Negative-Cycle Detection Algorithms

Boris V. Cherkassky *
Central Economics and Mathematics Institute
Krasikova St. 32
117418, Moscow, Russia
cher@cemi.msk.su

Andrew V. Goldberg
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
avg@research.nj.nec.com

March 1996

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 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 practical 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.

*This work was done while the author was visiting NEC Research Institute, Inc.

1 Introduction

The *negative cycle problem* is the problem of finding a negative length cycle in a network or proving that there are none (see *e.g.* [15]). The problem is closely related to the shortest path problem (see *e.g.* [1, 7, 16, 18, 19, 20]) 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 [14].

The best theoretical time bound, $O(nm)$, for the shortest path problem is achieved by the Bellman–Ford–Moore algorithm [1, 7, 18]. 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 [11] improves the Bellman–Ford–Moore bound unless N is very large. 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 [6, 7].

Most cycle detection strategies for the labeling method look for cycles in the graph of parent pointers maintained by the method. The facts that these cycles correspond to negative cycles in the input graph and that if the input graph has a negative cycle then eventually the parent pointer graph will have a cycle are well-known. Cycles in the parent graph, 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. We prove this result, which appears to have been unknown until the current paper. We also prove another basic result that appears to be new: If the input graph has a negative cycle, the distance labels maintained by the labeling method (with no cycle detection) will get arbitrarily negative.

Most experimental studies of shortest path algorithms, such as [2, 5, 8, 17], were conducted on graphs with no negative cycles. In this paper we study the practical performance of algorithms for the negative cycle problem. We also show that a cycle detection strategy of Tarjan [21] leads to improved algorithms for the shortest path problem. These algorithms are usually competitive with the fastest previous codes and are a good choice for many practical situations.

The previously known shortest path algorithms we study are the classical Bellman–Ford–Moore algorithm; the Goldberg–Radzik algorithm [12], which on shortest path problems per-

formed very well in a previous study [2]; an incremental graph algorithm of Pallottino [19], which performs well on some classes of shortest path problems; and an algorithm of Tarjan [21], which is a combination of the Bellman–Ford–Moore algorithm and a subtree-disassembly strategy for cycle detection.

We also study several 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.

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.

The Goldberg–Radzik algorithm with the admissible graph search cycle detection strategy is the most robust algorithm in our study. Tarjan's algorithm and its variations also perform very well.

This paper is organized as follows. 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.

3 Labeling Method

In this section we briefly outline the general *labeling method* [6, 7] for solving the shortest path problem. (See *e.g.* [3, 8, 22] 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 $d(v) = d(u) + \ell(u, v)$, $p(v) = u$.

Lemma 3.1 (See *e.g.* [22]) *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.*

The method terminates if and only if G does not have negative length cycles. 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.

```
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.
```

Figure 1: The SCAN operation.

The *scanning method* is a variant of the labeling method based on the scan operation. The method maintains for each vertex v the *status* $S(v) \in \{\text{unreached}, \text{labeled}, \text{scanned}\}$. Initially every vertex except s is unreached; $S(s) = \text{labeled}$. 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. 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 all vertices which were labeled at the end of pass $i-1$ and had correct distance labels at that time have been scanned. Note that we allow vertices made active during a pass to be scanned during this pass. We also allow vertices to be scanned several times during a pass.

This definition of a pass is more general than the one used in the context of the Bellman–Ford–Moore algorithm[1]. (see *e.g.* [22]). For this algorithm, a pass consists of scanning vertices that are labeled at the end of the previous pass. Our definition allows scans of other labeled vertices (as in the Goldberg–Radzik algorithm) and no scans of vertices with distance labels greater than the true distances (as in Tarjan's 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.

Lemma 3.2 (See *e.g.* [22]) *If there are no negative cycles reachable from s, then the scanning method terminates after at most $n-1$ passes.*

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

Proof. We prove by induction on k that if there is a shortest path P from s to v containing k arcs, then after k passes $d(v) = \ell(P)$. The case $k = 0$ is trivial. Assume $|P| = k > 0$ and let (u, v) be the last arc of P. Let $P = P' \cdot (u, v)$. By the inductive assumption, after pass $k - 1$ we have $d(u) = \ell(P')$. Note that after $d(u)$ is set to $\ell(P')$, $d(u)$ never changes. By the end of pass k, u has been scanned for the last time with $d(u) = \ell(P')$. After this scan, $d(v) = d(u) + \ell(u, v) = \ell(P)$. ■

4 Labeling Method and Negative Cycles

In this section we study the labeling method in the presence of negative cycles. By Lemma 3.1, 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 t, 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 $v : p(v) \neq$ **null**. This graph has the following properties.

Lemma 4.1 (See *e.g.* [22]) *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.* [22]. However, a cycle in G_p can appear after a labeling operation and disappear after a later labeling operation: see Figure 2. Surprisingly, it seems that this fact has not been known. In particular, we were unable to find Theorem 4.2 in the literature. 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.

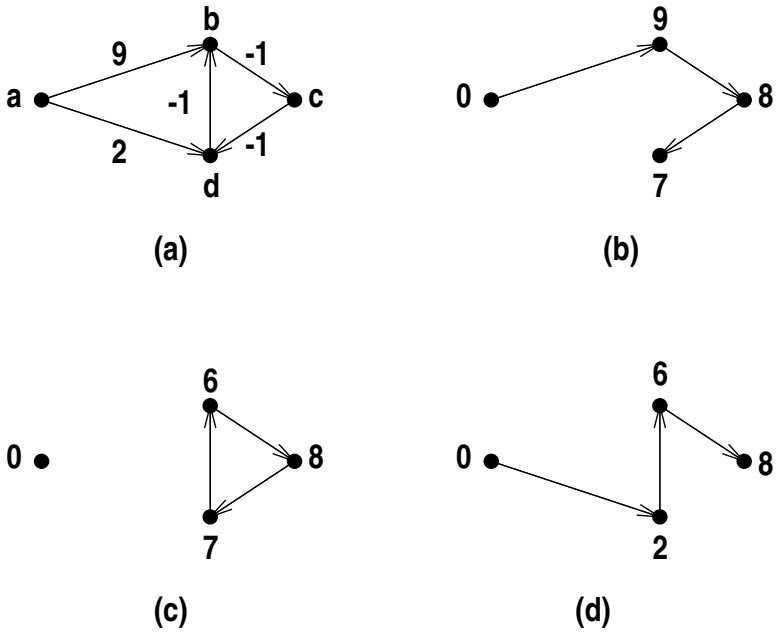


Figure 2: Disappearing cycle in G_p. (a) Input graph. (b) G_p and d after labeling operations applied to arcs $(a,b),(b,c),(c,d)$. (c) Next the labeling operation is applied to (d,b), creating a cycle. (d) Next the labeling operation is applied to (a,d), destroying the cycle.

Theorem 4.2 *If G contains a negative cycle reachable from s, then after a finite number of labeling operations G_p always has a 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 negative 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 4.3 *Suppose for some vertex v, $d(v)$ is less then the length of a shortest simple path from s to v. Then G_p has a cycle. Since $d(v)$ is non-increasing, 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 4.1 and the fact that $d(s) = 0$, we have $d(v) \geq \ell(\Gamma)$. This contradicts the definition of v. ■

Corollary 4.4 *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 we were unable to find a proof in the literature. Next we prove the following theorem.

Theorem 4.5 *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 then the length of a shortest simple path from s to u.*

Lemma 4.3 and Theorem 4.5 imply Theorem 4.2, but the proof of the latter theorem is simpler.

The proof of Theorem 4.5 requires several definitions and an auxiliary lemma. 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 non-empty path $P(v) = v_1 \cdot \ldots \cdot v_t$. If $d(s) < 0$, then Theorem 4.5 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 4.6 *For any $1 \leq i < t$, $d'(v_i) + \ell(v_i, v_{i+1}) = d'(v_{i+1})$.*

Corollary 4.7 *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.

Now we are ready to prove Theorem 4.5.

Proof of Theorem 4.5. Recall that we can assume that $d(s) = 0$. By definition of C, if a distance label of a vertex u is below $-(n-1)C$, then the length of a simple path from s to u is greater then $d(u)$. We prove that eventually the distance label of some vertex falls below $-(n-1)C$.

Partition a path $P(v)$ into subpaths containing $n+1$ vertices each; ignore the last subpath if incomplete. Traverse such a subpath. Since the subpath contains $n+1$ vertices, we will visit a vertex twice. Let v be the first vertex visited twice. Let Γ be the subpath of P traversed between the first and the second visit of v, *i.e.*, $\Gamma = v_i \cdot \ldots \cdot v_j$ and $v_i = v_j = v$. By its definition, Γ is a simple cycle. By Lemma 4.6, $d'_j = d'_i + \ell(\Gamma)$.

Let $\epsilon > 0$ be such that $-\epsilon$ is the biggest length of a negative simple cycle. Since the number of simple cycles is finite, ϵ is well-defined. We have $d'_j \leq d'_i - \epsilon$.

If G contains a negative cycle, then, because the paths do not repeat, for some vertex v the number of arcs on $P(v)$ must be unbounded as the algorithm runs. It is easy to see that the first finite distance label of a vertex is at most $(n-1)C$. Consider the point of execution when $P(v)$ contains at least $(n+1)n(1+2(n-1)C/\epsilon)$ vertices. Then for some vertex u, $d(u)$ decreases by at least ϵ at least $1+2(n-1)C/\epsilon$ times. Therefore at this point $d(u) < -(n-1)C$. ■

Remark. The bound on the number of labeling operatins in Theorem 4.5 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 3.2.

Lemma 4.8 *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 3.2 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 4.3 completes the proof. ■

5 Labeling Algorithms

Different strategies for selecting labeled vertices 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 [16, 20] and the threshold algorithm [9, 10], which were not as robust as other algorithms in our previous study [2].

5.1 The Bellman–Ford–Moore Algorithm

The Bellman–Ford–Moore algorithm, due to Bellman [1], Ford [7], and Moore [18], maintains the set of labeled vertices in a 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.

We define a *pass* over the queue inductively. Initialization, during which the source s is added to the queue, is pass 0. For $i > 0$, pass i consists of processing vertices which were added to the queue during pass $i - 1$.

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

Theorem 5.1 (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 Goldberg–Radzik Algorithm

Goldberg and Radzik [12] suggested the following 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 5.2 [12] *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.* [11].)

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 correctness or the running time bound given above.

Remark. 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.

5.3 Incremental-Graph Algorithms

In this section we describe the incremental graph framework and Pallottino's algorithm [19].

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 algorithms, TWO-Q 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). We call S_1 the *high-priority set* and S_2 the *low-priority set*.

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. As the above discussion of the restricted scan algorithms suggests, the worst-case running time of TWO_Q is polynomial.

Theorem 5.3 [19] *Pallottino's algorithm runs in $O(n^2m)$ time in the worst case, assuming no negative cycles.*

5.4 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.* [13].

At every step, a generic network simplex algorithm for shortest paths finds an arc (u, v) with negative reduced cost, applies a labeling operation to it, and updates distance labels of vertices in v'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 v's subtree become labeled. This tends to create many labeled vertices, most with distance labels greater than the true distance. 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 v 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 v, we add all children of v 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 5.4 *The optimized network simplex algorithm runs in $O(n^2m)$ time.*

Remark. 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.

This method has two major disadvantages. First, it determines if a negative cycle exists but does not find a negative cycle. Second, 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 4.5. If distance label of a vertex falls below $-(n-1)C$ or if the distance label of the source becomes negative, 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 a successor 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 simple cycle detection method, often discussed in textbooks, 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 4.2 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 [11], 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.* [13].) 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 [21], 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 greater than their true distances. 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 distance labels greater than their true distances, this tends to speed the algorithm up.

A combination of the FIFO selection rule and subtree disassembly yields Tarjan's algorithm [21] for the negative cycle problem.

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, the children of u 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.

7 Algorithms Studied

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 study time-out and distance lower bound strategies because we do not think

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

Figure 3: *Summary of negative cycle algorithms.*

these strategies can be competitive.

Two algorithms have natural cycle detection strategies associated with them: the Goldberg–Radzik algorithm has admissible graph search and the optimized network simplex algorithm has subtree traversal. We implemented these algorithms only with their natural cycle detection mechanisms. The resulting codes are GORC and SIMP, respectively. In our study of shortest path algorithms [2], we concluded that the Goldberg–Radzik algorithm is the best overall on problems with negative-length arcs. Data of this paper confirms this conclusion.

Consider the Bellman–Ford–Moore algorithm. We implemented it with walk to the root, amortized search, subtree disassembly, and subtree disassembly with update strategies. The corresponding codes are BFCF, BFCS, BFCT, and BFCTN, respectively. As we shall see, the first two codes are not good. The third code, BFCT, implements Tarjan's algorithm [21]. This code performed very well in our study, greatly outperforming implementations of the Bellman–Ford–Moore algorithm with scan order unaffected by cycle detection strategies. This is an interesting example of how the subtree disassembly strategy improves the performance of the underlying algorithm.

A simple variation of Tarjan's algorithm implements the "ideal" 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 implements the modified

algorithm.

Consider an execution of the modified algorithm. Given a tree, we say that the root is at level 1, and if a vertex is at level i then its children are at level $i+1$. 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 level in G_p of the vertex at the time of the scan is k. An equivalent statement is that for each scan, the algorithm selects a labeled vertex with the minimum level in G_p. Analysis of all known $O(nm)$ shortest path algorithms is based on the fact that at pass k, they scan all vertices at level k; the algorithms may scan other vertices as well but the analysis does not depend on this. In this sense, BFCM is the ideal $O(nm)$ algorithm – it performs exactly those scans which are needed for the analysis to go through.

Pallottino's algorithm is an incremental graph algorithm that uses the Bellman–Ford–Moore algorithm in the inner loop. Since Tarjan's algorithm gives improved performance, it is natural to use it in Pallottino's algorithm. The resulting code, PALT, can be viewed as Pallottino's algorithm with the subtree disassembly cycle detection strategy.

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 [8]. 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. 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.

The number of scans per vertex is a machine-independent measure of algorithm performance which is closely correlated with the running time for algorithms with inexpensive cycle detection

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

Figure 4: *Preliminary experiment. No cycles*

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

Figure 5: *Preliminary experiment. One small cycle*

strategies.

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, and BFCTN 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

X*Y	BFCF	BFCS	BFCT	BFCM	BFCTN
64	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
64	0.00	0.01	0.00	0.01	0.00
	0.07	0.20	0.04	0.04	0.04
128	**0.01**	**0.02**	**0.01**	**0.02**	**0.01**
128	0.01	0.01	0.01	0.01	0.01
	0.02	0.20	0.02	0.01	0.02
256	**0.02**	**0.07**	**0.03**	**0.03**	**0.02**
256	0.01	0.00	0.00	0.00	0.00
	0.01	0.20	0.01	0.01	0.01

Figure 6: *Preliminary experiment. X small cycles*

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

Figure 7: *Preliminary experiment. 16 medium cycles*

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

Figure 8: *Preliminary experiment. One Hamiltonian cycle.*

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.

Next we compare cycle detection strategies for the Bellman–Ford–Moore algorithm. In square grids, the tree paths are relatively long, and the walk to the root strategy is very expensive compared to the other strategies. See, for example, Table 4 for instances with no negative cycles where BFCF and BFCS do the same number of scans on the same problem instance. The former code, however, is much slower than the latter, and the speed ratio grows with the problem size.

The only exception is the family with many small cycles. On this family, the first cycle of parent pointers appears after only a small fraction of the graph is examined, the tree paths are relatively short, and the walk to the root strategy does reasonably well. This family demonstrates the disadvantage of the amortized search strategy, which finds the cycle much later than the other methods. For example, for the biggest problem, BFCS finds the cycle after doing about twenty times more scans than BFCF.

In these experiments, as in all other experiments, the subtree disassembly implementations BFCT, BFCM, and BFCTN outperform BFCF and BFCS. This happens for two reasons. First, subtree disassembly strategy has immediate cycle detection at low amortized cost. Second, this strategy tends to reduce the number of scans, often by a significant amount.

Comparing BFCT, BFTM, and BFCTN codes, we observe that the performance of these codes is very close. This holds for other problem families as well.

To avoid presenting uninteresting data, in the main experiment we do not give the data for the BFCF and BFCS codes, whose performance is never significantly better than that of BFCT, and often considerably worse. 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 16 problem families with four underlying graph types produced by two generators. Table 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

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	LNC01	long grids, $n = X \cdot 16$	0	
	LNC02		1	3
	LNC03		$X/8$	3
	LNC04		8	$X/8$
	LNC05		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

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

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.

Grid networks are grids embedded in a torus. Vertices of these networks correspond to points on the plane with integer coordinates $[x, y]$, $0 \leq x < X$, $0 \leq y < Y$. These points are connected "forward" by *layer* arcs of the form $([x, y], [x + 1 \bmod X, y])$, $0 \leq x < X$, $0 \leq y < Y$, 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 $[1, 100]$.

We use two types of grid networks in our experiment. Skeletons of SQNC** problems are square grids with $X = Y$. Skeletons of LNC** problems are long grids with $Y = 16$.

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** 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 Table 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. Because of the use of common sense and the preliminary experiment to filter out clearly uncompetitive algorithms, on many problem classes performance of the remaining algorithms is close.

11.1 Random Graphs

In the random graph experiment, the lower bound of the length range changes from zero to more and more negative values. More negative lower bounds lead to more negative cycles in the graph. Table 1 gives data for this experiment.

In general, more negative cycles lead to better performance for all codes. When the number of negative cycles is very small or very large, all codes perform similarly. This is because the code performance is similar for random graphs with no negative cycles, and for random graphs with very many negative cycles all codes find such a cycle almost immediately.

For intermediate numbers of negative cycles, GORC performs better than the other codes. This happens because GORC discovers the cycles during the very first depth-first search of the admissible graph.

Remark. We conducted experiments on random graphs with different densities, and the results were similar.

11.2 Square Grids

Tables 2–6 give data for the square grid families. With no negative cycles, the best code is PALT, but BFCT and GORC are not much slower. Because the subtrees traversed during pivots are relatively large, SIMP is the significantly slower than the other codes.

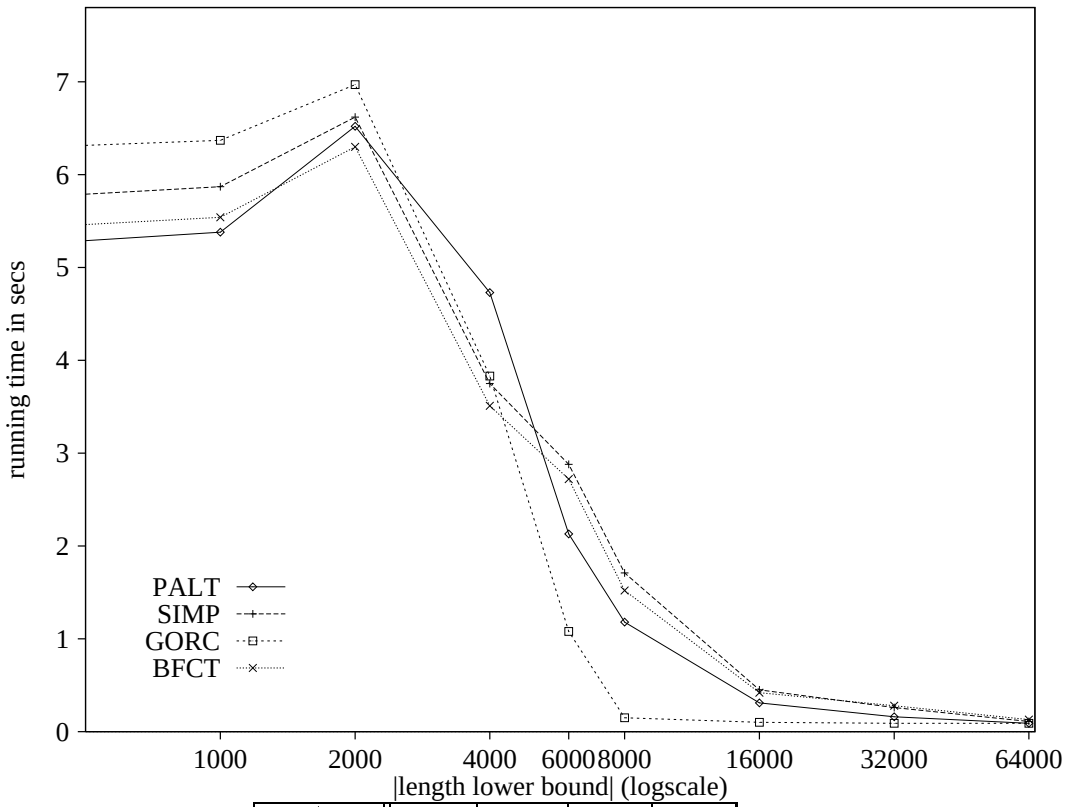
Adding a small negative cycle makes the problems simpler but preserves the relative performance of the codes.

Adding many small cycles makes the problems much simpler. All algorithms discover a negative cycle after looking at a small portion of the graph. All codes perform very well; PALT and SIMP are a little faster then the other two codes.

Adding a few long negative cycles makes the problem harder for all codes except SIMP, for which the problem becomes simpler because of the reduction in the subtree size. The codes perform very similarly on this family.

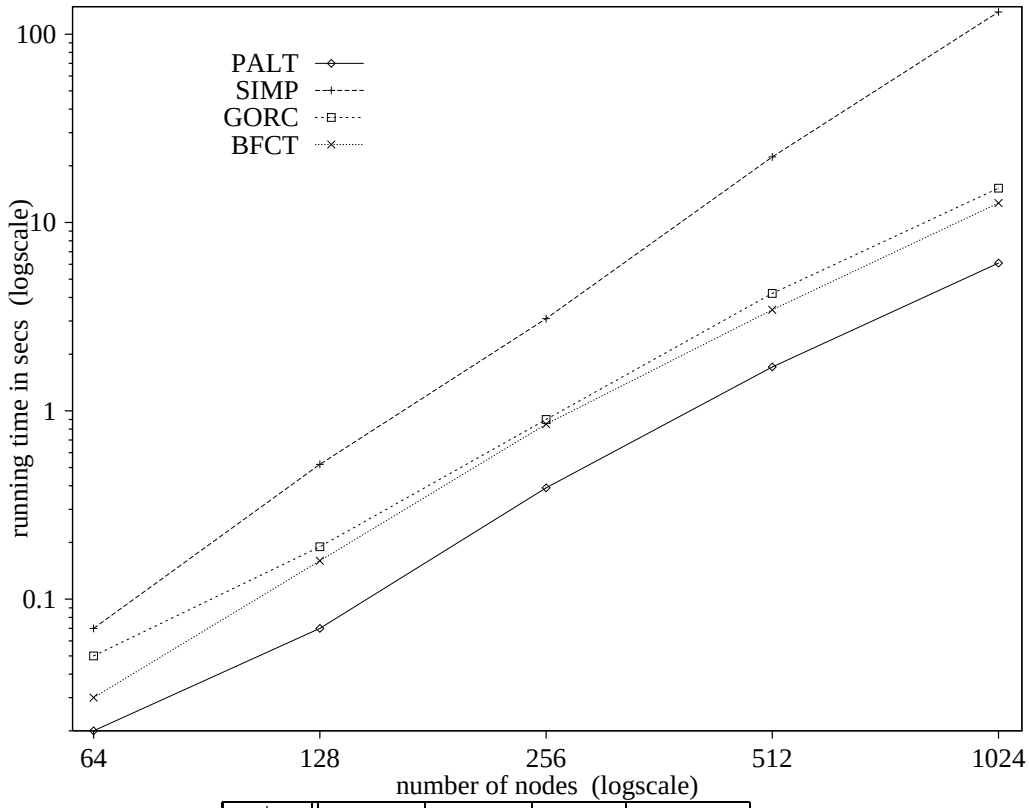
Adding one Hamiltonian cycle makes the problem harder for all codes except SIMP. All codes perform similarly, with BFCT and SIMP the fastest, GORC slightly slower, and PALT a little slower yet.

On square grids, BFCT and PALT have the best overall performance, with GORC not far behind. On these families SIMP is not as robust as the other codes.



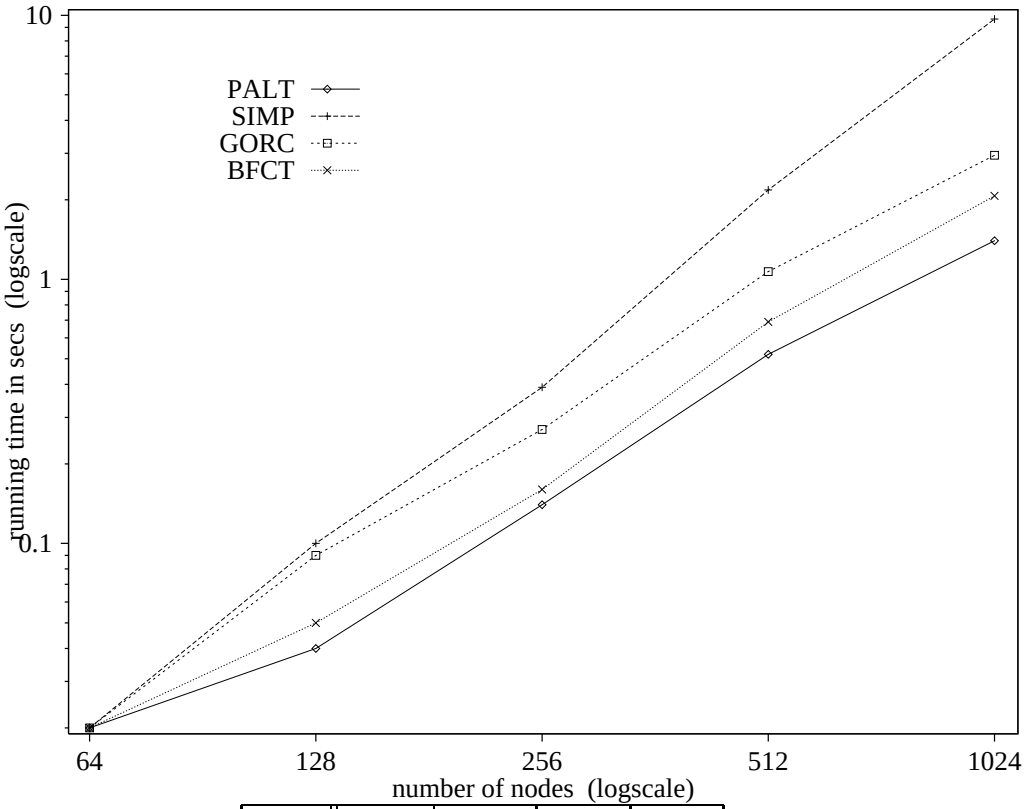
min/max	BFCT	GORC	SIMP	PALT
0	**4.76**	**5.83**	**5.07**	**4.47**
32000	0.17	0.19	0.17	0.21
	2.13	3.62	2.06	2.38
-1000	**5.54**	**6.37**	**5.87**	**5.38**
32000	0.45	0.26	0.46	0.61
	2.47	3.99	2.38	2.86
-2000	**6.30**	**6.97**	**6.62**	**6.52**
32000	1.92	2.36	2.02	1.95
	2.81	4.42	2.68	3.47
-4000	**3.51**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

Table 1: Rand-5 family data.



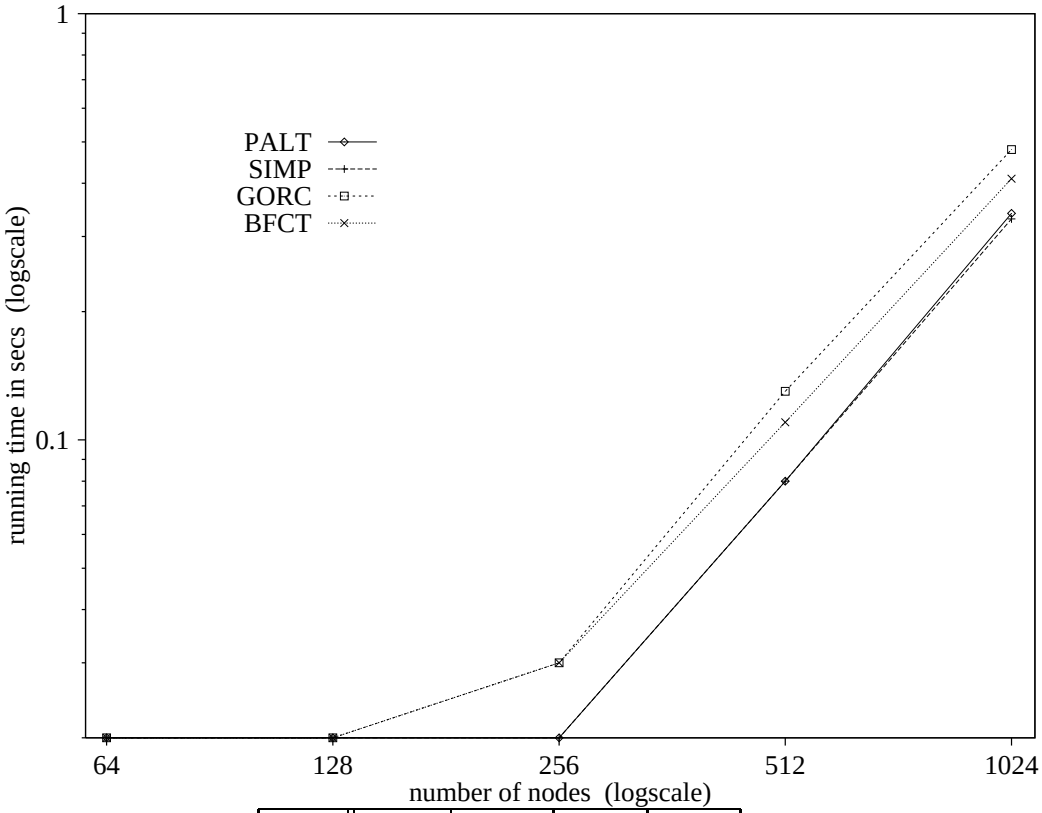
X*Y	BFCT	GORC	PALT	SIMP
64	**0.03**	**0.05**	**0.02**	**0.07**
64	0.00	0.01	0.01	0.00
	2.99	7.10	2.01	2.90
128	**0.16**	**0.19**	**0.07**	**0.52**
128	0.01	0.01	0.00	0.04
	2.96	7.01	1.91	2.84
256	**0.85**	**0.90**	**0.39**	**3.09**
256	0.03	0.01	0.01	0.14
	2.93	7.31	1.94	2.80
512	**3.44**	**4.20**	**1.71**	**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

Table 2: SQNC01 family data.



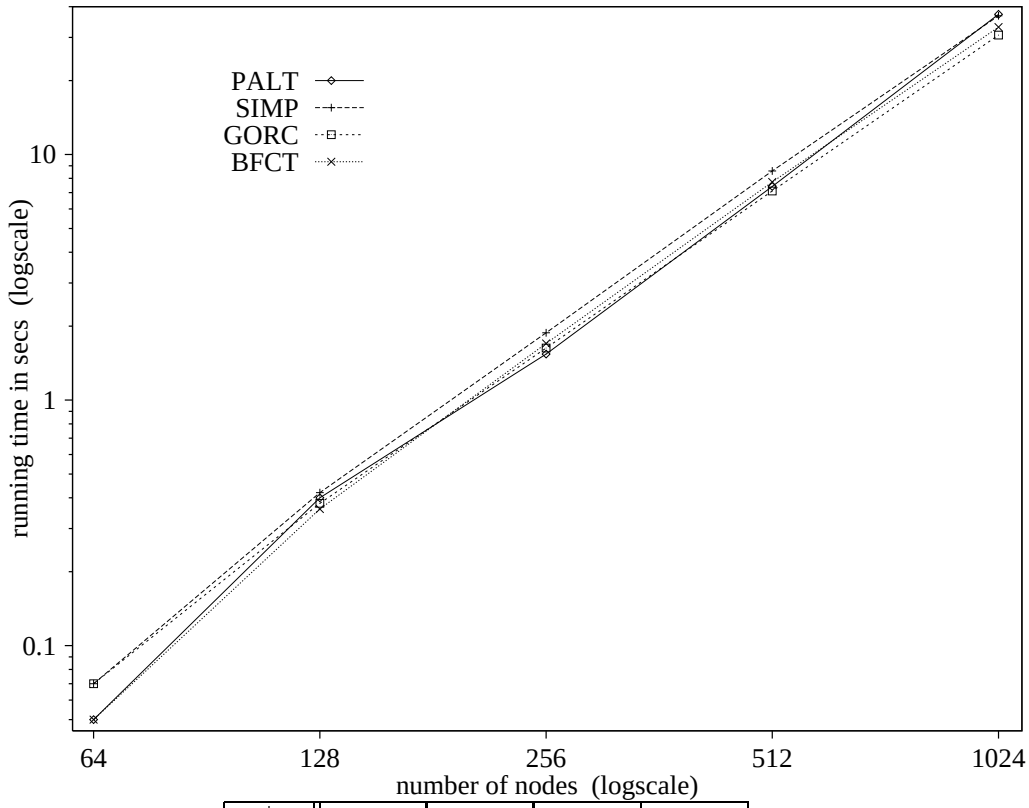
X*Y	BFCT	GORC	PALT	SIMP
64	**0.01**	**0.02**	**0.00**	**0.02**
64	0.01	0.01	0.01	0.02
	0.76	2.62	0.88	0.74
128	**0.05**	**0.09**	**0.04**	**0.10**
128	0.03	0.04	0.02	0.06
	0.76	2.76	0.91	0.74
256	**0.16**	**0.27**	**0.14**	**0.39**
256	0.14	0.23	0.11	0.41
	0.52	1.99	0.61	0.50
512	**0.69**	**1.07**	**0.52**	**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

Table 3: SQNC02 family data.



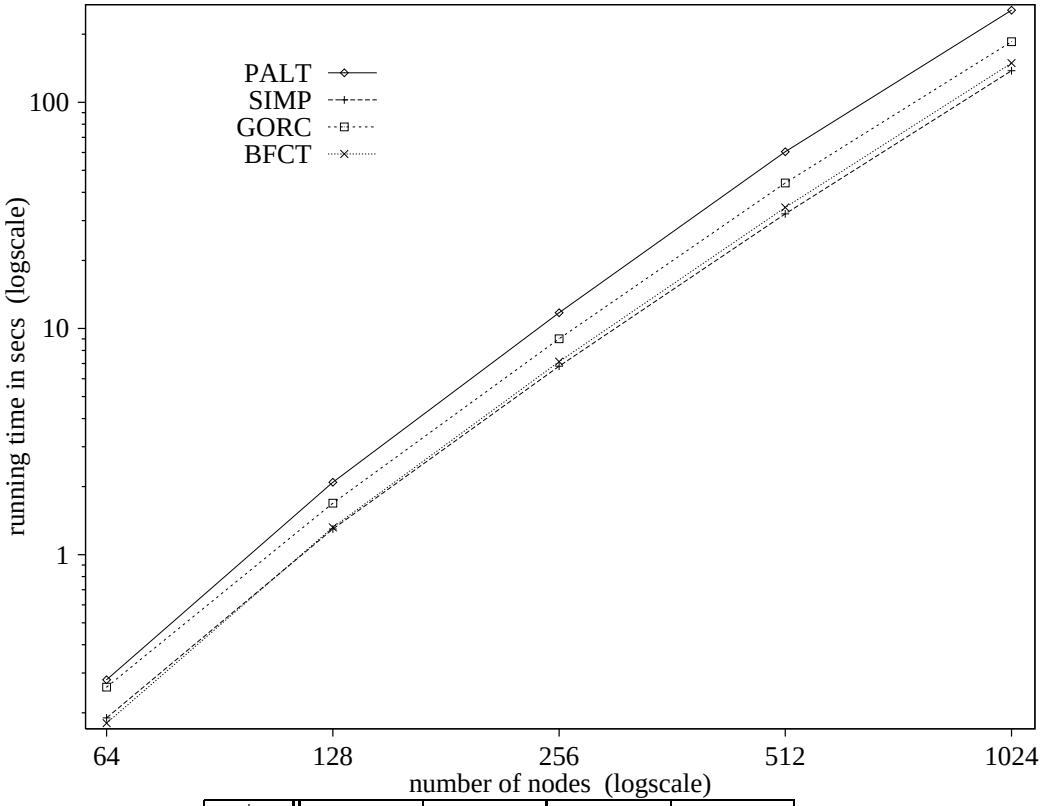
X*Y	BFCT	GORC	PALT	SIMP
64	**0.01**	**0.01**	**0.02**	**0.00**
64	0.01	0.01	0.01	0.01
	0.04	0.12	0.07	0.04
128	**0.01**	**0.00**	**0.00**	**0.00**
128	0.01	0.01	0.01	0.01
	0.02	0.04	0.04	0.02
256	**0.03**	**0.03**	**0.02**	**0.02**
256	0.00	0.01	0.00	0.00
	0.01	0.02	0.02	0.01
512	**0.11**	**0.13**	**0.08**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

Table 4: 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

Table 5: 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

Table 6: SQNC05 family data.

11.3 Long Grids

Tables 7–11 give data for the long grid families. With no negative cycles, PALT is the fastest code. The other codes are slower by about a factor of two.

Adding a small negative cycle makes the problems simpler but does not change the relative code performance.

Adding many small negative cycles makes the problem very simple for all codes and reduces the difference in performance.

Adding several long negative cycles makes the problem harder for all codes. All codes perform very similarly, although PALT is somewhat faster for small problem sizes and slower for large sizes.

Adding one Hamiltonian cycle makes the problem harder. Code performance is similar; BFCT and SIMP are the fastest codes and PALT is the slowest, but it is always within a factor of two of the fastest codes.

On long grids, all codes under consideration perform well.

11.4 Layered Graphs

Tables 12–16 give data for the layered graph families. Without negative cycles, BFCT and GORC are the fastest codes, and PALT is the slowest, loosing by about a factor of ten. SIMP is also slow, loosing to the fastest codes by about a factor of five.

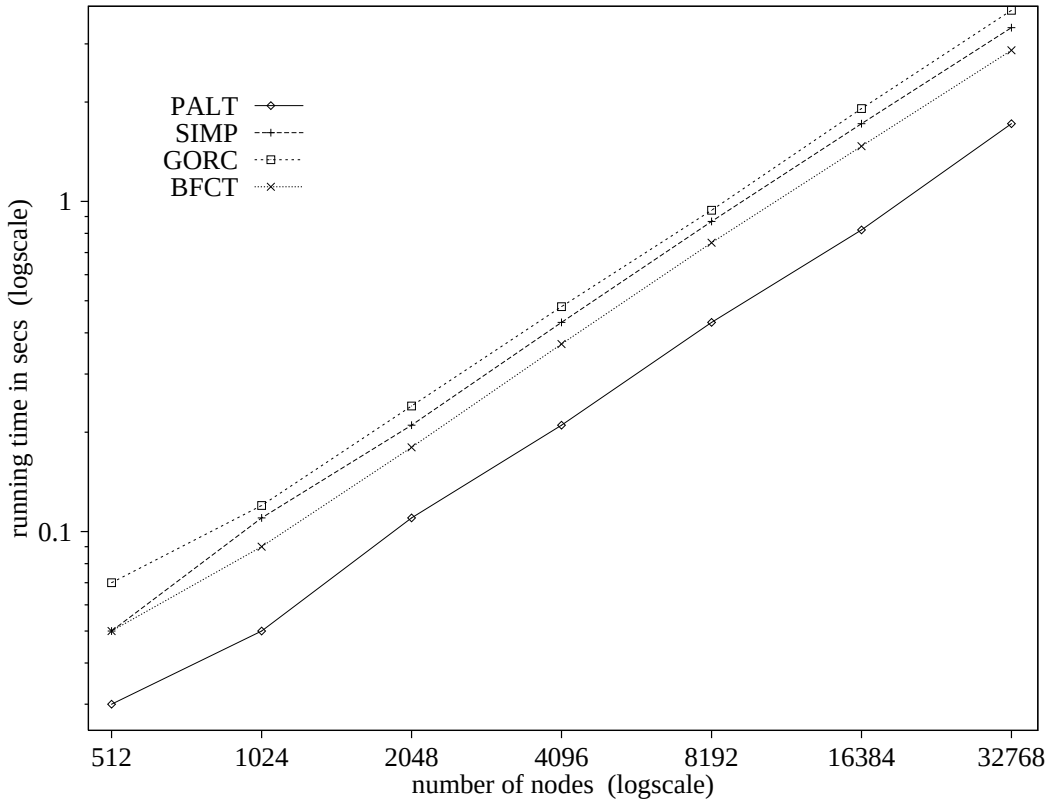
Adding a small negative cycle makes the problems simpler but does not change the relative code performance.

Adding many small negative cycles makes the problem very simple. GORC is somewhat slower than the other codes; the latter perform very similarly.

Adding several long negative cycles makes the problem simpler for PALT and SIMP. The latter code catches up with GORC. BRFC is by a small margin the best code on this family.

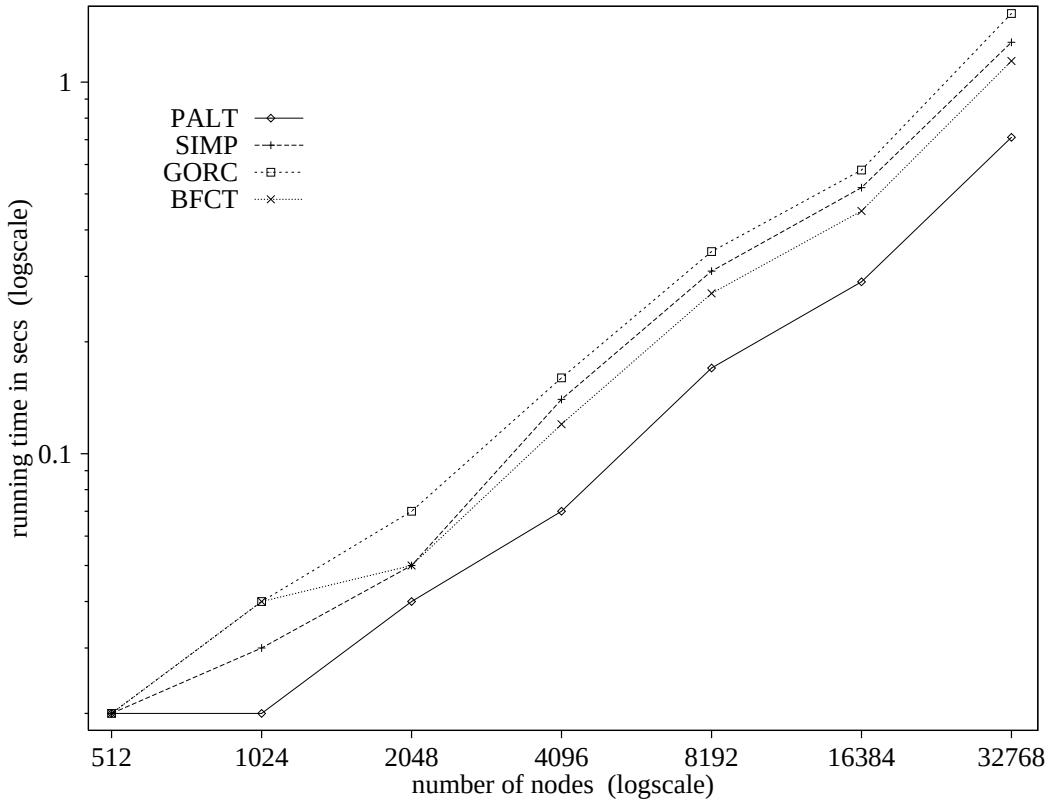
With one Hamiltonian cycle added, the code performance is similar. The fastest code is BFCT, with SIMP performing almost as well.

The overall best code on layered networks is BFCT; GORC also performs well. The other two codes are not as robust on these families.



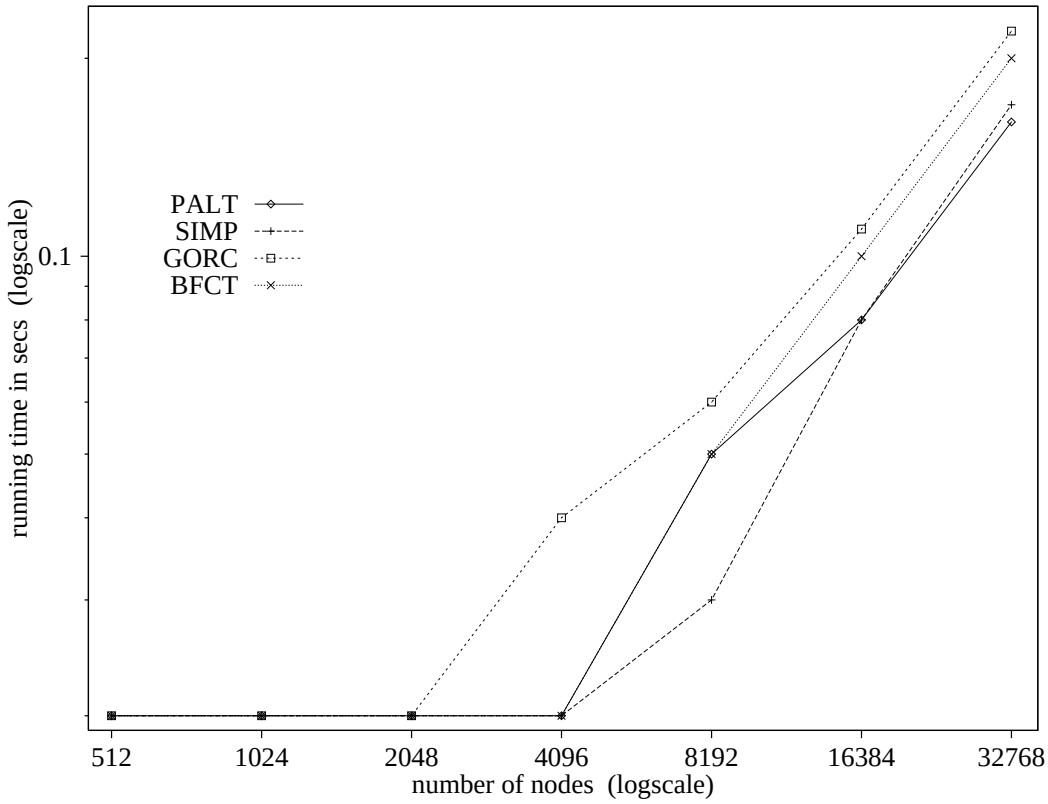
X*Y	BFCT	GORC	PALT	SIMP
512	**0.05**	**0.07**	**0.03**	**0.05**
16	0.01	0.01	0.00	0.00
	3.34	6.33	1.76	3.24
1024	**0.09**	**0.12**	**0.05**	**0.11**
16	0.01	0.00	0.00	0.01
	3.30	6.26	1.81	3.19
2048	**0.18**	**0.24**	**0.11**	**0.21**
16	0.00	0.01	0.01	0.01
	3.28	6.24	1.81	3.19
4096	**0.37**	**0.48**	**0.21**	**0.43**
16	0.00	0.01	0.01	0.00
	3.31	6.29	1.76	3.22
8192	**0.75**	**0.94**	**0.43**	**0.87**
16	0.01	0.01	0.00	0.01
	3.31	6.29	1.81	3.22
16384	**1.47**	**1.91**	**0.82**	**1.72**
16	0.00	0.01	0.00	0.03
	3.30	6.27	1.71	3.21
32768	**2.87**	**3.79**	**1.72**	**3.36**
16	0.02	0.04	0.00	0.02
	3.31	6.27	1.81	3.21

Table 7: LNC01 family data.



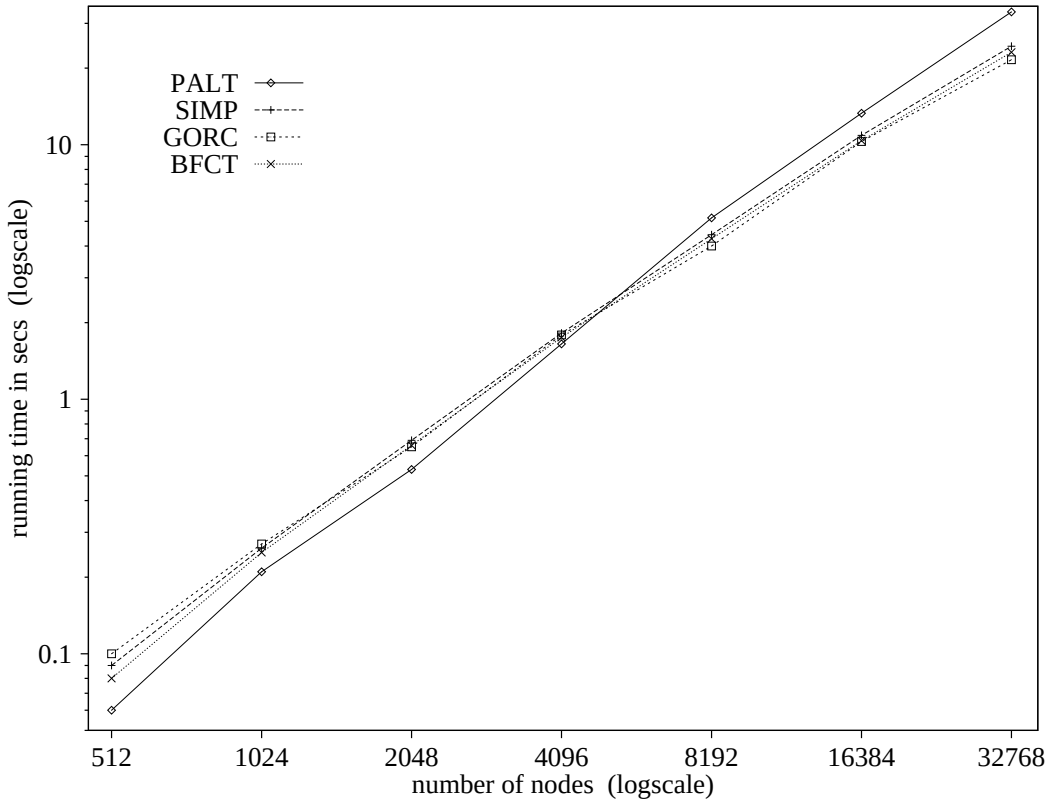
X*Y	BFCT	GORC	PALT	SIMP
512	**0.02**	**0.02**	**0.01**	**0.01**
16	0.00	0.00	0.01	0.01
	0.71	1.40	0.41	0.69
1024	**0.04**	**0.04**	**0.02**	**0.03**
16	0.01	0.02	0.00	0.02
	1.09	2.12	0.67	1.05
2048	**0.05**	**0.07**	**0.04**	**0.05**
16	0.02	0.03	0.02	0.02
	0.74	1.43	0.44	0.71
4096	**0.12**	**0.16**	**0.07**	**0.14**
16	0.04	0.05	0.02	0.05
	0.93	1.75	0.54	0.89
8192	**0.27**	**0.35**	**0.17**	**0.31**
16	0.15	0.21	0.08	0.17
	1.08	2.04	0.62	1.04
16384	**0.45**	**0.58**	**0.29**	**0.52**
16	0.22	0.29	0.13	0.27
	0.88	1.67	0.49	0.85
32768	**1.14**	**1.53**	**0.71**	**1.28**
16	0.67	0.90	0.39	0.80
	1.18	2.22	0.64	1.13

Table 8: LNC02 family data.



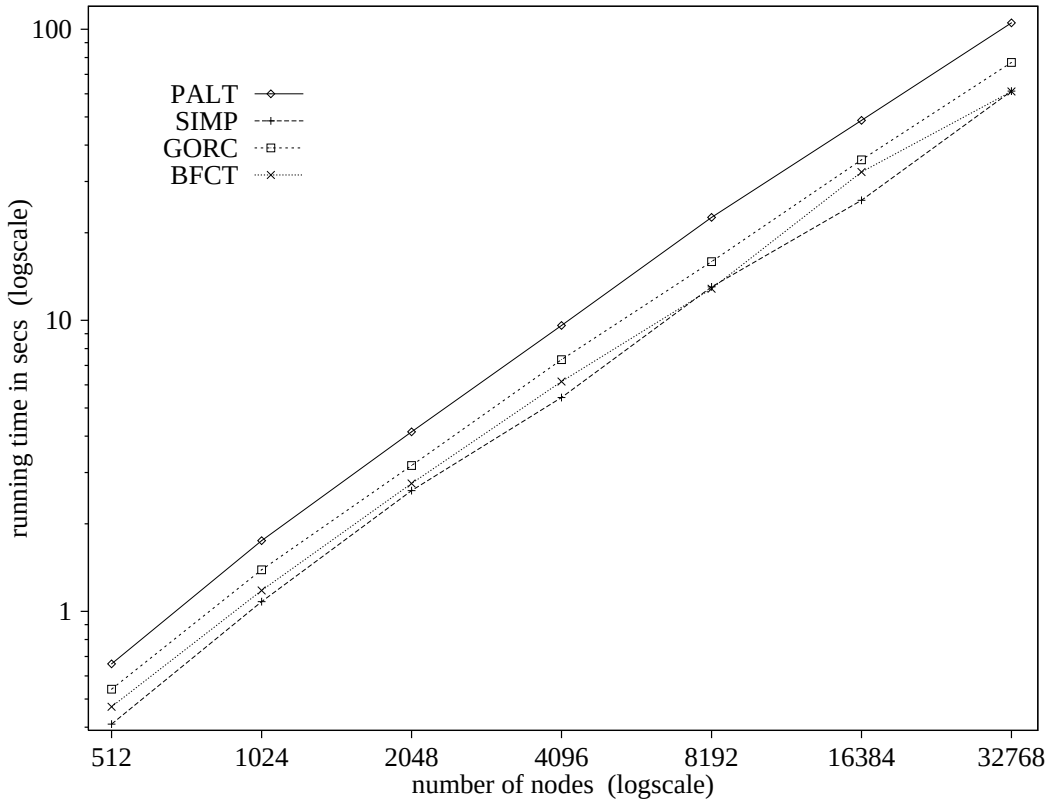
X*Y	BFCT	GORC	PALT	SIMP
512	**0.01**	**0.00**	**0.01**	**0.00**
16	0.01	0.01	0.01	0.01
	0.02	0.05	0.02	0.02
1024	**0.00**	**0.01**	**0.00**	**0.00**
16	0.00	0.01	0.01	0.01
	0.00	0.01	0.01	0.00
2048	**0.02**	**0.02**	**0.01**	**0.02**
16	0.01	0.01	0.01	0.01
	0.00	0.01	0.00	0.00
4096	**0.02**	**0.04**	**0.02**	**0.02**
16	0.00	0.01	0.00	0.00
	0.00	0.00	0.00	0.00
8192	**0.05**	**0.06**	**0.05**	**0.03**
16	0.00	0.02	0.01	0.01
	0.00	0.00	0.00	0.00
16384	**0.10**	**0.11**	**0.08**	**0.08**
16	0.00	0.04	0.00	0.00
	0.00	0.00	0.00	0.00
32768	**0.20**	**0.22**	**0.16**	**0.17**
16	0.01	0.03	0.01	0.00
	0.00	0.00	0.00	0.00

Table 9: LNC03 family data.



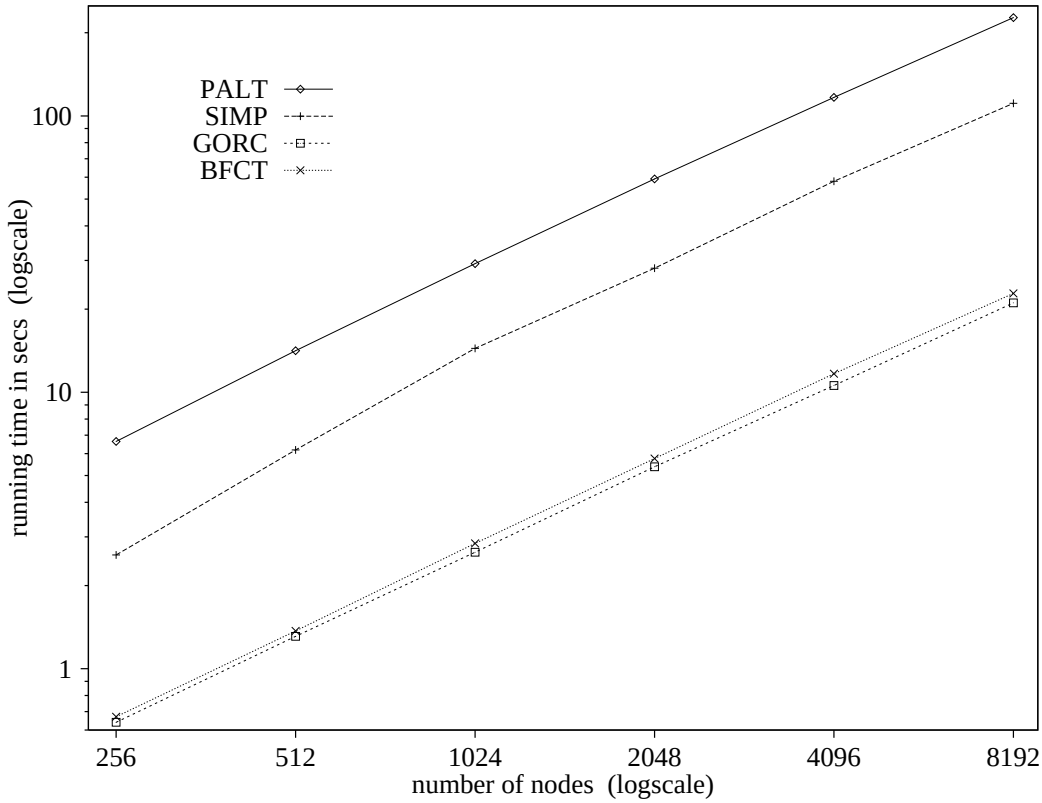
X*Y	BFCT	GORC	PALT	SIMP
512	**0.08**	**0.10**	**0.06**	**0.09**
16	0.01	0.01	0.02	0.01
	2.52	6.71	2.39	2.40
1024	**0.25**	**0.27**	**0.21**	**0.26**
16	0.01	0.04	0.03	0.02
	3.45	9.09	4.23	3.26
2048	**0.66**	**0.65**	**0.53**	**0.69**
16	0.03	0.02	0.07	0.02
	4.43	10.68	4.98	4.14
4096	**1.75**	**1.79**	**1.65**	**1.82**
16	0.12	0.07	0.29	0.13
	5.74	14.47	7.25	5.33
8192	**4.27**	**4.00**	**5.16**	**4.43**
16	0.13	0.27	0.40	0.11
	6.85	16.00	10.78	6.24
16384	**10.37**	**10.30**	**13.30**	**10.88**
16	0.63	0.36	1.60	0.56
	8.45	20.81	13.17	7.52
32768	**23.09**	**21.58**	**33.24**	**24.38**
16	2.14	1.16	1.75	0.92
	9.57	21.72	16.34	8.56

Table 10: LNC04 family data.



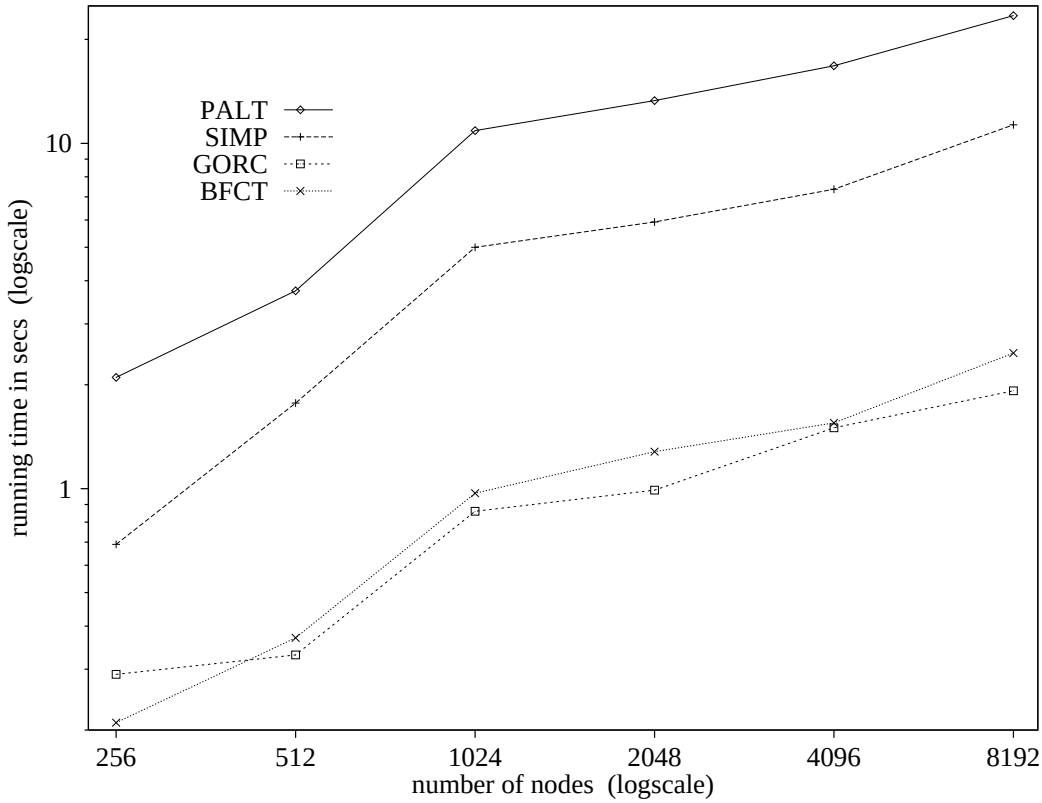
X*Y	BFCT	GORC	PALT	SIMP
512	**0.47**	**0.54**	**0.66**	**0.41**
16	0.02	0.04	0.03	0.03
	11.19	20.99	20.98	10.06
1024	**1.18**	**1.39**	**1.75**	**1.08**
16	0.11	0.12	0.17	0.10
	12.34	22.39	24.16	10.98
2048	**2.75**	**3.17**	**4.14**	**2.60**
16	0.19	0.22	0.50	0.24
	13.33	24.53	25.20	11.82
4096	**6.16**	**7.33**	**9.60**	**5.43**
16	0.37	0.35	0.58	0.39
	14.26	26.45	28.58	12.56
8192	**12.85**	**15.91**	**22.58**	**13.03**
16	0.54	0.97	1.40	0.69
	15.47	28.91	31.10	13.52
16384	**32.32**	**35.61**	**48.67**	**25.86**
16	0.47	2.28	4.43	1.66
	16.51	30.31	33.70	14.45
32768	**61.12**	**76.94**	**105.26**	**61.30**
16	4.35	5.02	9.40	2.36
	17.51	32.12	35.89	15.22

Table 11: LNC05 family data.



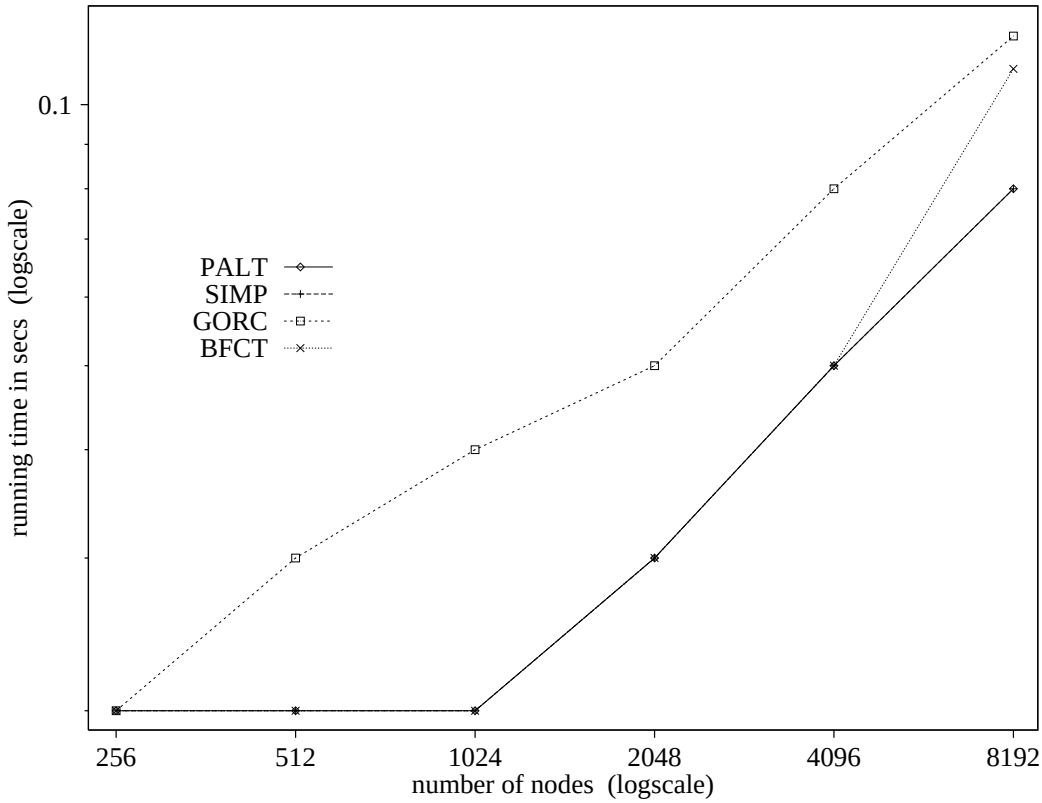
X*Y	BFCT	GORC	PALT	SIMP
256	**0.67**	**0.64**	**6.64**	**2.58**
32	0.03	0.02	0.24	0.13
	12.33	17.01	132.47	6.80
512	**1.37**	**1.31**	**14.15**	**6.19**
32	0.04	0.04	0.29	0.35
	12.60	17.66	139.96	6.93
1024	**2.84**	**2.64**	**29.20**	**14.42**
32	0.03	0.04	0.54	0.51
	12.85	17.57	144.27	7.08
2048	**5.76**	**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

Table 12: PNC01 family data.



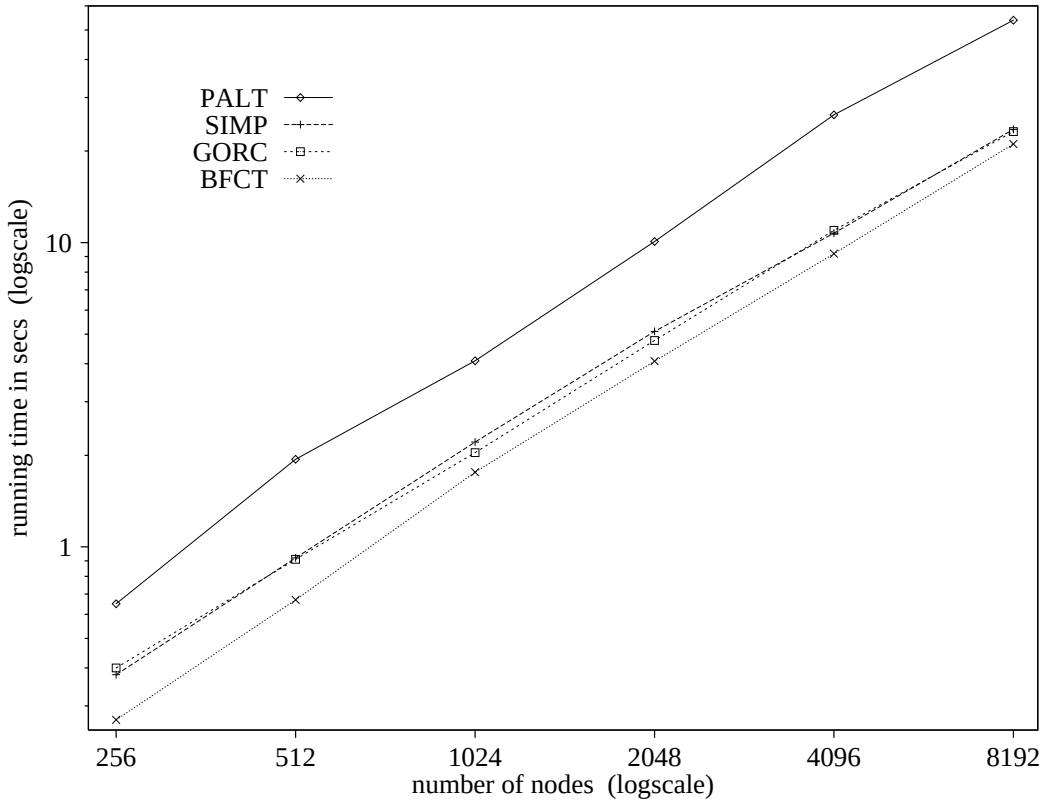
X*Y	BFCT	GORC	PALT	SIMP
256	**0.21**	**0.29**	**2.10**	**0.69**
32	0.19	0.18	2.33	0.64
	3.75	7.22	40.94	1.59
512	**0.37**	**0.33**	**3.74**	**1.77**
32	0.35	0.19	3.78	2.14
	3.34	3.93	36.58	1.73
1024	**0.97**	**0.86**	**10.89**	**5.00**
32	0.65	0.67	7.79	4.20
	4.35	5.35	53.25	2.45
2048	**1.28**	**0.99**	**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

Table 13: PNC02 family data.



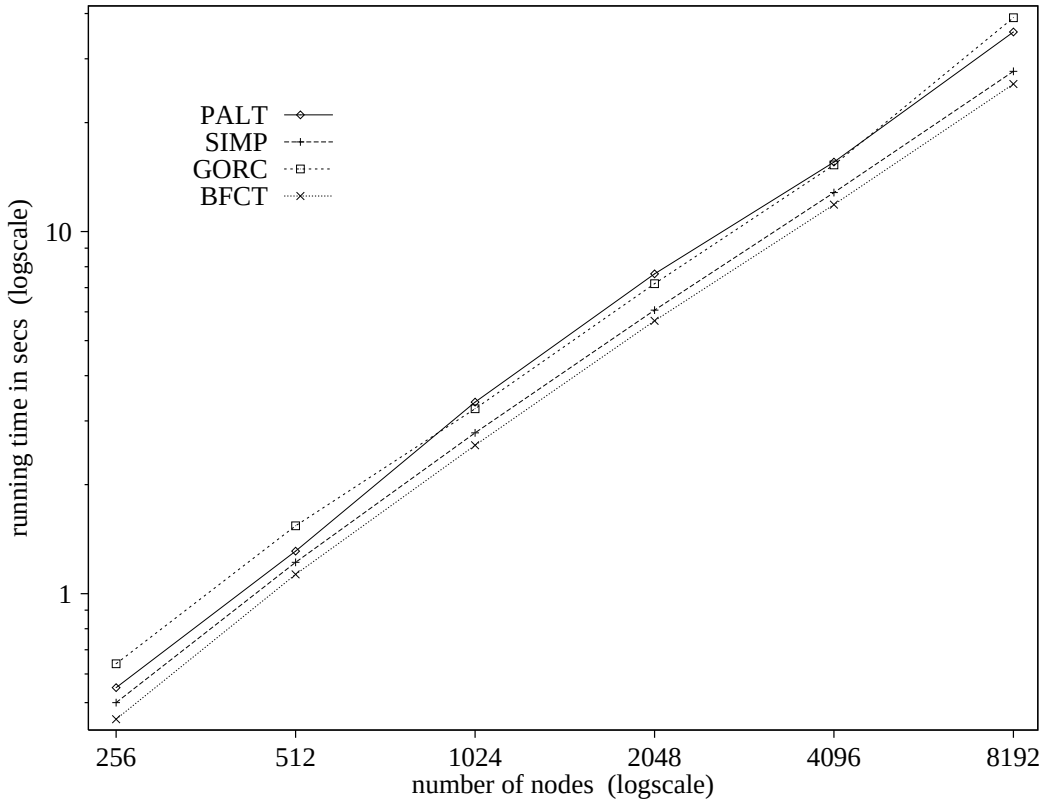
X*Y	BFCT	GORC	PALT	SIMP
256	**0.01**	**0.02**	**0.02**	**0.01**
32	0.01	0.00	0.01	0.01
	0.11	0.33	0.25	0.10
512	**0.01**	**0.03**	**0.02**	**0.02**
32	0.01	0.00	0.00	0.01
	0.03	0.19	0.10	0.03
1024	**0.02**	**0.04**	**0.02**	**0.02**
32	0.00	0.01	0.00	0.01
	0.02	0.12	0.04	0.01
2048	**0.03**	**0.05**	**0.03**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

Table 14: 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

Table 15: PNC04 family data.



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

Table 16: PNC05 family data.

11.5 Discussion

We have seen that adding one small negative cycle makes a problem simpler. This is because the cycle does not affect the computation much until it is discovered, at which time the computation terminates. When many small cycles are added, the problem becomes very simple because a cycle is discovered very quickly. Adding long negative cycles changes the problem structure and may make the problem simpler or harder, depending on the algorithm and on the problem class.

We eliminated several uncompetitive codes using common sense and the preliminary experiment. Further tests have shown that BFCT and GORC perform well on a wide variety of problems. The other two codes, PALT and SIMP, perform well in many tests but are not as robust.

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 a subset of shortest path problem 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 detailed description of these problem families, see [2].

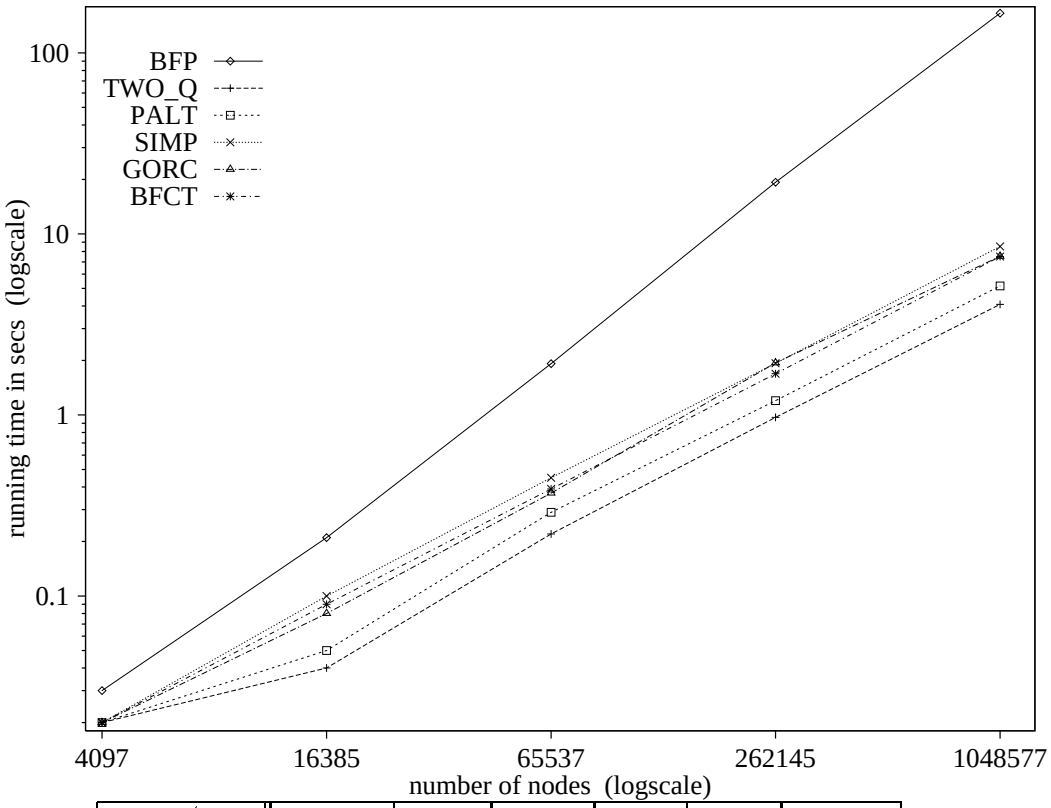
An interesting question is how much the subtree disassembly used by Tarjan's algorithm improves the Bellman–Ford–Moore algorithm, and how much this strategy improves Pallottino's algorithm. We also evaluate the optimized network simplex algorithm. Since the Goldberg–Radzik algorithm is one of the fastest and most robust in the presence of negative length arcs, 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].) We also use 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].

Results of the follow-up experiment appear in Tables 17 – 23.

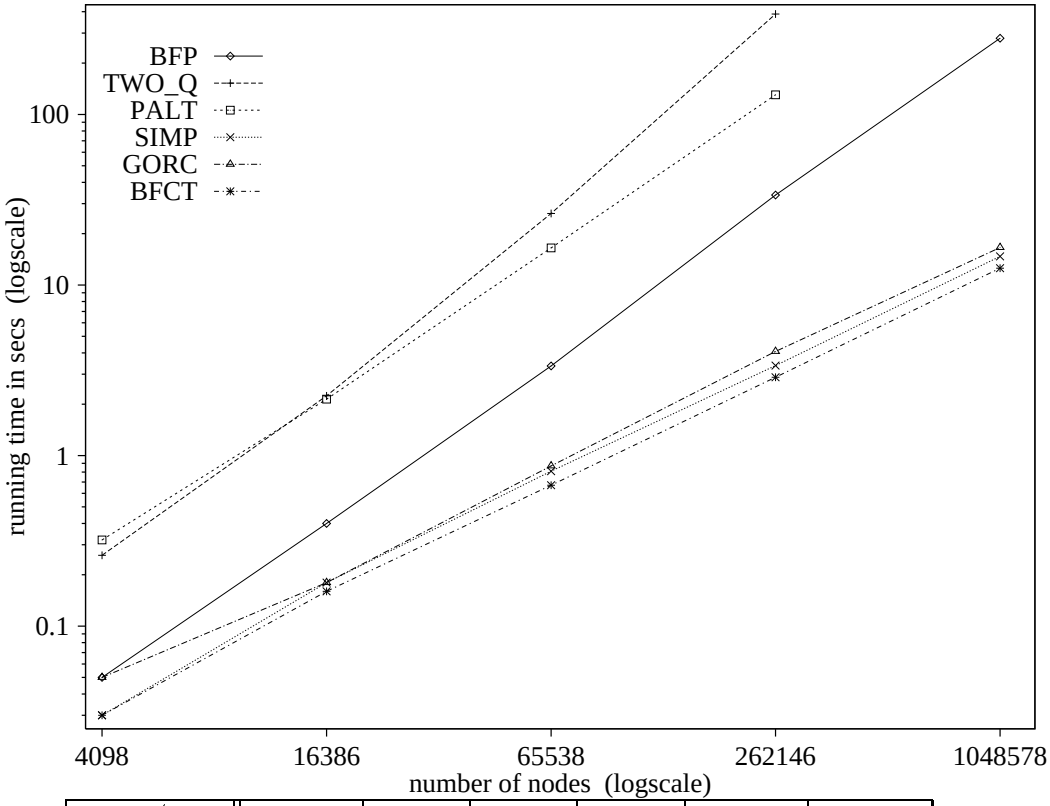
First we observe that BFCT outperforms BFP on all problem families, often by a very wide margin. This is due to the reduction in the number of scan operations due to subtree disassembly.

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



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

Table 17: Gird-SSquare family data



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		

Table 18: Gird-SSquare-S family data

reduce the number of scan operations and slightly increases the running time. On hard problems, subtree disassembly decreases the number of scan operations. The decrease can be relatively small, as on the Gird-SSquare-S family, or large, as on the Grid-PHard and Grid-NHard families. On hard problems, however, PALT is still not competitive with the best codes.

Next we compare GORC and BFCT. The former code performs better on Grid-PHard, Grid-NHard, and Acyc-Neg problem families. The latter performs better on Grid-Square-S, Rand-4, and Rand-1:4 families. The performance difference is relatively small except for the Acyc-Neg family, where GORC is much faster. This family is favorable for GORC, which on this family works similarly to the special-purpose algorithm for acyclic graphs. However, the performance of BFCT on this problem family shows that this algorithm is not completely robust.

Next we discuss SIMP. Performance of this code is reasonable but not spectacular. Although the number of SIMP pivots is usually less then the number of GORC or BFCT scans, pivots are more expensive. Sometimes SIMP is slower than GORC by a large margin; when SIMP is faster, it is by a small margin.

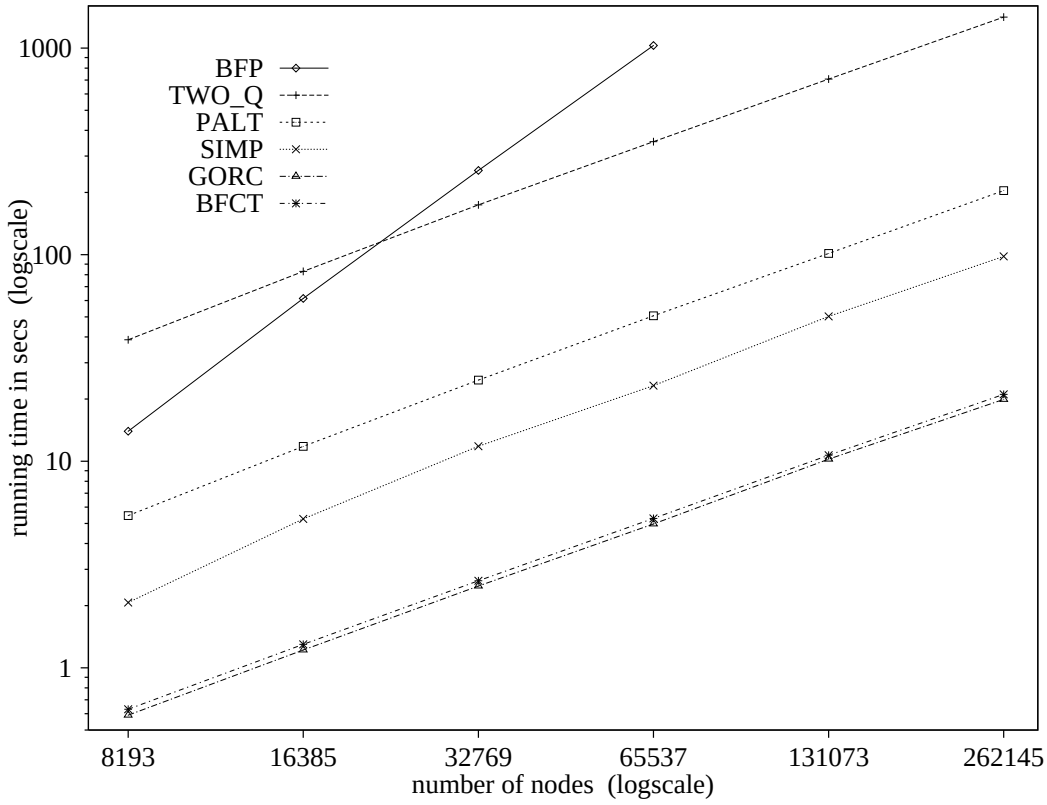
The data presented in this section shows that Tarjan's algorithm is almost as robust as the Goldberg–Radzik algorithm and should be a good choice for many shortest path applications.

13 Concluding Remarks

Among the algorithms for the negative cycle problem in our study, the Goldberg–Radzik algorithm and Tarjan's algorithm are the best. Both algorithms are very robust, although the former is somewhat more robust. The variant of Tarjan's algorithm implemented by BFCM and BFCTN performed similarly to Tarjan's algorithm.

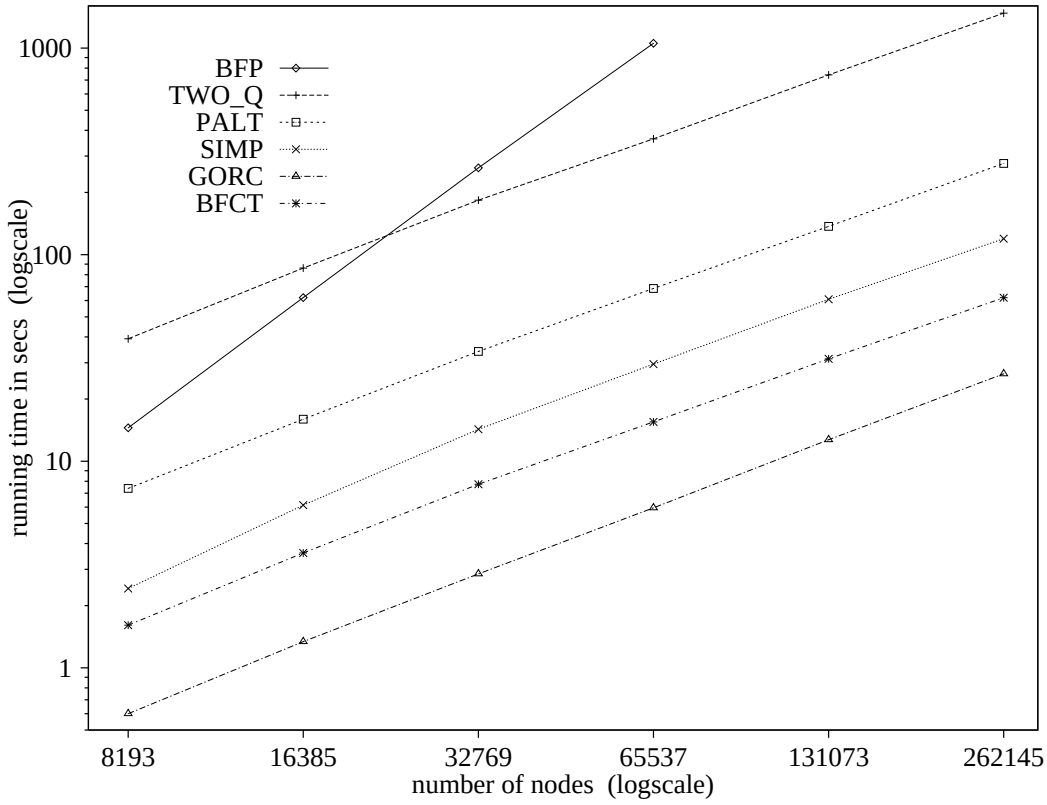
Because the Goldberg–Radzik algorithm performed so well on shortest path problems with negative length arcs in [2], we expected that it would perform well on many negative cycle problems. The good performance of Tarjan's algorithm was a surprise. This algorithm was motivated by adding immediate cycle detection to the Bellman–Ford–Moore algorithm so that the additional work can be amortized over the work done by labeling operations. Yet in practice the resulting algorithm is much faster than the Bellman–Ford–Moore algorithm.

Performance of BFCT and BFCM is extremely similar. Recall that BFCM implements the ideal the Bellman–Ford–Moore algorithm that at every step selects a labeled vertex with the minimum level in G_p to scan next. Tarjan's algorithm selects a minimal level labeled vertex to scan next (*i.e.*, a labeled vertex v such that the s–v path in G_p contains no other labeled vertices). Practical



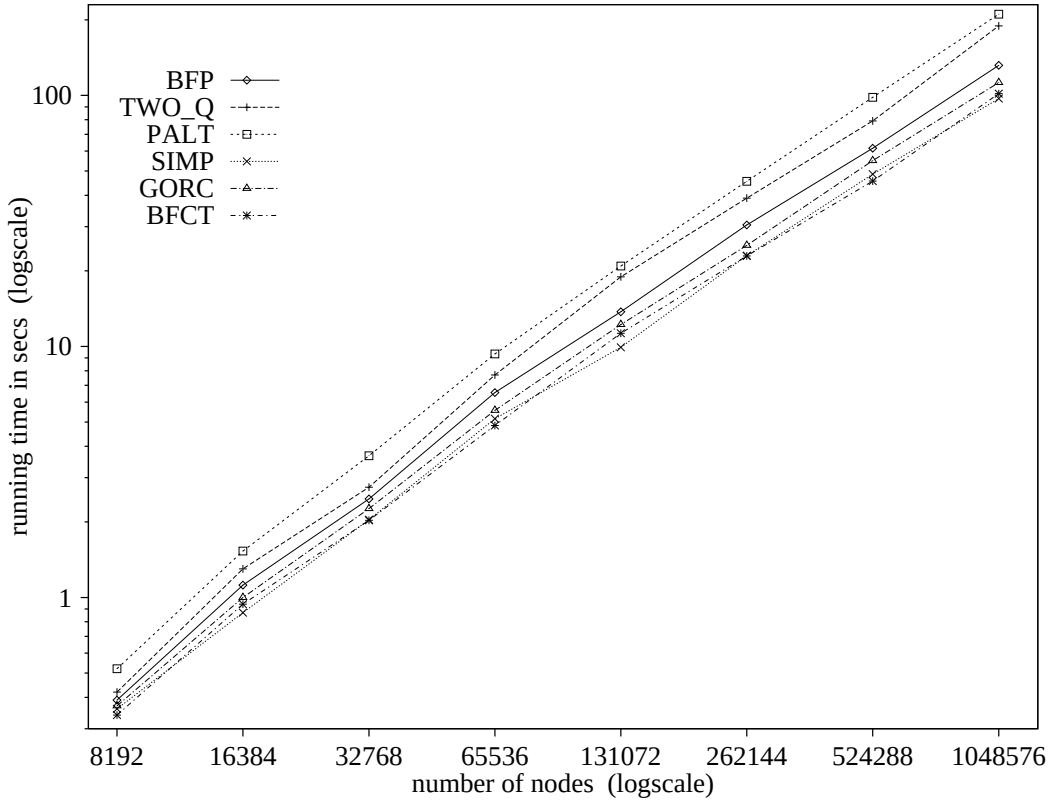
nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8193	**13.98**	**0.63**	**0.59**	**2.07**	**5.46**	**38.76**
63808	0.89	0.01	0.02	0.15	0.19	3.58
	390.13	12.44	16.90	6.73	132.21	1108.89
16385	**61.38**	**1.30**	**1.22**	**5.26**	**11.80**	**83.04**
129344	1.07	0.04	0.03	0.20	0.13	2.79
	799.87	12.53	17.98	6.94	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

Table 19: Gird-PHard family data



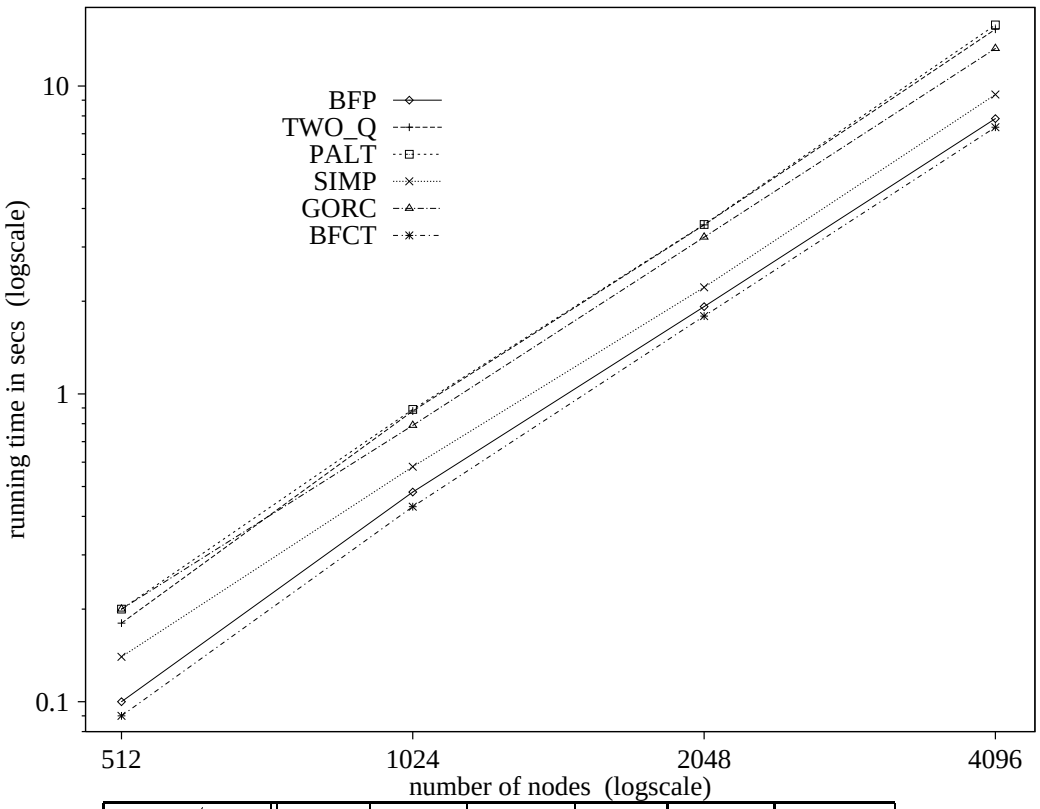
nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8193	**14.52**	**1.61**	**0.60**	**2.42**	**7.39**	**39.18**
63808	0.89	0.05	0.03	0.32	0.31	3.14
	392.09	27.01	17.88	8.68	166.24	1143.77
16385	**61.95**	**3.60**	**1.34**	**6.13**	**15.99**	**86.09**
129344	0.74	0.11	0.04	0.25	0.41	1.89
	803.31	29.11	19.89	8.94	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

Table 20: Gird-NHard family data



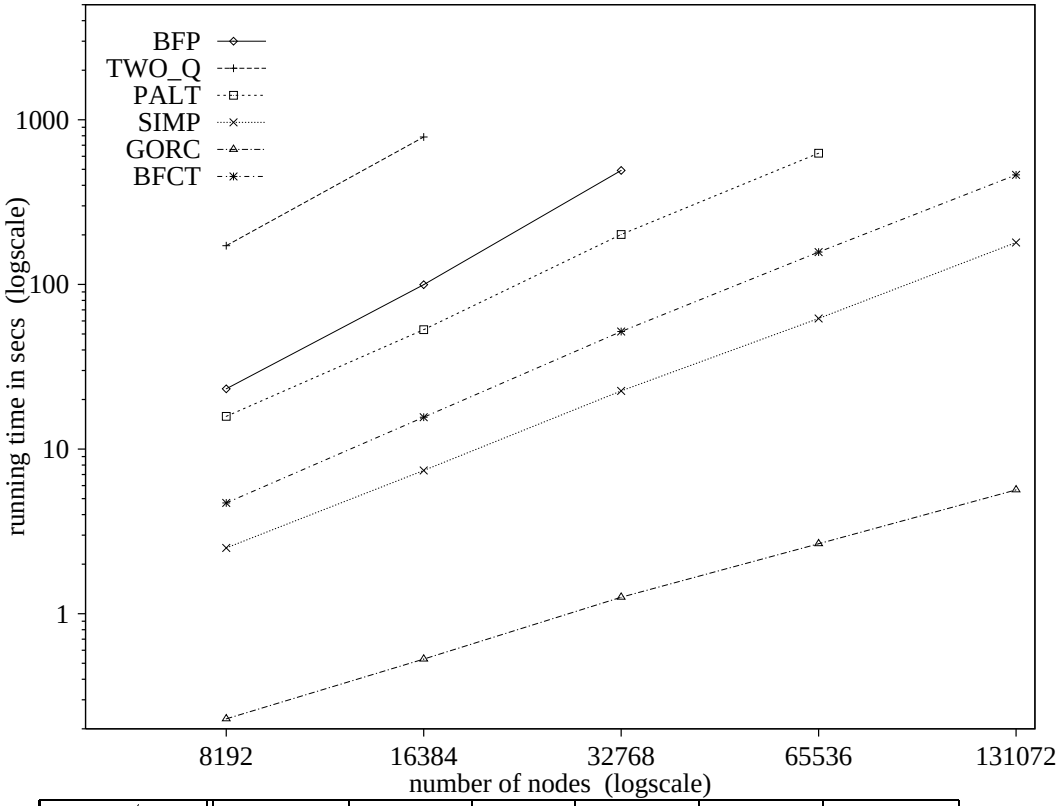
nodes/arcs	BFP	BFCT	GORC	SIMP	PALT	TWO-Q
8192	**0.39**	**0.34**	**0.37**	**0.36**	**0.52**	**0.42**
32768	0.03	0.03	0.03	0.02	0.07	0.07
	12.23	7.75	14.58	6.90	15.37	15.90
16384	**1.12**	**0.94**	**1.00**	**0.87**	**1.53**	**1.30**
65536	0.06	0.04	0.08	0.09	0.17	0.25
	13.45	8.61	16.19	7.57	17.99	18.83
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

Table 21: Rand-4 family data



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

Table 22: Rand-1:4 family data



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		

Table 23: Acyc-Neg family data

performance of the two strategies appears nearly identical. On every problem we studied, the numbers of scans performed by the two codes were very close and neither was dominant. The running times were close as well. Usually BFCT was faster than BFCM because BFCT does not move vertices in the queue and because its low level data structures are simpler. The time difference, however, was under 20% in most cases.

The fact that BFCM, the ideal $O(nm)$ negative cycle algorithm from the point of view of analysis, performs well in practice shows that the analysis is realistic. In hindsight, we should have expected BFCM to perform well, and because Tarjan's algorithm is so similar to BFCM, good performance of Tarjan's algorithm should not have been a surprise.

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.

Acknowledgments

We would like to thank Bob Tarjan for stimulating discussions, for simplified proofs of Theorem 4.5 and related lemmas, and for comments on a draft of this paper. We also would like to thank Harold Stone for comments that improved our presentation.

References

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

[2] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525, 1994. To appear in Math. Prog.

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

[4] G. B. Dantzig. Application of the Simplex Method to a Transportation Problem. In T. C. Koopmans, editor, *Activity Analysis and Production and Allocation*, pages 359–373. Wiley, New York, 1951.

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

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

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

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

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

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

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

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

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

[14] M. Klein. A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems. *Management Science*, 14:205–220, 1967.

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

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

[17] J-F. Mondou, T.G. Crainic, and S. Nguyen. Shortest Path Algorithms: A Computational Study with the C Progremming Language. *Computers and Oper. Res.*, 18:767–786, 1991.

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

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

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

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

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