Computer Science Extended Essay

Correlation between time complexity and runtime of graph search algorithms

To what extent can the time complexity of graph search algorithms be used to predict its runtime while it solves the single-source shortest path problem?

Word count: 3972 words

Table of Contents

1 INTRODUCTION 1

2 TIME COMPLEXITY 2

2.1 Input size and running time 2

2.2 The Big-O notation 3

3 GRAPHS IN COMPUTER SCIENCE 5

3.1 Directed and undirected edges 5

3.2 Weight of edges 6

4 SINGLE-SOURCE SHORTEST PATH PROBLEMS 6

4.1 Attributes of the vertices 7

4.2 Initializing process 8

4.3 Relaxation 9

4.4 Explanation of the Bellman-Ford algorithm 10

4.5 Explanation of Dijkstra's algorithm 11

5 EXPERIMENTAL ANALYSIS 13

5.1 Hypothesis 13

5.2 Methodology 15

5.3 Variables 16

5.4 Results and analysis 17

6 CONCLUSION 24

6.1 Limitations of the investigation 24

BIBLIOGRAPHY 26

APPENDIX 28

1 Introduction

In computer science, graphs are a data structure that models points (vertex) and connections between two points (edge). Many real-life problems can be modelled using graphs, including GPS navigation, animal migration prediction, and transport network design (Gibbons 1985). These problems are solved using graph search algorithms that determine the shortest path from one vertex to all other vertices (Cormen et al. 2009).

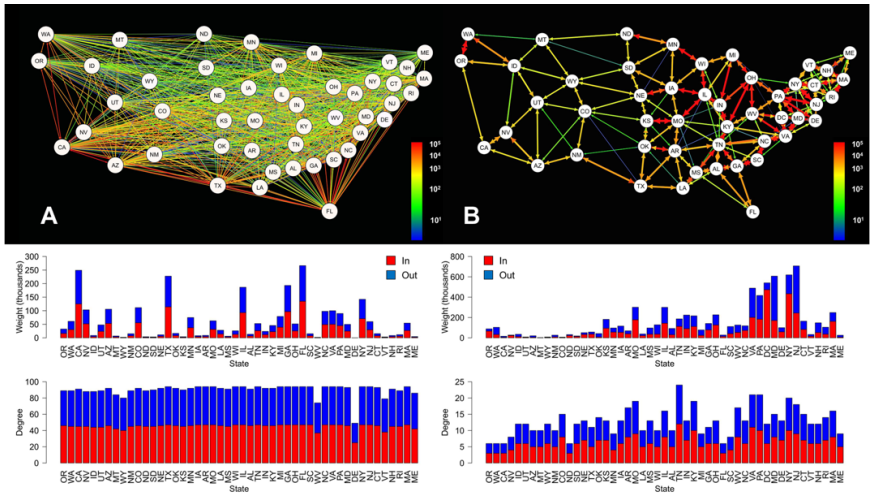


Figure 1. Graphs being used in U.S. traffic networking (Bozick, B. and Real, L. 2015)

As the size of data processed continues to increase with the advent of concepts like Big Data, the efficiency of algorithms is becoming ever more important. This is because efficient algorithms are needed to process large data in a reasonable runtime. Here, runtime refers to the time taken by an algorithm to execute (Christenson 2006).

One method of theoretically quantifying the runtime is time complexity. The time complexity of an algorithm is an approximate measure of the rate of increase of runtime represented as a function (Adamchik 2009). In theory, time complexity alone should be sufficient to determine the runtime of an algorithm. However, the time complexity in practice may not accurately predict its runtime due to various causes like operational downtime and disparities in the speed of different operations (Adamchik 2009). It is important to ask to

what extent can the runtime of an algorithm be estimated with its time complexity, especially for algorithms with large data size like graph search algorithms, because analysing time complexity of the two algorithms is much more convenient.

Therefore, the research question **'To what extent can the time complexity of graph search algorithms be used to predict its runtime while it solves the single-source shortest path problem?'** is formulated.

2 Time complexity

The time complexity of an algorithm is a theoretical measure of the amount of time required to execute an algorithm for a large input size (HackerEarth 2018). An algorithm's input size and running time is analysed to evaluate its time complexity.

2.1 Input size and running time

Input size of algorithms is a measure of how large the input is. For most algorithms, the number of elements can be used as a measure of the input size (Cormen et al. 2009). Although an input of most algorithms can be described with a single numerical value, input of graph search algorithms must be described with two values: number of vertices V and number of edges E (Cormen et al. 2009).

Running time of algorithms is defined as "the number of primitive operations executed." (Cormen et al. 2009, p.25) More simply, it is the total number of 'steps' taken by the algorithm to solve the problem. This is a different concept to runtime, which measures the time taken by algorithms. The definition of primitive operations can vary, but the pseudocode in this paper is written so that each line is equivalent to a primitive operation.

2.2 The Big-O notation

The Big-O notation (O-notation) is a way to represent the time complexity of an algorithm as a function that outputs its running time given its input size.

O-notation is first defined using mathematical functions. Formally, $f(n) = O(g(n))$ when there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ (Cormen et al. 2009, p.47).

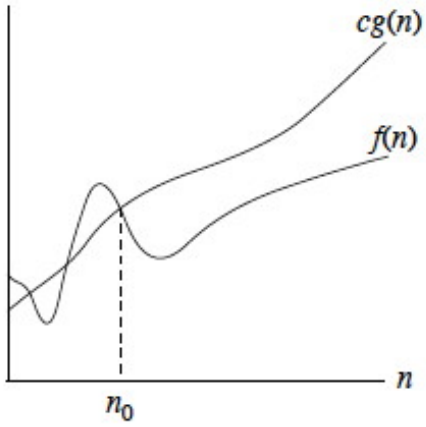


Figure 2.1. Relationship between $f(n)$ and $g(n)$ (Suthers 2014)

Intuitively, $f(n)$ is never larger than $cg(n)$ for large values of n, which implies that $f(n)$ has an equal or slower growth rate than $g(n)$. This means that $g(n)$ shows an approximate 'worst-case' growth rate for $f(n)$ (Adamchik 2009). Often $g(n)$ is defined as the function of the highest-order term of $f(n)$ as it has the most significant effect on the output value for large n. For example, when $f(n) = 3n^2 + 2n + 6$, $g(n) = n^2$.

O-notation can be applied to algorithm analysis by categorising the running time of algorithms into different types of functions. The running time of an algorithm is first rewritten as a function $f(n)$ (where n is the input size) using its pseudocode and analysing how many times an operation is executed. Then a function $g(n)$ is defined using only the highest-order term.

When different algorithms have the same $O(g(n))$, by the definition of O-notation, the algorithms have the same worst-case growth rate. This allows categorization of algorithms into different groups where each group has a similar running time.

Table 1. Hierarchy of $O(g(n))$ (HackerEarth 2018)

$O(g(n))$	$g(n)$
$O(\log n)$	$g(n) = 2\log n, 3 + 7\log n, \ldots$
$O(n)$	$g(n) = 6n + 7, 3n + \sqrt{n}, \ldots$
$O(n\log n)$	$g(n) = 2n\log n, 7n(2\log n + 7), \ldots$
$O(n^2)$	$g(n) = 3n^2 + 2n + 6, 7n^2 + \sqrt{n}, \ldots$
$O(n^3)$	$g(n) = 6n^3 + 2n^2 + 7n + 9, 12n^3 + 7, \ldots$

Table 1 shows the different groups of $O(g(n))$, where the group below has a marginally higher running time than the group above.

There are two properties of O-notation, derived from its definition, necessary for calculating the time complexity of algorithms (HackerEarth 2018).

- Property 1: $O\big(a(n)\big) + O\big(b(n)\big) = O(a(n) + b(n))$.
- Property 2: $O\big(a(n)\big) \times O\big(b(n)\big) = O(a(n) \times b(n))$.

For graph search algorithms, their time complexity is described by $O\big(g(V, E)\big)$, where $g(V, E)$ is a function of V and E. Examples of $g(V, E)$ include $V + E$ and VE^2. When comparing the time complexity of these algorithms, the relationship between the two input values must be known for fair comparison.

3 Graphs in Computer Science

Graphs are an abstract data structure that shows a set of points (vertices) interconnected by a set of lines (edges) (Gibbons 1985). A graph G has a set of vertices and a set of edges. Each vertex has a unique integer index value. An edge in a graph are identified by its source vertex u and destination vertex v and are denoted as (u, v).

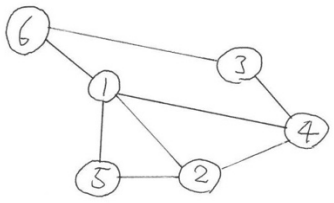


Figure 3.1. Example of a graph

Figure 3.1 shows a graph with vertices $\{1, 2, 3, 4, 5, 6\}$ and edges $\{(6, 1), (1, 5), (2, 5), (2, 1), (4, 1), (2, 4), (3, 4), (3, 6)\}$.

3.1 Directed and undirected edges

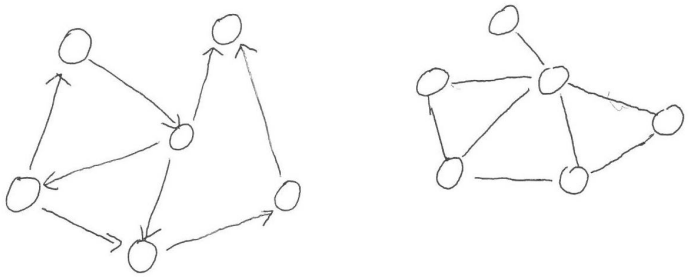


Figure 3.2. Directed graph (left) and undirected graph (right)

Edges in a graph can be either directed or undirected. Directed edges specify the direction of travel along the edge. Undirected edges, however, can be travelled in any direction. Directed graphs will be used because undirected edges can be emulated by using two directed edges travelling in an opposite direction, rendering directed edges more useful (Gibbons 1985).

3.2 Weight of edges

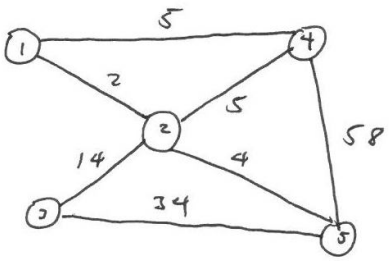


Figure 3.3. Weighted graph

Edges can also have a weight. Weight represents how much 'effort' is needed to travel across the edge, which can be applied to real-world concepts like distances. Formally, weights of edges are given by a weight function $\omega: E \to \mathbb{R}$ that maps each edge to a real value (Cormen et al. 2009).

4 Single-source shortest path problems

A path $p = \langle v_0, v_1, \dots, v_k \rangle$ is a connection between all vertices $v_0, v_1, \dots, v_k$. A weight of a path p is the sum of the weight of all edges making up the path, or

$$\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

$$= \omega(v_0, v_1) + \omega(v_1, v_2) + \cdots + \omega(v_{k-1}, v_k),$$

where $\omega(u, v)$ is the weight function of the edge (u, v) (Cormen et al. 2009). The shortest path between two vertices u and v is defined as the path connecting u and v with the smallest weight value.

Shortest path algorithms are graph search algorithms that solve single-source shortest path problems. Shortest path algorithms find the shortest path from a fixed source vertex to all other vertices in a graph (HackerEarth n.d.).

Shortest path algorithms rely on the property of shortest path that "shortest path between two vertices contain other shortest paths within it." (Cormen et al. 2009, p.644) This means that when the path $\langle s, \dots, v_{k-1}, v_k \rangle$ is the shortest path between s and v_k, the path $\langle s, \dots, v_{k-1} \rangle$ is also the shortest path between s and v_{k-1}, and so on. This is referred as "the optimal-substructure property of shortest path." (Cormen et al. 2009, p.644)

The shortest path may contain negative edges. When this is the case, the resulting path weight may be a finite negative value. However, shortest path cannot contain negative-weight cycles, because shortest-path weight can decrease indefinitely by continuously travelling along the negative-weight cycle (Cormen et al. 2009). It also cannot include positive-weight cycles, because the shortest path will not pass through a vertex more than once but a cycle ensures repeated visit to at least one vertex (Cormen et al. 2009).

There are two basic shortest-path algorithms: Bellman-Ford algorithm and Dijkstra's algorithm. Bellman-Ford algorithm is a more general algorithm that finds the single-source shortest path of graphs containing edges with any real-valued weights. This makes it useful for solving more variation of problems, but its algorithmic redundancy causes it to have a higher time complexity (HackerEarth n.d.). Dijkstra's algorithm has a lower time complexity by being less redundant, but it is less useful as it only works for graphs containing edges with nonnegative real-valued weights (HackerEarth n.d.).

4.1 Attributes of the vertices

All vertices have attributes $v.\pi$ and $v.d$, where $v.\pi$ is the vertex connected with v that will lead to the source s in the shortest path and $v.d$ is the shortest path weight between s and v (Cormen et al. 2009).

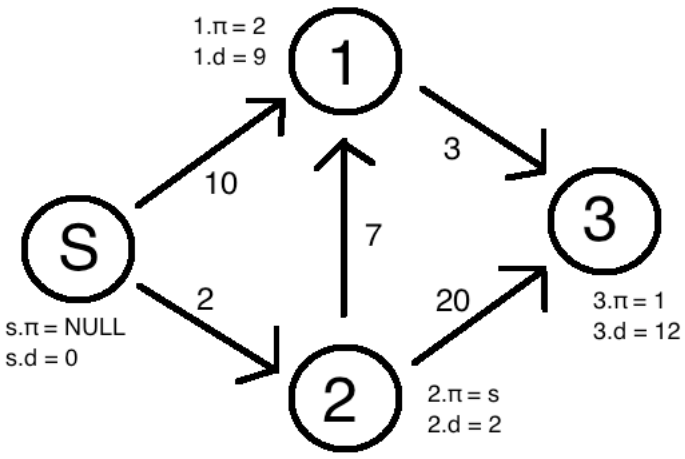


Figure 4.1. A graph and its attributes

Figure 4.1 shows a graph and the attributes of vertices. Shortest path from s to 3 is $\langle s, 2, 1, 3\rangle$. $3.\pi = 1$ because 1 precedes 3 in the shortest path. $3.d = 12$ because the shortest path length between s and 3 is 12. This applies to all other vertices.

s is an exception because it is the source vertex. $s.\pi = NULL$ because a source has no previous vertex in the path. $s.d = 0$ for the same reason.

4.2 Initializing process

Shortest path algorithms share a common initializing process, where the attributes for the vertices are initialized to a null (or equivalent) value (Cormen et al. 2009).

INITIALIZE-SINGLE-SOURCE(G, s)
For each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi =$ NULL
$s.d = 0$

Figure 4.2. Pseudocode of initializing process (Cormen et al. 2009)

Figure 4.2 shows the pseudocode. Because there is a loop that iterates through all vertices v of the graph G, the overall time complexity of the algorithm is $O(V)$. This knowledge is needed for calculating the overall time complexity of both algorithms.

4.3 Relaxation

Relaxation process is used repeatedly throughout the execution of shortest path algorithms. Relaxation process takes an edge (u, v) and tests whether going through this edge will yield the shortest path from source s to v and decrease $v.d$ (Cormen et al. 2009).

RELAXATION (u, v, ω)
if $v.d > u.d + \omega(u, v)$
$v.d = u.d + \omega(u, v)$
$v.\pi = u$

Figure 4.3. Pseudocode of relaxation process (Cormen et al. 2009)

Figure 4.3 shows the pseudocode. It takes an input of two vertices u and v and the weight function ω. The if statement checks if $v.d > u.d + \omega(u, v)$, and if true, update $v.d \rightarrow u.d + \omega(u, v)$ and update $v.\pi \rightarrow u$. Intuitively, it is checking whether going through the edge (u, v) will result in a smaller shortest path weight than the previous 'shortest' path from source to v.

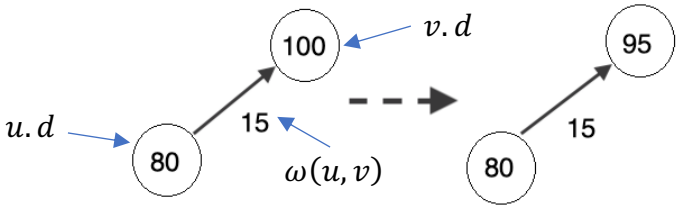


Figure 4.4. Process of relaxation

Let (u, v) be the edge shown in figure 4.4. Before relaxation, the shortest path from source s to v has a weight 100. However, when the shortest path goes through u, the shortest path weight would be $80 + 15 = 95$, which is smaller than 100. Therefore, the relaxation process will update $v.d \rightarrow 95$ and $v.\pi \rightarrow u$.

This paper considers each individual relaxation process on a single edge to take a constant $O(1)$ time.

4.4 Explanation of the Bellman-Ford algorithm

Bellman-Ford algorithm finds the shortest path of a graph by fundamentally relying on the notion that the shortest path cannot contain more than $V - 1$ edges (Cormen et al. 2009). This is because, if the shortest path contains more than $V - 1$ edges, a cycle must exist somewhere, but since the shortest path cannot contain any cycles, the path must not be the shortest path, leading to a contradiction.

```
BELLMAN-FORD (G, ω, s)
1       INTIALIZE-SINGLE-SOURCE(G,S)
2       for i = 1 to G.V − 1
3               for each edge (u, v) ∈ G.E
4                       RELAXATION(u, v, ω)
5       for each edge (u, v) ∈ G.E
6               if v.d > u.d + ω(u, v)
7                       return FALSE
8       return TRUE
```

Figure 4.5. Pseudocode of Bellman-Ford algorithm (Cormen et al. 2009)

Figure 4.5 shows the pseudocode of the Bellman-Ford algorithm. The algorithm first initializes the graph ($O(V)$). Then a loop iterates $V - 1$ times, and for each iteration it iterates through all edges and relaxes it ($(V - 1) \times O(E)$). It then checks whether a negative-weight cycle exists by iterating once more through each edge and check if relaxation could still occur ($O(E)$).

Overall, the time complexity of the algorithm is

$$O(V) + (V - 1) \times O(E) + O(E)$$

$$= O(V + (V - 1)E + E)$$

$$= O(V + VE)$$

$$= O(VE).$$

4.5 Explanation of Dijkstra's algorithm

Dijkstra's algorithm finds the shortest path of a graph by being fundamentally a greedy algorithm, meaning that it chooses the best solution for each step in hopes that it will lead to the best solution for the entire problem (Cormen et al. 2009).

In each step of relaxation, it chooses the vertex v with the minimum $v.d$ value and relaxes all edges leaving the vertex. This way, each edge is relaxed exactly once. By repeating this process until all vertices are chosen, the single-source shortest path between the source and all other vertices can be determined.

Although not all algorithms can find the solution using a greedy strategy, Dijkstra's algorithm can find the solution to the problem using a greedy approach, because the shortest path has the "optimal-substructure property of shortest paths" mentioned previously in chapter 4 (p.7).

In order to determine the vertex v with the lowest $v.d$ value, the algorithm utilizes the minimum-priority queue (min-priority queue) to store all vertices. A priority queue is a "data structure for maintaining a set S of elements, each with an associated value called a key" (Cormen et al. 2009, p.162). used to provide a hierarchy between elements. When the EXTRACT-MIN(S) operation is called on the min-priority queue, the element with the minimum key value will be called and removed from the queue (Cormen et al. 2009). Here, $v.d$ is used as the key value.

```
DIJKSTRA (G, ω, s)
1    INTIALIZE-SINGLE-SOURCE(G,s)
2    Q = G.V                                  // Initialize min-priority queue to
                                                 contain the set of all vertices in G
3    While Q ≠ ∅                              // While priority queue is not empty
4        u = EXTRACT-MIN(Q)
5        for each vertex v ∈ G.Adj[u]         // for all edges adjacent to u
6            RELAXATION(u, v, ω)
```

Figure 4.5 Pseudocode of Dijkstra's algorithm (Cormen et al. 2009)

Figure 4.5 shows the pseudocode of Dijkstra's algorithm. G is the graph, Q is the min-priority queue, $G.V$ is the set of all vertices in G, and $G.Adj[u]$ is the list of all vertices adjacent to the vertex u.

The algorithm initializes the graph ($O(V)$). It then initializes S and Q ($O(1)$). It then iterates through all vertices ($O(V)$) and each iteration extracts a vertex from Q and adds it to S ($V \times O$(EXTRACT-MIN)). It then relaxes all edges leaving the vertex once ($O(E)$).

The time complexity of EXTRACT-MIN operation depends on how the min-priority queue is implemented, but when an efficient implementation of the min-priority queue is used, such as the Fibonacci heap implementation, the time complexity of EXTRACT-MIN operation is $O(\lg v)$, where $\lg v$ is $\log_2 v$ (Cormen et al. 2009).

Overall, the time complexity of Dijkstra's algorithm is

$$O(V) + O(1) + V \times O(\lg V) + O(E)$$

$$= O(V + 1 + V \lg V + E)$$

$$= O(V(1 + \lg V) + E)$$

$$= O(V \lg V + E).$$

5 Experimental analysis

To answer the research question **'To what extent can the time complexity of graph search algorithms be used to predict its runtime while it solves the single-source shortest path problem?'** an experiment will be conducted that will compare the runtime of two graph search algorithms–Bellman-Ford algorithm and Dijkstra's algorithm–with vastly different time complexities. The experiment will observe whether the runtime of the two algorithms will grow in the pattern predicted by their time complexity.

5.1 Hypothesis

As shown above, the time complexity of the Bellman-Ford algorithm is $O(VE)$ and Dijkstra's algorithm is $O(V \lg V + E)$. To compare these time complexities, the relationship between V and E must be known.

The maximum number of edges possible in a graph with V vertices is $\frac{V(V-1)}{2}$. The edge density is defined as $\frac{number\ of\ edges\ in\ a\ graph}{maximum\ possible\ edges}$, and it is chosen to be 20%, because it is a compromise between having the minimum number of edges and ensuring that all vertices are connected. Therefore, the relationship between V and E can be known, where

$$20\% = \frac{E}{\frac{V(V-1)}{2}}$$

$$\Longrightarrow E = 0.2\left(\frac{V(V-1)}{2}\right) = 0.1(V^2 - V).$$

Knowing this relationship, the time complexity of the Bellman-Ford algorithm and Dijkstra's algorithm can be rewritten as a function of V.

Let $f(V) = VE$, which is the time complexity of the Bellman-Ford algorithm. Then

$$f(V) = VE$$

$$= V \times 0.1(V^2 - V)$$

$$= 0.1V^3 - 0.1V^2.$$

Using O-notation, $O(f(V)) = O(0.1V^3 - 0.1V^2) = O(V^3)$. Therefore, the time complexity of the Bellman-Ford algorithm in terms of V is $O(V^3)$.

Let $g(V) = V \lg V + E$, which is the time complexity of Dijkstra's algorithm. Then

$$g(V) = V \lg V + E$$

$$= V \lg V + 0.1(V^2 - V)$$

$$= 0.1V^2 + V \lg V - 0.1V.$$

Table 1 shows that V^2 has the highest order of growth. Hence, using O-notation, $O\big(g(V)\big) = O(0.1V^2 + V \lg V - 0.1V) = O(V^2)$. Therefore, the time complexity of Dijkstra's algorithm in terms of V is $O(V^2)$.

The hypothesis is that **'the time complexity of an algorithm can accurately predict the growth of its runtime.'** If the hypothesis is true, the runtime of Bellman-Ford algorithm must grow similar to the function V^3 and the runtime of Dijkstra's algorithm must grow similar to the function V^2.

5.2 Methodology

A Java program was developed that will measure the runtime of Bellman-Ford and Dijkstra's algorithm on a given graph.

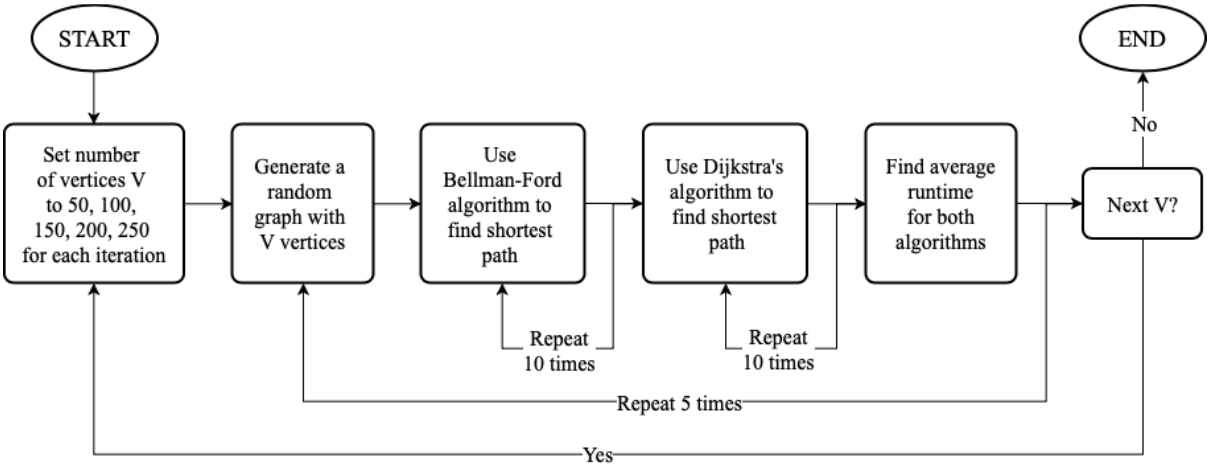


Figure 5.1. Flowchart of the methodology

Overall, the program iterates through the number of vertices 50, 100, 150, 200, and 250. For each iteration of vertices, the program generates a random graph with the density of edges set at 20%. The weight of the edges are also chosen at random.

The program uses the Bellman-Ford algorithm to find the shortest path between the source vertex and all other vertices on the graph. The runtime of the algorithm is measured in nanoseconds using Java's *System.nanoTime()* method (Rauh 2014). The runtime is then recorded onto a file. This is repeated ten times and the values are averaged.

It then uses Dijkstra's algorithm to measure the average runtime similar to the method outlined above for the Bellman-Ford algorithm.

After the average time for both algorithms are found, a new graph is generated and then the whole procedure is repeated five times with the same number of vertices. Then this entire procedure is repeated for different number of vertices outlined above.

5.3 Variables

Independent variable: Number of vertices in a graph

Dependent variable: Runtime of the algorithms

Several aspects were controlled in order to ensure a fair test.

Firstly, the experiment was conducted on the same computer system to ensure a fair comparison. Appendix A shows the system specifications.

Also, the algorithm is repeated extensively to minimize random error in the experiment, such as the fluctuations in the peak performance of CPU (Rauh 2014). For each number of vertices, 5 different randomly-generated graphs were created. For each graph, the algorithms were run 10 times and the results were averaged.

In addition, the weight of the edges are always non-negative, because graphs with negative edges cannot be evaluated using Dijkstra's algorithm, which inhibits a fair comparison between the two algorithms.

5.4 Results and analysis

Appendix B shows the raw data. Table 2 and 3 show the average runtime for the Bellman-Ford algorithm and Dijkstra's algorithm respectively.

Table 2. Average runtime of Bellman-Ford algorithm

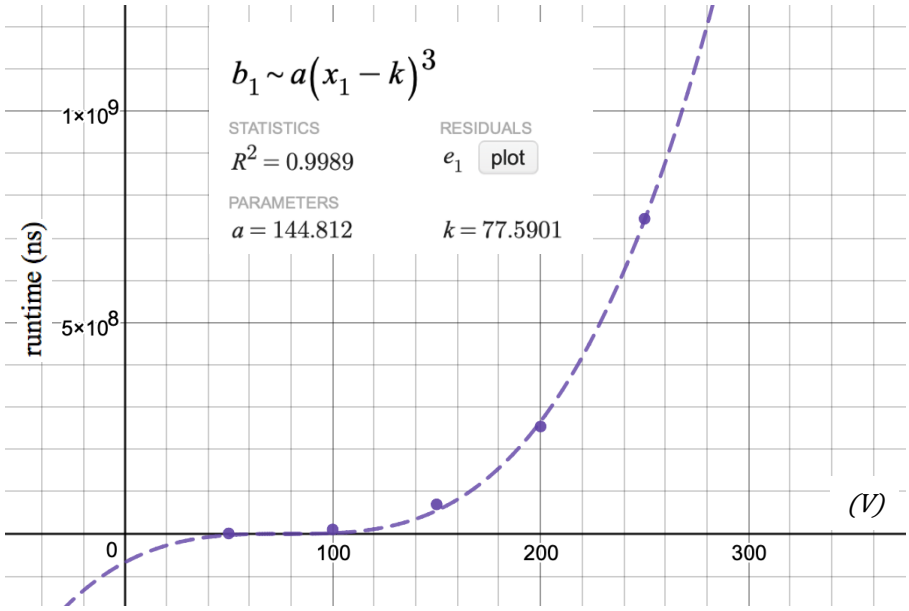
Number of Vertices	Average Runtime (ns)					
	Graph 0	**Graph 1**	**Graph 2**	**Graph 3**	**Graph 4**	**Average**
50	860686	1025488	956339	607678	661579	822354
100	10998592	10144607	9810249	10306823	9537810	10159616
150	71562149	68729223	70437377	69688570	67278419	69539147
200	257718339	250060013	253970656	247560845	259612930	253784557
250	746698968	751963900	766460746	738110532	723287520	745304333

Table 3. Average runtime of Dijkstra's algorithm

Number of Vertices	Average Runtime (ns)					
	Graph 0	**Graph 1**	**Graph 2**	**Graph 3**	**Graph 4**	**Average**
50	860686	1025488	956339	607678	661579	822354
100	10998592	10144607	9810249	10306823	9537810	10159616
150	71562149	68729223	70437377	69688570	67278419	69539147
200	257718339	250060013	253970656	247560845	259612930	253784557
250	746698968	751963900	766460746	738110532	723287520	745304333

Graph 1 plots the number of vertices against the average runtime of Bellman-Ford algorithm and Graph 2 plots the number of vertices against the average runtime of Dijkstra's algorithm. Using Desmos Graphing Calculator (Desmos 2018), a polynomial regression line, which has the same highest order term as the hypothesis, was plotted.

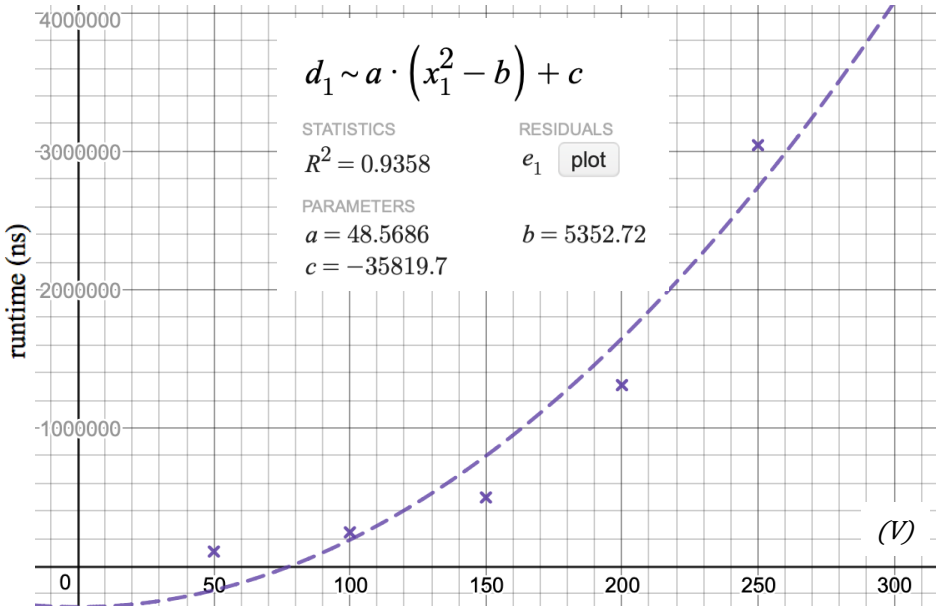
Graph 1. Bellman-Ford algorithm: number of vertices vs average runtime



Graph 1 suggests that the runtime of the Bellman-Ford algorithm matches very well with the regression line with an order of V^3 because the points deviate very little from the regression line. This suggests that the time complexity of the algorithm $O(V^3)$ accurately predicted the cubic pattern of runtime, which strongly supports the hypothesis.

The reason why the runtime fit well with V^3 pattern is because the algorithm's running time is predominantly occupied by the process that relaxes each edge $V - 1$ times (refer to line 2~4 of Figure 4.5) which has a time complexity of $O(VE)$, or $O(V^3)$ in terms of V. As all other processes like initialization have a trivial running time, $O(V^3)$ acts as a very good approximation of the real running time of the algorithm. Assuming that the running time (number of operations) is a good predictor of runtime (time in nanoseconds), $O(V^3)$ should also act as a good approximation for the runtime. This is shown in Graph 1.

Graph 2. Dijkstra's algorithm: number of vertices vs average runtime

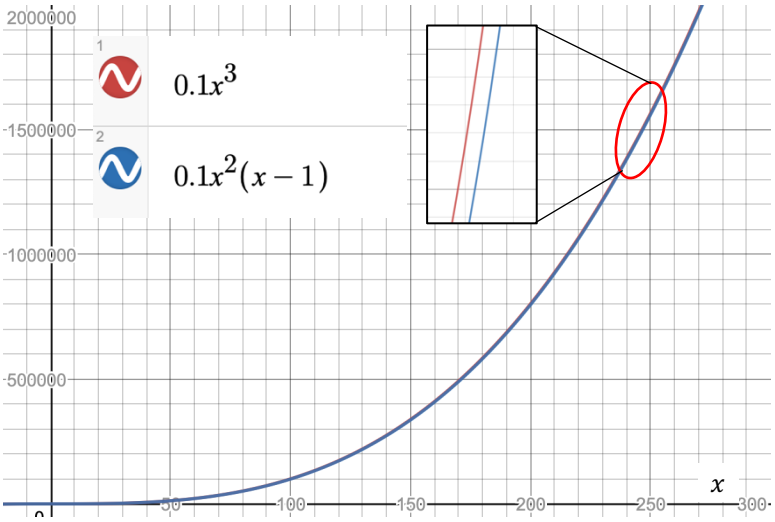


Graph 2, however, does not indicate that the regression line of order of V^2 matches well with the runtime of Dijkstra's algorithm because the points deviate irregularly from the regression line. This suggests that the result does not fully support the hypothesis, because the time complexity of the algorithm $O(V^2)$ predicted a quadratic pattern but the result did not show this.

By the same logic used to explain why the result for the Bellman-Ford algorithm supported the hypothesis, the time complexity of Dijkstra's algorithm should also have been a good predictor of the runtime. This means that the deviation did not come from the time complexity itself but from the process of rewriting the time complexity in terms of V in the hypothesis.

Writing the Bellman-Ford algorithm's time complexity in terms of V results in the expression $0.1V^3 - 0.1V^2$, which can be factorised as $0.1V^2(V - 1)$. The factorised expression clearly shows that the difference between $0.1V^3$ and $0.1V^2(V - 1)$ is trivial for sufficiently large values of V, because V and $V - 1$ is approximately equal for larger V.

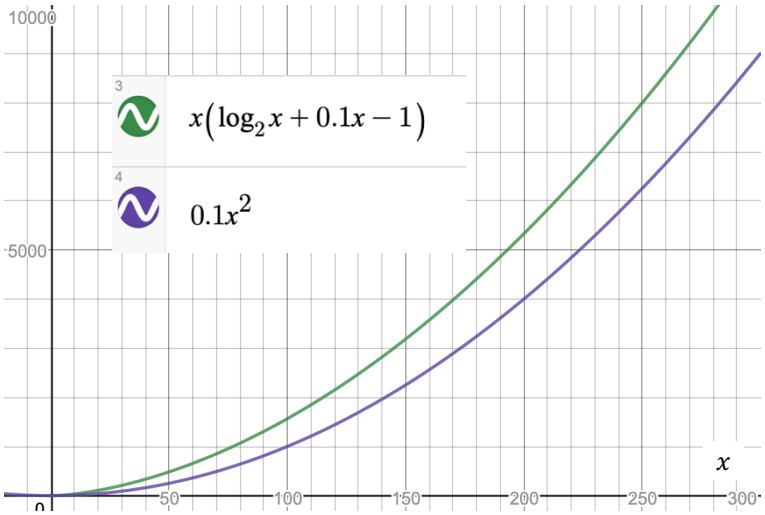
Graph 3. Time complexity approximation for the Bellman-Ford algorithm



Graph 3 clearly shows that the value of $0.1V^3$ and $0.1V^2(V-1)$ are virtually identical for all values of $V \geq 50$. This implies that $O(V^3)$ is a very good approximator for $O(0.1V^2(V-1))$, which is the original time complexity of the algorithm written in terms of V. This implies that, if the original time complexity of $O(VE)$ is a good approximator of the runtime, the regression line of order of V^3 should also be a good approximator of the runtime. This is why the result for the Bellman-Ford algorithm strongly supports the hypothesis.

For Dijkstra's algorithm, however, the process of rewriting the time complexity in terms of V introduces some deviation. When the time complexity of $O(V \lg V + E)$ is rewritten in terms of V, it results in the expression $0.1V^2 + V \lg V - 0.1V$, which is factorised as $0.1V(V + 10 \lg V - 1)$. Unlike the Bellman-Ford algorithm, the difference between $V + 10 \lg V - 1$ and V is not trivial, meaning that $O(V^2)$ may not be as good of an approximator for $O(V \lg V + E)$.

Graph 4. Time complexity approximation for Dijkstra's algorithm



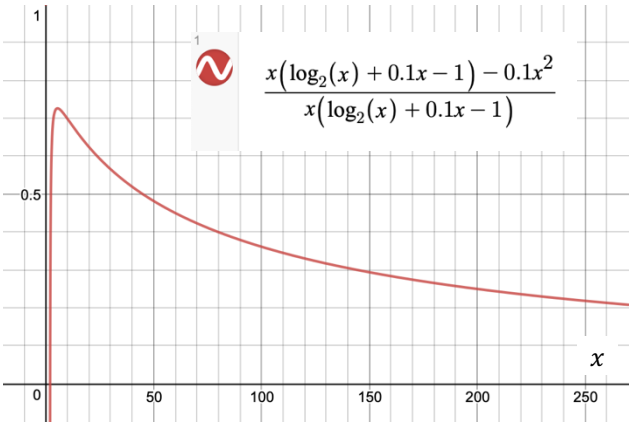
Graph 4 shows that the value of $0.1V(V + 10 \lg V - 1)$ and $0.1V^2$ are vastly different for all values of $V \geq 50$. This implies that $O(V^2)$ may not be a good approximator for the algorithm's original time complexity $O(0.1V(V + 10 \lg V - 1))$. Initially, it might seem that the difference is getting larger, but the magnitude of the difference itself is not as important to how much impact it has on the result. To quantify this, a simple ratio is set up like following:

$$error\ factor = \frac{difference\ between\ two\ functions}{value\ of\ the\ original\ function}$$

$$= \frac{V(0.1x + \lg V - 1) - 0.1V^2}{V(0.1x + \lg V - 1)}.$$

This ratio is then graphed to observe the behaviour of the error factor as V gets larger.

Graph 5. Error factor for Dijkstra's algorithm



Graph 5 shows that the error diminishes as V gets larger. This means that, for larger values of V, $O(V^2)$ acts as a better approximator of $O(V \lg V + E)$. This raises the possibility that the time complexity of $O(V^2)$ might still accurately predict the pattern of time complexity assuming that $O(V \lg V + E)$ is a good predictor.

This fact is significant, because the time complexity, or the O-notation, conveys the order of growth of running time for large input size, meaning that the time complexity more accurately predicts the pattern as the input size gets larger. If the regression line fits the points better for larger values of V, this means that the regression line can still be accurate even if it does not fit the points of lower V values.

Therefore, further investigation was conducted on the regression line for points $V \geq 200$. However, as the current data only has 2 points within the range, finding a quadratic regression line is redundant as it is equivalent to interpolation. Consequentially, further experiment was conducted for V values $\{210, 220, 230, 240\}$ to find the regression line.

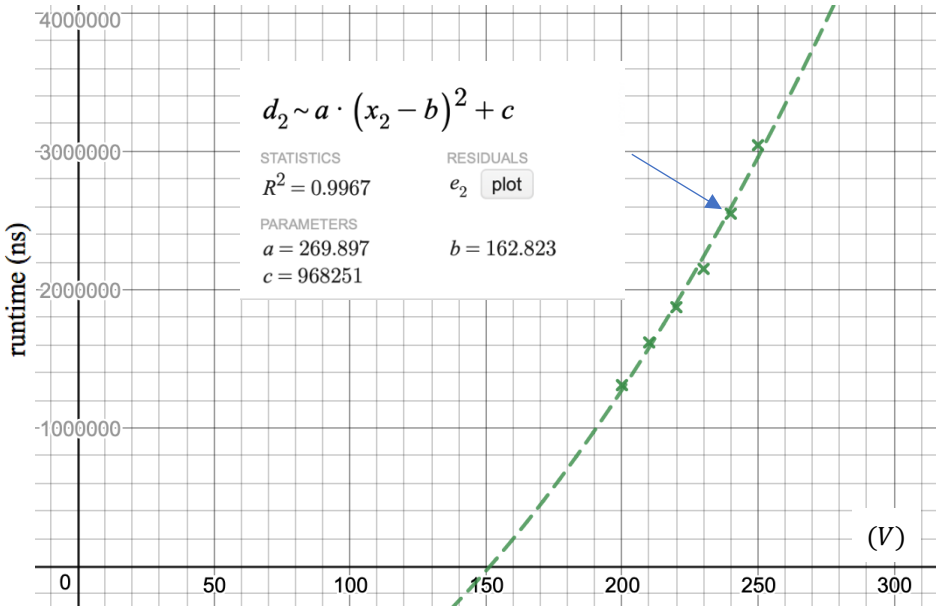
The raw data can be found in Appendix C. Table 4 shows the average runtime.

Table 4. Average runtime of Dijkstra's algorithm

Number of Vertices	Average Runtime (ns)					
	Graph 0	Graph 1	Graph 2	Graph 3	Graph 4	Average
210	1708465	1678975	1650557	1529511	1521485	1617799
220	1852724	1886971	1873182	1852015	1898029	1872584
230	2079467	2105660	2224823	2185162	2152780	2149578
240	2396487	2547511	2660240	2532655	2607668	2548912

Using Table 3 and 4, the average runtime of Dijkstra's algorithm for 200, 210, 220, 230, 240, and 250 are plotted on Graph 6.

Graph 6. Average runtime of Dijkstra's algorithm for $V \geq 200$



Graph 6 clearly shows that the regression line of V^2 clearly fits the points for $V \geq 200$ because the points deviate very little from the regression line. This implies that the time complexity of $O(V^2)$ is a good approximator of the runtime of Dijkstra's algorithm for larger values of V, meaning that the results for $V \geq 200$ supports the hypothesis.

6 Conclusion

The research question was **'To what extent can the time complexity of graph search algorithms be used to predict its runtime while it solves the single-source shortest path problem?'**

Overall, the extent to which the time complexity of graph search algorithms can be used to predict its runtime varied for different algorithms. For the Bellman-Ford algorithm, the time complexity could be used to accurately predict the pattern of its runtime for all input size. For Dijkstra's algorithm, the time complexity initially seemed insufficient to predict the pattern of its runtime, but further investigation revealed that time complexity could more accurately predict the pattern of its runtime for larger values of input size.

The conclusion suggests that, although the extent to which time complexity can be used to predict the graph search algorithm's runtime varies for different algorithms, it is always advisable to choose the algorithm with the lowest time complexity for achieving higher efficiency.

Further research could be conducted to evaluate whether this conclusion applies to other types of algorithms.

6.1 Limitations of the investigation

There were some significant limitations in the investigation that has to be addressed.

A significant limitation of this investigation is that only two algorithms were addressed in the experiment. This is because there are only two shortest path algorithms–Bellman-Ford and Dijkstra's algorithm–that can be explained within the scope of an Extended Essay. If other, more efficient algorithms with different time complexities could also be explained and used, it would either reinforce or disprove the conclusion of this

investigation. For example, an algorithm proposed by Thorup (Cormen et al. 2009), which runs in $O(E \lg(\lg(V)))$ time, could also be analysed.

Another limitation of the investigation is that Java's *System.nanoTime()* method is not the most accurate method to measure the time taken for a program to execute. This is because *System.nanoTime()* measures the time elapsed on the client side, which could cause the results to be affected though uncontrollable variations like the latency in *System.nanoTime()* method itself (Shipliëv 2014). This may result in the measurement to have an uncertainty of up to 15 milliseconds, which is significant considering its nanosecond scale of measurement (Rauh 2014). More sophisticated time measurement methods should be used to solve this problem.

Bibliography

Books

Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2009). *Introduction to Algorithms*. 3rd ed. Cambridge: The MIT Press

Gibbons, A. (1985). *Algorithmic Graph Theory*. Cambridge: Cambridge University Press

Images

Bozick, B. and Real, L.(2015). *Role of Human Transportation Networks*. [image] Available at: http://journals.plos.org/plospathogens/article?id=10.1371/journal.ppat.1004898 [Accessed 29 Jun. 2018]

Suthers, D. (2014). *ICS 311 Topic #3: Growth of Functions and Asymptotic Concepts*. [image] Available at: https://www2.hawaii.edu/~janst/311/Notes/Topic-03.html [Accessed 10 May. 2018]

Websites

Adamchik, V. (2009). *Algorithmic Complexity*. [online] Available at: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html [Accessed 10 May. 2018]

Christenson, P. (2006). *Runtime definition* [online] Available at: https://techterms.com/definition/runtime [Accessed 9 May. 2018]

Desmos Graphing Calculator. (2018). *Desmos Graphing Calculator*. [online] Available at: https://www.desmos.com/calculator [Accessed 31 Jul. 2018]

HackerEarth, (n.d.). *Shortest Path Algorithms*. [online] Available at: https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/ [Accessed 13 May. 2018]

HackerEarth, (2018). *Time and Space Complexity Tutorial & Notes*. [online] Available at: https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/ [Accessed 10 May. 2018]

Rauh, S. (2014). *NewsFlash: Can you rely on System.nanoTime()?*. [online] Available at: https://www.beyondjava.net/blog/newsflash-rely-system-nanotime/ [Accessed 14 May. 2018]

Shipliëv, A. (2014). *Nanotrusting the Nanotime*. [online] Available at: https://shipilev.net/blog/2014/nanotrusting-nanotime/ [Accessed 14 May. 2018]

Appendix

Appendix A: System Specifications

Computer Model: MacBook Pro (Retina, 15-inch, Mid 2015)

CPU: 2.5GHz quad-core Intel Core i7-4870HQ “Haswell”

GPU: Intel Iris 5200 Pro Built-in Graphics

RAM: 16GB 1600MHz DDR3

OS: macOS High Sierra Version 10.13.4 (17E202)

Appendix B: Raw data table

Number of vertices: 50

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	856192	173048	1018032	114052	960805	95038	634120	99998	652722	99015
2	872839	180270	1065744	97139	1005690	93209	591269	89250	657648	94097
3	893024	134861	1145571	127024	917728	93856	589316	85533	643532	91488
4	842303	127718	1006340	107083	1029345	92351	589582	83515	634488	88763
5	913999	116742	989263	95512	951107	91087	592386	84869	637868	92331
6	853422	187124	991297	100591	1021818	89289	588236	84641	633432	96013
7	844667	133023	987029	97641	939376	90556	587243	101776	636103	93952
8	840883	154132	984758	99365	932971	89849	684776	91187	738615	92813
9	846483	135891	992236	98973	907500	92451	633340	88986	742046	92781
10	843051	113591	1074608	113898	897045	90764	586513	89872	639338	94099
Average	860686	145640	1025488	105128	956339	91845	607678	89963	661579	93535

Number of vertices: 100

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	11089246	397341	10245974	368824	9969851	269187	10357659	244849	9570020	200755
2	10955945	365526	9983190	347171	9842029	154822	10242947	171400	9587340	181300
3	11037852	358547	10084541	345349	9924633	154703	10278877	181216	9486617	220309
4	10986895	358902	9853316	366073	9960261	150510	10317501	164565	9385302	325909
5	10957757	356730	10012878	346907	9669898	148589	10408927	163992	9514749	143180
6	11073386	356780	10406712	373419	9604941	150822	10158578	163380	9729316	138103
7	11155758	357463	10175361	365463	9761853	148764	10385477	170655	9565564	138620
8	11119907	359340	10160766	306024	9830371	151080	10517620	163660	9446475	254768
9	10975653	360724	10299534	227018	9879869	150160	10248353	165547	9537112	143379
10	10633524	361394	10223800	204924	9658787	204757	10152294	166267	9555607	137390
Average	10998592	363275	10144607	325117	9810249	168339	10306823	175553	9537810	188371

Number of vertices: 150

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	69046092	582387	66056444	473122	68761596	613634	66967085	549227	64503787	512537
2	69533233	488813	66037120	477333	67165292	516355	66624903	567873	68675045	572622
3	71022556	486212	66920519	472761	67280015	454816	66445523	518705	68667894	477905
4	71951046	489251	65890648	482812	69253335	486154	67418122	469663	68599981	445320
5	72494130	488557	70582448	507724	80783336	474093	70338935	505528	68378366	447483
6	72622161	488549	71680100	590317	72588081	474072	71917524	598970	66889711	449708
7	71661503	511951	70521959	681368	70266068	471647	71689160	456773	65674394	452524
8	70706885	487140	71233833	520096	69776616	452787	72352240	456626	66698177	449140
9	71794662	486561	70014802	528107	68955364	448487	72273923	456171	66326826	449056
10	74789222	483261	68354352	499935	69544063	469421	70858285	459291	68370008	442318
Average	71562149	499268	68729223	523358	70437377	486147	69688570	503883	67278419	469861

Number of vertices: 200

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	267167455	1378485	245940244	1545927	255952172	1234260	246265334	1279259	250942874	1351904
2	265071515	1297037	245704559	1294798	264371833	1225877	246519263	1224875	257053476	1374521
3	253771400	1294970	242650421	1294970	263652954	1337990	247579514	1226815	257289470	1321642
4	255605558	1375162	238749884	1381064	251473214	1224913	245949054	1225109	261001711	1323033
5	257942369	1295965	253207163	1443834	250285624	1317465	250188576	1226532	260107300	1325350
6	252802403	1374173	252336847	1282708	250576495	1247638	250775884	1224114	263333162	1439554
7	259535207	1315155	262582219	1278211	252057632	1410399	245526975	1301282	262766295	1390960
8	256158726	1309737	255827915	1280627	249718282	1226437	253190337	1222378	261451914	1363318
9	256204942	1301177	249736724	1279512	251407912	1226345	245400552	1220765	261783944	1299965
10	252923817	1425976	253864155	1482008	250210441	1222207	244212960	1219838	260399151	1284940
Average	257718339	1336784	250060013	1356366	253970656	1267353	247560845	1237097	259612930	1347519

Number of vertices: 250

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	764961702	2968152	769562636	3451788	766158428	3126263	760005131	2993098	735775979	2968940
2	765370384	2930295	752176206	3097515	749238819	3097243	732332935	2976283	718873059	2979645
3	757004286	2954292	752913841	3401047	782000186	3083271	740961771	2980346	725978966	2978265
4	741613496	2942598	752633787	2931828	782474474	3084166	733145265	2977485	724753470	2978476
5	729066478	2940596	750710851	3206213	784264214	3115394	733259328	2969698	718620835	2955747
6	762124850	2976234	746858198	2923291	762299412	3376765	742253071	2971990	725705886	3019678
7	745307933	2964210	749170339	2928402	769788736	3573807	737505092	2997618	722290309	2957925
8	734295418	2948726	762978805	3010280	767875999	3096027	732089322	3043725	724680005	2956287
9	738937379	3075862	756158767	2998609	750027706	3377876	732788712	2978879	719237126	2955932
10	728307751	2924837	726475573	2946897	750479482	3079829	736764693	2976007	716959565	3068266
Average	746698968	2962580	751963900	3089587	766460746	3201064	738110532	2986513	723287520	2981916

Appendix C: Raw data table for additional testing

Number of vertices: 210

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	324950366	1749637	321608305	1889575	326802974	1509187	300360134	1505309	312514269	1489059
2	328446243	1702515	330004853	1681946	331976083	1518858	302510373	1535339	310027468	1589758
3	324690006	1982960	326266302	1640533	325144712	1493423	307073044	1492507	316054415	1603857
4	326118467	2034613	328009242	1588999	328357989	1493185	310106956	1560741	313538097	1453970
5	316365050	1605518	335458989	1580901	328291655	1781728	305315362	1488471	320666430	1454014
6	325191328	1679640	328313333	1588842	313970525	1728205	312836872	1802679	309900845	1465443
7	324767737	1629556	325409741	1630860	330538957	1599638	310281874	1450384	311259279	1760495
8	317602229	1565051	317291752	1742156	319372851	1803031	306959979	1518310	304202907	1474174
9	316897219	1568921	325788457	1724697	329703924	1818610	305450472	1440315	302732184	1468977
10	324550317	1566239	329469287	1721242	325588474	1759701	311860247	1501059	296857865	1455105
Average	322957896	1708465	326762026	1678975	325974814	1650557	307275531	1529511	309775376	1521485

Number of vertices: 220

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	409628523	1879370	395491336	2105175	415831520	1878638	388525603	1835453	399028660	1906260
2	422485453	1825494	375755727	1948791	420846916	1861687	394347578	1924463	401142014	1879732
3	409152584	1871281	376117631	1781283	412435676	1876956	398466786	1837041	402772115	1887754
4	407362359	1997148	376145673	1782281	419936779	1860560	390077577	1821725	405234753	1879967
5	406579962	1816299	376909439	1971857	420355127	1931376	389049864	1852216	399751263	1880337
6	408240665	1815001	374239490	1872228	420345518	1860404	385712397	1924220	397509087	1875825
7	414826181	1876805	376227774	1789586	418840013	1879262	388123283	1854854	400733843	1974453
8	408300860	1816110	375949333	1812370	417503409	1862606	393361419	1826378	401546545	1889285
9	402332070	1815620	375810241	1797696	417380626	1860325	388102363	1824959	402554843	1928757
10	398099330	1814113	378921579	2008440	414705894	1860001	389802249	1818837	401445854	1877921
Average	408700799	1852724	378156822	1886971	417818148	1873182	390556912	1852015	401171898	1898029

Number of vertices: 230

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	501458158	2097416	476745716	2101459	522069081	2256872	509711018	2345674	481422062	2103105
2	496255265	2067862	469320751	2046966	519910279	2203081	513092951	2125250	483436064	2121018
3	497383346	2075220	484558396	2067151	514331737	2197928	506776602	2118689	482550552	2089705
4	493943117	2087295	485026641	2092614	524880685	2191618	509830267	2115820	484781463	2111749
5	486570004	2062284	485407930	2079910	509239575	2204713	511542937	2121437	483229815	2155966
6	477135445	2072075	480751839	2185624	516273553	2223262	509878943	2117606	501525376	2150868
7	472066544	2067816	475816085	2045510	515986573	2254882	491819496	2137031	517989491	2094481
8	475993782	2084010	479114227	2042917	515073695	2206338	510559242	2470307	503180539	2084310
9	475496656	2106865	472324580	2040973	493296618	2196988	492796815	2127968	499749718	2093618
10	472995880	2073829	458763698	2353472	497427691	2312552	483930816	2171836	505558133	2522975
Average	484929820	2079467	476782986	2105660	512848949	2224823	503993909	2185162	494342321	2152780

Number of vertices: 240

Attempt	Runtime (ns)									
	Graph 0		Graph 1		Graph 2		Graph 3		Graph 4	
	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra	B-F	Dijkstra
1	585888709	2397021	599314478	2562048	613311463	2627068	599642521	2508774	626607079	2630440
2	606342113	2431941	601330007	2517627	621352492	2532452	597434306	2599697	616438971	2569660
3	600798018	2378332	646495465	2490613	617230545	2827721	602726167	2502611	628022556	2577680
4	607431506	2383158	632005524	2530547	625103259	2612388	601700826	2523222	625238330	2563625
5	594747536	2381200	637972367	2484818	614266390	2589787	613255887	2571759	617218810	2546852
6	572334042	2387331	622787079	2486874	594742216	2859564	611510610	2529862	599906544	2599869
7	587086887	2390001	619432255	2491111	620882159	2587377	607738174	2496350	598747328	2605219
8	602598632	2394129	631800919	2766029	618230392	2578965	598216996	2608488	598658470	2869759
9	586763766	2385249	628945448	2585125	621668255	2825735	576022614	2491303	618485952	2620241
10	581281366	2436506	625739389	2560317	612259288	2561340	602452856	2494481	616677965	2493337
Average	592527258	2396487	624582293	2547511	615904646	2660240	601070096	2532655	614600201	2607668

Appendix D: Source code of program used

```
import java.io.*;
import java.nio.file.*;

public class MainClass
{
    private static final int NUM_GRAPH = 5;
    private static final int NUM_ATTEMPT = 10;
    private static int[] listVertexNumber = { 50, 100, 150, 200, 210, 220, 230, 240, 250};

    public static void main(String[] args) throws IOException
    {
        int numberOfVertex;

        for (int i: listVertexNumber) {
            numberOfVertex = i;
            Graph g = new Graph(numberOfVertex);
            GraphSearch gs = new GraphSearch();

            /* file output setup
             */
            Path path = Paths.get(System.getProperty("user.home")
                    + "/Downloads/output/" + Integer.toString(numberOfVertex) );

            File fileList = new File(path + "/vertex_list.txt");
            File fileTime = new File(path + "/timing.txt");
            PrintStream psList = null;
            PrintStream psTime = null;

            try {
                fileList.delete(); fileTime.delete(); // delete files
                Files.createDirectories(path); // initialize files
                fileList.createNewFile(); fileTime.createNewFile();

                psList = new PrintStream(
                        new FileOutputStream(fileList.getAbsolutePath(), true) );
                psTime = new PrintStream(
                        new FileOutputStream(fileTime.getAbsolutePath(), true) );
            }
            catch (IOException e) {
                e.printStackTrace();
            }

            /* process
             */
            for (int j=0; j<NUM_GRAPH; j++) {
                g.generateGraph(numberOfVertex);

                System.setOut(psList);
                System.out.printf("Graph %d\n", j);
                g.printAdjacencyList();
                System.setOut(psTime);
                System.out.printf("Graph %d\n", j);

                /* Bellman-Ford
                 */
                System.out.printf("Bellman-Ford\n");
                long bfStartTime = 0;
                long bfEndTime = 0;
                for (int k=0; k<NUM_ATTEMPT; k++) {
                    bfStartTime = System.nanoTime();
                    gs.bellmanFord(g);
                    bfEndTime = System.nanoTime();

                    System.out.printf("%d\n", (bfEndTime-bfStartTime) );
                }

                /* Dijkstra's
                 */
                System.out.printf("Dijkstra's\n");
                long dStartTime = 0;
                long dEndTime = 0;
                for (int k=0; k<NUM_ATTEMPT; k++) {
                    dStartTime = System.nanoTime();
                    gs.dijkstras(g);
                    dEndTime = System.nanoTime();

                    System.out.printf("%d\n", (dEndTime-dStartTime) );
                }

                /* output shortest path
                 */
                System.setOut(psList);
                g.printShortestPath();
                System.out.println();
            }

            psList.close();
            psTime.close();
        }
    }
}
```

```java
import java.util.*;

public class GraphSearch
{
    private void initializeSingleSource(Graph g)
    {
        for (int i=0; i<g.getNumberOfVertex(); i++) {
            g.getListVertex()[i].setD(Integer.MAX_VALUE);
            g.getListVertex()[i].setPi(-1);
        }
        g.getListVertex()[g.getIndexRoot() ].setD(0);
    }

    private void relaxation(Graph g, int indexU, int indexV)
    {
        Vertex u = g.getListVertex()[indexU];
        Vertex v = g.getListVertex()[indexV];

        if (u.getD() == Integer.MAX_VALUE) {
            return;
        }
        else if (v.getD() == Integer.MAX_VALUE) {
            v.setD(u.getD() + g.getEdgeWeight(u.getIndex(), v.getIndex()) );
            v.setPi(u.getIndex() );
        }
        else if (v.getD() > u.getD() +
                g.getEdgeWeight(u.getIndex(), v.getIndex()))
        {
            v.setD(u.getD() + g.getEdgeWeight(u.getIndex(), v.getIndex()) );
            v.setPi(u.getIndex() );
        }
    }

    public boolean bellmanFord(Graph g)
    {
        initializeSingleSource(g);

        /* Relaxation
         */
        int gNumVertex = g.getNumberOfVertex();

        for (int i=0; i<gNumVertex-1; i++) { // relax all edge V-1 times
            for (int j=0; j<gNumVertex; j++) { // for each source
                for (Edge e: g.getListEdge()[j]) { // for each destination
                    relaxation(g, e.getIndexSource(), e.getIndexDestination());
                }
            }
        }

        /* check for -ve weight cycle
         */
        Vertex src; Vertex dest;

        for (int i=0; i<gNumVertex; i++) { // for each source
            for (Edge e: g.getListEdge()[i]) { // for each destination
                src = g.getListVertex()[e.getIndexSource()];
                dest = g.getListVertex()[e.getIndexDestination()];
                if (dest.getD() > src.getD() +
                        g.getEdgeWeight(src.getIndex(), dest.getIndex()))
                {
                    return false; // -ve weight cycle found
                }
            }
        }
        return true; // no -ve weight cycle found
    }

    public void dijkstras(Graph g)
    {
        initializeSingleSource(g);

        ArrayList<Vertex> s = new ArrayList<Vertex>();
        PriorityQueue<Vertex> q = new PriorityQueue<Vertex>(g.getNumberOfVertex());

        for (int i=0; i<g.getNumberOfVertex(); i++) { // add all V to q
            q.add(g.getListVertex()[i] );
        }

        Vertex u = q.poll(); // pick lowest V
        while (u != null) { // while q not empty
            s.add(u);
            for (Edge e: g.getListEdge()[u.getIndex()] ) { // relax all adjacent edge
                relaxation(g, e.getIndexSource(), e.getIndexDestination());
            }
            u = q.poll(); // pick lowest V
        }
    }
}
```

```java
import java.util.*;

public class Graph
{
    private int numberOfVertex;
    private int indexRoot;
    private LinkedList<Edge>[] listEdge;
    private Vertex[] listVertex;

    public Graph(int numberOfVertex)
    {
        this.numberOfVertex = numberOfVertex;
        this.indexRoot = 0;
        this.resetGraph();
    }

    public int getNumberOfVertex()
    {
        return numberOfVertex;
    }

    public int getIndexRoot()
    {
        return indexRoot;
    }

    public LinkedList<Edge>[] getListEdge()
    {
        return listEdge;
    }

    public Vertex[] getListVertex()
    {
        return listVertex;
    }

    @SuppressWarnings("unchecked")
    public void resetGraph()
    {
        listEdge = new LinkedList[numberOfVertex];
        listVertex = new Vertex[numberOfVertex];

        for (int i = 0; i < numberOfVertex; i++) {
            listEdge[i] = new LinkedList<Edge>();
            listVertex[i] = new Vertex(i);
        }
    }

    public void addEdge(int src, int dest, int weight)
    {
        listEdge[src].addFirst(new Edge(src, dest, weight));
    }

    public int getEdgeWeight(int src, int dest)
    {
        int lengthEdgeList = listEdge[src].size();
        Edge tempEdge;

        for (int i = 0; i < lengthEdgeList; i++) { // for all edgeList
            tempEdge = listEdge[src].get(i);

            if (tempEdge.getIndexDestination() == dest) {
                return tempEdge.getWeight(); // if found
            }
        }
        return 0; // if not found
    }

    public void generateGraph(int numberOfVertex)
    {
        this.resetGraph();

        final double CHANCE = 20;
        final int MAX_WEIGHT = 100;
        int[] numberOfConnection =  new int[numberOfVertex];
        for (int i=0; i<numberOfVertex; i++) {
            numberOfConnection[i] = 0;
        }
        int weight; // randomly-generated weight [1,MAX_WEIGHT]

        /* Generate edges (i,j) if probability true
         */
        for (int i=0; i<numberOfVertex; i++) {
            for (int j=0; j<numberOfVertex; j++) {
                weight = (int) (Math.random() * (MAX_WEIGHT-1) + 1);

                /* create if probability true && NOT edge to itself
                 */
                if ((Math.random() * 100) < CHANCE && i != j) {
                    this.addEdge(i, j, weight);
                    numberOfConnection[i]++;
                }
```

```
            }
        }

        /* find the root vertex
         */
        for (int i=0; i<numberOfVertex; i++) {
            if (numberOfConnection[i] > numberOfConnection[indexRoot]) {
                indexRoot = i;
            }
        }
    }

    public void printAdjacencyList()
    {
        for (int i=0; i<numberOfVertex; i++) { // for each source
            System.out.printf("V %d:", i);
            for (Edge e: listEdge[i]) { // for each destination
                System.out.printf(" %d [%d] ->",  e.getIndexDestination(), e.getWeight());
            }
            System.out.printf(" \\end\n");
        }
    }

    public void printShortestPath()
    {
        System.out.printf("SHORTEST PATH source: %d\n", indexRoot);
        System.out.printf("_____________________________\n");

        String tempOutput;
        for (int i=0; i<numberOfVertex; i++) {
            System.out.printf("V%d", i);

            if (i == indexRoot) { // if source
                System.out.printf(" SOURCE\n");
                continue;
            }

            tempOutput = getPreviousPath(i, 0);
            if (tempOutput.endsWith("null")) { // if null case
                System.out.printf(" NOT CONNECTED\n");
                continue;
            }

            System.out.printf(": %s (Dist: %d)\n", tempOutput, listVertex[i].getD());
        }
    }

    private String getPreviousPath(int index, int count)
    {
        if (count > numberOfVertex || index == - 1) { // null: loop or no connection
            return "null";
        }
        else if (index == indexRoot) { // Base case
            return String.valueOf(index) + "";
        }
        else {
            return index + " <- " + getPreviousPath(listVertex[index].getPi(), count+1);
        }
    }
}
```

```java
public class Edge
{
    private int indexSource;
    private int indexDestination;
    private int weight;

    public Edge(int indexSource, int indexDestination, int weight)
    {
        this.indexSource = indexSource;
        this.indexDestination = indexDestination;
        this.weight = weight;
    }

    public int getIndexSource()
    {
        return indexSource;
    }

    public int getIndexDestination()
    {
        return indexDestination;
    }

    public int getWeight()
    {
        return weight;
    }

    public void setIndexSource(int indexSource)
    {
        this.indexSource = indexSource;
    }

    public void setIndexDestination(int indexDestination)
    {
        this.indexDestination = indexDestination;
    }

    public void setWeight(int weight)
    {
        this.weight = weight;
    }
}

public class Vertex implements Comparable<Vertex>
{
    private int index;
    private int pi; // default: -1
    private int d; // default: -1

    public Vertex(int index)
    {
        this.index = index;
    }

    public int getIndex()
    {
        return index;
    }

    public int getPi()
    {
        return pi;
    }

    public int getD()
    {
        return d;
    }

    public void setIndex(int index)
    {
        this.index = index;
    }

    public void setPi(int pi)
    {
        this.pi = pi;
    }

    public void setD(int d)
    {
        this.d = d;
    }

    @Override
    public int compareTo(Vertex otherVertex)
    {
        return -1 * Integer.compare(otherVertex.d, this.d);
    }
}
```