JOURNAL OF ALGORITHMS **22,** 1–29 (1997)
ARTICLE NO. AL950805

An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm

Andrew V. Goldberg*

NEC Research Institute, Princeton, New Jersey 08540

Received October 28, 1992

The scaling push-relabel method is an important theoretical development in the area of minimum-cost flow algorithms. We study practical implementations of this method. We are especially interested in heuristics which improve real-life performance of the method. Our implementation works very well over a wide range of problem classes. Some heuristics we develop may apply to other network algorithms. Our experimental work on the minimum-cost flow problem motivated theoretical work on related problems. © 1997 Academic Press

1. INTRODUCTION

Significant theoretical progress has been made recently in the area of minimum-cost flow algorithms (see [1, 22]). Practical performance evaluation of some of these algorithms is just starting [26]. Detailed studies of somewhat older methods include, for example, [4, 6–8, 16, 25, 31, 32].

In this paper we continue our work [21] on implementing one of the recent methods, the *successive approximation push-relabel method* of Goldberg and Tarjan [18, 24]. This method combines and extends the ideas of cost-scaling due to Röck [35] and Bland and Jensen [6], the push-relabel maximum flow method of Goldberg and Tarjan [17, 23], and the relaxation method of Bertsekas [3]. This new method looks promising for two reasons. First, the inner loop of the method is based on the push-relabel algorithm for the maximum flow problem. The maximum flow push-relabel method

* Most of this work was done while the author was at the Stanford University Computer Science Department and supported in part by ONR Young Investigator Award N00014-91-J-1855, NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT & T and DEC, Stanford University Office of Technology Licensing, and a grant from the Powell Foundation.

0196-6774/97 $25.00

has been shown superior to previous codes in several experimental studies, e.g., [2, 10−12, 23]. Second, the successive approximation technique used in the method requires fewer iterations of the inner loop compared to the closely related cost-scaling.

Performance of the previous implementation of the method [5, 21] is mixed: on some problem classes these implementations work well, while on other classes, not so well. The implementation described in this paper works better than our previous code SPUR [21] on most problems; in particular it does quite well on the classes where SPUR performed relatively poorly.

This is due to heuristics that improve the practical performance of the method (but not its theoretical worst-case bound). Many ideas for such heuristics were proposed in [5, 21, 24] and some have been shown to be effective. Our current implementation successfully uses two additional heuristics to update prices during the computation. These heuristics are closely related to the scaling shortest path algorithm [19], which has been motivated by our experimental work.

We compare our implementation to SPUR, to the established network simplex codes NETFLO [29] and RNET [25], and to the relaxation code RELAX [4]. The comparison is done on eight problem families produced by three generators. The problems in different families have different characteristics, and the behavior of the codes varies from one class to another. Our code is asymptotically fastest on most of the problem families, and as fast asymptotically as the fastest competing code on the remaining families. On the biggest problems in each family, our code is often better by orders of magnitude, and never loses by more than a small constant factor.

This paper is organized as follows. Section 2 gives the relevant definitions and outlines the method. Section 3 discusses heuristics that are used in our implementation. Section 4 describes our experimental setup. Section 5 gives the experimental results. Section 6 gives details of our code's performance on several problem instances and discusses this data. In Section 7, we give our conclusions and suggest directions for further research.

2. BACKGROUND

In this section we briefly review the push-relabel method. For more detail, see [24].

2.1. *Definitions and Notation.* Our implementation works with the *capacitated transshipment* version of the minimum-cost flow problem defined as follows. A *network* is a directed graph $G = (V, E)$ with a real-valued *capacity* $u(a)$ and a real-valued *cost* $c(a)$ associated with each arc a, and a real-valued *demand* $d(v)$ associated with each node v.[1] For the rest of this paper, we assume that all costs, capacities, and demands are integers, as is the case in our implementation. We assume that G is *symmetric*, i.e., $a \in E$ implies that the reverse arc $a^R \in E$. (We add the reverse arcs during parsing.) The cost function satisfies $c(a) = -c(a^R)$ for each $a \in E$ and the total demand is zero, i.e., $\sum_V d(v) = 0$. We denote the size of V by n, the size of E by m, and the biggest input cost by C.

A *pseudoflow* is a function $f\colon E \to \mathbf{R}$ satisfying the following *capacity* and *antisymmetry* constraints for each $a \in E$: $f(a) \le u(a)$, $f(a) = -f(a^R)$.

For a pseudoflow f and a node v, the *excess flow into* v, $e_f(v)$, is defined by $e_f(v) = \sum_{(u,v)\in E} f(u,v) - d(v)$. A node v with $e_f(v) > 0$ is called *active*. Note that $\sum_{v\in V} e_f(v) = 0$.

A (*feasible*) *flow* is a pseudoflow f such that, for each node v, the demand at v is met, i.e., $e_f(v) = 0$. Observe that a pseudoflow f is a flow if and only if there are not active nodes. The cost of a pseudoflow f is given by $cost(f) = \frac{1}{2}\sum_{n\in E} c(a)f(a)$. The minimum-cost flow problem is to find a flow of minimum cost (*optimal flow*).

For a given pseudoflow f, the *residual capacity* of an arc a is $u_f(a) = u(a) - f(a)$. An arc a is *saturated* if $u_f(v,w) = 0$, and *residual* if $u_f(a) > 0$. The *residual graph* $G_f = (V, E_f)$ is the graph induced by the residual arcs.

A *price function* is a function $p\colon V \to \mathbf{R}$. For a given price function p, the *reduced cost* of an arc (v,w) is $c_p(v,w) = c(v,w) + p(v) - p(w)$.

For a given f and p, an arc a is *admissible* if it is a residual arc of negative reduced cost. The *admissible graph* $G_A = (V, E_A)$ is the graph induced by the admissible arcs. A flow f is optimal if and only if there exists a price function p such that no arc is admissible with respect to f and p [14].

For a constant $\epsilon \ge 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function p* if, for every residual arc a, we have $c_p(a) \ge -\epsilon$. A pseudoflow f is ϵ-optimal if f is ϵ-optimal with respect to some price function p. If the arc costs are integers and $\epsilon < 1/n$, any ϵ-optimal flow is optimal [3].

2.2. *The Method.* First we give a high-level description of the successive approximation algorithm (see Fig. 1). The algorithm maintains a flow f and a price function p, such that f is ϵ-optimal with respect to p. The algorithm starts with $\epsilon = C$, with $p(v) = 0$ for all $v \in V$, and with any feasible flow. A feasible flow can be found using one invocation of any

[1] Sometimes we refer to an arc a by its endpoints, e.g., (v,w). This is ambiguous if there are several arcs from v to w. An alternative is to refer to v as the tail of a and to w as the head of a, which is precise but inconvenient.

```
procedure $min\text{-}cost(V, E, u, c)$;
    [initialization]
    $\epsilon \leftarrow C$;
    $\forall v, \quad p(v) \leftarrow 0$;
    if $\exists$ a flow then $f \leftarrow$ a flow else return(null);
    [loop]
    while $\epsilon \geq 1/n$ do
        $(\epsilon, f, p) \leftarrow refine(\epsilon, f, p)$;
    return($f$);
end.
```

FIG. 1. The successive approximation algorithm.

maximum flow algorithm. Any flow is C-optimal with respect to the zero price function. The main loop of the algorithm repeatedly reduces ϵ by a constant factor α, the choice of which is discussed later. When $\epsilon < 1/n$, the algorithm terminates. The algorithm takes $\lceil \log_\alpha(nC) \rceil$ iterations.

Reducing ϵ is the task of the subroutine *refine*. The input to *refine* is ϵ, f, and p such that f is ϵ-optimal with respect to p. The output from *refine* is ϵ reduced by a factor of α, a new f, and a new p such that f is ϵ-optimal with respect to p.

The generic *refine* subroutine (described in Fig. 2) begins by decreasing the value of ϵ and saturating every arc with negative reduced cost. This action converts the flow f into an ϵ-optimal pseudoflow (indeed, into a 0-optimal pseudoflow). Then the subroutine converts the ϵ-optimal pseudoflow into an ϵ-optimal flow by applying a sequence of *push* and *relabel* operations (see Fig. 3), each of which preserves ϵ-optimality. The generic algorithm does not specify the order in which these operations are applied.

A *push* operation applied to a residual arc (v, w) of negative reduced cost whose tail node v is active. It consists of pushing $\delta =$

```
procedure $refine(\epsilon, f, p)$;
    [initialization]
    $\epsilon \leftarrow \epsilon/\alpha$;
    $\forall(v, w) \in E$ do if $c_p(v, w) < 0$ then $f(v, w) \leftarrow u(v, w)$;
    [loop]
    while $\exists$ a push or a relabel operation that applies do
        select such an operation and apply it;
    return($\epsilon, f, p$);
end.
```

FIG. 2. The generic *refine* subroutine.

push(v, w).
Applicability: v is active, $u_f(v,w) > 0$, **and** $c_p(v,w) < 0$.
Action: send $\delta = \min(e_f(v), u_f(v,w))$ units of flow from v to w.

relabel(v).
Applicability: v is active **and** $\forall w \in V\ u_f(v,w) > 0 \Rightarrow c_p(v,w) \geq 0$.
Action: replace $p(v)$ by $\max_{(v,w)\in E_f}\{p(w) - c(v,w) - \epsilon\}$.

FIG. 3. The *push* and *relabel* operations described in the figure are somewhat restrictive. Some heuristics use different versions of these operations as discussed later.

$\min\{e_f(v), u_f(v,w)\}$ units of flow from v to w, thereby decreasing $e_f(v)$ and $f(w,v)$ by δ and increasing $e_f(w)$ and $f(v,w)$ by δ.

A *relabel* operation applies to an active node v that has no exiting residual arcs with negative reduced cost. It consists of decreasing $p(v)$ to the smallest value allowed by the ϵ-optimality constraints, namely $\max_{(v,w)\in E_f}\{p(w) - c(v,w) - \epsilon\}$. (Alternatively, $p(v)$ can be decreased by ϵ.)

The generic implementation of the algorithm needs one additional data structure, a set S containing all active nodes. Initially S contains all nodes whose excess becomes positive during the initialization step of refine. Updating S takes only $O(1)$ time per push or relabel operation. (Such an operation requires possibly deleting one node from S and adding one node to S.)

At a low level, the push and relabel operations are combined in the *discharge* operation, described in Fig. 4. A discharge operation applies push and relabel operations to an active node until the node becomes inactive, i.e., its excess drops to zero. We assume the adjacency list representation of the graph and maintain a current arc pointer for every node v. The current arc of a node is set to its first arc initially and after each relabeling of the node. The discharge(v) operation attempts to push flow along the current arc of v. If the current arc is not eligible for pushing, discharge

Discharge.
Applicability: v is active.
Action: apply *push/relabel* operations to v until v becomes inactive.

FIG. 4. The *discharge* operation.

advances the current arc pointer to the next arc on the edge list of v unless the current arc is the last arc on the list, in which case v is relabeled.

There remains the issue of the order in which to discharge active nodes. We implement the *first-in-first-out* (FIFO) algorithm, which maintains the set of active nodes as a queue, repeatedly discharging the front node on the queue and adding newly active nodes to the rear of the queue.

The worst-case theoretical bounds on the number of basic operations invoked during an execution of refine are as follows (see [24]):

- The number of relabel operations is $O(n^2)$.
- The number of push operations is $O(n^2m)$ in any implementation of the generic method.

A dynamic tree data structure can be used to do several push operations at once, as described in [24]. Our experience suggests that in practice the relabel operations are the bottleneck, an the dynamic trees are not likely to help. We did not experiment with the dynamic tree version of the algorithm.

3. HEURISTIC IMPROVEMENTS

In this section we discuss heuristics used in our implementation. These heuristics improve the typical running time of the algorithm and do not increase the asymptotic worst-case time bound.

3.1. *Price Updates*. The push−relabel method modifies prices locally, one node at a time. *Price update* heuristics modify prices in a more global way. In the maximum flow context, price updates, implemented using breadth-first search, have been shown to significantly improve practical performance of the push−relabel method. This heuristic does not help much on some problem classes, but results in asymptotic performance improvement on other classes.

The idea of price updates in the minimum cost flow context had been introduced in [24]. These updates, however, need to be done in such a way that the price function after an update is ‘‘better’’ than the price function before the update. In particular, the number of push and relabel operations should decrease even if the updates are performed infrequently. Our implementation is the first one to achieve this. The implementation uses the techniques introduced in [19, 21].

The price update heuristic is based on the *set-relabel* operation, which is defined as follows. Let S be a set of nodes such that S contains all nodes with negative excess, and $\bar{S}$, the complement of S, contains at least one node with positive excess. Suppose that no admissible arc goes from a node in $\overline{S}$ to a node in S. The set-relabel operation reduces the price of every node in $\overline{S}$ by ϵ.

It can be shown that the set-relabel operation satisfies the following conditions (see [19]).

(1) ϵ-optimality is preserved,

(2) the admissible graph remains acyclic,

(3) prices are monotonically decreasing, and

(4) prices of nodes with negative excess remain unchanged.

These facts imply the $O(n^2)$ bound on the number of relabels per refine; in fact they imply that each node participates in $O(n)$ relabels and set-relabels per refine.

The set-relabel operation can be applied in the following way. Initially, the set S contains a set of all nodes with negative excess. At each iteration, the set S is extended to include all nodes from which a node in S is reachable in the admissible graph. If all nodes with positive excess are in S, the computation terminates. If not, set-relabel is applied to S and the next iteration begins. This computation is implemented using buckets in a way similar to that of Dial's implementation [13] of Dijkstra's shortest path algorithm, as described in [19].

Our implementation of the price update heuristic maintains an array B of buckets. Each node is in at most one bucket. Define $B(v)$ to be index of the bucket containing v, or ∞ if v is no bucket. Initially all buckets except bucket zero are empty, bucket zero contains all nodes with negative excess, and current bucket index i is set to zero. At every step, a node is removed from the current bucket and scanned, or if the current bucket is empty the current bucket index is incremented. The scan of a node v examines all arcs (v, w), and if w has not been scanned yet and $k = \lfloor c_p(v, w)/\epsilon \rfloor + 1$ is less than $B(v)$, then v is removed from its bucket (if any) and inserted into bucket k. Immediately after v has been scanned, its label $l(v)$ is set to i and v is added to the set S of scanned nodes. The process is continued until all nodes with positive excess have been scanned. Then the prices of all scanned nodes v are reduced by $\epsilon l(v)$.

Since during refine node prices change by $O(n\epsilon)$ [24], $O(n\epsilon)$ buckets are sufficient for the price update implementation: the nodes in the buckets with high indices are never examined, so there is no need to put them in the buckets.

It is easy to see that this implementation of price updates is equivalent to the procedure described above, and that it works in time linear in the size of subgraph examined by it. If $\Omega(n)$ relabels take place before each price refine, the total cost of the latter oeprations during an execution of the algorithm is $O(nm\log(nC))$.

The ideal frequency of performing the global price updates in implementation and problem dependent. A good starting point is to perform the updates after every n relabels, and then experiment. Our implementation uses a slightly different strategy of using a linear combination of the number of relabels and the number of passes over the node queue (instead of just the number of relabels) to trigger global price updates. More precisely, a global update is performed when $\rho r + \pi q > n$, where ρ and π are constants and r and q are the numbers of relabels and queue passes from the previous price update (or from the beginning of refine if no price updates were performed). This price update strategy makes price updates more frequent when the number of active nodes is small (toward the end of refine).

3.2. *Price Refinement.* As suggested in [24], refine may produce a solution which is not only ϵ-optimal, but also (ϵ/α)-optimal. In fact, refine may produce an optimal flow even for $\epsilon > 1/n$. Our implementation uses the *price refine heuristic*. This heuristic decreases ϵ and does not change the flow f while modifying p in an attempt to find p such that f is ϵ-optimal with respect to p. The heuristic is applied every time ϵ is decreased by the algorithm, and also at the beginning of the algorithm if the zero flow in the input network is feasible.

Our implementation of the price refine heuristic is similar to a version of the scaling shortest path algorithm [19]. The implementation uses the cut-relabel operation implicitly. The description below provides implementation details.

The price refine heuristic starts with a flow f which is $(\alpha\epsilon)$-optimal with respect to a price function p and attempts to modify p so that f is ϵ-optimal with respect to the modified p. The implementation works in passes and maintains an array of buckets B. At the beginning of each pass B is empty. The pass starts by topologically sorting the current admissible graph. If the graph contains a cycle, the computation terminates (failing to produce the desired price function).

Otherwise the admissible graph is acyclic, and we compute approximate distances d' with respect to c_p in the admissible graph from the zero in-degree nodes. The approximate distances d' are measured in units of ϵ. The distances are nonpositive integers. Initially, $d' = 0$ for all v. We scan nodes in topological order. A node v is scanned by examining its arcs (v,w). If $d'(w) > d'(v) + \lceil c_p(v,w)/\epsilon \rceil$, then we set $d'(w) = d'(v) + \lceil c_p(v,w)/\epsilon \rceil$. After the distances are computed, each node v is placed into the bucket $-d'(v)$.

Next we go through the buckets in the decreasing order, removing nodes from the current bucket and scanning them until the bucket is empty. A node v is scanned by examining its arcs (v, w). If w has not been scanned, we may update $d'(w)$ (and more w to the bucket $-d'(w)$). If $c_p(v, w) < 0$ and $d'(w) > d'(v)$, we set $d'(w) = d'(v)$. If $c_p(v, w) \geq 0$ and $d'(w) > d'(v) + \lceil c_p(v, w)/\epsilon \rceil$, we set $d'(w) = d'(v) + \lceil c_p(v, w)/\epsilon \rceil$.

At the end of a pass, the price of every node v is updated as follows: $p(v) = p(v) + d'(v)$. If the current solution is ϵ-optimal, the heuristic terminates. Otherwise, a new pass begins.

Clearly a pass takes linear time. Since a price of at least one node decreases by at least ϵ during a pass, the number of passes is $O(n)$. Thus, one price refine computation takes $O(nm)$ time. This bound does not exceed the bound for refine, so the asymptotic running time of the algorithm does not increase by more than a constant factor.

A price refine computation fails if an admissible cycle is created during the computation and succeeds otherwise. In the latter case ϵ is decreased again or, if ϵ is small enough, the algorithm terminates. In the former case, one can either apply refine to decrease ϵ or contract admissible cycles and finish the shortest paths computation, then undo the contractions and apply refine. In our experience, the former alternative works better.

Our way of implementing the price refine heuristic has several advantages. One advantage is that the work done during price refinement is not lost even in the case of failure: the number of push and relabel operations in the subsequent refine usually decreases enough to pay for the price refinement. A possible reason for this is that the admissible graph is acyclic after a refine, but has cycles after a price refine fails. These cycles are saturated at the beginning of the subsequent execution of refine. The second advantage is that if an optimal flow is computed at some point of the algorithm, refine is never again applied and the computation is completed by using the scaling shortest paths algorithm. This saves time because price refine is faster than refine. In practice, several last iterations of the algorithm do not apply refine.

3.3. *Arc Fixing*. The arc fixing heuristic involves “deleting” some arcs from the graph, thus reducing the number of times the algorithm examines an arc. The version of this heuristic that we use is a modification of that used in [21].

The theoretical justification of this technique is as follows [24, 36]: if the current flow is ϵ-optimal and the absolute value of an arc cost exceeds $2n\epsilon$, the push−relabel method will not change the flow on this arc. Thus the arc does not need to be examined until the end of the algorithm. Arc fixing can be done after every execution of refine.

In the dual context, arc fixing corresponds to edge contraction. Fujishige *et al.* [15] propose contracting edges earlier than the theory suggests. We use this idea in the primal context and call the resulting heuristic *speculative arc fixing*.

This heuristic fixes all arcs with the absolute value of reduced cost greater than β, where β is a parameter that depends on the input. Fixed arcs are examined by refine very infrequently. The arcs with the current reduced cost absolute values of β or below are unfixed. Also, fixed arcs violating complimentary slackness are unfixed and saturated. In this case, we say that a *fix-in* occurred.

A proper choice of β is important. The smaller β is, the fewer arcs refine and price refine have to deal with, so they run faster. If β is too small, however, fix-ins happen often and refine and takes more time.

In practice, β is set below the theoretically justified value of $2n\epsilon$. This has bad side effects which come up during refine (although extremely rarely for a proper choice of β). The first side effect is that after an arc is fixed, the problem may become infeasible, i.e., there may be no feasible flow consistent with the fixed arc flow value. The second side effect is that during a price update, more buckets may be needed than the theory suggests. Implementations of refine and price update need to detect such situations and unfix some of the fixed arcs.

3.4. *Push Lookahead.* The following scenario seems to be common in practice. Consider two nodes, v and w, such that $e_f(v) > 0$ and $e_f(w) \geq 0$. Suppose v pushes flow to w, and the first time flow is pushed from w afterward, this flow is pushed back to v. Observe that this can happen only if w does not have any outgoing admissible arcs just before the flow is pushed into it. Intuitively, the work done during the two pushes is wasted.

Such a situation can be avoided using the *lookahead* heuristic, introduced in [21]: before pushing flow to a node w, check whether w has an outgoing admissible arc or whether $e_f(w) < 0$. If this is so, do the pushing; if not, relabel w. A technical difficulty is that w may be inactive and the relabel operation, as described in Section 2.2, may not apply. In this case, either a node with negative excess is reachable from w or no such node is reachable. In the former case, the method remains correct if relabel is applied to w by the same argument as that presented in [24] for active nodes. In the latter case, one can show that relabel still can be applied to w except when w has no outgoing residual arcs. If w has no outgoing arcs, the price of w can be decreased by an arbitrary amount without violating ϵ-optimality. For example, we can decrease the price of w by ϵ. Alternatively, we can decrease the price by a large enough amount so that all arcs adjacent to w will be fixed.

We use the lookahead heuristic in our implementation. This heuristic reduces the number of pushes significantly; in many cases, the number of pushes falls below the number of relabels. See [21] for more detail.

4. EXPERIMENTAL SETUP

In this section we describe minimum-cost flow codes and problem generators used in our study.

4.1. *Minimum-Cost Flow Codes*. Our code, CS (cost scaling), is an implementation of the successive approximation push–relabel method using FIFO ordering of node discharge oeprations and the following heuristics: price update, price refine, speculative arc fixing, and push lookahead. We used CS version 1.0 with default parameter values, which are as follows. The scale factor α is 12. The threshold parameter for speculative arc fixing, β, is $0.225 \cdot n^{3/4}$. The price update frequency parameters, ρ and π, are $2/3$ and $80/3$, respectively. CS is written in C and compiled using SUN C compiler with the optimization option "$-$O4."

We compare our code against SPUR [21] and three established codes: NETFLO of Kennington and Helgason [29], RNET of Grigoriadis [25], and RELAX of Bertsekas and Tseng [4]. SPUR implements the same basic method as CS, but SPUR does not use the price update and price refine heuristics. SPUR uses the early termination detection heuristic not used by CS. SPUR is written in C and compiled the same way as CS. We use RNET version 3.61, December 1979 and RELAX version RELAX-III, April 1990. SPUR, NETFLO, and RELAX has no user-defined parameters. We used default parameter values for RNET: $frq = 1$, $p_0 = 1$, and $p_1 = 1.5$. NETFLO, RNET, and RELAX are written in Fortran and compiled using SUN Fortran compiler with the optimization option "$-$O."

Our experiments were conducted on a SUN Sparc-2 workstations with 40 MHz CPU. The running times we measure are user execution times (measured with $1/60$ seconds precision). The input–output time is not included.

4.2. *Problem Families*. We evaluate the codes on eight network families produced by three generators, all obtained from DIMACS: GOTO, GRIDGRAPH, and NETGEN. These problem families are GOTO-8, GOTO-16, GOTO-I, GRID-SQUARE, GRID-LONG, GRID-WIDE, NETGEN-HI, and NETGEN-LO; the generators and parameters used for each family are described below. These problem families has been used by participants of the First DIMACS Implementation Challenge [26]. The data from [26] can be compared to ours. Different generators produce problems with very different structure. The results of the Challenge show that different algorithms often exhibit very different relative behavior on these families. A code designed or tuned specifically for one of these families is likely to perform poorly on some of the other families.

For every family we give average running times of the five codes on instances whose size starts at 2^8 nodes and doubles at every step. (If a generator cannot produce a problem of exactly the desired size, we use a close size that the generator can produce.) The upper limit on the number of nodes in networks we consider is 2^{16} except for the GOTO-I family, where the limit is 2^{13}. We take averages over four instances.

All problems we consider can be solved by CS in less than an hour. Some of the other codes perform very poorly on some of the problem classes and would have taken days or weeks on large problems in these classes. We did not run such codes on these problems.

The GOTO generator, due to Goldberg [21], produces problems which seem relatively hard for combinatorial minimum-cost flow algorithms (the results of [34] suggest that this is not the case for interior-point algorithms). The generator produces networks where the structure of the underlying graph is determined by the number of arcs and nodes specified in the input. The skeleton of the graph is a rectangular grid with vertical wrap around. The length of the grid is roughly equal to its width squared. There is a source connected to nodes on the left side of the grid, and the right side of the grid is connected to the sink. A grid node has several vertical arcs going to successive rows and several horizontal arcs going to successive columns. Arc costs and capacities are selected uniformly at random from intervals which depend on the corresponding input parameters and on the arc type (vertical or horizontal). Vertical arcs have small expected costs and large capacities. Horizontal arcs have large expected costs, and expected capacity of a horizontal arc decreases exponentially with the number of columns the arc goes across. The supply at the source and the demand at the sink is roughly equal to the capacity of the last column cut. To ensure feasibility, there is a Hamiltonian path from the source to the sink with arcs of large capacity and large cost.

We use the GOTO-8 family to study performance of the codes on sparse GOTO problems. The other two families, GOTO-16 and GOTO-I, are used to study what happens as the network density increases. The average degree of GOTO-8, GOTO-16, and GOTO-I networks is 8, 16, and $\lfloor\sqrt{n}\rfloor$, respectively,

The GOTO generator takes five parameters: number of nodes, number of edges, maximum capacity, maximum cost, and a seed for the random number generator. In all of the GOTO families, the maximum capacity parameter is set to 2^{14} (16,384) and the maximum cost to 2^{12} (4,096). The families are parameterized by the number of nodes n and differ in graph density as follows:

- GOTO-8 family examples have density 8;
- GOTO-16 family examples have density 16;
- GOTO-I family examples have density $\lfloor \sqrt{n} \rfloor$.

The GRIDGRAPH generator is written by Resende and Veiga [34] and based on a generator proposed by Karmarkar and Remakrishnan [28]. This generator produces networks which are a rectangular grids with a source and a sink. The arc costs and capacities are drawn at random from uniform distributions. The supply at the source and the demand at the sink are equal to the capacity of the minimum cut between the source and the sink.

In addition to being very natural, this generator is useful because some minimum-cost flow codes show a drastic dependence on the grid shape. The three families we use, GRID-SQUARE, GRID-LONG, and GRID-WIDE, have different shape (indicated by the family name).

The GRIDGRAPH generator has five parameters: grid height X, gird width Y, maximum capacity, maximum cost, and a seed for the random number generator. We use this generator to produce three problem families. In all these families, the maximum capacity and cost parameters are set to 10^4 (10,000). The values of X and Y are set as follows:

- GRID-SQUARE family, $X = Y$;
- GRID-WIDE family, $Y = 16$ and X increases;
- GRID-LONG family, $X = 16$ and Y increases.

NETGEN is a ‘‘classical’’ generator developed by Klingman *et al.* [30]. This generator first creates source and sink nodes and a sparse skeleton network with large capacity arcs to guarantee feasibility. Then random arcs out of nonsinks are added to achieve the desired number of arcs.

We used a version of NETGEN (obtained from DIMACS and fixed by Bland *et al.* [5]) to generate two example families, NETGEN-HI and NETGEN-LO, NETGEN-HI problems have the same number of nodes, approximately the same number of arcs, the same cost range, and the same capacity range as GOTO-8 problems. However, the problem structure and the relative performance of the codes we study is very different. NETGEN-LO problem parameters are identical to the NETGEN-HI parameters except for maximum capacity value, which is much lower for the NETGEN-LO family. This difference affects the relative code performance.

The assignments to the 15 parameters of NETGEN are as follows:

- NETGEN-HI:

1.	seed	Random number seed (a large integer)
2.	problem	Problem number (for output documentation)
3.	nodes	Number of nodes: $N = 2^n$
4.	sources	Number of sources: 2^{n-2}
5.	sinks	Number of sinks: 2^{n-2}
6.	density	Number of (requested) arcs: $2^{\overline{n}+3}$
7.	mincost	Minimum arc cost: 0
8.	maxcost	Maximum arc cost: 4096
9.	supply	Total supply: $2^{2\times(n-2)}$
10.	tsources	Transshipment sources: 0
11.	tsinks	Transshipment sinks: 0
12.	hicost	Percent of skeleton arcs given max cost: 100%
13.	capacitated	Percent of arcs to be capacitated: 100%
14.	mincap	Minimum capacity of capacitated arcs: 1
15.	maxcap	Maximum capacity of capacitated arcs: 16,384

- NETGEN-LO: Same as NETGEN-HI except maximum capacity (the last parameter) is 16.

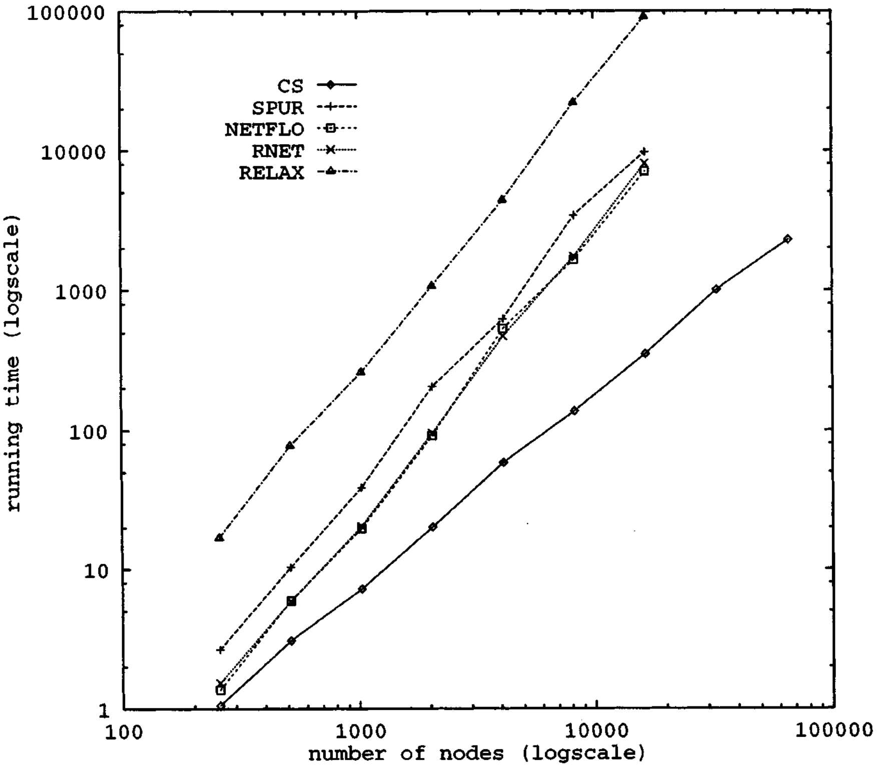
5. EXPERIMENTAL RESULTS

5.1. *GOTO Families*. We measured performance of the five codes on three families of GOTO problems with densities 8, 16, and $\sqrt{n}$. The results are summarized in Figs. 5, 6, and 7.

Compared to the other codes, CS performs significantly better on the GOTO families, and the speedup grows with problem size. The simplex codes perform noticeably better than RELAX. Their performance is very similar on the constant degree families; on the GOTO-I family, RNET performs better than NETFLO. SPUR performs somewhat worse than the simplex codes on the sparsest problems, and it performs better than NETFLO but worse than RNET on the GOTO-I family.

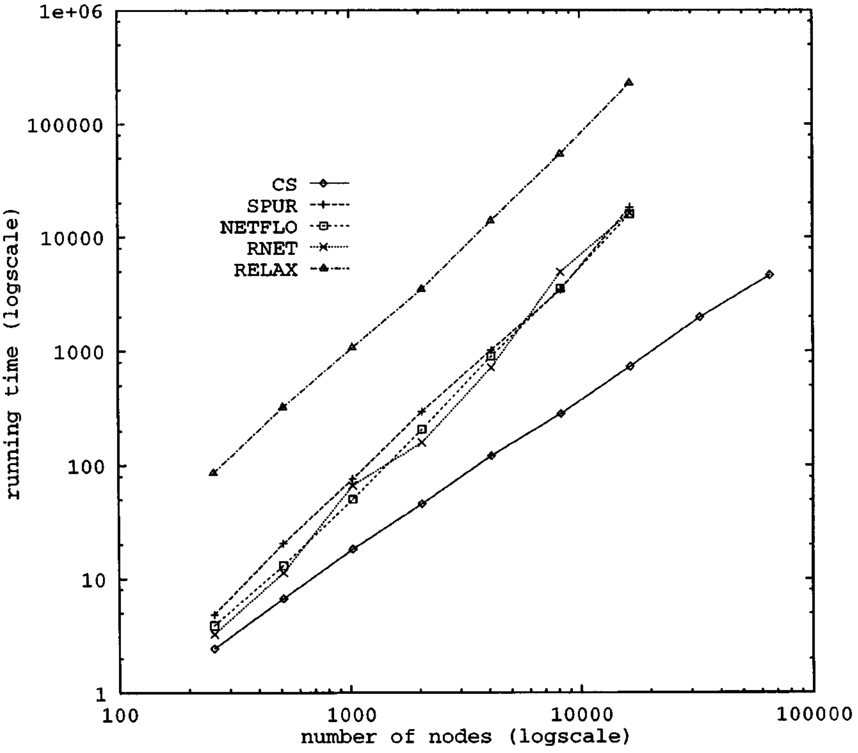
One reason for the three GOTO families is to study dependency of the algorithm performance on the graph density. For the GOTO networks, this dependency for CS appears to be roughly linear.

5.2. *GRIDGRAPH Families*. We measured performance of the five codes on three families of GRIDGRAPH problems: GRID-SQUARE, GRID-LONG, and GRID-WIDE. We ran the codes on problems with up to 2^{16} nodes. On many GRID-LONG and GRID-SQUARE problems, RNET warned about overflows and for the bigger problem sizes incorrectly declared the problems infeasible. We do not report RNET running times on these problems.



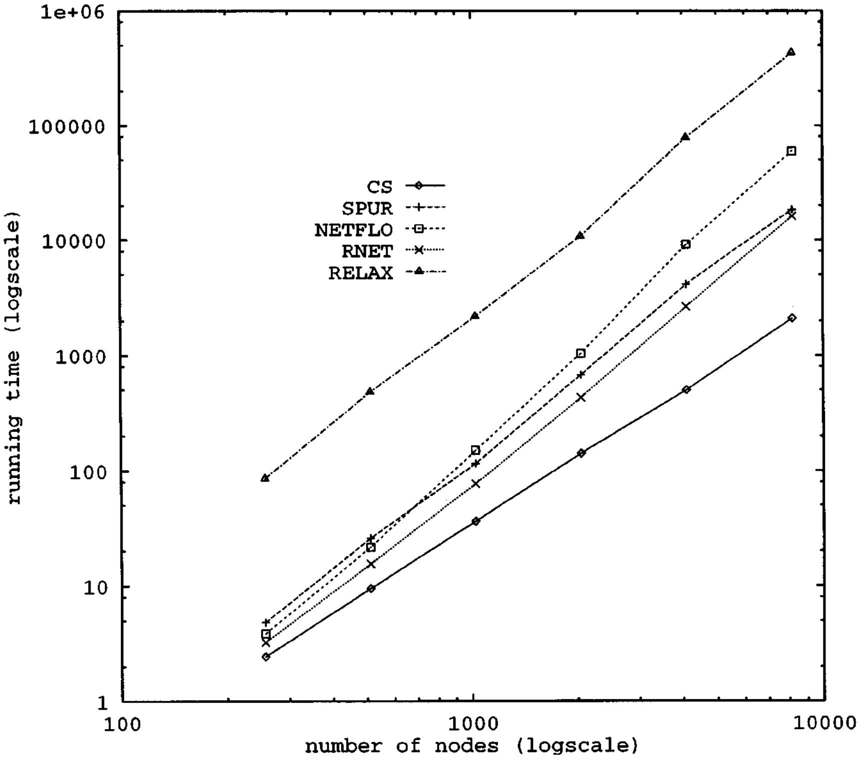
nodes	CS	SPUR	NETFLO	RNET	RELAX
256	**1.06**	2.66	1.37	1.53	16.99
512	**3.08**	10.36	5.95	5.89	77.78
1024	**7.24**	38.90	19.75	20.40	260.96
2048	**20.19**	205.77	91.85	96.12	1069.99
4096	**58.91**	626.22	535.55	471.51	4404.15
8192	**137.41**	3420.93	1644.74	1737.58	22173.18
16384	**352.40**	9728.40	7072.93	8054.14	90664.31
32768	**1005.93**	?	?	?	?
65536	**2279.03**	?	?	?	?

FIG. 5. GOTO-8 family data.



nodes	CS	SPUR	NETFLO	RNET	RELAX
256	**2.45**	4.84	3.88	3.26	85.75
512	**6.70**	20.53	13.13	11.30	322.67
1024	**18.34**	76.04	50.34	66.72	1071.15
2048	**46.07**	297.88	207.93	159.34	3501.65
4096	**121.90**	1018.83	902.77	717.51	13950.07
8192	**283.48**	3444.53	3541.59	4932.39	54075.50
16384	**732.80**	18331.56	15913.94	16250.28	227889.17
32768	**1985.14**	?	?	?	?
65536	**4616.34**	?	?	?	?

FIG. 6. GOTO-16 family data.



nodes	CS	SPUR	NETFLO	RNET	RELAX
256	**2.45**	4.84	3.88	3.26	85.75
512	**9.53**	25.90	21.67	15.54	479.76
1024	**36.42**	116.03	150.50	77.22	2192.21
2048	**141.97**	676.20	1040.06	430.21	10766.15
4096	**498.82**	4131.34	9080.58	2647.92	77956.25
8192	**2099.70**	18410.21	59543.16	16099.94	431195.25

FIG. 7. GOTO-I family data.

The GRIDGRAPH problems show the dependency of the code performance on the grid shape. CS performs similarly on square and wide grids of the same size, and a little better on long grids. Overall, CS running time is fairly independent of the grid shape. In contrast, SPUR is extremely sensitive to the grid shape: the longer the grid, the worse its performance. Even on wide grids, SPUR is somewhat slower than CS: on the long grids, SPUR is the slowest code by a wide margin; on the largest problems, it is two orders of magnitude slower than CS. NETFLO performs very well on long grids and much worse on the square and wide grids; its performance on the latter two classes is similar. RNET works best on wide grids, worse on long grids, and worse yet on square grids. On the latter two classes, the data are incomplete an may be affected by the above-mentioned overflows. Compared to SPUR, NETFLO, and RELAX, however, RNET performance on the grids is fairly robust. RELAX shows the most significant dependence on the grid shape. Its performance is reasonable on long grids, significantly slower on square grids, and extremely slow on wide grids.

On the GRID-SQUARE family (see Fig. 8), CS is the asymptotically fastest code, although the simplex codes are faster for smaller problem sizes. On the GRID-LONG family (see Fig. 9), NETFLO performed best for all problem sizes in our experiments, although CS seems to be asymptotically faster and wins on bigger problems.[2] On the GRID-WIDE family (see Fig. 10), RNET performs best. CS is slower by roughly a factor of two. The other two codes are asymptotically slower, although NETFLO is the fastest code for small problems.

5.3. *NETGEN Families*. On the NETGEN-HI family, RELAX is the fastest of the five codes. (See Fig. 11.) CS is a little slower, roughly by a factor of two on problems with over 2^{10} nodes. Performance of SPUR is very similar to that of CS on small problems and a little worse on large ones. The two simplex codes are asymptotically slower; NETFLO performs especially poorly.

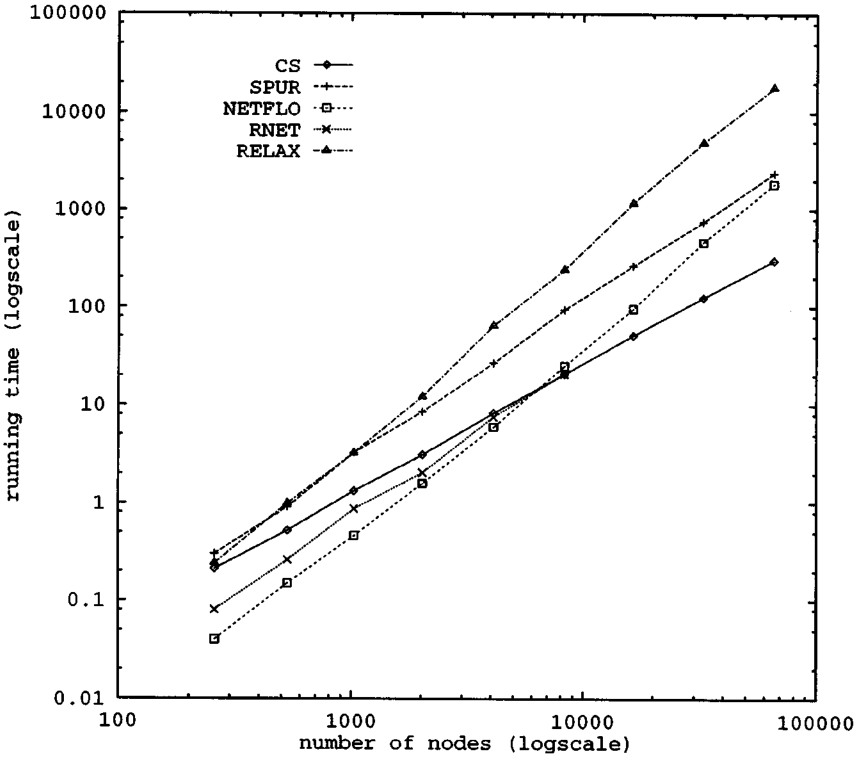
Figure 12 shows that CS is asymptotically fastest on the NETGEN-LO family. SPUR performance is very similar to that of CS on small problems and a little worse on large problems. RELAX if the fastest code on the smaller problems, but becomes slower than CS on problems with more than 2^{12} nodes. RNET has nearly the same rate of growth as RELAX, but is about a factor of two slower. NETFLO is the slowest code on this family.

6. CASE STUDY

In this section we study runs of CS on several problem instances. Although we report the data for individual instances, the variation in time and operation count was small among problems of the same size and class.

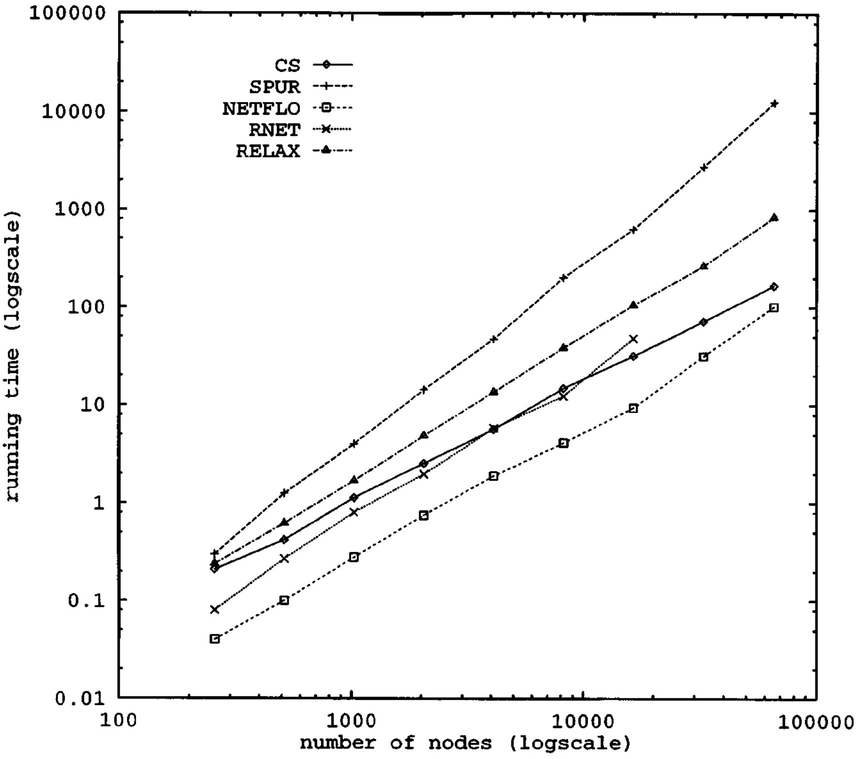
We study a biggest problem instance for each of the following classes: GRID-LONG, GRID-WIDE, NETGEN-HI, and GOTO-8. The number of nodes for these instances (in the order listed above) is 65,538, 65,538,

[2] This was verified by us and by Mauricio Resende (personal communication).



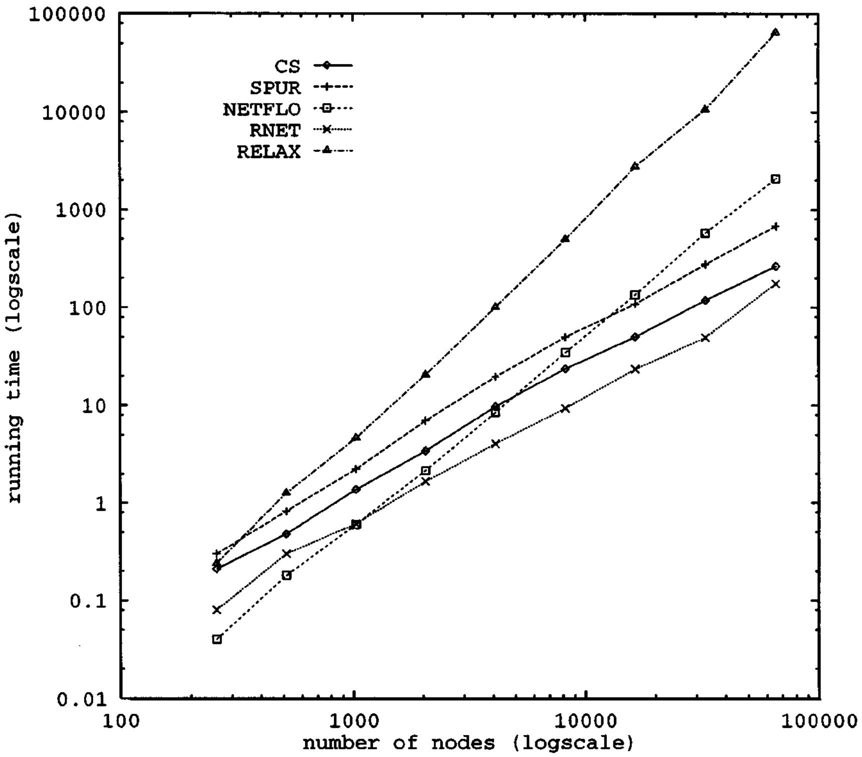
nodes	CS	SPUR	NETFLO	RNET	RELAX
258	0.21	0.30	**0.04**	0.08	0.24
531	0.52	0.91	**0.15**	0.26	0.99
1026	1.33	3.27	**0.46**	0.87	3.26
2027	3.12	8.56	**1.59**	2.05	12.20
4098	8.27	26.77	**5.97**	7.56	64.25
8283	**20.64**	93.73	25.02	20.66	240.91
16386	**51.27**	263.55	96.13	?	1170.66
32763	**124.95**	751.35	462.62	?	4922.02
65538	**299.82**	2385.33	1848.42	?	18154.00

FIG. 8. GRID-SQUARE family data.



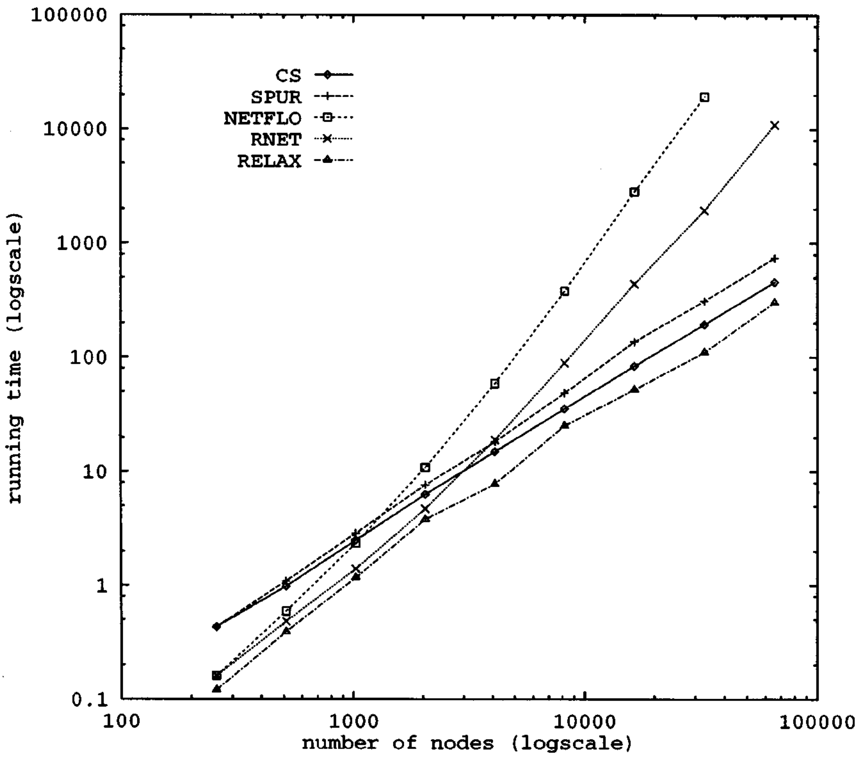
nodes	CS	SPUR	NETFLO	RNET	RELAX
258	0.21	0.30	**0.04**	0.08	0.24
514	0.42	1.25	**0.10**	0.27	0.62
1026	1.13	4.03	**0.28**	0.80	1.69
2050	2.56	14.25	**0.75**	1.98	4.88
4098	5.65	47.08	**1.90**	5.82	13.52
8194	14.79	201.29	**4.14**	12.35	38.20
16386	31.53	627.80	**9.38**	47.86	105.17
32770	71.55	2764.28	**31.42**	?	263.22
65538	166.62	12577.96	**101.53**	?	839.44

FIG. 9. GRID-LONG family data.



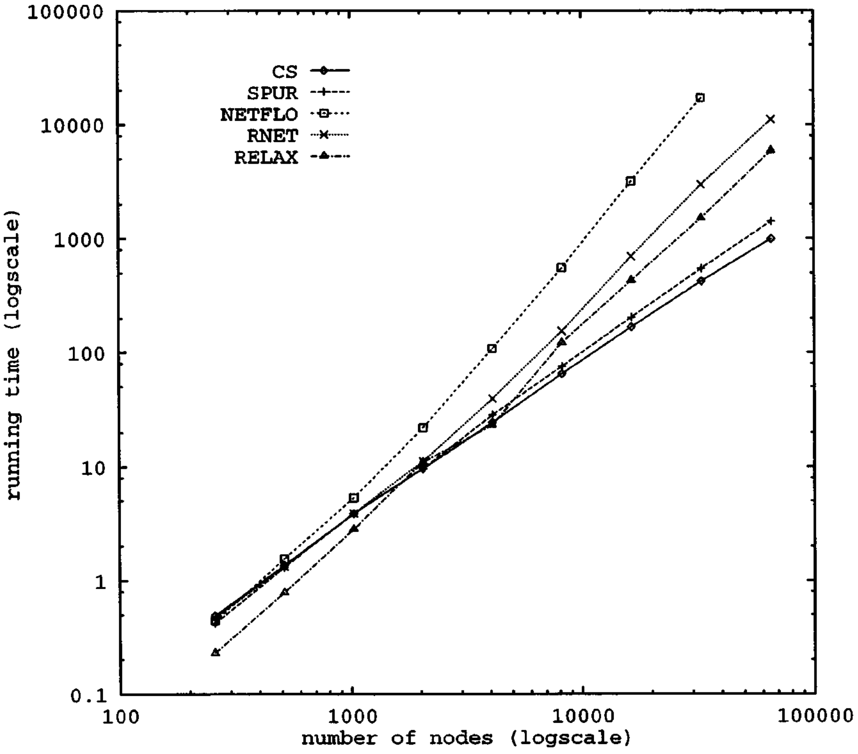
nodes	CS	SPUR	NETFLO	RNET	RELAX
258	0.21	0.30	**0.04**	0.08	0.24
514	0.48	0.82	**0.18**	0.30	1.26
1026	1.37	2.24	**0.60**	0.61	4.65
2050	3.44	7.00	2.17	**1.67**	20.52
4098	9.71	19.47	8.47	**4.06**	100.10
8194	23.57	49.94	34.60	**9.33**	493.00
16386	50.08	108.10	134.18	**23.36**	2737.45
32770	117.98	274.50	575.12	**49.38**	10637.39
65538	263.43	677.91	2075.91	**175.05**	64694.51

FIG. 10. GRID-WIDE family data.



nodes	CS	SPUR	NETFLO	RNET	RELAX
256	0.43	0.43	0.16	0.16	**0.12**
512	0.98	1.09	0.59	0.48	**0.39**
1024	2.50	2.88	2.35	1.40	**1.17**
2048	6.29	7.61	10.85	4.71	**3.81**
4096	14.98	18.30	59.09	19.01	**7.74**
8192	35.52	48.75	380.73	89.24	**25.31**
16384	83.73	136.70	2829.88	439.42	**52.17**
32768	194.11	310.86	19399.57	1945.48	**110.29**
65536	456.65	743.08	?	10969.25	**302.99**

FIG. 11. NETGEN-HI family data.



nodes	CS	SPUR	NETFLO	RNET	RELAX
256	0.49	0.42	0.44	0.47	**0.23**
512	1.37	1.32	1.56	1.34	**0.79**
1024	3.88	3.90	5.33	3.89	**2.84**
2048	**9.63**	9.68	22.04	11.20	11.05
4096	**24.49**	28.51	108.18	39.40	23.18
8192	**64.82**	75.12	551.48	154.00	122.15
16384	**167.64**	204.05	3192.57	697.24	429.89
32768	**422.03**	543.81	17063.05	2981.96	1507.76
65536	**993.41**	1415.01	?	11041.88	5895.59

FIG. 12. NETGEN-LO family data.

65,536, and 65,536. The number of arcs is 126,992, 135,152, 525,821, and 524,288. In addition to general observations, we compare the data for the former two problems and the latter two problems; each problem pair has very similar size, cost range, and capacity range, but different structure.

Figure 13 gives the data for macro operations relabel, price-update, and price-refine. The figure also gives the total running time (in seconds, excluding input and output). Since price updates are done during refine, the running time percentage given for price updates is that of the total refine time, not the total running time.

The refine and price refine operations account for most of the running time in each instance. Less than 5% of the running time is spent in other parts of the program (initialization, arc fixing, etc.). Relatively little work is done in price refine.

Figure 14 gives counts of the macro operations refine, update, and price refine. Note that the number of latter operations is determined by n and C. All instances considered here have nonzero demand function, so price refine is performed on every scaling iteration except the first one (if zero flow is feasible, price refine is performed on every iteration). For example, the number of price refine operations for the GRID-LONG instance, with $C = 10{,}000$, $n = 65{,}538$, and scale factor 12, is

$$\lceil \log_{12}(10{,}000 \cdot 65{,}538) \rceil - 1 = 8.$$

The number of refine operations is usually less than the number of price refine operations. This is because an optimal flow is usually reached when ϵ is much greater than $1/n$, and after the flow is optimal price refine always succeeds in reducing the value of ϵ and refine is never called.

The data given in Figs. 13 and 14 can be used to compute average time of a macro operation. The price refine times for the instances we consider are 3.50, 3.24, 5.08, and 5.25 seconds (in the same order as in Fig. 13). It seems that the price update time depends mostly on the network size and not on the network structure. The price refine times for the instances are 0.99, 1.00, 2.28, and 2.31 seconds. Again, these times are very close for networks of similar size.

family	time	refine	price update	price refine
GRID-LONG	185.53	80.6 %	55.2 %	15.1 %
GRID-WIDE	267.27	87.6 %	53.0 %	9.7 %
NETGEN-HI	462.13	87.9 %	43.7 %	7.7 %
GOTO-8	2623.27	97.8 %	37.9 %	1.4 %

FIG. 13. Macro operation running time percentages. Price update percentage is of the refine time, not the total time.

family	refine	price update	price refine
GRID-LONG	4	83	8
GRID-WIDE	5	124	8
NETGEN-HI	6	78	7
GOTO-8	5	421	7

FIG. 14. Macro operation count.

Figure 15 gives count (in thousands) of the following operations: push, relabel, discharge, node scan performed by price updates (*p-scan*), and node scan performed by price refine (*r-scan*).

Note that the number of push operations is less than the number of relabel operations on the GRID-LONG instance. This is due to the push lookahead heuristic; without the heuristic, the number of relabel operations cannot exceed the number of push operations.

Since a push operation is much faster than a relabel operation, and the work of discharge operations can be charged to that of relabel operations, one would conjecture that a substantial fraction of the time in refine is spent in the relabel and u-scan procedures. Profiles of program runs confirm this. Also, increasing the frequency of price updates increases the number of u-scan operations and usually decreases the number of relabel operations. These observations can be used to tune CS for a specific application.

7. CONCLUDING REMARKS

The difference in performance of SPUR and CS clearly demonstrates the usefulness of the price update and price refine heuristics. In the problems we consider, CS is slower only on small NETGEN-LO instances, and then only slightly slower. In all other instances, CS is faster. Furthermore, the speedup of CS over SPUR grows with the problem size. The rate of growth is especially big on the GOTO families and the GRID-LONG family, but barely noticeable on NETGEN and GRID-WIDE families. We conclude that price update and price refine heuristics make the underlying method significantly more robust.

family	push	relabel	discharge	u-scan	r-scan
GRID-LONG	3374	3271	3346	4260	499
GRID-WIDE	4562	5888	4070	6335	609
NETGEN-HI	4243	5809	3956	4890	474
GOTO-8	57026	29325	55216	19699	522

FIG. 15. Micro operations count in thousands.

Next we discuss how CS performance depends on the parameter values. CS is robust with respect to the scale factor α. Values of α between 4 and 50 are likely to produce running times which differ by a small constant factor (usually less than 2). The best choice of α varies from one problem family to another, and tuning α gains some performance, but not much. The best choice of the threshold parameter β depends on the problem distribution; this parameter should be set to the smallest value such that fix-ins almost never happen. The value of β used in our experiments was most constrained by the GRID-LONG family. For NETGEN families, for example, one can use a much smaller threshold and gain performance. However, for sparse graphs the performance gain cannot be high. CS is somewhat more sensitive to the choice of the price update frequency parameters ρ and π, especially on problems where CS significantly outperforms SPUR. Usually for a good update frequency, 40 to 60% of the refine time is spent on the price update operations.

Our experiments were done with fixed parameter values for every code. For an individual problem family, performance of some of the codes can be improved, in some cases considerably, by tuning the user-defined and build-in parameters to the problem class. (See [5, 25] for a discussion of how RNET parameters affect its performance.) Some algorithms can also be modified to take advantage of the network density or regularity. Also, some codes, such as RNET, were not designed for networks as large as those used in our tests, and their performance on large networks might be improved using different data structures and parameter values.

Our experimental data shows that CS compared favorably with the other three codes for the parameter values and problem families we considered. CS often outperforms the other codes by orders of magnitude for large problems sizes (for example, on GOTO-8 and GRID-SQUARE families). Its performance is robust in a sense that when it is slower than another code, it is only by a small factor. Because of the parameter and data structure issues discussed above, and the fact that on small problems CS is often not the fastest code, it is possible that another code can significantly outperform CS in some applications. However, our data strongly suggest that CS should be seriously considered for applications, especially if the problem size is large.

On easy or small problems cost scaling can be a disadvantage. Consider the best case running time. For some algorithm, such as network simplex, this time is linear. (Consider a problem on a network which is a path with a unit of supply on one end and a unit of demand on the other.) For CS, this time is linear for each scaling phase. Since the number of scaling phases can be nonnegligible, the relative performance of CS can be relatively poor. If the problem size is small enough and the largest cost C is big enough, an algorithm with a time bound independent of C may outperform CS as well.

Another implementation of the network simplex method, developed by Bronshtein and Cherkassky [9], is widely used in Russia. Limited experiments with a prototype UNIX version of this code suggests that it would have performed similarly to (but somewhat slower than) RNET.

Indirect comparison of CS with interior-point minimum-cost flow codes [27, 34] shows that CS is faster on all problems we studied.

CS was a starting point for an assignment problem code we developed jointly with Robert Kennedy [20]. This code compares favorably with the existing assignment codes. It is possible that the ideas developed in this paper in the context of the minimum-cost flow problem can be useful in developing fast codes for other network problems as well.

The network simplex method and is implementations have been studied for decades, where as the method behind CS is relatively new. Our results show that this method may be interesting from the practical point of view. Further improvements can make the method even more attractive in practice.

ACKNOWLEDGMENT

I am grateful to Joseph Cheriyan and Mauricio Resende for their help with the network simplex codes.

REFERENCES

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Network flows, *in* ‘‘Optimization. Handbooks in Operations Research and Management Science’’ (G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, Eds.), Vol. 1, pp. 211−369, North-Holland, Amsterdam, 1989.
2. R. J. Anderson and J. C. Setubal, Goldberg’s algorithm for the maximum flow in perspective: A computational study, *in* ‘‘Network Flows and Matching: First DIMACS Implementation Challenge’’ (D. C. Johnson and C. C. McGeoch, Eds.), pp. 1−18, AMS, Providence, 1993.
3. D. P. Bertsekas, ‘‘Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems,’’ Technical Report LIDS-P-1986, Laboratory for Decision Systems, M.I.T., Sept. 1986. (Revised, Nov. 1986).
4. D. P. Bertsekas and P. Tseng, Relaxation methods for minimum cost ordinary and generalized network flow problems, *Oper. Res.* **36** (1988), 93−114.
5. R. G. Bland, J. Cheriyan, D. J. Jensen, and L. Ladańyi, An empirical study of min cost flow algorithms, *in* ‘‘Network Flows and Matching: First DIMACS Implementation Challenge’’ (D. S. Johnson and C. C. McGeoch, Eds.), pp. 119−158, AMS, Providence, 1993.
6. R. G. Bland and D. L. Jensen, On the computational behavior of a polynomial-time network flow algorithm, *Math. Prog.* **54** (1992), 1−41.
7. G. Bradley, G. Brown, and G. Graves, Design and implementation of large scale transportation algorithms, *Management Sci.* **21** (1977), 1−38.

8. M. D. Chang and C. J. Chen, An improved primal simplex variant for pure processing networks, *ACM Trans. Math. Software* **15** (1989), 64–78.
9. B. V. Cherkassky, Personal communication, 1992.
10. B. V. Cherkassky and A. V. Goldberg, "On Implementing Push-Relabel Method for the Maximum Flow Problem," Technical Report STAN-CS-94-1523, Department of Computer Science, Stanford University, 1994.
11. U. Derigs and W. Meier, Implementing Goldberg's max-flow algorithm—A computational investigation, *ZOR—Methods Models Oper. Res.* **33** (1989), 383–403.
12. U. Derigs and W. Meier, Goldrmf/Goldnet-max-flow program. *Europ. J. Oper Res.* **46** (1990), 260.
13. R. B. Dial, Algorithm 360: Shortest path forest with topological ordering, *Commun. ACM* **12** (1969), 632–633.
14. L. R. Ford, Jr. and D. R. Fulkerson, "Flows in Networks," Princeton Univ. Press, Princeton, NJ, 1962.
15. S. Fujishige, K. Iwano, J. Nakano, and S. Tezuka, A speculative contraction method for the minimum cost flows: Toward a practical algorithm, *in* "Network Flows and Matching: First DIMACS Implementation Challenge" (D. S. Johnson and C. C. McGeoch, Eds.), pp. 219–246, AMS, Providence, 1993.
16. F. Glover, D. Karney, and D. Klingman, Implementation and computational comparison of primal, dual, and primal–dual computer codes for minimum cost network flow problem, *Networks* **4** (1974), 191–212.
17. A. V. Goldberg, "A New Max-Flow Algorithm," Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.
18. A. V. Goldberg, "Efficient Graph Algorithms for Sequential and Parallel Computers," Ph.D. thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Laboratory for Computer Science, M.I.T., 1987.)
19. A. V. Goldberg, Scaling algorithms for the shortest paths problem, *SIAM J. Comput.* **24** (1995), 494–504.
20. A. V. Goldberg and R. Kennedy, "An Efficient Cost Scaling Algorithm for the Assignment Problem," Technical Report STAN-CS-93-1481, Department of Computer Science, Stanford University, 1993.
21. A. V. Goldberg and M. Kharitonov, On implementing scaling push-relabel algorithms for the minimum-cost flow problem, *in* "Network Flows and Matching: First DIMACS Implementation Challenge" (D. S. Johnson and C. C. McGeoch, Eds.), pp. 157–198, AMS, Providence, 1993.
22. A. V. Goldberg, É. Tardos, and R. E. Tarjan, Network flow algorithms, *in* "Flows, Paths, and VLSI Layout" (B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, Eds.), pp. 101–164, Springer-Verlag, Berlin, 1990.
23. A. V. Goldberg and R. E. Tarjan, A new approach to the maximum flow problem, *J. Assoc. Comput. Mach.* **35** (1988), 921–940.
24. A. V. Goldberg and R. E. Tarjan, Finding minimum-cost circulations by successive approximation, *Math. Oper Res.* **15** (1990), 430–466.
25. M. D. Grigoriadis, An efficient implementation of the network simplex method, *Math. Prog. Study* **26** (1986), 83–111.
26. D. S. Johnson and C. C. McGeoch, "Network Flows and Matching: First DIMACS Implementation Challenge," AMS, Providence, 1993.
27. A. Joshi, A. S. Goldstein, and P. M. Vaidya, A fast implementation of a path-following algorithm for maximizing a linear function over a network polytope, *in* "Network Flows and Matching: First DIMACS Implementation Challenge" (D. S. Johnson and C. C. McGeoch, Eds.), pp. 267–298, AMS, Providence, 1993.

28. N. K. Karmarkar and K. G. Ramakrishnan, Computational results of an interior point algorithm for large scale linear programming, *Math. Prog.* **52** (1991), 555–586.
29. J. L. Kennington and R. V. Helgason, ''Algorithms for Network Programming,'' Wiley, New York, 1980.
30. D. Klingman, A. Napier, and J. Stutz, A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems, *Management Sci.* **20** (1974), 814–821.
31. I. Maros, Performance Evaluation of MINET minimum cost netflow solver, *in* ''Network Flows and Matching: First DIMACS Implementation Challenge'' (D. S. Johnson and C. C. McGeoch, Eds.), pp. 199–217, AMS, Providence, 1993.
32. J. Mulvey, Pivot strategies for primal-simplex network codes, *J. Assoc. Comput. Mach.* (1978), 266–270.
33. Q. C. Nguyen and V. Venkateswaran, Implementations of Goldberg–Tarjan maximum flow algorithm, *in* ''Network Flows and Matching: First DIMACS Implementation Challenge (D. S. Johnson and C. C. McGeoch, Eds.), pp. 19–42, AMS, Providence, 1993.
34. M. G. C. Resende and G. Veiga, An efficient implementation of a network interior point method, *in* ''Network Flows and Matching: First DIMACS Implementation Challenge'' (D. S. Johnson and C. C. McGeoch, Eds.), pp. 299–348, AMS, Providence, 1993.
35. H. Röck, Scaling techniques for minimal cost network flows, *in* ''Discrete Structures and Algorithms'' (U. Pape, Eds.), pp. 181–191, Carl Hansen, Münich, 1980.
36. É. Tardos, A strongly polynomial minimum cost circulation algorithm, *Combinatorica* **5(3)** (1985), 247–255.