An Experimental Study of Minimum Mean Cycle Algorithms

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

Abstract

We study algorithms for the minimum mean cycle problem, a parametric version of shortest path feasibility (SPF). The three basic approaches to the problem are cycle-based, binary search, and tree-based. The first two use an SPF algorithm as a subroutine, while the latter uses a parametric approach. When implementing the SPF-based methods, one has a choice of SPF algorithms and incremental optimization strategies. There are also several ways to handle precision issues. This leads to dozens of variants, which we systematically compare. Our experimental setup is more comprehensive than in previous studies. In our experiments, the tree-based method and two implementations of the cycle-based method outperformed other approaches, including binary search.

1 Introduction

Given a directed graph with arc lengths, the *minimum mean cycle* (MMC) problem is that of finding a cycle with minimum mean length. The mean length of a cycle is the ratio between its total length and its number of arcs. The MMC problem is a parametric version of the shortest path feasibility (SPF) problem. The goal of this problem is to find a cycle of negative total length (*negative cycle*) or to show that one does not exist by finding a potential function (a set of dual variables) that makes all arc lengths non-negative.

The MMC problem, and the closely related *minimum ratio cycle* problem, have applications in areas ranging from discrete event systems and computer-aided design to graph theory. See Dasdan [6] for a detailed discussion and references.

A number of algorithms for the problem have been proposed [11, 12, 13, 14, 17], most of which are extensions of SPF algorithms. In particular, some use an incremental version of an SPF algorithm to find an optimal solution to a subproblem, which is then modified and reoptimized. Two experimental studies [2, 6] show that improved SPF algorithms lead to improved MMC algorithms.

Our recent study of SPF algorithms [3], including incremental ones, puts us in a good position to study the MMC problem. In particular, we wanted to see if improved SPF algorithms lead to improved MMC algorithms. Efficient implementations of MMC algorithms require nontrivial engineering, including data structures, efficient incremental restart, early termination detection, and hybrid algorithms. Our contributions include performance optimizations, new variants, and systematic treatment of precision issues.

While SPF algorithms use only additions and subtractions, MMC algorithms also use division, and therefore exact solutions require significantly higher precision. In particular, double-precision floating point representation (providing 52 bits of precision), which has been used often in MMC implementations, is insufficient to solve to optimality problems with millions of vertices and arc lengths in the millions, problems that modern computers can easily handle. Even 64-bit integers may not be enough, and higher precision arithmetic is not currently supported by mainstream hardware. We discuss precision issues in detail in Section 4.

Previous empirical studies focused on real-world graphs representing circuits and on small random graphs. These problems are easy—good codes solve them in linear time, with very small constants. The studies disagree on which algorithm is the fastest: Chandrachoodan et al. [2] concluded that Lawler's binary search algorithm [14] using an incremental version of Tarjan's SPF subroutine [16] is the best, but Dasdan [6] (see also [7]) found the tree-based algorithm of Young et al. [17] to be the fastest, with an optimized version of Howard's cycle-based algorithm [11] a close second. Neither study reported operation counts, making it hard to compare their results to each other and to newer studies

[1]Hewlett-Packard Laboratories, Palo Alto, CA 94304. Current address: Informatics and Telecommunications Engineering Department, University of Western Macedonia, Kozani, Greece. E-mail: lgeorg@uowm.gr.

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

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

like ours. Since the experiments are limited to a small number of easy problem families, the conclusions have limited applicability.

We used a more extensive data set, including harder synthetic graph families and specially constructed bad-case graphs. Our results give a better picture of relative algorithm performance. We found that, although Howard's algorithm performs well on simple problems, it is generally not robust. Also, the binary search algorithm is not competitive with other alternatives. The best performers in our study are the algorithm of Young et al. and two variants of the cycle-based method.

This paper is organized as follows. Section 2 presents background information on the MMC problem, including definitions and algorithms for the SPF problem. We describe the basic MMC algorithms we study, including new variants, in Section 3. Section 4 deals with precision issues, and Section 5 discusses incremental restarts. Section 6 introduces pathological input families that are particularly challenging for some of the algorithms we study. Section 7 reports experimental results, and final remarks are made in Section 8.

2 Background

2.1 Definitions. Let $G = (V, A)$ be a graph with n vertices, m arcs, and an integral length function $\ell : A \to [-U, U]$. Given a cycle C, we denote by $\ell(C)$ its *length* (the sum of its arc lengths), by $|C|$ its number of arcs, and by $\mu(C)$ its *mean length* (defined as $\ell(C)/|C|$). A zero cycle is a cycle of length zero; a negative cycle is a cycle of negative length. The MMC problem is to find a cycle of minimum mean length; the length of such a cycle, which we denote by λ^*, is the *minimum cycle mean* of the graph.

To simplify the exposition, we work with an *augmented graph*, obtained from G by adding a (root) vertex s and arcs of length zero from s to every vertex in V. Since this adds no cycles, the minimum mean cycle does not change.

Reduced lengths with respect to a *potential function* π are defined as $\ell_\pi(v, w) = \ell(v, w) + \pi(v) - \pi(w)$ for each arc (v, w). This transformation does not affect cycle lengths. The *feasibility problem* is to find a negative cycle in G or a proof that none exists, given by a potential function that makes all reduced lengths non-negative.

For a parameter λ, the parametric length function is $\ell_\lambda = \ell - \lambda$. We say λ is *feasible* if no cycle has negative length with respect to ℓ_λ. If λ is feasible, we say that ℓ_λ is a feasible length function. With respect to ℓ_{λ^*}, there is a zero cycle but no negative cycle.

THEOREM 2.1. *λ is feasible if and only if $\lambda \leq \lambda^*$.*

2.2 SPF Algorithms. Using the previous theorem, one can solve the MMC problem using repeated calls to an SPF algorithm. We now discuss several SPF algorithms based on the *scanning method*. The method maintains for every vertex v a potential $\pi(v)$, a parent $p(v)$, and a *status* $S(v) \in \{\text{unreached, labeled, scanned}\}$. Initially s is labeled and other vertices are unreached. Unless there is a negative cycle, every unreached vertex eventually becomes labeled and then scanned. Given a labeled vertex v, SCAN(v) processes every arc (v, w) as follows: if $\pi(v) + \ell(v, w) < \pi(w)$, then set $\pi(w) \leftarrow \pi(v) + \ell(v, w)$, $p(w) \leftarrow v$, and $S(w) \leftarrow$ labeled. Once all outgoing arcs are processed, set $S(v) \leftarrow$ scanned.

The *parent graph* is the graph induced by the arcs $(p(v), v)$. All arcs of the parent graph have non-positive reduced lengths. All arcs out of scanned vertices have non-negative reduced lengths. To reinitialize if some arc lengths change, we must change to labeled the status of vertices with outgoing arcs of negative reduced length.

The simplest scanning method is Bellman-Ford-Moore (BFM) [1, 9, 15], which maintains labeled vertices in a queue: each newly labeled vertex is added to the back, and the next vertex to be scanned is removed from the front. The algorithm finishes when the queue becomes empty (there is no cycle) or after n passes over the queue (there is a cycle). We define *passes* inductively: pass 0 is the initialization, and pass i for $i > 0$ is the scanning of vertices labeled during pass $i - 1$. This algorithm can be generalized to a class of $O(n)$-pass algorithms [3].

A more efficient variant in practice, due to Tarjan [16], is BFCT (see also [4]). It also maintains labeled vertices in a queue, but uses *subtree disassembly* to detect cycles faster. It keeps the current shortest path tree as a doubly-linked list representing its preorder traversal. When the processing of an arc (u, v) results in a reduction of $\pi(v)$, the algorithm traverses the subtree rooted at v looking for u. If u is in this subtree, there is a negative cycle; otherwise, the algorithm disassembles the subtree and marks all its vertices except for v as unreached. The disassembly does not affect the correctness of the algorithm, because all the vertices marked unreached will later have their potentials decreased and hence become labeled. The cost of traversing the subtree can be charged to the cost of building it, thus preserving the $O(nm)$ worst-case time of the algorithm (the same as BFM).

A third variant is *robust Dijkstra* (RDH), introduced by Cherkassky et al. [3]. It is also an $O(n)$-pass algorithm that uses subtree disassembly. Instead of using a queue, however, RDH maintains labeled vertices in a heap. The priority of a vertex is given by the improvement in its potential since it was last scanned.

Vertices with larger improvements are scanned sooner. This is a generalization of Dijkstra's algorithm, and its worst-case running time is $O(nm \log n)$. In practice, however, it is often faster than BFCT, especially when a negative cycle is present.

3 MMC Algorithms

We now describe the MMC algorithms we study, including new variants. Precision issues are deferred to Section 4.

3.1 Initialization. The algorithms we discuss maintain an upper and/or a lower bound on λ^*. One can use U and $-U$ as naïve bounds. A potentially better upper bound can be found with *greedy initialization* (see Dasdan [6]): just pick the minimum cycle mean of the subgraph obtained by taking the minimum-length arc out of each vertex (if the subgraph is acyclic, simply use U as an upper bound). The MMC can be found in linear time with a simple depth-first search (DFS) on the original graph, ignoring all but the shortest arc out of each vertex.

We get a slightly stronger bound using a similar procedure, with two modifications. First, while scanning a vertex v (to determine the best outgoing arc), we check whether each arc (v, w) is a back arc, i.e., if w occurs before v on the current path; if so, we compute the corresponding cycle mean and update the upper bound on λ^*. Second, when choosing the outgoing arc in each step, we pick the minimum-length arc *leading to an unvisited vertex*. This allows us to grow longer paths, thus examining more back arcs (and cycles). These changes ensure that we always examine a superset of the cycles checked by Dasdan's initialization, while still scanning each vertex only once. We call our bound *enhanced greedy*.

3.2 Cycle-based. The simplest solution to the MMC problem is arguably the *cycle-based algorithm* (CYCLE). It maintains an upper bound $\lambda \geq \lambda^*$ and a cycle Γ with $\mu(\Gamma) = \lambda$. At each iteration, the algorithm runs an SPF subroutine for the length function $\ell - \lambda$. If a negative cycle C is found, we set $\Gamma \leftarrow C$ and $\lambda \leftarrow \mu(C)$. Otherwise, the current λ is optimal and the algorithm terminates. No polynomial bound is known for the generic CYCLE algorithm. We can prove polynomial bounds for specific implementations, but they are not very good.

A well-known implementation of CYCLE is Howard's algorithm [11]. As described by Dasdan [6], it uses the greedy initialization to find the first value of λ, then makes BFM passes through vertices. At the end of each pass, the parent graph is checked for cycles. If no cycle is found, the algorithm continues with the next pass or terminates if all reduced lengths for the length function $\ell - \lambda$ are non-negative. If cycles are found, λ is updated to make the most negative one have zero length, and the next pass starts. Dasdan [6] describes heuristics that improve algorithm performance.

Our implementations of CYCLE use BFCT and RDH as SPF subroutines, which are more efficient in practice than BFM. We use the enhanced greedy initialization.

3.3 Binary Search. Due to Lawler [14], the *binary search algorithm* (BINSEARCH) maintains two values, λ_1 and λ_2, such that $\lambda_1 \leq \lambda^* \leq \lambda_2$, and a cycle Γ with $\mu(\Gamma) = \lambda_2$. At each step, the algorithm sets λ to the average of λ_1 and λ_2 and tests whether the length function $\ell - \lambda$ is feasible. If a negative cycle is found, we set Γ to the cycle and λ_2 to $\mu(\Gamma)$ (which is less than λ). Otherwise we set $\lambda_1 \leftarrow \lambda$. The algorithm terminates when $\lambda_2 - \lambda_1 < \epsilon$, where ϵ is the desired approximation. If ϵ is sufficiently small (as explained in Section 4), the last Γ will be a minimum mean cycle. The number of iterations is $O(\log(nU))$.

The binary search algorithm has two related problems in practice. First, it is often the case that the initial value of λ_1 is too conservative, i.e., very far from the actual minimum cycle mean. Second, the algorithm often finds a minimum mean cycle well before the termination condition is satisfied, i.e., when λ_1 and λ_2 are still far apart. From this point on, only λ_1 will change. To speed up convergence, we tried a variant called *biased search*. Instead of taking a simple average of λ_1 and λ_2 in a given step, we actually set $\lambda \leftarrow \frac{\lambda_1 + k\lambda_2}{1+k}$, where k is a positive integer. When $k > 1$ this biases the step towards λ_2. The value of k is updated after each step: starting at 1, k doubles when a cycle is found, and is halved when none is (the minimum allowed value is 1).

3.4 Hybrid. One can also deal with the convergence issue by combining binary (or biased) search with the cycle-based algorithm. We call this approach HYBRID. It has the same worst-case guarantees as binary search ($O(\log(nU))$ iterations), and always finishes at most two iterations after finding an optimal cycle.

Like BINSEARCH, HYBRID maintains two values, λ_1 and λ_2 (with $\lambda_1 \leq \lambda^* \leq \lambda_2$), and a cycle Γ with $\mu(\Gamma) = \lambda_2$. Each step does an SPF test, with a new value of λ that depends on the result of the previous test: if a negative cycle was found (or if this is the first step), the new value is $\lambda = \frac{\lambda_1 + \lambda_2}{2}$; otherwise, it is $\lambda = \lambda_2$ (this forces the algorithm to check if the current value of λ_2 is optimal). If a cycle C is found with the new value of λ, the algorithm sets $\Gamma \leftarrow C$ and $\lambda_2 \leftarrow \mu(C)$; otherwise (if no cycle is found), it sets $\lambda_1 \leftarrow \lambda$. The

algorithm stops as soon as two consecutive steps find no negative cycle.

3.5 Tree-based. We refer to BinSearch, Cycle, and Hybrid as *SPF-based* methods, since they use a full SPF algorithm as a subroutine. We now discuss a *parametric* algorithm. Due to Young et al. [17] (improving on Karp and Orlin [13]), the *tree-based* algorithm (Tree) starts from a value of λ small enough to ensure all edges have positive length, and progressively increases λ (in carefully chosen increments) until a zero cycle is created.

Tree explicitly maintains the shortest path tree corresponding to the current value of λ. Initially, all vertices are children of the root s (in the augmented graph). As λ increases, the tree gets deeper.

Let λ_i be the value of λ after step i, and suppose no zero cycle has been detected yet. Define λ_{i+1} as the smallest $\lambda \geq \lambda_i$ that leads to a different shortest path tree. We associate with each vertex v a *breakpoint*, the smallest value of λ that would cause it to switch parents (assuming there is no other change to the tree). The smallest such breakpoint, over all vertices, is λ_{i+1}; we use a heap to find it.

Each vertex keeps two pieces of information: the number of arcs on the path from the root and the total (original) length of this path. We can then compute the breakpoint of any vertex v based only on information stored at v and at the tails of its incoming arcs.

When a vertex v changes parent, the entire subtree T_v rooted at v gets at least one arc further from the root. Each vertex $w \in T_v$ must be scanned twice: first, we look at its incoming arcs to update w's own breakpoint; then we look at its outgoing arcs to update the breakpoints of its neighbors (with more arcs from the root, w now looks more attractive). We also check if the new parent of v belongs to T_v: if it does, a zero cycle has been found, and the algorithm terminates.

Note that while SPF-based methods need only outgoing arc lists for every vertex (forward graph), the Tree method needs both incoming and outgoing lists (forward and reverse graphs).

Each vertex is scanned at most $O(n)$ times, so the algorithm spends $O(nm)$ time in scanning operations and performs $O(n^2)$ heap insertions and deletions. It may also perform $O(nm)$ breakpoint (heap) updates. With Fibonacci heaps, the total running time is $O(n^2 \log n + nm)$. Our implementation uses 4-heaps, which are more practical, and runs in $O(nm \log n)$ time.

4 Precision Issues

The following discussion and our implementation assume that distances in the input graph can be represented as 32-bit integers and that $n < 2^{32}$, which is reasonable for many applications. Our techniques, however, apply to arbitrary precision.

Even assuming integrality, mean cycle lengths λ are, in general, rational. Since we work with length functions of the form $\ell - \lambda$, we must deal with rational arithmetic or work with rounded length values, either implicitly (with floating-point arithmetic) or explicitly. This section discusses each technique in turn.

4.1 Floating-Point Representation. Suppose we want to approximate λ^* to an additive parameter $\epsilon > 0$. SPF-based MMC algorithms can use a standard technique (also used in [6]). We modify the scan operation so that $\pi(w)$ is updated only if it improves by at least ϵ: if $\pi(v) + \ell(v,w) < \pi(w) - \epsilon$, then set $\pi(w) \leftarrow \pi(v) + \ell(v,w)$ and $S(w) \leftarrow$ labeled. Although the algorithm may terminate with a non-optimal solution, reduced lengths will be at least $-\epsilon$, allowing λ^* to be approximated with ϵ precision. If ϵ is small enough, this technique solves the problem exactly:

Theorem 4.1. *With integral lengths, any two distinct mean cycle lengths are at least* $\frac{1}{n(n-1)}$ *apart.*

Proof. Take two cycles with mean lengths $\frac{c_1}{k_1} > \frac{c_2}{k_2}$, with c's and k's integral and $0 < k_1, k_2 \leq n$. Then

$$\frac{c_1}{k_1} - \frac{c_2}{k_2} = \frac{c_1 k_2 - c_2 k_1}{k_1 k_2}.$$

Note that the difference is at least $\frac{1}{k_1} \geq \frac{1}{n}$ if $k_1 = k_2$, and at least $\frac{1}{n(n-1)}$ otherwise. □

Therefore, if we use $\epsilon < \frac{1}{n(n-1)}$, the algorithm will find an optimal solution. This bound is tight, as we can have two cycles of length 1, with $n-1$ and n arcs.

Because floating point numbers have limited precision, ϵ cannot be arbitrarily small. Otherwise, rounding errors may cause the algorithm to terminate with an error higher than ϵ, or not terminate at all.

4.2 Integer Representation. An alternative approach to precision issues is to multiply all lengths by $n(n-1)$ in the beginning, then work only with integers throughout the algorithm. Since values of λ returned by the SPF algorithm may be fractional, they must be rounded down to the nearest integer. The modified length function $\ell - \lfloor \lambda \rfloor$ is then guaranteed to be integral, and the difference between two distinct mean cycle lengths will be at least one. Furthermore, a cycle is negative with respect to the input lengths if and only if it is negative with respect to the multiplied lengths.

Define $\ell_\lambda = \ell - \lambda$ and $\ell'_\lambda = \ell - \lfloor \lambda \rfloor$. We can show that rounding does not change the sign of cycle lengths:

LEMMA 4.1. *Suppose λ is the mean length of some cycle Γ in G: $\lambda = \mu(\Gamma)$. Let C be a cycle in G with $\ell_\lambda(C) \neq 0$. Then $\ell_\lambda(C) \cdot \ell'_\lambda(C) > 0$.*

Proof. Clearly $\ell_\lambda(C) \leq \ell'_\lambda(C)$; if the former is positive, so is the latter. Next we show that if $\ell_\lambda(C) < 0$, then $\ell'_\lambda(C) < 0$. If $\ell_\lambda(C) < 0$, then $\mu(C) < \lambda = \mu(\Gamma)$. Since cycle mean values are at least one apart after we multiplied the length function, $\mu(C) \leq \lambda - 1 < \lfloor\lambda\rfloor$. Thus $\ell'_\lambda(C) < 0$. □

Next we show that this rounding technique does not affect the correctness or the asymptotic worst-case performance of SPF-based algorithms.

For CYCLE, recall that the algorithm maintains a cycle Γ and $\lambda = \mu(\Gamma)$. If there is a negative cycle with respect to the length function $\ell - \lambda$, it is also negative with respect to $\ell - \lfloor\lambda\rfloor$ by Lemma 4.1, and the algorithm will find a negative cycle Γ' with respect to $\ell - \lfloor\lambda\rfloor$. Since $\mu(\Gamma') < \lfloor\lambda\rfloor \leq \lambda$, the algorithm makes progress. If the algorithm fails to find a negative cycle, Γ is a zero cycle, and there are no negative cycles.

For BINSEARCH, note that λ_1 and λ_2 are rounded, and the algorithm terminates when $\lambda_1 = \lambda_2$ (assuming ϵ is sufficiently small). It is easy to see that $\lambda_2 - \lambda_1$ decreases by a constant factor in each iteration; the constant is at least 3/2 when $\lambda_2 - \lambda_1 \geq 2$. What is left to show is that the cycle Γ is optimal when the algorithm terminates. Recall that $\lambda_2 = \lfloor\mu(\Gamma)\rfloor$, and that the length function $\ell - \lambda_1$ admits no negative cycle. Suppose that the algorithm terminates but there is a cycle C with $\mu(C) < \mu(\Gamma)$. Then $\mu(C) \leq \mu(\Gamma) - 1 < \lambda_2 = \lambda_1$, a contradiction. A similar argument shows that HYBRID is correct.

Note that, to achieve ϵ precision (for $\epsilon > 0$), we need to multiply the lengths by $\lceil 1/\epsilon \rceil$. In particular, given the input and the precision of the arithmetic operations, one can select the smallest appropriate value of ϵ. With the same number of bits, we get more precision with integers than floating-point numbers, since no bits are wasted on exponents.

4.3 Rational Representation. The tree-based algorithm represents potentials in parametrized form. Each vertex stores a pair (c, d) representing the number of arcs c and the total length d of the path from the root. Since the inputs are integers, so are c and d. The actual potential, $d - c\lambda$, depends on the current value of λ, and may be a rational.

The algorithm must also deal with breakpoints. Given a vertex v, a breakpoint is a value of λ that would cause v to switch parents in the current shortest path tree. In other words, given two parametrized potentials (c', d') and (c'', d'') (induced by two possible parents of v), the breakpoint is the value of λ such that $d' - c'\lambda = d'' - c''\lambda$. Here, $\lambda = \frac{d''-d'}{c''-c'}$ is a rational number that can be represented as a pair of integers. To compare breakpoints (in the heap), we must perform comparisons of the form $\frac{a'}{b'} < \frac{a''}{b''}$. This is essentially the same as checking if $a'b'' < a''b'$, with special cases when denominators are zero.

While our experiments assume it is enough to use 64 bits for this computation, in general it is not. If the original arc lengths use B bits and n uses N bits, path lengths require up to $B + N$ bits, and the comparison above ($a'b'' < a''b'$) needs $B + 2N$ bits. Note that an exact solution using integers needs even more: up to $B + 3N$ bits to represent the modified path lengths (from Theorem 4.1). For moderate values $B = N = 20$, 64 bits are enough with rationals, but not with integers.

4.3.1 SPF-based Algorithms. The rational representation can also be used to implement parametrized versions of SPF-based MMC algorithms (CYCLE, BINSEARCH, or HYBRID). Instead of storing a potential as a single number (either floating point or integer), we keep a pair (c, d) of integers, representing the number of arcs and the length of the path from the root. As in TREE, two potentials are compared using cross multiplication.

There is an additional issue to consider for BINSEARCH (and HYBRID). In CYCLE and TREE, every value of λ ever considered corresponds to an actual cycle or path in the graph. In contrast, BINSEARCH computes values of λ that are (weighted) averages of two other values (and averages of averages as the algorithm progresses). To address this issue, whenever we compute an average we replace it by a low-precision rational (a pair of 32-bit integers, in our implementation) calculated using continued fractions. When λ_1 and λ_2 are very close, however, this method may fail to yield a new value of λ that is actually between λ_1 and λ_2; when this happens, we simply switch to CYCLE.

SPF-based algorithms with rationals are fundamentally different from those using integers or floating-point numbers. Being parametric functions of λ, potentials implicitly change whenever λ does.

5 Incremental Restarts

SPF algorithms assign potentials to vertices in an attempt to prove that no negative cycle exists. If a negative cycle is found, an SPF-based MMC algorithm will adjust the value of λ and call the SPF routine again. One could restart with a brand new set of potentials (typically zero); we call this a *full restart*. In practice, however, it is often useful to reuse the potentials found in the previous call, performing a *partially incremental restart*. When the change in λ is small, this could spare

the algorithm from recomputing similar potentials, allowing it to concentrate on important parts of the graph. The algorithm described in [2], for example, is an implementation of BinSearch that preserves the potentials and uses Tarjan's SPF algorithm [16].

We take this idea one step further. SPF algorithms only need to scan vertices that may have outgoing arcs with negative reduced lengths. We call these *candidate vertices*; a vertex is added to this set when its potential changes (and during initialization), and removed right after it is scanned. When an SPF routine finds a negative cycle, the next call will use a smaller value of λ. If we preserve the potentials between calls, reduced lengths can only increase. Therefore, only vertices in the candidate set after the first call need to be marked as labeled for the second—no other vertex will have outgoing arcs with negative reduced lengths. This avoids the need to examine the entire graph during a restart.

Note that this *incremental reinitialization* technique can only be used when λ decreases, which happens in every iteration of Cycle, but only in some iterations of BinSearch and Hybrid. When λ increases, some reduced lengths may become negative, and all vertices must be marked as labeled (though potentials are preserved).

Another limitation of this technique is that it cannot be safely used with the parametrized implementation described in Section 4.3.1. Because potentials implicitly change with λ (by different amounts), any arc may acquire a negative reduced length. Instead of simply abandoning incremental reinitialization, we tried a more aggressive approach. When λ decreases, we still call the SPF algorithm with incremental reinitialization. If it does find a cycle, we proceed safely. If it does not, the result cannot be trusted: we mark all vertices as labeled and redo the computation using the new potentials as a starting point.

6 Pathological Instances

This section describes some graph families designed to be hard for some of the algorithms we study. In each family, a single parameter k controls the instance size: the number of vertices is always a linear function of k, and for most families this is also true for the number of arcs.

To simplify our constructions, we assume that the initialization phase returns an initial guess for λ^* that is larger than the mean length of any actual cycle in the network. To justify this assumption we note that it is relatively easy to modify any graph so as to force the greedy initialization to find an acyclic graph; we can then add an artificial cycle (disjoint from the original graph) with the desired mean length for the initial guess.

6.1 Cycle-based algorithm. We start with pathological instances for the BFCT version of the Cycle algorithm. We first present a sparse graph that forces BFCT to perform $\Theta(n)$ iterations, then give a dense graph that forces $\Theta(n^2)$ iterations. Recall that tight bounds are not known for the Cycle algorithm.

The first graph consists of k arc-disjoint triangles; the ith triangle has vertices x_i, y_i, and z_i and arcs (x_i, y_i), (y_i, z_i), and (z_i, x_i), all of length $-i$. Successive triangles have a vertex in common, i.e., $z_i = x_{i+1}$. By choosing the vertex identifiers appropriately, we guarantee that BFCT finds the cycle with the next largest mean first. To that end, we set $x_i = 2(i-1)+1$, $y_i = 2i$, and $z_k = 2k - 1$. Moreover, assuming full restart at the beginning of each iteration, BFCT will perform $\Theta(n)$ scans on average to find the next cycle, resulting in $\Theta(n^2)$ running time. We refer to this family as BAD1. See Figure 1.

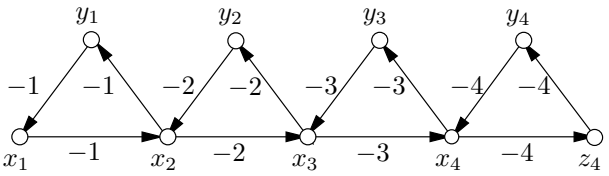

Figure 1: Bad-case instance for Cycle with $k = 4$ (BAD1).

By making the graph more complicated, we can in fact get quadratic running time even when we use incremental reinitialization. Augmenting the main path with some reverse arcs prevents the more sophisticated algorithms from finding multiple cycles in a single pass. We refer to the modified family as BAD2.

Another network that gives similar results when full restart is used is the following. We start with a path of k vertices $P = (x_1, x_2, \ldots, x_k)$, where $\ell(x_i, x_{i+1}) = \ell(x_{i-1}, x_i) - 1 = l - i$ (where l is a parameter to be chosen later). We add a vertex y_1 together with the arcs (x_k, y_1) and (y_1, x_i) for $1 \le i < k$. Let C_i^1 denote the cycle $(x_i, x_{i+1}, \ldots, x_k, y_1, x_i)$. We choose the arc lengths so that $\ell(C_i^1) = -i(k - i + 2)$, and hence $\lambda(C_i^1) = -i$. To that end, we set $l = 0$, $\ell(x_k, y_1) = -k$, and $\ell(y_1, x_i) = -i(k - i + 2) + k(k + 1)/2 - i(i - 1)/2$. To get the desired bound we need to assign appropriate identifiers to the vertices, and order the adjacency list of y_1 suitably. It suffices to set $x_i = i$, $y_1 = k + 1$, and order the adjacency list of y_1 by increasing identifier. See Figure 2. Then C_i^1 is discovered before C_{i+1}^1 and we get $k - 1$ iterations. Furthermore, each iteration needs $\Theta(k)$ scans, assuming full restarts.

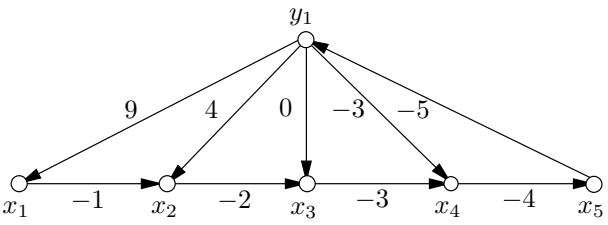

Figure 2: A quadratic-time instance for CYCLE with $k = 4$. By replicating vertex y_1 and its adjacent arcs k times (and modifying the lengths suitably), we get a dense graph that causes $\Theta(n^3)$ scans (BAD3).

We now extend the above network to achieve $\Theta(n^2)$ iterations. First, we add the vertices y_i, for $2 \leq i \leq k$. Then, for each y_i, we add the arcs (x_k, y_i) and (y_i, x_j), where $1 \leq j < k$. We let C_i^j denote the cycle $(x_i, x_{i+1}, \ldots, x_k, y_j, x_i)$. Here our goal is to have $\lambda(C_i^j) = -(k-1)j - i$, which can be accomplished similarly. This network gives $\Theta(n^3) = \Theta(mn)$ scans. We refer to this family as BAD3.

6.2 Tree-based algorithm. We now describe a family of sparse graphs that causes the worst-case running time of TREE: the shortest path tree changes $\Theta(n^2) = \Theta(mn)$ times.

The network consists of a source s, a sink t, three sets of k vertices each ($\{x_i\}$, $\{y_i\}$, and $\{z_i\}$), and two more vertices w_1 and w_2. We form the paths $P_1 = (x_1, \ldots, x_k)$ and $P_2 = (y_1, \ldots, y_k)$, with $\ell(x_i, x_{i+1}) = \ell(y_i, y_{i+1}) = i$. The two paths are connected to the source with the arcs (s, x_1) and (s, y_1). Also, we connect the odd vertices $(x_1, x_3, \ldots)$ of P_1 to w_1 and the even vertices $(y_2, y_4, \ldots)$ of P_2 to w_2. Then, from w_1 and w_2 there is an arc to each z_i, and an arc from z_i to t. All these arcs have zero length. Finally, we include the arc (t, s) with length high enough so that $\lambda^* \geq k$ and the cycle with mean λ^* has $\Theta(k)$ vertices: setting $\ell(t, s) = k^2$ satisfies these requirements. See Figure 3.

As λ increases, the shortest path to w_1 includes more and more vertices of P_1. Similarly, the shortest path to w_2 includes more and more vertices of P_2. Furthermore, at every moment these paths have different lengths and different numbers of arcs. The result is that the parent of each z_i in the shortest path tree keeps alternating between w_1 and w_2, changing $\Theta(k^2)$ times in total. We refer to this family as BAD4.

6.3 Quasi-DAG. We also include in our collection of pathological instances a family of dense graphs constructed as follows: we start with a complete DAG on n vertices, i.e., for any $i < j$ we have an arc from i to j with length $1 + j - n$. We then add the cycle arc $(n, 1)$ with length 0. We call this family BAD5. Its graphs have many cycles with different numbers of arcs and lengths.

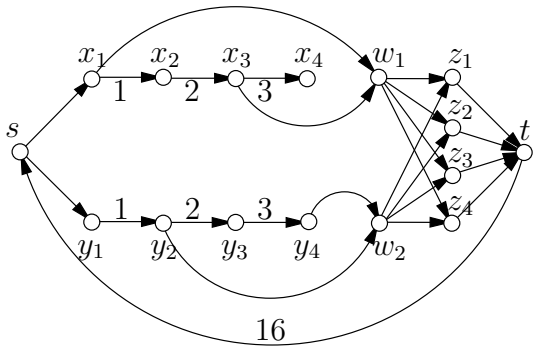

Figure 3: Worst-case family for TREE (BAD4) with $k = 4$. Only nonzero arc lengths are shown.

7 Experimental Results

7.1 Methodology. All algorithms were implemented in C++ and compiled with Microsoft Visual C++ 2008 with full optimization. Timed tests were conducted on a 2.4 GHz AMD Opteron machine with 16 GB of RAM, running Windows Server 2003 Enterprise x64 edition. All input values are 32-bit integers, but we used 64 bits to represent intermediate values internally. Running times do not include reading the input graph (which is done by all algorithms), but do include allocating, initializing, and destroying additional data structures (including the reverse graph needed by TREE).

We measured not only running times, but also the average number of scans per vertex (for TREE, a scan is a traversal of an adjacency list), which provides a machine-independent measure of performance. We did not count scans done during initialization, one per vertex for every algorithm.

Unless otherwise noted, we used an approximation factor of $\epsilon = 10^{-6}$ for the SP feasibility-based algorithms (CYCLE, BINSEARCH, and HYBRID). Note that the TREE algorithm does not use ϵ and always finds an optimal solution. For the instances tested, $\epsilon = 10^{-6}$ was small enough for all algorithms to find an optimal solution, i.e., a cycle with the same mean length as the one found by the TREE algorithm.

7.1.1 Algorithms. Although we have presented only four basic algorithms (CYCLE, TREE, HYBRID, and BINSEARCH), we actually ended up testing 76 distinct implementations, varying aspects such as the SP feasibility algorithm, the data representation, and the restart method. For ease of exposition, our initial focus is on a set of five variants that achieved good results. It in-

cludes two variants of CYCLE (using BFCT and RDH); two versions of HYBRID (using BFCT and RDH); and TREE. The first four algorithms use incremental reinitialization and represent potentials as integers. Both implementations of HYBRID use biased search. Section 7.4 addresses other variants.

7.1.2 Problem families. Our basic tests were conducted on two real-world problem families representing clocking problems on circuits. The first family, IBM, was provided by Ali Dasdan and includes sparse graphs with up to 210 613 vertices. The second, BONN, was provided by Stephan Held and contains denser graphs with millions of edges. See [6, 10] for details.

Because these instances are rather easy for most algorithms, we also tested synthetic families originally used in the evaluation of algorithms for the shortest path problem [4], and later for SP feasibility [3].

Created with the SPRAND generator [5], the rand5 family contains graphs with n vertices and $m = 5n$ arcs. Each graph consists of a random Hamiltonian cycle and $m - n$ additional random arcs, with lengths chosen uniformly at random from $[1, 1000]$.

For more structured instances, we used the TOR generator [4]. In *2-dimensional grids* with wrap-around, each vertex is connected to its neighbor above (in the same column) by a short arc (with length in $[1, 100]$), and to its neighbor to the right (in the next column) by a long arc (length in $[1000, 10000]$). We tested two families: sqnc consists of *square grids*, with $\lfloor\sqrt{n}\rfloor$ columns and rows, and lnc consists of *long grids*, with $\lfloor n/16\rfloor$ columns and 16 rows.

A third family generated by TOR, denoted by pnc, consists of *layered networks* embedded on a torus. Graphs are partitioned into layers, each containing a cycle of length 32 plus 64 random arcs (all with lengths in the range $[1, 100]$). Each vertex also has five arcs to forward layers (with wrap-around): an arc going x layers forward has length picked uniformly at random from $[1, 10000]$ and multiplied by x^2. All TOR families have a *source* connected to the first layer/column by zero-length arcs.

Finally, we tested *road networks* [8], planar graphs in which vertices represent intersections, arcs are road segments, and lengths are proportional to travel distances. We tested two instances, representing New York City ($n = 264\,346$) and Florida ($n = 1\,070\,376$).

7.1.3 Subfamilies. For a more systematic analysis, we explicitly added negative cycles in a controlled fashion. Since all original families have no negative arcs, these cycles necessarily change the minimum mean cycle.

Our filter adds zero or more disjoint negative cycles (connecting randomly chosen vertices) to a graph. We tested five subfamilies in which each added cycle has one arc of length -1 and the rest of length 0. These subfamilies are similar to those studied in [3]: (01) no cycles; (02) a single small (three arcs) cycle; (03) $\lfloor\sqrt{n}\rfloor$ small cycles; (04) $\lfloor\sqrt[3]{n}\rfloor$ medium ($\lfloor\sqrt{n}\rfloor$ arcs) cycles; (05) a single Hamiltonian cycle. In addition, subfamily (06), which is new, has cycles of varying length. Let $M = \lfloor\sqrt[3]{n}\rfloor$: we add M cycles with M, $2M$, ..., M^2 arcs. Every added arc has length $-M$, except for a single arc in each cycle with length $-M + 1$. Note that the minimum mean cycle is the one with the most arcs.

After the cycles are added to each subfamily, we "hide" them. A *potential perturbation* assigns a random value in $[0, 16384)$ to each vertex and replaces arc lengths by their reduced lengths. We also randomly permute all vertex identifiers and adjacency lists. These transformations do not change the minimum mean cycle of the graph.

For experiments with a randomized component (all that involve synthetic instances or subfamilies), we report averages of 25 runs with different random seeds.

7.2 Circuits. We start our experiments with BONN and IBM, the real-world families. The IBM instances (but not BONN) were originally created for the minimum ratio cycle (MRC) problem: each arc has both a length and a positive *delay*, and the goal is to find a cycle minimizing the ratio between its total length and its total delay. We transformed them into MMC instances by setting all delays to one. The average number of scans performed by each algorithm on IBM instances are shown in Table 1 (for the MMC problem) and Table 2 (for MRC). The corresponding results for BONN (for MMC only) are shown in Table 3.

In most cases, CYCLE and TREE perform slightly more than one scan per vertex. HYBRID usually performs slightly more than two scans on IBM, but one scan has the sole purpose of detecting optimality, which means the optimal cycle itself is found after little more than one scan per vertex. BONN is only slightly harder for HYBRID. On both families, the minimum cycle usually has three or fewer arcs, and never more than five. These problems are too easy for a meaningful comparison between the algorithms.

All algorithms discussed in this paper easily extend to the MRC problem, but without good theoretical performance guarantees. On the IBM family, however, solving the MRC problem instead of MMC makes little difference in practice, confirming Dasdan's observation [6]. We restrict ourselves to the MMC problem for the remainder of this paper, but note that every MMC

Table 1: Minimum Mean Cycle: Average number of scans per vertex on IBM.

instance			Cycle		Hybrid		Tree
name	n	m	bfct	rdh	bfct	rdh	
ibm01	12752	36681	**1.04**	**1.04**	2.06	2.07	1.17
ibm02	19601	61829	**1.01**	**1.01**	2.02	2.02	1.04
ibm03	23136	66429	**1.01**	**1.01**	2.02	2.02	1.05
ibm04	27507	74138	**1.01**	**1.01**	2.01	2.01	1.02
ibm05	29347	98793	**1.00**	**1.00**	2.00	2.00	1.01
ibm06	32498	93493	**1.01**	**1.01**	2.02	2.02	1.06
ibm07	45926	127774	**1.03**	**1.03**	2.04	2.04	1.11
ibm08	51309	154644	**1.00**	**1.00**	2.01	2.01	1.02
ibm09	53395	161430	**1.03**	**1.03**	2.04	2.04	1.10
ibm10	69429	223090	1.03	**1.01**	3.04	3.04	1.06
ibm11	70558	199694	**1.01**	**1.01**	2.02	2.02	1.05
ibm12	71076	241135	**1.01**	**1.01**	2.01	2.01	1.02
ibm13	84199	257788	**1.01**	**1.01**	2.01	2.01	1.03
ibm14	147605	394497	**1.00**	**1.00**	2.01	2.01	1.02
ibm15	161570	529562	**1.02**	**1.02**	2.04	2.04	1.10
ibm16	183484	589253	**1.01**	**1.01**	2.02	2.02	1.05
ibm17	185495	671174	**1.00**	**1.00**	2.00	2.00	**1.00**
ibm18	210613	618020	**1.00**	**1.00**	2.01	2.01	1.02

Table 2: Minimum Ratio Cycle: Average number of scans per vertex on IBM for the minimum cycle ratio problem.

instance			Cycle		Hybrid		Tree
name	n	m	bfct	rdh	bfct	rdh	
ibm01	12752	36681	**1.03**	**1.03**	2.04	2.04	1.11
ibm02	19601	61829	**1.01**	**1.01**	2.02	2.02	1.04
ibm03	23136	66429	**1.01**	**1.01**	2.02	2.02	1.05
ibm04	27507	74138	**1.01**	**1.01**	2.01	2.01	1.02
ibm05	29347	98793	**1.00**	**1.00**	2.00	2.00	1.01
ibm06	32498	93493	**1.01**	**1.01**	2.02	2.02	1.06
ibm07	45926	127774	**1.02**	**1.02**	2.03	2.03	1.09
ibm08	51309	154644	**1.00**	**1.00**	2.01	2.01	1.02
ibm09	53395	161430	**1.02**	**1.02**	2.03	2.03	1.09
ibm10	69429	223090	1.03	**1.01**	3.04	3.04	1.06
ibm11	70558	199694	**1.01**	**1.01**	2.02	2.02	1.05
ibm12	71076	241135	**1.01**	**1.01**	2.01	2.01	1.02
ibm13	84199	257788	**1.01**	**1.01**	2.01	2.01	1.03
ibm14	147605	394497	**1.00**	**1.00**	2.01	2.01	1.02
ibm15	161570	529562	**1.02**	**1.02**	2.03	2.03	1.08
ibm16	183484	589253	**1.01**	**1.01**	2.02	2.02	1.05
ibm17	185495	671174	**1.00**	**1.00**	2.00	2.00	**1.00**
ibm18	210613	618020	**1.00**	**1.00**	2.01	2.01	1.02

instance is a valid MRC instance.

7.3 Other Families. We now consider more challenging instances. We picked the largest IBM instance, which has 210 613 vertices, and instances of comparable size from other families: $n = 262\,144$ (2^{18}) for rand5, $n = 2^{18} + 1$ for TOR-generated families (lnc, pnc, and sqnc) and $n = 264\,346$ (New York) for the road family. (Section 7.6 will consider different sizes.)

Table 4 shows the number of scans per vertex on these families, each with six subfamilies. In every case, the best method is either Cycle or Tree; Hybrid is consistently worse. The Cycle algorithm is more robust with rdh than with bfct: it is never much worse on "easy" families, but sometimes significantly better when the number of scans is higher. For example, on subfamily 05 of pnc, Cycle performs almost an order of magnitude fewer scans with rdh.

Overall, these synthetic instances are harder than the real-world instances tested in Section 7.2. In particular, the original ibm18 instance requires fewer scans per vertex than any of its corresponding subfamilies—including subfamily 01, which adds a potential perturbation but no negative cycle.

Note, however, that even these synthetic families are not too hard: no algorithm performs more than 150 scans per vertex on average. In particular, both Tree and Cycle with rdh need no more than 26 scans per vertex. All algorithms consistently perform fewer scans when the negative cycles are short (subfamilies 02 or 03) rather than long (04 to 06).

Table 3: Minimum Mean Cycle: Average number of scans per vertex on BONN.

instance			Cycle		Hybrid		Tree
name	n	m	bfct	rdh	bfct	rdh	
bonn01	4609	2916202	1.04	**1.02**	4.00	4.03	**1.02**
bonn02	5361	4166868	1.44	1.42	3.11	3.26	**1.36**
bonn03	11867	8914917	**1.18**	1.76	2.00	2.09	1.31
bonn04	20072	1321426	**1.05**	1.12	3.05	3.10	1.13
bonn05	60309	3969429	**1.00**	1.03	4.00	4.03	1.04
bonn06	70346	892588	**1.01**	**1.01**	4.01	4.01	1.03
bonn07	82038	2603211	**1.00**	1.03	3.00	5.07	1.14
bonn08	95936	1324128	**1.08**	1.20	2.08	2.25	1.60
bonn09	346814	9222546	**1.00**	1.01	5.00	4.02	1.02

Despite its good worst-case bound, Hybrid is usually worse than Cycle, and never much better. Because at least half of its iterations use Cycle, however, it is rarely much worse, either (one exception is the rdh variant on subfamily 01 of sqnc).

Table 5 reports the average execution times on the same instances. The fastest code is usually either Tree or the bfct variant of Cycle. Because the rdh variant of Cycle has higher overhead per scan (due to the use of a heap), it is often slower than the bfct variant when the number of scans is similar. It is more competitive for harder instances, but on such instances Tree tends to be even faster.

7.4 Variants. We now analyze how the SPF-based algorithms are affected by the design choices we made.

Table 4: Average number of scans per vertex on instances with approximately 2^{18} vertices.

		Cycle		Hybrid		Tree
family	sub	bfct	rdh	bfct	rdh	
ibm18	01	**1.66**	1.75	2.95	3.09	3.33
	02	**1.67**	1.76	3.30	3.40	3.32
	03	**1.66**	1.73	2.75	2.82	3.35
	04	12.06	**7.39**	19.44	13.07	8.23
	05	49.86	23.79	61.15	26.01	**20.91**
	06	37.32	12.48	45.69	22.86	**12.45**
lnc	01	2.36	**2.30**	6.89	7.17	4.80
	02	2.36	**2.21**	4.24	5.71	4.65
	03	2.29	**2.11**	3.36	3.17	4.71
	04	17.69	**10.62**	29.04	15.81	13.36
	05	70.65	24.54	61.94	24.71	**22.86**
	06	47.81	**14.13**	46.23	22.10	14.15
pnc	01	2.99	**2.91**	4.14	4.07	7.82
	02	2.87	**2.85**	4.90	4.98	7.24
	03	2.89	**2.88**	4.04	4.05	7.74
	04	23.38	**12.89**	32.92	17.24	20.42
	05	148.53	**19.19**	73.42	21.96	19.57
	06	77.17	14.89	59.46	22.34	**14.80**
rand5	01	10.90	**6.60**	14.71	10.67	7.80
	02	**2.66**	3.10	5.29	6.04	5.34
	03	**2.15**	2.47	3.30	3.64	5.37
	04	31.84	**13.04**	38.91	16.30	13.30
	05	70.83	25.37	69.29	25.31	**23.07**
	06	50.90	**14.04**	53.70	22.04	14.17
road	01	**1.64**	1.72	2.74	2.82	3.32
	02	**1.65**	1.75	3.42	3.53	3.31
	03	**1.65**	1.73	2.75	2.83	3.34
	04	6.52	**4.11**	10.68	6.55	7.57
	05	43.66	23.66	61.75	25.97	**18.68**
	06	42.62	13.06	49.31	23.82	**12.28**
sqnc	01	10.56	9.38	20.60	46.25	**7.02**
	02	**3.15**	5.23	4.91	21.25	5.02
	03	2.45	**2.23**	3.52	3.29	5.07
	04	18.87	**10.60**	29.84	17.03	13.73
	05	71.83	24.52	65.29	25.19	**22.87**
	06	48.61	**13.79**	47.75	21.61	14.15

Table 5: Average time (in seconds).

		Cycle		Hybrid		Tree
family	sub	bfct	rdh	bfct	rdh	
ibm18	01	**0.29**	0.66	0.41	0.86	0.84
	02	**0.31**	0.66	0.46	0.92	0.84
	03	**0.29**	0.63	0.39	0.80	0.83
	04	2.30	2.37	3.13	3.62	**1.71**
	05	10.22	7.93	11.96	8.63	**4.82**
	06	7.25	4.85	8.36	7.75	**2.70**
lnc	01	**0.54**	0.98	0.84	2.03	1.20
	02	**0.51**	0.91	0.69	1.69	1.17
	03	**0.48**	0.82	0.57	0.98	1.18
	04	4.08	3.85	5.98	5.37	**2.68**
	05	16.85	10.64	14.00	10.37	**6.05**
	06	11.42	6.43	9.91	8.59	**3.37**
pnc	01	**1.23**	1.89	1.48	2.31	4.12
	02	**1.21**	1.99	1.59	2.78	3.84
	03	**1.20**	1.94	1.41	2.33	4.00
	04	8.96	**7.79**	11.18	9.58	9.03
	05	59.39	14.83	27.97	14.27	**10.35**
	06	28.46	11.69	20.68	13.67	**7.61**
rand5	01	3.62	5.25	4.06	6.10	**3.24**
	02	**0.97**	2.62	1.40	3.34	2.33
	03	**0.72**	1.76	0.92	2.11	2.29
	04	9.99	9.61	10.93	8.88	**5.00**
	05	21.90	14.86	20.66	13.36	**9.19**
	06	15.30	10.48	15.21	11.54	**5.56**
road	01	**0.37**	0.80	0.48	1.01	0.99
	02	**0.40**	0.82	0.58	1.16	1.02
	03	**0.36**	0.80	0.49	0.99	1.00
	04	**1.52**	1.60	2.01	2.05	1.90
	05	10.93	9.83	14.94	10.50	**5.19**
	06	10.60	6.70	11.26	10.30	**3.19**
sqnc	01	2.12	2.83	3.60	12.29	**1.59**
	02	**0.70**	1.72	0.89	5.62	1.21
	03	**0.52**	0.85	0.62	1.02	1.23
	04	4.37	3.87	6.29	5.99	**2.80**
	05	16.91	10.75	14.55	10.53	**6.07**
	06	11.67	6.32	10.50	8.40	**3.32**

We consider incremental reinitialization, data representation, and the benefits of the hybrid algorithm over "pure" binary search. To improve readability, instead of reporting the absolute number of scans per vertex, we report *relative* values, using as a basis the relevant benchmark algorithm (one of the four studied so far). Values greater than one favor the benchmark algorithms.

7.4.1 Incremental reinitialization. We begin with restart strategies. Our benchmark algorithms use *incremental restarts*, which preserve potentials and candidate vertices between calls to the SPF algorithm. We compare this with *full restarts*, which reset all potentials and mark all vertices as labeled in each call. Table 6 shows the average number of scans with full restarts relative to incremental restarts.

Incremental restarts are particularly beneficial on subfamilies 04 to 06, which are harder for all algorithms. Speedups of over an order of magnitude were observed. On subfamily 03, in contrast, incremental restarts have little effect. The benefits of incremental reinitialization are clearer for the Cycle algorithm, since Hybrid must use full restarts in roughly half of its SPF calls.

We also considered *partially incremental restarts*, which keep potentials but mark all vertices as labeled. Table 7 presents the number of scans obtained by this method, relative to fully incremental restarts (the benchmark method). The latter is still better, but never by a factor much greater than two. Simply preserving potentials is enough to achieve most of the speedup reported in Table 6.

Table 6: Average number of scans using full restarts (relative to incremental restarts).

ALGORITHM	FAMILY	01	02	03	04	05	06
CYCLE	ibm18	1.11	1.41	1.00	2.01	4.00	9.37
BFCT	lnc	4.56	1.40	1.00	2.91	5.14	9.47
	pnc	1.00	1.54	1.00	3.35	19.56	7.34
	rand5	2.09	1.66	1.00	2.00	4.32	9.10
	road	1.00	1.60	1.00	1.95	3.91	8.72
	sqnc	2.87	1.09	1.00	2.72	5.06	9.29
CYCLE	ibm18	1.10	1.27	1.00	2.41	7.83	6.84
RDH	lnc	2.94	1.75	1.00	3.70	10.57	6.88
	pnc	1.00	1.44	1.00	5.77	13.25	8.30
	rand5	3.20	2.13	1.00	4.34	9.80	8.31
	road	1.00	1.43	1.00	1.73	6.66	6.53
	sqnc	1.11	0.52	1.00	3.65	10.59	7.86
HYBRID	ibm18	1.09	1.26	1.00	1.88	2.87	3.26
BFCT	lnc	2.64	1.28	1.00	2.14	3.53	3.86
	pnc	1.00	1.40	1.00	2.58	4.09	3.50
	rand5	1.79	1.48	1.00	1.75	3.08	3.21
	road	1.00	1.35	1.00	1.69	2.76	2.82
	sqnc	2.03	1.11	1.00	2.08	3.40	3.71
HYBRID	ibm18	1.09	1.21	1.00	2.08	6.21	4.40
RDH	lnc	1.87	1.46	1.00	3.08	6.89	5.24
	pnc	1.00	1.34	1.00	4.82	5.77	5.38
	rand5	2.38	1.68	1.00	3.08	7.52	5.37
	road	1.00	1.29	1.00	1.58	5.84	4.25
	sqnc	0.44	0.34	1.00	2.77	7.02	5.10

Table 7: Average number of scans using partially incremental restarts (relative to incremental restarts).

ALGORITHM	FAMILY	01	02	03	04	05	06
CYCLE	ibm18	1.07	1.24	1.00	1.50	1.44	2.21
BFCT	lnc	2.66	1.22	1.00	1.70	1.42	2.19
	pnc	1.00	1.19	1.00	1.40	1.29	1.61
	rand5	1.34	1.30	1.00	1.41	1.41	2.05
	road	1.00	1.35	1.00	1.44	1.47	2.09
	sqnc	1.56	1.13	1.00	1.54	1.38	2.07
CYCLE	ibm18	1.07	1.18	1.00	1.22	1.72	1.55
RDH	lnc	1.69	1.31	1.00	1.31	1.80	1.53
	pnc	1.00	1.18	1.00	1.23	1.51	1.61
	rand5	1.31	1.28	1.00	1.25	1.69	1.61
	road	1.00	1.27	1.00	1.27	1.79	1.68
	sqnc	1.39	1.40	1.00	1.39	1.60	1.58
HYBRID	ibm18	1.04	1.13	1.00	1.30	1.25	1.39
BFCT	lnc	1.54	1.13	1.00	1.32	1.35	1.41
	pnc	1.00	1.12	1.00	1.19	1.24	1.28
	rand5	1.17	1.13	1.00	1.25	1.27	1.34
	road	1.00	1.17	1.00	1.27	1.26	1.35
	sqnc	1.25	1.08	1.00	1.31	1.23	1.46
HYBRID	ibm18	1.05	1.14	1.00	1.14	1.33	1.24
RDH	lnc	1.19	1.08	1.00	1.18	1.34	1.28
	pnc	1.00	1.12	1.00	1.28	1.34	1.29
	rand5	1.29	1.24	1.00	1.25	1.34	1.35
	road	1.00	1.15	1.00	1.18	1.35	1.22
	sqnc	1.06	1.00	1.00	1.22	1.35	1.29

7.4.2 Data representation. We considered three possible representations of potentials and λ: integers, floats, and rationals. The results reported so far use 64-bit integers, which have higher precision than 64-bit doubles. When precision is not an issue, both methods have almost identical operation counts and running times on our 64-bit test machine. On 32-bit machines, however, the running times can be lower with doubles due to better hardware support (the difference is a factor of roughly 2 on a Pentium 4). As for the third approach, Table 8 shows the average number of scans performed by CYCLE and HYBRID (both with BFCT) using rationals, relative to integers.

Table 8: Average number of scans per vertex using rationals (relative to integers). All algorithms use BFCT.

ALGORITHM	FAMILY	01	02	03	04	05	06
CYCLE	ibm18	1.10	1.34	1.00	0.85	0.85	0.93
	lnc	1.42	1.33	1.00	0.90	0.89	1.00
	pnc	1.00	1.32	1.00	0.83	0.88	0.93
	rand5	1.01	1.31	1.00	0.83	0.87	0.96
	road	1.00	1.41	1.00	1.02	0.92	0.89
	sqnc	0.87	1.09	1.00	0.93	0.89	1.01
HYBRID	ibm18	1.06	1.17	0.96	1.35	1.09	0.95
	lnc	1.74	1.18	0.97	1.08	1.28	1.09
	pnc	0.97	1.36	0.97	1.34	1.48	1.13
	rand5	1.44	1.34	0.96	1.04	1.15	1.02
	road	0.96	1.20	0.96	2.01	1.36	0.94
	sqnc	1.24	1.06	0.98	1.13	1.16	1.08

As explained in Section 4.3, rationals actually lead to a different algorithm, since changes in λ implicitly change all potentials. This explains why the numbers of scans sometimes differ substantially. Although in some cases the rational implementation is slightly better, it is often significantly worse. Given that this implementation is more complicated and incompatible with incremental restarts, integers should be preferred in practice.

7.4.3 Binary search. As mentioned in Section 3.3, we considered two techniques to speed up the convergence of the binary search algorithm (BINSEARCH): making the search biased and combining it with CYCLE. Our benchmark algorithm uses both: it is a biased hybrid search. We now consider the other three combinations: standard (unbiased) binary search, biased search, and unbiased hybrid search. Table 9 shows the average number of scans performed by each such method, with biased hybrid search as a benchmark. All methods use BFCT as the SPF algorithm.

HYBRID is superior to biased search, which in turn is clearly superior to standard binary search. The latter is particularly bad for the 03 subfamily, which can be solved by HYBRID in 5 scans or less. The two variants

Table 9: Average number of scans by variants of binary search, relative to biased hybrid, using BFCT.

ALGORITHM	FAMILY	01	02	03	04	05	06
BINSEARCH	ibm18	17.14	13.94	19.11	3.79	1.39	1.30
(unbiased)	lnc	6.09	12.62	20.01	2.96	1.53	1.37
	pnc	16.92	11.46	17.07	2.29	1.52	1.04
	rand5	3.85	8.99	18.43	2.61	1.39	1.22
	road	19.09	13.18	19.08	6.77	1.34	1.14
	sqnc	4.22	13.18	20.11	3.07	1.43	1.31
BINSEARCH	ibm18	4.90	4.26	5.34	1.73	1.09	0.85
(biased)	lnc	2.13	3.98	5.45	1.62	1.18	1.05
	pnc	4.76	3.47	4.78	1.38	1.03	0.96
	rand5	1.62	2.90	5.17	1.42	1.12	1.00
	road	5.34	3.92	5.34	2.54	1.09	0.73
	sqnc	1.63	4.15	5.44	1.65	1.10	1.02
HYBRID	ibm18	1.00	1.00	1.00	1.00	1.29	1.41
(unbiased)	lnc	1.09	0.99	1.00	1.07	1.57	1.59
	pnc	1.00	1.00	1.00	1.11	1.49	1.45
	rand5	1.20	1.03	1.00	1.22	1.41	1.44
	road	1.00	1.00	1.00	0.99	1.11	1.42
	sqnc	0.87	0.99	1.00	1.11	1.43	1.54

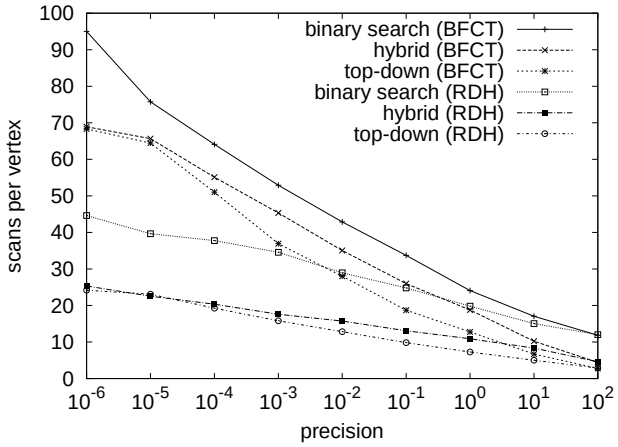

Figure 4: Scans per vertex on subfamily 05 of rand5 (with $n = 262\,144$) as a function of the precision (ϵ).

of HYBRID are roughly equivalent, with slight advantage to the benchmark (biased) one.

7.5 Precision. The experiments reported so far used very high precision ($\epsilon = 10^{-6}$). To test the dependence on ϵ, we varied ϵ from 10^{-6} to 10^2, and ran the four benchmark SP feasibility algorithms, as well as two versions on binary search (using BFCT and RDH), on subfamily 05 of rand5. Figure 4 shows the average number of scans per vertex as a function of ϵ. All algorithms are faster for larger values of ϵ. BINSEARCH is especially sensitive, but it is never better than HYBRID or CYCLE when using the same feasibility routine (BFCT or RDH).

7.6 Asymptotic Behavior. To assess the asymptotic behavior of the algorithms, we consider graphs with $n \approx 2^{20}$ vertices, four times bigger than the ones tested so far. Table 10 shows the number of scans per vertex on the larger graphs, relative to the smaller ones.

All algorithms have theoretical worst case at least $\Omega(n^2)$. If such growth were observed in practice, values in the table should be close to four, but all entries are significantly lower. All values are below two, and many are close to one, indicating near-linear behavior. The TREE algorithm is the most robust: its highest value is 1.06. This confirms what we have already observed in Section 7.3: these problem families are relatively easy.

7.7 Robustness. Our main experiments (Table 4) showed that our five benchmark algorithms usually perform far fewer than 100 scans per vertex. This does not mean that *any* reasonable algorithm works well.

For example, we found Dasdan's implementation of Howard's algorithm to be competitive on circuits and random graphs (confirming Dasdan's assessment), but much worse than CYCLE implemented with BFCT on harder instances (such as the 06 subfamily).

Similarly, many variants omitted from Table 4 often perform hundreds of scans per vertex on various instances. In particular, without incremental restarts, both versions of the CYCLE algorithm (using BFCT and RDH) require more than a thousand scans per vertex on subfamily 05 of pnc.

In fact, even some of the benchmark algorithms have superlinear behavior when run on the pathological instances described in Section 6, To confirm this, we ran them with $k = 100$ and $k = 400$. Table 11 shows the average number of scans per vertex with $k = 400$, relative to the corresponding values for $k = 100$. Values greater than one indicate superlinear dependence on n (quadratic for a factor of 4, and cubic for 16). For completeness, we consider all three reinitialization strategies for CYCLE: full (mode 0), partially incremental (1) and incremental (2).

8 Final Remarks

Out of many possibilities, our study shows that, on a wide class of problem families, two variants of the cycle-based algorithm and especially the tree-based algorithm of Young et al. are superior to other methods. Binary search is not competitive.

Note that while TREE has the best known theoretical time bound, not much is known about CYCLE. We are currently working on a better theoretical under-

Table 10: Average number of scans for graphs with $n \approx 2^{20}$, relative to graphs with $n \approx 2^{18}$.

ALGORITHM	FAMILY	01	02	03	04	05	06
CYCLE	lnc	1.00	1.00	1.00	1.09	1.33	1.45
BFCT	pnc	1.00	1.00	0.97	1.29	1.55	1.42
	rand5	1.04	1.09	1.00	1.12	1.22	1.44
	road	0.96	0.96	0.96	1.13	1.26	1.33
	sqnc	1.25	1.07	1.00	1.21	1.34	1.43
CYCLE	lnc	1.00	0.99	0.99	1.16	1.08	1.07
RDH	pnc	1.00	1.00	0.98	1.09	1.13	1.06
	rand5	0.95	1.02	1.00	1.10	1.06	1.15
	road	0.94	0.94	0.94	1.11	1.07	1.18
	sqnc	1.35	1.69	1.00	1.16	1.12	1.09
HYBRID	lnc	1.07	1.00	1.00	1.08	1.23	1.20
BFCT	pnc	1.00	1.00	0.98	1.10	1.22	1.18
	rand5	1.01	1.15	1.00	1.13	1.17	1.14
	road	0.97	1.03	0.97	1.19	1.11	1.10
	sqnc	1.26	1.10	1.00	1.26	1.22	1.13
HYBRID	lnc	1.15	1.07	1.00	1.10	1.07	1.01
RDH	pnc	1.00	1.01	0.99	1.08	1.07	1.03
	rand5	0.98	1.10	1.00	1.06	1.05	1.05
	road	0.96	1.02	0.96	1.27	1.06	0.93
	sqnc	1.33	1.53	1.00	1.10	1.04	1.09
TREE	lnc	1.00	1.00	0.99	1.00	1.01	1.05
	pnc	1.00	1.00	0.96	1.02	1.01	1.04
	rand5	0.99	1.00	1.00	1.03	1.01	1.06
	road	0.92	0.92	0.91	0.90	1.02	1.06
	sqnc	1.05	1.00	1.00	1.01	1.01	1.05

Table 11: Number of scans per vertex on pathological families with $k = 400$, relative to $k = 100$. Possible restart methods (RS) are full (0), partially incremental (1), and incremental (2).

ALGORITHM		RS	BAD1	BAD2	BAD3	BAD4	BAD5
CYCLE	BFCT	0	3.9	4.0	16.1	1.0	1.2
	BFCT	1	3.9	4.0	1.8	3.6	1.5
	BFCT	2	1.0	3.9	0.6	3.7	9.7
	RDH	0	1.0	1.0	4.0	1.0	1.2
	RDH	1	0.9	1.0	1.3	1.0	2.0
	RDH	2	1.0	1.0	0.4	1.0	1.7
HYBRID	BFCT	2	1.0	1.0	1.2	1.0	1.8
	RDH	2	0.3	1.0	0.6	1.0	2.3
TREE	—	—	1.0	1.0	1.0	3.9	2.1

standing of this algorithm.

Acknowledgements. We thank Ali Dasdan for sharing his code with us and for giving us access to the IBM instances. We also thank Stephan Held for the BONN instances.

References

[1] R. E. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[2] N. Chandrachoodan, S. S. Bhattaryya, and K. J. R. Liu. Adaptive Negative Cycle Detection in Dynamic Graphs. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages V–163–V–166, 2001.

[3] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck. Shortest Path Feasibility Algorithms: An Experimental Evaluation. In *Proceedings of the 10th International Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 118–132. SIAM, 2008.

[4] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Mathematical Programming*, 85:277–311, 1999.

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

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

[7] A. Dasdan, S. Irani, and R. K. Gupta. Efficient Algorithms for Optimum Cycle Mean and Optimum Cost to Time Ratio Problem. In *Proceedings of the 36th Design Automation Conference*, 1999.

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

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

[10] S. Held, B. Korte, J. Maßberg, M. Ringe, and J. Vygen. Clock scheduling and clocktree construction for high performance ASICs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 232–239, 2003.

[11] R. A. Howard. *Dynamic Programming and Markov Processes*. John Wiley, New York, 1960.

[12] R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Mathematics*, 23:309–311, 1978.

[13] R. M. Karp and J. B. Orlin. Parametric Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Mathematics*, 3:37–45, 1981.

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

[15] E. F. Moore. The Shortest Path Through a Maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

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

[17] N. Young, R.E. Tarjan, and J.B. Orlin. Faster Parametric Shortest Path and Minimum Balance Algorithms. *Networks*, 21:205–221, 1991.