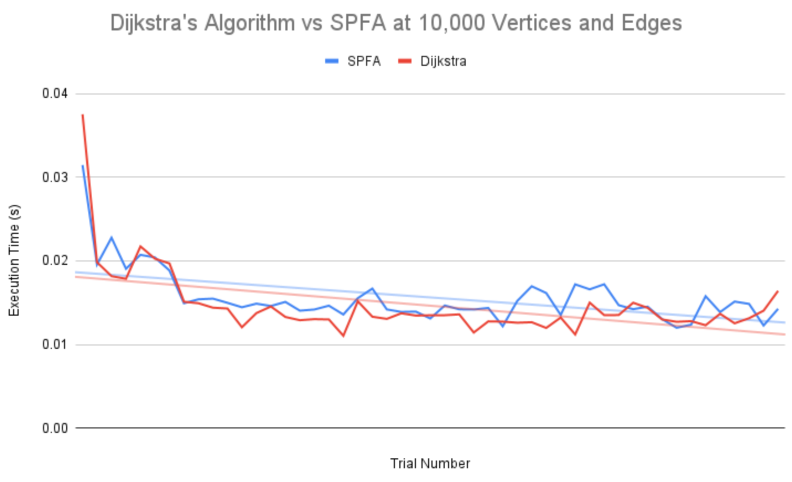
The correlation between input values and run time can be shown in the final two graphs. The x-axis of the graph is the number of data values inputted into the program and the y-axis is the run time measured in seconds.



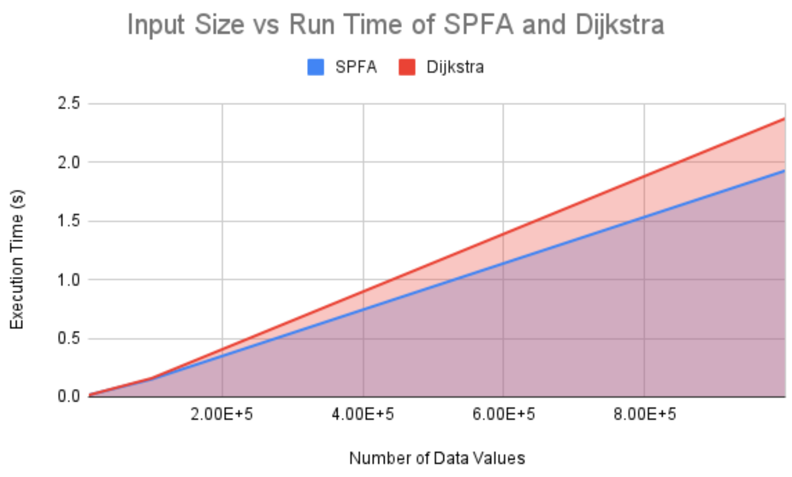
<u>Graph 1: Dijkstra and SPFA Performance at 1, 000, 000 Data Values</u>



<u>Graph 2: Dijkstra and SPFA Performance at 100, 000 Data Values</u>



<u>Graph 3: Dijkstra and SPFA Performance at 10, 000 Data Values</u>



<u>Graph 4: Correlation between Input Size and Run Time of Dijkstra and SPFA</u>

4.3: Data Analysis

The collected data is very conclusive with regards to the performance of both algorithms at various data sizes. In both the tabular and graphical representation of the collected data, it is evident that SPFA and Dijkstra have extremely similar run times at smaller data sizes. However, as the number of data values increases, so did the difference in run time. When the graph tested has 10^6 vertices and edges, the average run time of SPFA is nearly half a second faster.

4.3.1: Determining Experimental Average Time Complexity

In an earlier section of this paper, the processing speed of the computer used was determined to be around 4.6672 × 107 instructions per second. Therefore, given the number of instructions and time taken, the average time complexity of the algorithms can be calculated by the following algorithm:

$$\text{Number of Operations} = \text{Time (s)} \times \text{Instructions per Second}$$

A sample calculation for the number of operations of SPFA at 106 data values is shown below:

$$\begin{aligned}\text{Number of Operations} &= 1.9074 \times 4.6672 \times 10^7 \\ &= 8.9022 \times 10^7 \\ &= 89.022\ \text{N}\end{aligned}$$

Given the input size of the test was 106 data values, it can be shown that the number of operations done by the computer is around 89 times the input values. Therefore, the experimental average time complexity of SPFA at 106 data values would be O(89N). The following tables indicates the average time complexity of both algorithms at all three recorded data ranges.

<u>Table 2: Time Complexity of Dijkstra and SPFA at 100, 000 Data Values</u>

Number of Data Values	SPFA Time Complexity	Dijkstra Time Complexity
1, 000, 000	O(89N)	O(110N)
100, 000	O(79N)	O(93N)
10, 000	O(111N)	O(112N)
Average	O(93N)	O(105N)

As seen from the table above, the average experimental time complexity of both algorithm is around 102 times the number of input values. Generally, a relatively small constant multiplier can neglected. Therefore, the average time experimentally determined time complexity of both algorithms could be simplified to O(N).

4.3.2: Run Time Disparity

As seen in the final graph which shows the correlation between the run time of the two algorithms at different data sizes, as the number of input value increase, so does the run time disparity between the two algorithms. It is conclusive from the collected data that the Shortest Path Faster Algorithm outperforms Dijkstra's Shortest Path Algorithm at all three tested data values and is far more efficient at higher data sizes.

4.3.3: Inconsistency

Another interesting observation that can be made from the results of the comparison between the two algorithms would be that both algorithms are rather inconsistent. Take the example of the performance of SPFA at 10^6 data values. The error range of the experimental data is large as shown in the calculations below:

$$\begin{aligned}\text{Error Range} &= \frac{\text{Max} - \text{Min}}{2} \\ &= \frac{2.49675 - 1.52814}{2} \\ &= 0.48431\end{aligned}$$

The error range of 0.48431 account for a relative error range of over 25%. This clearly indicates that both algorithms are rather inconsistent and have large disparity in time complexity in randomized graphs between best and worst cases.

A possible explanation to the inconsistency noticed in the two algorithms would be edge cases. Take the example of SPFA, the algorithm adds the current nodes in a queue into a visited array to avoid adding repeated vertices into the queue. This is particularly efficient when a node has many connected edges to it. However, if many of the nodes only have a single connected edge, then the optimization would prove to be less than impactful.

5: Further Research Questions

5.1: Graphs with Negative Weight Edges

This investigation mostly focused on determining the efficiency of Dijkstra's shortest path algorithm and Shortest Path Faster Algorithm (SPFA) in bidirectional non-negative weighted graphs at various data set constraints. However, further research can be conducted on the performance of these algorithms in graphs with negative weight edges. Having negative weight as edges can significantly impact the efficiency of the algorithm or even render some useless as it affects the calculation of minimum distance.

5.2: Directional Graphs

The graphs investigated in this experiment were all bidirectional, meaning each edge can be travelled in both directions. However, this may affect the efficiency of algorithms if edges can only be travelled in a single direction, as it significantly limits the number of possible paths.

5.3: Cyclic Graphs

Another possible expansion to this research topic would be to investigate whether graphs with cycles will affect the time complexity of the algorithm. There is also the possibility that some of the algorithms will become stuck in an infinite loop within the cycle and eventually crash the program

6: Conclusion

In this paper, two single source shortest path algorithm, namely the Shortest Path Faster Algorithm (SPFA) and Dijkstra’s Shortest Path Algorithm were tested with randomly generated graphs at three different data sizes to determine the average run time over fifty trials. The experimental time complexity of both algorithms were also calculate and mathematical explanations for the observations made were also provided.

From the experimental results, the average experimental time complexity were determined to be O(93N) for SPFA and O(105N) for Dijkstra. The time complexity can be simplified to be O(N) as the constant multiplier is rather small. Furthermore, the data conclusively indicate that SPFA is more efficient that Dijkstra at all three tested data sizes and the trend indicates that the more data values inputted, the greater the run time disparity between the two algorithms.

However, both algorithms are noticed to be rather inconsistent with large disparities between their best case and worst case run times. This could be problematic at larger data values as the best- and worst-case scenarios can results in large differences in run times.

7: Works Cited

Most of the references made were only utilized in the introduction and background information sections of this paper.

Paoletti, T. (2011, May). Leonard Euler's Solution to the Konigsberg Bridge Problem . Mathematical Association of America. Retrieved September 12, 2021, from https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem.

Chapuis, G., & Andonov, R. (2017). GPU-accelerated shortest paths computations for planar graphs. Shortest Path Problem - An Overview . Retrieved September 12, 2021, from https://www.sciencedirect.com/topics/computer-science/shortest-path-problem.

Cornell University. (n.d.). CS 312 lecture 20 Dijkstra's shortest path algorithm. Shortest Path Algorithm. Retrieved September 12, 2021, from https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/lec20/lec20.htm.

Crovari, P. (2019, September 18). Google maps and graph theory. Impactscool Magazine. Retrieved September 12, 2021, from https://magazine.impactscool.com/en/speciali/google-maps-e-la-teoria-dei-grafi/.

Gregg, C. (2018). CS 106B. Lecture 25: Dijkstra's Algorithm and the A* Algorithm. Retrieved September 11, 2021, from https://web.stanford.edu/class/archive/cs/cs106b.1186/lectures/24-Dijkstra_AStar/24-Dijkstra_AStar.pdf.

Levenkova, N. (2014, March). Applications of graph theory to real-world networks. School of Mathematics and Statistics Faculty of Science. Retrieved September 11, 2021, from http://unsworks.unsw.edu.au/fapi/datastream/unsworks:12628/SOURCE02?view=true#:~:text=We%20apply%20graph%20theory%20to,gation%20of%20sexually%20transmitted%20infections.

Oracle. (2020, June 24). Class System. System (Java Platform SE 7). Retrieved September 12, 2021, from https://docs.oracle.com/javase/7/docs/api/java/lang/System.html.

Richards, H. (1972). Edsger Wybe Dijkstra. Edsger W. Dijkstra - A.M. Turing Award Laureate. Retrieved September 12, 2021, from https://amturing.acm.org/award_winners/dijkstra_1053701.cfm.

Richardson, T. (n.d.). A Discovery Algorithm for Directed Cyclic Graphs. Philosiphy Departement. Retrieved September 11, 2021, from https://arxiv.org/ftp/arxiv/papers/1302/1302.3599.pdf.

Weisstein, E. W. (2021, September 8). Graph. Retrieved September 12, 2021, from https://mathworld.wolfram.com/Graph.html.

玉人 香袭. (2010, December 10). 国产算法,求最短路径问题. SPFA 算法. Retrieved September 12, 2021, from https://wenku.baidu.com/view/f8344b1352d380eb62946d5a.html?fixfr=QO%252BzpmqBV8qHImvfF2Q6Ug%253D%253D&fr=income1-wk_sea_vip-search.

8: Appendix

The following code was used for my experiments. I used them to generate the random test data, which were then fed into the Dijkstra and SPFA algorithms. Each program has a single aspect that can be changed, which is the number of edges and vertices. The only input required for the program would be to enter the number of trials to be conducted and the program will output the time complexity for each trial as well as well as the average time taken over all the trails at the end.

The following code were compiled in IntelliJ IDEA 2021.2.1 Community Edition on a i5500k home desktop with 8GB of Randomly-Accessed Memory (RAM).

8.1: Dijkstra

```
import java.util.*;
import java.io.*;
class Graph {
    class Edge implements Comparable<Edge>{
        int node;
        int weight;
        Edge(int node, int weight) {
            this.node = node;
            this.weight = weight;
        }

        public int compareTo(Edge o) {
            return this.weight - o.weight;
        }
    }
    int N;
    ArrayList<LinkedList<Edge>> adj = new ArrayList<>();
    boolean[] visited;
    int[] distance;
    Graph(int N) {
        this.N = N;
        visited = new boolean[N];
        distance = new int[N];
        for (int i = 0; i < N; i++) {
            adj.add(new LinkedList<>());
```

```java
        }
    }
    void add(int source, int dest, int weight) {
        adj.get(source).add(new Edge(dest,weight));
        adj.get(dest).add(new Edge(source,weight));
    }
    void dijkstra(int root) {
        PriorityQueue<Edge> q = new PriorityQueue<>();

        Arrays.fill(distance,Integer.MAX_VALUE);
        distance[root] = 0;
        q.add(new Edge(root,0));

        while (!q.isEmpty()) {
            int a = q.remove().node;
            if (visited[a]) continue;
            visited[a] = true;

            for (Edge e: adj.get(a)) {
                int b = e.node;
                int w = e.weight;
                if (distance[a] +w < distance[b]) {
                    distance[b] = distance[a] + w;
                    q.add(new Edge(b,distance[b]));
                }
            }
        }
    }
}

public class finalDijkstra {
    public static final int mx = (int) Math.pow(10, 3) - 1;
    public static final int d = (int) Math.pow(10, 3) - 1;
    public static double sum = 0.0;
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        double trials = in.nextDouble();
        for(int f = 0; f < (int) trials; f++) {
            PrintWriter pw = new PrintWriter("test-data.txt");
            pw.println(mx +  " " + mx);
            for(int i = 0; i < mx; i++) {
                pw.print(GenerateRandInt(mx) + " ");
                pw.print(GenerateRandInt(mx) + " ");
                pw.println(GenerateRandInt(d));
            }
            pw.close();

            long start = System.nanoTime(); // track start time
            Scanner sc = new Scanner(new File("test-data.txt"));
            int N = sc.nextInt();
            int M = sc.nextInt();
            Graph g = new Graph(N+1);
            for (int i = 0; i < M; i++) {
                int u = sc.nextInt();
                int v = sc.nextInt();
                int w = sc.nextInt();
                g.add(u,v,w);
            }
            g.dijkstra(1);
            long end = System.nanoTime(); // track the final time
            System.out.println("Trial: " + f);
            System.out.println(roundNum((end-start)/Math.pow(10, 9)));
            sum += (end-start)/Math.pow(10, 9);
            System.out.println();
        }
        System.out.println("Average: " + roundNum(sum/trials));
    }
    public static double roundNum (double x) {
        return Math.round(x * 100000.0) / 100000.0;
    }
    public static int GenerateRandInt(int bound) {
        Random rand = new Random();
        return rand.nextInt(bound);
    }
}
```

8.2: SPFA

```
import java.util.*;
import java.io.*;
public class SPFA {
    public static final int mx = (int) Math.pow(10, 3) - 1;
    public static final int d = (int) Math.pow(10, 3) - 1;
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        double trials = in.nextDouble(), sum = 0.00;
        for(int f = 0; f < (int) trials; f++) {
            PrintWriter pw = new PrintWriter("test-data.txt");
```

```java
        pw.println(mx +  " " + mx);

        for(int i = 0; i < mx; i++) {
            pw.print(GenerateRandInt(mx) + " ");
            pw.print(GenerateRandInt(mx) + " ");
            pw.println(GenerateRandInt(d));
        }
        pw.close();
        long start = System.nanoTime();
        Scanner sc = new Scanner(new File("test-data.txt"));
        int V = sc.nextInt();
        int E = sc.nextInt();
        ArrayList<ArrayList<Edge>> adj = new ArrayList<ArrayList<Edge>>();
        for (int i = 0; i <= V; i++) {
            adj.add(new ArrayList<Edge>());
        }
        int dis[] = new int[V + 1];
        Arrays.fill(dis, 0x3f3f3f3f);
        boolean inQ[] = new boolean[V + 1];
        Arrays.fill(inQ, false);
        for (int i = 1; i <= E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            int w = sc.nextInt();
            adj.get(u).add(new Edge(v, w));
            adj.get(v).add(new Edge(u, w));
        }
        LinkedList<Integer> q = new LinkedList<Integer>();
        dis[1] = 0;
        inQ[1] = true;
        q.add(1);
        while (!q.isEmpty()) {
            int u = q.poll();
            inQ[u] = false;
            for (Edge e : adj.get(u)) {
                if (dis[u] + e.w < dis[e.v]) {
                    dis[e.v] = dis[u] + e.w;
                    if(inQ[e.v] == false) {
                        q.add(e.v);
                        inQ[e.v] = true;
                    }
                }
            }
        }
        long end = System.nanoTime();
```

```
                System.out.println("Trial: " + f);
                System.out.println(roundNum((end-start)/Math.pow(10, 9)));
                sum += (end-start)/Math.pow(10, 9);
                System.out.println();
            }
            System.out.println("Average: " + roundNum(sum/trials));
        }
        public static double roundNum (double x) {
            return Math.round(x * 100000.0) / 100000.0;
        }
        public static int GenerateRandInt(int bound) {
            Random rand = new Random();
            return rand.nextInt(bound);
        }
        public static class Edge {
            int v, w;
            public Edge(int v, int w) {
                this.v = v;
                this.w = w;
            }
        }
}
```

8.3: Additional Raw Data

The following raw data was collected during the data collection process and was used to process and structure arguments. The data was also present within the graphs presented but was not presented within this investigation.

<u>Table 5: Time Complexity of Dijkstra and SPFA at 1, 000, 000 Data Values</u>

Trials	Shortest Path Faster Algorithm (s)	Dijkstra's Algorithm (s)
11	1.87326	2.13480
12	2.31678	2.11351
13	2.08474	2.25621

14	2.25749	2.25914
15	2.02466	2.22848
16	1.61726	2.28105
17	2.02128	2.26954
18	1.76161	2.27725
19	2.14388	2.17608
20	1.64487	2.31275
21	2.06083	2.74767
22	1.95100	2.78806
23	1.67254	1.88384
24	1.59536	3.06979
25	1.94341	2.47521
26	2.17494	2.70911
27	2.04116	2.71700
28	2.00768	1.82287
29	1.63230	2.61581
30	1.93848	1.90082
31	2.05485	3.23478
32	1.57283	1.67237

33	1.51883	2.48384
34	1.59099	2.11852
35	2.09933	2.04606
36	2.05574	2.3502
37	1.98053	2.24287
38	1.96842	2.24802
39	1.57081	3.20362
40	1.95412	2.38787
41	1.70259	2.40637
42	2.00899	2.11040
43	2.10560	2.31433
44	2.04451	2.09513
45	2.09123	2.22178
46	2.17415	1.67255
47	1.70117	2.42638
48	2.12134	2.36321
49	2.34827	2.33865
50	1.98489	2.39429

Table 6: Time Complexity of Dijkstra and SPFA at 100, 000 Data Values

Trials	Shortest Path Faster Algorithm (s)	Dijkstra's Algorithm (s)
11	0.01541	0.01494
12	0.01549	0.01441
13	0.01499	0.01431
14	0.01446	0.01207
15	0.01489	0.01376
16	0.01461	0.01455
17	0.01510	0.0133
18	0.01405	0.01291
19	0.01419	0.01304
20	0.01465	0.01300
21	0.01359	0.01105
22	0.01555	0.01517
23	0.01669	0.01333
24	0.01420	0.01307
25	0.01390	0.01374
26	0.01394	0.01347
27	0.01312	0.01350
28	0.01467	0.01351

29	0.01421	0.01361
30	0.01419	0.01144
31	0.01438	0.01277
32	0.01219	0.01274
33	0.01521	0.01261
34	0.01697	0.01267
35	0.01616	0.01199
36	0.01354	0.01323
37	0.01720	0.01121
38	0.0166	0.01501
39	0.01721	0.01352
40	0.01471	0.01353
41	0.01423	0.01500
42	0.01453	0.01439
43	0.01304	0.01300
44	0.01201	0.01272
45	0.01238	0.01280
46	0.01578	0.01231
47	0.01388	0.01368

48	0.01514	0.01254
49	0.01486	0.01315
50	0.01228	0.01405

Table 7: Time Complexity of Dijkstra and SPFA at 10, 000 Data Values

Trials	Shortest Path Faster Algorithm (s)	Dijkstra's Algorithm (s)
11	0.11558	0.17455
12	0.12546	0.12384
13	0.19523	0.14568
14	0.20533	0.15307
15	0.15783	0.15824
16	0.15706	0.15752
17	0.16092	0.16418
18	0.15229	0.15251
19	0.15238	0.15880
20	0.15162	0.16653
21	0.15420	0.15029
22	0.17812	0.14548
23	0.15240	0.12203
24	0.15801	0.14421

25	0.14702	0.15477
26	0.15664	0.15061
27	0.15265	0.14781
28	0.11727	0.15620
29	0.11787	0.15570
30	0.11893	0.14266
31	0.15190	0.15498
32	0.15120	0.14732
33	0.14947	0.14763
34	0.15791	0.15354
35	0.14978	0.16328
36	0.15175	0.14547
37	0.11202	0.14525
38	0.15112	0.16101
39	0.15027	0.15529
40	0.15024	0.21013
41	0.14882	0.13467
42	0.15129	0.16492
43	0.12985	0.15591

44	0.15493	0.12726
45	0.15080	0.12870
46	0.15510	0.15295
47	0.14936	0.14779
48	0.12616	0.15547
49	0.15227	0.15228
50	0.11431	0.14831