Shortest-Path Feasibility Algorithms: An Experimental Evaluation

BORIS V. CHERKASSKY
Central Economics and Mathematics Institute of the Russian Academy of Sciences
LOUKAS GEORGIADIS
University of Western Macedonia
ANDREW V. GOLDBERG
Microsoft Research
ROBERT E. TARJAN
Princeton University
and
RENATO F. WERNECK
Microsoft Research

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 new algorithms) has very robust performance in practice. Our work advances the state of the art in the area.

Categories and Subject Descriptors: G.2.1 [**Discrete Mathematics**]: Graph Theory—Network Problems; G.2.1 [**Discrete Mathematics**]: Combinatorics—Combinatorial Algorithms

Authors' addresses: B. V. Cherkassky, Central Economics and Mathematics Institute of the Russian Academy of Sciences; email: bcherkassky@gmail.com. Part of this work has been done while this author was visiting Microsoft Research Silicon Valley; L. Georgiadis, Hewlett-Packard Laboratories, Palo Alto, CA, 94304. Current address: Informatics and Telecommunications Engineering Department, University of Western Macedonia, Kozani, Greece; email: lgeorg@uowm.gr; A. V. Goldberg and R. F. Werneck, Microsoft Research Silicon Valley, Mountain View, CA 94043; email: {goldberg,renatow}@microsoft.com; R. E. Tarjan, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08540 and Hewlett-Packard Laboratories, Palo Alto, CA, 94304; email: ret@cs.princeton.edu. Research at Princeton University partially supported by NSF grants CCF-0830676 and CCF-0832797. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.
DOI 10.1145/1498698.1537602 http://doi.acm.org/10.1145/1498698.1537602

General Terms: Algorithms, Experiments

Additional Key Words and Phrases: Shortest paths, negative cycles, feasibility

ACM Reference Format:

Cherkassky, B. V., Georgiadis, L., Goldberg, A. V., Tarjan, R. E., and Werneck, R. F. 2009. Shortest-path feasibility algorithms: An experimental evaluation. ACM J. Exp. Algor. 14, Article 2.7 (May 2009), 37 pages DOI 10.1145/1498698.1537602 http://doi.acm.org/10.1145/1498698.1537602

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 [Dijkstra 1959]. These are classical network optimization problems with many applications such as program verification [Tseitin 1970], real-time scheduling [Dechter et al. 1991], and circuit optimization [Chandrachoodan et al. 2001; Held et al. 2003; Dasdan 2004].

The classical Bellman-Ford-Moore (BFM) algorithm [Bellman 1958; Ford 1956; Ford and Fulkerson 1962; Moore 1959] 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 [1995] is an improvement, unless N is very large. The BFM bound can also be improved in the expected sense for many input distributions; see for example, [Kolliopoulos and Stein 1996].

Previous studies of shortest-path and feasibility algorithms include Pape [1974], Dial et al. [1979], Pallottino [1984], Gallo and Pallottino [1988], Cherkassky et al. [1996], Cherkassky and Goldberg [1999], Nonato et al. [1999], and Wong and Tam [2005]. The earliest experiments concluded that BFM performs poorly in practice and favored alternatives such as incremental graph algorithms [Pallottino 1984; Pape 1974] and the threshold algorithm [Dial et al. 1979]. These methods have relatively bad worst-case bounds, however, and are not robust in practice. More recently, Cherkassky and Goldberg [1999] showed that two $O(nm)$ algorithms, BFCT [Tarjan 1981] and GOR [Goldberg and Radzik 1993], are much more robust. These methods, which have been studied further in Nonato et al. [1999] and Wong and Tam [2005], 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 implemented several new algorithms and reimplemented existing ones, in one case correcting a heuristic used in previous implementations. Our experiments show that some of the new algorithms can significantly outperform previous ones on various problem classes; in particular, RD (robust Dijkstra) and HYB (a hybrid variant of BFCT) give very encouraging results.

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 in which one solves a sequence of feasibility problems, each a perturbation of the previous one. This incremental version of the feasibility problem has been studied before [Chandrachoodan et al. 2001] but restricted to BFCT.

This article 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 the existing algorithms used in our study, including improved implementations, and introduces some new methods. Section 6 describes families of instances that elicit the worst-case behavior of these algorithms. Section 7 describes our experimental setup and presents an empirical comparison of the various methods studied. Finally, concluding remarks are made in Section 8.

2. DEFINITIONS AND NOTATION

The inputs to FP are a directed graph $G = (V, E)$ and a length function $\ell : E \rightarrow \mathbf{R}$. 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 $\ell_d : E \rightarrow \mathbf{R} \cup \{-\infty, \infty\}$ as $\ell_d(v, w) = \ell(v, w) + d(v) - d(w)$. (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 in the context of the feasibility problem 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 [Ford 1956] for solving shortest-path and feasibility problems, on which all algorithms studied in

this article are based. (See, e.g., Tarjan [1983] 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 (v, w) such that $d(v) < \infty$ and $\ell_d(v, w) < 0$ and set $d(w) \leftarrow d(v) + \ell(v, w)$ and $p(w) \leftarrow v$. 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* $S(v) \in$ {unreached, labeled, scanned}. Given a labeled vertex v, SCAN(v) examines all arcs (v, w), and if $\ell_d(v, w) < 0$, 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 previously, 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 1. *If there is a shortest-path from s to v containing k arcs, then after at most k passes d(v) is exact. Therefore, 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. 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 cycle detection strategies. 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., Tarjan [1983]). 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 [Cherkassky and Goldberg 1999].

A more efficient solution is subtree disassembly, proposed by Tarjan [1981]. 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. A subtree can be disassembled in time proportional to its size, if we maintain a doubly linked list to represent a preorder traversal of the current shortest-path tree [Kennington and Helgason 1980]. Fortunately, this cost can be charged to the work of building the subtree, which keeps 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 [Cherkassky and Goldberg 1999]: If the potential of v decreases by δ, the 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 just $\delta - 1$, setting $d(w) \leftarrow d(w) - \delta + 1$. 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 [Goldberg 1995], which uses the fact that if $p(w) = v$, then (v, w) is in the admissible graph G_d. Therefore, if G_p contains a cycle, 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 in Section 5.4, 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. All other algorithms we tested use subtree disassembly with updates.

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.[1] 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 marked as unreached and skipped when they reach the front of the queue. BFCT runs in $O(nm)$ worst-case time.

5.2 Stack-Based Algorithms

A natural alternative implementation of the $O(n)$-pass algorithm is to use stacks instead of queues. We tested three variants, all of which maintain two stacks of labeled vertices, one for the current pass (S) and another for the next pass (T). When S becomes empty, we set $S \leftarrow T$, reset T, and start a new pass.

The first algorithm is the most similar to BFCT: It always adds newly labeled vertices to T. We call this algorithm LS, since it performs no more than a linear number of passes and uses a stack.

The second stack-based algorithm, LSG, adopts a greedier approach, in the sense that it tries to scan labeled vertices as early as possible. If a newly-labeled vertex has not yet been scanned in the current pass, it is inserted into S. If it has been already scanned, it must be inserted into T (or else one cannot guarantee that the number of passes will be linear). In contrast with LS and BFCT, this algorithm requires that we explicitly keep track of the last pass in which each vertex was scanned.

Our third stack-based algorithm is even greedier. Like LSG, newly labeled vertices are inserted into S if possible, and into T if already scanned in the current iteration. Moreover, when there is an improvement in the potential $d(v)$ of a vertex v that is already in S (meaning it is labeled and scheduled to be scanned in the current pass), v is moved to the top of S. Note that if every arc scan improved the potential of its head and there were no subtree disassembly, vertices would be scanned in depth-first order. For this reason, we call this algorithm DFS.

The rationale for DFS is that it can greedily follow a negative cycle. As Section 7 will show, DFS is indeed very fast at detecting obvious cycles. In general, however, it seems to be much less robust than BFCT. In an attempt to

[1]In Cherkassky and Goldberg [1999], 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.

combine the strengths of both methods, we also tried a hybrid algorithm (HYB). It works as DFS during the first pass only, then behaves as BFCT. At the end of the first pass, stack T (created by DFS) becomes the first queue used by BFCT (the bottom of the stack becomes the front of the queue). All four stack-based algorithms (LS, LSG, DFS, and HYB) use subtree disassembly with updates and run in $O(nm)$ time in the worst case.

5.3 Randomized Algorithms

The third class of algorithms we tested processes labeled vertices in random order, while still subject to the restrictions that ensure they are $O(n)$-pass methods. The algorithms maintain two sets: S contains vertices to be scanned in the current pass and T those to be scanned in the next pass. The next vertex to be scanned is picked uniformly at random from S; when S becomes empty, a new pass starts by setting $S \leftarrow T$ and resetting T.

We considered two versions of the randomized algorithm. The standard version, LR, always inserts newly-labeled vertices into T. The greedy version, LRG, inserts a newly-labeled vertex into S if it has not yet been scanned in the current pass and into T otherwise. Both algorithms use subtree disassembly with updates and run in $O(nm)$ worst-case time.

5.4 GOR

Proposed by Goldberg and Radzik [1993], GOR is an $O(n)$-pass algorithm that uses a more sophisticated processing order in each pass. It 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 suggested by Cherkassky and Goldberg [1999] avoids both the contraction and the SCC computation by running a simple depth-first search (DFS). When a back arc is discovered, the algorithm 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 cycle in G_d, which could cause the algorithm to terminate later or not at all (although the latter never happened in the experiments reported by Cherkassky and Goldberg [1999] and Wong and Tam [2005]). 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 [1972], 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 it 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 this 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.

5.5 RD

We now propose another $O(n)$-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 with 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, 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). This may not work well in practice because a potential transformation may affect the vertex selection order in every pass. For example, if for some vertex v we decrease the lengths of its incoming arcs by a large value and increase the lengths of its outgoing arcs by the same amount, v will always be scanned immediately after it is added to Q.

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

As its name suggests, RD is a generalization of Dijkstra's algorithm: On the shortest-path problem, both 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 [Denardo and Fox 1979]; a general (and potentially less efficient) data structure must be used instead. Using Fibonacci heaps [1987], the algorithm runs in $O(n(m + n \log n))$ time. Our implementation, which we call RDH, uses a 4-heap [Tarjan 1983] (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.

5.6 MBFCT

The MBFCT algorithm [Wong and Tam 2005] is a "local" variant of BFCT. Initially, all vertices are unreached and have zero potential. The following procedure is executed while there are unreached vertices: Pick an unreached vertex v, mark it labeled, set $p(v) \leftarrow s$, 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 scanned in each new 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 [Wong and Tam 2005], 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 [Pallottino 1984], which also works "locally" using the BFM algorithm. It assigns priorities to all vertices (either low or high). Every vertex starts with low priority, and priorities can only change from low to high. Labeled vertices are maintained in two FIFO queues, L and H, according to their priority (low and high, respectively). Initially, H is empty and L contains only s.

The algorithm works in phases. At the beginning of each phase, it removes a single vertex v from the head of L, changes the priority of v to high, and adds v to H (which is initially empty). The algorithm then proceeds to scan the

[3]We also tested buckets as stacks, but the results were less robust.

vertices in H. Each scan may add vertices to either H or L, depending on the priority of the labeled vertex (which does not change). The phase terminates when H becomes empty. At this point, the current potentials are feasible for the subgraph induced by the set of high-priority vertices.

Each phase of Pallottino's algorithm is analogous to a call to BFCT within MBFCT. The main differences between MBFCT and Pallottino's algorithm are that MBFCT does not restrict scans to high-priority vertices (every labeled vertex has high priority) and uses subtree disassembly (though Pallottino's algorithm could also use it, as shown in Cherkassky and Goldberg [1999]). Both methods 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.

5.7 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 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 Goldberg [1995]. To gain intuition, suppose the input graph has 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 consisting of (v, w) and the 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 AF. 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

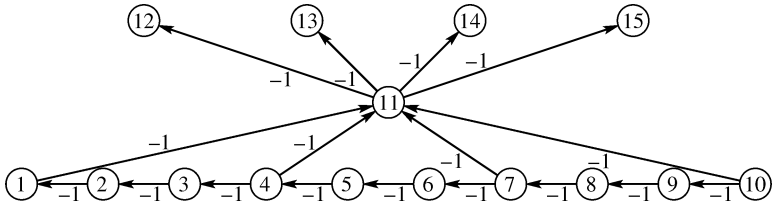

Fig. 1. Worst-case input for BFCT ($k = 4$).

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. Similarly, the original scaling algorithm by Goldberg [1995], which has good worst-case bounds but is even more complicated, was not included in our experiments.

6. WORST-CASE INSTANCES

This section introduces graph families that elicit the worst-case behavior of some of the algorithms we implemented. Although each example was designed for one specific algorithm, Section 7 will show that in many cases other algorithms have bad behavior as well. In particular, every algorithm we tested exhibits its worst-case behavior on at least one of the families described here.

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. 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.

For consistency, we implemented all of our algorithms so that vertices appear in increasing order of identifier in the initial queue (this is relevant for some of the worst-case families). In particular, the first vertex to be processed is vertex 1 (assuming vertices are numbered from 1 to n). To achieve this, vertices are initially inserted into the queue in increasing order of identifier, except for the stack-based algorithms (LS, LSG, DFS, and HYB), which insert vertices into the initial stack in decreasing order.

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 $3k - 1$, that is, we have the arcs $(3(i - 1) + 1, 3k - 1)$ for $1 \leq i \leq k$. Finally, vertex $3k - 1$ is connected to vertices $3k$ to $4k - 1$. All arcs in this graph have length -1. Figure 1 gives an example for $k = 4$.

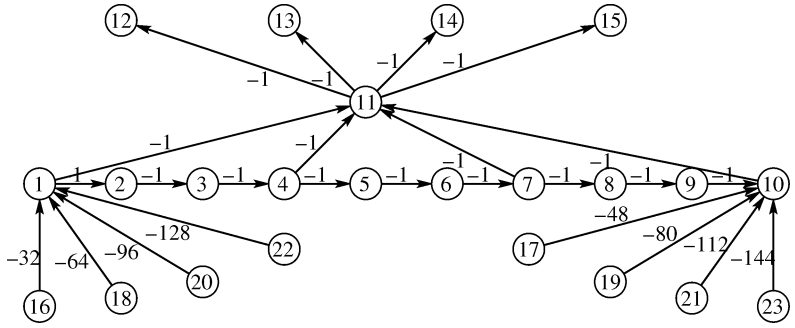

Fig. 2. Worst-case input for MBFCT ($k = 4$).

Recall 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 connected 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 $m = 7k-3$.

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-path algorithm [Pallottino 1984].

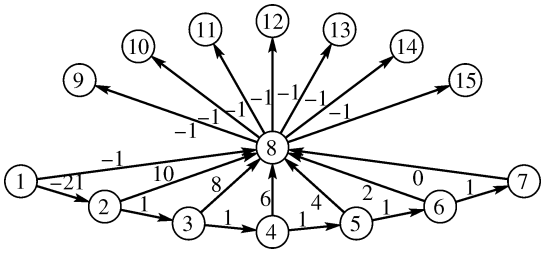

Fig. 3. Worst-case input for GOR ($k = 7$).

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 $2 \le i \le k-1$. Also, $\ell(k+1, k+1+i) = -1$ for $1 \le i \le k$, and $\ell(i, k+1) = 2(k-i)$ for $2 \le i \le k$. We have $n = 2k+1$ and $m = 3k-1$. Figure 3 gives an example for $k = 7$.

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 for this algorithm. 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 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})

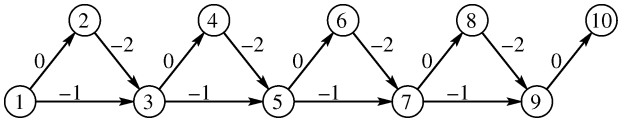

Fig. 4. Worst-case input for RD ($k = 5$) when the priority queue returns the most recently inserted vertex among all those with maximum key.

is processed after (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 $\Theta(k^2) = \Theta(n^2)$.

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.

The experimental results in Section 7.2 (Table VIII) suggest that our implementation of RDH also performs a quadratic number of scans on the graph of Figure 5. We refer to 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 $i < j$. The experimental results suggest that RDH also 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.

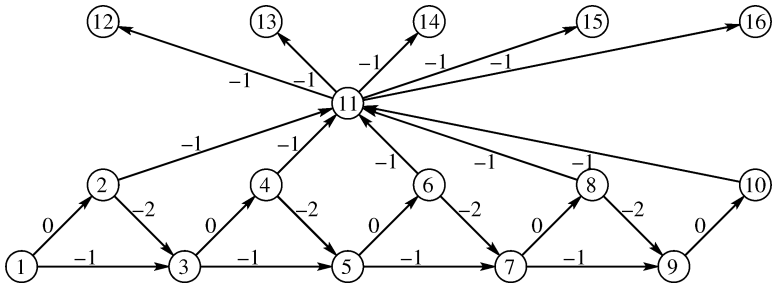

Fig. 5. Worst-case input for RDB and RDH ($k = 5$).

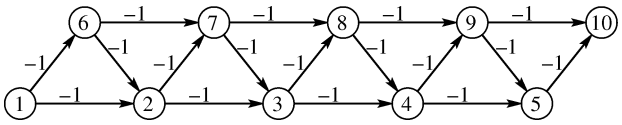

Fig. 6. Worst-case input for DFS and LSG ($k = 5$).

6.5 DFS and LSG

We construct a worst-case family for both DFS and LSG by augmenting the network $G(k)$ of Figure 4 with the arcs (y_i, y_{i+1}) for $1 \leq i < k$. For simplicity, we set all arc lengths to -1. Also, since LSG does not change the position of the vertices in the stack S (and because initially all vertices are pushed into S in decreasing order of identifier), we change the vertex identifiers so that LSG does not build the correct shortest-path tree in a single pass. To that end, we set $x_i = i$ and $y_i = k + i$, $1 \leq i \leq k$. The resulting network, shown in Figure 6 for $k = 5$, has $n = 2k$ and $m = 4k - 3$. We call this family BAD-DFS(k).

Notice that, as in $G(k)$, each path (x_i, y_i, x_{i+1}) is shorter than (x_i, x_{i+1}). Moreover, each path (y_i, x_{i+1}, y_{i+1}) is shorter than (y_i, y_{i+1}). Our goal is similar to the one we had for $G(k)$; we want to force the stack-based algorithm (DFS or LSG) to include in the tentative shortest-path tree the arcs (x_i, x_{i+1}) and (y_i, y_{i+1}) before (x_i, y_i) and (y_i, x_{i+1}). To that end, we order the adjacency lists so that y_i precedes x_{i+1} in the successor list of x_i, and x_{i+1} precedes y_{i+1} in the successor list of y_i. The effect is that when x_i (respectively, y_i) is scanned x_{i+1} (y_{i+1}) will be placed on top of y_i (x_{i+1}) in the stack (assuming that both become labeled). This results in $\Theta(k)$ passes, each consisting of $\Theta(k)$ scans.

6.6 Arc-Fixing Algorithm

Like COMP-DAG, this is a 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 arc lengths alternate between -1 and 1. Then we add k vertices, $2k + 3 \ldots 3k+2$. For $0 \leq i \leq k-1$, vertex $2k+3+i$ has an incoming arc of length $i-k-1$ from $2i + 1$, and $2(k - i)$ unit-length outgoing arcs to $2i + 3, 2i + 4, \ldots, 2k + 2$. Figure 7 gives an example for $k = 3$.

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

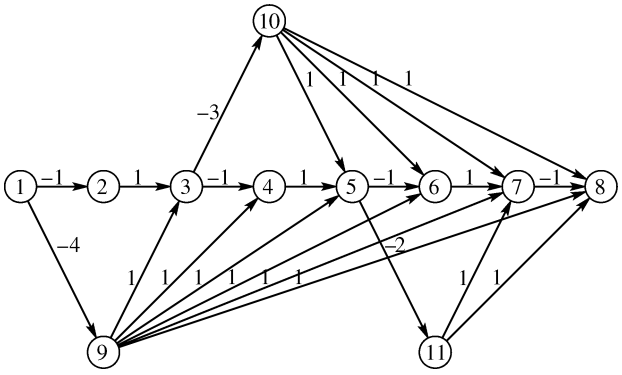

Fig. 7. Worst-case input for the arc-fixing algorithm ($k = 3$).

on the path $P_i = (2i + 3, 2i + 4, \ldots, 2k + 2)$ have the same potential, 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 $\Omega(nm)$.

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 Wong and Tam [2005] present encouraging results. Finally, we tested the new $O(n)$-pass algorithms introduced in this article: stack-based (LS, LSG, and DFS), hybrid (HYB), randomized (LR and LRG), and robust Dijkstra (RDH and RDB). For succinctness, however, we omit detailed results for LR (which is usually quite similar to LRG) and LS and LSG (since DFS and HYB make better use of the stack).

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 (those performed by GOR during its DFS phase). Unless otherwise noted, the results reported are averages taken over 10 instances with different pseudo-random generator seeds.

We also report some running times to measure the data structure overhead for different algorithms. Our experiments were run on a 2.4 GHz AMD Opteron with 16 GB of RAM running Windows Server 2003 Enterprise x64 Edition. 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 Microsoft Visual Studio 2008 using full optimization (`/Ox`). Arc lengths and potentials were represented as 64-bit signed integers. The code is available from the authors upon request, as are the graph generators.

7.2 Feasibility

We start with the standard (nonincremental) feasibility problem. As observed by Cherkassky and Goldberg [1999], 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 arc with length -1. From each base graph family, we create five subfamilies, defined by the number of negative cycles added: (01) no extra cycles; (02) one small cycle, with three arcs; (03) $\lfloor\sqrt{n}\rfloor$ small cycles, with three arcs each; (04) $\lfloor\sqrt[3]{n}\rfloor$ medium cycles, with $\lfloor\sqrt{n}\rfloor$ arcs each; and (05) one Hamiltonian cycle. These families are similar (but not identical) to those proposed by Cherkassky and Goldberg [1999].

After adding the negative cycles (or not, in case 01), we apply a potential transformation to "hide" them. A potential transformation with parameter P selects, for each vertex v, an integer $d(v)$ uniformly at random from $[0, P)$, 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. We use $P = 16{,}384$ in most cases (exceptions will be mentioned explicitly).

Unless otherwise noted, we further randomize the instances by permuting their vertex identifiers and the order in which the outgoing arcs appear in each adjacency list. This prevents the algorithm from exploiting any structure in the code used to generate the underlying graph or in the NEGCYCLE filter. In particular, NEGCYCLE adds the cycle arcs to the end of the adjacency lists; without the random permutation, DFS could find the "hidden" Hamiltonian cycle in the 05 subfamily within a single pass. In the preliminary version of this work [Cherkassky et al. 2008], we did not apply this additional randomization step, which helps explain the occasionally different results reported here.

Our testbed includes the families tested by Cherkassky and Goldberg [1999], augmented with road networks, circuits from real-world applications, and worst-case examples. We discuss each family in turn.

7.2.1 *Random Graphs.* The SPRAND generator [Cherkassky et al. 1996] 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 $L = 1$ 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 I 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, but only detects a cycle when processing the SCCs in the second pass. For this family, methods that tend to concentrate

Table I. Feasibility: Scans per Vertex on RAND5

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,144	2.0801	11.9908	3.9125	2.8660	2.4548	2.1479	2.4110	2.3899
	524,288	2.0792	12.1495	3.9130	2.8631	2.4547	2.1474	2.4146	2.3883
	1,048,576	2.0813	12.4130	3.9164	2.8648	2.4562	2.1486	2.4286	2.3939
	2,097,152	2.0806	13.2888	3.9146	2.8644	2.4558	2.1487	2.4121	2.3889
02	262,144	1.6828	1.5615	3.4113	2.0105	1.8577	1.1919	2.0424	1.8287
	524,288	1.5691	5.7760	3.3140	1.7042	1.6300	1.4795	1.7960	1.8865
	1,048,576	1.6060	4.6504	3.1287	1.7845	1.6693	1.4521	1.6260	1.8751
	2,097,152	1.7627	5.5588	3.5598	1.6493	1.8905	1.3903	1.5137	1.9966
03	262,144	0.2232	0.0103	2.3513	0.0078	0.0078	0.2594	0.0096	0.0097
	524,288	0.1940	0.0074	2.3484	0.0074	0.0074	0.2088	0.0074	0.0051
	1,048,576	0.1869	0.0081	2.3470	0.0035	0.0035	0.1980	0.0037	0.0057
	2,097,152	0.1363	0.0031	2.3473	0.0035	0.0035	0.1687	0.0046	0.0043
04	262,144	5.6553	20.1334	9.9471	4.8565	6.0416	5.2578	4.5227	4.2814
	524,288	5.8202	19.9483	10.2131	4.9389	6.2102	5.3185	4.6616	4.4606
	1,048,576	5.9639	21.1991	10.3760	4.9770	6.3485	5.4565	4.7902	4.4890
	2,097,152	6.0911	23.1698	10.6355	5.0080	6.4769	5.5843	4.8873	4.5927
05	262,144	11.1471	50.5397	23.0874	13.5330	11.5744	11.6743	9.8453	9.4198
	524,288	11.2032	46.0774	23.1960	13.6636	11.6334	11.7329	9.8968	9.5024
	1,048,576	11.2033	47.6920	23.0841	13.6307	11.6322	11.7268	9.9189	9.5416
	2,097,152	11.2704	53.8311	23.2688	13.7329	11.7002	11.7986	9.9719	9.5697

on a small region of the graph (MBFCT, DFS, HYB, RDB, and RDH) do better than BFCT and LRG, which take a more global approach. For subfamilies with longer or nonexisting negative cycles (01, 04, and 05), MBFCT is significantly worse than other methods.

A similar experiment reported in an earlier version of this article [Cherkassky et al. 2008] showed significantly better performance for MBFCT: it was competitive on all subfamilies, and often the best overall. The discrepancy is due to the fact that, in the previous article, the average potential perturbation (500) was significantly smaller than the average original arc length (16,000).[5] Because perturbations are on average much higher than arc lengths in the experiment reported in Table I, cycles can be much better hidden.

To assess the importance of potential perturbations, we performed the following experiment (also on RAND5): We fixed the range from which arc lengths are picked ([1,1,000], as before), but varied the amount of potential perturbation applied. Potentials were integers picked uniformly at random from the range $[0, P)$, where P varied from 1 (no perturbation) to 2^{21}. No negative cycles were added. Figure 8 reports the average number of scans per vertex on graphs with 65,536 vertices. Each data point is the average of 100 executions for MBFCT, and 10 for the remaining algorithms (which have lower variance).

All algorithms behave very similarly when perturbations are small. Because very few negative arcs are created (if at all), the initial set of potentials (zero) is likely to be feasible; even if it is not, it can be easily fixed. As the perturbations

[5]According to Cherkassky et al. [2008], both the original arc lengths and the potential perturbations were random numbers between 0 and 1,000. This description was incorrect, unfortunately. Due to a mistake in our scripts, the original arc lengths were actually picked from a much wider range (0 to 32,000).

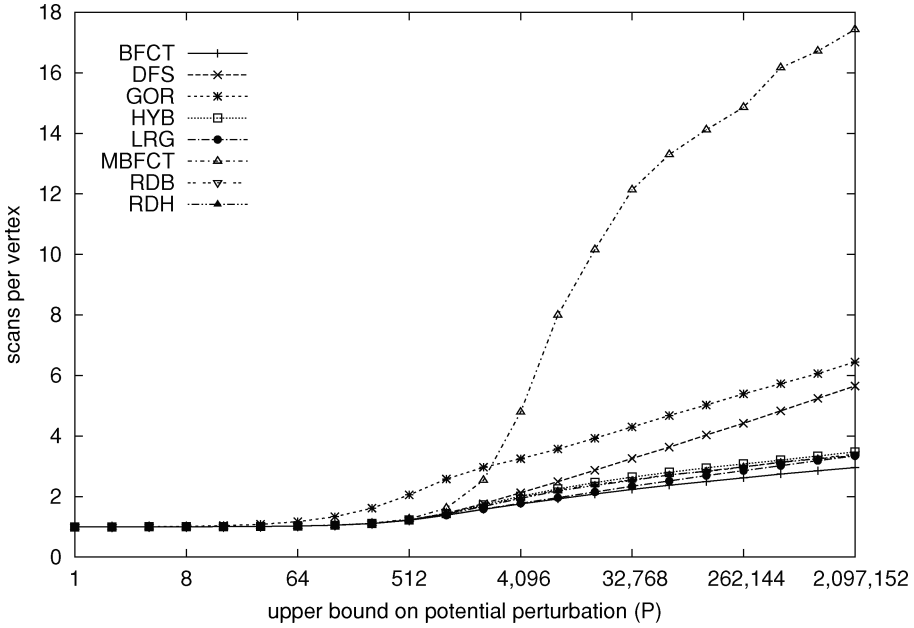

Fig. 8. Feasibility on RAND5 with $n = 65{,}536$, original arc lengths in $[1, 1{,}000]$, and integral potential perturbations in $[0, P)$, with P variable.

increase, there is more work to do, and one can observe a separation between the algorithms. Eventually, the fastest is BFCT, followed closely by LRG, RDH, RDB, and HYB. Both DFS and GOR are slower, but by a factor of less than 2. The MBFCT algorithm, however, is significantly slower. Recall that, instead of dealing with the entire graph at once, MBFCT works by "fixing" a subgraph (i.e., eliminating its negative arcs) that gradually increases in size. When there are many negative arcs, MBFCT often has to redo portions of the graph it has already dealt with.

This is not always the case, however, if the current subgraph happens to have a negative cycle, the algorithm can stop after visiting a small portion of the entire graph. Figure 9 is similar to Figure 8, but considers original graphs from subfamily 04 of RAND5: To the original random graph with $n = 65{,}536$ vertices, we add 40 negative cycles, each with 256 arcs and total length -1. Then we add integral potential perturbations in $[0, P)$, for various values of P.

When perturbations are small, the negative cycles are obvious enough to allow MBFCT to direct the search towards them. No other algorithm can do it as effectively. GOR is competitive when there are no perturbations at all: as soon as it finds a zero or negative arc, it simply follows it to find the cycle. Even a tiny perturbation, however, makes it worse than most other algorithms. As in the previous experiment, MBFCT becomes the worst algorithm when perturbations are very high. In such cases, RDB and RDH are the best at locating the hidden cycles.

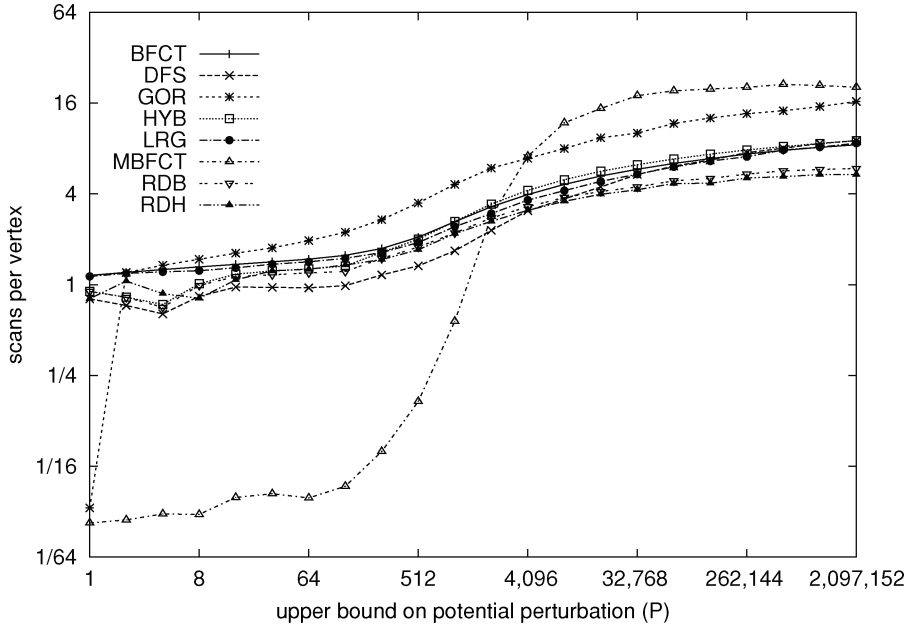

Fig. 9. Feasibility on subfamily 04 of RAND5 with $n = 65{,}536$, original arc lengths in [1, 1,000], and integral potential perturbations in $[0, P)$, with P variable.

The results of our next experiment, also on random graphs, are shown in Figure 10. Here we fix $n = 65{,}536$, $m = 5n$, the maximum arc length U at 1,000, and vary the minimum allowed arc length L from 0 to $-1{,}200$. We apply potential perturbations in [0, 2,000) to the resulting graph (this is not done in Cherkassky and Goldberg [1999], 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.

In this experiment, the algorithms exhibit a wide range of different behaviors. When L is very small, negative cycles are numerous and easy to find. Some algorithms do better than others, and the worst algorithms (BFCT, LRG) are two orders of magnitude slower than the best (RDH, HYB, DFS).[6] Note that the plots for HYB and DFS are identical in this range. In this experiment, RDB is significantly worse than RDH, which is similar but uses a heap instead of buckets. Since buckets are organized as queues, RDB tends to expand many candidate paths at once. In fact, organizing buckets as stacks (which tends to favor the most recently improved path) would make the algorithm about as fast as RDH in this experiment, but the stack-based implementation is less robust, in general.

As L increases, the cycles become fewer and harder to find. GOR, which is not far from RDH when L is very negative, becomes much slower around

[6]In the preliminary version of this article Cherkassky et al. [2008], BFCT showed significantly better performance for small values of L. This was an artifact of using graphs produced by the SPRAND generator without further randomization of the adjacency lists.

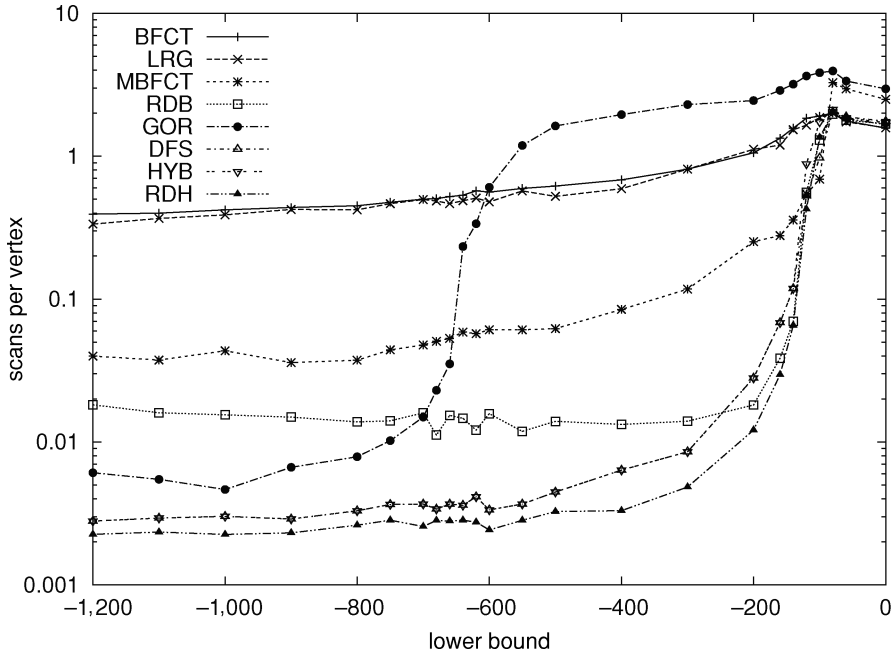

Fig. 10. Feasibility on random graphs with 65,536 vertices and arc lengths in $[L, 1{,}000]$, with L variable.

$L = -650$. This happens because the cycle-detection heuristic (which looks for negative back arcs during the depth-first search) ceases to be effective at this point. The performances of RDH, HYB, and DFS degrade as L increases, and the rate of the degradation becomes very high around $L = -200$. At this point, RDB suddenly becomes much worse as well. When L approaches zero, there are no negative cycles, and all algorithms perform similarly. In particular, LRG and BFCT are competitive for larger values of L. The three algorithms that were omitted for clarity (LS, LSG, and LR) perform similarly to LRG and BFCT in this experiment.

7.2.2 *Grids.* We also tested the algorithms on *grid networks*, which are grids embedded on a torus, created by the TOR generator [Cherkassky and Goldberg 1999]. 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 interlayer 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 II shows the average number of scans per vertex on square grids with negative cycles (SQNC), for which $X = Y$. All five subfamilies are shown.

The results obtained for SQNC, are in general, similar to those reported for RAND5, even though these families are very different. The relative

Table II. Feasibility: Scans per Vertex on SQNC

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,145	2.4209	2.3925	4.8671	2.2968	2.3226	2.4347	2.2167	2.1981
	524,177	2.4207	2.3926	4.8682	2.2949	2.3220	2.4356	2.2152	2.1977
	1,048,577	2.4205	2.3922	4.8671	2.2947	2.3209	2.4355	2.2144	2.1988
	2,096,705	2.4205	2.3918	4.8664	2.2954	2.3216	2.4353	2.2151	2.1988
02	262,145	1.2979	0.3768	3.1997	0.5927	0.7354	1.4121	0.7768	0.8168
	524,177	1.1159	1.0358	3.2105	1.3526	1.2924	1.3672	1.3278	1.2856
	1,048,577	1.3704	0.8742	3.2868	0.5844	0.5913	1.2135	0.7043	1.1767
	2,096,705	1.4918	0.6977	3.4331	1.0836	0.9909	1.4819	1.0231	1.1641
03	262,145	0.1858	0.0009	2.4577	0.0009	0.0009	0.2228	0.0009	0.0009
	524,177	0.1749	0.0007	2.4570	0.0010	0.0010	0.2130	0.0006	0.0012
	1,048,577	0.1761	0.0006	2.4571	0.0008	0.0008	0.1659	0.0008	0.0006
	2,096,705	0.1467	0.0004	2.4561	0.0005	0.0005	0.1481	0.0005	0.0003
04	262,145	5.3039	12.4204	11.0803	4.9039	5.2407	5.1909	4.3577	4.2227
	524,177	5.4475	11.9695	10.9137	4.8551	5.3805	5.2826	4.3692	4.2301
	1,048,577	5.5580	14.0709	11.2742	5.0103	5.4845	5.3072	4.4771	4.3096
	2,096,705	5.6350	13.2938	11.2562	5.1690	5.5562	5.4216	4.5850	4.3157
05	262,145	11.2065	44.3661	23.5564	13.8343	11.3739	11.7684	9.6032	9.2305
	524,177	11.1888	40.6998	23.6993	13.7763	11.3581	11.7238	9.6223	9.2340
	1,048,577	11.2013	48.6196	23.6040	13.8019	11.3696	11.7546	9.6887	9.2558
	2,096,705	11.2533	46.2369	23.4866	13.8848	11.4219	11.8086	9.7120	9.3187

Table III. Feasibility: Scans per Vertex on LNC

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,145	2.2514	2.3019	4.5180	2.1751	2.1690	2.2710	2.0835	2.0678
	524,289	2.2524	2.3045	4.5181	2.1762	2.1688	2.2716	2.0836	2.0685
	1,048,577	2.2519	2.3029	4.5185	2.1750	2.1682	2.2720	2.0829	2.0688
	2,097,153	2.2528	2.3037	4.5187	2.1758	2.1690	2.2718	2.0838	2.0687
02	262,145	1.3791	0.4109	3.0544	0.7963	0.8310	1.4280	0.8209	0.9285
	524,289	1.3026	1.2579	2.8831	1.1385	1.0462	1.4918	1.0549	0.9211
	1,048,577	1.4364	0.8030	3.2185	0.9784	0.8914	1.4350	0.9596	0.8919
	2,097,153	1.3063	1.0213	3.2486	1.0355	0.8952	1.5064	0.9923	1.0496
03	262,145	0.1858	0.0019	2.4570	0.0018	0.0018	0.2240	0.0020	0.0014
	524,289	0.1729	0.0011	2.4565	0.0010	0.0010	0.2376	0.0011	0.0011
	1,048,577	0.1748	0.0008	2.4566	0.0007	0.0007	0.1818	0.0007	0.0004
	2,097,153	0.1156	0.0008	2.4560	0.0007	0.0007	0.1374	0.0008	0.0005
04	262,145	5.2140	12.1350	10.5074	4.6037	5.1557	5.0291	4.2473	4.0790
	524,289	5.2997	13.4848	10.8646	4.8070	5.2328	5.0487	4.3098	4.0907
	1,048,577	5.3776	14.0994	10.7276	4.8250	5.3062	5.1960	4.3318	4.1451
	2,097,153	5.4339	14.0978	10.8719	4.7868	5.3560	5.2082	4.3781	4.2188
05	262,145	11.1969	44.5079	23.1050	13.8530	11.3653	11.7650	9.5825	9.2099
	524,289	11.2029	49.4086	23.8273	13.8486	11.3717	11.7626	9.6642	9.2114
	1,048,577	11.2026	48.9128	23.3418	13.8340	11.3707	11.7456	9.7022	9.2837
	2,097,153	11.2520	53.5594	23.3676	13.8973	11.4208	11.8131	9.7429	9.3248

performance of the algorithms is largely unchanged. An important exception is MBFCT, which is now the fastest on subfamily 02 and competitive on subfamily 01. Also note that, for greedy algorithms, subfamily 03 is even easier on SQNC.

Experiments with long grids with negative cycles (LNC), which have $X = n/16$ layers with $Y = 16$ vertices each, yielded results almost identical to SQNC, as Table III shows.

In these first three families (and others to come), LRG and BFCT have very similar performance to each other on all subfamilies. This is also true for LS

Table IV. Feasibility: Scans per Vertex on PNC

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,145	2.6785	6.7272	6.0453	3.7464	3.0162	2.8288	2.8797	2.6988
	524,289	2.6776	6.7303	6.0411	3.7470	3.0144	2.8264	2.8789	2.6979
	1,048,577	2.6757	6.7358	6.0380	3.7458	3.0131	2.8265	2.8777	2.6979
	2,097,153	2.6761	6.7299	6.0368	3.7441	3.0124	2.8251	2.8772	2.6973
02	262,145	1.9509	1.2152	4.5881	1.2253	1.5517	1.7459	1.6015	1.5018
	524,289	1.6425	2.9316	4.2214	1.1340	1.4760	1.9497	1.4688	1.2638
	1,048,577	2.0742	1.5089	4.3901	0.8192	1.1957	1.7554	1.3148	1.1813
	2,097,153	1.6851	2.3187	3.4103	1.2984	1.8822	1.8841	1.4532	1.6440
03	262,145	0.2266	0.0020	2.3870	0.0014	0.0014	0.2405	0.0019	0.0009
	524,289	0.2085	0.0025	2.3837	0.0017	0.0017	0.1963	0.0016	0.0010
	1,048,577	0.2047	0.0010	2.3827	0.0006	0.0006	0.1504	0.0010	0.0007
	2,097,153	0.1248	0.0019	2.3820	0.0011	0.0011	0.1386	0.0010	0.0008
04	262,145	7.5186	16.7286	15.4599	9.6573	7.8784	7.9337	6.6909	6.2885
	524,289	7.7681	21.6253	15.9993	10.0136	8.1280	8.0624	7.1345	6.3621
	1,048,577	8.0138	22.7582	16.5523	9.9991	8.3593	8.3418	7.0317	6.8063
	2,097,153	8.1529	24.8943	16.5881	10.2787	8.5053	8.5000	7.2177	6.7976
05	262,145	7.2984	32.3062	16.2659	10.9824	7.7079	7.7300	7.4562	7.2972
	524,289	7.2811	34.6342	16.3961	11.0009	7.7081	7.7141	7.5733	7.3018
	1,048,577	7.2966	35.5280	16.2828	10.9999	7.7162	7.7218	7.5740	7.3265
	2,097,153	7.3196	35.1856	16.8812	11.0496	7.7500	7.7599	7.6153	7.3546

and LR, not shown in the tables. Note that BFCT, LS, and LR are not greedy, in the sense that vertices labeled in one pass are only scanned in the next. (LRG is greedy, but it does not exploit this fact in any systematic way, and is, therefore, very similar to LR.) The fact that these methods behave similarly indicates that, for these nonadversarial families, using stacks or queues is no better than processing vertices in random order in each pass. Even picking vertices in topological order, as GOR does, is not very helpful; GOR performs roughly twice as many scans in total on most subfamilies (and does even worse on subfamily 03). Considering that each vertex is scanned twice in each pass, this means that the total number of non-DFS scans is roughly the same as in BFCT, LR, or LS. Of course, as Section 7.2.6 will show, there are instances for which these algorithms have very distinct behavior.

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]. A special source is connected to all vertices in the first layer by arcs with length in [1, 100,00]. In addition, vertices in each layer have five arcs to forward layers (including the next one). The length of an interlayer arc that goes x layers forward is picked uniformly at random from [1, 10,000] and multiplied by x^2. As a result, a shortest-path tree rooted at the source will be very deep (using short forward arcs), but long forward arcs induce single-source shortest-path algorithms to explore numerous shallower trees first (the effect should not be as pronounced for feasibility algorithms). As in previous families, we introduce negative cycles with the NEGCYCLE filter. Following Cherkassky and Goldberg [1999], we call this family PNC (p-hard with negative cycles). Table IV reports the average number of scans per vertex on PNC.

Table V. Feasibility: Scans per Vertex on CAL

FAM	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	1.5300	1.8861	3.0153	1.6398	1.5994	1.5403	1.5764	1.5677
02	1.1984	0.8607	2.6672	1.1507	1.1597	1.2481	1.1010	1.0382
03	0.1454	0.0006	2.4564	0.0005	0.0005	0.1490	0.0005	0.0006
04	3.0233	0.8795	5.4551	2.0740	3.0693	2.8107	2.5633	2.5108
05	8.7371	22.0675	19.5446	9.2412	8.9203	8.8653	7.9956	7.8242

Note that, when no cycles are present (subfamily 01), the performance of MBFCT is somewhat worse on PNC than on grids (SQNC and LNC). Also compared to grids, subfamily 04 is slightly harder for all algorithms, subfamily 05 is slightly easier, and there is a clearer separation between DFS and HYB on subfamilies with longer cycles. The hybrid algorithm seems to be more robust than DFS for harder instances, and it converges faster.

7.2.4 *Road Networks.* We also tested our algorithms on road networks, available at the 9th DIMACS Implementation Challenge web site [Demetrescu et al. 2007]. 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 arc lengths. Results for the CAL instance, which represents California and Nevada and has 1.9 million vertices, are shown in Table V. (Results for other road networks were very similar.) Once again, GOR is consistently slower than the average algorithm. MBFCT is among the fastest on easier instances (with small or numerous cycles), but slower when there are longer negative cycles. When there are no cycles (subfamily 01), all algorithms (except GOR) have similar performance.

7.2.5 *Circuits.* We now deal with real-world graphs from circuit design applications. These are actually instances for the minimum cycle mean problem (MCM), whose goal is to find the cycle C whose mean length (the sum of its arc lengths divided by the number of arcs in C) is minimum. This is equivalent to determining the constant λ, which, when subtracted from the length of each arc, yields a graph with no negative cycles but at least one zero cycle. Good algorithms for this problem share the same basic approach: guess a value of λ, determine whether it leads to a negative cycle (using a feasibility algorithm), and adjust λ, accordingly. Dasdan [2004] presents an overview of many such algorithms (see also Georgiadis et al. [2009]).

A detailed analysis of MCM algorithms would be out of scope for this article, so we abstract them away as follows. First, we multiply the original arc lengths by 10^5 to work with integers only. Then, we run the feasibility algorithms on each MCM instance twice, with distinct values of λ. The first, λ^+, is the largest integer value of λ that results in a feasible instance; the second, λ^-, is the smallest integer value of λ leading to a graph with at least one negative cycle. Note that $\lambda^- = \lambda^+ + 1$. Dividing either value by 10^5, we get an additive approximation of 10^{-5} to the MCM of the original graph (before arc lengths are multiplied).

We tested two families of MCM instances, both representing circuits. In the BONN family, kindly provided by Stephan Held, the MCM of a graph

Table VI. Feasibility: Circuits with No Negative Cycle

GRAPH	n	m	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
bonn01	4,609	2,916,202	1.0086	1.0159	1.0264	1.0126	1.0124	1.0093	1.0115	1.0112
bonn02	5,361	4,166,868	1.1177	1.4555	1.2959	1.1876	1.1813	1.1306	1.1575	1.1500
bonn03	11,867	8,914,917	1.0693	1.2500	1.1646	1.1146	1.1139	1.0693	1.0856	1.0719
bonn04	20,072	1,321,426	1.0294	1.0689	1.0487	1.0372	1.0368	1.0278	1.0385	1.0337
bonn05	60,309	3,969,429	1.0101	1.0347	1.0377	1.0159	1.0158	1.0103	1.0158	1.0163
bonn06	70,346	892,588	1.0069	1.0154	1.0189	1.0101	1.0095	1.0070	1.0090	1.0087
bonn07	82,038	2,603,211	1.0381	1.0716	1.1026	1.0529	1.0500	1.0353	1.0466	1.0423
bonn08	95,936	1,324,128	1.1173	1.3915	1.2746	1.1984	1.1911	1.1247	1.1929	1.1915
bonn09	346,814	9,222,546	1.0053	1.0097	1.0140	1.0067	1.0067	1.0053	1.0065	1.0063
ibm01	12,752	36,681	1.0423	1.0439	1.2645	1.0424	1.0424	1.0433	1.0424	1.0436
ibm02	19,601	61,829	1.0113	1.0114	1.0718	1.0113	1.0113	1.0110	1.0113	1.0104
ibm03	23,136	66,429	1.0116	1.0117	1.0758	1.0116	1.0116	1.0117	1.0116	1.0115
ibm04	27,507	74,138	1.0056	1.0056	1.0396	1.0056	1.0056	1.0059	1.0056	1.0061
ibm05	29,347	98,793	1.0024	1.0024	1.0161	1.0024	1.0024	1.0023	1.0024	1.0022
ibm06	32,498	93,493	1.0143	1.0144	1.0947	1.0143	1.0143	1.0141	1.0143	1.0140
ibm07	45,926	127,774	1.0286	1.0293	1.1832	1.0287	1.0287	1.0288	1.0287	1.0288
ibm08	51,309	154,644	1.0035	1.0035	1.0242	1.0035	1.0035	1.0034	1.0035	1.0035
ibm09	53,395	161,430	1.0265	1.0270	1.1596	1.0264	1.0264	1.0265	1.0264	1.0261
ibm10	69,429	223,090	1.0148	1.0150	1.0988	1.0148	1.0148	1.0150	1.0148	1.0150
ibm11	70,558	199,694	1.0112	1.0114	1.0757	1.0112	1.0112	1.0112	1.0112	1.0114
ibm12	71,076	241,135	1.0060	1.0061	1.0418	1.0060	1.0060	1.0062	1.0060	1.0061
ibm13	84,199	257,788	1.0064	1.0064	1.0431	1.0064	1.0064	1.0064	1.0064	1.0065
ibm14	147,605	394,497	1.0047	1.0047	1.0320	1.0047	1.0047	1.0046	1.0047	1.0046
ibm15	161,570	529,562	1.0238	1.0243	1.1493	1.0238	1.0238	1.0238	1.0238	1.0242
ibm16	183,484	589,253	1.0119	1.0120	1.0801	1.0119	1.0119	1.0118	1.0119	1.0118
ibm17	185,495	671,174	1.0011	1.0011	1.0079	1.0011	1.0011	1.0011	1.0011	1.0012
ibm18	210,613	618,020	1.0034	1.0034	1.0240	1.0034	1.0034	1.0034	1.0034	1.0035

corresponds to the maximum feasible reduction in clock cycle time for the corresponding circuit (see Held et al. [2003] for details). The IBM family, provided by Ali Dasdan, models a similar problem, but contains graphs that are somewhat smaller and sparser Dasdan [2004]. The IBM instances are actually for the related minimum cycle ratio, which assigns a delay to each arc (besides the length), and whose goal is to find a cycle minimizing the ratio between its total length and its total delay. We transformed them into MCM instances by setting all delays to one, as suggested by Dasdan. As noted in [Dasdan 2004], this has almost no effect in the performance of various minimum cycle ratio algorithms on these instances.

We randomly permuted vertex identifiers and adjacency lists when reading the graphs (as usual), but we did not apply potential perturbations, which serve no function in a real-world application. To avoid trivial cases, we discarded the self-loops present in the BONN family.

Table VI shows the average number of scans per vertex performed during a feasibility computation on graphs with no negative cycles ($\lambda = \lambda^+$). Note that, even though we picked the value of λ for which the absence of negative cycles was least obvious, the resulting instances were still rather easy. All algorithms needed little more than a single scan per vertex to find a feasible set of potentials, with very similar performance.

Table VII shows the corresponding results when negative cycles are present ($\lambda = \lambda^-$). Not surprisingly, these are even easier instances, since the algorithms

Table VII. Feasibility: Circuits Containing Negative Cycles

GRAPH	n	m	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
bonn01	4,609	2,916,202	0.9154	0.0922	1.0323	0.4260	0.4262	0.8256	0.3352	0.3295
bonn02	5,361	4,166,868	0.7783	0.8593	1.3475	1.0409	1.0211	0.8193	1.0060	1.0075
bonn03	11,867	8,914,917	0.8447	0.6053	1.2054	0.9465	0.9459	0.8135	0.9175	0.9035
bonn04	20,072	1,321,426	0.9548	0.2617	1.0779	0.5087	0.5074	0.9989	0.6332	0.4584
bonn05	60,309	3,969,429	0.8997	0.5576	1.0377	0.8157	0.8156	0.7822	0.8240	0.4838
bonn06	70,346	892,588	1.0048	0.0697	1.0189	0.5271	0.5275	0.9477	0.5357	0.9089
bonn07	82,038	2,603,211	0.9154	0.0995	1.1013	0.6438	0.6468	0.9675	0.6431	0.3622
bonn08	95,936	1,324,128	0.8682	0.5391	1.2720	0.6948	0.6851	0.9265	0.5023	0.6110
bonn09	346,814	9,222,546	0.8752	0.3375	1.0137	0.9476	0.9478	0.8537	0.9468	0.6154
ibm01	12,752	36,681	0.8122	0.5906	1.2627	0.5921	0.5919	0.7413	0.5958	0.6433
ibm02	19,601	61,829	0.7630	0.4557	1.0720	0.6035	0.6034	0.8066	0.6019	0.7507
ibm03	23,136	66,429	0.8410	0.5336	1.0793	0.8150	0.8148	0.8936	0.8102	0.5834
ibm04	27,507	74,138	0.7677	0.4909	1.0397	0.6446	0.6446	0.8308	0.6436	0.7554
ibm05	29,347	98,793	0.9295	0.6283	1.0207	0.8924	0.8924	0.9526	0.8925	0.5656
ibm06	32,498	93,493	0.8668	0.4086	1.0980	0.5639	0.5638	0.8065	0.5636	0.8633
ibm07	45,926	127,774	0.8918	0.3090	1.1824	0.5770	0.5814	0.8281	0.5815	0.4921
ibm08	51,309	154,644	0.7472	0.1786	0.2653	0.1784	0.1784	0.7282	0.1784	0.2618
ibm09	53,395	161,430	0.8918	0.5938	1.1586	0.8231	0.8224	0.8956	0.8161	0.6170
ibm10	69,429	223,090	0.9820	0.4530	1.0986	0.8030	0.8028	0.7929	0.7991	0.7522
ibm11	70,558	199,694	0.8613	0.4453	1.0757	0.6680	0.6679	0.7404	0.6678	0.7287
ibm12	71,076	241,135	0.8065	0.4798	1.0418	0.6614	0.6613	0.8556	0.6621	0.7864
ibm13	84,199	257,788	0.7755	0.4199	1.0457	0.5943	0.5943	0.7133	0.5940	0.6776
ibm14	147,605	394,497	0.8125	0.4683	1.0320	0.6585	0.6585	0.7247	0.6587	0.6441
ibm15	161,570	529,562	0.6894	0.3877	1.1509	0.4333	0.4333	0.8777	0.4333	0.3823
ibm16	183,484	589,253	0.9539	0.5730	1.0834	0.8632	0.8631	0.7255	0.8605	0.5881
ibm17	185,495	671,174	0.8113	0.4334	1.0122	0.5191	0.5191	0.7895	0.5189	0.8862
ibm18	210,613	618,020	0.7118	0.3676	0.2923	0.3669	0.3669	0.6518	0.3669	0.2882

can stop as soon as a negative cycle is found. There is a clearer separation among the algorithms, however. Those concentrating on a smaller portion of the graph (such as DFS, HYB, RDB, RDH, and especially MBFCT) are often significantly faster than other approaches.

7.2.6 *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. The average number of scans was never much higher than 50, even for graphs with more than 2 million vertices. For RDH, the number is below 10. All algorithms have essentially linear behavior. For a more complete analysis, we consider the BAD family, with the worst-case instances described in Section 6. In addition to the algorithms tested in the previous section, we also included LS, LSG, and LR in these experiments.

Recall that the number of vertices in each worst-case family is a linear function of a parameter k. The number of arcs is quadratic in k for BAD-AF and COMP-DAG and linear for the remaining families. To observe the asymptotic behavior of the algorithms, we ran them on instances with $k = 1{,}000$ and $k = 4{,}000$. Table VIII reports the average number of scans for $k = 4{,}000$ relative to $k = 1{,}000$ (i.e., the average number of scans per vertex with $k = 4{,}000$ divided by the average number of scans per vertex with $k = 1{,}000$). This value should be close to one for algorithms linear in k (and n) and to four for quadratic algorithms.

Table VIII. Feasibility on hard instances. Each entry reports the average number of scans per vertex on instances with $k = 4{,}000$, as a multiple of the average number of scans per vertex on instances with $k = 1{,}000$. Values close to one suggest linear behavior, whereas values close to four indicate quadratic behavior. For each family, we considered variants obtained by applying (•) or not (○), a random permutation of vertices and arcs (PERM), or a large potential perturbation (POT)

FAMILY	PERM	POT	BFCT	MBFCT	GOR	LS	LSG	DFS	HYB	LR	LRG	RDB	RDH
BAD-BFCT	○	○	4.0	4.0	1.0	1.0	1.0	1.0	1.0	3.8	1.2	1.0	1.0
	○	•	5.1	1.1	1.2	5.1	1.0	1.0	5.1	5.1	1.2	1.2	1.1
	•	○	3.3	4.1	1.0	4.1	1.1	1.0	3.8	4.1	1.2	1.0	1.0
	•	•	5.1	3.8	1.2	5.1	1.2	1.1	5.1	5.1	1.2	1.2	1.2
BAD-MBFCT	○	○	1.0	15.9	1.0	1.0	1.0	1.0	1.0	1.1	1.1	1.0	1.0
	○	•	3.7	31.2	1.1	3.7	1.0	1.0	3.7	3.6	1.1	1.0	1.0
	•	○	1.1	4.6	1.0	1.1	1.1	1.0	1.0	1.1	1.1	1.0	1.0
	•	•	3.6	4.6	1.1	3.6	1.2	1.0	3.7	3.6	1.1	1.0	1.0
BAD-GOR	○	○	1.0	1.0	4.0	1.0	1.0	1.0	1.0	3.9	1.0	1.0	1.0
	○	•	1.0	1.5	1.2	1.0	1.0	1.0	1.0	1.9	1.1	1.0	1.1
	•	○	1.0	1.0	4.0	4.0	1.0	1.0	1.0	4.0	1.0	1.0	1.0
	•	•	1.1	1.1	1.2	1.9	1.1	1.1	1.0	1.9	1.1	1.1	1.1
BAD-RD	○	○	1.0	1.0	1.0	1.0	1.0	1.0	1.0	3.8	3.8	4.0	4.0
	○	•	1.0	0.9	1.3	1.0	1.0	1.0	1.0	5.1	5.2	5.3	1.4
	•	○	3.7	3.4	1.0	3.8	1.1	4.0	4.2	3.5	3.8	3.9	4.2
	•	•	5.0	2.8	1.3	4.8	1.3	5.9	5.1	4.9	4.7	4.8	1.3
COMP-DAG	○	○	1.0	4.0	1.0	1.0	1.0	1.0	1.0	1.2	1.2	4.0	1.9
	○	•	1.0	4.4	1.0	1.0	1.0	1.1	4.9	1.3	1.3	5.3	2.2
	•	○	1.2	1.5	1.0	1.2	4.0	3.9	1.2	1.2	1.2	1.4	2.0
	•	•	1.3	1.3	1.1	1.3	5.1	4.9	1.3	1.3	1.3	1.6	2.0
BAD-DFS	○	○	1.0	1.0	1.0	1.0	4.0	4.0	1.0	1.2	1.2	1.0	1.0
	○	•	1.0	1.0	1.3	1.0	5.8	5.9	1.1	1.3	1.3	1.3	1.3
	•	○	1.2	1.2	1.0	1.2	4.0	4.0	1.0	1.2	1.2	1.2	1.0
	•	•	1.3	1.1	1.3	1.3	6.0	5.8	1.3	1.3	1.3	1.3	1.3
BAD-AF	○	○	4.0	4.0	1.0	4.0	1.0	4.0	4.0	3.9	2.0	1.3	2.1
	○	•	4.1	3.5	1.2	4.1	1.2	4.1	4.1	4.0	2.0	1.4	1.9
	•	○	3.3	3.2	1.2	3.3	3.9	3.9	3.3	3.3	2.0	1.5	2.2
	•	•	3.4	2.8	1.2	3.4	4.0	4.0	3.4	3.5	2.0	1.5	2.1

Recall (from Section 6) that vertex identifiers and arc lengths in the BAD family were very carefully chosen to elicit the worst-case behavior of various algorithms. To determine how sensitive the algorithms are to changes in these instances, the table reports results on four variants of each subfamily. The first variant corresponds to the original graphs, exactly as described in Section 6. The second applies a potential transformation to the input, assigning a random potential between 0 and 10^6 to each vertex. The third variant applies no potential perturbation, but randomly permutes all vertex identifiers and adjacency lists. Finally, the fourth variant applies both the perturbation and the permutations. Each entry in the table with a randomized component is the average of 50 runs, with different seeds. Note that a problem designed to be hard for one algorithm may be easy for others, but applying one or both of the previously mentioned transformations can make the problem harder.

We first consider the original graphs, with no perturbations or permutations. The table clearly shows the quadratic behavior of BFCT, DFS, GOR, RDB, and RDH in their respective worst-case families, as well as the cubic behavior of MBFCT on BAD-MBFCT. Every algorithm exhibits quadratic behavior in

at least one case, including methods for which there is no specific worst-case family, such as the randomized ones. GOR, LS, LSG, and HYB are superlinear in a single case. In contrast, LR is clearly superlinear in four cases, as is MBFCT. We also note that HYB, which is a combination of DFS and BFCT, performs a constant number of scans per vertex on both BAD-DFS and BAD-BFCT. It is, however, quadratic on BAD-AF.

Next, consider the second variant in each family, which adds potential perturbations to the original graph. Although most algorithms remain quadratic in at least one case, RDH and GOR are consistently faster (subquadratic) after the perturbation. Conversely, perturbations can make instances harder for some algorithms. For example, HYB scales linearly on the original BAD-BFCT family, but quadratically on the transformed problems. We observed that BAD-MBFCT is particularly sensitive to small perturbations of a few arcs. The high variance and the relatively small size of these instances help to explain why the ratios are occasionally higher than the worst-case analysis would predict: ratios are higher than 4 for some quadratic algorithms and higher than 16 for MBFCT.

We now turn to the third variant, which randomly permutes original vertex identifiers and adjacency lists. Most graphs remain adversarial for their respective algorithms, despite the fact that the analyses in Section 6 often rely on vertices being processed in a certain order during the first pass. Some algorithms do occasionally become faster; MBFCT, for instance, no longer exhibits cubic behavior on BAD-MBFCT, but it is still at least quadratic. The converse is also true: Some methods have significantly worse performance on the permuted graphs. In particular, LS becomes superlinear on BAD-BFCT, BAD-GOR, and BAD-RD. Not coincidentally, on these families LR is also quadratic, even without the permutation. On BAD-RD, for example, the expected number of passes needed by LR to process the main structure (shown in Figure 5) is linear, and in many of these passes the high-degree vertex and its outgoing neighbors are scanned.

Finally, the fourth variant of each subfamily combines both random permutations and perturbations. Note that the algorithms are almost always superlinear, albeit only slightly in some cases. Once again, RDH and especially GOR seem to be more robust than other algorithms.

7.2.7 *Variance.* We briefly mentioned that the variance was very high for some of the worst-case families. Although we did not observe such extreme behavior in our main experiments (with nonadversarial families), some algorithms were more stable than others. Table IX reports percent standard deviations in the number of scans per vertex (over 10 runs) on the four main synthetic families we studied, with $n \approx 2^{20}$ (values are similar for other instance sizes). Each entry is the ratio between the standard deviation and the average, multiplied by 100.

Note that, in general, deviations are lower for subfamilies with longer cycles (01, 04, and 05). On subfamilies 02 and 03, most algorithms find a negative cycle soon after any of the vertices in the cycle is scanned, but the time to reach such a vertex can vary significantly. In particular, the fastest algorithms on

Table IX. Feasibility: Percent Standard Deviation in the Number of Scans Per Vertex (over 10 runs) for $n \approx 2^{20}$

FAM	SUB	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
RAND5	01	0.09	7.93	0.11	0.09	0.06	0.09	0.24	0.50
	02	26.07	95.72	18.09	39.69	40.89	28.83	44.02	19.95
	03	28.65	99.95	0.05	61.05	61.05	36.40	65.17	75.49
	04	0.42	15.16	2.54	8.53	0.39	2.59	2.44	0.77
	05	0.79	19.58	2.36	0.60	0.76	0.71	0.64	0.73
SQNC	01	0.09	0.17	0.07	0.09	0.10	0.08	0.09	0.07
	02	25.38	84.97	20.98	81.64	79.27	23.88	92.29	77.52
	03	27.74	78.60	0.05	94.69	94.69	37.33	92.80	52.37
	04	0.63	25.18	0.64	7.73	0.62	0.60	3.02	2.42
	05	0.84	17.74	2.84	0.82	0.81	0.76	0.97	1.00
LNC	01	0.03	0.11	0.04	0.07	0.05	0.05	0.04	0.03
	02	12.85	90.12	18.23	66.30	59.00	22.61	82.51	81.35
	03	27.62	92.16	0.03	93.42	93.42	30.61	94.15	74.41
	04	0.54	22.10	2.67	6.17	0.55	1.66	2.98	2.68
	05	0.79	17.83	1.25	0.84	0.77	0.70	1.11	1.18
PNC	01	0.09	0.11	0.10	0.07	0.07	0.08	0.07	0.08
	02	13.99	128.16	21.86	92.33	90.79	26.51	60.54	76.35
	03	37.57	139.21	0.09	99.90	99.90	44.38	131.62	135.72
	04	1.16	33.31	6.99	7.29	1.17	3.12	4.60	5.02
	05	1.08	19.92	4.38	0.85	0.98	0.94	1.31	1.05

subfamily 03 tend to have higher variance. For similar reasons, MBFCT often has much higher variance than algorithms with comparable (or even better) average performance.

7.2.8 *Running Times.* So far, we have compared the algorithms only in terms of operation counts, which are machine-independent. We now discuss running times. For reference, the total running time for RAND5 is shown in Table X. For this family, the algorithms are usually within a factor of two or three from each other. The main exception is GOR, which is substantially slower on subfamily 03. MBFCT and RDH are other occasional outliers.

For a more detailed analysis, Table XI shows the average time per scan on the same family. Note that this table only takes into account 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 X 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 this reason, the results for this subfamily are omitted from Table XI.

In theory, for all algorithms the time per scan is proportional to vertex degree, which is constant for this family. The slight increase in scan times with n is usually due to the reduced number of cache hits on larger graphs. For RDH, the time per scan also depends on the heap size, which may grow with n. In practice, this data structure overhead makes scans more expensive for RDH than for the other algorithms.

Constant factors are also important. GOR, which executes cheaper scans during its depth-first search phase, has the lowest time per scan on average.

Table X. Feasibility: Average Total Time in Seconds for RAND5

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,144	0.402	2.381	0.588	0.702	0.667	0.620	0.709	1.228
	524,288	0.851	5.188	1.161	1.614	1.414	1.428	1.416	2.692
	1,048,576	1.880	10.639	2.483	3.256	2.892	2.885	3.159	5.844
	2,097,152	3.948	26.843	5.537	7.806	6.565	6.328	7.439	13.886
02	262,144	0.320	0.319	0.484	0.531	0.523	0.373	0.625	0.929
	524,288	0.673	2.390	0.937	1.086	0.986	1.011	1.156	2.258
	1,048,576	1.400	4.086	1.931	2.303	2.075	2.119	2.142	4.602
	2,097,152	3.414	11.810	4.888	4.767	5.261	4.462	4.919	12.425
03	262,144	0.044	0.006	0.331	0.008	0.015	0.086	0.016	0.025
	524,288	0.091	0.016	0.708	0.024	0.020	0.160	0.030	0.045
	1,048,576	0.163	0.025	1.574	0.036	0.044	0.322	0.055	0.097
	2,097,152	0.286	0.049	3.272	0.081	0.083	0.619	0.117	0.186
04	262,144	1.197	4.095	1.516	1.228	1.580	1.525	1.289	1.866
	524,288	2.707	8.944	3.273	2.675	3.205	3.155	2.758	4.136
	1,048,576	5.800	19.809	7.145	5.908	7.667	7.137	6.003	9.076
	2,097,152	12.856	46.471	16.109	12.762	16.125	15.302	14.375	21.122
05	262,144	2.638	11.373	4.314	3.480	3.099	3.442	2.748	3.623
	524,288	5.569	22.009	9.399	6.889	7.063	7.063	6.069	7.578
	1,048,576	12.267	47.840	18.742	14.914	14.039	15.521	13.162	16.361
	2,097,152	27.709	118.666	43.056	34.922	32.805	34.547	32.188	38.678

Table XI. Feasibility: Average Time Per Scan in Microseconds for RAND5

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,144	0.71	0.76	0.56	0.91	1.02	1.08	1.10	1.92
	524,288	0.77	0.81	0.55	1.06	1.08	1.25	1.10	2.12
	1,048,576	0.85	0.81	0.59	1.07	1.11	1.26	1.22	2.30
	2,097,152	0.89	0.96	0.66	1.29	1.26	1.39	1.45	2.74
02	262,144	0.70	0.79	0.52	0.98	1.05	1.19	1.15	1.95
	524,288	0.80	0.82	0.52	1.18	1.13	1.28	1.22	2.24
	1,048,576	0.81	0.85	0.57	1.23	1.17	1.37	1.24	2.33
	2,097,152	0.90	1.05	0.64	1.35	1.31	1.56	1.53	2.94
04	262,144	0.80	0.78	0.58	0.96	0.99	1.10	1.08	1.65
	524,288	0.88	0.86	0.61	1.03	0.98	1.12	1.12	1.75
	1,048,576	0.92	0.89	0.65	1.12	1.15	1.24	1.19	1.91
	2,097,152	1.00	0.96	0.72	1.21	1.18	1.30	1.39	2.18
05	262,144	0.90	0.86	0.71	0.98	1.02	1.12	1.06	1.46
	524,288	0.94	0.91	0.77	0.96	1.15	1.15	1.17	1.51
	1,048,576	1.04	0.96	0.77	1.04	1.15	1.26	1.26	1.63
	2,097,152	1.17	1.05	0.88	1.21	1.33	1.39	1.53	1.92

It is closely followed by BFCT and MBFCT, which obviously have similar performance per scan (LS and LSG, not shown in the table, are also comparable). The times for DFS and HYB are slightly higher, since these algorithms need more complicated data structures, allowing arbitrary deletion of labeled vertices. Data structure overhead also explains why RDB and especially RDH are even slower. In this family, however, RDH is no more than three times slower per scan than BFCT, regardless of graph size, which indicates that the heap is often very small. Finally, even though they have very simple data structures, LRG and LR (not shown) can be substantially slower (per scan) than the simplest methods, due to their calls to the pseudorandom number generator

Table XII. Feasibility: Average Time Per Scan in Microseconds for SQNC

FAM	n	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
01	262,145	0.46	0.38	0.25	0.48	0.57	0.70	0.59	0.91
	524,177	0.51	0.44	0.27	0.52	0.58	0.78	0.62	1.06
	1,048,577	0.54	0.48	0.29	0.63	0.67	0.87	0.69	1.19
	2,096,705	0.62	0.56	0.35	0.66	0.73	0.98	0.82	1.40
02	262,145	0.36	0.17	0.26	0.48	0.43	0.75	0.63	1.26
	524,177	0.38	0.41	0.25	0.52	0.56	0.84	0.59	1.06
	1,048,577	0.45	0.44	0.29	0.68	0.52	0.89	0.73	1.32
	2,096,705	0.55	0.51	0.35	0.69	0.67	1.04	0.73	1.20
04	262,145	0.58	0.52	0.41	0.63	0.74	0.83	0.68	0.96
	524,177	0.64	0.56	0.34	0.61	0.70	0.82	0.68	1.00
	1,048,577	0.70	0.67	0.39	0.71	0.78	0.95	0.79	1.11
	2,096,705	0.82	0.69	0.43	0.80	0.89	1.06	0.92	1.25
05	262,145	0.72	0.66	0.52	0.72	0.95	0.83	0.81	0.98
	524,177	0.78	0.66	0.45	0.75	0.82	0.93	0.85	1.12
	1,048,577	0.87	0.76	0.51	0.88	0.95	0.98	0.89	1.21
	2,096,705	0.95	0.82	0.58	0.97	1.07	1.13	1.14	1.42

and to the potentially worse locality associated with processing vertices in an unstructured way.

For comparison, Table XII shows times per scan for SQNC (once again, we omit subfamily 03). The results are not substantially different from those in Table XI. Most algorithms are slightly faster, since these graphs are more structured and have higher locality, but the relative performances are similar.

7.3 Shortest Paths

As already mentioned, the feasibility and shortest-path problems are closely related. In fact, most of the algorithms 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 XIII shows the average number of scans per vertex required to solve the shortest-path problem (as opposed to feasibility) on the main families tested. We used $n = 2^{20}$ for RAND5 and $n = 2^{20} + 1$ for SQNC, LNC, and PNC. For RAND5 and CAL, vertex 1 was used as the source. For the TOR-generated families, we used the special source added by TOR itself.

Note that these results are often significantly different from those obtained for the feasibility problem. 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 DFS algorithm is clearly ill-suited for the shortest-path problem, but other algorithms have poor performance as well. One example is RDB on 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.) Results for LSG (not shown) are almost as bad as those for DFS. In contrast, LS is much better, with similar performance to BFCT. Note that in this case HYB does what it is supposed to: By switching from DFS to BFCT after the first pass, its performance is much better than that of DFS.

Table XIII. Shortest Paths: Scans Per Vertex

FAM	SUB	BFCT	GOR	DFS	HYB	LRG	RDB	RDH
RAND5	01	2.1091	5.3997	8.4633	3.5625	3.4482	2.2378	2.5832
	02	1.2353	3.8956	5.0639	2.3343	2.0279	1.6061	1.7891
	03	0.0068	0.2338	0.0089	0.0089	0.0065	0.0053	0.0061
	04	11.8576	22.7002	16.0306	13.3141	12.7421	6.5809	6.2469
	05	19.8388	41.1453	31.8801	21.3157	22.2196	13.8144	13.1582
SQNC	01	2.8731	8.6576	157.3064	3.8733	66.8467	19.1232	15.0928
	02	0.6010	2.1986	0.6435	0.5689	0.6342	0.5327	0.3301
	03	0.0021	0.0064	0.0023	0.0023	0.0016	0.0014	0.0008
	04	8.0837	20.0803	17.7657	9.1128	9.6495	6.6632	5.8751
	05	17.3253	37.4779	30.8913	18.4549	19.5139	12.4973	11.8517
LNC	01	3.3136	7.1818	10,501.6749	4.3136	13.5353	156.8838	10.0531
	02	0.8770	1.8999	1.0290	0.7673	3.5139	0.2622	2.4736
	03	0.0010	0.0022	0.0033	0.0033	0.0005	0.0003	0.0003
	04	8.0192	20.1366	22.2135	9.0294	10.1721	6.5573	6.2214
	05	16.8893	36.6936	31.8305	17.9226	18.9917	12.3492	11.8120
PNC	01	10.8573	17.2985	9,850.2069	11.8401	13.1151	22.1327	28.8282
	02	2.8593	4.5822	3.5268	3.6851	3.4388	4.9745	3.4685
	03	0.0018	0.0087	0.0455	0.0455	0.0026	0.0012	0.0005
	04	8.2006	18.3189	21.0665	9.2228	9.7809	7.7816	7.4620
	05	7.2747	19.4978	17.8857	8.4513	8.6805	7.2640	7.8931
CAL	01	2.9122	6.1905	94.2071	3.9119	17.9860	12.0802	6.9223
	02	0.7999	2.6174	0.7244	0.6720	0.5683	0.3910	0.6405
	03	0.0005	0.0011	0.0009	0.0009	0.0004	0.0003	0.0003
	04	9.4416	18.5033	21.4810	10.4795	10.8013	6.0124	4.3224
	05	23.9267	52.2533	38.7570	25.0923	26.6874	15.5642	14.1749

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 some of the differences between our results and those reported by Wong and Tam [2005]. They compare their feasibility algorithm, MBFCT, with the shortest-path algorithms of Goldberg and Cherkassky [1999].

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

Table XIV. Incremental Feasibility: Scans Per Vertex on RAND5 ($n = 65{,}536$)

CASE	ARCS	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
−2	1	0.178	0.178	0.861	0.570	0.495	0.168	0.142	0.112
	2	0.157	0.127	1.046	0.388	0.459	0.123	0.172	0.111
	64	0.238	0.200	0.440	0.119	0.119	0.231	0.046	0.028
	1,024	0.247	0.149	0.661	0.018	0.018	0.240	0.024	0.011
−1	1	0.221	0.221	0.828	0.290	0.300	0.232	0.157	0.176
	2	0.374	0.285	1.377	0.592	0.429	0.482	0.180	0.133
	64	0.375	0.364	0.631	0.105	0.105	0.467	0.060	0.030
	1,024	0.287	0.328	0.724	0.028	0.028	0.320	0.026	0.011
0	1	0.516	0.516	1.086	0.598	0.573	0.553	0.431	0.423
	2	1.226	1.262	2.558	1.476	1.334	1.308	1.033	1.016
	64	3.542	5.984	9.524	4.966	3.827	3.729	2.850	2.830
	1,024	4.184	8.394	12.497	6.710	4.480	4.426	3.375	2.979
1	1	0.004	0.004	0.008	0.004	0.004	0.004	0.004	0.004
	2	0.012	0.012	0.024	0.012	0.012	0.012	0.012	0.012
	64	1.034	1.129	1.938	1.155	1.095	1.105	0.941	0.887
	1,024	5.132	8.341	9.202	6.021	5.336	5.410	3.680	3.394

problems, each a perturbation of the previous one. This occurs in applications such as the minimum cycle mean problem [Dasdan 2004; Chandrachoodan et al. 2001].

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 preinitialized 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 $-\lfloor D/2 \rfloor$ 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 XIV shows the average number of scans per vertex required to find feasible potentials starting from an SPRAND-generated graph with $n = 65{,}536$ and $m = 5n$. For each case, we vary the number of arcs made negative: 1, 2, 64, or 1,024. Tables XV, XVI and XVII present the average number of scans per vertex for SQNC, LNC, and PNC instances, all with 65,537 vertices.

Here, RDH is the most robust algorithm. It is almost never much worse than any other method, and often much better. Although RDB is about as fast as

Table XV. Incremental Feasibility: Scans Per Vertex on SQNC ($n = 65{,}537$)

CASE	ARCS	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
−2	1	1.434	1.434	29.339	16.537	1.919	2.410	0.693	0.701
	2	1.358	1.354	13.338	1.036	1.199	2.005	0.751	0.796
	64	1.962	0.470	8.936	0.043	0.043	1.717	0.505	0.416
	1,024	2.638	0.529	5.543	0.014	0.014	2.175	0.583	0.224
−1	1	1.597	1.597	23.420	15.467	1.915	9.669	3.961	0.703
	2	1.289	1.716	16.547	12.059	1.303	2.660	1.477	0.797
	64	1.812	0.380	8.928	0.037	0.037	1.560	0.501	0.378
	1,024	2.721	0.514	5.007	0.028	0.028	2.040	0.547	0.228
0	1	1.613	1.613	22.706	15.485	1.939	9.522	4.013	0.704
	2	2.072	2.070	26.374	19.950	2.476	12.334	5.046	1.062
	64	2.329	2.473	16.113	21.697	2.762	13.023	5.317	2.825
	1,024	2.515	3.997	11.069	20.885	2.751	12.668	5.593	3.412
1	1	0.612	0.612	5.871	4.645	0.846	3.304	1.011	0.349
	2	1.093	1.145	9.524	9.195	1.383	5.900	1.877	0.711
	64	2.732	2.944	16.551	22.381	3.149	12.720	5.653	3.940
	1,024	3.217	5.404	15.942	20.279	3.399	12.584	6.838	4.739

Table XVI. Incremental Feasibility: Scans Per Vertex on LNC ($n = 65{,}537$)

CASE	ARCS	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
−2	1	0.662	0.662	2.272	67.010	0.793	2.521	1.067	0.200
	2	0.380	1.118	1.458	0.614	0.670	1.548	2.537	0.361
	64	0.057	0.040	0.167	0.030	0.030	0.032	0.198	0.012
	1,024	0.519	0.003	0.685	0.050	0.050	0.220	0.393	0.002
−1	1	0.662	0.662	2.379	53.402	0.727	2.686	5.119	0.200
	2	0.380	1.119	1.529	33.129	0.501	1.548	3.344	0.382
	64	0.056	0.040	0.164	0.011	0.011	0.027	0.197	0.012
	1,024	0.517	0.003	0.687	0.050	0.050	0.242	0.306	0.002
0	1	0.662	0.662	2.105	53.402	0.727	2.686	5.041	0.200
	2	2.963	2.963	8.266	240.293	3.256	11.684	22.246	0.922
	64	3.295	3.293	6.510	267.368	3.480	13.153	24.083	1.655
	1,024	3.271	3.343	6.155	264.570	3.302	12.652	24.584	1.257
1	1	0.330	0.330	1.046	13.489	0.363	1.245	1.775	0.100
	2	1.602	1.561	4.886	71.385	1.756	6.185	8.830	0.512
	64	4.421	4.138	9.939	249.200	4.583	15.982	21.842	2.794
	1,024	4.515	4.632	9.346	263.014	4.636	16.212	22.182	4.169

RDH for random graphs, it is noticeably worse for TOR-generated instances. Unsurprisingly, BFCT and MBFCT have the exact same 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. When the number of negative arcs is small, the worst algorithm (by far) is DFS. With one negative arc, the algorithm ends up performing a single-source shortest-path computation, and Section 7.3 has already shown that DFS can have terrible performance in such cases. The results for LSG (not shown) are very similar, while the performance of LS is closer to that of BFCT. The hybrid algorithm is significantly more robust: its performance is generally comparable to that of BFCT, but it mimics DFS when the latter is really fast (cases −1 and −2 with many negative arcs). On SQNC with 1,024 negative arcs, HYB and DFS are the fastest methods by far, even compared to RDH.

Table XVII. Incremental Feasibility: Scans Per Vertex on PNC ($n = 65{,}537$)

CASE	ARCS	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
−2	1	5.365	5.365	7.923	144.172	5.330	5.807	7.016	0.493
	2	3.706	5.773	5.486	29.389	3.811	3.857	4.457	0.529
	64	0.002	0.305	0.017	0.093	0.093	0.022	0.073	0.025
	1,024	0.014	0.022	0.065	0.018	0.018	0.006	0.049	0.001
−1	1	5.009	5.009	7.816	118.639	4.978	5.450	6.258	0.497
	2	3.825	5.549	5.694	88.107	3.806	3.955	4.634	0.529
	64	0.004	0.308	0.025	0.146	0.146	0.047	0.086	0.025
	1,024	0.015	0.028	0.067	0.026	0.026	0.008	0.038	0.002
0	1	5.009	5.009	7.152	118.639	4.976	5.453	6.272	0.497
	2	7.980	7.980	11.357	191.479	7.961	8.651	10.009	0.795
	64	9.476	9.429	14.221	213.725	9.420	10.441	11.902	3.959
	1,024	6.887	10.605	13.753	63.318	6.910	8.505	9.463	2.692
1	1	2.307	2.307	3.319	31.576	2.297	2.489	2.785	0.249
	2	4.624	4.545	6.580	70.600	4.647	5.032	5.676	0.498
	64	11.621	12.416	18.026	242.371	11.619	12.804	14.125	3.133
	1,024	9.626	13.653	17.607	85.351	9.657	11.374	12.426	4.716

Table XVIII. Incremental Feasibility: Total Number of Scans on CAL

CASE	ARCS	BFCT	MBFCT	GOR	DFS	HYB	LRG	RDB	RDH
−2	1	2.0	2.0	13.5	2.0	2.0	2.0	2.0	2.0
	2	3.0	2.0	19.4	2.0	2.0	2.4	2.0	2.0
	64	65.0	2.0	329.1	2.0	2.0	10.3	2.0	2.0
	1,024	954.1	2.0	5124.1	2.0	2.0	30.4	2.0	2.0
−1	1	2.0	2.0	11.7	2.0	2.0	2.0	2.0	2.0
	2	3.0	2.0	16.7	2.0	2.0	2.4	2.0	2.0
	64	65.0	2.0	327.2	2.0	2.0	10.3	2.0	2.0
	1,024	954.1	2.0	5122.2	2.0	2.0	30.4	2.0	2.0
0	1	4.3	4.3	9.1	4.3	4.3	4.3	4.3	4.3
	2	7.3	7.3	16.0	7.5	7.5	7.3	7.3	7.3
	64	262.9	262.9	575.5	272.0	269.9	266.4	261.9	261.8
	1,024	4145.3	4145.5	9082.2	4219.7	4207.4	4170.3	4140.2	4136.3
1	1	2.7	2.7	6.7	2.7	2.7	2.7	2.7	2.7
	2	5.0	5.0	11.6	5.0	5.0	5.0	5.0	5.0
	64	172.0	172.0	391.3	174.1	173.9	172.8	171.8	171.8
	1,024	2727.9	2728.2	6254.8	2734.8	2734.3	2729.4	2727.8	2728.2

Table XVIII 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 this experiment, it is clear that MBFCT, DFS, HYB, RDB, and RDH are remarkably good when there are negative cycles (and so is LSG, which is not shown). It is not hard to see why. On 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 most algorithms; the main exceptions are BFCT, GOR, and, to a lesser degree, LRG.

8. CONCLUDING REMARKS

We developed an experimental framework for the feasibility problem that is more extensive than the previous one [Cherkassky and Goldberg 1999], which

actually compared shortest-path algorithms and concluded that GOR and BFCT had the best performance. For the feasibility problem, we found that it is often enough to scan a small subset of the vertices, which makes GOR usually much slower than BFCT. It is harder, however, to elicit the worst-case behavior of GOR. Another previously studied feasibility algorithm, MBFCT [Wong and Tam 2005], performs poorly on some problem classes but outperforms BFCT when negative cycles can be found locally.

We introduced an $O(n)$-pass framework for designing shortest-path and feasibility algorithms and proposed several new algorithms that fall into this framework. Two new algorithms, RDH and HYB, join BFCT and MBFCT as the most practical in our study, although none of these is dominant. The most established among these, BFCT, is often the fastest, but it is not competitive for simple problems with many easy-to-find cycles. In such cases, HYB can be substantially faster, even though in general it is slightly slower than BFCT. RDH performs the fewest scans overall, but it often takes more time because of the relatively high cost per operation. MBFCT is the least robust among these codes (with much inferior worst-case performance), but it performs well on easier problems, in which a cycle can be found while looking at a small subgraph.

For an application where many feasibility instances will be solved, we recommend running all four algorithms on a few sample problems and picking the best. If one needs to solve only a small number of instances, the discussion in the previous paragraph can be used to select an algorithm that fits the problem structure. If the structure is completely unknown, BFCT, HYB, and RDH are all reasonable choices.

ACKNOWLEDGMENTS

We thank Ali Dasdan and Stephan Held for providing us with the minimum cycle mean instances and three anonymous referees for numerous helpful suggestions.

REFERENCES

BELLMAN, R. E. 1958. On a routing problem. *Quart. Appl. Math. 16*, 87–90.

CHANDRACHOODAN, N., BATTACHARYYA, S. S., AND LIU, K. J. R. 2001. Adaptive negative-cycle detection in dynamic graphs. In *Proceedings of International Symposium on Circuits and Systems (ISCAS)*. IEEE, Los Alamitos, CA, 163–166.

CHERKASSKY, B., GEORGIADIS, L., GOLDBERG, A. V., TARJAN, R. E., AND WERNECK, R. F. 2008. Shortest-path feasibility algorithms: An experimental evaluation. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Philadelphia, 118–132.

CHERKASSKY, B. V. AND GOLDBERG, A. V. 1999. Negative-cycle detection algorithms. *Math. Prog. 85*, 277–311.

CHERKASSKY, B. V., GOLDBERG, A. V., AND RADZIK, T. 1996. Shortest-Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog. 73*, 129–174.

DASDAN, A. 2004. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *ACM Trans. Des. Autom. Electron. Syst. 9,* 4, 385–418.

DECHTER, R., MEIRI, I., AND PEARL, J. 1991. Temporal Constraint Networks. *Artif. Intell. 49*, 61–95.

DEMETRESCU, C., GOLDBERG, A., AND JOHNSON, D. 2007. 9th DIMACS Implementation Challenge: Shortest-Paths. http://www.dis.uniroma1.it/~challenge9/.

DENARDO, E. V. AND FOX, B. L. 1979. Shortest–Route Methods: 1. Reaching, pruning, and buckets. *Oper. Res. 27*, 161–186.

DIAL, R. B., GLOVER, F., KARNEY, D., AND KLINGMAN, D. 1979. A computational analysis of alternative algorithms and labeling techniques for finding shortest-path trees. *Networks 9*, 215–248.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numer. Math. 1*, 269–271.

FORD, JR., L. 1956. Network Flow Theory. Tech. rep. P-932, The Rand Corporation.

FORD, JR., L. AND FULKERSON, D. R. 1962. *Flows in Networks*. Princeton University Press, Princeton, NJ.

FREDMAN, M. L. AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach. 34*, 596–615.

GALLO, G. AND PALLOTTINO, S. 1988. Shortest-paths algorithms. *Annals of Oper. Res. 13*, 3–79.

GEORGIADIS, L., GOLDBERG, A. V., TARJAN, R. E., AND WERNECK, R. F. 2009. An experimental study of minimum mean cycle algorithms. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Philadelphia, 1–13.

GOLDBERG, A. V. 1995. Scaling algorithms for the shortest-paths problem. *SIAM J. Comput. 24*, 494–504.

GOLDBERG, A. V. AND RADZIK, T. 1993. A heuristic improvement of the Bellman-Ford algorithm. *Applied Math. Let. 6*, 3–6.

HELD, S., KORTE, B., MASSBERG, J., RINGE, M., AND VYGEN, J. 2003. Clock scheduling and clocktree construction for high performance ASICs. In *Proceedins IEEE/ACM International Conference on CAD (ICCAD'03)*. 232–239.

KENNINGTON, J. L. AND HELGASON, R. V. 1980. *Algorithms for Network Programming*. John Wiley and Sons, New York.

KOLLIOPOULOS, S. AND STEIN, C. 1996. Finding real-valued single-source shortest-paths in $o(n^3)$ expected time. In *Proceedings of the 5th International Programming and Combinatorial Optimization Conference.*

MOORE, E. F. 1959. The Shortest-Path Through a Maze. In *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press, 285–292.

NONATO, M., PALLOTTINO, S., AND XUEWEN, B. 1999. SPT_L Shortest-Path Algorithms: Reviews, New Proposals, and Some Experimental Results. Tech. rep. TR-99-16, Dipartmento di Informatica, Pisa University, Italy.

PALLOTTINO, S. 1984. Shortest-path methods: Complexity, interrelations and new propositions. *Networks 14*, 257–267.

PAPE, U. 1974. Implementation and efficiency of Moore algorithms for the shortest-root problem. *Math. Prog. 7*, 212–222.

TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 146–160.

TARJAN, R. E. 1981. Shortest-Paths. Tech. rep., *AT&T Bell Laboratories*, Murray Hill, NJ.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia.

TSEITIN, G. 1970. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*. 115–125.

WONG, C.-H. AND TAM, Y.-C. 2005. Negative-cycle detection problem. In *Proceedings of the 13th Annual European Symposium on Algorithms*. Springer, Berlin, 652–663.

Received March 2009; accepted March 2009