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

James B. Orlin [a], Kamesh Madduri [b,1], K. Subramani [c,*,2], M. Williamson [c]

[a] *Sloan School of Management, MIT, Cambridge, MA, USA*
[b] *Computational Research Division, Lawrence Berkeley Laboratory, Berkeley, CA, USA*
[c] *LDCSEE, West Virginia University, Morgantown, WV, USA*

ARTICLE INFO

ABSTRACT

In this paper, we propose an efficient method for implementing Dijkstra's algorithm for the Single Source Shortest Path Problem (SSSPP) in a graph whose edges have positive length, and where there are few distinct edge lengths. The SSSPP is one of the most widely studied problems in theoretical computer science and operations research. On a graph with n vertices, m edges and K distinct edge lengths, our algorithm runs in $O(m)$ time if $nK \leqslant 2m$, and $O(m \log \frac{nK}{m})$ time, otherwise. We tested our algorithm against some of the fastest algorithms for SSSPP on graphs with arbitrary but positive lengths. Our experiments on graphs with few edge lengths confirmed our theoretical results, as the proposed algorithm consistently dominated the other SSSPP algorithms, which did not exploit the special structure of having few distinct edge lengths.

© 2009 Published by Elsevier B.V.

1. Introduction

In this paper, we provide an algorithm for solving the Single Source Shortest Path Problem (SSSPP) on a graph whose edges have positive length. The SSSPP is an extremely well-studied problem in both the operations research and the theoretical computer science communities because of its applicability in a wide range of domains. Ahuja et al. [2] describe a number of applications of the SSSPP, as well as efficient algorithms for the same. This paper provides an efficient algorithm for the SSSPP in the case where the number of distinct edge lengths is small. Our motivation for focusing on problems with few distinct edge lengths comes from a problem that arises in social networks (see Section 3).

We consider a graph with n vertices, m edges, and K distinct edge lengths. We provide two algorithms: The first algorithm is a simple implementation of Dijkstra's algorithm that runs in time $O(m + nK)$. The second algorithm modifies the first algorithm by using binary heaps to speed up the FINDMIN() operation. Assuming that $nK > 2m$, its running time is $O(m \log \frac{nK}{m})$.

For various ranges of the parameters n, m, and K, the running time of our algorithm is less than the running time of Fredman and Tarjan's Fibonacci Heap implementation [8], which runs in $O(m + n \log n)$ time. In fact, it improves upon the Atomic Heap implementation of Fredman and Willard [9], which runs in $O(m + \frac{n \log n}{\log \log n})$ time. (This latter paper relies on a slightly different model of computation than is normally assumed in papers on algorithms.) In particular, our algorithm

* Corresponding author.
 E-mail addresses: jorlin@mit.edu (J.B. Orlin), kmadduri@lbl.gov (K. Madduri), ksmani@csee.wvu.edu (K. Subramani), mwilli65@mix.wvu.edu (M. Williamson).

runs in $O(m)$ time whenever $nK = O(m)$. We also note that even if all edge lengths are distinct, the running time of our algorithm is $O(m \log m)$, which is the same time as the binary heap implementation of Dijkstra's algorithm.

The main contributions of this paper are as follows.

 (i) A new algorithm for the SSSPP problem that is parameterized by the number of distinct edge lengths.
(ii) An empirical analysis of our algorithm that demonstrates the superiority of our approach when the number of distinct edge lengths is small.

The rest of this paper is organized as follows. Section 2 formally specifies the problem under consideration. Section 3 describes the motivation for our work. Related work in the literature is discussed in Section 4. Section 5 describes and analyzes the $O(m + nK)$ implementation of Dijkstra's algorithm. Section 6 describes the $O(m \log \frac{nK}{m})$ implementation and also provides a proof of its running time. In Section 7, we provide an empirical analysis, confirming the improved performance of our algorithm on graphs with few distinct edge lengths. We offer brief conclusions in Section 8.

2. Statement of problem

We consider a directed graph $G = (V, E)$, with a vertex set V with n vertices, and an edge set E with m edges. For each vertex $v \in V$, we let $E(v)$ denote the set of edges directed out of v. Let $L = \{l_1, \ldots, l_K\}$ be the set of distinct nonnegative edge lengths given in increasing order. We assume that L is provided as part of the input and stored as an array. Each edge $(i, j) \in E$ has an edge length $c_{ij} \in L$.

Rather than store the edge length c_{ij} explicitly, we assume that associated with each edge (i, j) is an index t_{ij} such that $c_{ij} = l_{t_{ij}}$. We note that in practice, one can determine L and all of the indices in $O(m + K \log K)$ expected time; we first use perfect hashing to identify the K distinct edge lengths, and then sort the edge lengths.

There is a special vertex $s \in V$ called the *source*. We let $\delta(v)$ denote the length of the shortest path in G from vertex s to vertex v. If there is no path from s to v, then $\delta(v) = \infty$. The goal of the SSSPP is to identify a shortest path from vertex s to each other vertex that is reachable from vertex s.

3. Motivation

Our work was motivated by the "gossip" problem for social networks. Consider a social network, which is composed of clusters of participants. We model the intra-cluster distance by the value 1 and the inter-cluster distances by a real number l, where $l > 1$. The goal is then to determine the fastest manner in which gossip originating in a cluster can reach all the participants in the social network. This is a special case of SSSPP in which $K = 2$.

Although the motivating example has $K = 2$, we have extended our results to values of K that can grow with the input size.

Given that the SSSPP problem arises in so many domains, researchers have called for a "toolbox" [5,10] of different implementations that are efficient for different types of input. Possibly, the algorithm presented here would be appropriate for such a toolbox.

4. Related work

The literature on the SSSPP problem is large; interested readers are referred to Ahuja et al. [1]. In what follows, we briefly outline various paradigms in algorithmic advancement and provide a context for our work.

The first polynomial time algorithm for the SSSPP problem was devised by Dijkstra [7], the running time of the algorithm depends upon the data structure used to implement the priority queue, which is part of the algorithm. Since then, advances in algorithmic techniques have been along the following fronts:

 (i) New design paradigms – Thorup provided a linear-time algorithm for the SSSPP when the graph edges are undirected [14]. He exploited the connection between the Minimum Spanning Tree of an undirected graph and the Single Source Shortest paths tree.
(ii) Data Structure improvements – Algorithms based on Dijkstra's approach perform a series of EXTRACT-MIN() and DECREASE-KEY() operations. Inasmuch as each vertex is extracted only once and each edge is processed at most once, the running time of any such algorithm can be represented as: $T(n, m) = n * \text{EXTRACT-MIN}() + m * \text{DECREASE-KEY}()$.
     An optimized priority queue design balances the costs between the two operations; in the comparison based-model, the most efficient priority queue for this pair of operations is the Fibonacci Heap. Dijkstra's algorithm with Fibonacci Heaps has a running time of $O(m + n \log n)$. Other heaps proposed for this problem include the d-heap [3] and the R-heap [6].
(iii) Parameterization – In this approach, the design focuses on a certain parameter (or parameters) that may be small in magnitude for an interesting subset of problems. One such parameter is the largest edge length (say C); Ref. [2] describes how Dijkstra's approach can be made to run in $O(m + n\sqrt{\log C})$ time.

(iv) Input restriction – Special-purpose algorithms have been designed for the case in which the edge lengths are drawn from certain distributions. The analysis of such an algorithm exploits this distribution and provides an estimate of the expected running time [12].

We believe that this is the first shortest path paper to express the running time in terms of the number of distinct edge lengths. However, this parameter or a closely related parameter has been part of the analysis of algorithms for other problems.

5. An $O(m + nK)$ implementation of Dijkstra's algorithm

In this section, we provide an $O(m + nK)$ implementation of Dijkstra's algorithm for solving the SSSPP. While running Dijkstra's algorithm, we maintain the following structures:

 (i) the set S, which denotes the set of permanently labeled vertices, and
(ii) the set $T = V - S$, which denotes the set of temporarily labeled vertices.

The value $d(j)$ is the distance label of vertex j. If $j \in S$, then $d(j) = \delta(j)$ is the length of the shortest path from vertex s to vertex j in G. Finally, we let $d^* = \max\{d(j): j \in S\}$ be the distance label of the vertex most recently added to S.

When implemented naively, the bottleneck operation in Dijkstra's algorithm is the FINDMIN() operation, which identifies the minimum distance label of a vertex in T. Each FINDMIN() operation takes $O(|T|)$ steps and thus the FINDMIN() operations take $O(n^2)$ steps in all. All other updates take $O(m)$ steps in total. In order to reduce the running time for the FINDMIN() step, Dijkstra's algorithm relies on one of many different implementations of a priority queue. Of these, the implementations with the best asymptotic bounds are the Fibonacci Heap implementation [8] and the Atomic Heap implementation [9]. Here we can dramatically speed up the FINDMIN() operations in the case that the number of distinct edge lengths is small.

Let $L = \{l_1, \ldots, l_K\}$ be the set of distinct edge lengths. For each $t = 1$ to K, the algorithm will maintain a list $E_t(S) = \{(i, j) \in E: i \in S, c_{ij} = l_t\}$. These edges are sorted in the order that the tail of the edge is added to S. That is, if edge (i, j) occurs prior to edge (i', j') on $E_t(S)$, then $d(i) \leqslant d(i')$. The algorithm also maintains $CurrentEdge(t)$, which is the first edge (i, j) of $E_t(S)$ such that $j \in T$. If no such edge exists in $E_t(S)$, then $CurrentEdge(t) = \emptyset$. If $(i, j) = CurrentEdge(t)$, then we let $f(t) = d(i) + l_t$, which is the length of the shortest path from vertex s to vertex i followed by edge (i, j). It is not necessarily the case that $f(t) = d(j)$ because there may be edges of other lengths directed into vertex j.

These additional data structures makes it possible to determine the vertex in T with minimum distance label by determining $argmin\{f(t): 1 \leqslant t \leqslant K\}$. The time for this FINDMIN() operation is $O(K)$ if implemented directly (and naively) without any priority queue data structure. This leads to an improvement in the overall running time for Dijkstra's algorithm when K is small.

The subroutine UPDATE(t) moves the pointer $CurrentEdge(t)$ so that it points to the first edge whose endpoint is in T (or sets $CurrentEdge(t)$ to $\emptyset$). If $CurrentEdge(t) = (i, j)$, then UPDATE(t) also sets $f(t) = d(i) + c_{ij}$. If $CurrentEdge(t) = \emptyset$, then UPDATE($t$) sets $f(t) = \infty$. The operator $CurrentEdge(t).next$ moves the $CurrentEdge$ pointer by one step in the linked list representing $E_t(S)$ (Algorithm 5.1).

Theorem 5.1. *Algorithm* 5.2 *determines the shortest path from vertex s to all other vertices in $O(m + nK)$ time.*

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

Algorithm 5.1. The initialization procedure.

Function NEW-DIJKSTRA()
1: INITIALIZE()
2: **while** $(T \neq \emptyset)$ **do**
3:     let $r = argmin\{f(t): \ 1 \leqslant t \leqslant K\}$.
4:     let $(i, j) = CurrentEdge(r)$.
5:     $d(j) := d(i) + l_r$; $pred(j) := i$.
6:     $S = S \cup \{j\}$; $T := T - \{j\}$.
7:     **for** (each edge $(j, k) \in E(j)$) **do**
8:         Add the edge (j, k) to the end of the list $E_t(S)$, where $l_t = c_{jk}$.
9:         **if** $(CurrentEdge(t) = NIL)$ **then**
10:             $CurrentEdge(t) := (j, k)$
11:         **end if**
12:     **end for**
13:     **for** $(t = 1$ to $K)$ **do**
14:         UPDATE(t).
15:     **end for**
16: **end while**

Algorithm 5.2. Dijkstra's algorithm with few distinct edge lengths.

Function UPDATE(t)
1: Let $(i, j) = CurrentEdge(t)$.
2: **if** $(j \in T)$ **then**
3:     $f(t) = d(i) + c_{ij}$.
4:     **return**
5: **end if**
6: **while** $((j \notin T)$ **and** $(CurrentEdge(t).next \neq NIL))$ **do**
7:     Let $(i, j) = CurrentEdge(t).next$.
8:     $CurrentEdge(t) = (i, j)$.
9: **end while**
10: **if** $(j \in T)$ **then**
11:     $f(t) = d(i) + c_{ij}$.
12: **else**
13:     Set $CurrentEdge(t)$ to $\emptyset$.
14:     $f(t) = \infty$.
15: **end if**

Algorithm 5.3. The update procedure.

Proof. The algorithm is identical to Dijkstra's algorithm except that it maintains some additional data structures to carry out the FINDMIN() operation. Therefore, Algorithm 5.2 computes the shortest paths from vertex s correctly.

The initialization takes $O(n)$ time. The potential bottleneck operations are the determining of $r = argmin\{f(t): \ 1 \leqslant t \leqslant K\}$ and the time to perform UPDATE(t) over all iterations. All other steps have running times dominated by one of these two steps. We first note that the time to compute (see Algorithm 5.3) $r = argmin\{f(t): \ 1 \leqslant t \leqslant K\}$ is $O(K)$ per iteration of the **while** loop and $O(nK)$ over all iterations.

We next consider the time needed to perform UPDATE(t) over all iterations. The procedure UPDATE(t) is called $O(nK)$ times, and its total running time is $O(m + nK)$. To see this, first note that the running time as restricted to iterations in which $CurrentEdge(t)$ is not changed is $O(nK)$. We now consider those iterations at which $CurrentEdge(t)$ is changed. Suppose $(i, j) = CurrentEdge(t)$ at the beginning of an iteration, and suppose that $i \in S$ and $j \in S$. Because the edges in $E_t(S)$ are scanned sequentially, the edge (i, j) is never scanned again after updating $CurrentEdge(t)$. So, the running time over all iterations in which $CurrentEdge(t)$ is changed is $O(m)$. We conclude that the total running time of Algorithm 5.2 is $O(m + nK)$. □

The original motivation for this paper was the case that $K = 2$, in which case the algorithm is particularly efficient. In the next section, we show how to speed up the algorithm in the case that K is permitted to grow with the problem size.

6. A faster algorithm if K is permitted to grow with problem size

We now revise the algorithm to improve its running time in the case that K is not a constant. We let $q = \frac{nK}{m}$. We assume that $q \geqslant 2$; if not, Algorithm 5.2 runs in linear time. To simplify the exposition, we will assume that q is an integer divisor of K, and we let $h = \frac{K}{q}$. Since we are focused on asymptotic analysis, we can make this assumption without loss of generality.

We will show that Algorithm 5.2 can be refined to run in time $O(m \log q)$. In order to achieve this running time, we need to speed up the bottlenecks in Algorithm 5.2 that depend on K. We need to compute r more efficiently, and we need to call UPDATE(t) less frequently. We first address speeding up the computation of r.

In order to speed up the determination of r, we will store the values $f()$ in a collection of h different binary heaps. The first binary heap stores the values $f(j)$ for $j = 1$ to q, the second binary heap stores the values $f(j)$ for $j = q + 1$ to $2q$,

and so on. We denote the heaps as $H_1, H_2, \ldots, H_h$. Finding the element with minimum value in the binary heap H_i takes $O(1)$ steps. The time to insert an element into H_i or delete an element from H_i takes $O(\log q)$ steps. (For more details on binary heaps see [3].)

We find the minimum $f()$ value by first finding the element with minimum value in each of the h heaps and then choosing the best of these h elements. Finding the minimum key in a heap takes $O(1)$ steps; so the time for this implementation of the FINDMIN() operations is $O(h)$ per iteration of the while loop, and $O(hn) = O(m)$ overall. After finding the minimum element in the h different heaps, we delete the element from its heap. This takes $O(n \log q)$ steps over all iterations.

We now address the time spent in UPDATE(). In order to minimize the time required by UPDATE(), we first relax the requirement on *CurrentEdge*. If *CurrentEdge(t)* has both of its endpoints in S, we say that *CurrentEdge(t)* is *invalid*. We permit *CurrentEdge(t)* to be invalid at some intermediate stages of the algorithm. We then modify the FINDMIN() again. If the minimum element in heap H_i is $f(t)$ for some i, and if *CurrentEdge(t)* is invalid, the algorithm then performs UPDATE(), followed by finding the new minimum element in H_i. It iterates in this manner until the minimum element corresponds to a valid edge. In this way, whenever the algorithm calls UPDATE(), it leads to a modification of *CurrentEdge()*. Moreover, whenever the algorithm selects the minimum element among the q different heaps, the minimum element in each of the heaps corresponds to a valid edge. Since every modification of *CurrentEdge()* leads to a change in one of the values in a heap, and since there are at most m modifications of *CurrentEdge()*, the total running time for UPDATE() over all iterations is $O(m \log q)$.

We summarize the previous discussion with Theorem 6.1.

Theorem 6.1. *The binary heap implementation of Dijkstra's algorithm with $O(\frac{K}{q})$ binary heaps of size $O(q)$ with $q = \frac{nK}{m}$ determines the shortest path from vertex s to all other vertices in $O(m \log q)$ time.*

7. Empirical results

7.1. Experimental setup

We evaluate performance of our algorithms on several graph families. Some of the generators and graph instances are part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [5]:

- *Random graphs*: We generate graphs according to the Erdos–Renyi random graph model, and ensure that the graph is connected. The generator may produce parallel edges as well as self-loops. The ratio of the number of edges to the number of vertices can be varied, and we experiment with both dense and sparse graphs. Random graphs have a low diameter and a Gaussian degree distribution.
- *Mesh graphs*: This synthetic generator produces two-dimensional meshes with grid dimensions x and y. We generate *Long* ($x = \frac{n}{16}$, $y = 16$) and *Square* grids ($x = y = \sqrt{n}$) in this study. The diameter of these graphs is significantly higher than random graphs and all the vertices have a constant degree.
- *Small-world graphs*: We use the R-MAT graph model for real-world networks [4] to generate graphs with small-world characteristics. These graphs have a low diameter and an unbalanced degree distribution.

The edge weights are chosen from a fixed set of distinct random integers, as our new algorithm is designed for networks with small K.

We compare the execution time of our algorithm with the reference SSSPP solver used in the 9th DIMACS Shortest Paths Challenge (an efficient implementation of Goldberg's algorithm [11,13], which has expected-case linear running time, and highly optimized for integer edge weights) and the baseline Breadth-First Search (BFS) on every graph family. The BFS running time is a natural lower bound for SSSPP implementations. It is also reasonable to directly compare the execution times of DIMACS reference solver code and our implementation: both use a similar adjacency array representation for the graph, are written in C/C++, and compiled and run in identical experimental settings. Note that our implementation can process graphs with real as well as integer weights, but is only efficient for networks with a few distinct edges. We use only integer weights in this study for comparison with the DIMACS solver.

Our test platform for performance results is a 2.8 GHz 32-bit Intel Xeon machine with 4 GB memory, 512 KB cache and running RedHat Enterprise Linux 4 (linux kernel 2.6.9). We compare the sequential performance of our implementation with the DIMACS reference solver [5]. Both the codes are compiled with the Intel C compiler (icc) version 9.0, and the optimization flag -O3. We report the average execution time of five independent runs for each experiment.

7.2. Results and analysis

We conduct an extensive study to empirically evaluate the performance dependence on the graph topology, the problem size, the value of K, and the edge weight distribution. We report the execution time of Breadth-First Search on all the graph instances we studied in Tables 1 and 2. The figures plot the execution times of the shortest path implementations normalized to the BFS time. Thus a ratio of 1.0 is the best we can achieve, and smaller values are desirable.

Table 1
Breadth-First Search execution time (in milliseconds) for various graph families on the test sequential platform.

Problem instance		BFS time (milliseconds)		
ID	Graph size	Long mesh	Square mesh	Random
1	100K vertices, 400K edges	50	55	95
2	500K vertices, 2M edges	290	350	540
3	1M vertices, 4M edges	660	870	1180
4	5M vertices, 20M edges	4160	6400	8390
5	10M vertices, 40M edges	8590	13 500	17 980

Table 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.

Problem instance		BFS time (milliseconds)
1	Sparse random, 2M vertices, 8M edges, $C = 10\,000$	6430
2	Dense random, 100K vertices, 100M edges, $C = 100$	150
3	Long mesh, 2M vertices, 8M edges, $C = 100$	3260
4	Square mesh, 2M vertices, 8M edges, $C = 100$	4900
5	Small-world graph, 2M vertices, 8M edges, $C = 10\,000$	5440
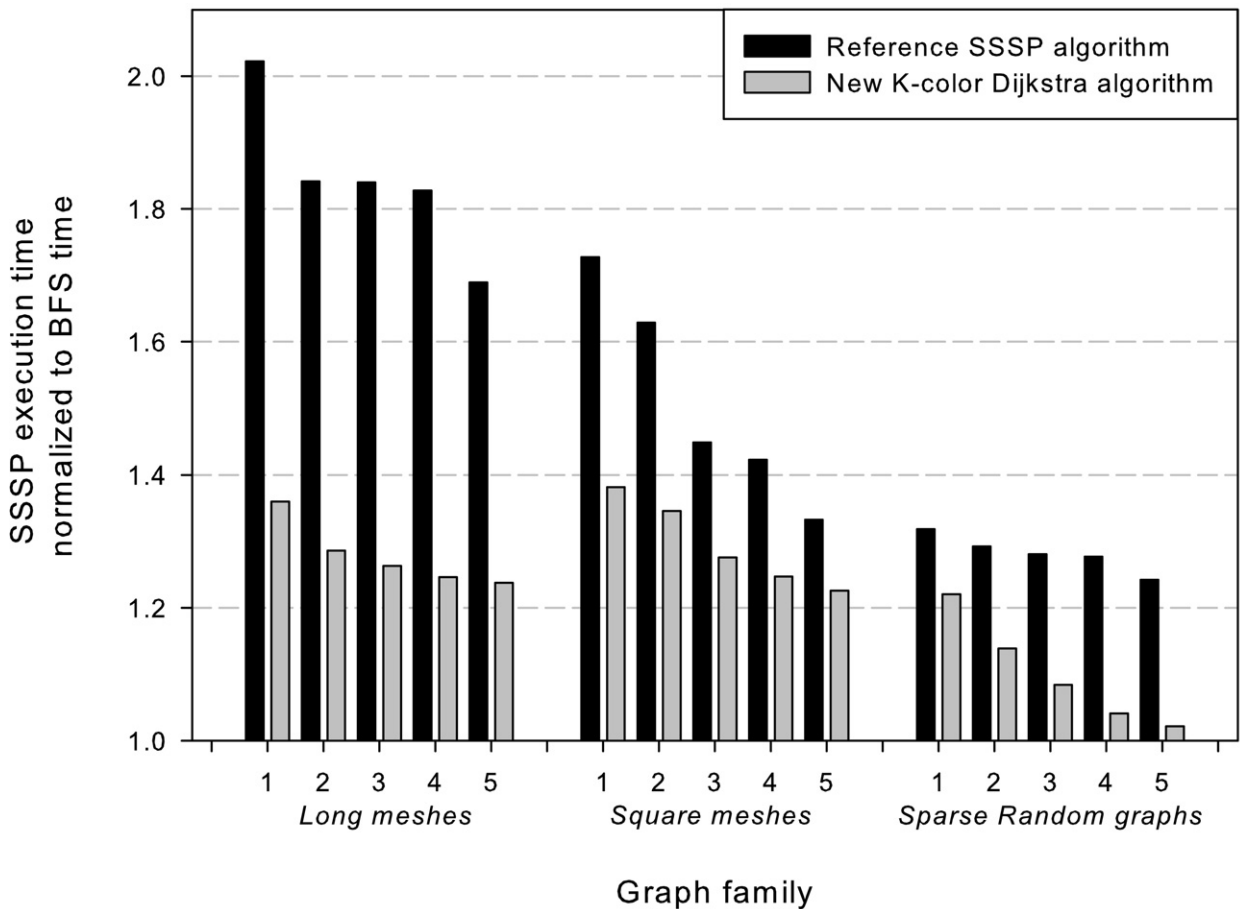
Fig. 1. Performance of our shortest implementation and the reference solver for three graph families, as the problem size is varied. Graph 1 corresponds to the smallest network in our study, and 5 is the largest.

We first study the dependence of execution time on the problem size. We vary the size up to two orders of magnitude for three different graph families and compute the average SSSP execution time for the reference code and our implementation. Fig. 1 plots these normalized values for the different graph families. The problem sizes are listed in Table 1. The value of K is set to 2 in this case, and the ratio of the largest to the smallest edge weight in the graph (denoted by C) is set to 100. Also,
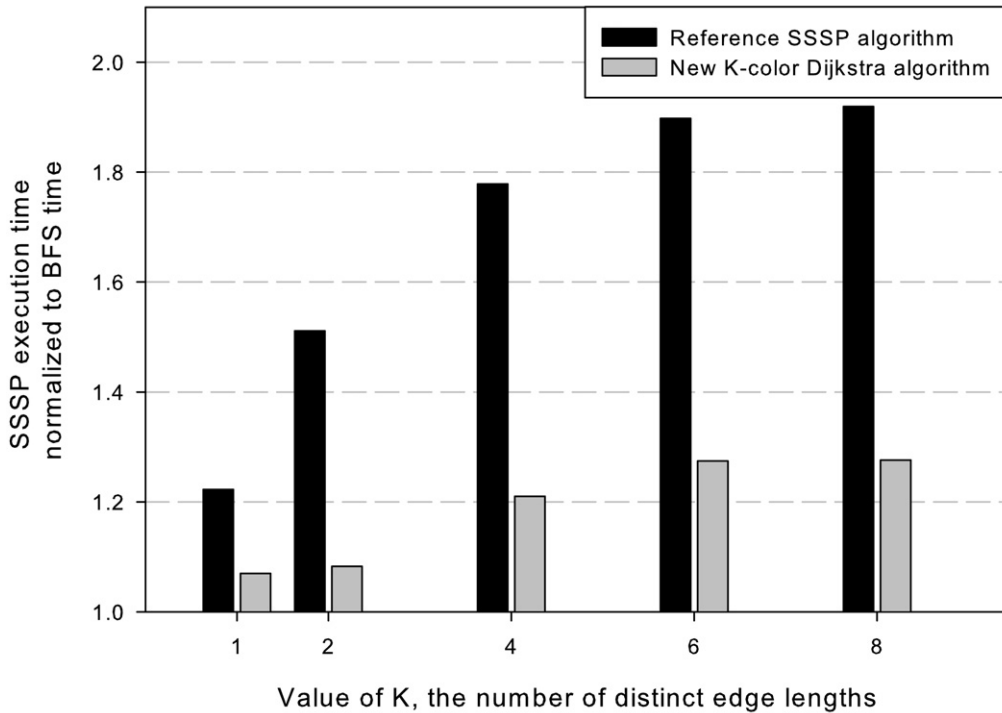
Fig. 2. Normalized SSSPP performance for a sparse random graph (4 million vertices, 16 million edges) as the value of K is varied.
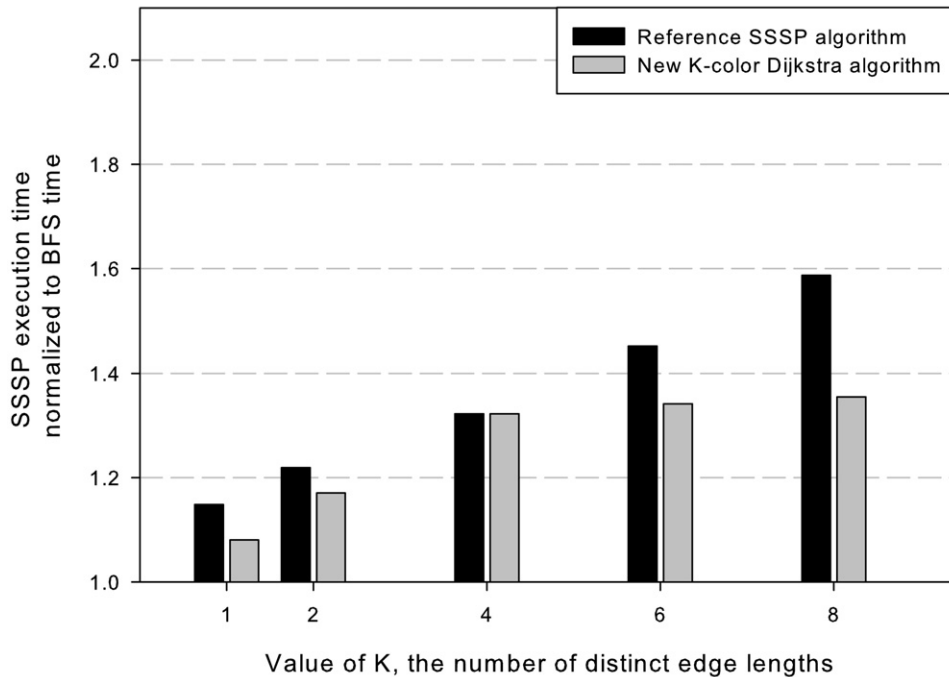

Fig. 3. Normalized SSSPP performance for a small-world graph (4 million vertices, 16 million edges) as the value of K is varied.

note that the ratio m/n is 4 in all the cases. In Fig. 1, we observe that our new implementation outperforms the reference solver for all graph families. Furthermore, the performance ratio is less than 2 in most cases, which is quite significant. On closer inspection, we observe that the performance improvements for long and square mesh graphs are comparatively higher than the random graphs. We attribute this to the fact that we do not maintain a priority queue data structure. Thus we avoid the priority queue overhead involved in evaluating long paths in mesh networks, such as frequent updates to the
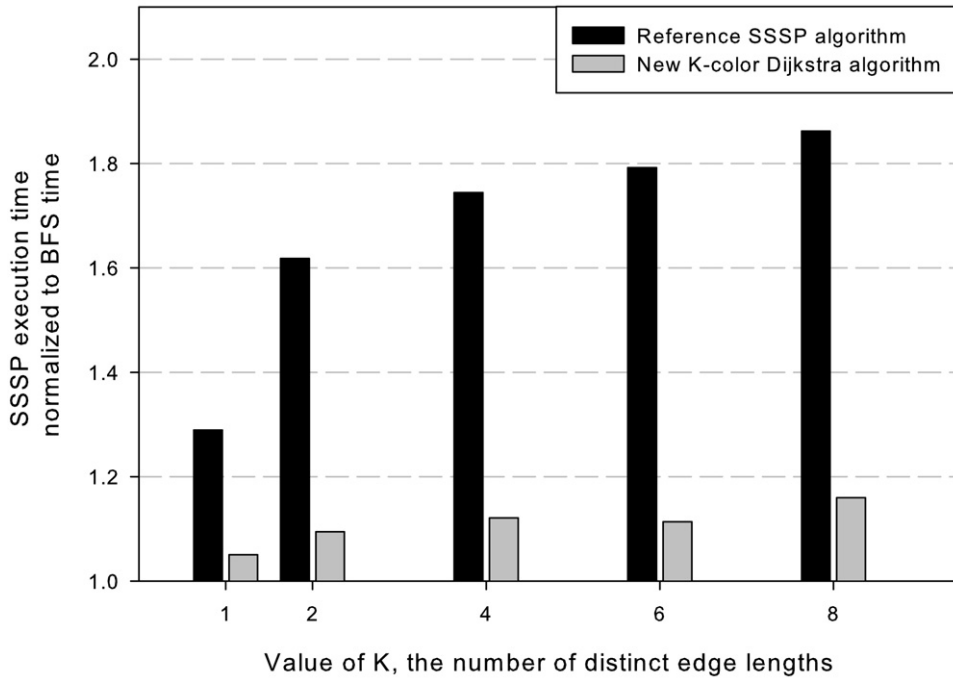
Fig. 4. Normalized SSSPP performance for a long mesh (4 million vertices, 16 million edges) as the value of K is varied.
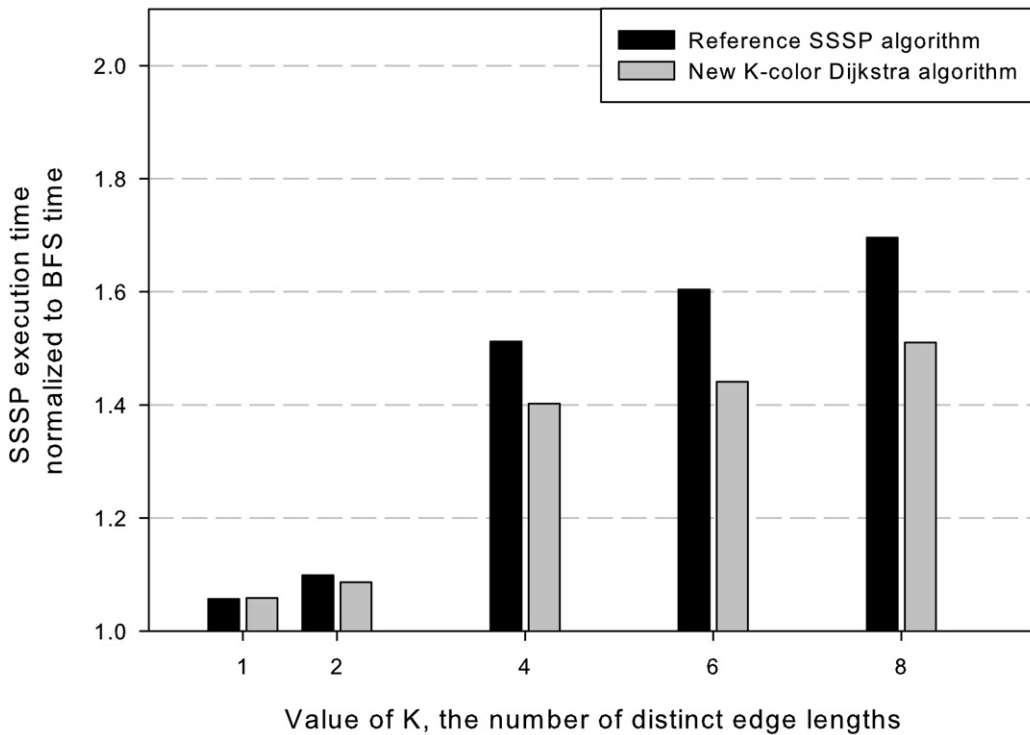

Fig. 5. Normalized SSSPP performance for a square mesh (4 million vertices, 16 million edges) as the value of K is varied.

distance values. We also observe that the performance is better for larger graph instances compared to smaller ones. Also, the performance ratios for sparse random networks (1.02–1.22) are very impressive for the problem instances we studied.

We next study the performance of the algorithm on each graph family as the value of K, the number of distinct edge weights is varied. We vary the value of K from 1 to 8 in each case, and plot the execution time. Fig. 2 plots the normalized
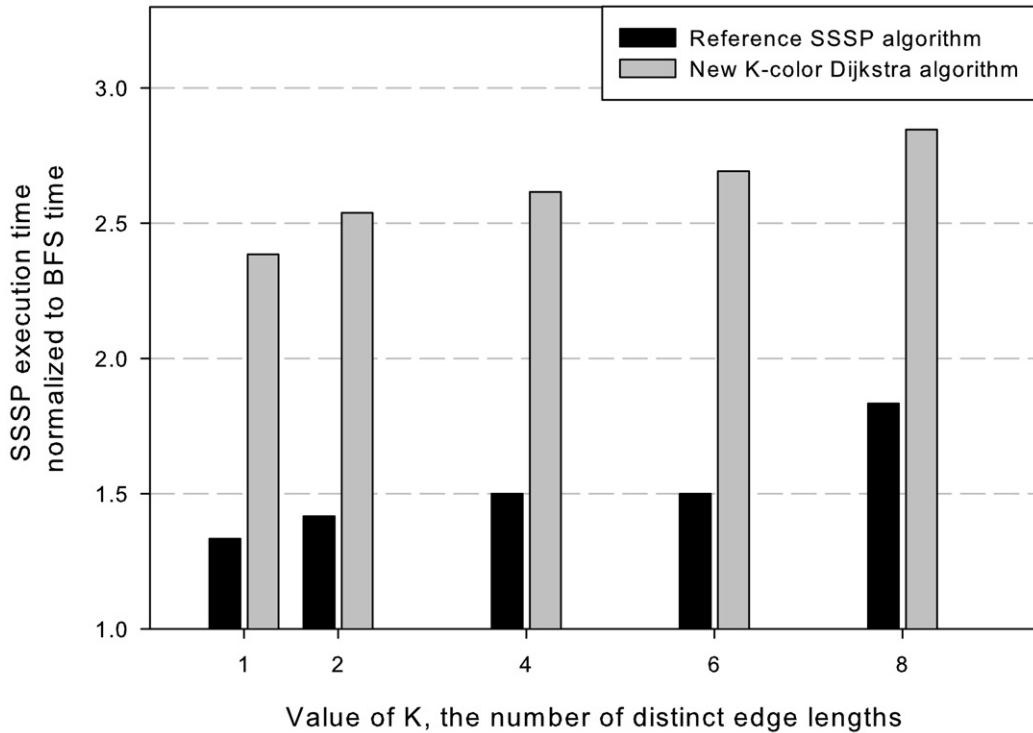
Fig. 6. Normalized SSSPP performance for a dense random graph (100K vertices, 10 million edges) as the value of K is varied.

execution time for a sparse random graph of 4 million vertices and 16 million edges. We observe that our implementation is significantly faster than the reference solver (up to 70% faster for $K = 8$), and also scales slower than the reference solver with increase in K. The SSSP running time for random graphs is comparatively lower than the execution time for other graph families.

Fig. 3 plots the normalized execution time for synthetic small-world graphs. These are low diameter graphs similar to sparse random networks, but also have topological properties that are observed in large-scale social and biological networks. In this case, we observe that the reference solver and our algorithm perform very similarly.

For long meshes (Fig. 4), we observe significant performance gains over the reference solver. Also, the execution time of our algorithm scales much slower than the reference solver as K is increased. For square meshes (Fig. 5) also, we find that our implementation dominates for all values of K.

Fig. 6 plots the normalized execution time for a dense random graph of 100K vertices and 100 million edges as the value of K is varied. We observe that the reference solver is faster than our algorithm in this case, but the speedup falls as the value of K increases. We attribute this to the fast execution time of BFS (as this is a comparatively small graph instance). The overhead in executing our algorithm is high in this case, and it does not perform as well as the reference solver for a dense graph of this size.

The execution times of BFS for all these graph instances are listed in Table 2. The ratio of the highest to the least edge weight in the graph (C) is also listed, as the worst case running time of the reference solver depends on this value. The performance of our algorithm, however, is independent of C.

8. Conclusions

The SSSPP is a fundamental problem within computer science and operations research communities. In this paper, we studied the SSSPP when the number K of distinct edge lengths is small. Our algorithm runs in linear time when the number of distinct edge lengths is smaller than the density of the graph.

Acknowledgements

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms and Applications, Prentice-Hall, 1993.
[2] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan, Faster algorithms for the shortest path problem, J. ACM 37 (2) (April 1990) 213–223.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2001.
[4] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A recursive model for graph mining, in: Proc. 4th SIAM Intl. Conf. on Data Mining, Florida, USA, April 2004.
[5] C. Demetrescu, A.V. Goldberg, D. Johnson, 9th DIMACS implementation challenge – Shortest Paths, http://www.dis.uniroma1.it/challenge9/, 2005.
[6] J.R. Driscoll, H.N. Gabow, R. Shrairman, R.E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, Commun. ACM 31 (11) (1988) 1343–1354.
[7] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[8] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (3) (1987) 596–615.
[9] M.L. Fredman, D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, J. Comput. Syst. Sci. 48 (3) (1994) 533–551.
[10] A.V. Goldberg, Network optimization library, http://www.avglab.com/andrew/soft.html.
[11] A.V. Goldberg, A simple shortest path algorithm with linear average time, in: ESA, 2001, pp. 230–241.
[12] A.V. Goldberg, A simple shortest path algorithm with linear average time, in: 9th Ann. European Symp. on Algorithms (ESA 2001), Aachen, Germany, in: Lecture Notes in Computer Science, vol. 2161, Springer, 2001, pp. 230–241.
[13] U. Meyer, Single-source shortest-paths on arbitrary directed graphs in linear average-case time, in: Proceedings of the Twelfth Annual ACM–SIAM Symposium on Discrete Algorithms (SODA-01), New York, January 7–9, 2001, ACM Press, 2001, pp. 797–806.
[14] M. Thorup, Undirected single source shortest path in linear time, in: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS-97), Los Alamitos, October 20–22, 1997, IEEE Computer Society Press, 1997, pp. 12–21.