A faster algorithm for the single source shortest path problem with few distinct positive lengths

Matthew D. Williamson
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
https://researchrepository.wvu.edu/etd/4551
A Faster Algorithm for the Single Source Shortest Path Problem with Few Distinct Positive Lengths

Matthew D. Williamson

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

K. Subramani, Ph.D., Chair
K. Madduri, Ph.D.
A. Ross, Ph.D.

Lane Department of Computer Science
and Electrical Engineering

Morgantown, West Virginia
2009

Keywords: algorithm, Dijkstra, Single Source Shortest Path Problem, graph
Abstract

A Faster Algorithm for the Single Source Shortest Path Problem with Few Distinct Positive Lengths

Matthew D. Williamson

This thesis proposes an algorithm that involves Dijkstra’s algorithm to solve the Single Source Shortest Path Problem in a graph where the length of each edge is positive, and the specific case where there is a distinct number of possible lengths. Given a graph where there are $n$ vertices, $m$ edges, and $K$ distinct edge lengths, the proposed algorithm runs in $O(m)$ time if $nK \leq 2m$ and $O(m \log \frac{nK}{m})$ time if $nK > 2m$. The algorithm is tested and compared with some of the fastest algorithms for the Single Source Shortest Path Problem for graphs with arbitrary lengths. The results verify the theoretical conclusions that the proposed algorithm is better, in the case where $K$ is small, than the compared algorithms that did not take advantage of having few distinct edge lengths.
I would like to thank my advisor, Dr. K. Subramani, for taking me in and introducing me to interesting problems such as the one presented in this thesis. Without his guidance and support, I would not be where I am today. I would also like to thank Professor Jim Orlin from MIT for his contributions towards the development of the algorithm discussed in this thesis. Furthermore, I would like to thank Dr. Kamesh Madduri for being on the committee and helping out with some of the code used in this research. Finally, I would also like to thank Dr. Arun Ross for being on the committee and offering assistance when needed and help with preparing for the thesis defense.

I want to also extend my gratitude towards my parents, Daniel and Pamela Williamson, and my brother, Aaron Williamson, for their support and encouragement. Finally, I would like to thank my friends Lonnie Bowe and Trevor Kemp for being there from the beginning of our graduate journey. Graduate school would have not been the same without them.
Contents

1 Introduction 1
   1.1 Contribution ......................................................... 2
   1.2 Organization ....................................................... 2

2 Statement of Problem 4

3 Motivation 5

4 Related Work 6
   4.1 New Design Paradigms ............................................. 7
   4.2 Data Structuring Improvements .................................... 7
   4.3 Parameterization ................................................... 8
   4.4 Input Restriction .................................................. 8
   4.5 Parallelization .................................................... 8
   4.6 Distinct Edge Lengths ............................................. 9

5 An $O(m + nK)$ implementation of Dijkstra’s Algorithm 10
   5.1 Example of the $O(m + nK)$ Algorithm .......................... 14

6 A Faster Algorithm if $K$ is Permitted to Grow with Problem Size 17

7 Empirical Results for the $K$-Color Algorithm 20
   7.1 Experimental Setup ................................................ 20
   7.2 Results and Analysis ............................................. 22

8 Extension of the Empirical Results 28
   8.1 Experimental Setup ................................................ 28
   8.2 Results and Analysis when $K = 2$ ............................... 29
   8.3 Results and Analysis when $K = 3$ ............................... 31
List of Figures

5.1 The execution of our $O(m + nK)$ algorithm. .......................... 15

7.1 Performance of our shortest implementation and the reference solver for three graph families, as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest. ............... 23

7.2 Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of $K$ is varied. ......................... 24

7.3 Normalized SSSP Performance for a small-world graph (4 million vertices, 16 million edges) as the value of $K$ is varied. ......................... 25

7.4 Normalized SSSP Performance for a long mesh (4 million vertices, 16 million edges) as the value of $K$ is varied. ................................. 26

7.5 Normalized SSSP Performance for a square mesh (4 million vertices, 16 million edges) as the value of $K$ is varied. ................................. 27

7.6 Normalized SSSP Performance for a dense random graph (100K vertices, 10 million edges) as the value of $K$ is varied. ................................. 27

8.1 Performance of our implementation and the reference solver for two graph families when $N = 8$ and $K = 2$ as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest. . . 30

8.2 Normalized SSSP Performance for a square mesh (4 million vertices, 16 million edges) as the value of $N$ is varied and $K = 2$. ......................... 31

8.3 Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of $N$ is varied and $K = 2$. ......................... 32

8.4 Performance of our implementation and the reference solver for two graph families when $N = 8$ and $K = 3$ as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest. . . 34

8.5 Normalized SSSP Performance for a square mesh (4 million vertices, 16 million edges) as the value of $N$ is varied and $K = 3$. ......................... 35
8.6 Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of N is varied and $K = 3$. 36
List of Tables

7.1 Breadth-First Search execution time (in milliseconds) for various graph families on the test sequential platform. ........................................... 23
7.2 Breadth-First Search execution time (in milliseconds) for various graph families (the value of K is varied in experiments) on the test sequential platform. ........................................................................... 26
8.1 Breadth-First Search execution time (in milliseconds) for various graph families on the test sequential platform when N = 8 and K = 2. ............ 29
8.2 Breadth-First Search execution time (in milliseconds) for various graph families on the test sequential platform when N = 8 and K = 3. ............ 33
Chapter 1

Introduction

In this thesis, we present an algorithm that solves the Single Source Shortest Path Problem for a graph where the length of each edge is positive. This particular problem is shown to be a fundamental research area for both theoretical and practical computer science applications. [AMOT90] and [CLRS01] provide efficient algorithms and several applications of the Single Source Shortest Path Problem. In this case, we examine the scenario where the number of distinct edge lengths is small. The motivation for this specific problem originates from a social networks problem as described in Chapter 3.

Suppose we have a graph with $n$ vertices, $m$ edges, and $K$ distinct edge lengths. We propose two different algorithms. The first algorithm is a variation of Dijkstra’s algorithm involving additional lists, one for each distinct edge length, that runs in $O(m+nK)$ time. The second algorithm is a modification of the first one by incorporating binary heaps to improve the FINDMIN() operation. The running time for this algorithm is $O(m \log \frac{nK}{m})$, where $nK > 2m$.

When compared to other known algorithms, we find that the proposed algorithm is an improvement. When the parameters $n$, $m$, and $K$ are varied, we see that the running time of the proposed algorithm is better than the running time of Fredman and Tarjan’s Fibonacci heap implementation [FT87], which is $O(m + n \log n)$. It is also better than
the Atomic Heap implementation by Fredman and Willard [FW94], which has a running
time of $O(m + \frac{n \log n}{\log \log n})$. This is because the proposed algorithm actually runs in $O(m)$
time whenever $nK = O(m)$. Furthermore, even if every edge of the graph has a distinct
edge length, the running time of the algorithm is $O(m \log m)$, which is the same as the
binary heap method of Dijkstra’s algorithm.

1.1 Contribution

The main focus of this thesis contains two contributions. The first contribution is to
propose a new algorithm for solving the Single Source Shortest Path Problem where we
can use the number of distinct edge lengths as an additional parameter. We find that
the running time for this algorithm has a lower bound compared to other algorithms
where we examine the case where there is a small number of distinct edge lengths.

The second contribution is to provide an empirical analysis of this algorithm to
verify that it is an improvement when the number of distinct edge lengths is small. We
show that our algorithm runs faster than other algorithms by running experiments using
different types of graphs with varying edges and vertices. Unlike other algorithms, we
observe that not only is this algorithm an improvement from a theoretical approach,
but it also is an improvement from a practical approach.

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 defines the graph problem that
the algorithm uses to solve the problem considered. Chapter 3 provides the motivation
behind the research. The related work in this area is mentioned in Chapter 4. In
Chapter 5, we explain the new implementation of Dijkstra’s algorithm and show how the running time is $O(m+nK)$. Chapter 6 gives a second algorithm where we prove that the running time is $O(m \log \frac{nK}{m})$. We provide an empirical analysis to observe that the first algorithm is an improvement in Chapter 7 and the same for the second algorithm in Chapter 8. In Chapter 9, we explore some open problems related to our original problem. We conclude our research in Chapter 10.
Chapter 2

Statement of Problem

We shall consider a graph $G = (V, E)$, where $V$ is the vertex set with $n$ vertices, and $E$ is the edge set with $m$ edges. For each edge $(i, j) \in E$, we let $c_{ij}$ be the length of that edge. We assume that $c_{ij} \geq 0$ and is a real number.

For each vertex $v \in V$, we denote $\text{Adj}(v)$ as the set of outgoing edges from $v$. We also let $L = \{l_1, ..., l_K\}$ be the set of distinct edge lengths, sorted in increasing order. We assume that $\text{Adj}(v)$ is sorted. Otherwise, we can sort the edges in $\text{Adj}(v)$ in $O(m + K \log K)$ time using perfect hashing. Note that the running time of this sorting method is bounded by our algorithm.

We let the vertex $s \in V$ be the source of the graph. We also let $\delta(v)$ be the shortest path from $s$ to $v$ in the graph. If a path from $s$ to $v$ does not exist, then $\delta(v) = \infty$. The goal of the Single Source Shortest Path Problem is to find the shortest paths from $s$ to all of the other reachable vertices in the graph.

In order to make the pseudocode of the algorithm easier to comprehend, we let $E_j(v)$ be the set of outgoing edges from $v$ with the length $l_j$ for every vertex $v$ and edge length $l_j$. We should note that these edge lists are not actually stored in the programming code.
Chapter 3

Motivation

The motivation behind our research revolves around the social network "gossip" problem. A social network is simply composed of multiple groups of participants. Given a specific group, we say that the distance between each of the participants in this group is the value 1, and the distance between any two groups is some real number $l$, where $l > 1$. For the "gossip" problem, we want to find the quickest way for gossip that begins in one group to reach all of the other groups and participants within those groups in a social network.

In our case of the Single Source Shortest Path Problem, we are letting $K = 2$. However, we also have extended our findings for when $K$ can grow with the input size. Since the Single Source Shortest Path Problem appears in different areas of research, there has been a call for a "tool box" [DGJ05, Gol] consisting of different methods for solving this problem for various types of input. The proposed algorithm could be a possibility for this "tool box".
Chapter 4

Related Work

The research completed in the Single Source Shortest Path problem is vast and too extensive to point out all of the advances made into solving this problem. Several methods can be found in [DP84]. In this chapter, we describe some of the advancements toward solving this problem and also provide the background information for our research.

The first polynomial and well-known algorithm for the Single Source Shortest Path problem was developed by Dijkstra [Dij59]. According to Dijkstra’s algorithm, given a graph, we start with a source vertex $s$, we initialize all of the vertices in the graph by setting $d(s) = 0$ and $d(v) = \infty \forall v \in V - \{s\}$, and we maintain a priority queue containing all of the vertices in a graph. We select the vertex $v$ with the smallest distance from $s$ and remove it from the queue. Then, we update the distances of all of the vertices adjacent to $v$ and repeat the process of removing vertices in the queue and updating distances until the queue is empty. The running time of this algorithm depends on how the priority queue is structured. There have been several advances toward improving Dijkstra’s algorithm in specific cases as shown below:
4.1 New Design Paradigms

One advancement in the Single Source Shortest Path Problem was developed by Thorup, who found a linear time algorithm in the case where the edges of the graph were undirected [Tho97]. With this algorithm, Thorup used the similarities between the Minimum Spanning Tree of an undirected graph and the tree representing the Single Source Shortest Path to derive the solution. However, it should be noted that this algorithm uses word operations instead of comparisons.

4.2 Data Structuring Improvements

As we have already seen, any algorithm that adopts Dijkstra’s approach to solving the Single Source Shortest Path Problem involves using the EXTRACT-MIN() operation to find the smallest value in the queue and the DECREASE-KEY() operation to update the distance labels for each of the vertices. This implies that each vertex is extracted from the queue exactly once, and each edge is relaxed at most once. Therefore, we can represent the running time of any algorithm based on Dijkstra’s approach as:

\[ T(n, m) = n \times \text{EXTRACT-MIN}() + m \times \text{DECREASE-KEY}(). \]

We want the priority queue to balance the costs of the EXTRACT-MIN() and DECREASE-KEY() operations. In fact, the most efficient priority queue for this case is the Fibonacci Heap. The running time of Dijkstra’s algorithm using this priority queue turns out to be \( O(m + n \log n) \). There are other heaps that can be used as the priority queue for this problem such as the \( d \)-heap [CLRS01] and the \( R \)-heap [DGST88].

There is also another approach using Johnson’s implementation [Joh82]. However, this method only works when the edge lengths are integers. With this data structure,
which takes $O(\log \log C)$ time for each heap operation, we can implement Dijkstra’s algorithm in $O(m \log \log C)$ time. This is actually faster than the Fibonacci Heap implementation if $m < (n \log n)/(\log \log n)$ and slower if $m > (n \log n)/(\log \log n)$ under the same edge constraints. Therefore, the graph would need to be very sparse for this implementation to work efficiently.

### 4.3 Parameterization

This method revolves around using a specific parameter (or perhaps parameters) that can be utilized in special cases of problems due to its small size. One possible parameter for Dijkstra’s algorithm is the largest edge length, denoted as $C$. [AMOT90] shows how using this approach allows us to run Dijkstra’s algorithm in $O(m + n\sqrt{\log C})$ time.

### 4.4 Input Restriction

There have been algorithms designed specifically for when the edge lengths can be made from certain distributions. One such example is an algorithm developed by Goldberg [Gol01] where the input weights are positive and uniformly distributed. This algorithm runs in linear time in the average case and $O(m + n \log C)$ time in the worst case.

### 4.5 Parallelization

[Pap94] explains how we can take algorithms that normally run sequentially and have them run in parallel time by having different tasks of the algorithm be performed by multiple processors simultaneously. This provides an improvement in the overall running
time of the algorithm since we can reduce the running time of each parallel step. In fact, the goal of parallel algorithms is to achieve a logarithmic (or polylogarithmic) parallel time with a polynomial amount of work.

[CMMS98] provides a PRAM algorithm that divides Dijkstra’s algorithm into different phases that can be parallelized. This algorithm runs in $O(n^{1/3} \log n)$ parallel time and $O(n \log n + dn)$ work, where each edge in the algorithm is chosen with probability $d/n$.

### 4.6 Distinct Edge Lengths

In our proposed algorithm, we will be using the number of distinct edge lengths as a parameter for the running time of Dijkstra’s algorithm. However, we should note that this parameter and similar parameters have been used in the analysis of different algorithms for other problems.
Chapter 5

An $O(m + nK)$ implementation of Dijkstra’s Algorithm

In this chapter, we describe the proposed algorithm, which is a variation of Dijkstra’s algorithm that runs in $O(m + nK)$ time. To use this algorithm, we need to maintain the following structures:

(i) The set of permanently labeled vertices, denoted as $S$, and

(ii) The set of temporarily labeled vertices, denoted as $T = V - S$.

We let $d(j)$ be the distance label for vertex $j$. If $j \in S$, then vertex $j$ is permanently labeled, which means that $d(j) = \delta(j)$. We also let $d^* = \max\{d(j) : j \in S\}$ be the distance label for the vertex $j$, where $j$ was the most recently vertex added to $S$.

When we run Dijkstra’s algorithm naively (i.e. using arrays), we find that the FINDMIN() operation ends up being a bottleneck. Recall that this operation finds the vertex in $T$ that has the smallest distance label. This means that each FINDMIN() operation takes $O(|T|) = O(n)$ steps. With $n$ vertices, this gives us a total of $O(n^2)$ steps. The updating only takes $O(m)$ steps. Therefore, we need to reduce the running time of the FINDMIN() operation in order to improve Dijkstra’s algorithm. This is why
we use different priority queue implementations. Currently, the best implementations
use the Fibonacci Heap [FT87] and the Atomic Heap [FW94]. These heaps allow us to
reduce the time of the FindMin() operation when the number of distinct edge lengths
is small.

In our implementation, we let \( L = \{l_1, \ldots, l_K\} \) be the set of distinct edge lengths.
For each item \( l_t \in L \), where \( t = 1 \) to \( K \), we have a linked list \( E_t(S) = \{(i, j) \in E : i \in S, c_{ij} = l_t\} \). Each of these edges, denoted as \((i, j)\), is sorted in the order that \( j \)
is added to \( S \). In other words, if edge \((i, j)\) comes before \((i', j')\) in \( E_t(S) \), that means
\( d(i) \leq d(i') \).

We also have the pointer \( CurrentEdge(t) \) which represents the first edge \((i, j)\) of
\( E_t(S) \), where \( j \in T \). If \( E_t(S) \) does not have any edges with that property, then we let
\( CurrentEdge(t) = \emptyset \). If we do have an edge where \((i, j) = CurrentEdge(t)\), then we
define \( f(t) \) as the length of the shortest path from the vertex \( s \) to the vertex \( i \) plus the
length of edge \((i, j)\). We denote this as \( f(t) = d(i) + l_t \). Although possible, it is not
always the case that \( f(t) = d(j) \). This is because there could be other edges of different
lengths that connect to vertex \( j \).

With the help of these data structures, we can easily determine the vertex in \( T \) with
the smallest distance label by finding \( \text{argmin} \ \{ f(t) : 1 \leq t \leq K \} \). When implemented
without any priority queue data structures, the FindMin() operation runs in \( O(K) \)
time. Therefore, when \( K \) is small, we get an improvement in the overall running time.

We use the subroutine Update\((t)\) to change \( CurrentEdge(t) \) so that it either points
to the first edge in \( E_t(S) \) where the endpoint is in \( T \) or sets it to \( \emptyset \). If \( CurrentEdge(t) \)
does point to some edge \((i, j)\), then the subroutine sets \( f(t) = d(i) + c_{ij} \). Otherwise, the
subroutine sets \( f(t) = \infty \) when \( CurrentEdge(t) = \emptyset \). We denote \( CurrentEdge(t).next \)
as the operation that moves the pointer \( CurrentEdge(t) \) to point to the next edge in
$E_t(S)$.

Function \textsc{Initialize()}

1: $S := \{s\}; T := V \setminus \{s\}.$
2: $d(s) := 0; \text{pred}(s) := \emptyset.$
3: for (each vertex $v \in T$) do
   4: \hspace{1em} $d(v) = \infty; \text{pred}(v) = \emptyset.$
5: end for
6: for $(t = 1$ to $K)$ do
   7: \hspace{1em} $E_t(S) := \emptyset.$
   8: \hspace{1em} $\text{CurrentEdge}(t) := \text{NIL}.$
9: end for
10: for each edge $(s,j)$ do
   11: \hspace{1em} Add $(s,j)$ to the end of the list $E_t(S)$, where $l_t = c_{sj}.$
   12: if (CurrentEdge$(t) = \text{NIL}$) then
   13: \hspace{1em} CurrentEdge$(t) := (s,j)$
   14: end if
15: end for
16: for $(t = 1$ to $K)$ do
17: \hspace{1em} \text{UPDATE}(t)
18: end for

\textbf{Algorithm 5.0.1:} The Initialization Procedure

\textbf{Theorem 5.0.1} Algorithm 5.0.2 determines the shortest path from vertex $s$ to all other vertices in $O(m + nK)$ time.

\textbf{Proof:} The algorithm is the exact same as Dijkstra’s algorithm. The only difference is that the algorithm uses some additional data structures while implementing the \textbf{FindMin()} operation. Therefore, we already know that Algorithm 5.0.2 correctly finds the shortest paths from vertex $s$ in the graph $G$.

The initialization process takes $O(n)$ time since it involves placing all of the adjacent vertices from $s$ into the possible linked lists. As previously stated, the bottleneck of Dijkstra’s algorithm is determining $r = \text{argmin}\{f(t) : 1 \leq t \leq K\}$ and running the \textbf{UPDATE}(t) operation for all iterations. All of the other steps in the algorithm have
Function **NEW-DIJKSTRA**()

1: **INITIALIZE**()

2: while \( T \neq \emptyset \) do

3: let \( r = \text{argmin} \ \{f(t) : 1 \leq t \leq K\} \).

4: let \((i, j) = \text{CurrentEdge}(r)\).

5: \(d(j) := d(i) + l_{ij}; \ pred(j) := i\).

6: \( S = S \cup \{j\}; T := T - \{j\} \).

7: for (each edge \((j, k) \in \text{Adj}(j)\)) do

8: Add the edge to the end of the list \(E_t(S)\), where \(l_t = c_{jk}\).

9: if (CurrentEdge\((t) = \text{NIL}\) then

10: CurrentEdge\((t) := (j, k)\)

11: end if

12: end for

13: for \((t = 1 \text{ to } K)\) do

14: **UPDATE**\((t)\).

15: end for

16: end while

**Algorithm 5.0.2:** Dijkstra’s Algorithm with Few Distinct Edge Lengths

---

Function **UPDATE**\((t)\)

1: Let \((i, j) = \text{CurrentEdge}(t)\).

2: if \((j \in T)\) then

3: \(f(t) = d(i) + c_{ij}\).

4: return

5: end if

6: while \(((j \notin T) \text{ and } (\text{CurrentEdge}(t).\text{next} \neq \text{NIL}))\) do

7: Let \((i, j) = \text{CurrentEdge}(t).\text{next}\).

8: \(\text{CurrentEdge}(t) = (i, j)\).

9: end while

10: if \((j \in T)\) then

11: \(f(t) = d(i) + c_{ij}\).

12: else

13: Set \(\text{CurrentEdge}(t)\) to \(\emptyset\).

14: \(f(t) = \infty\).

15: end if

**Algorithm 5.0.3:** The Update Procedure
running times that are dominated by one of the previously mentioned steps. We should note that the running time to find \( r \) is \( O(K) \) per iteration of the while loop and \( O(nK) \) for \( n \) iterations.

We now need to determine the time for the \( \text{UPDATE}(t) \) operation for all iterations. Notice that the \( \text{UPDATE}(t) \) operation has two possible outcomes: either \( \text{CurrentEdge}(t) \) is changed or \( \text{CurrentEdge}(t) \) is not changed. We first consider the case where \( \text{CurrentEdge}(t) \) is changed. Suppose we have \((i, j) = \text{CurrentEdge}(t)\) at the beginning of some iteration, where \( i \in S \) and \( j \in S \). Since each of the edges in \( E_t(S) \) is scanned sequentially, edge \((i, j)\) will not be scanned again after we update \( \text{CurrentEdge}(t) \). In other words, each edge is scanned at most once. Therefore, the running time of this case is \( O(m) \).

We now consider the case where \( \text{CurrentEdge}(t) \) is not changed. Since there is no change, the running time for a single iteration of this case is constant. However, we should note that the \( \text{UPDATE}(t) \) operation is called \( O(nK) \) times. This implies that the running time for all iterations of this case is \( O(nK) \). Therefore, we get the total running time for the \( \text{UPDATE}(t) \) operation as \( O(m + nK) \), which is also the total running time of Algorithm 5.0.2. \( \square \)

The original motivation for this research and thesis was to explore the case where \( K = 2 \). In this case, the algorithm is actually efficient. In the next chapter, we show how we can reduce the time of the algorithm whenever \( K \) can grow with respect to the size of the problem.

### 5.1 Example of the \( O(m + nK) \) Algorithm

We now demonstrate how to use this algorithm. Figure 5.1 shows the states of executing our algorithm on a graph with 4 vertices. Figure 5.1(a) is our starting graph. Since
$K = 2$, we have $l_1 = 1$ and $l_2 = 3$. In line 1 of the NEW-DIJKSTRA() procedure, we first run the INITIALIZE() procedure where $S = \{s\}$ and $T = \{v_1, v_2, v_3\}$. After scanning all of the edges adjacent from $s$, we get lists $E_1(S) = \{(s, v_2)\}$ where $CurrentEdge(1) = (s, v_2)$ and $E_2(S) = \{(s, v_1)\}$ where $CurrentEdge(2) = (s, v_1)$.

From line 3, we can see that $r = 1$, which means we let $d(v_2) = 1$ and add $v_2$ to $S$ giving us $S = \{s, v_2\}$ and $T = \{v_1, v_3\}$. Lines 7 to 11 scan the edges adjacent to $v_2$ and add them to $E_1(S)$ and $E_2(S)$ accordingly. This gives us the lists $E_1(S) = \{(s, v_2), (v_2, v_1)\}$ where $CurrentEdge(1) = (v_2, v_1)$ and $E_2(S) = \{(s, v_1), (v_2, v_3)\}$ where $CurrentEdge(2) = (s, v_1)$. This state of the algorithm is shown in Figure 5.1(b).

We now have to run the UPDATE($t$) procedure for each of our lists. For $CurrentEdge(1)$, since $v_2 \in S$, we update $CurrentEdge(1)$ to $(v_2, v_1)$ in lines 5 to 8 and let $f(1) = 2$ in lines 9 to 11. For $CurrentEdge(2)$, since $v_1 \in T$, we leave $CurrentEdge(2)$ alone and return.

We run the next iteration since $T \neq \emptyset$. We see that $r = 1$, which means we let $d(v_1) = 2$ and add $v_1$ to $S$ giving us $S = \{s, v_1, v_2\}$ and $T = \{v_2\}$. We scan the edges adjacent to $v_1$ and add them to our lists giving us $E_1(S) = \{(s, v_2), (v_2, v_1), (v_1, v_3)\}$ where $CurrentEdge(1) = (v_2, v_1)$ and $E_2(S) = \{(s, v_1), (v_2, v_3)\}$ where $CurrentEdge(2) = (s, v_1)$. When we run the UPDATE($t$) procedure, we change $CurrentEdge(1)$ to $(v_1, v_3)$,

![Figure 5.1: The execution of our $O(m+nK)$ algorithm.](image)
set $f(1) = 3$, change $CurrentEdge(2)$ to $(v_2, v_3)$, and set $f(2) = 4$. This state of the algorithm is shown in Figure 5.1(c).

We run one more iteration since $T \neq \emptyset$. We see that $r = 1$, which means we let $d(v_3) = 3$ and add $v_3$ to $S$ giving us $S = \{s, v_1, v_2, v_3\}$ and $T = \emptyset$. We scan the edges adjacent to $v_3$ and find that there are no other edges adjacent to $v_3$. This gives us $E_1(S) = \{(s, v_2), (v_2, v_1), (v_1, v_3)\}$ where $CurrentEdge(1) = (v_1, v_3)$ and $E_2(S) = \{(s, v_1), (v_2, v_3)\}$ where $CurrentEdge(2) = (v_2, v_3)$. When we run the Update($t$) procedure, we change both $CurrentEdge(1)$ and $CurrentEdge(2)$ to $\emptyset$ and set both $f(1)$ and $f(2)$ to $\infty$ since $v_3 \notin T$. The final state of the algorithm is shown in Figure 5.1(d).
Chapter 6

A Faster Algorithm if $K$ is Permitted to Grow with Problem Size

If $K$ is not a constant value, we can actually modify the proposed algorithm to achieve an even better running time. We let $q = \frac{nK}{m}$, where $q \geq 2$. Otherwise, our algorithm runs in linear time. Furthermore, we let $h = \frac{K}{q}$ to make our calculations simpler. We assume without loss of generality that $q$ is an integer divisor of $K$.

Given the changes with $K$, we can show that Algorithm 5.0.2 can be modified to run in $O(m \log q)$ time. In order for this to work, we need to improve upon the bottlenecks of the algorithm that depend on $K$. Specifically, we need to reduce the time it takes to determine $r$, and we need to reduce the number of times we call UPDATE($t$). We will first address the issue of the FINDMIN() operation, the process of computing $r$.

In order to improve the FINDMIN() operation to determine $r$, we can use $h$ different binary heaps to store the possible values of $f()$. More information about binary heaps can be found in [CLRS01]. The first binary heap, denoted as $H_1$, contains the values $f(j)$, where $j = 1$ to $q$. The second binary heap, denoted as $H_2$, contains the values $f(j)$, where $j = q + 1$ to $2q$. This process repeats up to $H_h$. We know that finding
the smallest value in a binary heap $H_i$ takes $O(1)$ time since we are simply using the Extract-Min() operation for binary heaps. The time to insert an item into heap $H_i$ or delete and item from heap $H_i$ is $O(\log q)$.

To find the minimum value of $f()$, we need to find the minimum value of each of the $h$ heaps, and then select the smallest of these values. We already know that finding the smallest value in a heap is $O(1)$ time. Since we have $h$ heaps, the running time for the FindMin() operation is $O(h)$ for each iteration of the while loop. With $n$ iterations, we get a total running time of $O(hn) = O(m)$. Once we find this value, we need to remove it from its respective heap. We know that this takes $O(\log q)$ time per iteration. Therefore, the total time is $O(n \log q)$ for all $n$ iterations.

We now need to look at improving the Update($t$) operation. For this to work, we need to change how we use the CurrentEdge pointer. We say that CurrentEdge($t$) is invalid if CurrentEdge($t$) = ($i, j$) and $i, j \in S$. To make the desired improvements, we need to allow CurrentEdge($t$) to be invalid while implementing the algorithm. This also involves modifying the FindMin() operation again. We now say that if the smallest value in heap $H_i$ is $f(t)$ where $1 \leq i \leq h$, and CurrentEdge($t$) is invalid, we execute the Update($t$) operation and find the next smallest value in $H_i$. We repeat this until CurrentEdge($t$) is valid.

With this method, whenever we call Update($t$), we guarantee that CurrentEdge($t$) is changed. Furthermore, whenever we select a minimum value among the heaps, we know that the minimum values in each of the heaps must correspond to a valid edge since each vertex with the minimum value is in $T$. Each time we modify CurrentEdge($t$), we are also modifying one of the heaps. This gives us a running time of $O(\log q)$. Since there are at most $m$ modifications to CurrentEdge($t$), we get a total running time for the Update($t$) operation as $O(m \log q)$.
We conclude our findings with this variation with Theorem 6.0.1.

**Theorem 6.0.1** The binary heap implementation of Dijkstra’s algorithm with $O(K_q)$ binary heaps of size $O(q)$ with $q = \frac{nK}{m}$ determines the shortest path from vertex $s$ to all other vertices in $O(m \log q) = O(m \log \left( \frac{nK}{m} \right))$ time.
Chapter 7
Empirical Results for the $K$-Color Algorithm

7.1 Experimental Setup

We now examine how well our $K$-color Dijkstra algorithm performs using different graph families. Some of the graph generators and instances used are part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [DGJ05]:

- **Random Graphs**: We use the Erdos-Renyi random graph model to generate the graphs for our testing. This guarantees that the graph is connected. However, we should note that it is possible for the generator to include parallel edges and/or self-loops. The experiments also use varying sparse and dense graphs.

- **Mesh Graphs**: This generator creates two-dimensional mesh graphs where $x$ and $y$ are the dimensions. This study involves generating both $Long$ ($x = \frac{n}{16}$, $y = 16$) and $Square$ ($x = y = \sqrt{n}$) grids. Unlike random graphs, the degree distribution of mesh graphs is uniform.

- **Small-World Graphs**: For real-world networks, we use the R-MAT graph model
in order to generate graphs containing small-world characteristics [CZF04]. These kinds of graphs have low diameters and unbalanced degree distributions.

Since our algorithm is designed for networks where $K$ is small, we choose the edge weights from a fixed set of distinct random integers.

We compare the execution time of our algorithm with two other algorithms on every graph family. The first algorithm is a reference SSSPP solver that is used in the 9th DIMACS Shortest Paths Challenge. This algorithm is an efficient implementation of Goldberg’s algorithm, [Gol01] which runs in linear time in the expected case and is efficient for integer edge weights. The second algorithm for comparison is the baseline Breadth-First Search (BFS), where the running time is a natural lower bound for SSSPP implementations.

Not only can we compare the running times of our algorithm with the DIMACS reference solver, but we can also compare the code used for both algorithms. Both algorithms use adjacency arrays to represent the graphs, they are written in C/C++, and they are compiled and run in identical experimental settings. However, we should note that although the implementation for our algorithm works for both real and integer edge weights, it is only efficient when the number of edge lengths is distinct. Since the DIMACS solver only works for integer weights, we only used integer edge weights in our experiments.

The testing is done on a 2.8 GHz 32-bit Intel Xeon machine with 4GB memory, 512 KB cache, and running RedHat Enterprise Linux 4 (linux kernel 2.6.9). We compare the performance of our implementation with the DIMACS reference solver [DGJ05]. Both implementations are compiled with the Intel C compiler (icc) version 9.0, and the optimization flag is set to -O3. We report the average execution time of five trials for
each test.

7.2 Results and Analysis

We examine the empirical results of our algorithm and how well it performs based on the type of graph, the size of the graph, the value of $K$, and the distribution of the edge weights. The execution times of Breadth-First Search of all graph instances studied are noted in Tables 7.1 and 7.2. The figures below give us the execution times of the shortest path implementations normalized to the BFS time. This means that the best we can get is a ratio of 1.0. However, smaller values are desirable.

The study involves looking at the execution times based on the size of the problems. We use three different graph families, vary the size up to two orders of magnitude, and compute the average SSSPP execution times for both our code and the reference code. Figure 7.1 gives the normalized values for the different graph types. The sizes of the problem are given in Table 7.1. We let $K = 2$, and the ratio of the largest edge length to the smallest edge length, denoted as $C$, is 100. We should also note that the ratio of edges to vertices, denoted as $\frac{m}{n}$, is 4 in all of our tests. In Figure 7.1, we find that our algorithm is an improvement compared to the reference solver for all of the different graph families. Furthermore, the performance ratio is less than 2 in most of the cases which is actually very significant.

Not only do we have improvements with all of the graph families, but we also see that the performance improvements for long and square mesh graphs are much higher than random graphs. This is because we do not use the priority queue data structure in our implementation. This allows us to avoid the overhead involved with using a priority queue for long paths in mesh graphs, such as frequently updating the distance values.
We should also note that the performance for larger graphs ended up being better than smaller graphs. Furthermore, the performance ratios for spare random graphs (1.02-1.22) are very impressive for the cases we studied.

<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>BFS time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Long mesh</td>
</tr>
<tr>
<td>1 100K vertices, 400K edges</td>
<td>50</td>
</tr>
<tr>
<td>2 500K vertices, 2M edges</td>
<td>290</td>
</tr>
<tr>
<td>3 1M vertices, 4M edges</td>
<td>660</td>
</tr>
<tr>
<td>4 5M vertices, 20M edges</td>
<td>4160</td>
</tr>
<tr>
<td>5 10M vertices, 40M edges</td>
<td>8590</td>
</tr>
</tbody>
</table>

Table 7.1: Breadth-First Search execution time (in milliseconds) for various graph families on the test sequential platform.

Figure 7.1: Performance of our shortest implementation and the reference solver for three graph families, as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest.

The next study involves looking at the performance of the algorithm for each of the graph families as we vary the value of $K$, the number of distinct edge weights. In each
case, we look at when $K$ is 1, 2, 4, 6, and 8 and plot the execution times. Figure 7.2 gives the normalized execution times for a sparse random graph containing 4 million vertices and 16 million edges. We find that our algorithm is significantly faster compared to the reference solver. In fact, we see that when $K = 8$, our implementation is up to 70% faster. Furthermore, our algorithm scales slower compared to the reference solver as $K$ increases. Our results show that the execution times for the SSSPP for random graphs is lower than other graph families.

![Figure 7.2: Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of $K$ is varied.](image)

Figure 7.2: Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of $K$ is varied.

Figure 7.3 gives the normalized execution times for small-world graphs. Like sparse random graphs, small-world graphs have low diameters. However, they also have topological properties that are used in large-scale social and biological networks. In our case, we see that both our implementation and the reference solver perform very similarly.

For long meshes, as shown in Figure 7.4, we find that the performance of our algorithm is a significant improvement compared to the reference solver. Furthermore, the
Figure 7.3: Normalized SSSP Performance for a small-world graph (4 million vertices, 16 million edges) as the value of $K$ is varied.

The execution time of our implementation scales a lot slower than the reference solver as $K$ increases. As we can see in Figure 7.5, we have similar results for square meshes for all values of $K$.

Figure 7.6 gives the normalized execution times for a dense random graph containing 100 thousand vertices and 100 million edges as the value of $K$ varies. In this case, the reference solver ends up being faster than our algorithm. However, as $K$ increases, the speedup falls. This is because BFS has a fast execution time since this is a small graph instance. As a result, the overhead in running our algorithm is quite high. Therefore, its execution time is not as fast as the reference solver for dense graphs of this size.

We can find the execution times of BFS for these graph families in Table 7.2. The ratio of the highest edge weight to the lowest edge weight is also given, denoted as $C$. This is because the worse case running time of the reference solver needs this value. However, our algorithm is independent of $C$. 

25
Figure 7.4: Normalized SSSP Performance for a long mesh (4 million vertices, 16 million edges) as the value of K is varied.

Table 7.2: Breadth-First Search execution time (in milliseconds) for various graph families (the value of K is varied in experiments) on the test sequential platform.
Figure 7.5: Normalized SSSP Performance for a square mesh (4 million vertices, 16 million edges) as the value of K is varied.

Figure 7.6: Normalized SSSP Performance for a dense random graph (100K vertices, 10 million edges) as the value of K is varied.
Chapter 8

Extension of the Empirical Results

8.1 Experimental Setup

We now extend the experiments established in Chapter 7. This time, we look at the performance of our algorithm for varying edge lengths. As done in the first set of tests, we will use the graph generators from the 9th DIMACS Shortest Path Implementation Challenge benchmark package [DGJ05]. With these tests, we will only be looking at sparse random graphs and square mesh graphs.

We compare the execution time of this algorithm with the reference SSSPP solver provided by [Gol] and BFS on each of the graph families. For this series of experiments, both implementations use the same adjacency array representation for the graph, are written in C/C++, and compiled and run in the same experimental settings. Both implementations can process graphs with real and integer weights, but we only use integer weights for these tests.

Our test platform for performance results is a 2.0 GHz 32-bit Intel Core 2 Duo machine with 4 GB memory, 2 MB cache and running Ubuntu version 8.10. We compare the sequential performance of both of the implementations. Both of the codes are compiled with the Intel C compiler (icc) version 11.0 and the optimization flag is set
to -O3. However, we should note that because these series of tests are performed on a
different machine, we will see differences in the execution times compared to the first
series of experiments. We report the average execution time of five independent trials
for each test.

8.2 Results and Analysis when $K = 2$

We examine the empirical results of our algorithm and the reference SSSPP solver to see
how they perform against each other. This series of tests is based on the graph topology,
the size of the graph, and the distribution of edge weights. As we did in the first set of
tests, we include the execution time of BFS on all of the graph families studied in Table
8.1. All of the figures give the shortest path implementations normalized to BFS time.
Therefore, the best we can achieve is a ratio of 1.0.

<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>BFS time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Graph size</td>
</tr>
<tr>
<td></td>
<td>Square mesh</td>
</tr>
<tr>
<td>1</td>
<td>100K vertices, 400K edges</td>
</tr>
<tr>
<td>2</td>
<td>500K vertices, 2M edges</td>
</tr>
<tr>
<td>3</td>
<td>1M vertices, 4M edges</td>
</tr>
<tr>
<td>4</td>
<td>5M vertices, 20M edges</td>
</tr>
<tr>
<td>5</td>
<td>10M vertices, 40M edges</td>
</tr>
</tbody>
</table>

Table 8.1: Breadth-First Search execution time (in milliseconds) for various graph fam-
ilies on the test sequential platform when $N = 8$ and $K = 2$.

We examine the execution times based on the size of the graphs. We use two dif-
ferent graph families and find the average execution times for both our algorithm and
the reference SSSPP reference solver provided by Goldberg. Figure 8.1 provides the
normalized values for the different graph types, which is provided in Table 8.1. For
these tests, we let $K = 2$. Furthermore, we let the largest edge length be $10^N$, where $N$
is some integer. For Figure 8.1, we let $N = 8$. Also, like in the last series of tests, we let the ratio of edges to vertices be 4. As we can see in Figure 8.1, our algorithm is a significant improvement compared to the reference solver when the length of the largest edge is substantially large (i.e. $10^8$). This is primarily because the reference solver uses buckets, which implies that the running time of the solver is dependent on the length of the largest edge.

![Figure 8.1: Performance of our implementation and the reference solver for two graph families when $N = 8$ and $K = 2$ as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest.](image)

The next set of experiments explore the performance of our algorithm as $N$ varies. In these cases, we looked at when $N$ is 1, 2, 4, 6, and 8. Figure 8.2 gives the normalized
execution times for square mesh graphs, and Figure 8.3 gives the normalized execution
times for sparse random graphs. As we can see for both graphs, our algorithm is a
significant improvement, especially as $N$ increases.

![Chart showing normalized SSSP performance for different values of $N$.](image)

Figure 8.2: Normalized SSSP Performance for a square mesh (4 million vertices, 16
million edges) as the value of N is varied and $K = 2$.

### 8.3 Results and Analysis when $K = 3$

We now perform the same experiments except $K = 3$ in this case. Like the previous set
of tests, we base our tests on the graph topology, the size of the problem, and how the
dge weights are distributed. The execution time of BFS for these tests can be found in
Figure 8.3: Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of N is varied and K = 2.

Table 8.2. Finally, all of the figures give the times in normalized BFS time.

We next look at the results of our experiments. Just like in the tests where K = 2, we use two graph families and find the average execution times for the two algorithms. Figure 8.4 provides the normalized values for the different graph types, which is provided in Table 8.2. For these tests, we let K = 3. We have 10^N remain as the largest edge length. For Figure 8.1, we let N = 8. As shown in Figure 8.4, even when K = 3, we see that our algorithm is faster than the reference solver.

We then examine the cases where N is 1, 2, 4, 6, and 8. Figure 8.5 gives the nor-
Table 8.2: Breadth-First Search execution time (in milliseconds) for various graph families on the test sequential platform when \(N = 8\) and \(K = 3\).

<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>BFS time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Graph size</td>
</tr>
<tr>
<td>1</td>
<td>100K vertices, 400K edges</td>
</tr>
<tr>
<td>2</td>
<td>500K vertices, 2M edges</td>
</tr>
<tr>
<td>3</td>
<td>1M vertices, 4M edges</td>
</tr>
<tr>
<td>4</td>
<td>5M vertices, 20M edges</td>
</tr>
<tr>
<td>5</td>
<td>10M vertices, 40M edges</td>
</tr>
</tbody>
</table>

As we can see for both graphs, our algorithm is a significant improvement, especially as \(N\) increases.
Figure 8.4: Performance of our implementation and the reference solver for two graph families when $N = 8$ and $K = 3$ as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest.
Figure 8.5: Normalized SSSP Performance for a square mesh (4 million vertices, 16 million edges) as the value of N is varied and $K = 3$. 
Figure 8.6: Normalized SSSP Performance for a sparse random graph (4 million vertices, 16 million edges) as the value of N is varied and $K = 3$. 
Chapter 9

Open Problems

9.1 Graphs with Negative Edges

Given the graph described in Chapter 2, suppose we allow at least one of the edge lengths be negative. In the case where \( K = 2 \), we can have \( l_1 < 0 \) and \( l_2 \geq 0 \). The problem now is if we can develop an algorithm for the Single Source Shortest Path Problem for graphs with negative edges that exploits the number of distinct edge lengths and provides an improvement in the running time when \( K \) is small. We already know that our current algorithm will not work since it assumes that for each vertex \( v \) removed from \( T \) and added to \( S \), \( d(v) \) is at least as large as the distance labels of all of the vertices already in \( S \). This can cause \( d(v) \) to have an incorrect value.

Currently, the best known algorithm for the Single Source Shortest Path Problem for graphs with negative edges is the Bellman-Ford algorithm, which is described in [Bel58] and runs in \( O(mn) \) time. Therefore, we wish to explore modifying the Bellman-Ford algorithm or even using a different approach that gives us an algorithm that runs better than \( O(mn) \) when \( K \) is small.
9.2 Randomized Algorithms

As described in both [CLRS01] and [Pap94], randomized algorithms have grown in popularity due to the simplicity of implementing these algorithms and how fast their expected running times are. Given these benefits, we would be curious to see if we can use a randomized approach to make our current algorithm simpler. One idea with this problem is to see what would happen if we randomly select our $f(t)$ and determine the probability of getting the correct distance labels.

9.3 Parallel Algorithms

In chapter 4, we mentioned how we can have our algorithms be either work-efficient or work-optimal or even both by running several tasks of the algorithm in parallel by using multiple processors. We would be interested in seeing if we can use this information to develop a work-optimal and work-efficient parallel algorithm for solving this problem.

9.4 All Pairs Shortest Path

Although this thesis discusses the shortest path problem for when there is a single source, there are cases where we have multiple sources. In fact, one such problem is the All-Pairs Shortest Paths Problem, which is defined in [CLRS01] as follows: Given a weighted, directed graph $G = (V, E)$, we wish to find, for every pair of vertices $u, v \in V$, the shortest path from $u$ to $v$.

If the edge lengths are positive, then we can use Dijkstra’s algorithm $|V|$ times, which would give us a running time of $O(n^3)$. Using the Fibonacci Heap implementation, we get $O(mn + n^2 \log n)$. However, if the edge lengths are negative, we would have to use
an implementation such as Bellman-Ford, which would run in \( O(n^2m) \).

The biggest difference between these two problems is how we set up the graph for the problems. With the single source problem, we use the adjacency list to represent our graph. However, for the all-pairs problem, we mainly use the adjacency matrix to represent our graph. Furthermore, we can use dynamic programming with the adjacency matrix to solve this problem in \( \Theta(n^3) \) time. This approach is known as the Floyd-Warshall algorithm [Flo62].

Suppose we applied our constraint to this problem, where there is a distinct number of edge lengths. If we ran our algorithm \( n \) times, one for each vertex, we would get a running time of \( O(mn+n^2K) \). We would like to see if we can use a dynamic programming approach to achieve a running time better than this.
Chapter 10

Conclusions

The SSSPP problem is a fundamental research problem in theoretical and practical applications in both computer science and operations research. In this thesis, we examined this problem when the number of distinct edge lengths, denoted as \( K \), is small. By exploiting this parameter, we developed an algorithm that computes the single source shortest path for a graph in \( O(m + nK) \) time.

Furthermore, we explained how we can improve our algorithm for when \( K \) is permitted to grow with the input size. This was accomplished by incorporating binary heaps to reduce the running time of the \texttt{Find-Min()} operation. As a result, our modified algorithm runs in \( O(m \log \frac{nK}{m}) \).

Finally, we empirically showed that our algorithm is efficient compared to Goldberg’s reference solver through empirical analysis. We demonstrate this by comparing the two implementations based on the values of \( K \), graph families, the size of the graphs, the weight of the largest edge length, and the weight distributions. The results show that our implementation does indeed run faster than Goldberg’s reference solver the vast majority of the cases studied.
Bibliography


