AN EFFICIENT IMPLEMENTATION OF A SCALING MINIMUM-COST FLOW ALGORITHM

ANDREW V. GOLDBERG
COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CA 94305

August 1992

ABSTRACT. 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. In our experiments, it was always competitive with the established codes, and usually outperformed these codes by a wide margin.

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.

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 form the Powell Foundation.

1. Introduction.

Significant theoretical progress has been made recently in the area of minimum-cost flow algorithms (see [1, 18]). Practical performance evaluation of some of these algorithms is just starting [23]. Detailed studies of somewhat older methods include investigations of network simplex [21], cost-scaling [8], and relaxation [5].

In this paper we continue our work [17] on implementing one of the recent methods, the *successive approximation push-relabel method* of Goldberg and Tarjan [15, 20]. This method combines and extends the ideas of cost-scaling of Röck [30] (see also [8]), the push-relabel maximum flow method of Goldberg and Tarjan [14, 19], and the relaxation method of Bertsekas [4]. 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, which in that context has been shown superior to previous codes in several experimental studies [2, 3, 9, 11, 22, 28]. 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 implementations of the method [7, 17] 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 then our previous code SPUR [17] on most problems; in particular it does quite well on the classes where SPUR performed relatively poorly.

The improvement 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 [7, 17, 20] and some have been shown to be effective. Our current implementation succeeds in using two additional heuristics to update prices during the computation. These heuristics are closely related to the scaling shortest path algorithm [16], which has been motivated by our experimental work.

We compare our implementation to the established and widely available network simplex codes NETFLO [26] and RNET [21], and to the relaxation code RELAX [5]. 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.

Our code is best overall. In particular on big problems, it is usually better by orders of magnitude, and never looses by more then 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. In Section 6, 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 [20].

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 $f : E \to R$ satisfying the following *capacity* and *antisymmetry* constraints for each $a \in E$: $f(a) \leq u(a)$, $f(a) = -f(a^R)$.

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 no active nodes. The *cost* of a pseudoflow f is given by $cost(f) = \frac{1}{2}\sum_{a\in E} c(a)f(a)$. 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 $p : V \to R$. 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 [12].

For a constant $\epsilon \geq 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function p* if, for every residual arc a, we have $c_p(a) \geq -\epsilon$. 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 [4].

2.2. The Method. First we give a high-level description of the successive approximation algorithm (see Figure 1). The algorithm maintains a flow f and a price function

[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 if to refer to v as the tail of a and to w as the head of a, which is precise but inconvenient.

```
procedure min-cost(V, E, u, c);
    [initialization]
    ε ← C;
    ∀v,  p(v) ← 0;
    if ∃ a flow then f ← a flow else return(null);
    [loop]
    while ε ≥ 1/n do
        (ε, f, p) ← refine(ε, f, p);
    return(f);
end.
```

FIGURE 1. The successive approximation algorithm.

```
procedure refine(ε, f, p);
    [initialization]
    ε ← ε/α;
    ∀(v, w) ∈ E do if c_p(v, w) < 0 then f(v, w) ← u(v, w);
    [loop]
    while ∃ a push or a relabel operation that applies do
        select such an operation and apply it;
    return(ε, f, p);
end.
```

FIGURE 2. The generic *refine* subroutine.

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 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 on Figure 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, each of which preserves ϵ-optimality. The generic algorithm does not specify the order in which these operations are applied.

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

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** $\max_{(v,w)\in E_f}\{p(w) - c(v, w) - \epsilon\}$**.**

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

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

FIGURE 4. The *discharge* operation.

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 Figure 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* 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:

- 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 [20]. Our experience suggests that in practice the *relabel* operations are the bottleneck, and 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, which remains $O(n^2m\log(nC))$.

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 improvement of the performance on other classes.

The idea of price updates in the minimum-cost flow context had been introduced in [20]. These updates, however, need to be done in such a way that the price function after an update "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 [16, 17].

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 $\bar{S}$ to a node in S. The *set-relabel* operation reduces the price of every node in $\bar{S}$ by ϵ.

It can be shown that the *set-relabel* operation satisfies the following conditions.

(1) ϵ-optimality is preserved,
(2) the admissible graph remains acyclic,
(3) prices are monotonically decreasing,
(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 a

priority queue in a way similar to that of Dijkstra's shortest path algorithm, as described in [16]. The resulting implementation takes linear time.

If $\Omega(n)$ relabels take place before each *set-relabel*, the total cost of the latter operations during an execution of the algorithm is $O(nm \log(nC))$.

The ideal frequency of performing the global price updates is 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.

3.2. Price Refinement. As suggested in [20], *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 refinement* 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 implementation of the price refinement heuristic is based on the scaling loop of the shortest path algorithm of [16], which runs in $O(nm)$ time. This bound does not exceed the bound for *refine*, so the asymptotic running time of the algorithm does not increase by more then a constant factor.

The shortest path computation fails if an admissible cycle is created during the computation and succeeds otherwise. At 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.

This way of implementing price refinement has several advantages. One advantage is that the work done during price refinement reduces the number of *push* and *relabel* operations even in the case of failure. 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. 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 the one used in [17].

The theoretical justification of this technique is as follows [31], [20]: 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 optimal flow value computation. Arc fixing can be done after every execution of *refine*.

In the dual context, arc fixing corresponds to edge contraction. Fujishige et. al. [13] 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* has to deal with. If β is too small, however, fix-ins happen often and *refine* takes more time. We used $\beta = \Theta(n^{3/4})$ in our experiments.

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 afterwards, 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 [17]: 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 the one presented in [20] 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. In this case, however, 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.

4. Experimental Setup

We evaluated our code on eight network families produced by three generators, all obtained from DIMACS: GOTO, NETGEN, and GRIDGRAPH.

The GOTO generator is described in [17]. The generator takes five parameters: number of nodes, number of edges, maximum capacity, maximum cost, and a seed for the random number generator. We use GOTO to produce three example families. In all of these families, the maximum capacity parameter is set to 2^{14} (16384) and the maximum cost to 2^{12} (4096). 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, written by Resende [29] and based on a generator proposed by Karmarkar and Remakrishnan [25], produces networks that form rectangular grids with a source and a sink. This generator has five parameters: grid height X, grid width Y, maximum capacity, maximum cost, and a seed for the random number generator. We use this generator to produce three example families. In all these families, the maximum capacity and cost parameters are set to 10^4 (10000). 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, Napier, and Stutz [27]. We used a version of NETGEN (obtained from DIMACS and fixed by Bland et. al. [6]) to generate two example families, NETGEN-HI and NETGEN-LO. The families are identical except for maximum capacity value.[2] 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^{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: 16384.

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

Our code is written in C and compiled using SUN C compiler with the optimization option "–O4".

[2]This difference, however, may result in a substantial difference in the generated examples.

We compare our code against three widely available codes: NETFLO of Kennington and Helgason [26], RNET of Grigoriadis [21], and RELAX of Bertsekas and Tseng [5]. The version of RELAX we used was RELAXT, April 1990, All these codes are written in Fortran and compiled using SUN Fortran compiler with the optimization option "–O".

Our experiments were conducted on a SUN Sparc-2 workstation with 40 MHz CPU and 96 MB of memory. The running times we measure are user execution times (measured with 1/60 seconds precision). The input-output and preprocessing time is not included.

5. Experimental Results

We describe our experimental results below. For every family we give average running times of our code, CS (**C**ost **S**caling), 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.) We also give the average running times of NETFLO, RNET, and RELAX on the same sets of instances. We take averages over four instances of each size.

We use a fixed set of parameters for CS; in particular, the scale factor is set to 12.

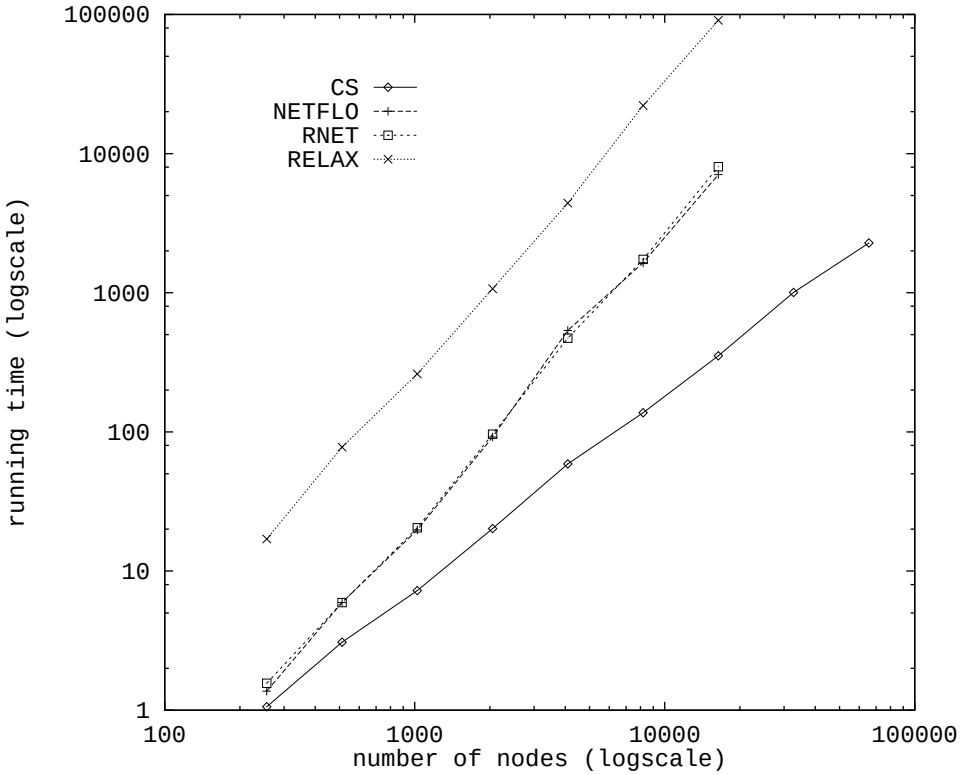
5.1. GOTO Families. We measured performance of the four codes on three families of GOTO problems with densities 8, 16, and $\sqrt{n}$. On the constant degree families, we ran CS on problems with up to 2^{16} nodes. We ran the other codes on problems with up to 2^{14} nodes because these codes were slow. On the increasing density family, we ran the experiments on problems with up to 2^{13} nodes.

The results are summarized in Figures 5, 6, and 7. Compared to the other codes, CS performs significantly better on the GOTO families. The speedup grows with the problem size. For the biggest problems, CS is two orders of magnitude faster than RELAX and an order of magnitude faster than the simplex codes. The simplex codes perform noticably better than RELAX. Their performance is very similar on the constant degree families; on the GOTO-I family, RNET performs better than NETFLO.

The main reason for having three GOTO families is to study dependency of the algorithm performance on the graph density. This dependency for CS appears to be roughly linear.

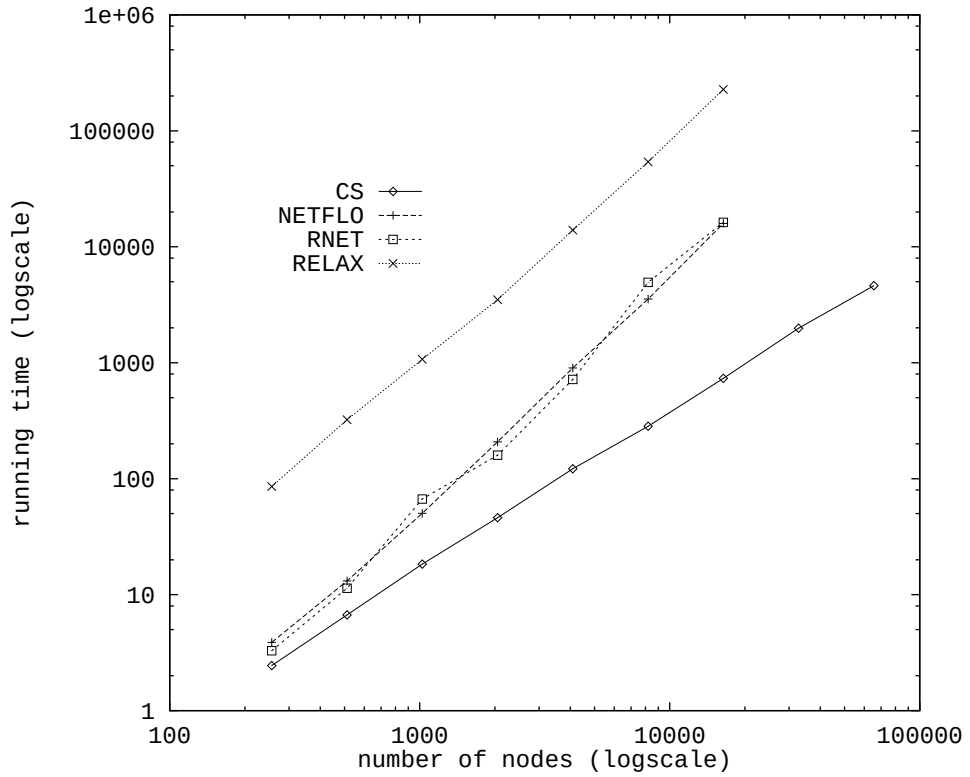
5.2. GRIDGRAPH Families. We measured performance of the four codes on three families of GRIDGRAPH problems: GRID-SQUARE, GRID-LONG, and GRID-WIDE. We ran the codes on problems with up to 2^{16} nodes. On many GRID-LONG and GRID-SQUARE problems, RNET warned about overflows and for the bigger problem sizes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

In addition to being very natural, the GRIDGRAPH problems show the dependency of the algorithm performance on the grid shape. This dependency is different for different algorithms.



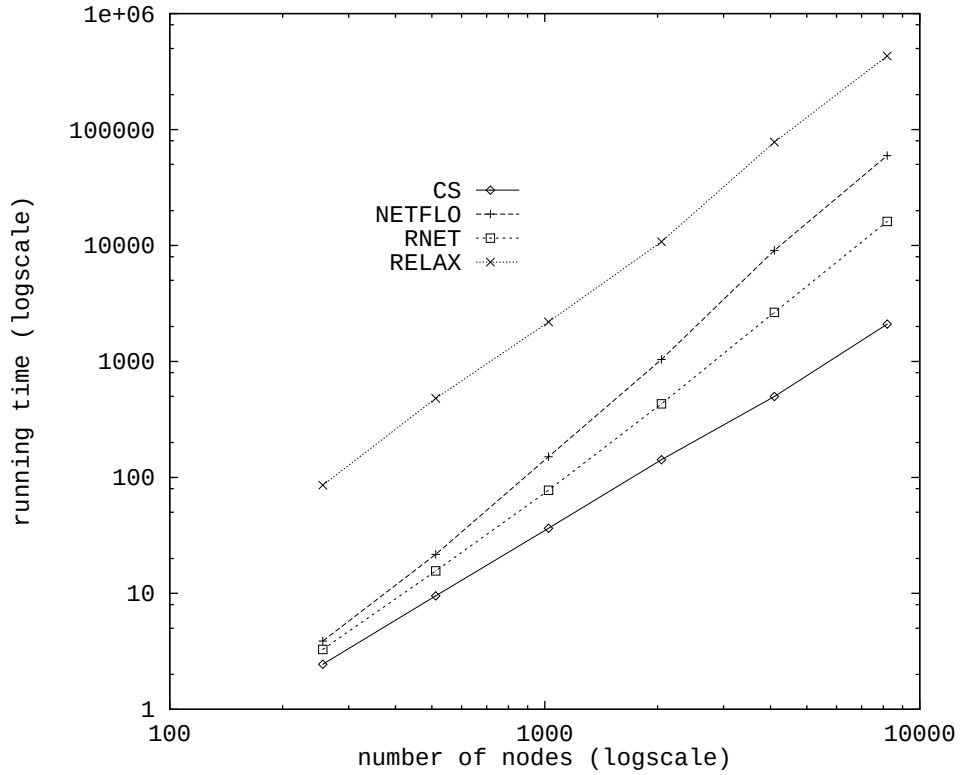
#	Running time (seconds)				Normalized time			
nodes	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
256	1.06	1.37	1.56	16.99	1.00	1.30	1.47	16.06
512	3.08	5.95	5.93	77.78	1.00	1.93	1.92	25.23
1024	7.24	19.75	20.47	260.96	1.00	2.73	2.83	36.04
2048	20.19	91.85	96.26	1069.99	1.00	4.55	4.77	53.00
4096	58.91	535.55	471.81	4404.15	1.00	9.09	8.01	74.76
8192	137.41	1644.74	1738.18	22173.18	1.00	11.97	12.65	161.37
16384	352.40	7072.93	8055.32	90664.31	1.00	20.07	22.86	257.27
32768	1005.93	?	?	?	1.00	?	?	?
65536	2279.03	?	?	?	1.00	?	?	?

FIGURE 5. GOTO-8 family data.



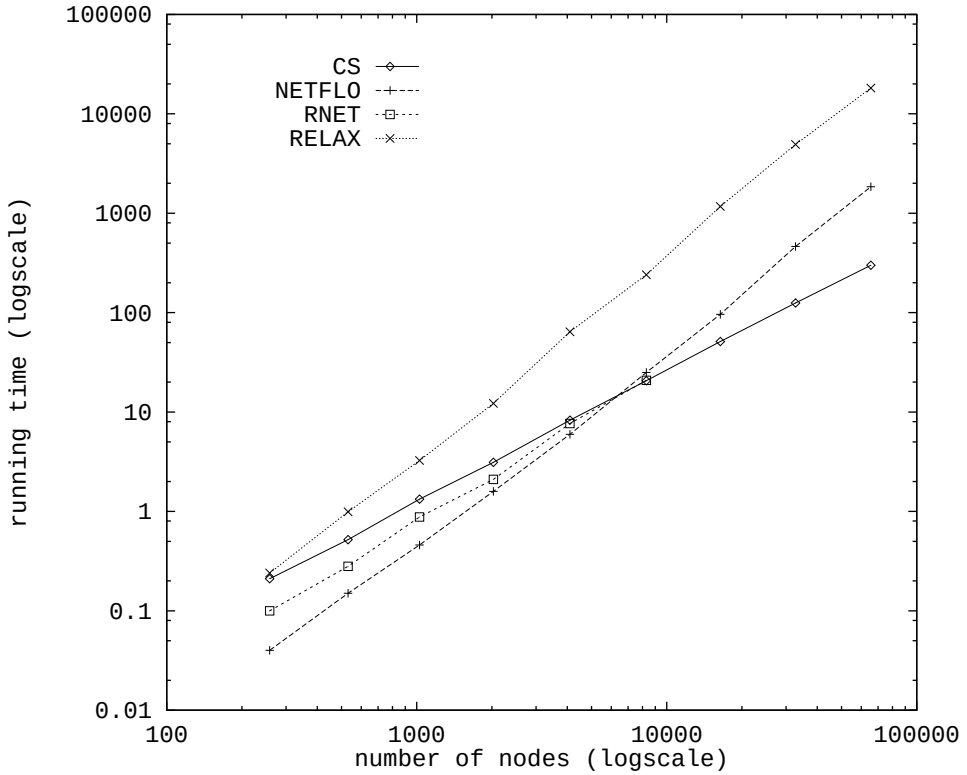
# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
256	2.45	3.88	3.28	85.75	1.00	1.58	1.34	34.94
512	6.70	13.13	11.38	322.67	1.00	1.96	1.70	48.19
1024	18.34	50.34	66.85	1071.15	1.00	2.74	3.64	58.40
2048	46.07	207.93	159.62	3501.65	1.00	4.51	3.46	76.01
4096	121.90	902.77	718.07	13950.07	1.00	7.41	5.89	114.44
8192	283.48	3541.59	4933.55	54075.50	1.00	12.49	17.40	190.75
16384	732.80	15913.94	16252.66	227889.17	1.00	21.72	22.18	310.99
32768	1985.14	?	?	?	1.00	?	?	?
65536	4616.34	?	?	?	1.00	?	?	?

FIGURE 6. GOTO-16 family data.



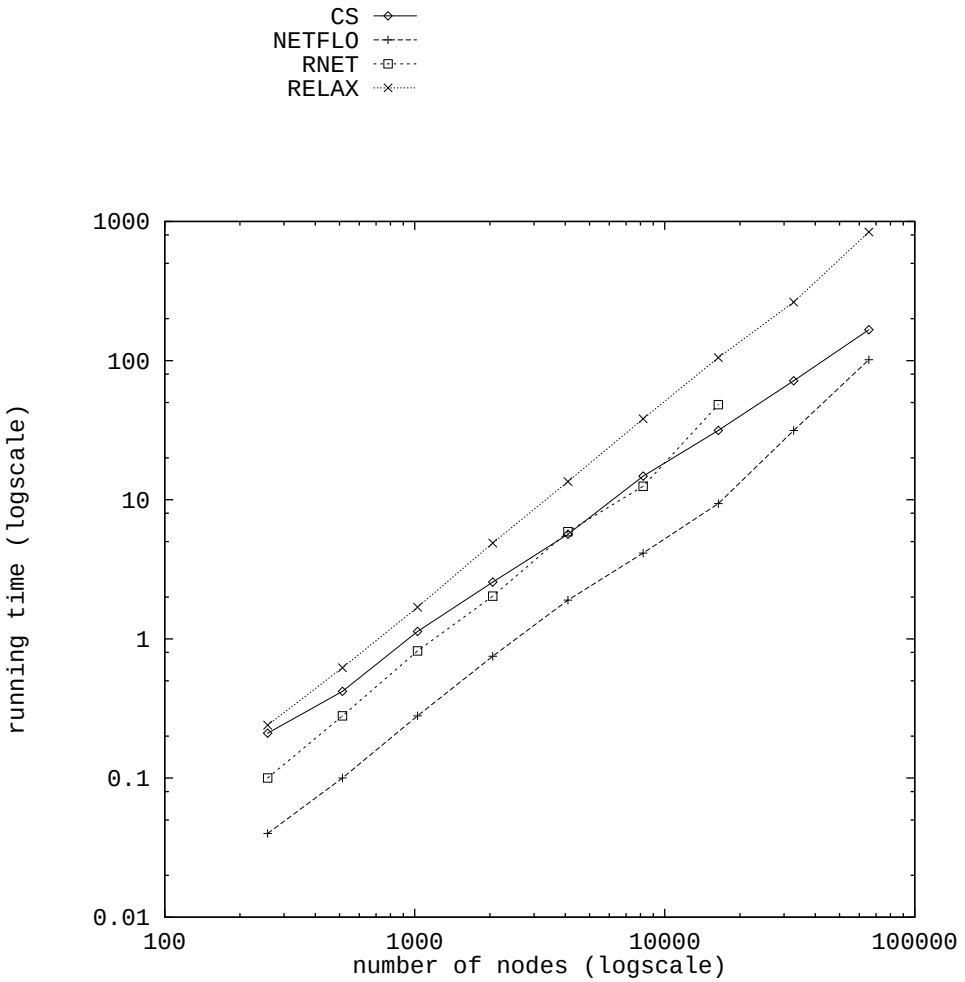
# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
256	2.45	3.88	3.28	85.75	1.00	1.58	1.34	34.94
512	9.53	21.67	15.63	479.76	1.00	2.27	1.64	50.33
1024	36.42	150.50	77.49	2192.21	1.00	4.13	2.13	60.18
2048	141.97	1040.06	430.98	10766.15	1.00	7.33	3.04	75.84
4096	498.82	9080.58	2650.27	77956.25	1.00	18.20	5.31	156.28
8192	2099.70	59543.16	16106.33	431195.25	1.00	28.36	7.67	205.36

FIGURE 7. GOTO-I family data.



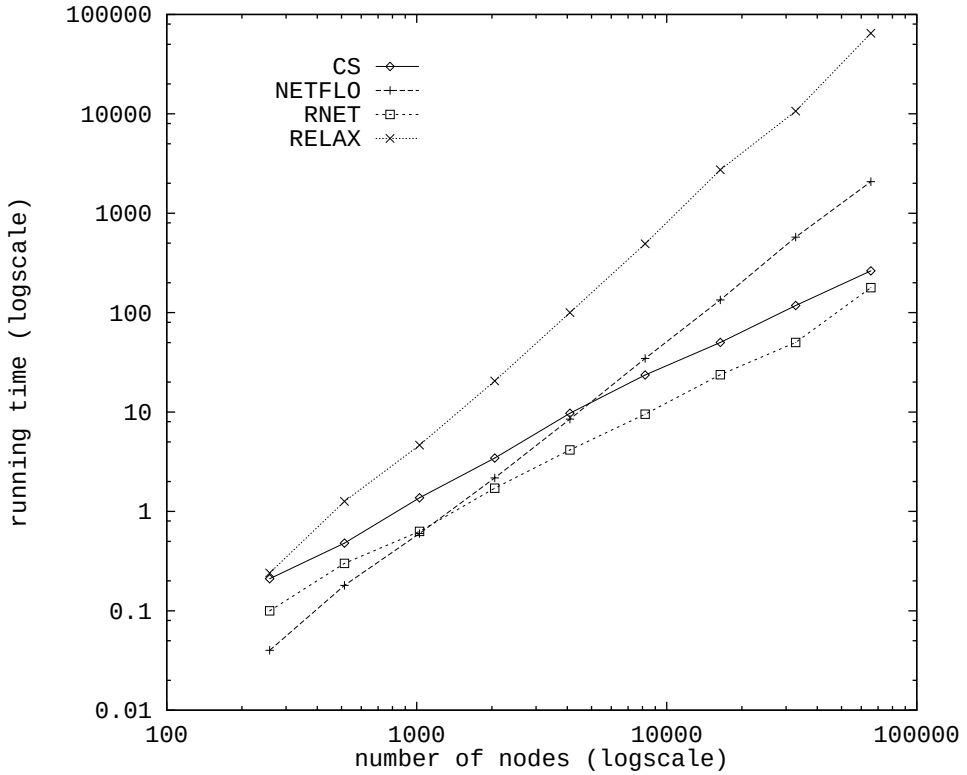
# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
258	0.21	0.04	0.10	0.24	5.10	1.00	2.30	5.70
531	0.52	0.15	0.28	0.99	3.44	1.00	1.83	6.58
1026	1.33	0.46	0.88	3.26	2.88	1.00	1.89	7.05
2027	3.12	1.59	2.10	12.20	1.96	1.00	1.32	7.66
4098	8.27	5.97	7.65	64.25	1.39	1.00	1.28	10.77
8283	20.64	25.02	20.83	240.91	1.00	1.21	1.01	11.67
16386	51.27	96.13	?	1170.66	1.00	1.88	?	22.83
32763	124.95	462.62	?	4922.02	1.00	3.70	?	39.39
65538	299.82	1848.42	?	18154.00	1.00	6.17	?	60.55

FIGURE 8. GRID-SQUARE family data.



# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
258	0.21	0.04	0.10	0.24	5.10	1.00	2.30	5.70
514	0.42	0.10	0.28	0.62	4.04	1.00	2.72	5.92
1026	1.13	0.28	0.82	1.69	4.06	1.00	2.96	6.06
2050	2.56	0.75	2.03	4.88	3.39	1.00	2.69	6.48
4098	5.65	1.90	5.89	13.52	2.98	1.00	3.11	7.13
8194	14.79	4.14	12.49	38.20	3.57	1.00	3.02	9.23
16386	31.53	9.38	48.24	105.17	3.36	1.00	5.14	11.21
32770	71.55	31.42	?	263.22	2.28	1.00	?	8.38
65538	166.62	101.53	?	839.44	1.64	1.00	?	8.27

FIGURE 9. GRID-LONG family data.



# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
258	0.21	0.04	0.10	0.24	5.10	1.00	2.30	5.70
514	0.48	0.18	0.30	1.26	2.68	1.00	1.70	7.02
1026	1.37	0.60	0.63	4.65	2.28	1.00	1.05	7.69
2050	3.44	2.17	1.71	20.52	2.01	1.27	1.00	11.98
4098	9.71	8.47	4.16	100.10	2.33	2.04	1.00	24.07
8194	23.57	34.60	9.51	493.00	2.48	3.64	1.00	51.83
16386	50.08	134.18	23.74	2737.45	2.11	5.65	1.00	115.32
32770	117.98	575.12	50.09	10637.39	2.36	11.48	1.00	212.38
65538	263.43	2075.91	178.30	64694.51	1.48	11.64	1.00	362.85

FIGURE 10. GRID-WIDE family data.

CS performs similarly on square and wide grids of the same size, and a little better on long grids. 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 is incomplete and may be affected by the above mentioned overflows. RELAX is the slowest code on all GRIDGRAPH families, and shows the most significant dependence on the grid shape. Its performance is reasonable on long grids, significantly slower on square grids, and even slower on wide grids.

On the GRID-SQUARE family, CS is the asymptotically fastest code, although the simplex codes are faster for smaller problem sizes. On the GRID-LONG family, NETFLO performed best for all problem sizes in our experiments, although CS seems to be asymptotically faster and probably would win on bigger problems. On the GRID-WIDE family, 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 two NETGEN families, we ran CS, RNET, and RELAX on the problems with up to 2^{16} nodes. We did not run NETFLO on problems with 2^{16} nodes because it was slow.

On the NETGEN-HI family, RELAX is the fastest of the four codes. (See Figure 11.) CS is a little slower, roughly by a factor of 1.6 on problems with over 2^{10} nodes. The two simplex codes are asymptotically slower; NETFLO performs especially poorly.

The NETGEN-LO family differs from the NETGEN-HI family in the capacity upper bound, which is set to 16. The resulting problems, however, have different structure then those in NETGEN-HI family.

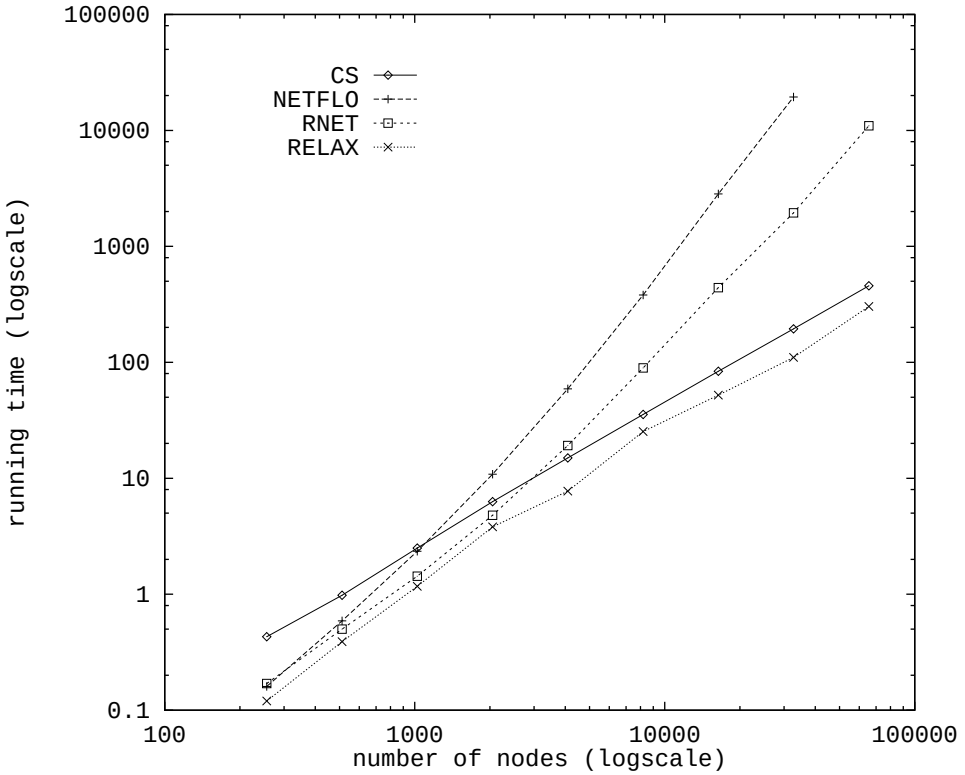
Figure 12 shows that CS is asymptotically fastest on the NETGEN-LO family. RELAX if the fastest code on the smaller problems, but becomes slower than CS on problems with more than 2^{12} nodes. RNET has nearly the same rate of growth as RELAX, but is about a factor of two slower. NETFLO is the slowest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

6. Discussion

Our experimental data shows that CS compares favorably with the other three codes. On many problem families, it outperforms the other codes by orders on magnitude for large problems sizes. Its performance is robust in a sense that when it is slower than another code, it is only by a small factor.

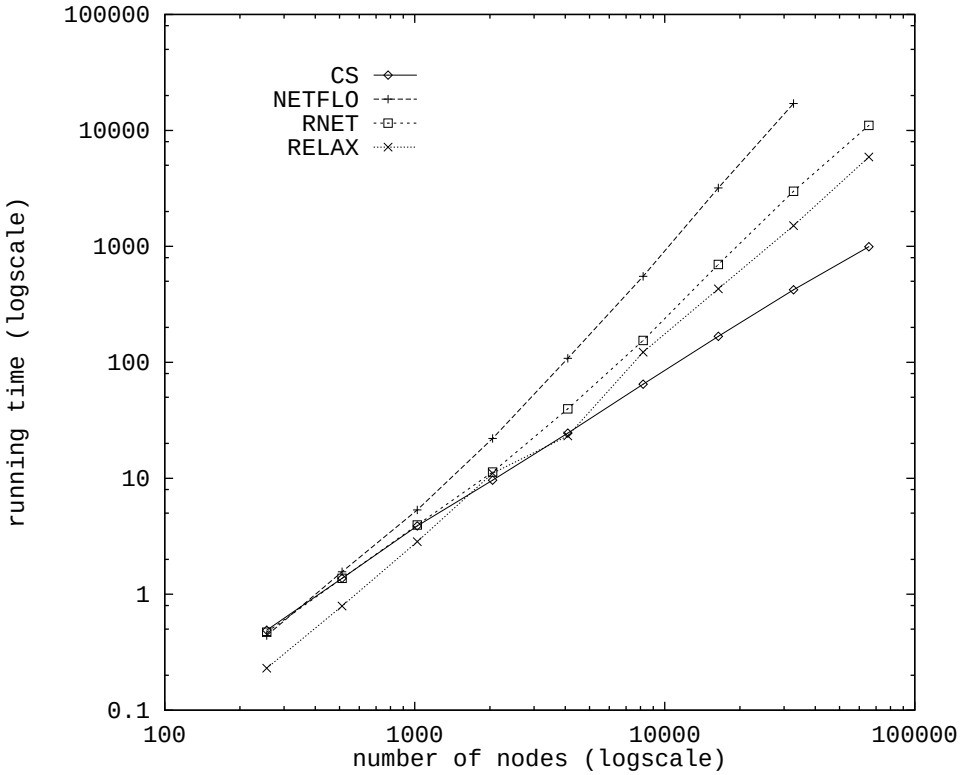
We would like to note that CS is written in C while the other codes are written in Fortran. The latter language tends to produce more efficient machine code. Also, fixing the overflow problem that RNET has on some of the GRIDGRAPH problems may require higher precision computation in RNET, and that can slow it down a little. These facts make the method behind CS even more attractive.

One of the facts about our implementation is that it is somewhat complex because of



# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
256	0.43	0.16	0.17	0.12	3.44	1.30	1.37	1.00
512	0.98	0.59	0.50	0.39	2.54	1.52	1.30	1.00
1024	2.50	2.35	1.43	1.17	2.13	2.00	1.22	1.00
2048	6.29	10.85	4.80	3.81	1.65	2.85	1.26	1.00
4096	14.98	59.09	19.18	7.74	1.94	7.64	2.48	1.00
8192	35.52	380.73	89.58	25.31	1.40	15.04	3.54	1.00
16384	83.73	2829.88	440.15	52.17	1.61	54.25	8.44	1.00
32768	194.11	19399.57	1947.05	110.29	1.76	175.89	17.65	1.00
65536	456.65	?	10973.24	302.99	1.51	?	36.22	1.00

FIGURE 11. NETGEN-HI family data.



# nodes	Running time (seconds)				Normalized time			
	CS	NETFLO	RNET	RELAX	CS	NETFLO	RNET	RELAX
256	0.49	0.44	0.47	0.23	2.13	1.91	2.05	1.00
512	1.37	1.56	1.37	0.79	1.73	1.97	1.73	1.00
1024	3.88	5.33	3.95	2.84	1.37	1.88	1.39	1.00
2048	9.63	22.04	11.32	11.05	1.00	2.29	1.18	1.15
4096	24.49	108.18	39.66	23.18	1.06	4.67	1.71	1.00
8192	64.82	551.48	154.54	122.15	1.00	8.51	2.38	1.88
16384	167.64	3192.57	698.34	429.89	1.00	19.04	4.17	2.56
32768	422.03	17063.05	2984.18	1507.76	1.00	40.43	7.07	3.57
65536	993.41	?	11046.32	5895.59	1.00	?	11.12	5.93

FIGURE 12. NETGEN-LO family data.

the heuristics involved. This causes performance penalties, in particular because of the cost of initialization inside of heuristics. This is especially noticeable for small or easy problems, where the initialization time is substantial compared to the overall running time. The heuristics also require additional fields in the network data structure, which decreases the cache hit ratio. If one is interested only in small or easy problems, better performance might be achieved by eliminating some of the heuristics.

Another implementation of the network simplex method, developed by Bronshtein and Cherkassky [10], is widely used in Russia. Limited experiments with a prototype UNIX version of this code suggest that it would have performed similarly to the network simplex codes the used in our tests.

Our experimental results suggest that some of the new minimum-cost flow algorithms are not only theoretically efficient, but also perform very well in practice. For a long time, network simplex has been the method of choice in practice; our implementation is the first one to outperform this method on a wide range of problem classes.

A lot of experimental work in the area of minimum-cost flow algorithms still remains. One could try to obtain a better implementation of the scaling push-relabel method by experimenting with different orderings of the *push* and *relabel*relabel operations or by finding other effective heuristics. Other approaches to the problem, for example those based on capacity scaling or interior-point techniques, may also prove fruitful. For recent results on the interior-point implementations, see [29, 24].

Acknowledgment

I would like to thank 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 G. L. Nemhauser, A. H. G. Rinnooy Kan, , and M. J. Todd, editors, *Optimization. Handbooks in Operations Research and Management Science, Vol. 1*, pages 211–369. North-Holland, Amsterdam, 1989.
2. R. K. Ahuja and J. B. Orlin. Personal communication. 1987.
3. R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Prespective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
4. D. P. Bertsekas. Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems. Technical Report LIDS-P-1986, Lab. for Decision Systems, M.I.T., September 1986. (Revised November, 1986).
5. D. P. Bertsekas and P. Tseng. Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems. *Oper. Res.*, 36:93–114, 1988.
6. R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladańyi. Personal communication. 1991.
7. R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladańyi. An Empirical Study of Recent Min Cost Flow Algorithms. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
8. R. G. Bland and D. L. Jensen. On the Computational Behavior of a Polynomial-Time Network Flow Algorithm. *Math. Prog.*, 54:1–41, 1992.

9. B. V. Cherkassky. Personal communication. 1991.
10. B. V. Cherkassky. Personal communication. 1992.
11. U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.
12. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.
13. S. Fujishige, K. Iwano, J. Nakano, and S Tezuka. A Speculative Contraction Method for the Minimum Cost Flows: Toward a Practical Algorithm. The First DIMACS International Implementation Challenge, 1991.
14. A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.
15. A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).
16. A. V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. Technical Report STAN-CS-92-1429, Department of Computer Science, Stanford University, 1992.
17. A. V. Goldberg and M. Kharitonov. On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
18. A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.
19. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988. A preliminary version appeared in *Proc. 18th ACM Symp. on Theory of Comp.*, 136-146, 1986.
20. A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. of Oper. Res.*, 15:430–466, 1990. A preliminary version appeared in *Proc. 19th ACM Symp. on Theory of Comp.*, 7-18, 1987.
21. M. D. Grigoriadis. An Efficient Implementation of the Network Simplex Method. *Math. Prog. Study*, 26:83–111, 1986.
22. M. D. Grigoriadis. Personal communication. 1988.
23. D. S. Johnson and C. C. McGeoch, editors. *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
24. 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 D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
25. N. K. Karmarkar and K. G. Ramakrishnan. Computational Results of an Interior Point Algorithm for Large Scale Linear Programming. *Math. Prog.*, 52:555–586, 1991.
26. J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming.* John Wiley and Sons, 1980.
27. D. Klingman, A. Napier, and J. Stutz. Netgen: A Program for Generating Large Scale Capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems. *Management Science*, 20:814–821, 1974.
28. Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
29. M. G. C. Resende and G. Veiga. An efficient implementation of a network interior point method. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
30. H. Röck. Scaling Techniques for Minimal Cost Network Flows. In U. Pape, editor, *Discrete Struc-*

tures and Algorithms, pages 181–191. Carl Hansen, Münich, 1980.

31. É. Tardos. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, 5(3):247–255, 1985.