Fall 2017

ECE608 Computational Models and Methods

Assignment: Single-Source Shortest Path Problem

Name: *Yajie Geng* ***PUID:*** *0029970227*

Purdue University, West Lafayette
Department of Electrical and Computer Engineering

Abstract

Single-source shortest-path problem has been a classic topic in algorithm. There are several famous algorithms to solve the single-source shortest-path problem. However, those algorithms are of different performance under different input graphs. Therefore, an efficient algorithm to find which algorithm is more efficient for a given input graph is needed. In this work, a selection algorithm between *Dijkstra's Algorithm* and *Bellman-Ford Algorithm* is introduced. The selection algorithm basically follows two selection rules. The first one is whether the graph has negative-weight edges, and the second one is which algorithm takes less time. The second rule is hard to define, but it is found that the running time is based linearly on the proportion of the number of vertices and the number of edges. In this work, the process of finding the factor of the linear relationship is presented. Furthermore, the random graph generation is important in the experiments, in order to accurately find the linear factor and objectively test the algorithm's performance. Therefore, a random graph generator is also introduced. The experiment result shows that the selection algorithm following the two rules mentioned above is effective and saves time compared with random selection.

Keywords: Single-source shortest-path problem, Dijkstra's algorithm, Bellman-Ford algorithm, Random graph generation, Negative-weight edges, Running time, Linear relationship, Selection algorithm

Contents

1 Introduction **4**
1.1 Shortest-Path Problem 4
1.2 Single-Source Shortest-Path Problem 4
1.3 Strategy of Selecting the Algorithm 5

2 Algorithms of Single-Source Shortest Path Problem **5**
2.1 Dijkstra's Algorithm 5
2.2 Bellman-Ford Algorithm 6

3 Selection Between Dijkstra and Bellman Ford **7**
3.1 Selection Criteria 7
3.1.1 Negative-Weight Edges 7
3.1.2 The Running Time 7
3.2 The Selection Algorithm 8

4 Evaluation **8**
4.1 Random Directed Graph Generation 8
4.2 Implementation and Experiments 9
4.2.1 Graphs with Negative-Weight Edges 9
4.2.2 Graphs without Negative-Weight Edges 10

5 Conclusions **14**

References **15**

1 Introduction

1.1 Shortest-Path Problem

Suppose someone wants to find the shortest route to drive from Purdue to Chicago, and given a road map on which the distances between all adjacent intersections are marked, how can he determine the shorted route?[1] Enumerating all possible routes and choose the one with minimum distance is a possible way. However, there are usually huge number of possibilities, so the brute and force method takes la lot work and is not worth doing.[1]

Actually, the problem described above can be considered as a **shortest-paths problem** in graph theory, if we let the intersections represent the vertices, the road connecting adjacent intersections represent the edges, and the distance between the intersections represents the weights of edges. To find the shortest route between two intersections is just like to find the shortest path weight between two connected vertices in a graph. Accordingly, **shortest-paths problem** is defined as follows:

Given a weighted directed graph $G = (V, E)$, with weight function $w : E \to R$. The weight of path $p =< v_0, v-1, \cdots, v_k >$ is the sum of the weights of the edges $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. The shortest-path weight from u to v is defined as

$$\delta(u,v) = \begin{cases} min\{w(p) : u \overset{p}{\to} v\} & \text{if there is a path from u to v} \\ \infty & \text{onterwise.} \end{cases} \tag{1}$$

A shortest path from u to v is any path p with weight $w(p) = \delta(u, v)$.[1]

There are many variants of the shortest-path problem, such as **single-source shortest-path problem**, **Ssngle-destination shortest-paths problem**, **single-pair shortest-path problem**, and **All-pairs shortest-paths problem**.[1] In this work, I will focus on **single-source shortest path problem**, analyzing two single-source shortest-path algorithms and implementing a strategy to choose the fastest algorithm for various inputs of graphs.

1.2 Single-Source Shortest-Path Problem

The **single-source shortest-path problem** is defined as: given a graph $G = (V, E)$, find a shortest path from a given **source** vertex $s \in V$ to each vertex $v \in V$.[1]

In the route choice problem mentioned in 1.1, the shortest path is problem is well defined since all the distances are positive (all edges are of positive weights). In this case, the shortest path can be found using a greedy strategy. One of the most common greedy algorithm to solve the single-source shortest-path problem is *Dijkstra's Algorithm*, which will be discussed in detail later in this work.

Although, in most of the problems in real life, only graphs with all positive-weight edges are considered, graphs with negative-weight edges still worth considering in graph theory. Given a graph with negative-weight edges, if there are no negative-weight cycles, the shortest path problem is still well defined. If there are negative-weight cycles, for any path going through the cycle, the weight of the path can be smaller by going through the cycle once more, causing the weight of the shortest path to be $-\infty$.[1] In this case, the shortest-path problem is hard to define. The *Dijkstra's Algorithm* may cause wrong output if there are negative-weight edges. Therefore, another algorithm I will talk about in this work is *Bellmen-Ford Algorithm*, which can deal with graphs with negative-weight edges and detect negative-cycles.

1.3 Strategy of Selecting the Algorithm

In the previous section, I mentioned that *Dijkstra's Algorithm* cannot deal with graphs with negative-weight edges while *Bellmen-Ford Algorithm* can. Therefore, when determining while algorithm to use, the first thing we should considered is whether the graph has negative-weight edges. If it does, *Bellmen-Ford Algorithm* must be chosen.

Besides, if the graph has no negative-edges, we should decide which algorithm to use according to the efficiency, i.e., complexity of the algorithm. The complexity of *Dijkstra's Algorithm* is $O(|V|^2)$, and the complexity of *Bellmen-Ford Algorithm* is $O(|V||E|)$. Basically, when $|V| > |E|$, *Bellmen-Ford Algorithm* is better, and when $|V| < |E|$, *Dijkstra's Algorithm* is better.

In this work, I implement the choosing algorithm, and evaluate it by comparing the execution time of it with *Dijkstra's Algorithm* and *Bellmen-Ford Algorithm*, using randomly generated graphs.

2 Algorithms of Single-Source Shortest Path Problem

In this section, I will talk about *Dijkstra's Algorithm* and *Bellmen-Ford Algorithm.* In both of these algorithms, two variables are maintained for each vertex:

- $d[v]$, which is an upper bound on the weight of the shortest path from s to v at current stage.[1] As the algorithm terminates, the final value of $d[v]$ is the weight of the shortest path from s to v.
- $\Pi[v]$ is the vertex through which the path with weight $d[v]$ reaches v.[1]

The initial state of the two algorithm is the same. For all vertex $v \in V$, set $d[v] = \infty$ and $\Pi[v] = NULL$. Finally, initialize $d[s]$ as 0. The initialize function is implemented as follows:

Algorithm 1 Initialize(G, s)

```
1: for each $v \in V$ do
2:     $d[v] \leftarrow \infty$
3:     $\Pi[v] \leftarrow NULL$
4: end for
5: $d[s] \leftarrow 0$
```

A relaxation step to reduces $d[v]$ and update $\Pi[v]$ is defined below.[1]

Algorithm 2 Relax(u, v, w)

```
1: if $d[v] > d[u] + w(u, v)$ then
2:     $d[v] \leftarrow d[u] + w(u, v)$
3:     $\Pi[v] \leftarrow u$
4: end if
```

2.1 Dijkstra's Algorithm

Dijkstra's Algorithm is applied to the case where the input graph G has no negative-weight edges. The algorithm maintains a collection S of the vertices whose shortest paths have already been determined[1], as well as an array Q containing vertices whose shortest paths have not been determined. In each iteration, the algorithm chooses the vertex with minimum $d[v]$ in Q, puts it into S, and relax all the edges leaving it. The total complexity of the algorithm is $O(|V|^2)$.[1]

Algorithm 3 Dijkstra(G, w, s)

```
1: Initialize($G$, $s$)
2: $S \leftarrow \emptyset$
3: $Q \leftarrow V[G]$
4: while $Q \neq \emptyset$ do
5:     $u \leftarrow$ExtractMin($Q$)
6:     $S \leftarrow S \cup \{u\}$
7:     for each $v \in adj[u]$ do
8:         Relax($u, v, w$)
9:     end for
10: end whilereturn TRUE
```

The ExstractMin Method is shown below.

Algorithm 4 ExstraMin(Q)

```
1: $min \leftarrow Q(1)$
2: for $i \leftarrow 2$ to sizeof[$Q$] do
3:     if $d[Q(i)] < d[min]$ then
4:         $min \leftarrow Q(i)$
5:     end if
6: end for
7: return $min$
```

2.2 Bellman-Ford Algorithm

The *Bellmen-Ford Algorithm* works for the graph with negative-weight edges, and can detect negative-weight cycles. There are $|V| - 1$ iterations in total, and in each iteration, all edges are relaxed once. After finishing all the iterations, negative-weight cycles can be identified if $d[v] > d[u] + w(u, v)$ can still be found for some vertices. The total complexity of the algorithm is $O(|V||E|)$.[1]

Algorithm 5 BellmanFord(G, w, s)

```
1: Initialize($G$, $s$)
2: for $i \leftarrow 1$ to $|V[G]| - 1$ do
3:     for each $(u, v) \in E[G]$ do
4:         Relax($u, v, w$)
5:     end for
6: end for
7: for each $(u, v) \in E[G]$ do
8:     if $d[v] > d[u] + w(u, v)$ then return FALSE
9:     end if
10: end forreturn TRUE
```

3 Selection Between Dijkstra and Bellman Ford

3.1 Selection Criteria

3.1.1 Negative-Weight Edges

When determining which algorithms to apply to a given graph, the first thing needing considering is whether the graph has negative-weight edges. Since *Dijkstra's Algorithm* can make mistake when there are negative-weight edges, *Bellmen-Ford Algorithm* is the only choice in this case whatever the complexity is. Generally, the first criterion to follow when selecting the algorithm should be

- If there exist an edge $(u, v) \in E$ such that $w(u, v) < 0$, apply *Bellmen-Ford Algorithm.*

3.1.2 The Running Time

If all the edges are of non-negative weights, the running time should be taken into account during the selection. Given the number of vertice and number of edges, the one consuming less time should be selected. The complexity of *Dijkstra's Algorithm* is $O(|V|^2)$, and the complexity in the worst case of *Bellmen-Ford Algorithm* is $O(|V||E|)$. Intuitively, when the graph is dense ($|V| < |E|$), *Dijkstra's Algorithm* will run faster; and when the graph is sparse ($|V| > |E|$), *Bellmen-Ford Algorithm* will run faster.

However, the original version of Dijkstra's run time is $O(|V|^2 + |E|)$ assuming that the Extract-Min of Q takes $O(V)$ time and the Relaxation takes $O(1)$. $O(|V|^2)$ is only the simplified expression because the upper bound of $|E|$ is $O(|V|^2)$[1]. For a sparse graph, $|E|$ can be much smaller than $|V|^2$. Additionally, in the real implementation, the run time of ExtraMin can be optimized. Hence, the run time of *Dijkstra's Algorithm* can be rewritten as $O(|V|T_{ex}(|V|) + |E|T_{rlx})$, where T_{ex} represents the running time of ExtractMin in term of $|V|$, and the constant T_{rlx} represents the running time of Relax.

Accordingly, the run time of *Bellmen-Ford Algorithm* in terms of T_{rlx} is $O(|V||E|T_{rlx})$. Generally, when $|V|T_{ex} + |E|T_{rlx} \leq |V||E|T_{rlx}$, *Dijkstra's Algorithm* should be selected; otherwise, *Bellmen-Ford Algorithm* should be selected. By solving the equation $|V|T_{ex}(V) + |E|T_{rlx} < |V||E|T_{rlx}$, we can see more clearly the relationship of $|V|$ and $|E|$.

$$|V|T_{ex}(|V|) + |E|T_{rlx} < |V||E|T_{rlx} \tag{2}$$

$$\Rightarrow |E| > \frac{|V|T_{ex}(|V|)}{(|V|-1)T_{rlx}} \approx \frac{T_{ex}(|V|)}{T_{rlx}} \tag{3}$$

Since the ExstractMin (Algorithm 4) method is implemented by a linear search way, $T_{ex}(|V|)$ is simply a proportional function of V. Suppose $T_{ex}(|V|) = t|V|$, equation (3) above can be written as

$$|E| > \alpha|V|, \tag{4}$$

where $\alpha = \frac{t}{T_{rlx}}$ and is a constant.

Therefore, the second criterion to follow when selecting the algorithm can be concluded as follows:

- If $w(u, v) \geq 0$ for all edges $(u, v) \in E$ and $|V| \leq |E|/\alpha$, apply *Dijkstra's Algorithm.*
- If $w(u, v) \geq 0$ for all edges $(u, v) \in E$ and $|V| > |E|/\alpha$, apply *Bellmen-Ford Algorithm.*

For a sparse graph where $|V| > |E|$, the running time complexity of *Dijkstra's Algorithm* will be much fewer than $O(|V|^2)$, so only when $|V| >> |E|$ will the *Dijkstra's Algorithm* take much time than *Bellmen-Ford Algorithm.* Hence, we can conclude that $\alpha << 1$. Since the value of α varies in different implementations of the algorithm, it will be evaluated during the experiment.

3.2 The Selection Algorithm

Based on the criteria listed in the previous section, the algorithm to select between *Dijkstra's Algorithm* and *Bellmen-Ford Algorithm* should be:

Algorithm 6 selAlgorithm(G, w, s)

```
1: for each $(u, v) \in E[G]$ do
2:     if $w(u, v) < 0$ then return BellmanFord$(G, w, s)$
3:     end if
4: end for
5: if $V[G] \leq E[G]/\alpha$ then
6:     return Dijkstra$(G, w, s)$
7: else
8:     return BellmanFord$(G, w, s)$
9: end if
```

In the algorithm above, line 1-5 checks if there are any negative-weight edges. If not, line 6-10 determines which algorithm to apply in terms of the size of vertices and edges.

4 Evaluation

4.1 Random Directed Graph Generation

In order to evaluate the effectiveness of the selection algorithm, graphs with negative-weight edges and non-negative-weight edges, as well as graphs with various number of vertices and edges should be run on the algorithm. A random graph generation algorithm is helpful to objectively test the selection algorithm. Indicating the size of vertices, the algorithm firstly adds all the vertices to the empty graph. Then, the algorithm randomly choose two different vertices as source vertex and destination vertex for an edge. If the edge has not been added yet, add it to the graph and randomly set its weight. The weight of the edge can be set to a random integer or a random non-negative integer to satisfy different testing requirements. The algorithm will stop after the required number of edges are added.

In the random graph generation algorithm, no self-cycle edge will be generated because it's trivial in the shortest-path problem:

- If the self-cycle edge is of negative weight, the shortest-path algorithm will detect it as negative cycle and return false.
- If the self-cycle edge is of non-negative weight, adding it to the path will increase the weight, and the shortest-path algorithm will not include it in the shortest path.

Therefore, the self-cycle edges do not need considering in the shortest-path problem anyway.

Besides, the random graph generation algorithm will not generate multiple edges with same source vertex and destination vertex. In the shortest-path problem, the shortest-path will finally only consider the edge with smallest weight among the multiple edges and other edges will be redundant. Therefore, the upper bound of the esize should be vsize∗(vise−1) when there is one edge between every pair of source vertex and destination vertex.

Algorithm 7 RandomGraph(vsize, esize, negEdge)

```
if esize>vsize∗(vsize−1) then
    return FALSE                                      ▷ edge size out of range
else
    $G \leftarrow$ new $G(V, E)$
    $V[G] \leftarrow \emptyset$
    $E[G] \leftarrow \emptyset$
    for $i \leftarrow 1$ to vsize do                      ▷ add vertices to the graph
        Add vertex $V[i]$ to $V[G]$
    end for
    while $|E[G]| <$ esize do    ▷ randomly choose the source and destination vertices
        $s \leftarrow$ random vertex in $V[G]$
        $d \leftarrow$ random vertex in $V[G]\backslash\{s\}$
        if $(s, d) \notin E[G]$ then     ▷ add the edge to the graph if it doesn't exist yet
            if negEdge then
                $w(s, d) \leftarrow$ random integer           ▷ if edge weight can be negative
            else
                $w(s, d) \leftarrow$ random non-negative integer   ▷ if edge weight cannot be negative
            end if
            Add edge $(s, d)$ to $E[G]$
        end if
    end while
    return $G[V, E]$
end if
```

4.2 Implementation and Experiments

In the first part of the experiments, I examine the performance of the selection algorithm when the input graph has negative-weight edges; and in the second part, I run the *Dijkstra's Algorithm* and *Bellman-Ford Algorithm* with different input graphs without negative-weight edges, evaluate the value of α based on the result, and examine the effectiveness of the selection algorithm with the evaluated value of α. All of the four algorithms (*Dijkstra's Algorithm*, *Bellman-Ford Algorithm*, the selection algorithm, and the random graph generation algorithm) are developed in Java and run on a Macbook with 2.7GHz Intel Core i5 processor and 8GB 1867MHz DDR3 memeory.

4.2.1 Graphs with Negative-Weight Edges

In this part, I generate graphs with negative-weight edges using the random graph generator. The number of vertices varies from 0 to 1000, and the number of edges varies from 0 to the upper bound $|V|(|V|-1)$. Table 1 shows part of the result.

No. of Vertice	No. of Edges	Dijksra	Bellman-Ford	Selection
60	1180	false	false	false
	2360	false	false	false
	3540	false	true	true
70	1610	false	false	false
	3220	false	true	true
	4830	false	true	true
80	2106	false	true	true
	4212	false	true	true
	6318	false	true	true
90	2670	false	true	true
	5340	false	true	true
	8010	false	true	true
100	3300	false	true	true
	6600	false	true	true
	9900	false	true	true

Table 1: Graphs with negative-weight edges as input

As expected, in case of negative-weight edges, *Dijkstra's Algorithm* will not work, and the output of the selection algorithm will always be consistent with the output of *Bellman-Ford Algorithm.* The output false means that the input graph contains negative-weight edges.

4.2.2 Graphs without Negative-Weight Edges

In this part, the algorithms are run with random generated graphs without negative-weight edges of vertex sizes 100-1000 and edge sizes 0-vsize∗(vise−1), and the execution times are measured. As explained in section 3.1.2, the value of α is less than 1. In order to evaluate the value of α more accurately, a good strategy to determine the number of edges given the number of verteice should be applied. In the experiments, the number of vertices is chosen by the following equation:

$$|V| = n * 20, n \in \{5, 6, 7, \cdots, 50\}. \tag{5}$$

Now, given the number of vertices, the number of edges can be chosen by the following equations:

$$|E| = \begin{cases} i * 10 & i \in \{1, 2, \cdots, |V|/10\} \\ |V| + j * |V| * (|V| - 2)/10 & j \in \{1, 2, \cdots, 10\} \end{cases} \tag{6}$$

- **Evaluation of the α**

Based on the choice strategy for the number of vertices and edges, I generated random graphs satisfying the requirements. Then I run *Dijkstra's Algorithm* and *Bellman-Ford Algorithm* with these graphs as inputs. By comparing the running time of the two algorithms, the critical points where *Dijkstra's Algorithm* spends longer time can be found. Then the value of α should be the proportion of $|V|$ and $|E|$ at the critical point. I run the algorithms for 46 different vertice sizes based on equation (5). For each size of vertice and size of edges, the algorithms are run 5 times. The result data is of a large amount, so I just listed the one for $n = 5$ and $|V| = 100$ as an example in Table 2. The results for other vertice sizes are similar.

No. of Vertice	No. of Edges	$\lvert V\rvert/\lvert E\rvert$	T_{D}(ms)	T_{B}(ms)	$T_{\text{D}} > T_{\text{B}}$	$1/\alpha$
100	10	10.0	7.117821	3.274913	Y	3.33
	20	5.0	1.357316	0.669783	Y	
	30	3.3333333	0.605694	0.596838	Y	
	40	2.5	0.644598	0.774664	N	
	50	2.0	0.615382	0.808471	N	
	60	1.6666666	0.676469	1.072984	N	
	70	1.4285715	0.62389	1.946806	N	
	80	1.25	0.755426	2.147333	N	
	90	1.1111112	0.620377	1.902074	N	
	100	1.0	0.629825	1.831765	N	
	100980	100.0	2.765212	16.303625	N	
	1001960	50.0	10.249766	23.873327	N	
	1002940	33.333332	2.440229	11.943913	N	
	1003920	25.0	3.761583	38.987175	N	
	1004900	20.0	12.316635	27.139858	N	
	1005880	16.666666	6.745994	28.601171	N	
	1006860	14.285714	2.698196	19.871737	N	
	1007840	12.5	2.053272	19.958248	N	
	1008820	11.111111	24.8443	50.847065	N	
	1009800	10.0	2.818483	58.554	N	

Table 2: Estimated α when $|V| = 100$

After the experiments, I should have $46 * 5 = 230$ measured values of $1/\alpha$. The result is shown in Table 3.

By sorting these 230 values, the quartiles can be obtained:[2]

$$q1 = 2.88 \tag{7}$$
$$q2 = 3.13 \tag{8}$$
$$q3 = 3.48 \tag{9}$$

The interquartile range is:[2]

$$ipr = q3 - q1 = 0.60 \tag{10}$$

Then the data points lying outside the range $[q1 - \frac{3}{2}iqr, q3 + \frac{3}{2}iqr]$ are outliers and I do not consider them in the estimation[2]

$$q1 - \frac{3}{2}iqr = 1.98 \tag{11}$$
$$q3 + \frac{3}{2}iqr = 4.38 \tag{12}$$

Discarding the outliers, the average of the rest of the data points is calculated as 3.08.[2] Hence the estimated value of $1/\alpha = 3.08$, and the estimated value of $\alpha = 0.32$.

No. of Vertice	1	2	3	4	5	No. of Vertice	1	2	3	4	5
100	10.00	3.33	2.00	3.33	3.33	560	2.80	4.31	3.50	3.29	3.11
120	6.00	6.00	6.00	6.00	6.00	580	3.05	3.63	3.05	3.05	3.63
140	4.66	2.33	2.80	2.33	4.67	600	5.45	3.00	2.86	3.00	3.00
160	3.20	3.20	1.45	1.60	1.60	620	3.10	3.65	3.10	3.10	2.95
180	1.64	2.00	2.57	2.00	2.00	640	4.57	3.05	3.76	3.76	3.37
200	2.00	2.22	1.82	2.86	1.82	660	3.00	2.75	3.00	3.14	3.14
220	4.40	2.00	2.20	2.44	4.40	680	2.72	3.40	3.09	2.62	3.40
240	12.00	2.40	2.18	2.67	2.18	700	3.04	3.18	3.33	3.18	3.68
260	2.36	8.67	2.36	2.00	3.25	720	3.13	3.13	3.27	3.13	4.80
280	2.33	3.11	2.33	2.80	2.15	740	3.36	3.08	3.22	4.63	3.08
300	2.73	5.00	3.00	2.50	2.50	760	3.17	3.04	2.92	3.30	3.17
320	2.91	2.91	2.67	2.91	2.91	780	3.00	3.55	3.90	3.55	3.25
340	8.50	2.83	2.43	2.83	2.62	800	3.33	3.20	3.64	3.64	3.48
360	2.77	2.77	2.57	3.27	2.57	820	2.93	3.28	3.28	4.56	3.28
380	2.71	2.71	3.17	2.92	2.71	840	3.36	3.11	3.23	3.82	3.23
400	3.33	4.00	2.67	2.86	2.86	860	3.31	3.91	3.44	4.53	3.44
420	3.50	3.00	3.00	3.00	3.00	880	3.52	3.26	3.38	3.38	3.67
440	2.93	2.93	4.40	3.14	2.93	900	3.00	3.10	3.75	3.33	5.29
460	2.88	2.88	2.88	2.88	3.07	920	3.54	3.29	4.38	3.68	3.54
480	3.00	3.00	3.00	3.20	3.00	940	4.95	8.55	3.48	3.48	4.09
500	2.94	4.55	3.13	3.13	2.94	960	3.10	4.36	3.31	3.84	3.43
520	3.06	3.06	3.06	3.06	3.06	980	4.45	3.38	4.45	3.38	3.38
540	2.70	3.00	3.00	3.38	4.50	1000	3.70	3.45	3.45	3.23	3.45

Table 3: Measurement of α for different $|V|$

- **Performance of the Selection Algorithm**

Applying the α estimated above, I run the selection algorithm and compare its performance with *Dijkstra's Algorithm* and *Bellman-Ford Algorithm.* The result is shown in Table 4.

No. of Vertices	No.of Edges	T_D(ms)	T_B(ms)	$T_D < T_B$	Selection	T_S(ms)	Saved Time(ms)
100	10	6.22	1.74	FALSE	B	0.63	3.35
	21	1.15	0.57	FALSE	B	1.23	-0.37
	3321	8.59	63.23	TRUE	D	14.04	21.87
	6610	6.65	67.13	TRUE	D	8.54	28.35
200	21	4.91	0.26	FALSE	B	0.23	2.36
	42	1.53	0.67	FALSE	B	0.31	0.79
	13309	5.56	215.45	TRUE	D	4.39	106.11
	26554	2.37	139.69	TRUE	D	1.94	69.09
300	32	0.59	0.20	FALSE	B	0.20	0.20
	64	1.14	0.62	FALSE	B	0.61	0.26
	29964	4.07	335.04	TRUE	D	3.78	165.78
	59832	4.80	373.91	TRUE	D	4.67	184.69
400	42	0.96	0.30	FALSE	B	0.30	0.33
	85	1.08	0.74	FALSE	B	0.79	0.12
	53285	5.82	490.08	TRUE	D	10.81	237.14
	106442	10.34	822.42	TRUE	D	9.07	407.31
500	53	1.46	0.49	FALSE	B	0.49	0.48
	106	3.20	1.21	FALSE	B	1.60	0.60
	83273	17.38	2527.75	TRUE	D	14.61	1257.96
	166386	16.02	1655.78	TRUE	D	19.88	816.02
600	64	2.24	0.81	FALSE	B	0.79	0.73
	128	2.47	1.82	FALSE	B	1.91	0.24
	119928	24.91	4853.52	TRUE	D	23.32	2415.90
	239664	25.59	3082.38	TRUE	D	25.16	1528.83
700	74	3.01	1.01	FALSE	B	1.09	0.92
	149	2.98	3.05	TRUE	B	2.29	0.73
	163249	25.06	3524.00	TRUE	D	38.86	1735.67
	326274	40.10	5031.83	TRUE	D	37.03	2498.94
800	85	3.66	1.40	FALSE	B	1.44	1.09
	170	3.98	2.88	FALSE	B	2.83	0.60
	213237	48.60	12679.36	TRUE	D	52.31	6311.67
	426218	52.06	7133.47	TRUE	D	53.96	3538.81
900	96	5.04	1.84	FALSE	B	1.86	1.58
	192	7.09	4.20	FALSE	B	4.64	1.00
	269892	65.01	18244.35	TRUE	D	66.69	9087.99
	539496	78.82	10618.11	TRUE	D	81.11	5267.35
1000	106	5.84	2.42	FALSE	B	2.48	1.65
	213	6.05	4.69	FALSE	B	4.58	0.79
	333213	88.02	26422.00	TRUE	D	89.03	13165.98
	666106	96.67	15676.01	TRUE	D	98.64	7787.70

Table 4: Performance of the selection algorithm

The saved time is calculated by $(T_D + T_B)/2 - T_S$, where $(T_D + T_B)/2$ represents the average running time if selecting the algorithm randomly from *Dijkstra's Algorithm* and *Bellman-Ford Algorithm*. We can see that in almost all the cases, the selection algorithm will make the right decision, and apply the algorithm that spends less time. Among the 40 test cases in total, there is only one mistake in the case where $|V| = 100$ and $|E| = 1$. The running time of the selection algorithm is generally a little longer than the shorter one of *Dijkstra's Algorithm* and *Bellman-Ford Algorithm*, because the selection process takes some time. However, the selection algorithm still saves time compared with random selection. In particular, when the number of edges is far more larger than the number of vertices, the selection algorithm will save more than 10 seconds.

5 Conclusions

In this report, I analyze two algorithms for the single-source shortest-path problem. I analyze the application constrains and running time complexity of *Dijkstra's Algorithm* and *Bellman-Ford Algorithm*, come up with the selecting strategy between these two algorithms based on the analysis, implement the selection algorithm, and test its performance by experiments. When the input graph has negative-weight edges, only *Bellman-Ford Algorithm* can be applied. The more interesting case is that the input graph has no negative-weight edges. In this case, running time analysis must be applied in the selection process. Intuitively, when the graph is sparse, *Bellman-Ford Algorithm* is more efficient; and when the graph becomes relatively dense, *Dijkstra's Algorithm* is more efficient. The key idea is to find the critical point where the graph becomes dense and *Dijkstra's Algorithm* better. Based on the complexity analysis, I found that the $|V|$ and $|E|$ satisfies the relationship $\alpha|V| = |E|$ at the critical point, then I determined the value of α by experiments. After determining α, I evaluated the selection algorithm using different test cases. All of the experiments are based on the randomly generated graphs. During the implementation of the random graph generator, I add the constrains (the upper bound of edges, no self-cycles, and etc.) so as to only generate graphs that are worth considering in the shortest-path problem. The result shows that the selection algorithm saves time compared with selecting randomly. Furthermore, as $|V|$ becomes larger, and $|E|$ becomes far more larger that $|V|$, the selection algorithm can save more than 10 seconds for processing one input graph.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, third ed. The MIT Press, Cambridge Massachusetts and London England, 2009.

[2] Georgii, H.-O. *Introduction to Probability and Statistics.* Walter de Gruyter, Berlin and New York, 2007.