Shortest Path Feasibility Algorithms: An Experimental Evaluation

Boris V. Cherkassky[1]   Loukas Georgiadis[2]   Andrew V. Goldberg[3]
Robert E. Tarjan[4]   Renato F. Werneck[3]

Abstract

This is an experimental study of algorithms for the shortest path feasibility problem: Given a directed weighted graph, find a negative cycle or present a short proof that none exists. We study previously known and new algorithms. Our testbed is more extensive than those previously used, including both static and incremental problems, as well as worst-case instances. We show that, while no single algorithm dominates, a small subset (including a new algorithm) has very robust performance in practice. Our work advances state of the art in the area.

1 Introduction

The *shortest path feasibility problem* (FP) is to find a negative-length cycle in a given directed, weighted graph, or to present a proof (a set of feasible potentials) that no such cycle exists. This is closely related to the problem of finding shortest path distances in a network. A solution to the shortest path problem is also a solution to FP. Furthermore, given a feasible set of potentials, one can solve the shortest path problem in almost linear time with Dijkstra's algorithm [9]. These are classical network optimization problems with many applications, such as program verification [25] and real-time scheduling [5].

The classical Bellman–Ford–Moore (BFM) algorithm [1, 11, 18] achieves the best known strongly polynomial time bound for FP: $O(nm)$ (where n and m denote the number of vertices and arcs in the graph, respectively). With the additional assumption that arc lengths are integers bounded below by $-N \leq -2$, the $O(\sqrt{n}m \log N)$ bound of Goldberg [14] is an improve-

ment, unless N is very large. The BFM bound can also be improved in the expected sense for many input distributions; see *e.g.* [17].

Previous studies of shortest path algorithms include [2, 3, 8, 13, 19, 20, 21, 26]. The earliest experiments concluded that BFM performs poorly in practice, and favored alternatives such as incremental graph algorithms [20, 21] and the threshold algorithm [8]. These methods have relatively bad worst-case bounds, however, and are not robust in practice. More recently, Cherkassky and Goldberg [2] showed that two $O(nm)$ algorithms, BFCT [23] and GOR [15], are much more robust. These methods, which have been studied further in [19, 26], serve as the basis of our experimental comparison.

Our first contribution is an algorithm design framework that generalizes BFM and is a natural basis for new algorithms.

As our second contribution, we implement several new algorithms and reimplement existing ones, in one case correcting a heuristic used in previous implementations. Our experiments show that a new algorithm, RD (Robust Dijkstra), is not only very robust, but also significantly outperforms previous ones on some problem classes.

Finally, we propose a new set of benchmark instances that is more extensive than existing ones. It includes natural problems, bad-case instances for all algorithms, and experiments modeling incremental scenarios where one solves a sequence of feasibility problems, each a perturbation of the previous one.

This paper is organized as follows. Section 2 presents some definitions and notation. Section 3 reviews the scanning method and gives a general framework for designing $O(n)$-pass algorithms. The negative cycle detection techniques used by the algorithms we implemented are discussed in Section 4. Section 5 reviews existing algorithms used in our study, including their improved implementations, and introduces some new algorithms. Section 6 describes families of instances that elicit the worst-case behavior of these methods. Section 7 describes our experimental setup and presents an empirical comparison of the various algorithms studied.

---

[1]Central Economics and Mathematics Institute of the Russian Academy of Sciences. E-mail: bcherkassky@gmail.com. Part of this work has been done while this author was visiting Microsoft Research Silicon Valley.

[2]Hewlett-Packard Laboratories, Palo Alto, CA, 94304. E-mail: loukas.georgiadis@hp.com.

[3]Microsoft Research Silicon Valley, Mountain View, CA 94043. E-mail: {goldberg,renatow}@microsoft.com.

[4]Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08540 and Hewlett-Packard Laboratories, Palo Alto, CA, 94304. E-mail: ret@cs.princeton.edu.

Finally, Section 8 presents concluding remarks.

2 Definitions and Notation

The inputs to FP are a directed graph $G = (V, E)$ and a length function $\ell : E \to \mathbf{R}$. A *potential function* maps vertices to values in $\mathbf{R} \cup \{\infty\}$. We refer to these values as *potentials*. Given a potential function d, we define the *reduced cost function* $\ell_d : E \to \mathbf{R} \cup \{\infty\}$ by $\ell_d(v, w) = \ell(v, w) + d(v) - d(w)$. (If $d(v) = d(w) = \infty$, we define $\ell_d(v, w) = \ell(v, w)$.) We refer to arcs with negative reduced cost as *negative arcs*. The potential function is *feasible* if there are no negative arcs.

Note that replacing ℓ with ℓ_d may change the length of the shortest path between two distinct vertices, but the path itself (the sequence of arcs) will remain the same. It is also easy to see that this change (a *potential transformation*) preserves the length of every cycle.

The goal of FP is to find a feasible set of potentials or a negative-length cycle (or simply *negative cycle*) in G. A feasible potential function exists if and only if there is no negative cycle.

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 $G_d = (V, E_d)$.

To deal with the fact that the input graph may not be strongly connected and that some algorithms for FP maintain a tentative shortest path tree, we add a *root vertex* s to V and connect s to all other vertices with zero-length arcs. Further references to G will thus be to this transformed graph. A *shortest path tree* of G is a spanning tree rooted at s such that for any $v \in V$, the s-to-v path in the tree is a shortest path from s to v in G.

We say that $d(v)$ is *exact* if the distance from s to v in G is equal to $d(v)$, and *inexact* otherwise.

3 Scanning Algorithms

This section briefly outlines the general *labeling method* [10] for solving shortest path and feasibility problems, on which all algorithms studied in this paper are based. (See *e.g.* [24] for more details.) We maintain for every vertex v its potential $d(v)$ and parent $p(v)$ (possibly *null*). Typically, all potentials are initially zero, $p(s)$ is set to *null* and the remaining vertices v have $p(v)$ set to s. At every step, we perform a *labeling operation*: pick an arc (u, v) such that $d(u) < \infty$ and $\ell_d(u, v) < 0$ and set $d(v) = d(u) + \ell(u, v)$ and $p(v) = u$. If G has no negative cycle, eventually there will be no negative arcs, at which point d will be feasible. As Section 4 will show, a simple modification ensures that this method also terminates in the presence of a negative cycle.

The *scanning method* is a variant of the labeling method based on the SCAN operation. The method maintains for each vertex v a *status* $S(v) \in \{\text{unreached}, \text{labeled}, \text{scanned}\}$. Given a labeled vertex v, SCAN(v) examines all arcs (v, w), and if $d(v) + \ell(v, w) < d(w)$ then $d(w)$ is set to $d(v) + \ell(v, w)$, $p(w)$ to v, and $S(w)$ to labeled. A labeled vertex becomes scanned when a SCAN operation is applied to it, and a vertex of any status becomes labeled whenever its potential decreases. Scanning algorithms for FP typically start with all potentials set to zero, which requires the initial set of labeled vertices to contain all vertices with outgoing negative arcs.

3.1 $O(n)$-pass algorithms. As described, the scanning algorithm does not necessarily run in polynomial time. We can, however, define a general class of scanning methods that perform $O(nm)$ labeling operations. This class generalizes BFM and GOR, among other algorithms.

We partition the sequence of vertex scans into *passes* (subsequences of scans) with the following properties: (a) each vertex is scanned at most once during a pass; and (b) each vertex that is already labeled when the pass begins must be scanned during the pass. Note that a vertex that becomes labeled during the pass can also be scanned, but it does not need to be.

LEMMA 3.1. *If there is a shortest path from s to v containing k arcs, then after at most k passes $d(v)$ is exact. Thus in the absence of negative cycles, the algorithm terminates after at most $n - 1$ passes.*

The proof is by induction on the number of passes: one shows that after pass i, every vertex whose shortest path from s has i arcs will have exact potential.

BFM can be seen as an implementation of this general algorithm. It maintains labeled vertices in a queue: the vertex to be scanned next is removed from the front, and newly labeled vertices are inserted into the back. Note that every vertex that is currently labeled will be scanned before any vertex that is not, as required by a pass.

4 Negative Cycle Detection

To ensure that the labeling method terminates in the presence of negative cycles, we must use a *cycle detection strategy*. This section discusses some of them.

Let the *parent graph* G_p be the subgraph of G induced by the arcs $(p(v), v)$, for all v such that $p(v) \neq null$. This graph has several useful properties (see *e.g.* [24]). In particular, the arcs in G_p have nonpositive reduced costs and (if G_p is acyclic) form a tree rooted at s. If G_p does have a cycle, it will correspond to a negative cycle in the original graph. Conversely, if G

contains a negative cycle, then G_p will provably contain a cycle after a finite number of labeling operations.

Although a cycle can appear in G_p after one labeling operation and disappear after the next, it can be detected as soon as it appears. Suppose a labeling operation is applied to an arc (u, v) and G_p is acyclic: a cycle will appear in G_p if and only if u is a descendant of v in the current tree. This could be tested by following parent pointers from u, but this takes $\Theta(n)$ worst-case time and does not work well in practice [2].

A more efficient solution is *subtree disassembly*, proposed by Tarjan [23]. When the labeling operation is applied to (u, v), one traverses the entire subtree rooted at v. If u is found, the algorithm terminates with a negative cycle. Otherwise, all vertices visited (except v itself) are removed from the current tree and marked as unreached. They will only be scanned after becoming labeled again. Disassembling a subtree takes linear worst-case time if we maintain a doubly-linked list to represent a preorder traversal of the current tree [16]. Fortunately, this cost can be charged to the work of building the subtree, which makes the amortized cost of each labeling operation constant. In particular, applying subtree disassembly to an $O(n)$-pass algorithm does not change its worst-case complexity. Marking a labeled vertex as unreached may delay its next scan, but this is arguably positive because its potential is known to be inexact. The net effect is usually a decrease in the total number of scans.

A slight variant of this method is *subtree disassembly with updates* [2]: if the potential of v decreases by δ, potentials of all proper descendants w of v can be updated to $d(w) - \delta$. Because the new potentials may be exact, additional bookkeeping is required to make sure these vertices are scanned at least once more.

Here we propose a simpler alternative for the case when arc lengths are integral. Instead of decreasing the potentials by δ, we decrease them by $\delta - 1$, to $d'(w) = d(w) - \delta + 1$. Because the new potentials are not exact, the updated vertices are guaranteed to be scanned again. Moreover, the optimization is still effective: after an update, only paths to w with length equal to $d(w) - \delta$ (or shorter) will cause w to become labeled.

An alternative to methods based on subtree traversal is *admissible graph search* [14], which uses the fact that if $p(w) = v$, then (v, w) is in G_d. Therefore, if G_p contains a cycle, the admissible graph G_d contains a negative cycle. Since the arcs in G_d have nonpositive reduced cost, a negative cycle can be found in $O(n + m)$ time using an augmented depth-first search. As we will see, admissible graph search is a natural strategy for GOR: since it already performs a depth-first search of

G_d at each iteration, cycle detection has very little overhead.

5 Algorithms

This section describes the algorithms we tested, including both existing and new ones.

5.1 BFCT. This is the BFM algorithm with subtree disassembly, including updates. Initially, all vertices are labeled and have zero potential. Labeled vertices are maintained in a FIFO queue. A vertex to be scanned is removed from the front of the queue, and newly labeled vertices are inserted at the back. When an arc (u, v) is scanned and the potential of v is reduced, we disassemble and update the subtree rooted at v. Any labeled vertices found in the subtree are deleted from the queue, since they are no longer labeled. BFCT runs in $O(nm)$ worst-case time.[1]

5.2 GOR. Proposed by Goldberg and Radzik [15], GOR is an $O(n)$-pass algorithm that works as follows. Define a *source* as a vertex with outgoing negative arcs. Suppose the admissible graph G_d is acyclic. Intuitively, improvements in potential values will propagate along the arcs of this graph. In each pass, GOR sorts in topological order all sources and the vertices reachable from them in G_d, then scans them in this order.

Because G_d is not necessarily acyclic, we compute its strongly connected components (SCCs). If there is an SCC containing a negative arc, it also contains a negative cycle, and GOR terminates. Otherwise, there is a zero-reduced-cost path between any two vertices within each SCC. Therefore, we can make G_d acyclic by contracting each SCC into a single vertex (until the end of the computation). Because of its performance overhead and implementation complexity, however, graph contraction should be avoided in practice.

The implementation of [2] avoids both the contraction and the SCC computation by running a simple depth-first search (DFS). When a back arc is discovered, it checks if the corresponding cycle is negative. If it is, the algorithm terminates; otherwise, it ignores the back arc and resumes the DFS. If no negative cycle is found, the DFS produces a topological ordering of the graph obtained from G_d when the ignored arcs are deleted. After a careful analysis of this procedure, however, we discovered that there are rare situations in which the DFS may fail to find an existing negative cy-

---

[1]In [2], the abbreviation "BFCT" refers to an implementation of the algorithm that does no updates, and a more complicated implementation of updates is considered. Our new implementation of updates has essentially no overhead, making the implementation with no updates obsolete.

cle in G_d, which could cause the algorithm to terminate later or not at all (although the latter never happened in the experiments reported in [2, 26]). Note that the problem is not in the original GOR algorithm, but in the way this particular implementation avoids contractions.

We propose and use a new implementation of GOR that corrects this problem while still avoiding contractions. It runs the SCC algorithm of Tarjan [22], which performs a single DFS while maintaining a second stack (in addition to the one used by the DFS) to store SCC-related information. When an SCC is fully processed, its vertices are at the top of the second stack. While the algorithm pops vertices of the SCC from the stack, our implementation of GOR examines their adjacency lists to check if there is a negative arc inside the SCC. If there is, the algorithm terminates with a negative cycle; otherwise, it produces a topological ordering of G_d with back arcs deleted. Vertices will be scanned in that order. The resulting implementation still runs in $O(nm)$ time and is guaranteed to find a negative cycle in G_d if there is one. In our experiments, we observed that this new implementation has an additional overhead of about 20% on graphs without negative cycles.

As a heuristic to speed up the detection of "obvious" cycles, we also check for negative back arcs during the DFS, and stop immediately if one is found.

5.3   RD. We now discuss a new $O(n)$-pass algorithm, which we call *robust Dijkstra* (RD). Unlike previous methods, RD takes potentials into account when deciding which vertex to scan next. It does so by associating to each labeled vertex v a *key*, defined as the (strictly positive) difference between its *previous potential* (the value of $d(v)$ during the last scan of v) and its current potential. If v has not yet been scanned, we use its initial potential as its *previous potential*. Note that the *previous potential* does not change when a vertex is labeled.

The algorithm partitions the labeled vertices into two sets, Q and S: Q contains those that have yet to be scanned in the current pass, and S contains the rest. Set Q is maintained as a priority queue ordered by key values, and S as an ordinary queue. Initially, all vertices are labeled and have zero potential. At each step, we extract a vertex with maximum key from Q and scan it. When a vertex becomes labeled, it is added to S if it has been scanned during the current pass or to Q otherwise.[2] If the potential of a labeled vertex v in Q decreases, its key increases and an increase-key operation is performed on v. When Q becomes empty, a new pass starts by moving vertices from S to Q and heapifying Q. The algorithm uses subtree disassembly with updates, which causes vertices in the affected subtrees to be removed from S or Q.

It may seem more natural to give priority to vertices with the lowest potentials (instead of highest improvement). In practice, however, it does not work well because a potential transformation may affect the vertex selection order in every pass. For example, if for some vertex v we decrease the length of incoming arcs by a large value and increase the length of outgoing arcs by the same amount, v will be scanned immediately after it is added to Q.

As its name suggests, RD is a generalization of Dijkstra's algorithm: on the shortest path problem, these algorithms will scan the same sequence of vertices if arc lengths are nonnegative. (RD initialization is slightly different for the shortest path problem: the source potential is still set to zero, but all others must be set to M, a finite number larger than the length of any possible shortest path.) Note that, if there are negative input arcs, RD cannot use a monotone priority queue, such as multilevel buckets [7]; a general (and potentially less efficient) data structure must be used instead. Using Fibonacci heaps [12], the algorithm runs in $O(n(m + n \log n))$ time. Our implementation, which we call RDH, uses a 4-heap [24] (which is more efficient in practice) and runs in $O(nm \log n)$ time.

In an effort to reduce the data structure overhead, we also implemented RDB, which maintains an approximate bucket-based priority queue. Each bucket consists a doubly-linked list of vertices that behaves as a queue with support for arbitrary deletions.[3] Bucket 0 contains vertices with key 0, and bucket i contains vertices with keys in $[2^{i-1}, 2^i)$. We also maintain an upper bound μ on the index of the biggest-key nonempty bucket. This data structure supports the insert, delete, and increase-key operations in $O(\log \log M)$ time and extract-max in $O(\log M)$ time. This leads to an $O(n(m \log \log M + n \log M))$ time bound for RDB.

5.4   MBFCT. Wong and Tan's MBFCT algorithm [26] is a "local" variant of BFCT. All vertices start the algorithm unreached and with zero potential. The following procedure is executed while there are unreached vertices: Pick an unreached vertex v, mark it labeled, set $p(v) = s$, and run BFCT; terminate if a negative cycle is found, otherwise proceed to the next iteration. The potential function modified by BFCT is maintained throughout MBFCT, which means that only vertices whose potentials improve are visited in each new

---

[2]A variant that adds all newly labeled vertices to S proved to be less robust in our tests.

[3]We also tested buckets as stacks, with largely similar results.

iteration.

Since each call to BFCT can only reduce the number of unreached vertices, the algorithm terminates after at most n calls. It runs in $O(n^2m)$ worst-case time, and our experiments include a family of instances that match this bound. Although this is not an $O(n)$-pass algorithm, in practice it is often faster than BFCT [26], especially on graphs with several small negative cycles. Unlike the original implementation of MBFCT, ours does updates during subtree disassembly, which sometimes makes it more efficient.

Note that there is some similarity between MBFCT and Pallottino's algorithm [20]. The latter also works "locally" using the Bellman-Ford-Moore algorithm.

Next we sketch Pallottino's algorithm. It partitions vertices into low- and high-priority sets. Initially every vertex has low priority, and priorities can only change from low to high. The algorithm maintains two queues, H and L, which contain labeled vertices of high and low priority, respectively. Initially, L contains all labeled vertices. The algorithm works in phases. At the beginning of each phase H is empty. The algorithm removes a vertex from the head of L, changes its priority to high, and adds it to H. Then the vertices in H are scanned in FIFO order. A scan can add vertices to both H and L, depending on the priority of the labeled vertex. The phase terminates when H becomes empty. (One can show that at this point the current potentials are feasible for the subgraph induced by the set of high-priority vertices.)

Two main differences between MBFCT and Pallottino's algorithm are that MBFCT does not restrict scans to high-priority vertices and uses subtree disassembly (though Pallottino's algorithm can also use subtree disassembly, as shown in [2]). Both algorithms run in $O(n^2m)$ time, however, and our bad-case example for MBFCT, described in Section 6, is also bad for Pallottino's algorithm.

5.5 Other algorithms. We experimented with several other algorithms. This section discusses some that, although natural, ended up being relatively less competitive in practice.

A natural approach is to combine the ideas of topological sort and subtree disassembly. One can either do GOR-like topological sorting at each pass of BFCT, or perform subtree disassembly (which may make mark some labeled vertices as unreached) while running GOR. We tried several variants of these ideas, but failed to get a robust performance improvement on our best implementations.

We also considered an algorithm that has potentially useful theoretical performance guarantees in the incremental context. The *arc-fixing algorithm* (AF) is motivated by the chain-elimination procedure of [14]. To gain intuition, suppose the input graph as only one negative arc (v,w) with $\ell(v,w) = -x$. We can change $\ell(v,w)$ to zero and apply Dijkstra's algorithm to the resulting graph with w as the source. If the distance to v is less than x, the input graph has a negative cycle formed by (v,w) and the computed shortest path from w to v. Otherwise one can use the computed distances to get feasible potentials, "fixing" the arc (v,w).

Next we describe the arc-fixing algorithm. All vertices start with zero potential. Each iteration begins by checking if the admissible graph G_d has a negative cycle. If so, the algorithm terminates. Otherwise, it tries to update the potentials by running two shortest path computations.

Let G' be the graph induced by the arcs out of the root vertex s (with length zero) together with all arcs (v,w) with $\ell_d(v,w) < 0$. This graph is acyclic. In linear time, we compute the distance $\pi(v)$ in G' from s to each vertex v. Note that $\pi(v) \leq 0$.

The algorithm then builds a new graph G'', which has the same topology as G, but different arc lengths: arcs out of s have length $\pi(v)$, and the remaining ones have length $\max\{0, \ell_d(a)\}$. Since all negative arcs of G'' are adjacent of s, Dijkstra's algorithm correctly finds the distances $p(v)$ in G'' from s to each vertex v while scanning each vertex once. Note that $p(v) \leq 0$.

The algorithm completes an iteration by updating $d(v)$ to $d(v) + p(v)$.

One can show that the arc-fixing algorithm has the following properties: (i) once an arc becomes nonnegative, it remains nonnegative; (ii) in each iteration, either there is a vertex that has outgoing negative arcs in the beginning but not at the end, or the next iteration discovers a negative cycle. Therefore, if the number of vertices with outgoing negative arcs in G is k, the algorithm terminates in k iterations. Although this is a promising bound for incremental scenarios, this algorithm was not competitive with the best algorithms in our study.

6 Worst-Case Instances

This section introduces graph families that elicit the worst-case behavior of the algorithms we implemented. We concentrate on the feasibility problem, but we note that simple modifications of these families give worst-case instances for the single-source shortest path problem as well. Within a family, the number of vertices in each network is a linear function of a parameter k, which indicates how many times a given gadget is repeated. The number of arcs is also linear in k for most families, but there are two cases in which it is

quadratic. To the best of our knowledge, these are the first families to match the theoretical worst-case bounds of any of the algorithms we study.

6.1 BFCT. The network BAD-BFCT(k) contains $n = 4k - 1$ vertices and $m = 5k - 3$ arcs. Vertices 1 to $3k - 2$ together with the arcs $(i + 1, i)$, $1 \leq i \leq 3k - 3$, form a path P. Every third vertex on P is connected to vertex $3k - 1$, i.e., we have the arcs $(3(i - 1) + 1, 3k - 1)$ for $1 \leq i \leq k$. Finally, $3k - 1$ is connected to vertices $3k$ to $4k - 1$. All arcs in this graph have length -1. Figure 1 gives an example for $k = 4$.
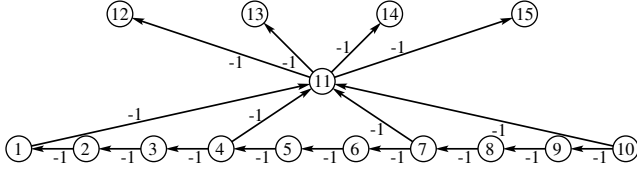

Figure 1: Worst-case input for BFCT ($k = 4$).

Assume that the vertices are initially ordered in the BFCT queue by their identifier, from smallest to largest. During the first pass, the vertices on P are scanned in increasing order. After the first pass, the vertices on P (except the last vertex) are scanned once more, in decreasing order. Because of subtree disassembly, only one vertex on P is in the queue at a time. Consider what happens when vertex $3i$ $(1 \leq i < k)$ is scanned for the second time. At this point only $3i$ and $3k - 1$ are in the queue. After scanning $3i$, vertex $3i - 1$ is inserted into the queue following $3k - 1$. After scanning $3k - 1$, vertices $3k, \ldots, 4k - 1$ are inserted into the queue and scanned after $3i - 1$. Scanning $3i - 2$ disassembles the subtree of $3k - 1$ and produces a similar state as before scanning $3i$. The same pattern continues for $j = i - 1, \ldots, 1$. This implies a total of $\Theta(k^2) = \Theta(n^2)$ scans.

We note that by adding the arcs $(3i - 1, 3k - 1)$, the subtree of $3k - 1$ is disassembled before scanning vertices $3k$ to $4k - 1$, which results to $O(1)$ scans per vertex. It is also interesting to note that without subtree disassembly the subgraph induced by P alone is a worst-case instance.

6.2 MBFCT. With a similar network we can construct a worst-case instance for MBFCT. To obtain BAD-MBFCT(k) from BAD-BFCT(k) we first reverse the orientation of the path P. Let $G(k)$ be the resulting graph. Then we add new vertices $4k, \ldots, 6k - 1$, which are connected to the extremes of P as follows: the even vertices are connected to vertex 1, and the odd vertices are con-

nected to $3k - 2$. The length of the arc leaving vertex $4k + i$ is $-4k(i + 2)$, $0 \leq i \leq 2k - 1$. Figure 2 gives an example for $k = 4$. Note that $n = 6k - 1$ and $m = 7k - 3$.
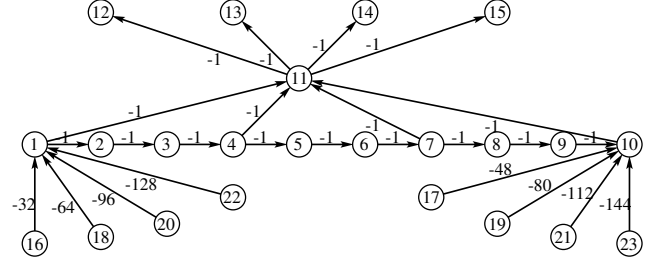

Figure 2: Worst-case input for MBFCT ($k = 4$).

After vertex $4k + 2i$ is scanned, MBFCT needs $\Theta(k^2)$ scans to process $G(k)$. The role of each vertex $4k + 2i + 1$ is to disassemble the subtree rooted at $3k - 1$, so that the next pass over $G(k)$ will still require $\Theta(k^2)$ scans. This gives a total of $\Theta(k^3) = \Theta(n^3)$ scans.

We note that, if we connect an artificial source to all original vertices with arcs of length zero, then we obtain a worst-case ($\Theta(n^3)$ scans) instance for Pallottino's shortest paths algorithm [20].

6.3 GOR. The worst-case network BAD-GOR(k) consists of a path P of k vertices $1, \ldots, k$, a vertex $k + 1$ with k incoming and k outgoing arcs, and k vertices $k + 2, \ldots, 2k + 1$. Set $\ell(1, 2) = -3k$, $\ell(1, k + 1) = -1$, and $\ell(i, i + 1) = 1$ for $2 \leq i \leq k - 1$. Also, $\ell(k + 1, k + 1 + i) = -1$ for $1 \leq i \leq k$, and $\ell(i, k + 1) = 2(k - i)$ for $2 \leq i \leq k$. We have $n = 2k + 1$ and $m = 3k - 1$. Figure 3 gives an example for $k = 7$.

Figure 3: Worst-case input for GOR ($k = 7$).

Note that GOR needs $\Theta(k)$ passes to process P: Initially the only admissible arc on P is $(1, 2)$, and scanning vertex 2 makes $(2, 3)$ admissible. These are the only arcs on P that are considered during the first pass of the algorithm. Each subsequent pass makes two more arcs on P admissible. The arcs leaving vertex $k + 1$ remain admissible in each pass, because the length of the path $(1, 2, .., i, k + 1)$ is smaller than the length of

$(1, 2, .., i, i+1, k+1)$ for $i = 2, \ldots, k-1$. This causes GOR to scan vertices $k+1, \ldots, 2k+1$ in each pass. It follows that the total number of scans is $\Theta(k^2) = \Theta(n^2)$.

6.4 RD. The elaborate rule that RD employs for selecting the next vertex to scan (combined with subtree disassembly with updates) makes it hard to construct worst-case instances. We can simplify the problem slightly by making additional assumptions on how the priority queue breaks ties. This is the case in our implementation of RDB, since each bucket is implemented as a queue. But in a heap-based implementation of RDH it is hard to predict which element among the ones with maximum key will be returned. Our implementations of RDB and RDH do exhibit quadratic running time for the worst-case families we created, even though the latter does not use any special tie-breaking rule. The reader should be aware, however, that other implementations may behave differently.

For an intuition on the construction of the worst-case networks, consider the graph $G(k)$ shown in Figure 4. It consists of k arcs $(x_i, y_i) = (2i-1, 2i)$ for $1 \le i \le k$, together with the arcs (x_i, x_{i+1}) and (y_i, x_{i+1}) for $1 \le i < k$. We require each path (x_i, y_i, x_{i+1}) to be shorter than (x_i, x_{i+1}). To that end we can set $\ell(x_i, y_i) = 0$, $\ell(y_i, x_{i+1}) = -2$ and $\ell(x_i, x_{i+1}) = -1$. In the first pass of RD, vertex x_{i+1} has priority over y_i; hence all the x_i's will be scanned first, and then all the y_i's. In the following passes, when x_i is scanned y_i and x_{i+1} get the same priority, equal to the distance improvement for x_i. Suppose that (x_i, x_{i+1}) is processed before (x_i, y_i). Then, assuming that the priority queue of RD favors vertices that are inserted more recently, x_{i+1} will be scanned first. In this case, RD needs $\Theta(k)$ passes[4] to compute the shortest path from 1 to $2k$. The total number of scans is therefore $\Theta(k^2) = \Theta(n^2)$.
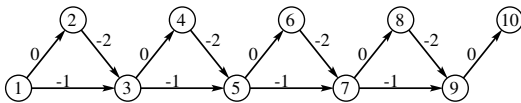

Figure 4: Worst-case input for RD ($k = 5$) when the priority queue returns the most recently inserted vertex among all those with maximum key.

The graph $G(k)$ is actually a worst-case input for the version of RDB where each bucket is implemented as a stack, since among all vertices in the same bucket the one most recently inserted will be preferred. If we

represent each bucket as a queue (as in our implementation), however, RDB can process this graph with a linear number of scans. We can get a worst-case instance for this version of RDB as follows.

First, we swap the order in which the arcs leaving each odd vertex in $G(k)$ are processed, so that when x_i is scanned, x_{i+1} is inserted into the queue before y_i. To simplify the discussion, consider a shortest path computation starting from x_1 and assume that all labeled vertices are inserted in a single bucket. (Similar arguments to the ones below apply to the actual RDB algorithm for the feasibility problem.) After x_1 is scanned, x_2 and y_1 are inserted, in this order, into the queue, and x_2 is scanned before y_1. When y_1 is scanned, the distance to x_2 is improved but the current pass ends, as x_2 was already scanned. Then, the second pass begins with x_2 being scanned again. The same pattern continues until all odd vertices (except x_1) are scanned twice and all even vertices are scanned once, giving a total of k passes and $\Theta(k)$ scans in total.

To make the number of scans quadratic, we augment this graph with a set of new vertices that will be scanned in every pass. We connect each even vertex y_i to a new vertex $2k+1$ and connect $2k+1$ to k new vertices $2k+2, \ldots, 3k+1$. All new arcs have length -1. The new graph has $n = 3k+1$ vertices and $m = 5k-2$ arcs. Figure 5 gives an example for $k = 5$. The new vertices will be scanned in each pass, thus resulting in $\Theta(k^2)$ scans.

Figure 5: Worst-case input for RDB and RDH ($k = 5$).

The experimental results in Section 7.2 (Table 6) suggest that our implementation of RDH also exhibits quadratic number of scans for the graph of Figure 5. We refer to the this family as BAD-RD(k).

It is interesting to note that another graph inducing $\Theta(k^2)$ scans for RDB is a complete directed acyclic graph (DAG) on k vertices. In this graph we have an arc (i, j) with length -1, for each $i < j$. Also, the experimental results suggest that RDH requires a superlinear number of scans for this family, which we call COMP-DAG(k). Note that, unlike all previous families, these graphs are dense.

[4]The number of passes is exactly $k+1$ if subtree disassembly is not used, and $k/2 + 1$ if it is.

6.5 Arc-fixing algorithm. This is another family of dense graphs. We have $n = 3k + 2$ and $m = k^2 + 4k + 1$. To construct a graph, we start with a path $P = (1, 2, \ldots, 2k + 2)$ where the arcs lengths alternate between -1 and 1. Then we add k vertices, $2k + 3$ to $3k + 2$. For $0 \leq i \leq k - 1$, vertex $2k + 3 + i$ has an incoming arc of length $k + 1 - i$ from $2i + 1$, and $2(k - i)$ unit-length outgoing arcs to $2i + 3, 2i + 4, \ldots, 2k + 2$. Figure 6 gives an example for $k = 3$.

Figure 6: Worst-case input for the arc-fixing algorithm ($k = 3$).

Each iteration of the arc-fixing algorithm causes the reduced cost of one negative arc on P to become zero. At the end of the i-th iteration all vertices on the path $P_i = (2i + 3, 2i + 4, \ldots, 2k + 2)$ have equal potentials, so the reduced costs of the arcs on P_i do not change. Therefore, the algorithm requires $\Theta(k) = \Theta(n)$ iterations. Since each iteration runs Dijkstra's algorithm, the total running time is $\Omega(mn)$.

7 Experiments

7.1 Experimental setup. The improved implementations of BFCT and GOR, the most robust algorithms in previous studies, are the baseline of our experimental evaluation. We also include MBFCT, for which [26] gives encouraging results. Finally, we test our new algorithms, RDH and RDB, which are superior on some problem families.

Our main measure of performance is the number of scans per vertex, which is machine-independent. It includes both shortest path scans and auxiliary scans (such as those during DFS). All results reported are averages taken over 10 (or more, when noted) instances with different pseudorandom generator seeds.
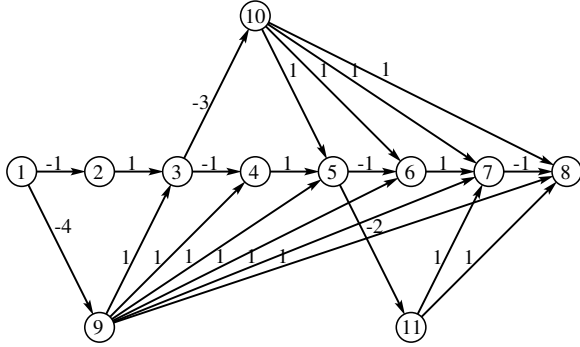
We also report some running times to measure the data structure overhead for different algorithms. Our timed experiments were run on a 3 GHz Intel Xeon with 32 GB of RAM running Red Hat Enterprise Linux

5. For readability and succinctness, however, we omit running times for most experiments.

All algorithms were implemented in the same style and made as efficient as possible. They were coded in C++ and compiled with gcc v. 4.1.2 with the -O4 flag. Arc lengths and potentials were represented as 32-bit signed integers.

7.2 Feasibility. We start with the standard (non-incremental) feasibility problem. As seen in [2], the number of negative cycles and their cardinality (number of arcs) can have a significant effect on performance. For a tight control over the structure of the instances, we created a filter called NEGCYCLE. It takes as input an arbitrary base graph with no negative arcs and adds vertex-disjoint negative cycles to it. Every arc on these cycles has length zero, except for one with length -1. As in [2], from each base graph family we create five subfamilies: (01) no negative cycles; (02) one negative cycle with three arcs (small cycle); (03) many small negative cycles; (04) a constant number of medium negative cycles; and (05) one Hamiltonian negative cycle.

After adding the negative cycles (or not, in case 01), we apply a potential transformation to "hide" them. A potential transformation with parameter X selects, for each vertex v, an integer $d(v)$ uniformly at random from $[0, X)$, adds $d(v)$ to each of v's incoming arcs, and subtracts $d(v)$ from the outgoing arcs; the effect is to replace lengths by reduced costs. Unless otherwise noted, we use $X = 1000$.

Our testbed includes the graph families tested in [2], augmented with road networks and worst-case examples. We discuss each family in turn.

7.2.1 Random graphs. The SPRAND generator [3] creates an instance with n vertices and m arcs by building a Hamiltonian cycle on the vertices and adding $m - n$ arcs at random. All arc lengths are selected uniformly at random from $[L, U]$.

In our first experiment, we set $L = 0$ and $U = 1000$ and fix $m = 5n$ (results were similar with other graph densities). We vary n, and apply the NEGCYCLE filter in each case. Table 1 shows the average number of scans per vertex for this family (which we call RAND5).

Note that the number of scans per vertex barely increases (if at all) with graph size, which suggests that on these graphs the algorithms perform much better than their worst-case bounds. For the 03 subfamily (which has many cycles of size 3), most algorithms actually do better (relative to n) as the graph size increases. The main exception is GOR, which performs a first pass that visits essentially every arc in the graph,
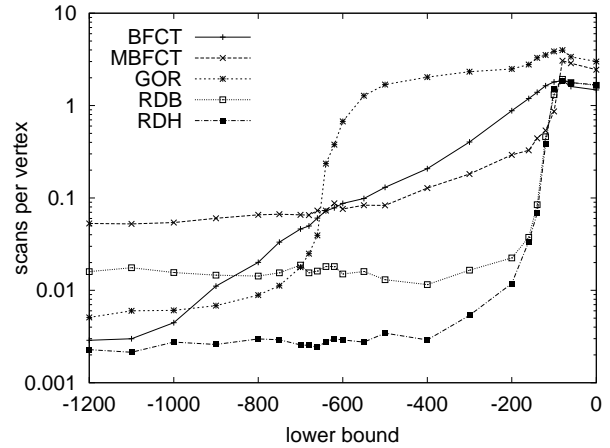
Table 1: Feasibility: Scans per vertex on RAND5.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262144	1.0105	1.0107	1.1016	1.0107	1.0180
	524288	1.0105	1.0107	1.1013	1.0106	1.0180
	1048576	1.0105	1.0107	1.1019	1.0106	1.0181
	2097152	1.0105	1.0106	1.1019	1.0106	1.0182
02	262144	0.9813	0.5803	1.0939	0.9311	0.7954
	524288	0.9713	0.4916	1.0934	0.7703	0.7805
	1048576	1.0087	0.6095	1.0939	0.9508	0.7082
	2097152	1.0078	0.5313	1.0938	0.7200	0.7851
03	262144	0.0798	0.0001	1.2186	0.0001	0.0001
	524288	0.0667	$<10^{-4}$	1.1947	$<10^{-4}$	$<10^{-4}$
	1048576	0.0397	$<10^{-4}$	1.2191	$<10^{-4}$	$<10^{-4}$
	2097152	0.0319	$<10^{-4}$	0.9882	$<10^{-4}$	$<10^{-4}$
04	262144	2.1171	0.2050	3.3318	1.9126	1.8541
	524288	2.1399	0.3257	3.3741	1.9669	1.8826
	1048576	2.1677	0.4349	3.4460	1.9929	1.9222
	2097152	2.1388	0.3724	3.5164	1.9872	1.9083
05	262144	5.2216	3.0555	10.6780	4.6662	4.6957
	524288	5.1291	2.5635	10.7117	4.6237	4.5978
	1048576	5.1830	3.0098	10.9752	4.6047	4.5592
	2097152	4.9904	2.5356	10.9011	4.5558	4.6200

Figure 7: Feasibility on random graphs with 65536 vertices and arc lengths in $[L, 1000]$, with L variable.

but only detects a cycle when processing the SCCs in the second pass. For this family, MBFCT, RDH and RDB, which tend to concentrate on a small region of the graph, do better than BFCT, which takes a more global approach. For the 04 subfamily, MBFCT is the winner.

The results of our second experiment, also on random graphs, are shown in Figure 7. Here we fix $n = 65536$, $m = 5n$, the maximum arc length U at 1000, and vary the minimum allowed arc length L from 0 to -1200. We apply a potential transformation with parameter 2000 to the resulting graph (this is not done in [2], but it leads to more interesting results). The figure shows the average number of scans per vertex (over 50 different random seeds) as a function of L. Every instance with $L \leq -100$ had a negative cycle.

All algorithms are faster for lower values of L, when negative cycles are more numerous. RDH needs the fewest scans to find such cycles, in some cases winning by several orders of magnitude—the priority queue helps finding "obvious" cycles by focusing on a small number of candidate paths. The bucket-based version of the same algorithm, RDB, cannot find a cycle as fast: since buckets are organized as queues, the algorithm tends to expand many candidate paths at once. In fact, preliminary experiments have shown that organizing buckets as stacks (which tends to favor the most recently improved path) would make the algorithm about as fast as RDH for this problem family, but the stack-based implementation is less robust in general.

Both BFCT and GOR are not much slower than RDH

when L is very negative, but degrade as it increases. GOR does so suddenly, at around $L = -650$, when the cycle-detection heuristic (which looks for negative back arcs during the DFS) ceases to be effective. BFCT degrades more slowly, but soon becomes worse than MBFCT.

7.2.2 Grids. We also tested the algorithms on *grid networks*, which are grids embedded on a torus, created by the TOR generator [2]. Specifically, vertices correspond to points on the plane with integer coordinates $[x, y]$, $0 \leq x < X$, $0 \leq y < Y$. These points are connected "upward" by *layer* arcs of the form $([x, y], [x, y + 1 \bmod Y])$, and "forward" by *inter-layer* arcs of the form $([x, y], [x + 1 \bmod X, y])$. In addition, there is a source connected to all vertices with $x = 0$. Lengths are chosen uniformly at random from $[1, 100]$ for layer arcs, and from $[1\,000, 10\,000]$ for inter-layer arcs (including those from the source). Once the grid is formed, we apply the NEGCYCLE filter to create distinct subfamilies.

Table 2 shows the average number of scans per vertex on *square grids with negative cycles* (SQNC), for which $X = Y$. All five subfamilies are shown.

This family is slightly harder than RAND5 for all algorithms, but their relative performance remains largely unchanged. An important exception is BFCT becoming faster than MBFCT on subfamily 05, which has a Hamiltonian negative cycle. Also, RDB outperforms RDH on the first four subfamilies.

Experiments with *long grids with negative cycles* (LNC), which have $X = n/16$ layers with $Y = 16$ vertices each, yielded very similar results, as Table 3

Table 2: Feasibility: Scans per vertex on SQNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	1.0153	1.6324	4.0449	1.7912	2.3171
	524177	1.0109	1.6303	4.0443	1.7906	2.3209
	1048577	1.0077	1.6276	4.0418	1.7908	2.3237
	2096705	1.0054	1.6255	4.0411	1.7905	2.3260
02	262145	0.9897	0.2255	3.1748	0.9189	1.2664
	524177	0.9213	0.3262	3.1536	0.6701	1.0183
	1048577	0.9667	0.5502	3.2424	0.4878	1.1787
	2096705	0.8345	0.3608	3.1213	0.6290	1.4416
03	262145	0.1973	0.0011	2.7220	0.0011	0.0023
	524177	0.1760	0.0011	2.7227	0.0010	0.0014
	1048577	0.2068	0.0003	2.7232	0.0003	0.0008
	2096705	0.1078	0.0008	2.7234	0.0008	0.0007
04	262145	2.1214	2.6874	6.0961	2.4650	2.7870
	524177	1.8846	1.5407	5.4931	2.3570	2.6913
	1048577	1.6985	1.1161	5.2201	2.2186	2.6100
	2096705	1.7382	0.6888	5.4070	2.2453	2.6151
05	262145	17.1357	48.5386	37.0674	12.2037	11.2036
	524177	17.7833	49.9466	37.6437	12.6346	11.6750
	1048577	18.8587	52.1875	39.6365	12.9990	12.0022
	2096705	19.8007	54.4070	42.1887	13.4241	12.2391

Table 3: Feasibility: Scans per vertex on LNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	1.3564	1.9224	4.0411	1.7901	2.0752
	524289	1.3566	1.9262	4.0428	1.7919	2.0769
	1048577	1.3566	1.9264	4.0427	1.7916	2.0737
	2097153	1.3567	1.9257	4.0433	1.7918	2.0761
02	262145	1.0063	0.3684	2.9553	0.3818	0.6307
	524289	0.9934	0.6405	2.9375	0.6405	1.4967
	1048577	0.9716	0.5137	3.1860	0.7144	1.3092
	2097153	1.0624	0.5253	3.2567	0.8406	0.5776
03	262145	0.1508	0.0005	2.6767	0.0003	0.0006
	524289	0.0965	0.0004	2.6765	0.0002	0.0001
	1048577	0.0829	0.0001	2.6762	0.0001	0.0001
	2097153	0.0762	$<10^{-4}$	2.6763	$<10^{-4}$	0.0001
04	262145	3.7127	7.1427	8.9451	3.5805	3.6780
	524289	3.9816	8.3143	9.4718	3.8881	3.8592
	1048577	5.0501	9.5579	11.7692	4.5792	4.4755
	2097153	4.4770	10.8935	10.2916	4.3081	4.2728
05	262145	16.9175	46.6560	36.0015	12.1477	11.0841
	524289	18.1748	49.0632	38.1384	12.8522	11.7341
	1048577	19.2769	52.5656	40.2583	13.2109	11.9235
	2097153	19.9966	53.2295	41.8637	13.5788	12.1767

shows. An important difference is that subfamily 04 is harder for all algorithms, especially MBFCT.

7.2.3 Layered networks. These graphs, also created by TOR, are partitioned into layers $0, \ldots, X - 1$, each consisting of a cycle of length 32 plus 64 random arcs. Arc lengths within a layer are chosen uniformly at random from the interval $[1, 100]$. In addition, each vertex has five arcs to forward layers (including the next one). The length of an inter-layer arc that goes x layers forward is picked uniformly at random from $[1, 10\,000]$ and multiplied by x^2. A special source is connected to all vertices in the first layer by zero-length arcs. Finally, we introduce negative cycles with the NEGCYCLE filter. We call this family PNC (*p-hard with negative cycles*). Table 4 reports the results for this family. Note that, compared to other families, PNC is particularly bad for MBFCT when no cycles are present (subfamily 01).

7.2.4 Road networks. We also tested our algorithms on road networks, available at the 9th DIMACS Implementation Challenge web site [6]. Vertices represent intersections and arcs represent road segments (with lengths proportional to the travel distance). We used the NEGCYCLE filter to add negative cycles and perturb arcs lengths. Results for the CAL instance, which represents California and Nevada and has 1.9 million vertices, are shown in Table 5. (Results for other road networks were very similar.) Here MBFCT is usually the best algorithm, especially for the 04 subfamily.

7.2.5 Hard instances. Although the instances tested so far shed light on the feasibility problem, they are not very hard for any of the algorithms studied. In no case was the average number of scans higher than 55, even for graphs with more than 2 million vertices. All algorithms have essentially linear behavior. For a more complete analysis, we created the BAD family, with the worst-case instances described in Section 6.

We tested instances from this family with $k = 200$, 400, 800, and 1600. Recall that the number of vertices in each graph is a linear function of k. The number of arcs is quadratic in k for BAD-AF and COMP-DAG, and linear for the remaining families. Table 6 shows the number of scans per vertex in each case.

The table clearly shows the quadratic worst case of BFCT, GOR, RDB, and RDH in their respective families. Interestingly, MBFCT not only has a cubic worst case on its own subfamily, but also has quadratic behavior in some of the other families. In this sense, it is less robust than the other algorithms. RDB and RDH also have superlinear running time on BAD-AF and COMP-DAG.

It should be noted that all worst-case instances are somewhat sensitive to perturbations, such as permutations of vertex identifiers or potential transformations. This is especially true for BAD-RD, the hardest family to create.

7.2.6 Running times. So far, we have compared the algorithms only in terms of operation counts, which are machine-independent. We now discuss running times.

Table 4: Feasibility: Scans per vertex on PNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	2.0239	8.5484	5.8403	2.8259	2.5846
	524289	2.0243	8.5580	5.8373	2.8268	2.5822
	1048577	2.0245	8.5655	5.8356	2.8265	2.5830
	2097153	2.0244	8.5624	5.8347	2.8262	2.5825
02	262145	1.4243	2.1267	3.8669	0.7111	1.5076
	524289	1.0842	2.3483	4.2981	1.1895	1.1215
	1048577	1.4933	2.1911	3.8810	0.9444	1.1428
	2097153	1.3424	2.5123	4.1195	0.8123	1.2812
03	262145	0.1303	0.0005	2.3687	0.0004	0.0005
	524289	0.0968	0.0003	2.3681	0.0001	0.0002
	1048577	0.0969	0.0005	2.3674	0.0002	0.0002
	2097153	0.0764	0.0001	2.3672	$<10^{-4}$	0.0001
04	262145	6.3613	20.0428	14.0924	6.5855	6.0718
	524289	6.8502	20.0095	15.6653	6.9911	6.4192
	1048577	6.5174	25.3448	13.9551	6.6889	6.3908
	2097153	7.2917	26.2656	15.9607	7.0950	6.6919
05	262145	8.2741	22.0434	16.1135	7.1392	6.8433
	524289	8.7537	17.7014	17.7396	7.3837	6.9304
	1048577	8.9492	28.6557	18.1954	7.5952	6.8891
	2097153	9.5213	21.6117	18.9055	7.9059	7.3997

Table 6: Feasibility: Scans per vertex on hard instances.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
BAD-AF	602	34.7	34.4	3.0	3.2	6.6
	1202	68.1	67.7	3.0	3.9	7.5
	2402	134.7	134.4	3.0	4.4	13.2
	4802	268.1	267.7	3.0	5.0	16.0
BAD-BFCT	799	51.8	374.6	2.3	2.3	2.3
	1599	101.8	749.6	2.3	2.3	1.3
	3199	201.8	1499.6	2.3	2.3	2.2
	6399	401.8	2999.6	2.3	2.3	1.3
BAD-GOR	401	1.0	1.2	103.6	1.0	1.0
	801	1.0	1.2	203.6	1.0	1.0
	1601	1.0	1.2	403.6	1.0	1.0
	3201	1.0	1.2	803.6	1.0	1.0
BAD-MBFCT	1199	1.7	6840.0	2.3	1.7	1.7
	2399	1.7	27012.3	2.3	1.7	1.7
	4799	1.7	107356.7	2.3	1.7	1.7
	9599	1.7	428045.6	2.3	1.7	1.7
BAD-RD	601	1.0	2.7	3.0	61.6	35.4
	1201	1.0	2.7	3.0	128.5	68.8
	2401	1.0	2.7	3.0	262.0	135.4
	4801	1.0	2.7	3.0	528.9	268.8
COMP-DAG	200	1.0	50.5	3.0	50.5	11.0
	400	1.0	100.5	3.0	100.5	15.7
	800	1.0	200.5	3.0	200.5	21.4
	1600	1.0	400.5	3.0	400.5	30.2

Table 5: Feasibility: Scans per vertex on CAL.

FAM	BFCT	MBFCT	GOR	RDB	RDH
01	1.0500	1.0533	1.3251	1.0513	1.0519
02	1.0213	0.4729	1.3218	0.6290	0.8492
03	0.0482	$<10^{-4}$	1.3897	$<10^{-4}$	$<10^{-4}$
04	2.4935	0.4038	4.0262	2.2669	2.1212
05	5.6818	3.3757	12.3628	5.2562	5.2181

For reference, the total running time for RAND5 is shown in Table 7. For this family, the algorithms are usually within a factor of two or three from each other. The main exceptions are GOR, which is substantially slower than other algorithms in subfamily 03, and MBFCT, which is substantially faster in subfamily 04.

For a more detailed analysis, Table 8 shows the average time per scan on the same instance. Note that this table only reports the time spent during the main loop of each algorithm, during which all scans are performed. The time required to allocate and initialize the data structures specific to each method is not considered here. (The total times listed in Table 7 do include initialization.) As a side effect, we could not measure the average time per scan for subfamily 03 with enough precision, since the number of scans is very small. For that reason, the results for this subfamily are not presented in Table 8.

Based on the subfamilies for which we could measure times reliably, we can draw some conclusions. In general, and not surprisingly, MBFCT and BFCT take about the same time to scan a vertex. GOR, which ex-

Table 7: Feasibility: Average total time in seconds for RAND5.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262144	0.105	0.115	0.124	0.157	0.186
	524288	0.220	0.241	0.260	0.338	0.398
	1048576	0.459	0.505	0.541	0.711	0.880
	2097152	0.962	1.059	1.131	1.532	1.804
02	262144	0.102	0.069	0.124	0.144	0.145
	524288	0.211	0.123	0.259	0.260	0.307
	1048576	0.457	0.314	0.539	0.671	0.592
	2097152	0.958	0.579	1.124	1.109	1.387
03	262144	0.014	0.006	0.148	0.011	0.016
	524288	0.026	0.012	0.304	0.022	0.032
	1048576	0.042	0.024	0.651	0.043	0.063
	2097152	0.078	0.048	1.122	0.086	0.122
04	262144	0.413	0.047	0.588	0.508	0.584
	524288	0.909	0.153	1.294	1.187	1.307
	1048576	2.006	0.433	2.892	2.602	2.903
	2097152	4.205	0.820	6.403	5.638	6.293
05	262144	1.328	0.778	2.194	1.620	1.897
	524288	2.841	1.407	4.773	3.653	4.115
	1048576	6.242	3.602	10.745	7.867	9.035
	2097152	12.752	6.425	23.024	16.920	19.647

Table 8: Feasibility: Average time per scan in microseconds for RAND5.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262144	0.37	0.41	0.39	0.55	0.64
	524288	0.39	0.43	0.41	0.60	0.69
	1048576	0.41	0.45	0.43	0.63	0.77
	2097152	0.43	0.48	0.45	0.68	0.79
02	262144	0.37	0.41	0.39	0.55	0.62
	524288	0.39	0.43	0.41	0.59	0.67
	1048576	0.41	0.45	0.43	0.63	0.71
	2097152	0.43	0.48	0.45	0.68	0.76
04	262144	0.73	0.81	0.66	0.99	1.17
	524288	0.80	0.87	0.72	1.13	1.29
	1048576	0.87	0.91	0.79	1.22	1.41
	2097152	0.93	1.01	0.85	1.33	1.54
05	262144	0.97	0.96	0.78	1.32	1.53
	524288	1.05	1.04	0.84	1.50	1.69
	1048576	1.14	1.13	0.93	1.62	1.88
	2097152	1.21	1.20	1.00	1.76	2.02

ecutes cheaper scans during DFS, is often faster than BFCT. RDH is the only method whose time per scan may depend on n. In this family, however, it is less than twice as slow as BFCT, regardless of graph size, which indicates that the heap is often very small. RDB is faster (per vertex) than RDH, but by less than 20% when RDH does well. In contrast, for the SQNC family the dependence of RDH on n is more obvious, as Table 9 shows (once again, we omit subfamily 03). For RDH, scans are slower, especially when the heap is large, as one would expect.

Table 9: Feasibility: Average time per scan in microseconds for SQNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	0.05	0.08	0.07	0.38	0.77
	524177	0.05	0.08	0.07	0.40	0.87
	1048577	0.05	0.08	0.07	0.42	0.98
	2096705	0.05	0.08	0.08	0.44	1.13
02	262145	0.05	0.06	0.05	0.27	0.52
	524177	0.05	0.08	0.05	0.22	0.43
	1048577	0.05	0.08	0.05	0.18	0.58
	2096705	0.05	0.08	0.05	0.23	0.77
04	262145	0.22	0.28	0.11	0.42	0.71
	524177	0.22	0.30	0.11	0.44	0.80
	1048577	0.22	0.31	0.10	0.46	0.91
	2096705	0.24	0.31	0.10	0.48	1.03
05	262145	0.75	0.74	0.48	1.09	1.27
	524177	0.83	0.81	0.53	1.18	1.39
	1048577	0.89	0.88	0.57	1.28	1.54
	2096705	0.97	0.96	0.62	1.43	1.72

7.3 Shortest paths. As already mentioned, the feasibility and shortest path problems are closely related.

In fact, most of the algorithm discussed here can be used to solve the latter problem with minor modifications in the initialization phase. In particular, potentials are initialized to M (a value larger than the length of any shortest path) instead of zero, and the root is the only labeled vertex initially.

Table 10 shows the average number of scans per vertex required to solve the shortest path problem (as opposed to feasibility) on the main families tested. For RAND5, SQNC, LNC, and PNC, we set $n = 2^{20} + 1$.

Table 10: Shortest paths: Scans per vertex.

FAMILY		BFCT	GOR	RDB	RDH
RAND5	01	2.1210	4.4187	1.9610	1.0005
	02	1.2957	2.4887	1.1922	0.2349
	03	0.0003	0.0007	0.0003	$<10^{-4}$
	04	18.8214	38.3942	12.2307	4.7244
	05	21.9508	45.2572	14.4742	7.0170
SQNC	01	2.5172	7.2834	11.3351	13.5625
	02	0.2909	1.8643	0.2693	0.4220
	03	0.0022	0.0065	0.0022	0.0007
	04	5.6998	14.4458	5.9138	4.5004
	05	17.5253	38.1337	13.0139	11.8274
LNC	01	3.2505	6.2108	182.9656	9.7147
	02	0.8680	1.6579	0.2646	2.2448
	03	0.0001	0.0003	0.0001	$<10^{-4}$
	04	10.6546	26.1818	9.7350	8.5018
	05	17.4264	37.3533	12.8423	11.7624
PNC	01	10.4459	17.1255	21.8359	28.3594
	02	2.2423	3.6602	3.9441	2.2051
	03	0.0008	0.0044	0.0004	0.0001
	04	8.4816	19.1209	9.3675	8.5197
	05	6.9598	14.7687	6.2270	6.5254
CAL	01	2.9274	5.9919	12.2258	1.3121
	02	0.5841	1.5298	0.4309	0.3067
	03	$<10^{-4}$	$<10^{-4}$	$<10^{-4}$	$<10^{-4}$
	04	17.5119	37.0022	11.6149	6.2877
	05	24.0836	52.7844	15.9154	10.5812

Comparing these results with those obtained for the feasibility problem, we see that they can be significantly different. When there are no cycles (in subfamily 01), feasibility algorithms are consistently faster than their SP counterparts, since the former do not need to build the full shortest path tree. The difference is considerable for RDB on subfamily 01 of LNC. (In this case, representing buckets as stacks instead of queues would not improve performance; in fact, it would be much worse, according to our preliminary experiments.)

Even when negative cycles are present, there are cases in which feasibility algorithms are significantly faster than the corresponding SP version. This often happens when the negative cycles are relatively long, as in subfamilies 04 and 05.

An interesting case is that of subfamily 03, with numerous but very small cycles. We have seen that

almost all feasibility algorithms can find a negative cycle while visiting a very small portion of the graph. The exception was GOR, which had to traverse the entire graph before finding a negative cycle. This only happens because all vertices are initially labeled; in the SP version of GOR, in contrast, only the source is labeled in the beginning. As a result, the algorithm is about as fast as the other methods.

The results of this section suggest that one should not compare feasibility and shortest path algorithms directly, and explains the difference between our results and those of [26].

7.4 Incremental feasibility. The experiments reported up to this point have assumed that one must solve the feasibility problem from scratch. There are cases, however, when one must simply adjust a feasible set of potentials after a small modification in the underlying graph. We call this the *incremental feasibility problem* (IFP). In general, we want to study a dynamic situation in which one solves a sequence of feasibility problems, each a perturbation of the previous one. This occurs in applications such as the minimum cycle mean problem [4].

To assess how an FP algorithm would behave in the incremental context, we give it two inputs: the graph itself and a list of all vertices with negative outgoing arcs. Since all potentials are pre-initialized to zero, the algorithm needs to focus initially only on these vertices. Unlike the standard FP case, the measurements we report (operation counts) do *not* include the initialization phase, since our goal is to model a situation in which the algorithm is restarted after computing a feasible set of potentials.

To create an IFP instance, we start with a graph containing only nonnegative arcs and make a random subset of them negative. To change the length of an arc (v, w), we first use Dijkstra's algorithm to determine the distance D from w to v with respect to the current arc lengths. If v is not reachable from w, we set $\ell(v, w)$ to -1. Otherwise, we decrease $\ell(v, w)$ to $-2D-1$, $-D-1$, $-D$, or $-D/2$ to create a very negative cycle, a slightly negative cycle, a zero cycle, or a positive cycle. We denote these cases by -2, -1, 0, and 1, respectively.

We must be careful when changing the lengths of more than one arc. In cases 0 and 1, we perform these changes sequentially: when computing the distance between the endpoints of the i-th arc, we use the potentials from previous Dijkstra's computations to maintain arc nonnegativity. In cases -1 and -2, which create negative cycles, we perform all shortest path computations on the original graph, and change all arc lengths at once.

Table 11: Incremental feasibility: Scans per vertex on RAND5 ($n = 65536$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	0.122	0.122	0.905	0.145	0.112
	2	0.143	0.132	1.115	0.123	0.140
	64	0.221	0.146	0.405	0.038	0.032
	1024	0.361	0.223	0.785	0.021	0.007
-1	1	0.222	0.222	0.833	0.157	0.176
	2	0.365	0.335	1.326	0.246	0.172
	64	0.571	0.579	0.610	0.061	0.037
	1024	0.513	0.200	0.769	0.018	0.007
0	1	0.512	0.512	1.073	0.431	0.423
	2	1.229	1.251	2.503	1.004	0.979
	64	3.543	5.711	8.911	2.828	2.710
	1024	4.199	9.171	12.795	3.230	3.083
1	1	0.004	0.004	0.008	0.004	0.004
	2	0.012	0.012	0.024	0.012	0.012
	64	1.034	1.137	1.933	0.960	0.850
	1024	5.151	8.930	9.207	3.724	3.391

Table 11 shows the average number of scans per vertex required to find feasible potentials starting from an SPRAND-generated graph with $n = 2^{16}$ and $m = 5n$. For each case, we vary the number of arcs made negative: 1, 2, 64 or 1024. Tables 12, 13 and 14 present the average number of scans per vertex for SQNC, LNC, and PNC instances, all with 65537 vertices.

Table 12: Incremental feasibility: Scans per vertex on SQNC ($n = 65537$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	1.433	1.433	30.412	0.693	0.701
	2	1.355	1.185	13.544	0.733	0.796
	64	1.938	0.350	8.763	0.365	0.396
	1024	2.619	0.489	5.138	0.622	0.217
-1	1	1.597	1.597	23.622	3.756	0.703
	2	1.286	1.452	16.768	1.270	0.797
	64	1.844	0.290	8.709	0.371	0.338
	1024	2.635	0.502	5.043	0.591	0.219
0	1	1.612	1.612	22.908	3.817	0.704
	2	2.066	2.068	26.526	4.929	0.973
	64	2.326	2.288	14.749	5.124	2.271
	1024	2.515	4.962	10.007	5.793	3.918
1	1	0.612	0.612	5.831	1.024	0.349
	2	1.093	1.086	9.256	1.923	0.663
	64	2.738	2.545	14.984	5.280	4.243
	1024	3.227	4.382	14.738	6.717	5.668

Here RDH is the most robust algorithm. It is never much worse than any other method, and often much better. RDB has comparable behavior for random graphs, but is noticeably worse for TOR-based instances. Unsurprisingly, BFCT and MBFCT have the exact same

Table 13: Incremental feasibility: Scans per vertex on LNC ($n = 65537$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	0.650	0.650	2.430	1.278	0.200
	2	0.390	0.800	1.541	3.291	0.315
	64	0.057	0.079	0.167	0.295	0.029
	1024	0.500	0.004	0.754	0.590	0.001
-1	1	0.650	0.650	2.506	5.518	0.200
	2	0.390	0.800	1.590	3.640	0.315
	64	0.056	0.079	0.165	0.246	0.029
	1024	0.497	0.004	0.757	0.589	0.001
0	1	0.650	0.650	2.232	5.416	0.200
	2	2.908	2.908	8.509	23.461	0.969
	64	3.233	3.231	6.169	26.759	1.563
	1024	3.209	3.285	5.762	25.394	1.655
1	1	0.324	0.324	1.114	1.883	0.100
	2	1.585	1.531	5.089	9.016	0.493
	64	4.377	3.727	9.457	23.142	3.320
	1024	4.492	3.994	9.061	23.392	4.120

Table 14: Incremental feasibility: Scans per vertex on PNC ($n = 65537$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	5.183	5.183	8.425	6.924	0.493
	2	3.623	3.492	5.992	4.566	0.337
	64	0.002	0.117	0.019	0.058	0.010
	1024	0.016	0.031	0.059	0.057	0.003
-1	1	4.853	4.853	8.407	6.159	0.497
	2	3.645	3.489	5.975	4.507	0.337
	64	0.003	0.122	0.032	0.133	0.010
	1024	0.017	0.035	0.061	0.099	0.003
0	1	4.853	4.853	7.743	6.183	0.497
	2	7.735	7.735	12.254	9.904	0.861
	64	9.174	9.900	15.224	11.704	4.782
	1024	6.678	16.118	14.239	9.288	4.050
1	1	2.248	2.248	3.542	2.744	0.249
	2	4.535	4.257	7.098	5.518	0.473
	64	11.239	10.779	18.913	13.911	5.131
	1024	9.279	22.839	18.486	12.183	5.701

performance when there is only one original negative arc, since they reduce to the same algorithm. With more arcs, MBFCT is better than BFCT when negative cycles are obvious, but BFCT is clearly superior when they are absent. Once again, GOR is not competitive.

Table 15 presents the corresponding results for CAL. Since all algorithms perform very few scans per vertex, we report the actual number of scans in this case. In particular, MBFCT, RDB and RDH are remarkably good when there are negative cycles. It is not hard to see why. In road networks, an arc tends to be the shortest path between its endpoints. In addition, the input instance is undirected: for each arc (v, w) there is a corresponding arc (w, v) with the same length. As a result, the negative cycles created by our filter tend to have only two arcs, and are quickly discovered by MBFCT, RDB and RDH.

8 Concluding Remarks

We developed an experimental framework for the feasibility problem that is more extensive than the previous one [2], which was actually designed for the shortest path problem. This led to interesting results. For the feasibility problem, where in some cases it suffices to scan a small subset of the vertices, GOR is not as robust as BFCT. Another previously studied algorithm, MBFCT [26], performs poorly on some problem classes but outperforms BFCT when negative cycles can be found locally. Our new algorithm, RDH, performs even better than MBFCT on many of the classes where the latter works well, although MBFCT still wins on some classes.

Although no single algorithm is dominant, BFCT, MBFCT, and RDH are the most practical among those studied in this and previous papers. But there is no reason to think that these are the best. It is quite possible that better algorithms exist, even if we restrict ourselves to algorithms within our $O(n)$-pass framework.

References

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

[2] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Math. Prog.*, 85:277–311, 1999.

[3] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.

[4] A. Dasdan. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *ACM Trans. on Design Automation of Electronic Systems*, 9(4):385–418, 2004.

[5] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[6] C. Demetrescu, A.V. Goldberg, and D.S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http:// www.dis.uniroma1.it/~challenge9/, 2007.

[7] E. V. Denardo and B. L. Fox. Shortest–Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[8] R. B. Dial, F. Glover, D. Karney, and D. Klingman. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks*, 9:215–248, 1979.

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

Table 15: Incremental feasibility: Total number of scans on CAL.

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	2.0	2.0	13.3	2.0	2.0
	2	3.0	2.0	19.1	2.0	2.0
	64	65.0	2.0	327.4	2.0	2.0
	1024	1000.1	2.0	5125.3	2.0	2.0
-1	1	2.0	2.0	11.4	2.0	2.0
	2	3.0	2.0	17.2	2.0	2.0
	64	65.0	2.0	325.6	2.0	2.0
	1024	1000.1	2.0	5123.4	2.0	2.0
0	1	4.3	4.3	9.1	4.3	4.3
	2	7.3	7.3	16.0	7.3	7.3
	64	262.8	262.8	575.3	262.1	261.8
	1024	4144.8	4144.8	9079.3	4140.5	4136.8
1	1	2.7	2.7	6.7	2.7	2.7
	2	5.0	5.0	11.6	5.0	5.0
	64	172.1	172.1	391.2	171.8	171.8
	1024	2728.0	2728.5	6254.5	2728.0	2727.9

[10] L.R. Ford, Jr. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956.

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

[12] 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.

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

[14] A. V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. *SIAM J. Comput.*, 24:494–504, 1995.

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

[16] J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming*. John Wiley and Sons, 1980.

[17] S.G. Kolliopoulos and C. Stein. Finding Real-Valued Single-Source Shortest Paths in $o(n^3)$ Expected Time. In *Proc. 5th Int. Programming and Combinatorial Optimization Conf.*, 1996.

[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] M. Nonato, S. Pallottino, and B. Xuewen. SPT_L Shortest Path Algorithms: Reviews, New Proposals, and Some Experimental Results. Technical Report TR-99-16, Dipartmento di Informatica, Pisa University, Italy, 1999.

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

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

[22] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[23] R. E. Tarjan. Shortest Paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

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

[25] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1970.

[26] C-H. Wong and Y-C Tam. Negative Cycle Detection Problem. In *Proc. 13th Annual European Symposium on Algorithms*, pages 652–663, 2005.