Math. Program. 85: 277–311 (1999)

Digital Object Identifier (DOI) 10.1007/s10107990053a

Boris V. Cherkassky · Andrew V. Goldberg

Negative-cycle detection algorithms

Received June 14, 1996 / Revised version received June 22, 1998
Published online January 20, 1999

Abstract. We study the problem of finding a negative length cycle in a network. An algorithm for the negative cycle problem combines a shortest path algorithm and a cycle detection strategy. We survey cycle detection strategies, study various combinations of shortest path algorithms and cycle detection strategies and find the best combinations. One of our discoveries is that a cycle detection strategy of Tarjan greatly improves computational performance of a classical shortest path algorithm, making it competitive with the fastest known algorithms on a wide range of problems. As a part of our study, we develop problem families for testing negative cycle algorithms.

Key words. algorithms – graph theory – computational evaluation – shortest paths – negative-length cycles

1. Introduction

The *negative cycle problem* is to find a negative length cycle in a network or to prove that there are none (see *e.g.* [22]). The problem is closely related to the shortest path problem (see *e.g.* [1,10,23,25–27]) of finding shortest path distances in a network with no negative cycles. The negative cycle problem comes up both directly, for example in currency arbitrage, and as a subproblem in algorithms for other network problems, for example the minimum-cost flow problem [20].

The best theoretical time bound, $O(nm)$, for the shortest path problem is achieved by the Bellman–Ford–Moore algorithm [1,10,25]. Here n and m denote the number of vertices and arcs in the network, respectively. With the additional assumption that arc lengths are integers bounded below by $-N \leq -2$, the $O(\sqrt{n}m \log N)$ bound of Goldberg [14] improves the Bellman–Ford–Moore bound unless N is very large. Better expected time bounds hold over a wide class of input distributions; see *e.g.* [21]. The same bounds hold for the negative cycle problem.

All known algorithms for the negative cycle problem combine a shortest path algorithm and a cycle detection strategy. We study combinations of shortest path algorithms and cycle detection strategies to determine the best combination. The shortest path algorithms we study are based on the labeling method of Ford [9,10].

Most cycle detection strategies for the labeling method look for cycles in the graph of parent pointers maintained by the method, which correspond to negative cycles in

B.V. Cherkassky: Central Economics and Mathematics Institute, Krasikova St. 32, 117418, Moscow, Russia
e-mail: cher@cher.msk.su. This work was done while the author worked for NEC Research Institute, Inc.

A.V. Goldberg: NEC Research Institute, 4 Independence Way,Princeton, NJ 08540, USA.
Current address: InterTrust Technology Corp., 460 Oakmead Parkway, Sunnyvale, CA 94086, USA,
e-mail: goldberg@intertrust.com, URL: http://www.intertrust.com/star/goldberg/index.html

the input graph. These parent graph cycles, however, can appear and disappear. Some cycle detection strategies depend on the fact that after a finite number of steps of the labeling method, the parent pointer graph always has a cycle. Another cycle detection strategy is based on the fact that if the input graph has a negative cycle, the distance labels maintained by the labeling method (with no cycle detection) will get arbitrarily negative. The latter two cycle detection strategies are well-known and their correctness is easy to prove for integral lengths. However, the real-valued case is much harder and we were unable to find the proofs in the literature. We give correctness proofs for these cycle detection strategies in the real-valued case.

Other cycle detection strategies are based on levels of the parent pointer graph [16, 28] and on admissible graph search [15]. (See Section 4 for details.)

Most experimental studies of shortest path algorithms, such as [5, 7, 11, 12, 18, 24], were conducted on graphs with no negative cycles. The study of [16] investigates a limited number of algorithms on graphs with and without negative cycles. In this paper we survey cycle detection strategies and study the practical performance of algorithms for the negative cycle problem. We attempted to make our study as complete as possible, and this lead us to interesting results. In particular, our data shows that a cycle detection strategy of Tarjan [30] leads to improved algorithms for the shortest path problem. These algorithms are often competitive with the fastest previous codes and are worth considering for many practical situations. We introduce the notion of a *current distance label* and use it to explain good practical performance of algorithms that incorporate Tarjan's cycle detection strategy.

The previously known shortest path algorithms we study are the classical Bellman–Ford–Moore algorithm; the Goldberg–Radzik algorithm [15], which on shortest path problems performed very well in a previous study [2]; an incremental graph algorithm of Pallottino [26], which performs well on some classes of shortest path problems; an algorithm of Tarjan [30], which is a combination of the Bellman–Ford–Moore algorithm and a subtree-disassembly strategy for cycle detection; and a level-based algorithm that compared well with the Bellman–Ford–Moore algorithm in a previous study [16].

Our study leads us to a better understanding of computational behavior of shortest path algorithms and suggests new algorithm variations. We develop a version of the network simplex method [4] optimized specifically for the negative cycle problem. We note that a simple modification of Tarjan's algorithm gives the "ideal" version of the Bellman–Ford–Moore algorithm and study this version. We also study another variation of Tarjan's algorithm and an incremental graph algorithm that is similar to Pallottino's but uses Tarjan's algorithm in the inner loop. In the follow-up study, we compare the best previous shortest path algorithms with the most promising new algorithms.

Performance of algorithms for the negative cycle problem depends on the number and the size of the negative cycles. In general, problems with many small negative cycles are the simplest. We develop a collection of problem families for testing negative cycle algorithms. Our problem families combine several network types with several negative cycle structures.

Our shortest path codes and problem generators are publically available. This should facilitate further research in the area as well as the process of selecting algorithms for practical applications.

This paper consists of three parts: a survey of negative cycle detection strategies, a computational study of the cycle detection algorithms, and a study of computational improvements certain cycle detection strategies bring to some shortest path algorithms. Section 2 gives basic definitions and notation. Section 3 reviews the labeling method. Section 4 describes theoretical results on negative cycle detection in the labeling method context. Section 5 describes shortest path algorithms relevant to our study, and Section 6 discusses cycle detection strategies and their incorporation in the shortest path algorithms. Section 7 summarizes the negative cycle algorithms used in our study. We describe our experimental setup in Section 8. Section 9 describes a preliminary experiment that motivates our main experiment and filters out uncompetitive codes. Section 10 describes problem generators and families used in our study. Section 11 gives results of our main experiment. This experiment suggests that some of the new negative cycle algorithms may be good shortest path algorithms. This motivates a follow-up experiment, described in Section 12, that evaluates these as shortest path algorithms. We present concluding remarks in Section 13.

2. Definitions and notation

The input to the single-source shortest path problem is (G, s, ℓ), where $G = (V, E)$ is a directed graph, $\ell : E \to \mathbf{R}$ is a length function, and $s \in V$ is the source vertex. The goal is to find shortest paths from s to all vertices of G reachable from s if no negative length cycle in G is reachable from s. We refer to a negative length cycle as a *negative cycle*. We say that the problem is *feasible* if G does not have a negative length cycle reachable from s. The *negative cycle problem* is to determine if the problem is feasible, and to compute the distances if it is and a negative cycle if it is not. We denote $|V|$ by n, $|E|$ by m, and the biggest absolute value of an arc length by C.

A *distance labeling* is a function on vertices with values in $\mathbf{R} \cup \{\infty\}$. Given a distance labeling d, we define the *reduced cost function* $\ell_d : E \to \mathbf{R} \cup \{\infty\}$ by

$$\ell_d(v, w) = \ell(v, w) + d(v) - d(w).$$

We say that an arc a is *admissible* if $\ell_d(a) \leq 0$, and denote the set of admissible arcs by E_d. The *admissible graph* is defined by $G_d = (V, E_d)$. Note that if $d(v) < \infty$ and $d(w) = \infty$, the arc (v, w) is admissible. If $d(v) = d(w) = \infty$, we define $\ell_d(v, w) = \ell(v, w)$.

A *shortest 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. Given a tree and a vertex v in the tree, by the *depth* of v we mean the number of arcs on the path from the root to v.

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

Given a path Γ and a vertex v, we denote by $\Gamma \cdot v$ the path obtained by concatenating v to the end of A.

3. Labeling method

In this section we briefly outline the general *labeling method* [9,10] for solving the shortest path problem. (See *e.g.* [3,11,31] for more detail.) Most shortest path algorithms, and all those which we study in this paper, are based on the labeling method.

For every vertex v, the method maintains its distance label $d(v)$ and parent $p(v)$. Initially for every vertex v, $d(v) = \infty$, $p(v) =$ **null**. The method starts by setting $d(s) = 0$. At every step, the method selects an arc (u, v) such that $d(u) < \infty$ and $d(u) + \ell(u, v) < d(v)$ and sets $d(v) = d(u) + \ell(u, v)$, $p(v) = u$. (We call this the *labeling operation*.) If no such arcs exist, the algorithm terminates.

Lemma 1 (See e.g. [31]). *The labeling method terminates if and only if G contains no negative cycle. If the method terminates, then d gives correct distances and the parent pointers give a correct shortest path tree.*

If the method terminates, the parent pointers define a correct shortest path tree and, for any $v \in V$, $d(v)$ is the shortest path distance from s to v. In the next section we discuss how to modify the labeling method so that if G has negative cycles, the method finds such a cycle and terminates.

The *scanning method* is a variant of the labeling method based on the scan operation. The method maintains for each vertex v the *status* $S(v) \in$ {unreached, labeled, scanned}. Initially every vertex except s is unreached and s is labeled. The SCAN operation applies to a labeled vertex v. The operation is described in Figure 1. Note that if v is labeled, then $d(v) < \infty$ and $d(v) + \ell(v, w)$ is finite. Vertex status is updated as follows: a vertex becomes scanned while a scan operation is applied to it. A vertex whose distance label decreases becomes labeled.[1] After a SCAN operation, some unreached and scanned vertices may become labeled. The scanning method is correct because if there are no labeled vertices, then d gives the shortest path distances.

Given a scanning algorithm, we define *passes* inductively. Pass zero consists of the initial scanning of the source s. Pass i starts as soon as pass $i - 1$ ends, and ends as soon as the scan operation has been applied to all vertices which were labeled at the end of pass $i - 1$ and had exact distance labels at that time. Note that we allow vertices marked labeled during a pass to be scanned during this pass. We also allow vertices to be scanned several times during a pass.

```
procedure SCAN(v);
    for all (v, w) ∈ E do
        if d(v) + ℓ(v, w) < d(w) then
            d(w) ← d(v) + ℓ(v, w);
            S(w) ← labeled;
            p(w) ← v;
    S(v) ← scanned;
end.
```

Fig. 1. The SCAN operation

[1] As we shall see later, Tarjan's subtree disassembly strategy may declare a labeled or scanned vertex unreached.

This definition of a pass is more general than the one used in the context of the Bellman–Ford–Moore algorithm[2]. (see *e.g.* [31]). For this algorithm, a pass consists of scanning vertices that are labeled at the end of the previous pass. The general definition allows scans of other labeled vertices (as in the Goldberg–Radzik algorithm).

Under our definition of a pass, there is no polynomial time bound on a pass in general. An *efficient pass* is a pass such that each vertex is scanned at most once. Passes of the Bellman–Ford–Moore, the Goldberg–Radzik, and Tarjan's algorithms are efficient and take $O(m)$ time. The following lemma is the key to the analysis of these algorithms. Its proof is similar to the corresponding result for the Bellman–Ford–Moore algorithm (*e.g.* [31]).

Lemma 2. *If there is a shortest path from s to v containing k arcs, then after at most k passes $d(v)$ is exact. Thus in the absence of negative cycles, the labeling method terminates after at most $n - 1$ passes.*

4. Labeling method and negative cycles

In this section we study the labeling method in the presence of negative cycles. By Lemma 1, in this case the labeling method does not terminate. A *cycle detection strategy* is used to stop the method in this case. Most cycle detection strategies are based on the facts that cycles in the parent graph (defined below) correspond to negative cycles in the input graph and that if the input graph has a negative cycle then after a finite number of labeling operations the parent graph always has a cycle. Another cycle detection strategy is based on the following two facts. First, if the input graph has a negative cycle and the labeling method is applied with no cycle detection strategy, the distance label of some vertex will get arbitrarily negative. Second, if the distance label of a vertex v is smaller than the length of a shortest simple path from s to v, then the input graph has a negative cycle.

To discuss cycle detection strategies, we need the following definition. The *parent graph* G_p is the subgraph of G induced by the arcs $(p(v), v)$ for all $v : p(v) \neq \textbf{null}$. This graph has the following properties.

Lemma 3 (See e.g. [31]). *Arcs of G_p have nonpositive reduced costs. Any cycle in G_p is negative. If G_p is acyclic, then its arcs form a tree rooted at s.*

In the presence of negative cycles, it is relatively easy to show that after a finite number of labeling operations, G_p must contain a cycle. See *e.g.* [31]. However, a cycle in G_p can appear after a labeling operation and disappear after a later labeling operation: see Figure 2. Some of the cycle detection strategies do not check for cycles in G_p after every labeling operation. Correctness of these strategies is based on the following theorem.

Theorem 1. *If G contains a negative cycle reachable from s, then after a finite number of labeling operations G_p always has a cycle.*

[2] The algorithms mentioned here are described in Section 5

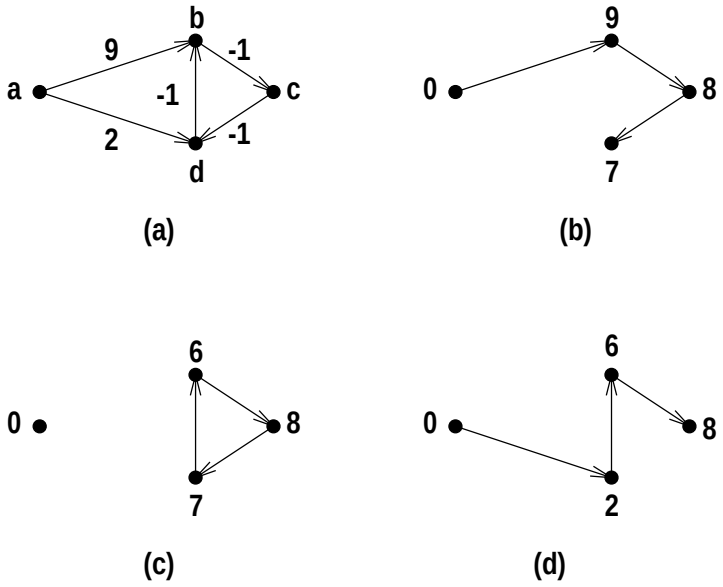


Fig. 2a-d. Disappearing cycle in G_p. (a) Input graph. (b) G_p and d after labeling operations applied to arcs (a, b), (b, c), (c, d). (c) Next the labeling operation is applied to (d, b), creating a cycle. (d) Next the labeling operation is applied to (a, d), destroying the cycle

Proof. Consider an execution of the labeling method. Let A be the set of vertices whose distance labels change finitely many times during the execution and let B be the remaining vertices. Because of the negative cycle, the execution does not terminate and B is not empty. A vertex $u \in A$ can become the parent of a vertex $v \in B$ only once after each change in $d(u)$. Thus for each $v \in B$, the sequence of $p(v)$ contains finitely many elements of A. Therefore after a finite number of labeling operations, for every $v \in B$ we have $p(v) \neq$ **null** and $p(v) \in B$.

Consider the subgraph of G_p induced by B. This subgraph has $|B|$ arcs and every vertex in it has in-degree one. Such a subgraph must have a cycle.

□

The "distance lower bound" cycle detection strategy stops a labeling algorithm and declares that there is a negative cycle as soon as $d(s) < 0$ or $d(v) < -(n-1)C$ for some $v \in V$. Correctness of this strategy is based on the following lemma.

Lemma 4. *Suppose for some vertex v, $d(v)$ is less than the length of a shortest simple path from s to v. Then G_p has a cycle. Since $d(v)$ is nonincreasing, G_p has a cycle at any later point of the execution.*

Proof. Note that the parent of a vertex has a finite distance label and all vertices with finite distance labels except s have parents. The source s has a parent if and only if $d(s) < 0$.

Suppose we start at v and follow the parent pointers. If we find a cycle of parent pointers in this process, we are done. The only way we can stop without finding a cycle is if

we reach s and $d(s) = 0$. In this case there is a simple s-to-v path Γ in G_p. By Lemma 3 and the fact that $d(s) = 0$, we have $d(v) \geq \ell(\Gamma)$. This contradicts the definition of v.

□

Corollary 1. *If $d(s) < 0$, then G_p has a cycle.*

The above lemma shows that the distance lower bound strategy is correct but does not assure termination of the labeling method with this cycle detection strategy. Termination is easy to prove for integral lengths, but for the real-valued case the proof is nontrivial. Next we prove the following theorem.

Theorem 2. *If G contains a negative cycle reachable from s, then, after a finite number of labeling operations, for some vertex u, $d(u)$ is less than the length of a shortest simple path from s to u.*

Lemma 4 and Theorem 2 imply Theorem 1, but the proof of the latter theorem is simpler.

The proof of Theorem 2 requires several definitions and auxiliary lemmas. We define the path $P(v)$ of a vertex v inductively. Initially all vertices have empty paths except for s, and $P(s) = s$. Applying the labeling operation to (u, v) replaces $P(v)$ by $P(u) \cdot v$. Note that the paths are not necessarily simple.

Consider a nonempty path $P(v) = v_1 \cdot \ldots \cdot v_t$. If $d(s) < 0$, then Theorem 2 holds, so for the rest of the proof we assume that $d(s) = 0$. Then $s = v_1$. Define $d'(s) = 0$. For $i > 0$, the vertex v_i on $P(v)$ was added to $P(v)$ by a labeling operation. Let $d'(v)$ be the distance label assigned to v by this operation. By the definition of $P(v)$, we have the following lemma.

Lemma 5. *For any $1 \leq i < t$, $d'(v_i) + \ell(v_i, v_{i+1}) = d'(v_{i+1})$.*

Corollary 2. *For any $v \in V$ and at any point during execution of the algorithm, $\ell(P(v)) = d(v)$.*

Since each time $P(v)$ changes $d(v)$ decreases, paths $P(v)$ do not repeat.

Recall that every path can be decomposed into a simple path and a collection of simple cycles.

Lemma 6. *For any path $P(v)$, the path can be decomposed into a simple path from s to v and a collection of simple cycles of negative length.*

Proof. It is sufficient to show that $P(v)$ cannot contain a cycle of nonnegative length. Consider a labeling operation applied to an arc (i, j) and suppose that this operation creates a new cycle Γ in one of the paths. Then Γ must include (i, j). Let $\Gamma(j, i)$ be the path from j to i obtained by deleting (i, j) from Γ. Just before the labeling operation, $d'(i) + \ell(i, j) < d'(j)$. But $d'(i) = d'(j) + \ell(\Gamma(j, i))$, and therefore $\ell(\Gamma) = \ell(\Gamma(j, i)) + \ell(i, j) < 0$.

□

Now we are ready to prove Theorem 2.

Proof of Theorem 2. Let ϵ be the absolute value of the length of the least negative simple cycle of G. Since G contains a negative cycle and the number of simple cycles is finite, ϵ is well-defined.

Since the paths $P(v)$ do not repeat, for some vertex u the number of arcs on $P(u)$ must be unbounded as the algorithm runs. Thus the number of simple cycles in the decomposition of $P(u)$ is also unbounded. Since a simple path has a length of at most $(n-1)C$ and a simple cycle in the decomposition has a length of at most $-\epsilon$, then when $P(u)$ contains more than $2(n-1)C/\epsilon$ simple cycles, we have $d(u) < -(n-1)C$.

□

Remark 1. The bound on the number of labeling operations implicit in Theorem 2 depends on the arc lengths as well as on the input network size. It is easy to modify the network of Figure 2 to show that the bound *must* depend on the arc lengths.

The following lemma complements Lemma 2.

Lemma 7. *If G contains a negative cycle reachable from s, then after the first labeling operation of pass n, G_p always contains a cycle.*

Proof. The proof of Lemma 2 shows that after $n-1$ passes distance labels are at least as small as the corresponding shortest simple path lengths. The first labeling operation after that reduces a distance label below the shortest simple path length. An argument similar to that of Lemma 4 completes the proof.

□

5. Labeling algorithms

Different strategies for selecting a labeled vertex to be scanned next lead to different algorithms. In this section we discuss some of these strategies and algorithms. We do not discuss some of the algorithms such as the Pape–Levit algorithm [23,27] and the threshold algorithm [12,13], which were not as robust as other algorithms in our previous study [2]. Since in this paper we are interested in networks with negative length arcs, we do not discuss Dijkstra's algorithm [8], which, both in theory and in practice, performs poorly on such networks. (Dijkstra's algorithm performs well on network with nonnegative arc lengths.)

First, however, we discuss techniques for improving efficiency of labeling algorithms. It is well-known that after a vertex with exact distance label is scanned, it is never scanned again. One would like to scan only such vertices, but efficient implementations of this idea are known only for the special cases of the problem, such as acyclic networks or networks with nonnegative arc lengths. However, there are efficient heuristics for preventing *some* scans of vertices with inexact distance labels.

If the parent graph G_p is a tree, we say that $d(v)$ is *current* if the distance from s to v in G_p is equal to $d(v)$. If $d(v)$ is not current, it must be inexact. This observation is the basis of several heuristic improvements of the Bellman–Ford–Moore algorithm discussed below.

Assume that G_p is a tree. One can show that during pass i, vertices at depth less than $i-1$ in G_p have exact distance labels and vertices at depth $i-1$ have current distance

labels. We say that a variant of the Bellman–Ford–Moore algorithm is *ideal* if at pass i it scans only the vertices at depth $i - 1$. For further discussion of this idea, see [16].

Desrochers [6] proposes to follow the path to the root before scanning a vertex, and proceeding with the scan only if the depth of the vertex is i. This, however, may be computationally expensive. Goldfarb et al. [16] propose efficient heuristics that identify some (but not all) vertices with depth greater than $i - 1$. See Section 5.2.

As we shall see in Section 6.7, there is an efficient way of scanning only vertices with current distance labels.

Goldberg and Radzik [15] propose to order labeled vertices before each pass so that a scanned vertex is more likely to have a current distance label. See Section 5.3.

5.1. The Bellman–Ford–Moore algorithm

The Bellman–Ford–Moore algorithm, due to Bellman [1], Ford [10], and Moore [25], maintains the set of labeled vertices in a first-in, first-out (FIFO) queue. The next vertex to be scanned is removed from the head of the queue; a vertex that becomes labeled is added to the tail of the queue if it is not already on the queue.

The performance of the Bellman–Ford–Moore algorithm is as follows, assuming that there are no negative cycles.

Theorem 3 (See e.g. [31]). (i) *Each pass takes $O(m)$ time.* (ii) *The number of passes is bounded by the depth of a shortest path tree.* (iii) *The algorithm runs in $O(nm)$ time in the worst case.*

5.2. The dynamic breadth-first search algorithm

Goldfarb et al. [16] suggested an improvement of the Bellman–Ford–Moore algorithm based on maintaining levels. Their algorithm maintains the levels t and the pass count i. After removing a vertex v from the queue, the algorithm scans v if $t(v) = i - 1$. Otherwise, the vertex is put back on the queue (to be re-processed at the next pass).

Note that if $t(v) = i - 1$, the depth of v in the current tree may be greater than i. Thus the algorithm may scan some vertices with depth i or more. Goldfarb et al. suggest the following heuristic that decreases the number of such vertices. Given a parameter ψ, their algorithm finds a vertex u that is ψ steps up from v in G_p and makes sure that $t(u) = i - 1 - \psi$ before scanning v. (The case when the depth of v is less than ψ is handled by defining dummy ancestors of s with the appropriate t values.)

5.3. The Goldberg–Radzik algorithm

Goldberg and Radzik [15] suggested another improvement of the Bellman–Ford–Moore algorithm that achieves the same worst-case time bound but usually outperforms the Bellman–Ford–Moore algorithm in practice. The algorithm maintains the set of labeled vertices in two sets, A and B. Each labeled vertex is in exactly one set. Initially $A = \emptyset$ and $B = \{s\}$. At the beginning of each *pass*, the algorithm uses the set B to compute

the set A of vertices to be scanned during the pass, and resets B to the empty set. A is a linearly ordered set. During the pass, elements are removed according to the ordering of A and scanned. The newly created labeled vertices are added to B. A pass ends when A becomes empty. The algorithm terminates when B is empty at the end of a pass.

The algorithm computes A from B as follows.

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

The algorithm achieves the same bound as the Bellman–Ford–Moore algorithm, again assuming no negative cycles.

Theorem 4 ([15]). *The Goldberg–Radzik algorithm runs in $O(nm)$ time.*

Now suppose G has cycles of zero or negative length. In this case G_d need not be acyclic. If, however, G_d has a negative length cycle, we can terminate the computation. If G_d has zero length cycles, we can contract such cycles and continue the computation. This can be easily done while maintaining the $O(nm)$ time bound. (See *e.g.* [14].)

Our implementation of the Goldberg–Radzik algorithm has one simplification. The implementation uses depth-first search to compute topological ordering of the admissible graph. Instead of contracting zero length cycles, we simply ignore the back arcs discovered during the depth-first search. The resulting topological order is in the admissible graph minus the ignored arcs. This change does not affect the algorithm's correctness or the running time bound given above.

Remark 2. When counting the number of scans done by the Goldberg–Radzik algorithm, we count both the shortest path SCAN operations and the processing of vertices done by the depth-first searches. We count the latter only if a depth-first search completed processing a vertex and backtracked from it.

5.4. Incremental-Graph algorithms

In this section we describe the incremental graph framework and Pallottino's algorithm [26].

An algorithm in the *restricted scan* framework maintains a set W of vertices and scans only labeled vertices in W. The set W is monotone: once a vertex is added to W, it remains in W. If there are labeled vertices but no labeled vertex is in W, some of the labeled vertices must be added to W. Vertices may also be added to W even if W already contains labeled vertices. Note that if the labeled vertices in W are processed in FIFO order, then a simple modification of the analysis of the Bellman–Ford–Moore algorithm shows that in $O(nm)$ time, either the algorithm terminates or W grows. This leads to an $O(n^2m)$ time bound.

Pallottino's algorithm defines W as the set of vertices which have been scanned at least once; when no labeled vertex is in W, a labeled vertex is added to W. More

precisely, the algorithm maintain the set of labeled vertices as two subsets, S_1 and S_2, the first containing labeled vertices which have been scanned at least once and the second containing those which have never been scanned ($S_1 \subseteq W$ and $S_2 \subseteq V - W$). The next vertex to be scanned is selected from S_1 unless S_1 is empty, in which case the vertex is selected from S_2 (*i.e.*, this vertex is added to W).

Pallottino's algorithm maintains S_1 and S_2 using FIFO queues, Q_1 and Q_2. The next vertex to be scanned is removed from the head of Q_1 if the queue is not empty and from the head of Q_2 otherwise. A vertex that becomes labeled is added to the tail of Q_1 if it has been scanned previously, or to the tail of Q_2 otherwise. The algorithm terminates when both queues are empty.

Theorem 5 ([26]). *Pallottino's algorithm runs in $O(n^2m)$ time in the worst case, assuming no negative cycles.*

5.5. Network simplex algorithm

In this section we describe a specialization of the network simplex method [4] to the shortest path problem. The resulting algorithm is a labeling algorithm, but not a scanning algorithm.

The main invariant maintained by the network simplex method is that the current tree arcs have zero reduced costs. To preserve the invariant, when the distance label of a vertex v decreases, the method decreases labels of vertices in the subtree rooted at v by the same amount. This is equivalent to traversing the subtree and applying labeling operations to the tree arcs. We implement this tree traversal procedure by maintaining an in-order list of tree nodes, as in many network simplex codes. See *e.g.* [19].

At every step, a generic network simplex algorithm for shortest paths finds an arc (v, w) with negative reduced cost, applies a labeling operation to it, and updates distance labels of vertices in w's subtree. A step of this algorithm is called a *pivot*, and (v, w) is called the *pivot arc*. Implementations of the simplex algorithm differ in how they find the next pivot arc.

A natural way to find the next pivot arc in the shortest path context is to use the idea of the scanning method: Maintain the set L of labeled vertices, select one such vertex, and scan it to find arcs with negative reduced costs. Note that if we pivot on an arc (v, w), then all vertices in w's subtree become labeled. This tends to create many labeled vertices, most with inexact distance labels. Scanning such vertices is wasteful because they will need to be rescanned after their distance label decreases.

The following heuristic cuts down the number of wasteful scans. Suppose v is an ancestor of w in the tree and v and w are labeled. Then if $d(w)$ is the correct distance from s, then so is $d(v)$. It is possible, however, that $d(v)$ is correct and $d(w)$ is not. Therefore scanning v before w is a good idea. To implement this idea, we maintain $L' \subseteq L$ such that L' contains all labeled vertices v such that no ancestor of v in the tree is labeled, and scan only vertices from L'.

The sets L and L' are maintained as follows. Initially $L = L' = \{s\}$. We pick a vertex u to scan and remove it from L and L'. When we pivot on (u, v), we add all vertices in the subtree rooted at v to L if they are not already in L and delete

descendants of v from L' if they are in L'. When we are finished scanning u, we add all children of u to L'. To see that the resulting algorithm is correct, note that L is exactly the set of all descendants of vertices in L'. Note also that we do not need to maintain L explicitly.

Our implementation of the network simplex algorithm maintains the set L' as a FIFO queue. When deleting an element of the queue, we mark it as deleted instead of physically removing it. When adding an element to the queue, we mark it as undeleted if it is already on the queue, and add it to the queue otherwise. To find the next vertex to scan, we remove vertices from the queue until we get an undeleted vertex.

We call the resulting algorithm *optimized network simplex*. Note that the optimization is heuristic; it does not improve the worst-case time bound but improves typical running times. Next we analyze this algorithm.

Using an analysis similar to that for the Bellman–Ford–Moore algorithm, one can show that the number of vertex scans is $O(n^2)$ and the number of pivots is $O(nm)$. Since each pivot takes $O(n)$ time we have the following result.

Theorem 6. *The optimized network simplex algorithm runs in $O(n^2m)$ time.*

Remark 3. For the optimized network simplex algorithm, the number of vertex scans is equal to the number of pivots. However, the algorithm (implicitly) applies labeling operations to tree arcs when updating subtree vertex labels.

6. Cycle detection strategies

In this section we discuss cycle detection strategies. Desirable features of these strategies are low amortized cost and immediate cycle detection. The latter means that a cycle in G_p is detected the first time G_p contains a cycle.

6.1. Time out

Every labeling algorithm terminates after a certain number of labeling operations in the absence of negative cycles. If this number is exceeded, we can stop and declare that the network has a negative cycle.

A major disadvantage of this method is that if there is a negative cycle, the number of labeling operations used by the method is equal to the worst-case bound. This method is uncompetitive and we did not implement it.

6.2. Distance lower bound

This method is based on Theorem 2. If distance label of a vertex falls below $-(n-1)C$, then G_p must contain a cycle, which can be found in $O(n)$ time. The drawback of this method is that the cycle is usually discovered much later than it first appears. The method is uncompetitive and we did not implement it.

6.3. *Walk to the root*

Suppose the labeling operation applies to an arc (u, v) and G_p is acyclic. Then G_p is a tree, and this operation will create a cycle in G_p if and only if v is an ancestor of u in the current tree. Before applying the labeling operation, we follow the parent pointers from u until we reach v or s. If we stop at v, we have found a negative cycle; otherwise, the labeling operation does not create a cycle.

This method gives immediate cycle detection and can be easily combined with any labeling algorithm. However, since paths to the root can be long, the cost of a labeling operation becomes $O(n)$ instead of $O(1)$. On certain kinds of graphs, the average tree path length is long, and this method is slow, as we will demonstrate below.

6.4. *Amortized search*

Another popular cycle detection method is to use amortization to pay the cost of checking G_p for cycles. Since the cost of such a search is $O(n)$, we can perform the search every time the underlying shortest path algorithm performs $\Omega(n)$ work without increasing the running time by more than a constant factor if there are no negative cycles. Theorem 1 implies that a labeling algorithm using this strategy terminates.

This method allows one to amortize the work of cycle detection and can be easily used with any labeling algorithm. However, the method does not discover negative cycles immediately. Furthermore, since cycles in G_p can disappear, we are not guaranteed to find a cycle at the first search after the first cycle in G_p appears. In fact, the cycle can be found much later.

6.5. *Admissible graph search*

This method, due to Goldberg [14], is based on the fact that the arcs in G_p are admissible. Therefore if G_p contains a cycle, the admissible graph G_d contains a negative cycle. Since all arcs in G_d have nonpositive reduced costs, a negative cycle in the graph can be found in $O(n + m)$ time using depth-first search. Since G_d may contain a negative cycle even if G_p does not, it is possible that this method finds a negative cycle before the first cycle in G_p appears.

One can use an admissible graph search instead of a search of G_p in the amortized search framework. Searching G_d, however, is more expensive than searching G_p, and the searches need to be less frequent. With this method, cycle detection is not immediate.

Admissible graph search is a natural cycle detection strategy for the Goldberg–Radzik algorithm, which performs a depth-first search of G_d at each iteration. This allows cycle detection at essentially no additional cost. We used the admissible graph search strategy only with the Goldberg–Radzik algorithm.

6.6. *Subtree traversal*

The idea behind this strategy is similar to the idea behind the walk to the root strategy. Suppose the labeling operation applies to an arc (u, v) and G_p is acyclic. Then G_p is

a tree, and this operation will create a cycle in G_p if and only if u is an ancestor of v in the current tree. We can check if this is the case by traversing the subtree rooted at v.

In general, subtree traversal needs to be applied after every labeling operation and increases the cost of a labeling operation to $O(n)$. (A good way to implement subtree traversal is using standard techniques from the network simplex method for minimum-cost flows; see *e.g.* [19].) With this strategy, cycle detection is immediate.

This strategy fits naturally with the network simplex method. During a pivot on (u, v), we already traverse the subtree rooted at v. Although we apply labeling operations to the tree arcs as we traverse the subtree, these operations do not change the tree and cannot create cycles in G_p. The subtree traversal strategy allows the method to detect cycles at essentially no extra cost.

We use the subtree traversal strategy only with the network simplex method.

6.7. Subtree disassembly

This method, due to Tarjan [30], is a variation of the subtree traversal strategy that allows one to amortize the subtree traversal work over the work of building the subtree. The method is a variation of the scanning method where some unreached vertices may have finite labels but **null** parents. Distance labels of such vertices, however, are inexact. One can easily show that the method remains correct in this case.

When the labeling operation is applied to an arc (u, v), the subtree rooted at v is traversed to find if it contains u (in which case there is a negative cycle). If u is not in the subtree, all vertices of the subtree except v are removed from the current tree and marked as unreached. The SCAN operation does not apply to these vertices until they become labeled.

The work of subtree disassembly is amortized over the work to build the subtree, and cycle detection is immediate. Because this strategy changes the status of some labeled vertices to unreached, it changes the way the underlying scanning algorithm works. However, since the vertices whose status changes have inexact distance labels, this tends to speed the algorithm up.

A combination of the FIFO selection rule and subtree disassembly yields Tarjan's algorithm [30] for the negative cycle problem. Like the Bellman-Ford-Moore algorithm, Tarjan's algorithm maintains a queue of labeled vertices and adds newly labeled vertices at the tail of the queue if they are not already on it. Initially the queue contains only s. At each step, the algorithm removes the head vertex v from the queue. If v is still labeled, the algorithm scans v and, for each vertex w whose distance label improves during the scan, the algorithm disassembles the subtrees rooted at w. Note that some vertices may become unreached as a side-effect of the subtree disassembly. The algorithm has an interesting property (recall the definition of the current distance label from Section 5):

Lemma 8. *Tarjan's algorithm scans only vertices with current distance labels.*

The proof of the lemma is a straight-forward induction on the number of scan operations.

A variation of subtree disassembly is subtree disassembly with update. This strategy can be viewed as the network simplex method with subtree disassembly strategy. As the subtree rooted at v is traversed and disassembled, the distance labels of proper

descendants of v are decreased by the same amount as $d(v)$, and the descendants become unreached. After a scan of a vertex u is complete, all vertices which were u's children immediately before the scan become labeled. A combination of the FIFO selection rule and this cycle detection strategy yields an algorithm with performance that is close to that of Tarjan's algorithm.

6.8. Level-based strategy

The *level-based* cycle detection strategies (see e.g. [16, 28]) are based on the following modification of the labeling method. We maintain a level, $t(v)$, at every vertex v. Initially $t(v) = \infty$ for all $v \neq s$ and $t(s) = 0$. Every time we set $p(v) = u$ during a scan of (u, v), we set $t(v) = t(u) + 1$. One can show that if $d(v)$ is exact, then $t(v)$ is equal to the number of arcs on the path from s to v in G_p. Therefore in the absence of negative cycles, during pass i, for each $0 \leq j < i$ there must be a vertex v with $t(v) = j$.

An algorithm with level-based cycle detection maintains levels t and an array C of counts: $C[j]$ contains the number of vertices v with $t(v) = j$. When $t(v)$ changes, the counts are updated. If during pass i $C[j]$ becomes zero for $0 \leq j < i$, the algorithm terminates and declares that the network contains a negative cycle.

7. Algorithms studied

Algorithm/Strategy	Bellman–Ford–Moore	Goldfarb–Hao–Kai	Goldberg–Radzik	Pallottino's	Network Simplex
Walk to the root	BFCF $O(n^2m)$				
Amortized search	BFCS $O(nm)$				
Subtree traversal					SIMP $O(n^2m)$
Subtree disassembly	BFCT, BFCM $O(nm)$			PALT $O(n^2m)$	
Subtree disassembly with update	BFCTN $O(nm)$				
Level-based		GHK3 $O(nm)$			
Admissible graph search			GORC $O(nm)$		

Fig. 3. Summary of negative cycle algorithms

Figure 3 gives a summary of the negative cycle algorithms used in our study. The table includes running time bounds, which follow from the results of Sections 3 – 6. Recall that a negative cycle algorithm is a combination of a shortest path algorithm and a cycle detection strategy, and that we did not implement time-out and distance lower bound strategies.

Three algorithms have natural cycle detection strategies associated with them: the optimized network simplex algorithm has subtree traversal, the Goldfarb et al. algorithm has levels, and the Goldberg–Radzik algorithm has admissible graph search. We implemented these algorithms only with their natural cycle detection mechanisms. For the second algorithm, we set $\psi = 3$ (this appears to be a reasonable choice in view of the data of [16]). The resulting codes are SIMP, GHK3, and GORC respectively.

Consider the Bellman–Ford–Moore algorithm. We implemented it with walk to the root, amortized search, subtree disassembly, and subtree disassembly with update strategies. Names of our Bellman–Ford–Moore cycle detection codes start with BFC. The last letters identify cycle detection strategies. The codes are BFCF (*follow* path to the root), BFCS (amortized *search*), BFCT (*Tarjan's* algorithm), and BFCTN (*Tarjan's natural* variant), respectively.

A simple variation of Tarjan's algorithm implements the ideal variation of the Bellman–Ford–Moore algorithm. This variation differs from Tarjan's algorithm only in one place. After applying a labeling operation to (u, v), Tarjan's algorithm adds v to the tail of the queue if v is not in the queue. The modified algorithm adds v to the tail of the queue if v is not in the queue and moves v to the tail of the queue if v is in the queue. Our code BFCM (*minimum* tree depth) implements the modified algorithm.

Consider an execution of the modified algorithm. Suppose no negative cycles have been found so far, so G_p is a tree. Induction on k shows if a vertex is scanned at pass k, then the depth in G_p of the vertex at the time of the scan is $k-1$. An equivalent statement is that for each scan, the algorithm selects a labeled vertex with the minimum depth in G_p.

Since Tarjan's algorithm gives improved performance, it is natural to use the subtree disassembly strategy in Pallottino's algorithm. This is what our PALT code does.

8. Experimental setup

Our experiments were conducted on a 133MHZ Pentium machine with 128MB memory and 256K cache running LINUX 1.2.8. Our codes are written in C and compiled with the LINUX `gcc` compiler using the `O4` optimization option.

Our implementations use the adjacency list representation of the input graph, similar to that of [11]. We attempted to make our implementations of different algorithms uniform to make the running time comparisons more meaningful. We also tried to make the implementations efficient.

The running times we report are user CPU times in seconds, averaged over several instances generated with the same parameters except for a pseudorandom generator seed. Each data point consists of the average running time (in bold), standard deviation, and the average number of scans per vertex. The number of scans per vertex is a machine-independent measure of algorithm performance which is very useful. For example, we use it to compare the overhead of vertex selection and cycle detection and to determine effectiveness of heuristics aimed at reducing the number of scan operations. Except for two families, we average over five instances. For the Rand-5 and the SQNC02 families (described below), we average over ten instances because of higher standard deviations.

We put a 30 minute limit of CPU running time for each problem instance. Note that the clock precision is 1/60 of a second.

When scoring code performance on a problem family, we use the following scale: good (◯), fair (⊙), poor (⊗), and bad (●). We assign performance based on the running times for the biggest instances of the problem family. (The only exception is the Rand-5 family, where the problem size is constant and the range of the arc lengths varies. For that family, we use the instances with the highest relative performance difference for scoring.) We normalize the times by that of the fastest code and use a factor of four as the threshold between adjacent scores. If the fastest code runs in x seconds, a code running in $2x$ seconds is rated good, in $7x$ seconds – fair, in $25x$ seconds – poor, and any code running in $64x$ seconds or more is rated bad. Our choice of the threshold makes it unlikely that a code not rated good in our experiment would be the fastest under a different compiler and machine architecture combination. We provide detailed data in addition to the summary scores, so, if desired, readers can interpret the data according to their own system.

9. Preliminary experiment

Before describing our main experiments, we give preliminary data to demonstrate the issues involved. We also cut down the number of codes evaluated in the main experiment by eliminating uncompetitive and similar codes.

The preliminary experiment compares codes BFCF, BFCS, BFCT, BFCM, BFCTN and GHK3 on square grids generated as in the square grid experiment described in Section 10. Suppose we have an $X \cdot X$ square grid. The five problem families of this experiment differ by the number of negative cycles in the graph and the cardinality (number of arcs) of these cycles. The families have no negative cycles, one small negative cycle (with three arcs), X small negative cycles, 16 moderately long (cardinality X) negative cycles, and one Hamiltonian negative cycle. The data is given in Figures 4–8. The number and cardinality of negative cycles greatly affect algorithm performance. For example, problems with many small negative cycles are easy.

Figure 4 gives data in the absence of negative cycles. The data shows how much different heuristics reduce the number of scans. Codes BFCF and BFCS have the same number of scans as the Bellman–Ford–Moore algorithm without cycle detection would

X*Y	BFCF	BFCS	GHK3	BFCT	BFCM	BFCTN
64	**3.94**	**0.70**	**0.22**	**0.03**	**0.03**	**0.03**
64	0.67	0.06	0.03	0.00	0.01	0.00
	28.85	28.85	17.48	2.99	2.99	2.97
128	**111.45**	**5.21**	**1.99**	**0.13**	**0.16**	**0.18**
128	15.93	0.52	0.14	0.01	0.02	0.00
	49.91	49.91	30.73	2.96	2.98	2.95
256		**46.61**	**18.22**	**0.71**	**0.75**	**0.91**
256		3.96	1.46	0.06	0.06	0.06
		98.01	58.78	2.93	2.93	2.92

Fig. 4. Preliminary experiment. No cycles

X*Y	BFCF	BFCS	GHK3	BFCT	BFCM	BFCTN
64	**0.07**	**0.06**	**0.24**	**0.01**	**0.01**	**0.01**
64	0.08	0.05	0.05	0.01	0.01	0.01
	2.41	2.52	15.66	0.76	0.73	0.76
128	**1.50**	**0.50**	**1.64**	**0.04**	**0.05**	**0.05**
128	1.41	0.37	0.24	0.03	0.02	0.03
	4.97	5.06	24.97	0.76	0.87	0.76
256	**44.38**	**2.99**	**16.15**	**0.14**	**0.16**	**0.16**
256	62.98	3.50	2.47	0.12	0.14	0.15
	6.30	6.44	48.16	0.52	0.52	0.52

Fig. 5. Preliminary experiment. One small cycle

X*Y	BFCF	BFCS	GHK3	BFCT	BFCM	BFCTN
64	**0.00**	**0.00**	**0.12**	**0.00**	**0.00**	**0.00**
64	0.00	0.01	0.03	0.00	0.01	0.00
	0.07	0.20	4.61	0.04	0.04	0.04
128	**0.01**	**0.02**	**0.98**	**0.01**	**0.02**	**0.01**
128	0.01	0.01	0.23	0.01	0.01	0.01
	0.02	0.20	8.73	0.02	0.01	0.02
256	**0.02**	**0.07**	**7.90**	**0.03**	**0.03**	**0.02**
256	0.01	0.00	1.31	0.00	0.00	0.00
	0.01	0.20	15.97	0.01	0.01	0.01

Fig. 6. Preliminary experiment. X small cycles

X*Y	BFCF	BFCS	GHK3	BFCT	BFCM	BFCTN
64	**0.23**	**0.25**	**0.09**	**0.05**	**0.05**	**0.06**
64	0.02	0.01	0.01	0.01	0.00	0.01
	8.84	8.95	5.41	3.40	3.45	3.24
128	**3.25**	**1.60**	**0.65**	**0.33**	**0.35**	**0.37**
128	0.30	0.12	0.05	0.04	0.02	0.02
	12.22	12.32	7.97	4.12	4.15	3.93
256	**48.76**	**8.04**	**3.62**	**1.43**	**1.63**	**1.71**
256	3.54	0.71	0.16	0.08	0.12	0.07
	14.81	14.92	10.00	4.54	4.55	4.35

Fig. 7. Preliminary experiment. 16 medium cycles

X*Y	BFCF	BFCS	GHK3	BFCT	BFCM	BFCTN
64	**6.66**	**0.67**	**0.29**	**0.15**	**0.17**	**0.19**
64	1.01	0.04	0.02	0.01	0.00	0.01
	17.54	17.67	14.01	10.15	10.14	9.43
128	**119.62**	**3.84**	**1.92**	**1.15**	**1.16**	**1.29**
128	10.78	0.15	0.07	0.11	0.09	0.04
	20.23	20.36	16.31	12.03	12.06	11.19
256		**21.34**	**10.95**	**6.03**	**6.39**	**6.90**
256		0.78	0.35	0.79	0.64	0.24
		23.48	19.12	14.17	14.19	13.14

Fig. 8. Preliminary experiment. One Hamiltonian cycle

have in this case. The heuristics used in GHK3 reduce the number of scans by almost a factor of two. Subtree disassembly drastically reduces the number of scans – the number of scans per vertex is small and, on this problem family, does not grow with problem size.

The data also shows relative overheads of cycle detection strategies. On the square grid problems, walk to the root is asymptotically more expensive than amortized search and BFCF is much slower than BFCS. The level-based strategy has a slightly lower overhead than amortized search. Combined with the smaller number of scans, this explains why GHK3 is faster than BFCS. Subtree disassembly has amortized overhead and drastically reduces the number of scan operations. The three codes based on this strategy perform similarly to each other, and much better than the other codes used in this experiment.

Next we compare the cycle detection strategies in the presence of negative cycles. The family with many small cycles (Figure 6) shows the benefits of immediate cycle detection. All algorithms with this property, even BFCF, are extremely fast on this family. The amortized search strategy finds a cycle somewhat later and the level-based strategy – much later, and the corresponding codes are slower.

In these experiments, as in all other experiments, the subtree disassembly codes BFCT, BFCM, and BFCTN outperform the other codes. Comparing BFCT, BFTM, and BFCTN codes, we observe that the performance of these codes is very close.

Although we do not present the data in this paper, for all problem families in our study BFCF, BFCS, and GHK3 never perform significantly better than BFCT, and often perform considerably worse. To avoid presenting uninteresting data, in the main experiment we do not give the data for the former codes. We also do not give the data for BFCM and BFCTN, whose performance is very similar to BFCT.

10. Problem generators and families

In the main experiment we use eleven problem families with four underlying graph types produced by two generators. These problem families were selected by exploring many more families (for example, long and wide grids) and selecting ones which are natural or give insight in the algorithm performance and avoid duplication. Figure 9 summarizes these problem families.

The first generator we use is SPRAND [2]. To produce a problem with n vertices and m arcs, this generator builds a Hamiltonian cycle on the vertices and then adds $m - n$ arcs at random. One of the vertices is designated as a source. In the experiments of this paper, the lengths of all arcs, including the cycle arcs, are selected uniformly at random from the interval $[L, U]$.

The Rand-5 family is generated using the SPRAND generator with a fixed network size: $n = 200,000$ and $m = 1,000,000$. The maximum arc length U is fixed at $32,000$, and the minimum arc length L varies from 0 to $-64,000$.

The second generator we use is TOR, derived from the SPGRID generator of [2]. We use this generator to produce two types of skeleton networks: grid networks and layered networks. The skeleton networks have no negative cycles.

Generator	Class name	Brief description	# cycles	|cycle|
SPRAND	Rand-5	random graphs of degree 5	lengths-dependent	
TOR	SQNC01	square grids, $n = X \cdot X$	0	
	SQNC02		1	3
	SQNC03		X	3
	SQNC04		16	X
	SQNC05		1	n
TOR	PNC01	layered graphs, $n = X \cdot 32$	0	
	PNC02		1	3
	PNC03		$X/4$	3
	PNC04		8	$X/4$
	PNC05		1	n

Fig. 9. Summary of problem classes. Here # cycles is the number of negative cycles and |cycle| is the cardinality of a negative cycle

Grid networks are grids embedded in a torus. Vertices of these networks correspond to points on the plane with integer coordinates $[x, y]$, $0 \le x < X$, $0 \le y < Y$. These points are connected "forward" by *layer* arcs of the form $([x, y], [x + 1 \bmod X, y])$, $0 \le x < X$, $0 \le y < Y$, and "upward" by *inter-layer* arcs of the form $([x, y], [x, y + 1 \bmod Y])$. In addition there is a source connected to all vertices with $x = 0$. Layer arc lengths are chosen uniformly at random from the interval $[1, 000, 10, 000]$. Inter-layer arc lengths are chosen uniformly at random from the interval $[1, 100]$. Arcs from the source are treated as inter-layer arcs. Skeletons of SQNC** (*square with negative cycles*) problems are square grids with $X = Y$.

Layered networks consist of layers $0, \ldots, X - 1$. Each layer is a simple cycle plus a collection of arcs connecting randomly selected pairs of vertices on the cycle. The lengths of the arcs inside a layer are chosen uniformly at random from the interval $[1, 100]$. There are arcs from one layer to the next one, and, in addition, there are arcs from a layer to "forward" layers. Consider an inter-layer arc (u, v) which goes x layers forward. The length of this arc is selected uniformly at random from the interval $[1, 10, 000]$ and multiplied by x^2. In addition there is a source connected to all vertices with $x = 0$. These networks are similar to the Grid-PHard networks of [2], except inter-layer arcs "wrap around" modulo X. Skeletons of PNC** (*P-Hard with negative cycles*) problems are layered networks with each layer containing 32 vertices and $X = n/32$.

Arcs forming vertex-disjoint negative cycles are added after the skeleton network has been generated. All arcs on these cycles have length zero except for one arc, which has a length of -1. As we have seen in the preliminary experiment, the number and the cardinality of negative cycles greatly affects the algorithm performance. Each TOR family has a certain type of negative cycles. Families with names ending in "01" have no negative cycles. Families with names ending in "02" have one small negative cycle. Families with names ending in "03" have many small negative cycles. Families with names ending in "04" have a few medium negative cycles. Families with names ending in "05" have one Hamiltonian negative cycle. See Figure 9 for details.

After adding the cycles, we apply a potential transformation to "hide" them. For each vertex, we select a potential uniformly at random from the same interval as the inter-layer arc lengths. Then we add the potential to the lengths of the incoming arcs and subtract the potential from the length of the outgoing arcs.

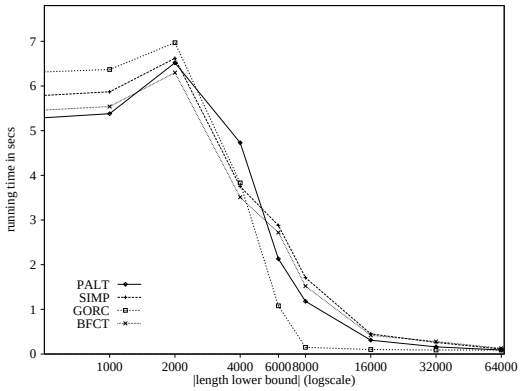
11. Experimental results

In this section we describe results of our main experiment. Fig. 10 summarizes these results and Figs. 11–21 give detailed data. The discussion in this section should be taken in the context of our experiment. Our use of the common sense and the preliminary experiment to filter out clearly uncompetitive algorithms explains why the remaining algorithm performance is good on many problem classes.

GORC has the highest scores. However, the only problem family it outscores BFCT on is Rand-5, and the score difference is due to the difference in performance for a small range of parameter values. On other problem classes, BFCT's performance is often

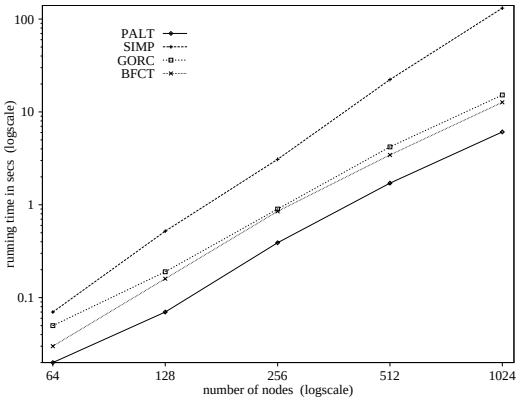
	BFCT	GORC	PALT	SIMP
Rand-5	⊙	○	⊙	⊙
SQNC01	○	○	○	⊗
SQNC02	○	○	○	⊙
SQNC03	○	○	○	○
SQNC04	○	○	○	○
SQNC05	○	○	○	○
PNC01	○	○	⊙	⊙
PNC02	○	○	⊙	⊙
PNC03	○	○	○	○
PNC04	○	○	○	○
PNC05	○	○	○	○

Fig. 10. Summary of algorithm performance. ○ means good, ⊙ means fair, and ⊗ means poor



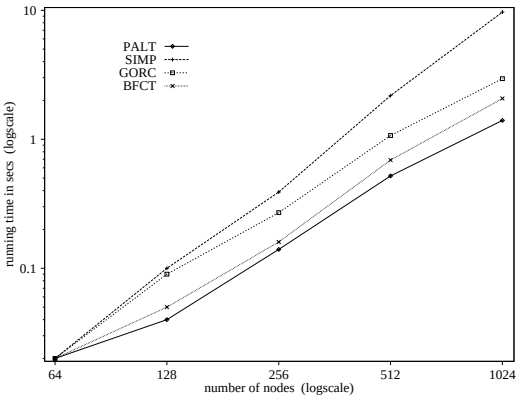
min/max	BFCT	GORC	PALT	SIMP
0	**4.76**	**5.83**	**4.47**	**5.07**
32000	0.17	0.19	0.21	0.17
	2.13	3.62	2.38	2.06
-1000	**5.54**	**6.37**	**5.38**	**5.87**
32000	0.45	0.26	0.61	0.46
	2.47	3.99	2.86	2.38
-2000	**6.30**	**6.97**	**6.52**	**6.62**
32000	1.92	2.36	1.95	2.02
	2.81	4.42	3.47	2.68
-4000	**3.51**	**3.83**	**4.73**	**3.75**
32000	1.85	2.36	2.65	2.03
	1.34	2.45	2.38	1.24
-6000	**2.72**	**1.08**	**2.13**	**2.88**
32000	1.40	0.64	1.06	1.47
	0.96	0.65	1.04	0.85
-8000	**1.52**	**0.15**	**1.18**	**1.71**
32000	0.96	0.06	0.83	1.15
	0.50	0.04	0.55	0.47
-16000	**0.42**	**0.10**	**0.31**	**0.45**
32000	0.23	0.01	0.16	0.26
	0.13	0.00	0.12	0.13
-32000	**0.28**	**0.09**	**0.16**	**0.26**
32000	0.24	0.01	0.08	0.21
	0.07	0.00	0.05	0.06
-64000	**0.13**	**0.09**	**0.09**	**0.11**
32000	0.08	0.01	0.03	0.07
	0.02	0.00	0.02	0.02

Fig. 11. Rand-5 family data



X*Y	BFCT	GORC	PALT	SIMP
64	**0.03**	**0.05**	**0.02**	**0.07**
64	0.00	0.01	0.01	0.00
	2.99	7.10	2.01	2.90
128	**0.16**	**0.19**	**0.07**	**0.52**
128	0.01	0.01	0.00	0.04
	2.96	7.01	1.91	2.84
256	**0.85**	**0.90**	**0.39**	**3.09**
256	0.03	0.01	0.01	0.14
	2.93	7.31	1.94	2.80
512	**3.44**	**4.20**	**1.71**	**22.30**
512	0.18	0.04	0.02	1.46
	2.80	7.48	1.95	2.66
1024	**12.69**	**15.22**	**6.09**	**131.29**
1024	0.68	0.22	0.06	7.71
	2.78	7.56	1.96	2.63

Fig. 12. SQNC01 family data



X*Y	BFCT	GORC	PALT	SIMP
64	**0.01**	**0.02**	**0.00**	**0.02**
64	0.01	0.01	0.01	0.02
	0.76	2.62	0.88	0.74
128	**0.05**	**0.09**	**0.04**	**0.10**
128	0.03	0.04	0.02	0.06
	0.76	2.76	0.91	0.74
256	**0.16**	**0.27**	**0.14**	**0.39**
256	0.14	0.23	0.11	0.41
	0.52	1.99	0.61	0.50
512	**0.69**	**1.07**	**0.52**	**2.18**
512	0.51	0.84	0.36	2.29
	0.52	1.83	0.54	0.49
1024	**2.07**	**2.95**	**1.40**	**9.68**
1024	1.14	1.60	0.68	9.54
	0.36	1.18	0.34	0.38

Fig. 13. SQNC02 family data

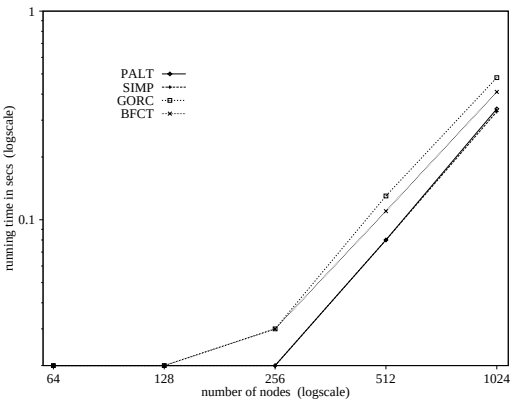
somewhat better than that of GORC. We conclude that these two codes are the best in our experiments.

PALT has lower scores on the Rand-5 family (again due to a small range of parameter values) as well as on the PNC01 and PNC02 families. However, it was the fastest or nearly the fastest code on several families, including SQNC01, SQNC02, SQNC03, and PNC03.

SIMP, with four fair and one poor score, is the least robust code in our study. Although the number of pivot operations of this code is often smaller than the number of scan operations of BFCT, a pivot is usually more time-consuming than a scan, and SIMP is often slower.

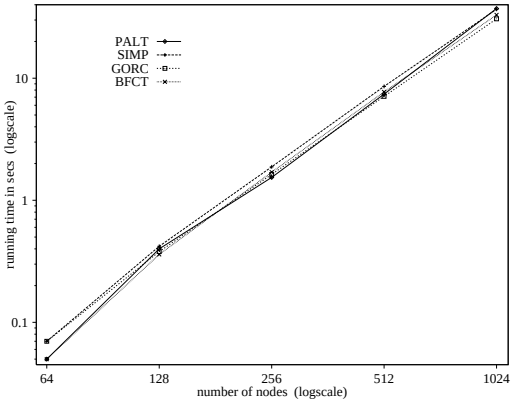
In general, more negative cycles lead to better performance for all codes. This is because more cycles make it easier to find one. In the case of many small cycles, algorithms often terminate after examining only a small portion of the graph.

Next we comment on individual problem families.



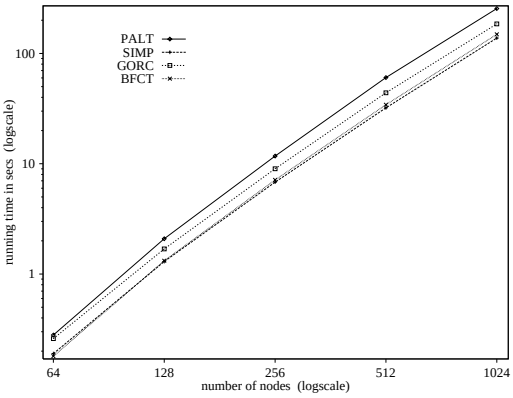
X*Y	BFCT	GORC	PALT	SIMP
64	**0.01**	**0.01**	**0.02**	**0.00**
64	0.01	0.01	0.01	0.01
	0.04	0.12	0.07	0.04
128	**0.01**	**0.00**	**0.00**	**0.00**
128	0.01	0.01	0.01	0.01
	0.02	0.04	0.04	0.02
256	**0.03**	**0.03**	**0.02**	**0.02**
256	0.00	0.01	0.00	0.00
	0.01	0.02	0.02	0.01
512	**0.11**	**0.13**	**0.08**	**0.08**
512	0.01	0.02	0.01	0.00
	0.00	0.01	0.01	0.00
1024	**0.41**	**0.48**	**0.34**	**0.33**
1024	0.01	0.03	0.01	0.00
	0.00	0.01	0.00	0.00

Fig. 14. SQNC03 family data



X*Y	BFCT	GORC	PALT	SIMP
64	**0.05**	**0.07**	**0.05**	**0.07**
64	0.01	0.00	0.01	0.00
	3.40	7.54	3.97	3.16
128	**0.36**	**0.38**	**0.40**	**0.42**
128	0.02	0.04	0.07	0.03
	4.12	10.39	5.63	3.85
256	**1.70**	**1.63**	**1.54**	**1.88**
256	0.06	0.06	0.30	0.07
	4.54	11.40	5.01	4.24
512	**7.73**	**7.10**	**7.41**	**8.57**
512	0.25	0.30	0.71	0.34
	5.30	12.45	5.82	4.94
1024	**33.03**	**30.65**	**37.24**	**36.71**
1024	1.18	2.03	1.22	1.40
	6.00	14.37	7.59	5.51

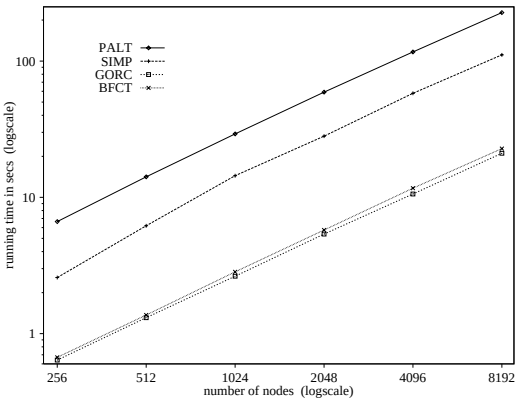
Fig. 15. SQNC04 family data



X*Y	BFCT	GORC	PALT	SIMP
64	**0.18**	**0.26**	**0.28**	**0.19**
64	0.00	0.01	0.03	0.01
	10.15	19.46	19.08	9.10
128	**1.32**	**1.69**	**2.09**	**1.30**
128	0.04	0.12	0.15	0.04
	12.03	22.38	23.21	10.73
256	**7.14**	**9.01**	**11.75**	**6.81**
256	0.24	0.47	0.80	0.21
	14.17	26.13	28.27	12.56
512	**34.29**	**43.96**	**60.43**	**32.13**
512	0.95	1.94	5.00	0.68
	16.39	30.76	34.09	14.36
1024	**149.01**	**185.24**	**255.33**	**137.90**
1024	2.15	2.28	14.77	0.67
	17.87	32.91	37.11	15.56

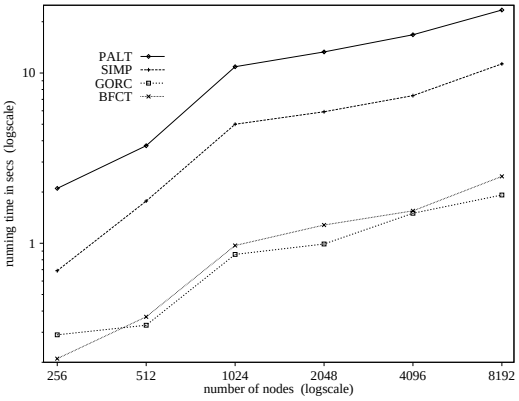
Fig. 16. SQNC05 family data

The Rand-5 family is related to the probability model considered in [29]. In that model, a network is a random graph with arc probability p. Arc lengths are chosen



X*Y	BFCT	GORC	PALT	SIMP
256	**0.67**	**0.64**	**6.64**	**2.58**
32	0.03	0.02	0.24	0.13
	12.33	17.01	132.47	6.80
512	**1.37**	**1.31**	**14.15**	**6.19**
32	0.04	0.04	0.29	0.35
	12.60	17.66	139.96	6.93
1024	**2.84**	**2.64**	**29.20**	**14.42**
32	0.03	0.04	0.54	0.51
	12.85	17.57	144.27	7.08
2048	**5.76**	**5.38**	**59.14**	**28.13**
32	0.13	0.05	0.59	0.34
	12.82	17.93	146.42	6.96
4096	**11.68**	**10.58**	**116.70**	**57.99**
32	0.11	0.13	1.40	1.24
	12.95	17.73	146.10	7.02
8192	**22.79**	**21.06**	**226.88**	**111.13**
32	0.26	0.20	2.26	2.44
	12.89	17.81	147.29	7.01

Fig. 17. PNC01 family data

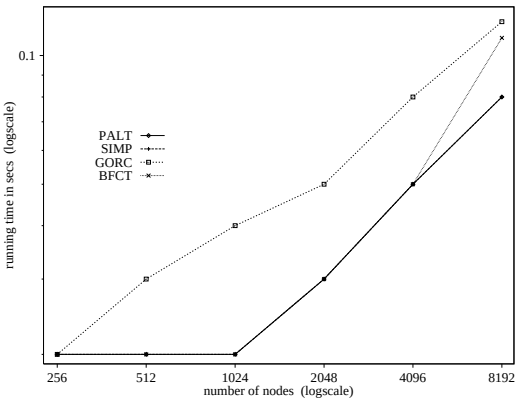


X*Y	BFCT	GORC	PALT	SIMP
256	**0.21**	**0.29**	**2.10**	**0.69**
32	0.19	0.18	2.33	0.64
	3.75	7.22	40.94	1.59
512	**0.37**	**0.33**	**3.74**	**1.77**
32	0.35	0.19	3.78	2.14
	3.34	3.93	36.58	1.73
1024	**0.97**	**0.86**	**10.89**	**5.00**
32	0.65	0.67	7.79	4.20
	4.35	5.35	53.25	2.45
2048	**1.28**	**0.99**	**13.30**	**5.92**
32	1.01	0.97	11.82	5.42
	2.76	3.10	32.29	1.47
4096	**1.55**	**1.50**	**16.77**	**7.36**
32	0.84	0.72	8.01	4.03
	1.62	2.34	20.65	0.93
8192	**2.47**	**1.92**	**23.43**	**11.31**
32	1.08	0.82	9.85	5.20
	1.26	1.47	14.62	0.69

Fig. 18. PNC02 family data

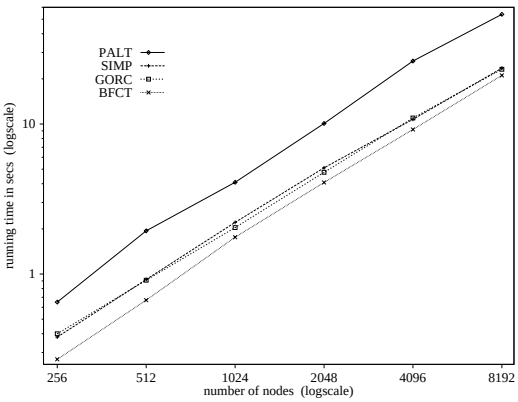
independently from the same distribution, which is symmetric around zero. One can easily show that for any $\epsilon > 0$ and $p \geq \frac{2+\epsilon}{n}$, the probability that a n'-vertex graph does not have a negative cycle is exponentially small in n'. This suggests that an incremental graph algorithm running in polynomial time, such as PALT, has $O(1)$ expected running time in the model, assuming that the initialization takes constant time. The distribution of Rand-5 graphs for $U = 32,000$ and $L = -32,000$ is similar to (but not the same as) that of random graphs with $p = 5/n$. Fig. 11 shows that for these values of U and L, PALT scans a small fraction of vertices.

However, other codes scan a small fraction of vertices as well. For a small range of parameter values (e.g. for the length lower bound of -8000), GORC scans significantly fewer vertices than the other codes and runs much faster. We conclude that in some cases, the admissible graph search strategy is superior.



X*Y	BFCT	GORC	PALT	SIMP
256	**0.01**	**0.02**	**0.02**	**0.01**
32	0.01	0.00	0.01	0.01
	0.11	0.33	0.25	0.10
512	**0.01**	**0.03**	**0.02**	**0.02**
32	0.01	0.00	0.00	0.01
	0.03	0.19	0.10	0.03
1024	**0.02**	**0.04**	**0.02**	**0.02**
32	0.00	0.01	0.00	0.01
	0.02	0.12	0.04	0.01
2048	**0.03**	**0.05**	**0.03**	**0.03**
32	0.00	0.01	0.00	0.00
	0.01	0.05	0.03	0.01
4096	**0.05**	**0.08**	**0.05**	**0.05**
32	0.01	0.02	0.00	0.00
	0.01	0.03	0.01	0.00
8192	**0.11**	**0.12**	**0.08**	**0.08**
32	0.01	0.00	0.00	0.01
	0.00	0.01	0.00	0.00

Fig. 19. PNC03 family data



X*Y	BFCT	GORC	PALT	SIMP
256	**0.27**	**0.40**	**0.65**	**0.38**
32	0.01	0.02	0.15	0.03
	4.61	10.08	13.08	4.03
512	**0.67**	**0.91**	**1.94**	**0.92**
32	0.03	0.07	0.16	0.09
	5.43	11.36	17.97	4.80
1024	**1.76**	**2.04**	**4.09**	**2.21**
32	0.10	0.09	0.14	0.09
	6.57	12.71	18.36	5.71
2048	**4.08**	**4.77**	**10.09**	**5.11**
32	0.24	0.28	1.85	0.17
	7.41	14.78	21.89	6.63
4096	**9.20**	**10.98**	**26.30**	**10.75**
32	1.46	1.05	5.02	1.30
	7.96	17.11	26.83	6.91
8192	**21.10**	**23.17**	**53.84**	**23.58**
32	1.22	1.88	6.26	1.79
	8.71	18.08	27.40	7.72

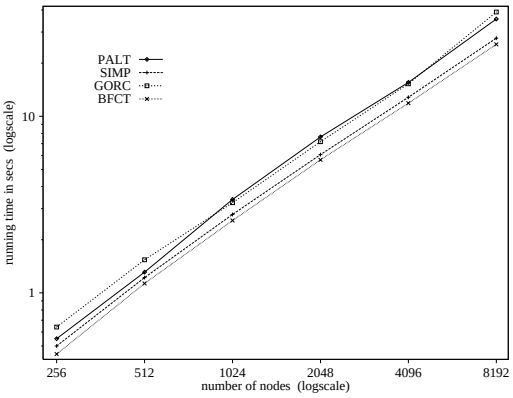
Fig. 20. PNC04 family data

On the SQNC01 and SQNC02 families, PALT clearly outperforms the other codes, although the margin is not big enough to cause a score difference.

On PNC03 problems, GORC terminates after scanning significantly more vertices than the other codes, in particular BFCT and SIMP. Similar phenomena happens on the PNC04 and PNC05 families. Thus the admissible graph cycle detection strategy can be inferior to the subtree traversal and subtree disassembly strategies.

12. Follow-up experiment

Results of Section 11 suggest that Tarjan's algorithm performs well as a shortest path algorithm. To see if this is the case, we ran the BFCT code on shortest path problem



X*Y	BFCT	GORC	PALT	SIMP
256	**0.45**	**0.64**	**0.55**	**0.50**
32	0.02	0.08	0.03	0.03
	5.77	11.02	8.43	5.42
512	**1.13**	**1.54**	**1.31**	**1.22**
32	0.03	0.14	0.08	0.03
	6.10	11.79	8.76	5.73
1024	**2.57**	**3.24**	**3.38**	**2.78**
32	0.13	0.45	0.26	0.13
	6.43	11.47	10.37	6.07
2048	**5.67**	**7.18**	**7.64**	**6.07**
32	0.08	0.83	0.65	0.15
	6.77	12.28	11.05	6.39
4096	**11.87**	**15.28**	**15.56**	**12.81**
32	0.56	1.77	1.27	0.60
	6.88	12.63	11.59	6.46
8192	**25.57**	**38.95**	**35.59**	**27.70**
32	0.48	9.85	2.54	0.75
	7.36	15.51	12.72	6.92

Fig. 21. PNC05 family data

families from [2]. The families we use are Grid-SSquare (square grids), Grid-SSquare-S (square grids with an artificial source), Grid-PHard (layered graphs with nonnegative arc lengths), Grid-NHard (layered graphs with arbitrary arc lengths), Rand-4 (random graphs of degree 4), Rand-1:4 (random graphs of degree $n/4$), and Acyc-Neg (acyclic graphs with negative arc lengths). For a detailed description of these problem families, see [2].

An interesting question is how much the subtree disassembly used by Tarjan's algorithm improves the Bellman–Ford–Moore algorithm, and how much this strategy improves Pallottino's algorithm. We also evaluate the optimized network simplex algorithm. Since the Goldberg–Radzik algorithm performed well in our previous study, one would like to know how the new codes compare with it.

To answer these questions, we include GORC, SIMP, and PALT in these experiments. (The former code is almost the same as GOR of [2].) To gauge how subtree disassembly affects performance, we also include an implementation BFP of the Bellman–Ford–Moore algorithm, and an implementation TWO-Q of Pallottino's algorithm. We use the same codes as in [2].

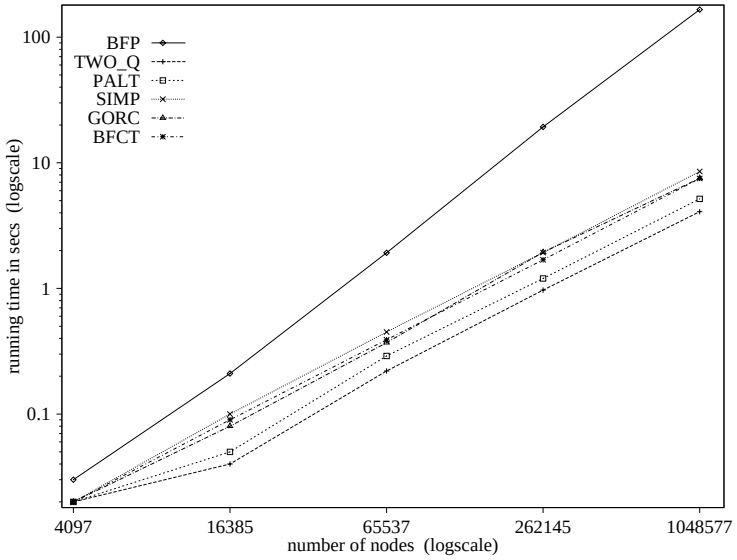
Note that for networks with nonnegative arc lengths, good implementations of Dijkstra's algorithm perform very well. In the presence of negative-length arcs, however,

	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
Grid-SSquare	⊗	◯	◯	◯	◯	◯
Grid-SSquare-S	⊗	◯	◯	◯	•	•
Grid-PHard	•	◯	◯	⊙	⊗	•
Grid-NHard	•	◯	◯	⊙	⊙	⊗
Rand-4	◯	◯	◯	◯	◯	◯
Rand-1:4	◯	◯	◯	◯	◯	◯
Acyc-Neg	•	•	◯	⊗	•	•

Fig. 22. Summary of algorithm performance in the follow-up experiment. ◯ means good, ⊙ means fair, ⊗ means poor, and • means bad

the algorithm performs poorly. For this reason, we do not include it in this study. However, the results of [2] can be used for an indirect comparison.

A summary of the follow-up experiment appear in Fig. 22 and the details – in Figs. 23–29.

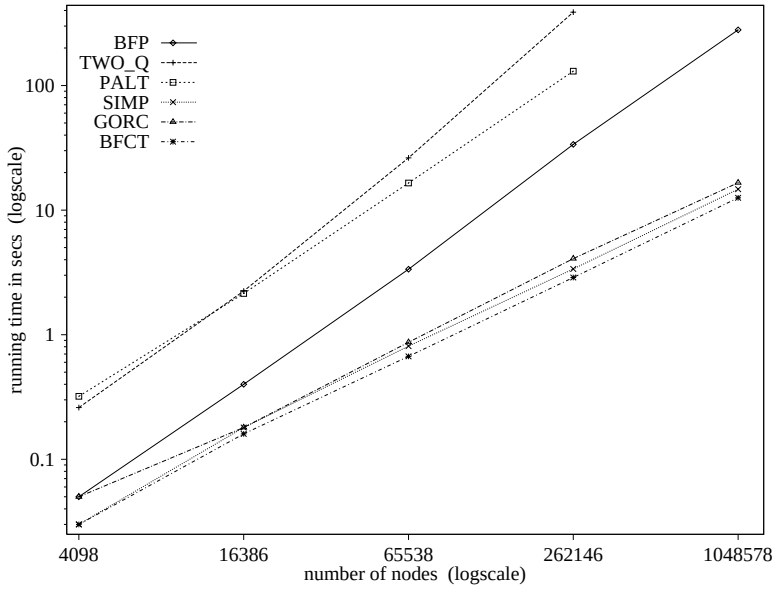


nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
4097	**0.03**	**0.02**	**0.02**	**0.02**	**0.01**	**0.02**
12288	0.00	0.01	0.00	0.00	0.01	0.01
	2.74	1.35	2.26	1.34	1.25	1.25
16385	**0.21**	**0.09**	**0.08**	**0.10**	**0.05**	**0.04**
49152	0.03	0.02	0.00	0.00	0.01	0.01
	5.05	1.43	2.29	1.42	1.26	1.26
65537	**1.92**	**0.39**	**0.37**	**0.45**	**0.29**	**0.22**
196608	0.09	0.01	0.00	0.01	0.01	0.01
	9.66	1.48	2.28	1.46	1.27	1.27
262145	**19.30**	**1.69**	**1.94**	**1.93**	**1.20**	**0.97**
786432	0.48	0.05	0.01	0.06	0.00	0.00
	19.68	1.52	2.29	1.50	1.27	1.27
1048577	**165.57**	**7.50**	**7.52**	**8.52**	**5.16**	**4.08**
3145728	4.70	0.32	0.02	0.23	0.01	0.01
	41.78	1.57	2.30	1.54	1.27	1.27

Fig. 23. Grid-SSquare family data

GORC is the most robust code in our tests, with the highest score on every problem family. Its performance is always within a factor of two of the fastest code.

BFCT's record is marred only by its bad performance on Acyc-Neg graphs. Otherwise, this is a robust algorithm. One may argue that the Acyc-Neg family is favorable for GORC, which uses depth-first search to order vertex scans line the special-purpose algorithm for acyclic graphs. However, the depth-first search is an integral part of GORC, which, unlike the special-purpose algorithm, works even if cycles are added so that the graph is no longer acyclic.

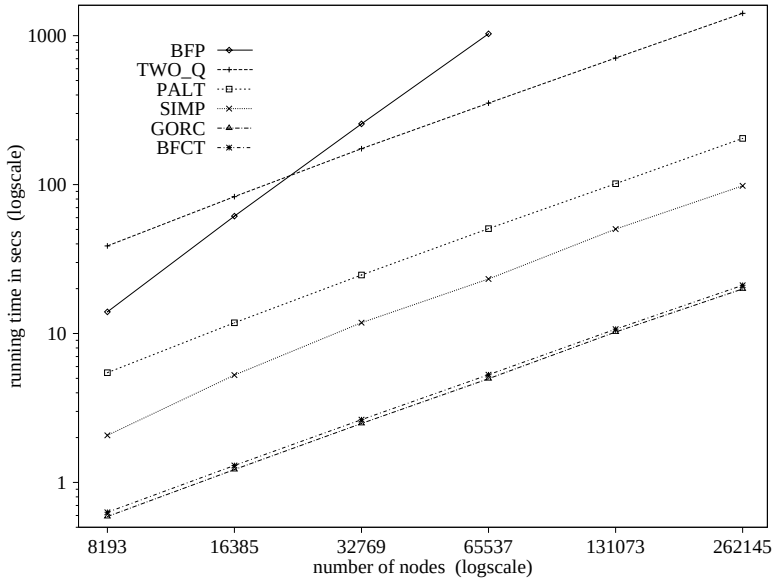


nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
4098	**0.05**	**0.03**	**0.05**	**0.03**	**0.32**	**0.26**
16385	0.01	0.01	0.01	0.00	0.03	0.02
	4.78	2.37	4.51	2.37	29.10	38.14
16386	**0.40**	**0.16**	**0.18**	**0.18**	**2.14**	**2.24**
65537	0.03	0.01	0.01	0.01	0.05	0.10
	9.19	2.48	4.57	2.46	39.21	71.31
65538	**3.35**	**0.67**	**0.87**	**0.81**	**16.49**	**26.28**
262145	0.13	0.01	0.01	0.01	0.22	0.90
	17.43	2.52	4.59	2.50	71.46	166.50
262146	**33.66**	**2.88**	**4.08**	**3.37**	**130.42**	**387.80**
1048577	0.80	0.04	0.02	0.07	3.03	8.67
	34.12	2.56	4.62	2.54	124.35	489.18
1048578	**279.75**	**12.55**	**16.58**	**14.72**		
4194305	6.38	0.21	0.80	0.27		
	70.97	2.62	4.62	2.58		

Fig. 24. Grid-SSquare-S family data

With no bad scores, one poor score and two fair scores, the network simplex algorithm performance is the third overall. Performance of SIMP is reasonable but not spectacular. Although the number of SIMP pivots is usually less than the number of GORC or BFCT scans, pivots are more expensive.

We conjectured that subtree disassembly will make TWO-Q algorithm more robust, and in fact PALT scores are higher than those of TWO-Q. Comparing PALT and TWO-Q, we note that on problems which are easy for Pallottino's algorithm (where the number of scans per vertex is close to one), subtree disassembly does not reduce the number of scan operations and slightly increases the running time. See for example Figure 23. On hard problems, subtree disassembly decreases the number of scan operations. The decrease can be relatively small, as on the Grid-



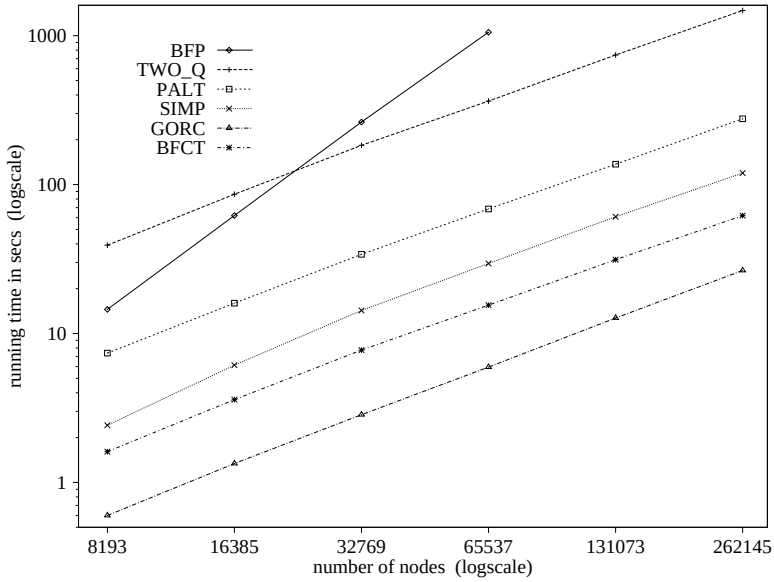
nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8193	**13.98**	**0.63**	**0.59**	**2.07**	**5.46**	**38.76**
63808	0.89	0.01	0.02	0.15	0.19	3.58
	390.13	12.44	16.90	6.73	132.21	1108.89
16385	**61.38**	**1.30**	**1.22**	**5.26**	**11.80**	**83.04**
129344	1.07	0.04	0.03	0.20	0.13	2.79
	799.87	12.53	17.98	6.94	140.82	1145.92
32769	**255.78**	**2.64**	**2.49**	**11.82**	**24.71**	**174.19**
260416	7.25	0.08	0.06	0.48	0.57	5.14
	1612.36	12.75	17.87	6.97	144.87	1190.10
65537	**1028.94**	**5.29**	**4.97**	**23.24**	**50.60**	**352.18**
522560	6.83	0.08	0.13	0.81	0.64	4.40
	3175.98	12.83	17.84	6.98	146.07	1190.76
131073		**10.70**	**10.24**	**50.24**	**101.39**	**708.27**
1046848		0.14	0.09	0.92	1.24	16.83
		12.93	17.98	7.01	146.11	1199.97
262145		**21.09**	**19.93**	**98.14**	**204.28**	**1412.31**
2095424		0.20	0.18	1.24	1.88	18.43
		12.87	17.82	7.01	147.20	1194.03

Fig. 25. Grid-PHard family data

SSquare-S family (Figure 24), or large, as on the Grid-PHard and Grid-NHard families (Figures 25 and 26). Even PALT's scores, however, are dominated by SIMP's scores.

Although the fact that BFP has the lowest scores is not surprising, it is very informative to compare it to BFCT and to see how much performance is gained by subtree disassembly. This speedup is due to the reduction in the number of scan operations.

The data presented in this section shows that Tarjan's algorithm is almost as robust as the Goldberg–Radzik algorithm and in many cases is slightly faster.



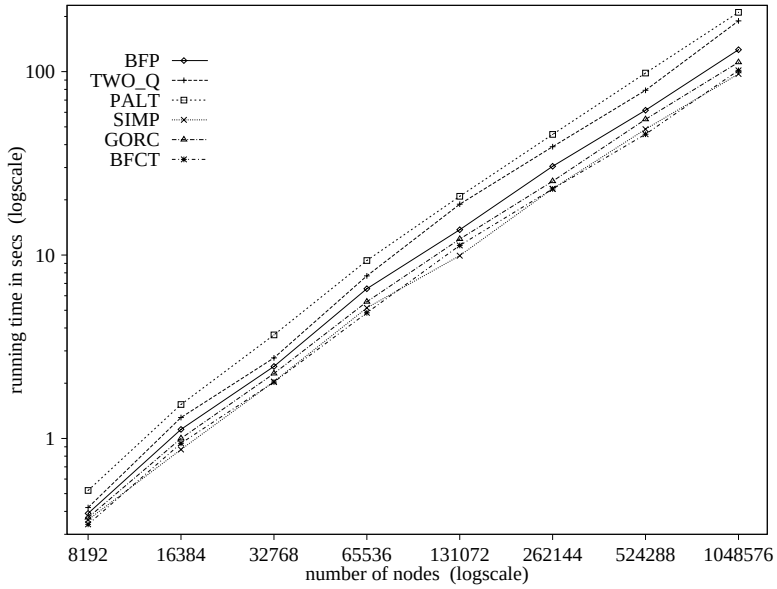
nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8193	**14.52**	**1.61**	**0.60**	**2.42**	**7.39**	**39.18**
63808	0.89	0.05	0.03	0.32	0.31	3.14
	392.09	27.01	17.88	8.68	166.24	1143.77
16385	**61.95**	**3.60**	**1.34**	**6.13**	**15.99**	**86.09**
129344	0.74	0.11	0.04	0.25	0.41	1.89
	803.31	29.11	19.89	8.94	177.25	1187.05
32769	**263.14**	**7.74**	**2.85**	**14.29**	**34.00**	**183.49**
260416	7.14	0.23	0.09	1.08	0.89	5.58
	1619.02	30.65	20.66	9.08	183.57	1231.74
65537	**1054.18**	**15.50**	**5.95**	**29.52**	**68.58**	**363.71**
522560	8.57	0.15	0.32	1.18	0.72	2.76
	3190.63	30.87	21.30	9.16	184.67	1231.16
131073		**31.32**	**12.72**	**60.85**	**137.10**	**741.91**
1046848		0.36	0.37	0.99	1.59	20.00
		31.32	22.58	9.09	185.41	1239.82
262145		**61.87**	**26.50**	**119.46**	**276.66**	**1475.43**
2095424		0.29	0.33	3.00	2.86	12.17
		31.31	23.47	9.13	186.74	1234.86

Fig. 26. Grid-NHard family data

13. Concluding remarks

While this paper was being reviewed, we learned of another negative cycle detection algorithm. This algorithm is a special case of Howard's algorithm [17] for the problem of finding an optimal policy for a Markov decision problem. Howard's algorithm, developed in the 50's, is well-known in the Control Theory community.

An interpretation of Howard's algorithm for the negative cycle problem is as follows. Assume that the source has no incoming arcs; if it does, add a new source connected to the old one by a zero-length arc. Start as in the labeling method. Then initialize the parent graph G_p to a maximal spanning tree of G rooted at s. At each iteration, if G_p

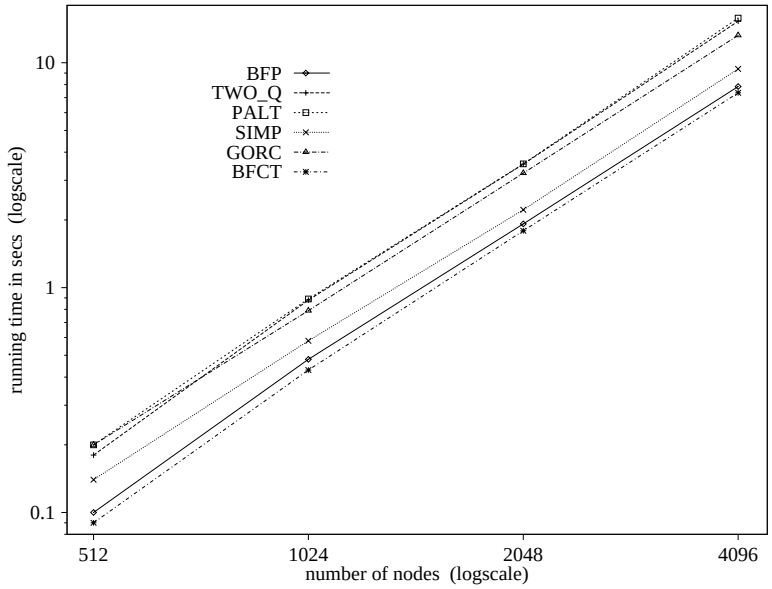


nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8192	**0.39**	**0.34**	**0.37**	**0.36**	**0.52**	**0.42**
32768	0.03	0.03	0.03	0.02	0.07	0.07
	12.23	7.75	14.58	6.90	15.37	15.90
16384	**1.12**	**0.94**	**1.00**	**0.87**	**1.53**	**1.30**
65536	0.06	0.04	0.08	0.09	0.17	0.25
	13.45	8.61	16.19	7.57	17.99	18.83
32768	**2.47**	**2.03**	**2.26**	**2.04**	**3.67**	**2.75**
131072	0.23	0.22	0.15	0.05	0.31	0.28
	13.73	8.78	16.47	7.70	18.87	19.38
65536	**6.55**	**4.84**	**5.56**	**5.15**	**9.33**	**7.70**
262144	0.45	0.37	0.40	0.31	0.72	0.69
	15.51	9.80	18.21	8.50	22.33	23.38
131072	**13.74**	**11.28**	**12.22**	**9.92**	**20.90**	**18.94**
524288	1.28	1.07	0.69	0.66	2.32	1.56
	16.75	10.48	19.48	8.97	25.08	26.07
262144	**30.46**	**22.93**	**25.27**	**22.98**	**45.47**	**38.97**
1048576	2.38	2.12	1.24	1.80	4.52	2.93
	17.61	11.07	20.76	9.47	26.25	26.95
524288	**61.60**	**45.56**	**54.97**	**48.57**	**98.29**	**79.15**
2097152	3.35	0.82	3.89	1.87	11.25	6.01
	18.19	11.38	21.02	9.69	27.92	28.41
1048576	**131.68**	**101.62**	**112.55**	**97.15**	**210.67**	**189.17**
4194304	11.39	6.64	7.84	6.70	15.86	14.28
	19.12	12.00	22.40	10.19	30.12	30.70

Fig. 27. Rand-4 family data

is a tree rooted at s, then for every vertex v set $d(v)$ to the distance from s to v in G_p. Otherwise, terminate: G has a negative cycle. To complete the iteration, examine all arcs and for every arc apply the labeling operation to it.

This algorithm (which we shall call Howard's algorithm as well) is similar to the Bellman-Ford-Moore algorithm with the amortized search cycle-detection strategy. The



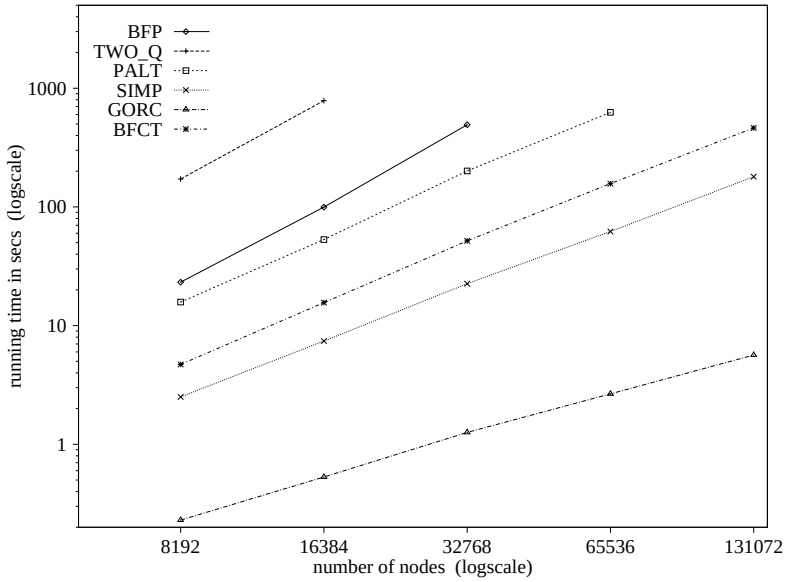
nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
512	**0.10**	**0.09**	**0.20**	**0.14**	**0.20**	**0.18**
65536	0.01	0.01	0.02	0.01	0.01	0.02
	5.32	4.40	7.91	4.02	8.45	8.70
1024	**0.48**	**0.43**	**0.79**	**0.58**	**0.89**	**0.88**
262144	0.03	0.03	0.06	0.06	0.08	0.13
	5.10	4.29	7.91	3.94	8.74	8.88
2048	**1.92**	**1.79**	**3.23**	**2.22**	**3.55**	**3.54**
1048576	0.18	0.21	0.15	0.07	0.33	0.20
	4.65	4.01	7.23	3.64	8.27	8.33
4096	**7.84**	**7.35**	**13.24**	**9.39**	**15.80**	**15.29**
4194304	0.34	0.65	0.19	0.22	1.25	1.26
	4.65	4.10	7.24	3.68	8.81	8.92

Fig. 28. Rand-1:4 family data

two major differences are that Howard's algorithm sets vertex labels to tree distances at the beginning of every iteration and that it examines all arcs during a pass (not only arcs out of the labeled vertices).[3] We ran Howard's algorithm on our problem families and found that it is not competitive with the best codes in our study. However, the idea of using G_p to update distance labels in an amortized manner is probably the first heuristic improvement of the Bellman–Ford–Moore algorithm that preserves the $O(nm)$ time bound.

In our study, GORC and BFCT are the best codes overall. The former is somewhat more robust, but the latter is in many cases a little faster. The variants of Tarjan's algorithm implemented by BFCM and BFCTN performed similarly to BFCT.

[3] The original Bellman–Ford–Moore algorithm examined all arcs as well; its scanning method variant is a later development.



nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8192	**23.24**	**4.71**	**0.23**	**2.51**	**15.81**	**171.85**
131072	1.23	0.19	0.01	0.13	0.60	23.72
	466.95	45.21	2.00	12.38	311.21	4590.70
16384	**99.58**	**15.61**	**0.53**	**7.42**	**53.08**	**786.24**
262144	3.26	0.81	0.01	0.26	1.33	111.38
	887.44	64.83	2.00	14.53	464.61	9699.84
32768	**492.75**	**51.76**	**1.26**	**22.54**	**201.15**	
524288	21.77	1.60	0.06	0.77	9.00	
	1724.82	92.76	2.00	16.19	665.26	
65536		**157.08**	**2.66**	**62.11**	**626.02**	
1048576		11.68	0.12	4.00	54.74	
		130.20	2.00	18.42	976.15	
131072		**462.43**	**5.65**	**179.79**		
2097152		48.34	0.31	13.52		
		185.73	2.00	20.56		

Fig. 29. Acyc-Neg family data

Because the Goldberg–Radzik algorithm performed so well on shortest path problems with negative length arcs in [2], we expected that it would perform well on many negative cycle problems. The good performance of Tarjan's algorithm was a surprise to us. This algorithm was motivated by adding immediate cycle detection to the Bellman–Ford–Moore algorithm at low cost. Yet in practice the resulting algorithm is much faster than the Bellman–Ford–Moore algorithm. Lemma 8 gives an explanation for this phenomena.

Performance of BFCT and BFCM is extremely similar. This seems to indicate that scanning only labeled vertices with current distance labels is more relevant to performance than scanning the vertices with the lowest tree depth.

The admissible graph search strategy works well with the Goldberg–Radzik algorithm. This strategy does not give immediate cycle detection, but in some cases finds a negative cycle before the first cycle appears in G_p.

Subtree disassembly (with or without updates) is a very good cycle detection strategy. It gives immediate cycle detection, never adds a significant overhead, and usually speeds up the underlying algorithm. Our study shows that this strategy improves the Bellman–Ford–Moore algorithm and Pallottino's algorithm. The strategy also allows a simple implementation of the ideal Bellman–Ford–Moore algorithm. This strategy may prove useful in the context of other shortest path algorithms as well.

Most students are taught only the Bellman-Ford-Moore algorithm as an algorithm for the shortest path problem with negative arc lengths. Teaching Tarjan's algorithm will expose them to an algorithm with better practical performance and build-in cycle detection while illustrating the use of amortization in algorithm design.

Acknowledgements. We would like to thank Bob Tarjan for stimulating discussions, for simplified proofs of Theorem 2 and related lemmas, and for comments on a draft of this paper. We thank Donald Goldfarb and Uwe Schwiegelshohn for bringing the level-based cycle detection strategy to our attention. We also would like to thank Harold Stone for comments that improved our presentation. We thank Satish Rao for an insightful discussion of Howard's algorithm. Finally, we would like to thank the referees for their helpful suggestions and comments.

References

1. Bellman, R.E. (1958): On a routing problem. Quart. Appl. Math. **16**, 87–90
2. Cherkassky, B.V., Goldberg, A.V., Radzik, T. (1996): Shortest paths algorithms: theory and experimental evaluation. Math. Program. **73**, 129–174
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L. (1990): Introduction to Algorithms. MIT Press, Cambridge, MA
4. Dantzig, G.B., (1951): Application of the Simplex Method to a Transportation Problem. In: Koopmans, T.C., ed., Activity Analysis and Production and Allocation, pp. 359–373. Wiley, New York
5. Denardo, E.V., Fox, B.L. (1979): Shortest–Route methods: 1. reaching, pruning, and buckets. Oper. Res. **27**, 161–186
6. Desrochers, M. (1987): A note on the partitioning shortest path algorithms. Oper. Res. Lett. **6**, 183–187
7. Dial, R.B., Glover, F., Karney, D., Klingman, D. (1979): A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. Networks **9**, 215–248
8. Dijkstra, E.W. (1959): A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271
9. Ford, L. (1956): Network Flow Theory. Technical Report P-932, The Rand Corporation
10. Ford Jr., L.R., Fulkerson, D.R. (1962): Flows in Networks. Princeton Univ. Press, Princeton, NJ
11. Gallo, G., Pallottino, S. (1988): Shortest paths algorithms. Ann. Oper. Res. **13**, 3–79
12. Glover, F., Glover, R., Klingman, D. (1984): Computational study of an improved shortest path algorithm. Networks **14**, 25–37
13. Glover, F., Klingman, D., Phillips, N. (1985): A new polynomially bounded shortest paths algorithm. Oper. Res. **33**, 65–73
14. Goldberg, A.V. (1995): Scaling algorithms for the shortest paths problem. SIAM J. Comput. **24**, 494–504
15. Goldberg, A.V., Radzik, T. (1993): A heuristic improvement of the Bellman-Ford algorithm. Applied Math. Lett. **6**, 3–6
16. Goldfarb, D., Hao, J., Kai, S.-R. (1991): Shortest path algorithms using dynamic breadth-first search. Networks **21**, 29–50
17. Howard, R.A. (1960): Dynamic Programming and Markov Processes. John Wiley, New York
18. Hung, M.S., Divoky, J.J. (1988): A computational study of efficient shortest path algorithms. Comput. Oper. Res. **15**, 567–576
19. Kennington, J.L., and Helgason, R.V. (1980): Algorithms for Network Programming. John Wiley, New York
20. Klein, M. (1967): A primal method for minimal cost flows with applications to the assignment and transportation problems. Manage. Sci. **14**, 205–220

21. Kolliopoulos, S.G., and Stein, C. (1996): Finding Real-Valued Single-Source Shortest Paths in $o(n^3)$ Expected Time. In: Proc. 5th Int. Programming and Combinatorial Optimization Conf.
22. Lawler, E.L. (1976): Combinatorial Optimization: Networks and Matroids. Holt, Reinhart, Winston, New York, NY
23. Levit, B.Ju., and Livshits, B.N. (1972): Nelineinye Setevye Transportnye Zadachi. Transport, Moscow. In Russian
24. Mondou, J.-F., Crainic, T.G., and Nguyen, S. (1991): Shortest path algorithms: A computational study with the C programming language. Comput. Oper. Res. **18**, 767–786
25. Moore, E.F. (1959): The Shortest Path Through a Maze. In: Proc. of the Int. Symp. on the Theory of Switching, pp. 285–292. Harvard University Press
26. Pallottino, S. (1984): Shortest-Path methods: complexity, interrelations and new propositions. Networks **14**, 257–267
27. Pape, U. (1974): Implementation and efficiency of Moore algorithms for the shortest root problem. Math. Program. **7**, 212–222
28. Schwiegelshohn, U. (1987): A shortest-path algorithm for layout compaction. In: Proceedings of the European Conference on Circuit Theory and Design, pp. 453–458
29. Spirakis, P., Tsakadidis, A. (1986): A Very Fast, Practical Algorithm for Finding a Negative Cycle in a Digraph. In: Proc. 13th ICALP, Lecture Notes in Computer Science 226, pp. 59–67. Springer
30. Tarjan, R.E. (1981): Shortest Paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ
31. Tarjan, R.E. (1983): Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA