A Computer Science Extended Essay

To what extent does the time complexity of Dijkstra’ s Algorithm and Shortest Path Algorithm (SPFA) differ on randomly generated unweighted bi-directional graphs?

Peter Wang

3976 Words

Contents

1 Introduction **2**
1.1 Research Question . . . 2
1.2 Hypothesis . . . 2

2 Background Information **3**
2.1 Algorithmic Efficiency . . . 3
2.2 Dijkstra's Algorithm . . . 3
2.2.1 Algorithm Explanation . . . 3
2.2.2 Time Complexity . . . 3
2.2.3 Implementation . . . 3
2.3 Shortest Path Faster Algorithm (SPFA) . . . 4
2.3.1 Algorithm Explanation . . . 4
2.3.2 Time Complexity . . . 5
2.3.3 Implementation . . . 5
2.4 Data Set Constraints . . . 5

3 Experimental Methodology **6**
3.1 Experimental Procedure . . . 7
3.2 Measurement of Data . . . 8
3.3 Experimental Accuracy . . . 8
3.4 Variables . . . 8
3.4.1 Independent Variable . . . 8
3.4.2 Dependent Variable . . . 8
3.4.3 Controlled Variable . . . 8

4 Experimental Results **8**
4.1 Tabular Representation of Experimental Data . . . 8
4.2 Graphical Representation of Data . . . 10
4.3 Data Analysis . . . 12
4.3.1 Determining Experimental Average Time Complexity . . . 12
4.3.2 Run Time Disparity . . . 13
4.3.3 Inconsistency . . . 13

5 Further Research Questions **14**
5.1 Graphs with Negative Weight Edges . . . 14
5.2 Directional Graphs . . . 14
5.3 Cyclic Graphs . . . 14

6 Conclusion **14**
6.1 Limitations of Scope . . . 14

7 Bibliography **15**

8 Appendix **15**
8.1 Dijkstra . . . 15
8.2 SPFA . . . 17

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 (Richardson, 2021, [6]). The concept of graphs was first proposed by famous mathematician Leonhard Euler in his "Seven Bridges of Königsberg" paper (Paoletti, 2011, [5]). 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, [5]). 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, [2]). 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 (Bajer, 2021, [1]). 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. 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 (Lindholm, 2013, [3]), 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 (郭建科, 2007, [7]).

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 $O(M \times N)$ 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 $O(\frac{N}{2})$ 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 (Macharis, 2018, [4]).

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 $O(E^+V\log(V))$ 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.

```
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 (郭建科, 2007, [7]). This reduces the number of nodes that must be checked and improves the time complexity of the algorithm.

2.3.2 Time Complexity

The theoretical worst case time complexity would be O($V \times E$) (郭建科, 2007, [7]). 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) where E is the number of edges within the graph. (郭建科, 2007, [7]). 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.

```
int V = sc.nextInt(), E = sc.nextInt();
ArrayList<ArrayList<Edge>> adj;
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;
    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

In order to determine the data set constraints that will be tested, the computational power of the computer must first be determined. The processing power of computers are measured in Million Instructions per Second (MIPS), which denotes the number of millions 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.

```
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. On the first ten trials of the raw data were presented in the table below. The collected data is recorded with a high level of precision as changes between trails are relatively small and could not be differentiated with if less number of significant figures were kept.

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

Trials	Million Instructions Per Second (MIPS)
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 an average of 0.0467 million instructions each second, which is around 4.6672×10^7 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 10^6 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 10^5 and 10^4 vertices and edges.

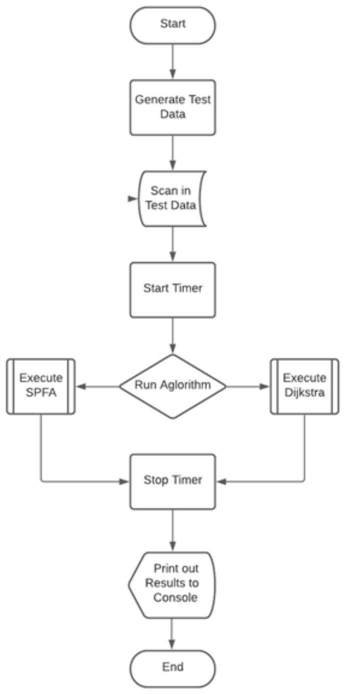
3 Experimental Methodology

The main source of raw data used for this investigation comes from primary sources. 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 complete 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 data sets with an order of magnitude larger than 10^7 could not be realistically 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 using a scanner, and once all the data has been stored locally, the stopwatch timer begins. The program will either execute Dijkstra' s algorithm or SPFA and the program will answer the randomly generated query and print out the distance between two random nodes within a graph. Once both programs have concluded its processing and outputted all the data values, the timer stops and the difference between the final and initial time values will be printed out to the console. This procedure is repeated for fifty unique trials to ensure the integrity of the data collected as well as account for any outliers in the data. The process described above can also be visually interpreted in the flow chart below.

Figure 1: Visualization of Experimental Procedure



The procedure can also be implemented as shown in the program 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 various data sizes, Java's built in System.nanoTime() function 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.

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

3.3 Experimental Accuracy

The theoretical accuracy of the System.nanoTime function should be ±1ns. However, after repeated trails of the function, the rightmost three digits in the produced time is always zero. This results in an experimental accuracy of the function to be ±1 microsecond. Furthermore, as this value only accounts for an 0.0001% error, and a difference of a single microsecond will not have significant real-world implications, therefore, it is unnecessary to include these values within the data processing.

3.4 Variables

3.4.1 Independent Variable

The independent variable of this investigation is the number of nodes and edges within the randomly generated graphs as they directly affect the time complexity of the algorithms.

3.4.2 Dependent Variable

The dependent variable of this investigation is the run time of the two algorithms measured in microseconds.

3.4.3 Controlled Variable

1. **Processor:** The same processor must be used to compute both the programs as different processing units have different number of MIPs and hence will make the recorded time complexity inaccurate.
2. **Programming Language:** The programming language used to code and executed the algorithms must also be kept the same as different programming languages take different amount of time to compile and execute the code. Hence, if different programming languages are used between trials and algorithms, the recorded test data is inconclusive.
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 data generation as well as the input process are not taken into the consideration for the time complexity.

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. The first ten trails for each independent variable is shown below and the rest of the raw data can be located within the appendix section of this investigation.

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 high degree of precision was maintained to ensure high accuracy in the recorded results, as the difference between a selective number of trails were minuscule.

Table 2: 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 3: 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	0.14483	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
Average	0.17037	0.17455

Table 4: Time Complexity of Dijkstra and SPFA at 10, 000 Data Values

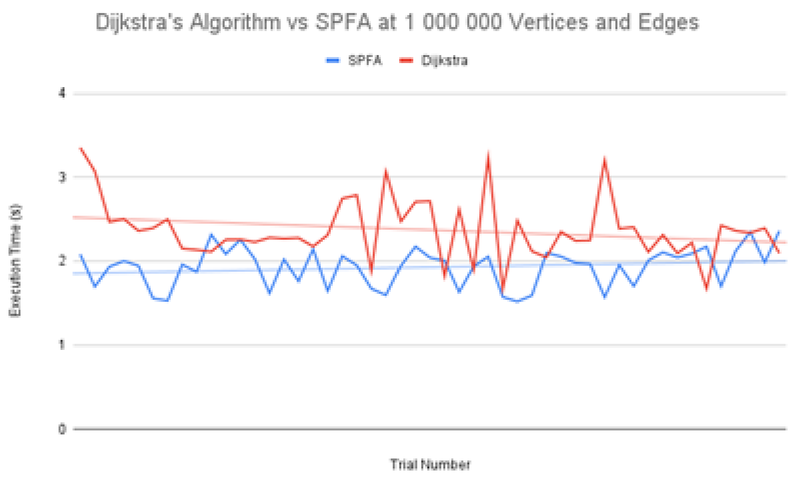
Trials	Shortest Path Faster Algorithm (s	Dijkstra's Algorithm (s)
1	0.05457	0.05463
2	0.03146	0.03753
3	0.01957	0.01984
4	0.02277	0.01818
5	0.01906	0.01786
6	0.02073	0.02173
7	0.02041	0.02026
8	0.01885	0.01970
9	0.01494	0.01512
10	0.01541	0.01494
Average	0.02378	0.02398

4.2 Graphical Representation of Data

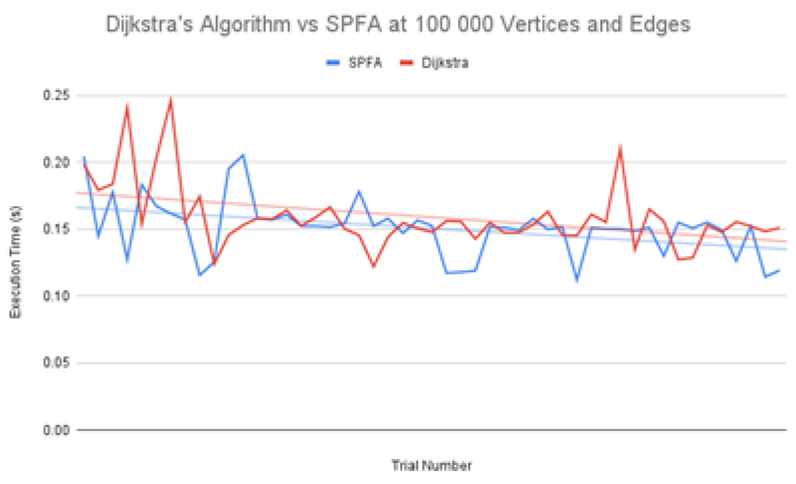
In order to better understand the trends in performance, the collected experimental data has been converted into a double line graph. The red line denotes the performance of the Dijkstra' Algorithm, and the blue line represents SPFA. The x-axis of the graph is the trial number and has no units, while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

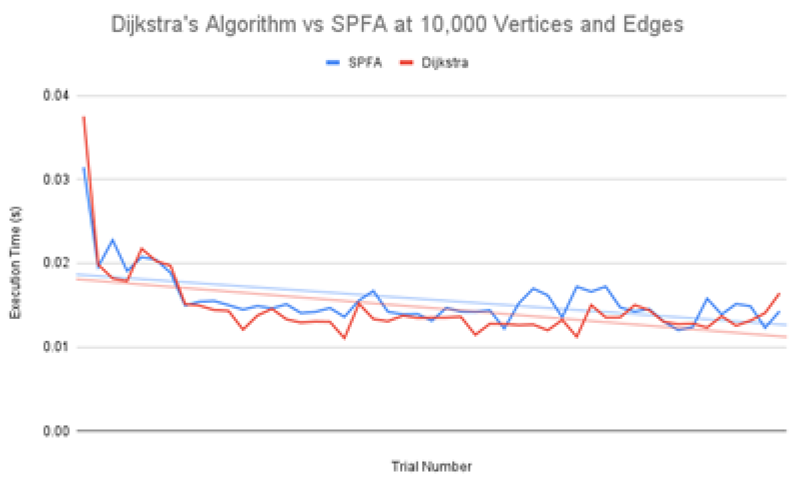
Graph 1: Dijkstra and SPFA Performance at 1, 000, 000 Data Values



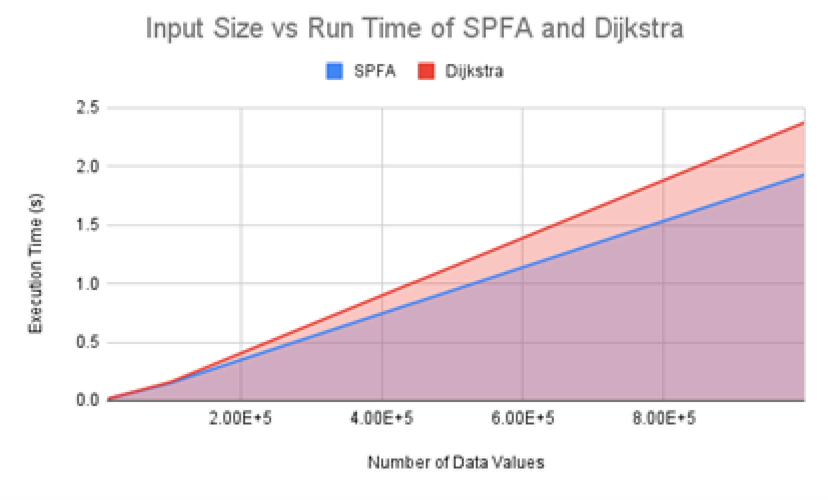
Graph 2: Dijkstra and SPFA Performance at 100, 000 Data Values



Graph 3: Dijkstra and SPFA Performance at 10, 000 Data Values



Graph 4: Correlation between Input Size and Run Time



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, the performance difference also became more evident. This can be seen when both algorithms were tested at 10^6.

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 4.6672×10^7 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:

$$\text{Number of Operations} = \text{Time (s)} \times \text{Instructions per Second} \tag{1}$$

A sample calculation for the number of operations of SPFA at 10^6 data values is shown below:

$$\begin{aligned}\text{Number of Operations} &= 1.9074s \times 4.6672 \times 10^7\\ &= 8.9022 \times 10^7\\ &= 89.022\text{N}\end{aligned} \tag{2}$$

Given the input size of the test was 10^6 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 10^6 data values would be O(89N). The following tables indicates the average time complexity of both algorithms at all three recorded data ranges.

Table 5: Time Complexity of Dijkstra and SPFA at various Data Sizes

Data Size	**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 a constant 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). As previously mentioned in the introduction section, the average time complexity of SPFA can approach O(N) on randomly generated graphs which matches the recorded average time complexity. However, Dijkstra' s Algorithm also averaged a time complexity relatively to O(N) which does not match the theoretical predicted values.

4.3.2 Run Time Disparity

As seen in Graph 4, 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. The collected data also demonstrates that the SPFA algorithm is far more consistent than Dijkstra at all recorded data values and has the better worst- and best-case time complexity.

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 rather significant at over 48% as shown in the calculations below:

$$\begin{aligned}\text{Error Range} &= \frac{\text{Max} - \text{Min}}{2} \\ &= \frac{2.49675 - 1.52814}{2} \\ &= 0.48431 \\ \frac{0.48431}{1.90738} &= 25.4\,\%\end{aligned} \tag{3}$$

The 25% error account for a relative error range which clearly indicates that both algorithms are rather inconsistent and have large disparity in time complexity in randomized graphs between the best- and worst-case scenarios.

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 as lazy propagation does not work on negative weight edges.

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 and will make keeping a queue with all the current nodes to be less significant at improving the overall time complexity.

5.3 Cyclic Graphs

Another possible expansion to this research topic would be to investigate whether graphs with cycles, which will also 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 and not produce an accurate result.

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 was also calculated and mathematical explanations for the observations made were also provided.

From the experimental results, the average experimental time complexity was 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 result in large differences in run times.

In conclusion, this investigation proved to be rather successful as both algorithms were tested rigorously using primary data and the time complexity were determined experimentally. The hypothesis of this investigation was verified and the disparity in time complexity was properly accounted for.

6.1 Limitations of Scope

The test data used for the purpose of this experiment was randomly generated locally for each trial which could result in weaker test data that does not accurately represent the performance of the algorithms. This issue can be solved by using literature test data values that contain significant amounts of edge cases to ensure that the algorithm can be tested to their full extents. If the processing power were to be improved, the number of total data values that can be tested can also be increased to investigate the performance of the algorithms at 10^7 or more data values.

7 Bibliography

Most of the references made were only utilized in the introduction and background information sections of this paper as the data generation and collection were done locally and no external sources were consulted

References

[1] Krystyna Bajer et al. "Graph Databases". In: *The Digital Journey of Banking and Insurance, Volume III*. Springer, 2021, pp. 35–49.

[2] Natalya Levenkova. "Applications of graph theory to real-world networks". In: (2014).

[3] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 7 Edition: Java Virt Mach Spec Java_3*. Addison-Wesley, 2013.

[4] Cathy Macharis, Dries Meers, and Tom Van Lier. "Modal choice in freight transport: combining multi-criteria decision analysis and geographic information systems". In: *International Journal of Multicriteria Decision Making* 5.4 (2015), pp. 355–371.

[5] Teo Paoletti. "Leonard Euler' s solution to the Königsberg bridge problem". In: *Convergence* (2011).

[6] Thomas S Richardson. "A discovery algorithm for directed cyclic graphs". In: *arXiv preprint arXiv:1302.3599* (2013).

[7] 郭建科 et al. "Dijktra 改进算法及其在地理信息系统中的应用". In: 计算机系统应用 16.1 (2007), pp. 59–62.

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<>());
        }
    }
    void add(int source, int dest, int weight) {
        adj.get(source).add(new Edge(dest,weight));
        adj.get(dest).add(new Edge(source,weight));
        //prune input as it comes in
    }
    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++) {
                int a = GenerateRandInt(mx);
                int b = GenerateRandInt(mx);
                int c = GenerateRandInt(mx);
                pw.println(a + "␣" + b + "␣" + c);
            }
            pw.close();

            long start = System.nanoTime(); // track start time
            Scanner sc = new Scanner(new File("test-data.txt"));
            int N = sc.nextInt(), 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();
            System.out.println("Trial:␣" + f);
            System.out.println(roundNum((end-start)/1e9));
            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.io.*;
import java.text.DecimalFormat;
import java.util.*;
import java.math.*;

public class finalSPFA {
    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");
            pw.println(mx +  "␣" + mx);

            for(int i = 0; i < mx; i++) {
                int a = GenerateRandInt(mx);
                int b = GenerateRandInt(mx);
                int c = GenerateRandInt(mx);
                pw.println(a + "␣" + b + "␣" + c);
            }
            pw.close();

            long start = System.nanoTime();
            Scanner sc = new Scanner(new File("test-data.txt"));
            int V = sc.nextInt(), E = sc.nextInt();
            ArrayList<ArrayList<Edge>> adj;
            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(sum);
        }
        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;
        }
    }
}
```