Chapter 57
Shortest Paths Algorithms: Theory and Experimental Evaluation

Boris V. Cherkassky* Andrew V. Goldberg† Tomasz Radzik‡

Abstract

We conduct an extensive computational study of shortest paths algorithms, including some very recent algorithms. We also suggest new algorithms motivated by the experimental results and prove interesting theoretical results suggested by the experimental data. Our computational study is based on several natural problem classes which identify strengths and weaknesses of various algorithms. These problem classes and algorithm implementations form an environment for testing the performance of shortest paths algorithms. The interaction between the experimental evaluation of algorithm behavior and the theoretical analysis of algorithm performance plays an important role in our research.

1 Introduction

The shortest paths problem is one of the most fundamental network optimization problems. This problem comes up in practice and arises as a subproblem in many network optimization algorithms. Algorithms for this problem have been studied for a long time. See *e.g.* [2, 4, 5, 6, 17, 18, 20]. However, advances in the theory of shortest paths algorithms are still being made. See *e.g.* [1, 8, 9, 13]. A good description of the classical algorithms and their implementations appears in [10].

On a network with negative-length arcs, the best currently known time bound of $O(nm)$ is achieved by the Bellman-Ford-Moore algorithm [2, 6, 18]. (Here n and m denote the number of nodes and arcs in the network, respectively.) With the additional assumption that arc lengths are integers bounded below by $-N \leq -2$, the $O(\sqrt{n}m \log N)$ bound [13] improves the Bellman-Ford-Moore bound unless N is very large. If the arc lengths are nonnegative, implementations of Dijkstra's algorithm achieve better bounds. An implementation of [7] runs in $O(m + n \log n)$ time. An improved time bound of $O(m + n \log n / \log \log n)$ [8] can be obtained in a random access machine computation model that allows certain word operations. Under the assumption that arc lengths are integers in the interval $[0, \ldots, C]$, $C \geq 2$, the implementation of [1] runs in $O(m + n\sqrt{\log C})$ time.

In this paper we study practical performance of several shortest paths algorithms, including established methods [2, 5, 6, 11, 17, 18, 19, 20], recently proposed algorithms [1, 14], and new algorithms. The development of the new algorithms was based on the experimental feedback. We give theoretical explanation of the observed behavior of the algorithms and prove complexity bounds on the new algorithms.

We also prove an interesting result suggested by the experimental data. This result, summarized in Theorem 11.1, shows that some algorithms, for example the Bellman-Ford-Moore algorithm, are *potential-invariant*, *i.e.*, behave in exactly the same way on two networks one of which is obtained from the other one by replacing the lengths by the reduced costs with respect of a potential function. This result has several interesting implications.

An important part of our work is the development of several natural shortest paths problem generators and their use to create families of problems. Of special interest to us are the families that give insight into the relative algorithm performance, robustness, and dependence of the performance on the network structure and the arc cost distribution.

The collection of algorithms we test is larger than that of any previous study we are aware of, and the set of test problems is much richer. We show that the algorithm performance varies significantly more than previously believed and that some algorithms previously considered robust may fail dramatically. For example, we exhibit a family of problems that are hard for all established algorithms, although a recent algorithm of [14] solves these problems quickly (see Section 7).

Although our research does not produce a single best code for the shortest paths problem, two codes we developed are very competitive in their domains, networks with nonnegative and mixed arc lengths, respectively. One of the codes is a new implementation of Di-

*Central Institute for Economics and Mathematics, Krasikova St. 32, 117418, Moscow, Russia. This work was done while the author was visiting Stanford University Computer Science Department and supported by the NSF Presidential Young investigator and Powell Foundation grants mentioned below.

†Computer Science Department, Stanford University, Stanford, CA 94305. Supported in part by ONR Young Investigator Award N00014-91-J-1855, NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T, DEC, and 3M, a grant from Powell Foundation.

‡Computer Science Department, King's College, London WC2R 2LS, United Kingdom. This work was done while the author was at SORIE, Cornell University, and supported by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS-8920550, and by the Packard Fellowship of Éva Tardos.

jkstra's algorithm using a double bucket data structure. Another code, which is a modification of a recent algorithm of Goldberg and Radzik [14], matches the $O(nm)$ bound of the Bellman-Ford-Moore algorithm and also achieves the optimal $O(m + n)$ time bound on acyclic networks.

Our codes, generators, and generator inputs form a testing environment for shortest paths algorithms. A new code can be compared against the existing ones to determine its relative performance. The environment can be augmented as interesting codes, problem generators, and problem families are developed. Our codes, generators, and generator inputs are available through a mail server.

The shortest paths environment can be used in several ways. Practitioners looking for an efficient code for an application can test our codes on their problems and select one that performs well. The number of codes which need to be compared can be narrowed down using the results of the current paper. Researchers evaluating a new shortest paths code can run the code on the problem families we suggest and compare its performance with the performance of our codes. The environment can also be used in teaching algorithms to demonstrate importance of proper algorithms and data structures.

2 Preliminaries

The input to the single-source shortest paths problem is (G, s, ℓ), where $G = (V, E)$ is a directed graph, $\ell : E \to \mathbf{R}$ is a length function, and $s \in V$ is the source node. The goal is to find shortest paths from s to all other nodes of G or to find a negative length cycle in G. If G has a negative length cycle, we say that the problem is *infeasible*. We assume, without loss of generality, that all nodes are reachable from s in G. We denote $|V|$ by n, $|E|$ by m, and the biggest absolute value of an arc length by C.

A *potential function* is a real-valued function on nodes. Given a potential function d, we define a *reduced cost function* $\ell_d : E \to \mathbf{R}$ by $\ell_d(v, w) = \ell(v, w) + d(v) - d(w)$. We say that an arc a is *admissible* if $\ell_d(a) \leq 0$, and denote the set of admissible arcs by E_d. The *admissible graph* is defined by $G_d = (V, E_d)$. Note that if $d(v) < \infty$ and $d(w) = \infty$, the arc (v, w) is admissible. If $d(v) = d(w) = \infty$, we define $\ell_d(v, w) = \ell(v, w)$.

A *shortest paths tree* of G is a spanning tree rooted at s such that for any $v \in V$, the reversal of the v to s path in the tree is a shortest path from s to v.

3 The Labeling Method

In this section we briefly outline the general *labeling* method for solving the shortest paths problem. (See *e.g.* [3, 10, 22] for more detail.)

For every node v, the method maintains its potential $d(v)$, parent $\pi(v)$, and status $S(v) \in \{\textit{unreached}, \textit{labeled}, \textit{scanned}\}$. The potential of a node v is also called the *distance label* of v. Initially for every node v, $d(v) = \infty$, $\pi(v) =$ nil, and $S(v) =$ unreached. The method starts by setting $d(s) = 0$ and $S(s) =$ labeled, and applies the SCAN operation to labeled nodes until none exists, in which case the method terminates. The SCAN operation applies to a labeled node v. For every $(v, w) \in E$ such that $d(v) + \ell(v, w) < d(w)$, the operation updates $d(w)$ and $\pi(w)$ in a natural way and sets $S(w)$ to *labeled*. At the end of SCAN $S(v)$ is set to *scanned*.

4 Labeling Algorithms

4.1 Bellman-Ford-Moore Algorithm

The Bellman-Ford-Moore algorithm, due to Bellman [2], Ford [6], and Moore [18], maintains the set of labeled nodes in a FIFO queue. The next node to be scanned is removed from the head of the queue; a node that becomes labeled is added to the tail of the queue. Our code BF implements this algorithm.

Performance of the Bellman-Ford-Moore algorithm is as follows.

Theorem 4.1 The algorithm runs in $O(nm)$ time in the worst case.

Although the $O(nm)$ worst case bound is the best bound known for shortest paths algorithms, in practice the Bellman-Ford-Moore algorithm is often slower than other methods. We introduce the following *parent-checking* heuristic: scan a node v only if its parent $\pi(v)$ is not on the queue. The BFP algorithm is a variant of BF that uses this heuristic. One can easily prove the bounds of Theorem 4.1 for this algorithm.

In practice, BFP seems never to make more scans than BF and is never significantly slower. In the vast majority of cases, BFP is faster than BF and the two codes differ by only one "if" statement. We use the BFP code in our experiments below.

4.2 Dijkstra's Algorithm Dijkstra's algorithm [5] selects a labeled node with the minimum potential as the next node to be scanned.

Theorem 4.2 If the length function is nonnegative, Dijkstra's algorithm scans each node exactly once.

Remark. It is easy to show that if negative arc lengths are allowed, the number of scans may be exponential.

We first assume that arc lengths are nonnegative, and treat the other case at the end of this section. Also, when discussing below R-heap and bucket-based implementations of Dijkstra's algorithm, we assume that the length function is integral.

The worst-case complexity of Dijkstra's algorithm on networks with nonnegative arc lengths depends on the way of finding the labeled node with the smallest distance label. A naive implementation that examines all labeled nodes to find the minimum runs in $O(n^2)$ time [5]. The implementation using k-ary heaps (see *e.g.* [3]) runs in $O(m \log n)$ time (for a constant k).

The implementation using Fibonacci heaps [7] runs in $O(m + n \log n)$ time. The implementation using one-level R-heaps [1] runs in $O(m+n \log C)$ time and the one using two-level R-heaps together with Fibonacci heaps, in $O(m+n\sqrt{\log C})$ time. We evaluated implementations that use k-ary heaps with k set to 3 (DIKH), Fibonacci heaps (DIKF), and one-level R-heaps (DIKR). (Note that the R-heap data structure is based on buckets and thus similar to bucket-based implementations discussed below.)

Another way to implement Dijkstra's algorithm is by using the bucket data structure, as proposed by Dial [4]. This implementation maintains an array of buckets, with the i-th bucket containing all nodes v with $d(v) = i$. When distance label of a node changes, the node is removed from a bucket corresponding to its old distance label (if the label was finite) and inserted into the bucket corresponding to the new one. The implementation maintains an index L. Initially, $L = 0$, and L has the property that all buckets $i < L$ are empty. The next node to be scanned is removed from bucket L or, if this bucket is empty, L is incremented. The following theorem follows easily from the observation that bucket deletions and insertions take linear time and at most nC buckets need to be examined by the algorithm.

Theorem 4.3 [4] If the length function is nonnegative, Dial's algorithm runs in $O(m + nC)$ time.

Although the algorithm, as stated, needs nC buckets, an observation that only $C + 1$ consecutive buckets can be occupied at any given time allows the use of $C + 1$ buckets. Our code DIKB implements Dial's algorithm. We maintain nodes in a bucket in the FIFO order. Our implementation places a limit of 300000 on the maximum arc length (which determines the number of buckets).

Next we introduce two simple ways to reduce the memory requirement of Dial's algorithm. In the *overflow bag* implementation, the number of buckets is set to $B < C + 1$. At the i-th stage of the algorithm, the buckets contain nodes with distance labels in the range $[B_i, B_i + B - 1]$. The labeled nodes with distance label $B_i + B$ and above are maintained in a special set (the bag). Initially $i = 0$ and $B_i = 0$. When the value of L reaches $B_i + B$, the value of i is incremented and B_i is set to the minimum distance label of a node in the bag. Then the bag is scanned, nodes with distance labels in the range $[B_i, B_i + B - 1]$ are moved into appropriate buckets, and the next stage begins. The time-memory tradeoff of this implementation is as follows.

Theorem 4.4 If the length function is nonnegative, the overflow bag implementation of Dijkstra's algorithm runs in $O(m + n((C/B) + B))$ time.

Choosing $B = \sqrt{C}$ yields an $O(m + n\sqrt{C})$ time bound. Our code DIKBM implements this algorithm. We set $B = \min(50000, C/3)$.

In the *approximate bucket* implementation, a bucket i contains nodes with distance labels in the range $[i\Delta, (i + 1)\Delta - 1]$, where Δ is a fixed parameter. Nodes in the bucket are processed in the FIFO order. This implementation needs $\lceil C/\Delta \rceil + 1$ buckets. The time-memory tradeoff for this implementation is as follows.

Theorem 4.5 If the length function is nonnegative, the approximate bucket implementation runs in $O(m\Delta + n(\Delta + C/\Delta))$ time.

Our code DIKBA implements this algorithm. We set $\Delta = \lceil C/2^{11} \rceil$.

The ideas of the above two algorithms can be combined to obtain the *double bucket* implementation of Dijkstra's algorithm. This implementation has two kinds of buckets, *high-level* and *low-level*. The number of low-level buckets is Δ. A high-level bucket i contains the set of nodes with distance labels in the range $[i\Delta, (i + 1)\Delta - 1]$ except for the nonempty high-level bucket with the smallest index L. Nodes v with distance label in the range $[L\Delta, (L + 1)\Delta - 1]$ are in the low-level bucket $d(v) - L\Delta$. After all low-level buckets are examined and the nodes in these buckets are scanned, L increases. If the corresponding high-level bucket is not empty, its nodes are moved to the corresponding low-level buckets and the next stage begins.

The number of high-level buckets needed by this implementation is $\lceil (C + 1)/\Delta \rceil$. The running time of the implementation is as follows.

Theorem 4.6 If the length function is nonnegative, the double bucket implementation runs in $O(m + n(\Delta + C/\Delta))$ time.

For the best theoretical bound, the value of Δ should be $\Theta(\sqrt{C})$. Our code DIKBD implements this algorithm. We set Δ to the biggest power of two that is less than $\sqrt{C}$.

The double bucket implementation can be generalized to the *k-level bucket* implementation. The k-level bucket implementation requires $O(kC^{1/k})$ buckets and runs in $O(m + n(k + C^{1/k}))$ time. The details appear in the full version of the paper. Setting $k = \lceil 2 \log C / \log \log C \rceil$ yields an $O(m + n \log C / \log \log C)$ time bound.

We conclude this section with a discussion of implementations of Dijkstra's algorithm when arc lengths can be negative. A "strict" implementation of the algorithm selects a labeled node with the smallest distance label at every step. This is what our code DIKH does.

An alternative is to maintain the value t of the biggest distance label of a node scanned so far, and to select a labeled node with the distance label of t or less if such a node exists and a labeled node with the smallest distance label otherwise. This strategy is more natural for bucket and R-heap implementations and we use it in the corresponding codes. If the nodes eligible for scanning are maintained in FIFO manner, one can show polynomial-time bounds for this variant of Dijkstra's algorithm on networks with arbitrary arc

lengths.

4.3 Incremental Graph Algorithms In this section we describe two algorithms. The first one was developed independently by Pape [20] and Levit [17]. The second algorithm was proposed by Pallottino [19]. He also introduced the incremental graph framework that unified these two algorithms. Our implementations of the above algorithms are called PAPE and TWO_Q, respectively.

Pape-Levit and Pallottino's algorithms maintain the set of labeled nodes as two subsets, S_1 and S_2, the first containing labeled nodes which have been scanned at least once and the second containing those which have never been scanned. The next node to be scanned is selected from S_1 unless S_1 is empty, in which case the node is selected from S_2. We call S_1 the *high-priority set* and S_2 the *low-priority set.*

The Pape-Levit algorithm maintains S_1 as a LIFO stack and S_2 as a FIFO queue. (This algorithm is usually implemented using the *dequeue* data structure, which is a queue that allows insertions at either end. See *e.g.* [10, 19].) This algorithm has exponential worst-case time bound.

Theorem 4.7 [15, 21] The Pape-Levit algorithm runs in $\Theta(n2^n)$ time in the worst case.

Pallottino's algorithm maintains S_1 and S_2 using FIFO queues, Q_1 and Q_2.

Theorem 4.8 [19] Pallottino's algorithm runs in $O(n^2m)$ time in the worst case.

4.4 The Threshold Algorithm Glover et. al. [11] suggest the following method, which combines the ideas lying behind the Bellman-Ford-Moore, Dijkstra's, and incremental graph algorithms. (See also [10, 12].) The method partitions the set of labeled nodes into two subsets, NOW and NEXT, which are maintained as FIFO queues. At the beginning of each iteration of the algorithm, NOW is empty. The method also maintains a threshold parameter t which is set to a weighted average of the minimum and average distance labels of the nodes in NEXT. During an iteration, the algorithm transfers nodes v with $d(v) \leq t$ from NEXT to NOW and scans nodes in NOW. Nodes that become labeled during the iteration are added to NEXT. The algorithm terminates when NEXT is empty at the end of an iteration. Our code THRESH implements the threshold algorithm suggested in [11] with parameter values MINWT = 45 and WTCNG = 25.

The running time of THRESH is as follows.

Theorem 4.9 [12] If the length function is nonnegative, THRESH runs in $O(nm)$ time.

4.5 The Topological Ordering Algorithms Suppose nodes v and w are labeled and there is a path from v to w in the admissible graph containing a negative reduced cost arc. Then it is better to scan v before w, since we know that $d(w)$ is greater than the true distance from s to w. A recent algorithm of Goldberg and Radzik [14] is based on this idea. To simplify the algorithm description, we first assume that G has no cycles of length zero or less, and therefore for any d, the admissible graph G_d is acyclic.

The Goldberg-Radzik algorithm maintains the set of labeled nodes in two sets, A and B. Each labeled node is in exactly one set. Initially $A = \emptyset$ and $B = \{s\}$. At the beginning of each *pass*, the algorithm uses the set B to compute the set A of nodes to be scanned during the pass, and resets B to the empty set. A is a linearly ordered set. During the pass, elements are removed according to the ordering of A and scanned. The newly created labeled nodes are added to B. A pass ends when A becomes empty. The algorithm terminates when B is empty at the end of a pass.

The algorithm computes A from B as follows.

1. For every $v \in B$ that has no outgoing arc with negative reduced cost, delete v from B and mark it as scanned.
2. Let A be the set of nodes reachable from B in G_d. Mark all nodes in A as labeled.
3. Apply topological sort to order A so that for every pair of nodes v and w in A such that $(v, w) \in G_d$, v precedes w and therefore v will be scanned before w.

The algorithm achieves the same bound as the Bellman-Ford-Moore algorithm.

Theorem 4.10 [14] The Goldberg-Radzik algorithm runs in $O(nm)$ time.

Now suppose G has cycles of zero or negative length. In this case G_d need not be acyclic. If, however, G_d has a negative length cycle, we can terminate the computation. If G_d has zero length cycles, we can contract such cycles and continue the computation. This can be easily done while maintaining the $O(nm)$ time bound. (See *e.g.* [13].)

Our code GOR is an implementation of the Goldberg-Radzik algorithm with one simplification. The implementation uses depth-first search to compute topological ordering of the admissible graph (see *e.g.* [3]). Instead of contracting zero length cycles, we simply ignore the back arcs discovered during the depth-first search. The resulting topological order is in the admissible graph minus the ignored arcs. This change does not affect the algorithm correctness or the above running time bound.

We also implement the following modification, GOR1, of GOR. Recall that we use depth-first search to compute the topological ordering. When an arc (v, w) is examined by the depth-first search, the arc is first scanned in the shortest-path sense, *i.e.*, if $d(v) + \ell(v, w) < d(w)$, $d(w)$ is set to $d(v) + \ell(v, w)$ and $\pi(w)$ is set to v. (Note that this changes the admissible graph.) The following theorem gives a theoretical justification for this change.

Theorem 4.11 GOR1 runs in $O(nm)$ time. On an acyclic network, GOR1 terminates in one pass and therefore runs in $O(m+n)$ time.

Remark. When counting the number of scans done by GOR and GOR1, we count both the shortest paths SCAN operations and processing of nodes done by the depth-first searches.

5 Experimental Setup

Our experiments were conducted on SUN Sparc-10 workstation model 41 with a 40MHZ processor running SUN Unix version 4.1.3. The workstation had 160 Meg. memory. Our codes were written in C and compiled with the SUN cc compiler version 1.0 using the O4 optimization option.

Our implementations use the adjacency list representation of the input graph. We experimented with several folklore low-level representations of the graph and found that the one described in detail by Gallo and Pallottino [10] is the most efficient. Our implementations of the traditional algorithms (BF, PAPE, TWO_Q, THRESH) are also very similar to those described in [10]. We attempted to make our implementations of different algorithms uniform to make the running time comparisons more meaningful. We also tried to make the implementations efficient.

The codes compared in our main experiments are BFP, GOR, GOR1, DIKH, DIKBD, PAPE, TWO_Q, and THRESH. We do not include all the Dijkstra's algorithm implementations because they often perform very similarly. We chose DIKH because it is the most widely known version of Dijkstra's algorithm and DIKBD because it is the best overall implementation of Dijkstra's algorithm in our tests. We also compare DIKH, DIKBD, DIKR, DIKF, DIKB, DIKBM, and DIKBA on a subset of the problems that shows strengths and weaknesses of these codes.

When tabulating results of our experiments, we give the running time in seconds (in bold) and the number of scan operations per node (below). The running time is the user CPU time and excludes the input and output times. To obtain a data point for a shortest paths code, we make five runs of the code on problems produced with the same generator parameters except for the pseudorandom generator seed. The data we tabulate is the average over the five runs.

We place a 20 minute limit on the user CPU time of each computation on a problem instance. This leaves over 15 minutes for the shortest paths computation (excluding input and output). Since all problems in our tests are solvable in well under a minute by the code that is fastest for this problem, the codes that exceed the limit on a problem are losing to the fastest code by over an order of magnitude.

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
4097	**0.02**	**0.02**	**0.03**	**0.03**	**0.02**	**0.01**	**0.01**	**0.01**
12288	2.74	2.26	4.82	1.00	1.00	1.25	1.25	1.05
16385	**0.21**	**0.08**	**0.21**	**0.17**	**0.10**	**0.03**	**0.04**	**0.05**
49152	5.05	2.29	5.25	1.00	1.00	1.26	1.26	1.05
65537	**1.96**	**0.37**	**1.52**	**0.70**	**0.50**	**0.22**	**0.24**	**0.28**
196608	9.66	2.28	7.41	1.00	1.00	1.27	1.27	1.13
262145	**24.07**	**2.02**	**7.40**	**3.22**	**2.03**	**1.70**	**1.53**	**1.44**
786432	19.68	2.29	8.11	1.00	1.00	1.27	1.27	1.16
1048577	**231.33**	**7.18**	**42.50**	**16.28**	**8.90**	**4.33**	**4.48**	**7.02**
3145728	41.78	2.30	11.25	1.00	1.00	1.27	1.27	1.19

Figure 1: Grid-SSquare family data.

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
4098	**0.03**	**0.04**	**0.04**	**0.07**	**0.08**	**0.75**	**0.24**	**0.03**
16385	4.78	4.51	4.73	1.00	1.00	153.32	38.14	2.24
16386	**0.32**	**0.24**	**0.23**	**0.34**	**0.23**	**8.31**	**2.01**	**0.16**
65537	9.19	4.57	5.19	1.00	1.00	326.03	71.31	2.29
65538	**3.04**	**1.17**	**1.38**	**1.85**	**0.83**	**203.40**	**22.62**	**0.96**
262145	17.43	4.59	6.48	1.00	1.00	1664.52	166.50	2.30
262146	**40.01**	**5.06**	**7.43**	**10.29**	**3.09**		**309.16**	**4.55**
1048577	34.12	4.62	7.53	1.00	1.00		489.18	2.27
1048578	**351.96**	**20.39**	**36.74**	**53.86**	**12.78**			**19.46**
4194305	70.97	4.62	9.46	1.00	1.00			2.26

Figure 2: Grid-SSquare-S family data.

6 Simple SPGRID Problems

First we experiment with rectangular grid networks produced by our SPGRID generator. Nodes of these graphs correspond to points on the plane with integer coordinates $[x,y]$, $1 \le x \le X$, $1 \le y \le Y$. These points are connected "forward" by arcs of the form $([x,y],[x+1,y])$, $1 \le x < X$, $1 \le y \le Y$, "up" by arcs of the form $([x,y],[x,y+1(\mathrm{mod}\,Y)])$, $1 \le x \le X$, $1 \le y \le Y$, and "down" by arcs of the form $([x,y],[x,y-1(\mathrm{mod}\,Y)])$, $1 \le x \le X$, $1 \le y \le Y$. Thus a *layer*, a set of nodes $[x,y]$ with x fixed and $1 \le y \le Y$, is a doubly connected cycle. There is also an additional source node connected to all nodes in the first layer, *i.e.*, the nodes with coordinates $[1,y]$, $1 \le y \le Y$. For the rectangular grid experiments, arc lengths are selected uniformly at random from the interval $[0, 10000]$.

6.1 Square Grids Figure 1 presents results of experiments on Grid-SSquare family of square grids. For this family $X = Y$.

The best performance on this family is achieved by PAPE and TWO_Q. The performance of GOR, DIKBD, and THRESH is also good. These code lose to the best codes by less than a factor of 3. Somewhat slower is DIKH; it loses to the fastest codes by about a factor of four on the largest problem size.

The worst performance on this family is that of BFP. The second-worst code is GOR1. On the largest problem size, it is an order of magnitude slower than the fastest codes but an order of magnitude faster than the slowest code.

When designing or implementing algorithms that use a shortest paths subroutine, it is often convenient to assume that all nodes of the network are reachable from the source. One way to assure this property is to introduce an artificial source and connect it to the original source by a zero length arc and to the other nodes of the graph by very long arcs. This is exactly

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
8193	0.03	0.04	0.05	0.08	0.05	0.02	0.02	0.02
24576	1.42	2.21	3.03	1.00	1.00	1.24	1.24	1.02
16385	0.05	0.09	0.13	0.20	0.10	0.05	0.05	0.07
49152	1.43	2.23	3.02	1.00	1.00	1.24	1.24	1.01
32769	0.19	0.24	0.31	0.50	0.26	0.14	0.13	0.19
98304	1.43	2.22	3.05	1.00	1.00	1.24	1.24	1.02
65537	0.43	0.53	0.72	1.29	0.59	0.30	0.30	0.47
196608	1.44	2.22	3.01	1.00	1.00	1.24	1.24	1.02
131073	1.21	1.25	1.76	3.58	1.45	0.87	0.88	1.23
393216	1.43	2.23	3.03	1.00	1.00	1.24	1.24	1.01
262145	2.97	3.06	4.64	9.76	3.53	2.31	2.40	3.61
786432	1.44	2.25	2.97	1.00	1.00	1.25	1.25	1.01
524289	6.00	6.15	7.31	23.68	7.94	4.50	4.68	8.16
1572864	1.44	2.25	3.04	1.00	1.00	1.25	1.25	1.01

Figure 3: Grid-SWide family data.

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
8193	0.28	0.03	0.11	0.05	0.05	0.02	0.02	0.03
24576	19.84	2.26	7.41	1.00	1.00	1.25	1.25	1.39
16385	1.26	0.07	0.29	0.10	0.10	0.03	0.04	0.07
49152	36.04	2.27	9.09	1.00	1.00	1.26	1.26	1.40
32769	5.07	0.18	0.82	0.23	0.22	0.10	0.10	0.14
98304	70.51	2.27	9.88	1.00	1.00	1.26	1.26	1.46
65537	20.89	0.33	1.78	0.48	0.45	0.22	0.23	0.31
196608	154.44	2.27	10.57	1.00	1.00	1.26	1.26	1.47
131073	120.33	0.67	4.12	0.97	0.92	0.45	0.45	0.62
393216	318.07	2.27	11.65	1.00	1.00	1.26	1.26	1.50
262145	689.46	1.52	9.92	1.93	1.82	0.94	0.96	1.38
786432	666.76	2.27	12.42	1.00	1.00	1.26	1.26	1.50
524289		3.03	18.25	3.85	3.68	1.73	1.82	2.48
1572864		2.27	12.24	1.00	1.00	1.26	1.26	1.49

Figure 4: Grid-SLong family data.

how we obtain the Grid-SSquare-S family from the Grid-SSquare family.

Figure 2 shows the results of the Grid-SSquare-S experiment. The best codes in the first experiment are the worst by a wide margin in the second experiment. In particular, PAPE is the only code that ran over time limit on the second largest problem size. In the second experiment, TWO_Q performs much better than PAPE but much worse than the other codes.

For larger problems, DIKBD is the fastest code in this experiment.

6.2 Wide and Long Grids Next we examine how the performance depends on the shape of the grid. We study two problem families, Grid-SWide and Grid-SLong. The grids in the first family have $X = 16$, *i.e.*, the length of these grids is fixed and the width grows with the problem size. The grids in the second family have $Y = 16$ and their length grows with the problem size.

The wide grids are easy for all algorithms, as one can see in Figure 3. The fastest codes for this problem family are PAPE and TWO_Q, and all other codes except DIKH are within a factor of 2 from the fastest codes. Even the slowest code, DIKH, loses by less than an order of magnitude.

The situation changes on long grids, as can be seen in Figure 4. The most affected code is BFP, which is very good on wide grids but very bad on long grids, where it is the slowest code by a wide margin. The performance of DIKH is also affected significantly; its performance improves, especially on big problems. Other codes are less affected: their running times change by less than a factor of 4.

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
8193	13.68	0.54	0.33	0.21	0.12	348.05	34.88	2.07
63808	390.13	16.90	11.66	1.00	1.00	14988.77	1108.89	57.55
16385	67.46	1.22	0.81	0.42	0.26	589.40	68.66	5.83
129344	799.87	17.98	12.52	1.00	1.00	12694.03	1145.92	79.25
32769	309.98	2.39	1.88	0.85	0.52		149.19	25.73
260416	1612.36	17.87	13.07	1.00	1.00		1190.10	178.42
65537		4.71	4.20	1.71	1.05		295.65	298.97
522560		17.84	14.01	1.00	1.00		1190.76	803.15
131073		9.54	9.21	3.48	2.11		584.69	
1046848		17.98	14.73	1.00	1.00		1199.97	
262145		18.82	19.25	6.86	4.23			
2095424		17.82	15.12	1.00	1.00			

Figure 5: Grid-PHard family data.

n/m	BFP	GOR	GOR1	DIKBD	PAPE	TWO_Q	THRESH
8193	12.59	0.55	0.33	38.56	359.73	34.88	29.26
63808	392.09	17.88	11.66	876.92	16092.24	1143.77	912.61
16385	68.86	1.36	0.82	186.71	633.44	71.75	161.48
129344	803.31	19.89	12.52	1850.02	13846.24	1187.05	1854.19
32769	317.84	2.99	1.90	888.04		153.64	741.89
260416	1619.02	20.66	13.07	3844.69		1231.74	3859.87
65537		6.45	4.28			305.02	
522560		21.30	14.01			1231.16	
131073		14.05	9.28			609.94	
1046848		22.58	14.73			1239.82	
262145		29.38	19.34				
2095424		23.47	15.12				

Figure 6: Grid-NHard family data. DIKH exceeded the time limit on all problems.

7 Harder SPGRID Problems

The SPGRID generator can also produce more complicated networks. These networks consist of layers and the source connected to the nodes of the first layer. Each layer is a simple cycle plus a collection of arcs connecting randomly selected pairs of nodes on the cycle. The length of the arcs inside a layer is small and nonnegative. There are arcs from one layer to the next one, as in simple grids, but in addition, there are generally arcs from lower to higher numbered layers. For the Grid-PHard problems the inter-layer arcs have nonnegative length, and for Grid-NHard problems, nonpositive length. The length of these arcs is selected uniformly at random from a wide range of integers. Additionally, in the Grid-PHard problems the length of an arc from layer x_1 to layer x_2 is multiplied by $(x_2 - x_1)^2$. The Grid-PHard and Grid-NHard networks are significantly more complicated than simple grids.

The computational results on the Grid-PHard family appear in Figure 5. Only four codes, GOR, GOR1, DIKH, and DIKBD, solve all problems in this family within the time limit. The fastest code for this experiment is DIKBD, with DIKH close behind, losing by less than a factor of 2. In this test PAPE has the worst performance. In the set time, it is able to solve problems of the two smallest sizes only, losing to the best code by three orders of magnitude.

Figure 6 gives results of the Grid-NHard experiment. On this problem family, GOR1 and GOR are by far the best codes.

Although the performance of BFP, PAPE, and TWO_Q codes is not exactly the same in this experiment as in the previous one, it is quite similar. Much worse performance is exhibited by THRESH, DIKBD, and DIKH. The latter code is the worst, exceeding the time limit

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
8192	0.32	0.34	0.25	0.15	0.05	0.32	0.35	0.20
32768	12.23	14.58	10.92	1.00	1.00	16.32	15.90	6.31
16384	1.16	1.09	0.91	0.39	0.14	1.22	1.40	0.79
65536	13.45	16.19	11.72	1.00	1.00	20.14	18.83	7.22
32768	3.41	2.99	2.56	0.98	0.40	3.56	3.98	2.40
131072	13.73	16.47	12.00	1.00	1.00	20.08	19.38	7.22
65536	9.37	8.03	7.09	2.47	0.97	10.60	11.39	7.12
262144	15.51	18.21	13.29	1.00	1.00	25.16	23.38	8.56
131072	23.20	19.04	15.91	5.87	2.23	25.45	28.06	17.45
524288	16.75	19.48	14.15	1.00	1.00	27.31	26.07	9.16
262144	51.39	42.74	33.81	13.15	4.79	54.73	61.36	40.43
1048576	17.61	20.76	14.62	1.00	1.00	27.80	26.95	10.02
524288	109.36	89.36	71.58	29.28	10.11	118.04	134.44	86.66
2097152	18.19	21.02	15.02	1.00	1.00	28.91	28.41	10.09
1048576	235.63	194.31	143.54	63.27	21.09	257.31	302.76	186.23
4194304	19.12	22.40	15.18	1.00	1.00	31.23	30.70	10.56

Figure 7: Rand-4 family data.

n/m	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
512	0.09	0.13	0.12	0.02	0.02	0.15	0.16	0.08
65536	5.32	7.91	6.22	1.00	1.00	9.03	8.70	4.26
1024	0.46	0.70	0.59	0.11	0.12	0.76	0.79	0.46
262144	5.10	7.91	6.21	1.00	1.00	9.11	8.88	4.78
2048	1.97	3.07	2.71	0.48	0.49	3.39	3.53	1.94
1048576	4.65	7.23	6.01	1.00	1.00	8.43	8.33	4.45
4096	8.37	13.45	11.41	1.95	2.02	16.18	16.51	9.21
4194304	4.65	7.24	5.87	1.00	1.00	9.12	8.92	4.99

Figure 8: Rand-1:4 family data.

[L, U]	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
[1, 1]	1.50	2.31	6.43	2.71	1.77	1.45	1.49	2.16
	1.00	1.61	4.48	1.00	1.00	1.00	1.00	1.00
[0, 10]	4.12	5.22	7.97	3.88	2.07	3.39	3.50	2.30
	2.67	4.36	5.94	1.00	1.00	2.85	2.84	1.01
[0, 100]	9.03	9.71	10.16	4.66	2.18	8.29	8.99	4.24
	6.15	8.93	8.21	1.00	1.00	8.07	8.05	1.78
[0, 10000]	23.17	19.01	15.67	5.80	2.23	24.99	27.97	17.12
	16.75	19.48	14.15	1.00	1.00	27.31	26.07	9.16
[0, 1000000]	35.09	26.27	11.99	6.23	2.23	41.76	43.64	27.25
	26.77	27.96	12.26	1.00	1.00	47.60	41.89	16.03

Figure 9: Rand-Len family data. All problems have 131072 nodes and 524288 arcs.

P	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
0	23.18	19.04	15.68	5.82	2.24	25.00	27.98	17.17
	16.75	19.48	14.15	1.00	1.00	27.31	26.07	9.16
1000	23.18	18.35	15.67	8.36	2.48	25.00	27.96	15.80
	16.75	18.89	14.15	1.28	1.28	27.31	26.07	7.77
5000	23.18	18.62	15.68	15.96	4.95	25.02	27.97	16.45
	16.75	19.21	14.15	2.99	2.99	27.31	26.07	8.00
10000	23.18	18.83	15.67	26.33	8.57	24.99	27.97	17.45
	16.75	19.42	14.15	5.55	5.55	27.31	26.07	8.40
100000	23.18	19.33	15.66	95.49	33.23	24.99	27.97	44.33
	16.75	19.76	14.15	23.72	22.85	27.31	26.07	20.86
1000000	23.21	19.43	15.68	137.67	44.42	25.05	28.02	56.53
	16.75	19.78	14.15	33.80	30.44	27.31	26.07	26.77
5000000	23.18	19.20	15.65	147.31	46.32	25.01	28.00	58.88
	16.75	19.53	14.15	35.93	31.40	27.31	26.07	27.88

Figure 10: Rand-P family data. All problems have 131072 nodes and 524288 arcs.

on all test problems.

8 Experiments with SPRAND Families

In this section we study performance of the codes on graphs produced by the SPRAND generator. All graphs we consider are constructed by creating a hamiltonian cycle and then adding arcs with distinct random end points. In our experiments we set the length of the arcs on the cycle to 1 and pick the lengths of other arcs uniformly at random from a certain interval. For all problem families except Rand-Len, this interval is $[0, 10000]$.

Note that if we were to pick the cycle arcs lengths in the same ways as the other arc lengths, the resulting graphs would be essentially random. We found, however, that the resulting problems are easy for all the codes. Setting the cycle arc lengths to 1 makes the problems more interesting and the experiments more insightful.

8.1 Sparse and Dense Networks The graphs in Rand-4 family have $m = 4n$. As one can see in Figure 7, the Dijkstra's codes are the best on these problems, with DIKBD clearly the fastest code and DIKH slower by a factor of about 2 for the smaller problems and a factor of about 3 for the bigger problems. Other codes are noticeably slower, with TWO_Q and PAPE being the slowest.

The graphs in Rand-1:4 family have $m = n^2/4$. As one can see in Figure 8, there is little difference in relative performance of the codes, except DIKH improves relative to DIKBD and becomes the fastest code, although DIKBD is only slightly slower.

8.2 Dependency on Arc Lengths Problems in the Rand-Len family are the same except for the interval from which the arc lengths are selected. The arc length is fixed to 1 for the first problem in the family and selected from an interval $[0, U]$ for the other problems. See Figure 9. Note that because the cycle arcs' lengths are set to 1, the structure of the shortest paths tree changes as U increases. For bigger values of U, the cycle arcs are more likely to be in the tree and the tree is likely to be taller.

On the unit length problems, BFP, DIKH, DIKBD, PAPE, TWO_Q, and THRESH make one scan per node. The running times of BFP, PAPE, and TWO_Q are the fastest (and almost the same). The worst code, GOR1, loses by about a factor of 5.

As the length range expands, the algorithms become slower. DIKBD shows very little dependence on the arc length range and is the fastest except for the unit length case. The performance of GOR1 and DIKH is also affected very little. Other codes are significantly affected; their performance decreases by over an order of magnitude for the $[0, 1000000]$ length range (compared to the unit length case).

8.3 Node Potentials Problems in the Rand-P family are the same except the length function ℓ is modified by assigning each node v a potential $p(v)$ chosen uniformly at random from the interval $[0, P]$ and replacing ℓ by ℓ_p. (For $P = 0$, the problems are the same as the 131072 node problems of the Rand-4 family.) While ℓ is nonnegative, ℓ_p can take on negative values. However, for small P, the expected fraction of negative length arcs is small.

Note that BFP, GOR1, PAPE, and TWO_Q make the same number of scans regardless of the potentials. This observation is justified by Theorem 11.1.

n/m	ACC	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
8192	**0.13**	**0.64**	**0.80**	**0.14**	**0.26**	**0.13**	**0.63**	**0.72**	**0.35**
131072	1.00	8.80	12.51	2.00	1.00	1.00	11.87	11.63	5.15
16384	**0.33**	**1.93**	**2.49**	**0.39**	**0.66**	**0.33**	**1.87**	**2.23**	**1.10**
262144	1.00	9.76	14.52	2.00	1.00	1.00	13.92	13.50	5.78
32768	**0.90**	**5.97**	**6.95**	**1.12**	**1.64**	**0.89**	**5.51**	**6.52**	**3.19**
524288	1.00	10.09	14.55	2.00	1.00	1.00	14.33	14.22	5.89
65536	**2.65**	**18.58**	**23.66**	**2.99**	**4.11**	**2.40**	**18.38**	**19.72**	**9.84**
1048576	1.00	11.36	17.23	2.00	1.00	1.00	15.47	15.17	6.60
131072	**6.54**	**42.86**	**54.96**	**7.24**	**9.67**	**5.38**	**46.43**	**48.66**	**23.39**
2097152	1.00	11.44	16.78	2.00	1.00	1.00	16.29	15.87	6.53

Figure 11: Acyc-Pos family data.

n/m	ACC	BFP	GOR	GOR1	DIKBD	PAPE	TWO_Q	THRESH
8192	**0.13**	**24.12**	**0.18**	**0.17**	**54.60**	**185.55**	**154.47**	**66.83**
131072	1.00	466.95	2.00	2.00	964.79	6188.81	4590.70	1453.33
16384	**0.33**	**123.98**	**0.42**	**0.42**	**266.25**		**623.41**	**325.61**
262144	1.00	887.44	2.00	2.00	1841.51		9699.84	2797.25
32768	**0.97**	**618.04**	**1.17**	**1.15**				
524288	1.00	1724.82	2.00	2.00				
65536	**2.87**		**3.41**	**3.28**				
1048576	1.00		2.00	2.00				
131072	**7.44**		**8.65**	**8.51**				
2097152	1.00		2.00	2.00				

Figure 12: Acyc-Neg family data. Within the time limit, DIKII solves only the smallest problems; the average time is 1047.35 seconds and the average number os scans per node is 9037.71.

n/m	DIKH	DIKF	DIKR	DIKB	DIKBM	DIKBA	DIKBD
8193	**0.08**	**0.14**	**0.09**	**0.05**	**0.05**	**0.03**	**0.05**
24576	1.00	1.00	1.00	1.00	1.00	1.00	1.00
16385	**0.20**	**0.35**	**0.22**	**0.08**	**0.11**	**0.07**	**0.10**
49152	1.00	1.00	1.00	1.00	1.00	1.00	1.00
32769	**0.50**	**0.87**	**0.50**	**0.22**	**0.27**	**0.22**	**0.26**
98304	1.00	1.00	1.00	1.00	1.00	1.00	1.00
65537	**1.29**	**2.17**	**1.13**	**0.53**	**0.62**	**0.53**	**0.59**
196608	1.00	1.00	1.00	1.00	1.00	1.00	1.00
131073	**3.58**	**5.41**	**2.73**	**1.32**	**1.52**	**1.30**	**1.45**
393216	1.00	1.00	1.00	1.00	1.00	1.00	1.00
262145	**9.76**	**13.95**	**6.80**	**3.27**	**3.91**	**3.16**	**3.53**
786432	1.00	1.00	1.00	1.00	1.00	1.00	1.00
524289	**23.68**	**33.32**	**15.83**	**7.57**	**9.24**	**7.03**	**7.94**
1572864	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 13: Performance of Dijkstra's implementations on Grid-SWide problems.

9 Experiments with SPACYC Families

In this section we study performance of the codes on acyclic networks. The shortest paths problem on an acyclic graph can be solved in linear time (see *e.g.* [3]) and the experiments of this section include the linear time algorithm for acyclic graphs, ACC.

Experiments with acyclic graphs are interesting for several reasons. Shortest paths problems in acyclic graphs come up in applications, such as PERT network analysis (see *e.g.* [16]). Furthermore, some networks that come up in applications have large acyclic subgraphs (*e.g.* electric networks) and an algorithm that behaves poorly on acyclic networks is likely to behave poorly on networks with large acyclic subgraphs. Acyclic networks are also easy to use in certain experiments because negative length cycles are not a problem for these networks.

The networks used in the experiments of this section are generated as follows. The nodes are numbered from 1 to n, and there is a path of arcs $(i, i+1)$, $1 \leq i < n$. These arcs are called the *path arcs*. Additional arcs are generated by picking two distinct nodes at random and creating an arc from the lower to the higher numbered node. The lengths of the additional arcs are selected uniformly at random from the interval $[L, U]$.

9.1 Positive Arc Length For the Acyc-Pos family, the path arcs' length is set to 1 and the other arc lengths are selected from the interval [0, 10000]. The unit length of the path arcs makes these problems more difficult for some of the codes. Figure 11 shows how the codes perform on this problem family.

The fastest codes for this family are DIKBD and ACC. These codes perform similarly, but the former is a little faster on bigger problems, in spite of the fact that ACC is especially designed for acyclic graphs. These two algorithms make the same number of scans; the additional overhead of ACC is a topological sort of the graph and the additional overhead of DIKBD is in maintaining the bucket data structure. The latter overhead is smaller than the former for large Acyc-Pos problems.

The slowest codes for this family are GOR, PAPE, and TWO_Q.

9.2 Negative Arc Length For the Acyc-Neg family, the path arc length is set to -1 and the other arc lengths are selected uniformly at random from the interval $[-10000, 0]$. Figure 12 shows how the codes perform on this problem family.

In this experiment, ACC and GOR1 perform similarly to the previous experiment, and GOR performs better than in the previous experiment, matching GOR1.

All other codes perform worse by a very wide margin.

10 Experiments With Variations of Dijkstra's Algorithm

The above experiments involve two implementations of Dijkstra's algorithm, the "classical" k-ary heap implementation DIKH and our double bucket implementation DIKBD. In this section we compare these implementations with several other implementations on problem families Grid-SWide, Grid-SLong, Grid-SSquare-S, Grid-PHard, and Rand-Len. The problem families are chosen to emphasize differences in the codes' performance. The additional implementations we evaluate are the R-heap implementation DIKR, the Fibonacci heap implementation DIKF, Dial's implementation DIKB, the overflow bag implementation DIKM, and the approximate bucket implementation DIKBA.

Figure 13 presents data for the Grid-SWide family. Here DIKBA performs best, with DIKB, DIKBD, and DIKBM close behind. Note that DIKBA makes only one scan per node on these problems. The slowest code in this test is DIKF.

Figure 14 presents data for the Grid-SLong family. On this family, DIKH and DIKBD are the fastest codes. DIKB and DIKBM are significantly slower.

n/m	DIKH	DIKF	DIKR	DIKB	DIKBM	DIKBA	DIKBD
8193	0.05	0.09	0.08	0.37	0.33	0.06	0.05
24576	1.00	1.00	1.00	1.00	1.00	1.00	1.00
16385	0.10	0.19	0.17	0.73	0.63	0.13	0.10
49152	1.00	1.00	1.00	1.00	1.00	1.00	1.00
32769	0.23	0.42	0.37	1.47	1.30	0.28	0.22
98304	1.00	1.00	1.00	1.00	1.00	1.00	1.00
65537	0.48	0.90	0.75	2.95	2.62	0.58	0.45
196608	1.00	1.00	1.00	1.00	1.00	1.00	1.00
131073	0.97	1.80	1.49	6.06	5.45	1.17	0.92
393216	1.00	1.00	1.00	1.00	1.00	1.00	1.00
262145	1.93	3.65	2.98	11.99	10.52	2.33	1.82
786432	1.00	1.00	1.00	1.00	1.00	1.00	1.00
524289	3.85	7.48	5.96	23.67	21.03	4.68	3.68
1572864	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 14: Performance of Dijkstra's implementations on Grid-SLong problems.

n/m	DIKH	DIKF	DIKR	DIKBM	DIKBA	DIKBD
4098	0.07	0.07	0.05	0.15	0.03	0.08
16385	1.00	1.00	1.00	1.00	1.82	1.00
16386	0.34	0.37	0.24	0.41	0.13	0.23
65537	1.00	1.00	1.00	1.00	1.75	1.00
65538	1.85	2.00	1.13	1.95	0.65	0.83
262145	1.00	1.00	1.00	1.00	1.72	1.00
262146	10.29	9.51	4.87	17.03	2.92	3.09
1048577	1.00	1.00	1.00	1.00	1.70	1.00
1048578	53.86	45.60	21.62	129.90	12.88	12.78
4194305	1.00	1.00	1.00	1.00	1.70	1.00

Figure 15: Performance of Dijkstra's implementations on Grid-SSquare-S problems.

Figure 15 presents data for the Grid-SSquare-S family. Here DIKBA performs best and DIKBD is somewhat worse on smaller problems but catches up with DIKBA on the larger problems. The code DIKR is somewhat slower; DIKF and DIKH are significantly slower than the fastest codes, and DIKBM is slower than DIKH.

Figure 16 presents data for the Grid-PHard family. Here DIKR performs best, with DIKBD a very close second. Another code that does very well on these problems is DIKBM. The performance of DIKH and DIKF is reasonably good, and these codes perform very similarly. The worst code, DIKBA, loses to the best by about a factor of 3.

Figure 17 presents data for the Rand-Len family. On problems with small lengths, DIKB, DIKBA, and DIKBD are the fastest codes and on problems with big lengths, DIKBM is the fastest. However, the difference among all these codes is small, except that DIKB exceeds its limit on the number of buckets and does not run on the problems with the biggest arc length range. Somewhat slower than the fastest codes is DIKH. The code DIKF is the slowest except on the problem with the biggest arc lengths, where it is the second slowest.

n/m	DIKH	DIKF	DIKR	DIKBM	DIKBA	DIKBD
8193	0.21	0.20	0.13	0.14	0.37	0.12
63808	1.00	1.00	1.00	1.00	6.50	1.00
16385	0.42	0.42	0.25	0.28	0.74	0.26
129344	1.00	1.00	1.00	1.00	6.42	1.00
32769	0.85	0.88	0.52	0.58	1.53	0.52
260416	1.00	1.00	1.00	1.00	6.54	1.00
65537	1.71	1.75	1.04	1.17	3.04	1.05
522560	1.00	1.00	1.00	1.00	6.47	1.00
131073	3.48	3.57	2.08	2.30	6.12	2.11
1040848	1.00	1.00	1.00	1.00	6.51	1.00
262145	6.86	7.13	4.16	4.59	12.21	4.23
2095424	1.00	1.00	1.00	1.00	6.47	1.00

Figure 16: Performance of Dijkstra's implementations on Grid-PHard problems.

$[L, U]$	DIKH	DIKF	DIKR	DIKB	DIKBM	DIKBA	DIKBD
[1, 1]	2.71	4.01	2.16	1.61	2.14	1.62	1.77
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
[0, 10]	3.85	4.34	2.51	1.91	2.14	1.91	2.07
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
[0, 100]	4.66	5.45	2.67	2.18	2.03	2.20	2.18
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
[0, 10000]	5.80	6.07	2.45	2.44	2.07	2.30	2.23
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
[0, 1000000]	6.23	5.35	2.33		1.73	2.27	2.23
	1.00	1.00	1.00		1.00	1.05	1.00

Figure 17: Performance of Dijkstra's implementations on Rand-Len problems. For the largest length interval, DIKB requires too many buckets and does not run.

11 Discussion

Our experimental data motivated an interesting theoretical discovery which we describe next. We say that two instances of the shortest path problem are *equivalent* if the underlying networks, including their representations, are identical and the two length functions, ℓ' and ℓ'', satisfy $\ell'_d = \ell''$ for some potential function d. (If networks are given in the adjacency list representations, identical representations have the corresponding nodes and arcs appearing in the same order.) A labeling shortest paths algorithm is *potential-invariant* if it performs the same sequence of node scans on two equivalent problem instances. Figure 10 shows that GOR, DIKH, DIKBD, and THRESH algorithms are not potential-invariant.

Theorem 11.1 Algorithms BF, BFP, GOR1, PAPE, and TWO_Q are potential-invariant.

Theorem 11.1 is powerful and useful. For example, the theorem shows that no heuristic for computing a "good" initial potential function can improve performance of a potential-invariant algorithm such as BF. Note that any feasible shortest paths problem has an equivalent one with nonnegative arc lengths. If the problem with nonnegative arc lengths is computationally simpler than the general problem, the theorem suggests that a potential-invariant algorithm cannot be superior to all other algorithms on problems with nonnegative arc lengths.

12 Concluding Remarks

Our study does not produce a single best code for all classes of shortest paths problems. We can, however, suggest two algorithms, one for networks with negative arcs and one for networks without negative arcs. These algorithms may not be the best on a particular problem class, but their running time is likely to be of the same order of magnitude as that of the fastest algorithm and often will be much closer.

For problems with nonnegative arc lengths, Dijkstra's algorithm is robust and an appropriate implementation of this algorithm is usually quite competitive. In our tests, the double bucket implementation, DIKBD, is the best overall. This implementation also seems to work reasonably well if the network has a small number

of negative length arcs. For problems with many negative length arcs, GOR1 appears to be a good choice. This code also works well on graphs that have large node-induced acyclic subgraphs.

In practice, problems often have a very specific structure, and algorithms that can take advantage of this structure may perform very well. For example, practical problems are often quite "metric" and incremental graph algorithms may work well on these problems. Our experiments suggest, however, that extra care is needed if one decides to use these algorithms because small changes (such as addition of an artificial source) may drastically decrease performance of these algorithms. Our experiments give strong evidence that TWO_Q is more robust than PAPE and is a safer choice in practice.

The relatively good performance of the R-heap and the double-bucket implementations compared to the k-ary heap and bucket implementations, respectively, show that sophisticated data structures may be worth implementing. On the other hand, the relatively poor performance of the Fibonacci heap implementation compared to the k-ary heap implementation shows that a sophisticated data structure with a better theoretical worst-case bound is not necessarily better in practice.

We implemented a scaling algorithm of [13]. Performance of our implementation was not especially good, but a better implementation may be possible.

We experimented with networks without negative cycles. An interesting question is which algorithms are best at detecting a negative cycle if there is one.

Code Availability

The codes of our implementations and generators, the generator inputs used in our experiments, and a description of our network representation format are available via a mail server. To obtain the codes and the other data, send mail to `ftp-request@theory.stanford.edu` and put `send splib.tar` as the subject line. The reply will contain a uuencoded tar file with the codes, generator inputs, and documentation.

Acknowledgments

We would like to thank Robert Kennedy for comments on a draft of this paper.

References

[1] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. Technical Report CS-TR-154-88, Department of Computer Science, Princeton University, 1988.

[2] R. E. Bellman. On a Routing Problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[4] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.

[5] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numer. Math.*, 1:269–271, 1959.

[6] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[7] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[8] M. L. Fredman and D. E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. In *Proc. 31st IEEE Annual Symposium on Foundations of Computer Science*, pages 719–725, 1990.

[9] H. N. Gabow and R. E. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, pages 1013–1036, 1989.

[10] G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.

[11] F. Glover, R. Glover, and D. Klingman. Computational Study of an Improved Shortest Path Algorithm. *Networks*, 14:25–37, 1984.

[12] F. Glover, D. Klingman, and N. Phillips. A New Polynomially Bounded Shortest Paths Algorithm. *Oper. Res.*, 33:65–73, 1985.

[13] A. V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 222–231, 1993.

[14] A. V. Goldberg and T. Radzik. A Heuristic Improvement of the Bellman-Ford Algorithm. *Applied Math. Let.*, 6:3–6, 1993.

[15] A. Kershenbaum. A Note on Finding Shortest Paths Trees. *Networks*, 11:399, 1981.

[16] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart, and Winston, New York, NY., 1976.

[17] B. Ju. Levit and B. N. Livshits. *Neleneinye Setevye Transportnye Zadachi.* Transport, Moscow, 1972. In Russian.

[18] E. F. Moore. The Shortest Path Through a Maze. In *Proc. of the Int. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[19] S. Pallottino. Shortest-Path Methods: Complexity, Interrelations and New Propositions. *Networks*, 14:257–267, 1984.

[20] U. Pape. Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem. *Math. Prog.*, 7:212–222, 1974.

[21] D. Shier and C. Witzgall. Properties of Labeling Mehtods for Determinimg Shortest Paths Trees. *J. Res. Natl. Bur. Stand.*, 86:317–330, 1981.

[22] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.