DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem

ANDREW V. GOLDBERG AND MICHAEL KHARITONOV

July 1992

ABSTRACT. The invention of the scaling push-relabel method is an important theoretical development in the area of minimum-cost flow algorithms. In this paper we study implementations of this method. We are especially interested in heuristics that improve the performance of the algorithms in practice. Our results show that the technique works well on several problem classes with very different network structure. We also find a problem class on which the method does not perform as well and suggest directions for further improvement.

1. Introduction.

Significant theoretical progress has been made recently in the area of minimum-cost flow algorithms (see [**1, 14**] for surveys). Little is known, however, about the practical behavior of recent methods. Detailed studies of somewhat older methods include investigations of network simplex [**17**], cost-scaling [**6**], and relaxation [**4**].

The goal of this paper is to study practical performance of one recent technique, the *successive approximation push-relabel method* of Goldberg and Tarjan [**12, 16**]. This method combines and extends the ideas of cost-scaling, developed independently by Röck [**27**] and Bland and Jensen [**6**], the push-relabel maximum flow framework of Goldberg and Tarjan [**11, 15**], and the relaxation method of Bertsekas [**3**]. The experimental study of this new method is promising for two reasons.

1991 *Mathematics Subject Classification.* Primary 90B10; Secondary 68Q20.

The authors were 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 Mitsubishi Electric Laboratories.

The second author was supported in part by a Fannie and John Hertz Foundation Fellowship

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, 7, 8, 18**]. Second, the successive approximation technique used in the method requires fewer iterations of the inner loop compared to the closely related cost-scaling.

One of the main advantages of the push-relabel framework is its flexibility: it runs in polynomial time regardless of the order in which its basic operations, *push* and *relabel*, are applied. Both theoretical and practical performance, however, depend on the operation selection strategy. This is true in the contexts of both the maximum flow and the minimum-cost flow problems; in the latter context, cost scaling adds another dimension of flexibility.

We study *heuristics* that improve the method's performance on a wide range of problems without necessarily changing its worst-case asymptotic behavior. Such heuristics proved crucial in the maximum flow context; a good example is gap-relabel of Derigs and Meier [**8**]. In some cases, heuristics apply in both maximum flow and minimum-cost flow contexts, but often the maximum flow heuristics do not apply or do not help in the minimum-cost flow context and vice versa. For example, gap-relabel does not apply to the minimum-cost flow problem.

Because of the variety of available heuristics and their interrelation, finding a combination of heuristics that works best in practice is a difficult task. We describe the heuristics we found useful in our experiments and those that we believe are worth further investigation. We implement several heuristics whose combination significantly improves the algorithm's performance.

Our experimental results show that the method works very well on a wide class of problems. We also identify a class of problems on which our implementation does not work as well, suggesting a direction for further improvement.

Recently, the first author developed another code based on the lessons learned from the work described in this paper and the ideas from [**13**]. The performance improvement of the new code over SPUR is considerable. These results are briefly discussed in Section 8, and will be described in detail in an upcoming paper.

This paper is organized as follows. Section 2 gives the relevant definitions and outlines the method. Section 3 talks about variations of the algorithm and comments on some implementation aspects. Section 4 discusses heuristics that can be used in the method. Section 5 describes our experimental setup and Section 6 gives the experimental results. Section 7 examines how much heuristics help. In Section 8, 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 [**16**].

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 [**9**].

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 [**3**].

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 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

[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.

repeatedly reduces ϵ by a constant factor α, the choice of which is discussed later. When $\epsilon < 1/n$, the algorithm terminates.

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 algorithm terminates in $\lceil \log_\alpha(nC) \rceil$ iterations. After an optimal flow is found, our implementation uses a Bellman-Ford shortest path computation to obtain an optimal price function.

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 negative reduced cost. It consists of decreasing $p(v)$ to the smallest value

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.

allowed by the ϵ-optimality constraints, namely $\max_{(v,w)\in E_f}\{p(w) - c(v,w) - \epsilon\}$. (If such a maximum is taken over the empty set, the problem is infeasible.)

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. The *first-in first-out* (FIFO) algorithm consists of maintaining 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. Another possibility is to maintain the set of active nodes as a stack, popping a node to be discharged from the stack and pushing newly activated nodes on the stack, as the LIFO algorithm does.

Two other ways of ordering the *discharge* operations use the fact that the *admissible graph remains acyclic* during *refine*. These two methods, called *first-active*

```
procedure first-active;
    let L be a list of all nodes;
    let v be the first node on L;
    while ∃ an active node do begin
        if v is active then begin
            discharge(v);
            if the discharge has relabeled v then
              move v to the front of L;
        end;
        else replace v by the node after v on L,
    end;
end.
```

FIGURE 5. The *first-active* method.

and *wave*, are described next.

The first-active method maintains a list L of all the nodes of G in topological order with respect to the current admissible graph G_A, *i.e.*, if (v, w) is an arc of G_A, v appears before w on L. Initially L contains the nodes of G in any order. The method consists of repeating the following step until no active nodes remain: Find the first active node on L, say v, apply a *discharge* operation to v, and move v to the front of L if the *discharge* operation has relabeled v.

In order to implement this method, we maintain a *current node* v of L, which is the next candidate for discharging. Initially v is the first node on L. The implementation, described in Figure 5, repeats the following step until there are no active nodes left: If v is active, apply a *discharge* operation to it, and if this operation relabels v, move v to the front of L; otherwise (*i.e.*, v is inactive), replace v by the node currently after it on L. The reordering of L maintains a topological ordering with respect to G_A. This implies that the implementation is correct.

The *wave* method differs from the first-active method in the following way: after moving a node v to the front of L, set the current node to the node after the *old* position of v; if v is the last node on L, set the current node to the first node on L.

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.
- The number of *push* operations in the first-active and wave methods is $O(n^3)$.

A dynamic tree data structure can be used to do several *push* operations at once, as described in [**16**]. 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. Method Variations and Implementation Notes

3.1. Scaling Factor Selection. Recall that the goal of *refine* is to reduce the error parameter ϵ by a factor of α. The running time of the algorithm depends on the value of α. This dependence, however, is not uniform: a value of α that is the best for one problem instance may not be the best for another instance.

We found that scaling factors of 4, 5, and 6 work well on the classes of problems we encountered. We used $\alpha = 5$ in our experiments. The scaling factor of 2 and large scaling factors (20 and above) are usually a bad choice.

As one would expect, if two scaling factors result in the same number of executions of *refine*, the smaller scaling factor almost always gives a better running time. Unfortunately, our implementation cannot take advantage of this observation because it uses early termination detection (described below) and cannot predict the number of executions of *refine*.

3.2. A Simpler Relabeling. The push-relabel method can use a simpler version of the *relabel* operation which replaces $p(v)$ by $p(v)-\epsilon$ instead of the maximum computed as described in Figure 3. This version of *relabel* is more efficient but results in a higher number of *relabel* operations because it often does not reduce $p(v)$ as much as the variant described in the Figure 3. We experimented with the two versions of *relabel* and found them to be very close in performance. Our final implementation uses *relabel* given in Figure 3.

We would like to note that the relative performance of the two variants of the method depends on the implementation details and on the problem structure.

3.3. A Variation of Discharge. The *discharge* procedure, as described in Figure 4, terminates when v is no longer active. An alternative, with the same worst-case asymptotic time bounds, is to terminate *discharge* when v is no longer active or when v has been relabeled by the *discharge*. We found the former version to be superior for most implementations.

3.4. Implementing First-Active and Wave Methods. Recall that the first-active algorithm selects an active node with no active predecessors. Implemented as described in Section 2.2, this algorithm maintains a topologically ordered (with respect to the admissible graph) list of nodes and finds the next active node by scanning the list.

Several researchers [**28, 30**] have observed that in practice a small percentage of nodes is active during an execution of *refine*. We also observed that the number of active nodes, although large at first, decreases quickly and stays low during most of the execution. The fraction of active nodes seems to go down with problem size; even for a moderate problem size (a thousand nodes) it is already below one percent.

Because of the small number of active nodes, the above mentioned implementation spends most of its time scanning the list in search of the next active node. In theoretical work, this problem came up in the context of an implementation that uses dynamic trees to reduce the number of *push* operations [**16**]. The proposed solution was to use finger search trees. The theoretical motivation for using this data structure, which is quite sophisticated, is to avoid introducing a logarithmic factor into the running time bound. In the context of the implementation discussed here (which does not use dynamic trees), this theoretical objective does not apply, and of course in practice one uses the data structure that works best.

One alternative is to use balanced (or self-adjusting) search trees to maintain the list of nodes. However, the logarithmic and constant factors introduced by this data structure are relatively large. We obtain a better implementation of the first-active method by using instead a priority queue implemented as a linear list. We refer to this implementation as the PFA implementation throughout the rest of the paper.

The PFA implementation maintains topological numbers of nodes which are updated during relabels. The priority queue maintains the list of active nodes ordered by their topological numbers. The active node with the largest topological number is removed from the priority queue and discharged, and newly activated nodes are added to the priority queue. Note that the PFA implementation is very much like the FIFO implementation, except that the topological numbers are maintained and a priority queue is used instead of a fifo queue. The PFA implementation works fairly well because priority queues are quite efficient in practice and the number of active nodes is small on the average.

The corresponding implementation of the wave algorithm is slightly more complicated and uses two priority queues, *current* and *next*. A node v to be discharged is removed from the current queue, and its topological number l is saved. A newly activated node w is added to the current queue if its topological number is less than l and to the next queue otherwise (the latter case can occur if *discharge* relabels v). When the current queue is empty, the current and the next queues are exchanged.

3.5. Comparison of Selection Strategies. As mentioned in Section 2.2, the efficiency of the method depends on the strategy used to select the next node to be discharged. In addition to the FIFO, LIFO, first-active, and wave selection strategies, we experimented with several other strategies, including selecting a node with maximum excess and selecting an active node with the minimum topological number.

The FIFO and LIFO strategies give similar results, both in terms of running times and operation count. The first-active and wave implementations seem to result in a slightly larger number of operations and add to the computational costs of *relabel* and node selection operations. The maximum excess strategy results in a significantly higher number of operations and these operations require more computation as well.

Although our first-active and wave implementations are not quite as fast as the FIFO and LIFO implementations, they are not drastically slower. With addition of some heuristics, the former implementation ran faster than the latter ones on some instances. We used the FIFO strategy in our final tests. According to preliminary experiments on moderate sized problems, the LIFO strategy would give similar results.

The node selection strategy performance depends on heuristics used by the implementation. Although FIFO and LIFO strategies are easier to implement, have lower computational cost, and worked faster in out study, the first-active and wave strategies cannot be dismissed; it is possible that one of these strategies will win when combined with appropriate heuristics.

4. Heuristic Improvements

4.1. Price Refinement and Termination Detection. As suggested in [**16**], refine may produce a solution which is not only ϵ-optimal, but also (ϵ/α)-optimal. This can be tested by adding ϵ/α to the reduced costs and performing a Bellman-Ford shortest path computation on the residual graph with respect to the resulting length function. If no negative cycles are detected, f is (ϵ/α)-optimal and the price function showing this can be obtained as a by-product. In this case we say that the heuristic succeeds. We call this heuristic *price refinement.*

As observed by Bland et. al. [**5**], price refinement rarely succeeds except at the end of the computation when the current solution is optimal even though ϵ is greater than $1/n$. This observation suggests using the Bellman-Ford computation (without changing the reduced costs) to test for optimality.

We check for optimality after every iteration of refine when ϵ is 1 or less, and call this *early termination detection.*

Price refinement can be also done using a minimum-mean cycle computation [**21, 22**]; see [**16**] for details. We did not evaluate this way of doing price refinement.

4.2. Arc Fixing. The arc fixing heuristic involves "removing" some arcs from the graph, thus reducing the number of times the algorithm examines an arc.

The theoretical justification of this technique is as follows [**29**], [**16**]: if the current circulation 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. [**10**] 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 not examined by *refine*, and the flow on these arcs does not change.

Speculative arc fixing is done after every execution of *refine*. During this process, we first check if all fixed arcs with current reduced cost below $-\epsilon$ are saturated. If this is not the case, the current flow is not ϵ-optimal; we say that the speculative arc fixing has failed. In this case we use the regular arc fixing strategy and invoke *refine* again without decreasing ϵ; the resulting solution is ϵ-optimal. Then we decrease ϵ and examine all the arcs again, fixing arcs speculatively.

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, speculative arc fixing fails often and the number of invocations of *refine* increases. We used $\beta = 5 + 0.7n^{5/8}$ in our experiments.

Our implementation uses the speculative arc fixing heuristic.

4.3. 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.

When price update heuristics are used, the following conditions must be satisfied for the theoretical analysis of the method to apply:

(i) ϵ-optimality must be preserved,
(ii) acyclicity of the admissible graph must be preserved,
(iii) prices must be monotonically decreasing,
(iv) prices of nodes with negative excess must remain unchanged.

One way to update prices is at the beginning of *refine,* when all negative reduced cost arcs are saturated. This can be done using a shortest path computation on the residual graph with respect to the length function given by reduced costs. For each node, the shortest distance to the set of nodes with negative excesses is computed and subtracted from the node's price. Preliminary experiments suggest that this price update heuristic decreases running times a little on some classes of problems and does not help on other classes. Furthermore, this heuristic can be used at most once per an execution of *refine.*

Suppose the algorithm rounds costs to the nearest multiple of ϵ every time ϵ decreases. Then prices and reduced costs are always in units of ϵ. To update prices at any point during an execution of *refine* we can add ϵ to the reduced costs and use the resulting function as a length function. This length function is nonnegative on residual arcs. We can update prices by computing distances from all nodes to the set of nodes with negative excesses and subtracting these distances from the node prices to obtain new prices.

The rounding of costs is needed to assure that the admissible graph remains acyclic. When the costs are rounded, the only possible negative value of reduced cost of a residual arc is $-\epsilon$ by ϵ-optimality. Since the admissible graph is acyclic before a price update, the minimum mean cycle cost in the residual graph is greater than $-\epsilon$. A cycle in the admissible graph must have a mean cost of $-\epsilon$, so the admissible graph cannot have cycles after the update.

Although this kind of price update, when done often enough, reduces the number of *push* and *relabel* operations, it is expensive and did not produce consistent running time improvements in our experiments.

One of the drawbacks of the price update technique described above is that it is too global in a sense that it may change prices of most nodes. Next we describe a price update strategy that does not have this drawback and does not require the rounding of costs. This update strategy is a modification of Dijkstra's shortest path algorithm. It also can be viewed as a relaxed version of positive cut canceling [**4, 19**].

We define a *set-relabel* operation as follows. Let S be a set of nodes that 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

$$\epsilon + \min\{c_p(v,w) | (v,w) \in E_f,\ v \in \bar{S},\ w \in S\}.$$

(Alternatively, the prices can be reduced by ϵ.) The $O(n^2)$ bound on the number of relabels per *refine* applies to set-relabels as well. If fact, if set-relabels are used as described below, we can make a stronger claim that each node participates in $O(n^2)$ 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 can be implemented using a priority queue in a way similar to that of Dijkstra's shortest path algorithm. Note that this implementation of price update works correctly even if it is stopped after an arbitrary iteration.

Although SPUR does not use any price update heuristics, subsequent work shows that price updates based on the *set-relabel* operation substantially improves performance of the method. See Section 8 for discussion.

4.4. Lookahead and Generalizations. 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: 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 [**16**] 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. See Section 7 for experimental data.

The lookahead heuristic can be generalized in the following way. Define the *admissible capacity* of a node x to be $a_f(x) = -e_f(x) + \sum_{(x,y)\in E_A} u_f(x,y)$. (The sum taken over the empty set is defined to be zero.) The admissible capacities can be maintained by the *push* and *relabel* operations.

The *relabel* operation can be extended to apply to nodes v with nonnegative excess and no outgoing admissible arcs. The *push* operation can be modified so that the amount pushed from v to w is bounded by $a_f(w)$ as well as by e_f and $u_f(v,w)$. If $a_f(w)$ provides the tightest constraint on the amount pushed, (v,w) remains admissible and v remains active. In this case v cannot be relabeled until w is relabeled. We call such a node w *hyperactive.* We assume that discharging of v stops if it makes a node w hyperactive, and v is not processed again until w is relabeled.

Note that while a hyperactive node v is being discharged (and before it is relabeled), another node w can become hyperactive, and w must be relabeled before v is. Note, however, that the arc (v,w) remains admissible until w is relabeled. Since the admissible graph is acyclic, we can process the hyperactive nodes in the proper order. For example, we can keep such nodes on a stack. Alternatively, we can select a hyperactive node with the smallest topological number with respect to the admissible graph.

This generalization of the lookahead heuristic adds another dimension to the flexibility of the push-relabel method. Preliminary experiments show that this heuristic reduces the number of *push* operations, but additional work is needed to determine if some version of this heuristic reduces the number of *relabel* operations and if such a version yields running time improvements.

4.5. Good Initial Solution. The method could benefit from a good initial flow and price function. For example, after finding an initial flow, one can use a minimum cycle mean computation to find a price function that minimizes the error parameter ϵ (see [**16**]). Better initial prices may reduce the number of iterations of *refine*, especially on networks that have a small number of arcs with very large costs (such arcs are supposed to be used only if the problem would be infeasible without them). Other ways of finding good initial solutions may be possible.

4.6. Heuristics Used. Our implementation uses early termination detection, lookahead, and speculative arc fixing heuristics.

5. 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 the appendix. 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 $\lfloor\sqrt{n}\rfloor$.

NETGEN is a "classical" generator developed by Klingman, Napier, and Stutz [**24**]. 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. 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.

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

The GRIDGRAPH generator, written by Resende [**26**] and based on a generator proposed by Karmarkar and Remakrishnan [**20**], 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.

We also experimented with GTE instances. These instances were obtained by Leong, Shor, and Stein as subproblems in a multicommodity flow algorithm [**25**] applied to a real-life problem. The GTE networks all have the same graph structure but different demands and costs. These instances are quite small; each has 49 nodes and 520 edges.

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

As a benchmark to compare our code against, we use the code RELAX of Bertsekas and Tseng [**4**]. The version we used was RELAXT, April 1990, compiled using SUN Fortran compiler with the optimization option "–O".

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

A machine calibration experiment with the "wmatch" program provided by DIMACS was performed. The C version of the program was compiled without optimization and with level 4 optimization (used to compile our code). In each case the "run.me" shell script provided by DIMACS was run four times and results were averaged over these runs. For the wmatch test 1 running times were 2.8 real, 2.7 user, and 0.1 system (not optimized) and 2.2 real, 2.0 user, and 0.1 system (optimized). For the wmatch test 2 running times were 24.6 real, 24.2 user, and 0.2 system (not optimized) and 19.2 real, 18.1 user, and 0.3 system (optimized).

6. Experimental Results

We describe our experimental results below. For every family and every problem size, we give average running times of our code, SPUR (**S**caling **PU**sh-**R**elabel), 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 RELAX on the same sets of instances. We take averages over four instances of each size.[3] For

[3] Because of the large variation in running times, a bigger sample size would have given more uniform results, but would limit experiments on large problem sizes.

each problem family, we try to go to the biggest possible size, subject to the limits imposed by computation times and generator restrictions.

6.1. GOTO Families. We measured performance of SPUR on three families of GOTO problems with densities 8, 16, and $\sqrt{n}$. The results are summarized in Figure 6. These experiments show that the algorithm's performance depends on both the number of nodes and the number of edges. The running times seem to depend more on the number of nodes than on the number of edges.

Figures 7, 8, and 9 compare the running times of SPUR on the GOTO families with those of RELAX. SPUR is faster by an order of magnitude. The running times ratio depends on the problem density. There is no noticeable asymptotic growth of the ratio as the number of nodes increases.

6.2. NETGEN Families. Figure 10 gives running time of SPUR on the NETGEN families. Recall that these families differ only by the capacity upper bound. SPUR's performance is only slightly worse for tightly capacitated problems.

Figure 11 gives the running times of SPUR and RELAX on the NETGEN-HI family. RELAX is faster by a factor of two to three most of the time.

The NETGEN-LO family differs from the NETGEN-HI family in the capacity upper bound, which is set to 16. RELAX is a little faster on small instances, and SPUR is faster on larger instances. The RELAX/SPUR running time ratio grows with problem size, and on networks with 2^{16} nodes SPUR is faster by more than a factor of four.

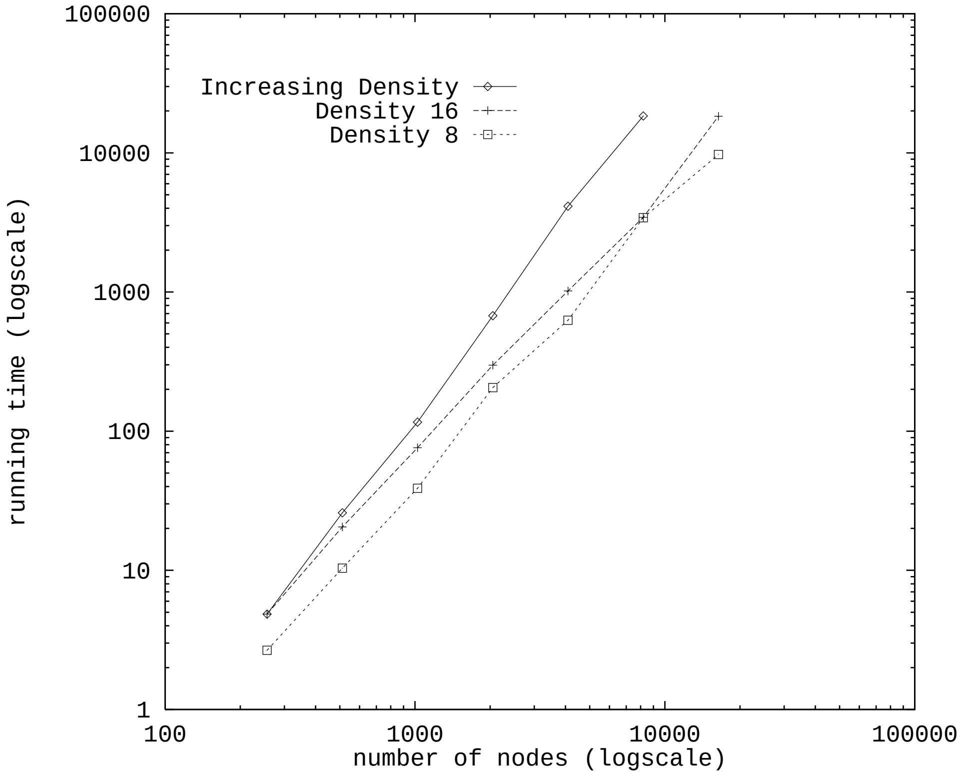
6.3. GRIDGRAPH Families. Figure 13 summarizes our experiments with the GRIDGRAPH families. The data indicates that SPUR runs much faster on shorter networks than it does on longer ones.

RELAX running times depend on the grid geometry in the opposite way. As the result, SPUR is asymptotically faster on wide grids and RELAX on long grids. On square grids, SPUR is asymptotically faster. See Figures 14 – 16 for running time comparisons.

Observe that the running times of SPUR on wide grids are very close to the running times of RELAX on long grids of the same size. However, the running times of SPUR on long grids are asymptotically better than those of RELAX on wide grids. Thus SPUR has a better "worst-case" performance on GRIDGRAPH examples.

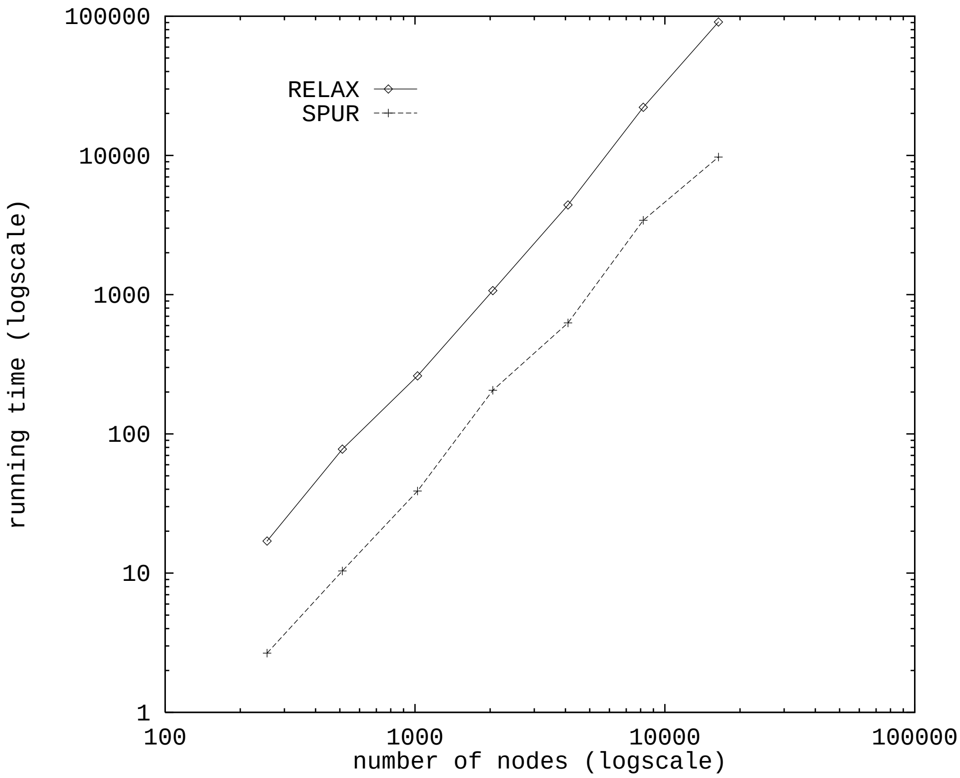
6.4. GTE Instances. Figure 17 gives running times of SPUR and RELAX on GTE instances. Recall that these instances are subproblems in a multicommodity flow algorithm and have the same underlying graph, but different demands and costs.

The graph is small and SPUR solves each instance in a fraction of a second. Although some problems are harder for SPUR than others, its running times are all within a factor of three of each other.



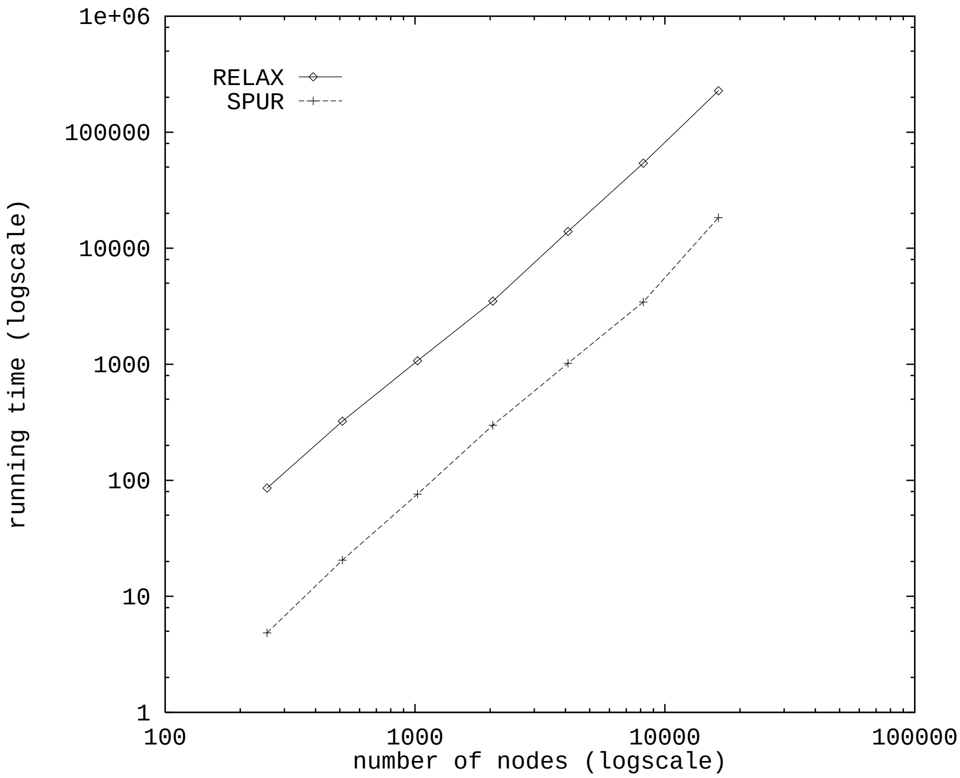
Problem size (nodes)	SPUR running time			$\sqrt{n}$
	Density 8	Density 16	Density $\sqrt{n}$	
256	2.662	4.842	4.842	16
512	10.358	20.525	25.904	23
1024	38.904	76.037	116.033	32
2048	205.767	297.875	676.196	45
4096	626.221	1018.833	4131.342	64
8192	3420.925	3444.529	18410.209	91
16384	9728.400	18331.559	timeout	128

FIGURE 6. Average running times (in seconds) of SPUR on GOTO families.



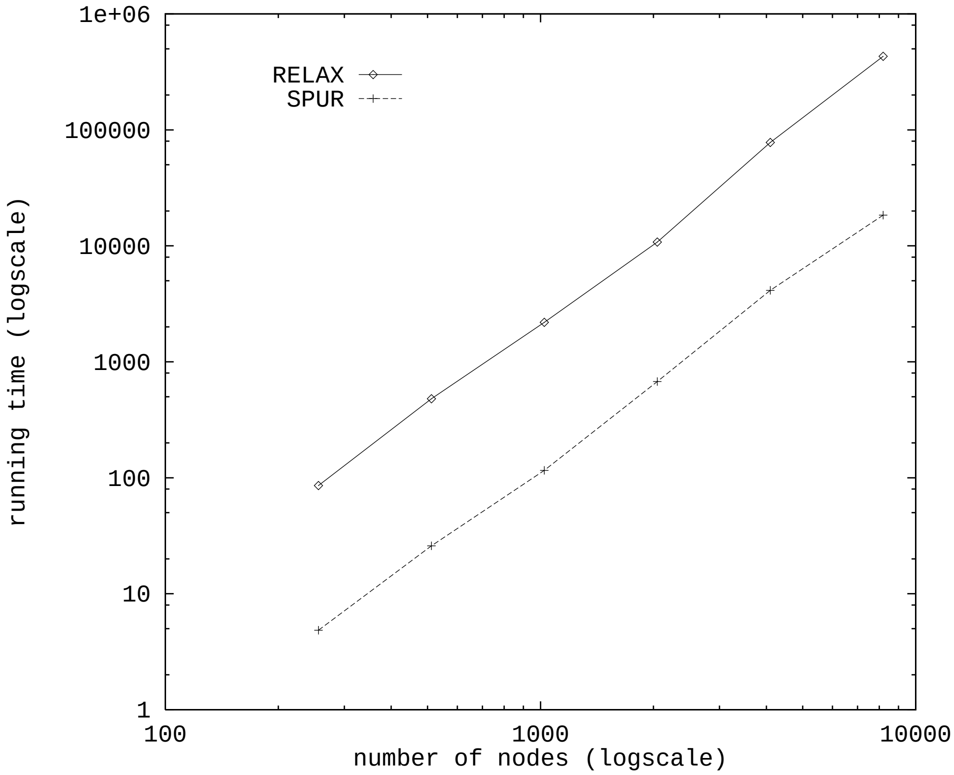
Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
256	2048	2.662	16.992	1	6.38
512	4096	10.358	77.783	1	7.51
1024	8192	38.904	260.958	1	6.71
2048	16384	205.767	1069.992	1	5.20
4096	32768	626.221	4404.150	1	7.03
8192	65536	3420.925	22173.180	1	6.48
16384	131072	9728.400	90664.312	1	9.32

FIGURE 7. Average running times (in seconds) of SPUR and RELAX on the GOTO-8 family.



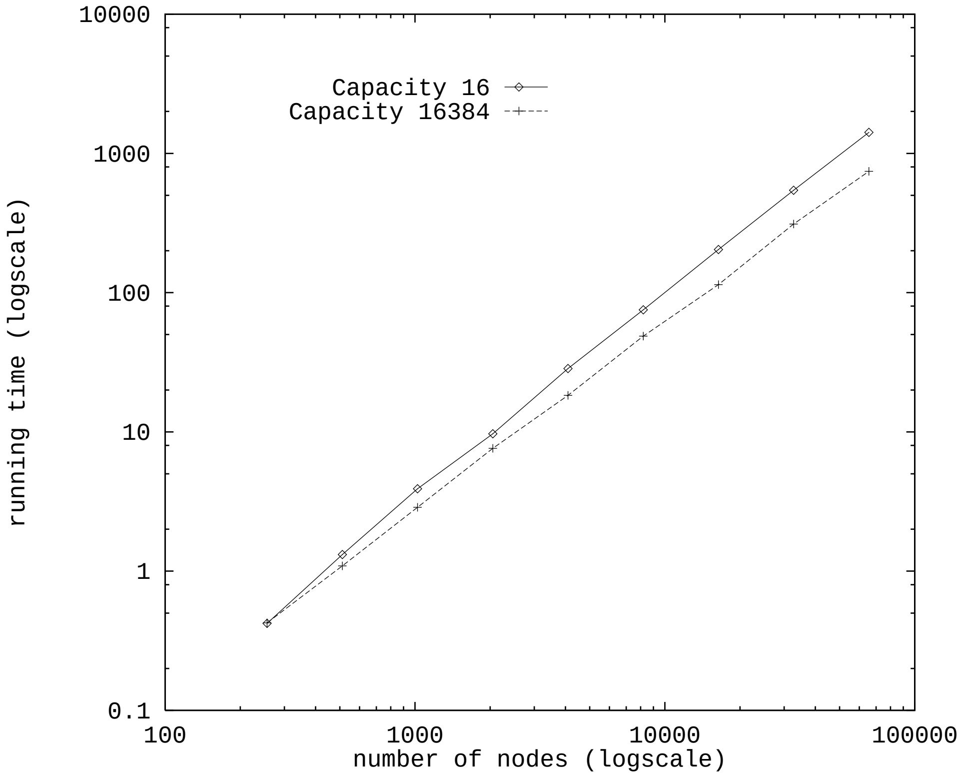
Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
256	4096	4.842	85.750	1	17.71
512	8192	20.525	322.671	1	15.72
1024	16384	76.037	1071.154	1	14.09
2048	32768	297.875	3501.650	1	11.76
4096	65536	1018.833	13950.066	1	13.69
8192	131072	3444.529	54075.504	1	15.70
16384	262144	18331.559	227889.172	1	12.43

FIGURE 8. Average running times (in seconds) of SPUR and RELAX on the GOTO-16 family.



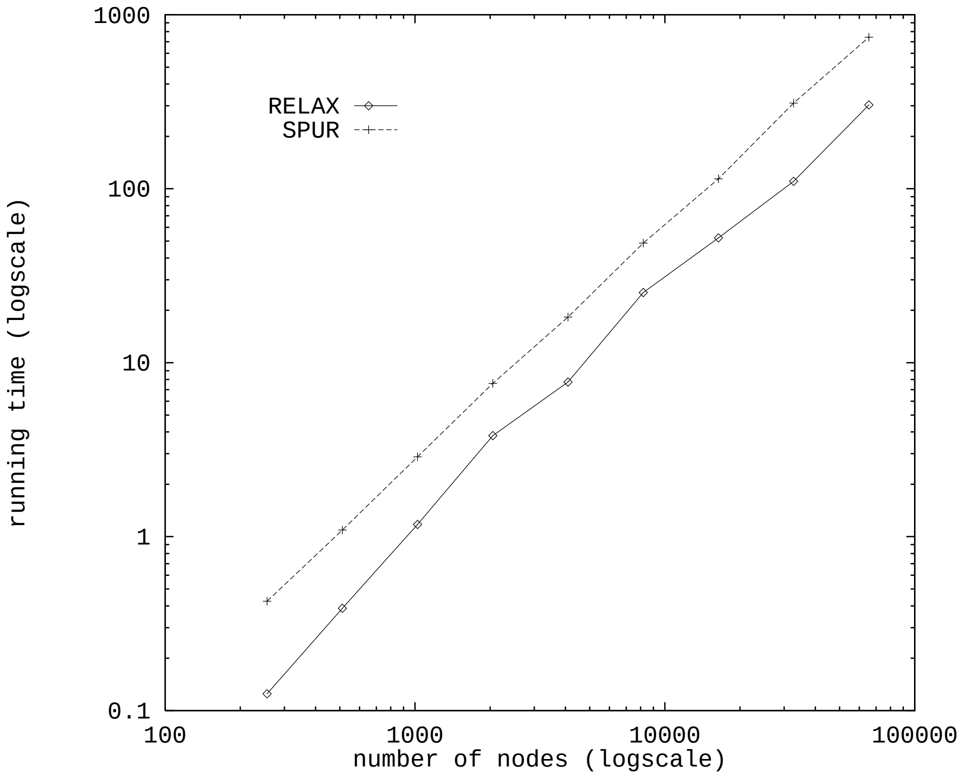
Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
256	4096	4.842	85.750	1	17.71
512	11585	25.904	479.763	1	18.52
1024	32768	116.033	2192.212	1	18.89
2048	92682	676.196	10766.146	1	15.92
4096	262144	4131.342	77956.250	1	18.87
8192	741455	18410.209	431195.250	1	23.42

FIGURE 9. Average running times (in seconds) of SPUR and RELAX on the GOTO-I family.



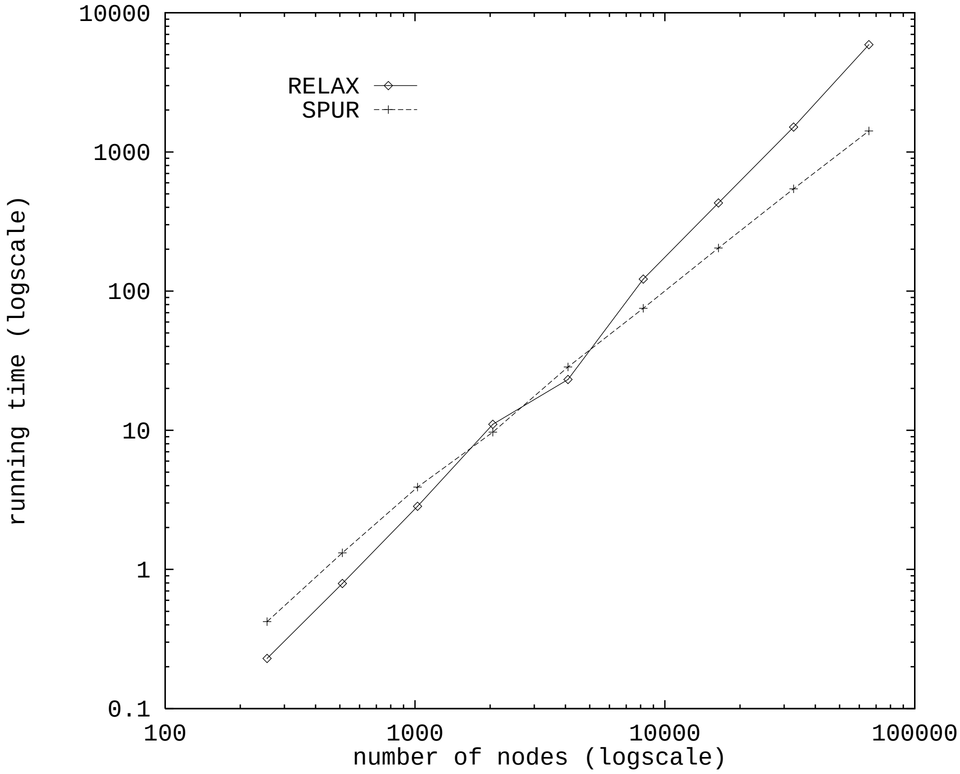
Problem size		SPUR running time	
nodes	edges	High Capacity	Low Capacity
256	2048	0.425	0.421
512	4104	1.092	1.316
1024	8200	2.875	3.904
2048	16418	7.613	9.683
4096	32854	18.300	28.508
8192	65714	48.750	75.188
16384	131458	114.096	204.054
32768	262892	310.858	543.808
65536	525830	743.075	1415.009

FIGURE 10. Average running times (in seconds) of SPUR on NETGEN families.



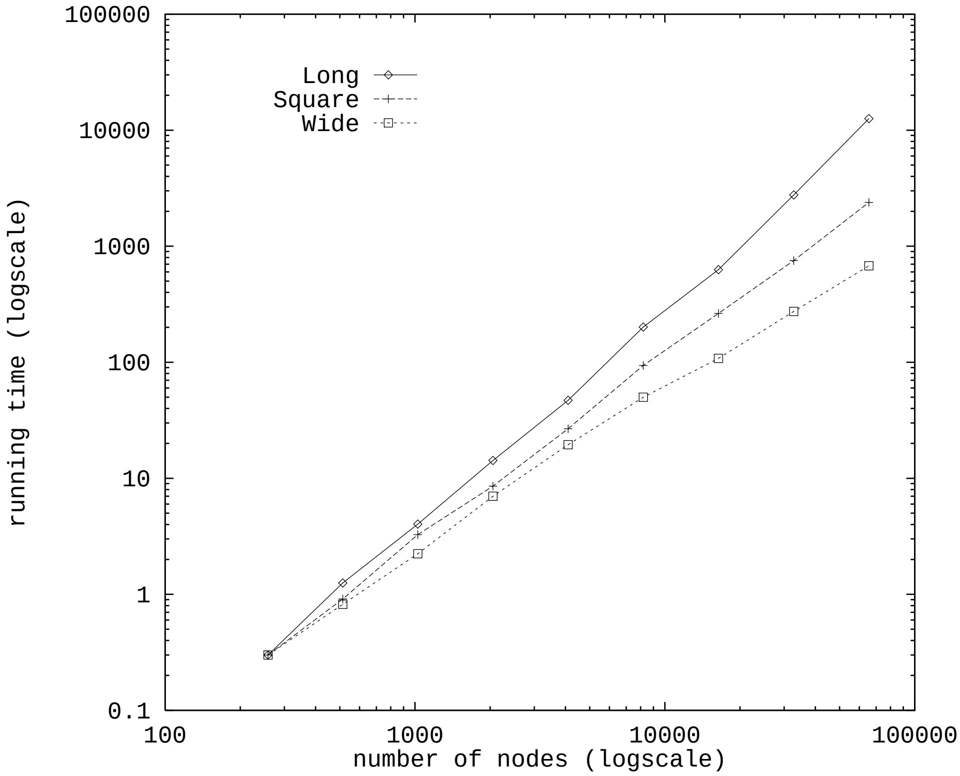
Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
256	2048	0.425	0.125	3.40	1
512	4104	1.092	0.387	2.82	1
1024	8200	2.875	1.175	2.45	1
2048	16418	7.613	3.808	2.00	1
4096	32854	18.300	7.738	2.37	1
8192	65714	48.750	25.312	1.93	1
16384	131458	114.096	52.167	2.19	1
32768	262892	310.858	110.292	2.82	1
65536	525830	743.075	302.992	2.45	1

FIGURE 11. Average running times (in seconds) of SPUR and RELAX on the NETGEN-HI family (maximum capacity 16384).



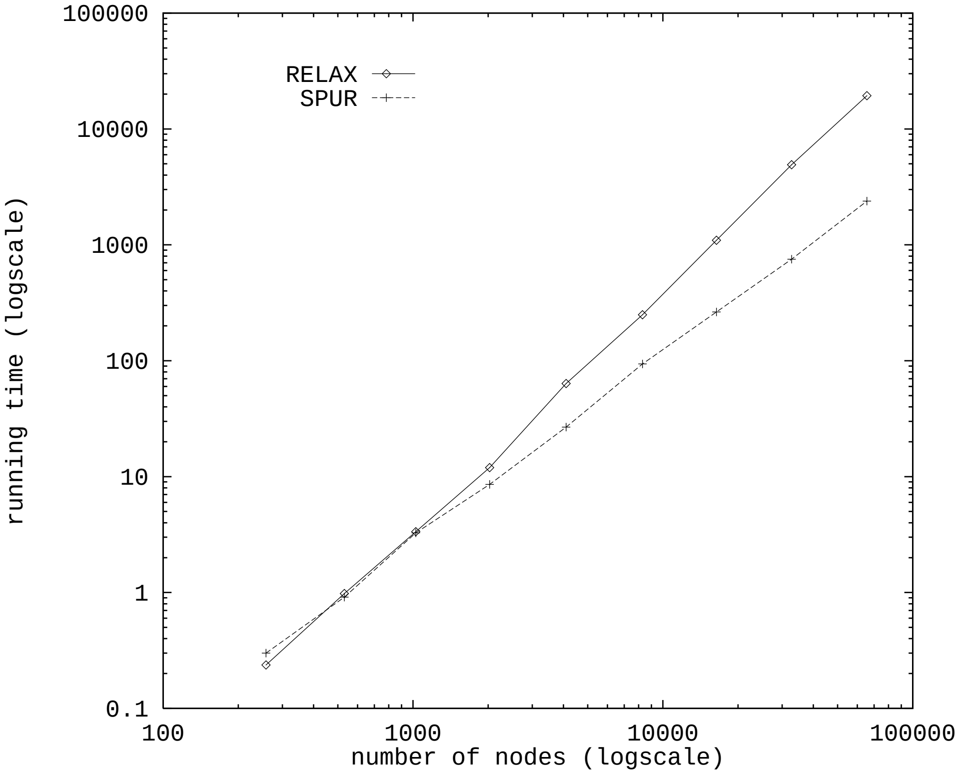
Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
256	2048	0.421	0.229	1.84	1
512	4104	1.316	0.792	1.66	1
1024	8200	3.904	2.842	1.37	1
2048	16418	9.683	11.050	1	1.14
4096	32854	28.508	23.183	1.23	1
8192	65714	75.188	122.146	1	1.62
16384	131458	204.054	429.888	1	2.11
32768	262892	543.808	1507.762	1	2.77
65536	525830	1415.009	5895.587	1	4.17

FIGURE 12. Average running times (in seconds) of SPUR and RELAX on the NETGEN-LO family (maximum capacity 16).



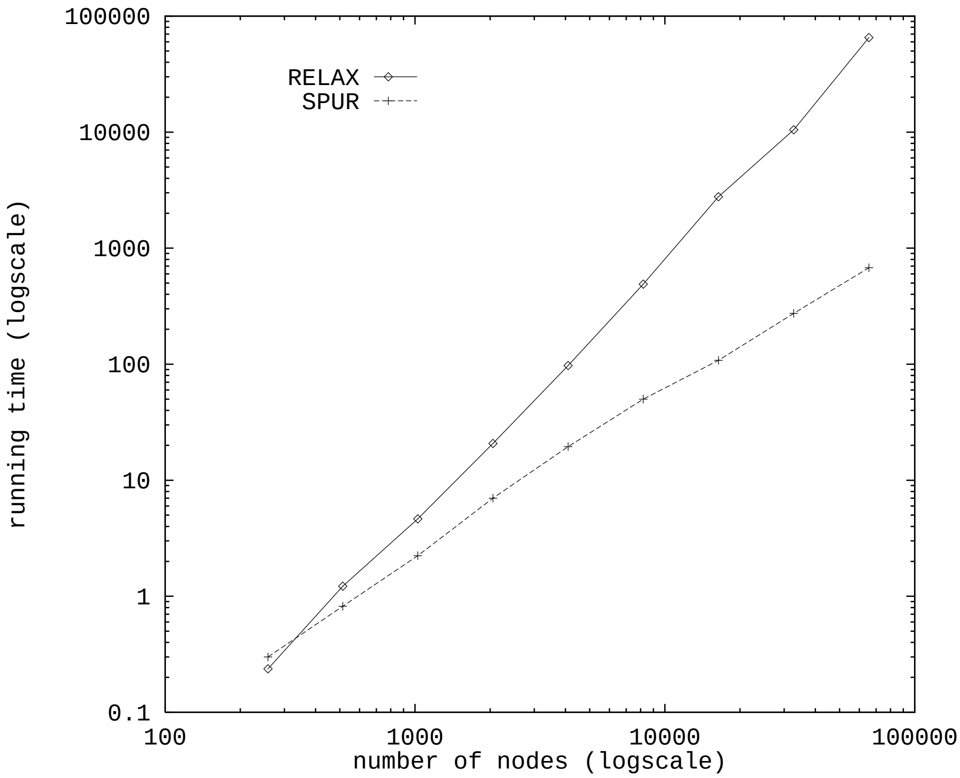
Problem size (nodes)	SPUR running time		
	Wide grid	Square grid	Long grid
258	0.300	0.300	0.300
514	0.821	0.913	1.254
1026	2.238	3.271	4.025
2050	7.008	8.575	14.254
4098	19.471	26.771	47.079
8194	49.938	93.729	201.288
16386	108.096	263.550	627.804
32770	274.504	751.346	2764.283
65538	677.912	2385.329	12577.959

FIGURE 13. Average running times (in seconds) of SPUR on GRIDGRAPH families.



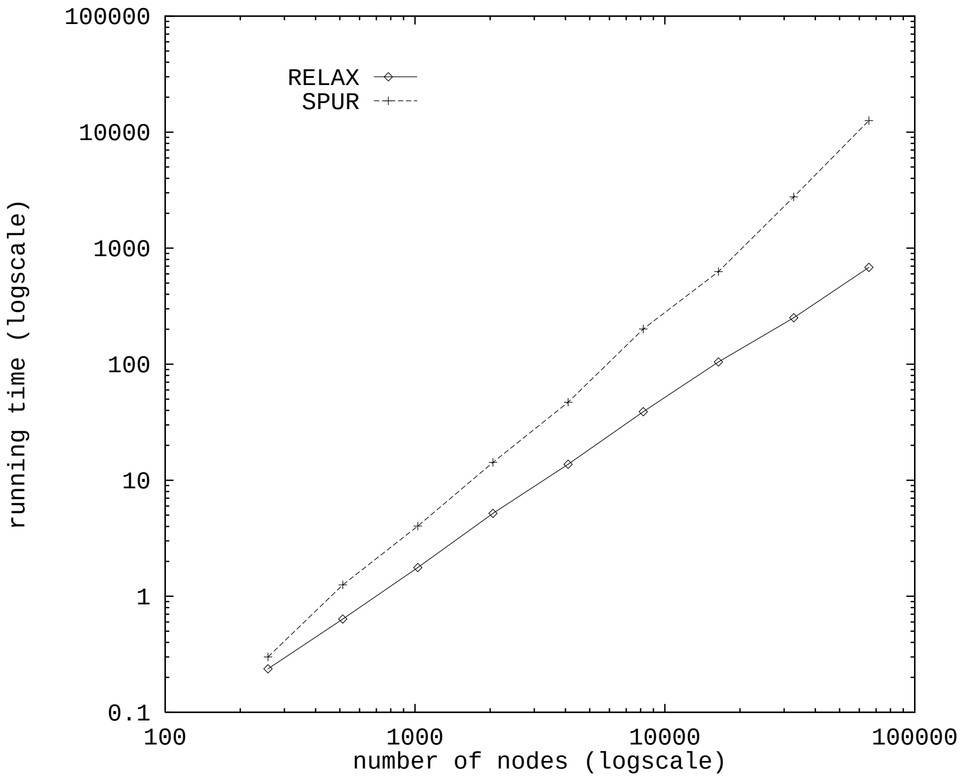
Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
258	512	0.300	0.237	1.26	1
531	1058	0.913	0.979	1	1.07
1026	2048	3.271	3.350	1	1.02
2027	4050	8.575	11.971	1	1.40
4098	8192	26.771	63.529	1	2.37
8283	16562	93.729	249.617	1	2.66
16386	32768	263.550	1094.171	1	4.15
32763	65522	751.346	4915.750	1	6.54
65538	131072	2385.329	19387.441	1	8.13

FIGURE 14. Average running times (in seconds) of SPUR and RELAX on the GRID-SQUARE family.



Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
258	512	0.300	0.237	1.26	1
514	1040	0.821	1.221	1	1.49
1026	2096	2.238	4.654	1	2.08
2050	4208	7.008	20.721	1	2.96
4098	8432	19.471	97.338	1	5.00
8194	16880	49.938	489.737	1	9.81
16386	33776	108.096	2772.962	1	25.65
32770	67568	274.504	10487.862	1	38.21
65538	135152	677.912	65298.406	1	96.32

FIGURE 15. Average running times (in seconds) of SPUR and RELAX on the GRID-WIDE family.



Problem size		Running time		Normalized time	
nodes	edges	SPUR	RELAX	SPUR	RELAX
258	512	0.300	0.237	1.26	1
514	1008	1.254	0.637	1.97	1
1026	2000	4.025	1.767	2.28	1
2050	3984	14.254	5.188	2.75	1
4098	7952	47.079	13.738	3.43	1
8194	15888	201.288	39.012	5.16	1
16386	31760	627.804	104.583	6.00	1
32770	63504	2764.283	250.975	11.01	1
65538	126992	12577.959	684.162	18.38	1

FIGURE 16. Average running times (in seconds) of SPUR and RELAX on the GRID-LONG family.

Problem	Running time		Normalized time	
	SPUR	RELAX	SPUR	RELAX
gte.20	0.050	0.017	2.99	1
gte.40	0.067	0.017	4.01	1
gte.60	0.067	0.017	4.01	1
gte.160	0.150	0.067	2.25	1
gte.200	0.100	0.233	1	2.33
gte.460	0.100	0.600	1	6.00
gte.510	0.100	0.817	1	8.17
gte.1160	0.100	1.333	1	13.33
gte.1700	0.133	0.450	1	3.38
gte.6410	0.150	4.000	1	26.67
gte.6830	0.100	7.333	1	73.33
gte.15100	0.133	0.033	3.99	1
gte.15710	0.150	0.067	2.25	1
gte.35620	0.133	33.400	1	251.13
gte.49320	0.150	22.850	1	152.33
gte.60090	0.150	58.600	1	390.67
gte.65330	0.150	0.767	1	5.11
gte.176280	0.133	0.050	2.66	1
gte.298300	0.100	0.450	1	4.50
gte.451760	0.150	67.750	1	451.67
gte.469010	0.133	70.717	1	531.70
gte.508829	0.133	97.017	1	729.45
Total	2.632	366.583	1	139.28

FIGURE 17. Running times (in seconds) of SPUR and RELAX on GTE instances. Each network consists of 49 nodes and 520 arcs.

The behavior of RELAX on these instances is different. On some of them it runs very fast, and on others extremely slow. The ratio between the slowest and fastest RELAX running times is over 5700.

The total running time of SPUR on the GTE instances is better than the total running time of RELAX by two orders of magnitude.

6.5. Discussion. Figure 18 compares the execution times of SPUR on GOTO-8 and NETGEN-HI families, which have the same maximum cost and maximum capacity values. The difference in running times for the same problem size is quite drastic. GOTO problems are much harder for SPUR than NETGEN problems. (For RELAX, the difference is even bigger.) NETGEN seems to produce relatively easy instances of the minimum-cost flow problem.

In general, practical running times vary from generator to generator. By observing the behavior of a program on different families of problems, one may gain insight into frequently occurring network structures that are bad for a particular implementation, suggesting directions for improvement.

For example, the behavior of SPUR on the GRID-LONG family shows an area where the algorithm needs to be improved. We discuss a possible reason for the relatively poor performance of our code on this family in Section 8.

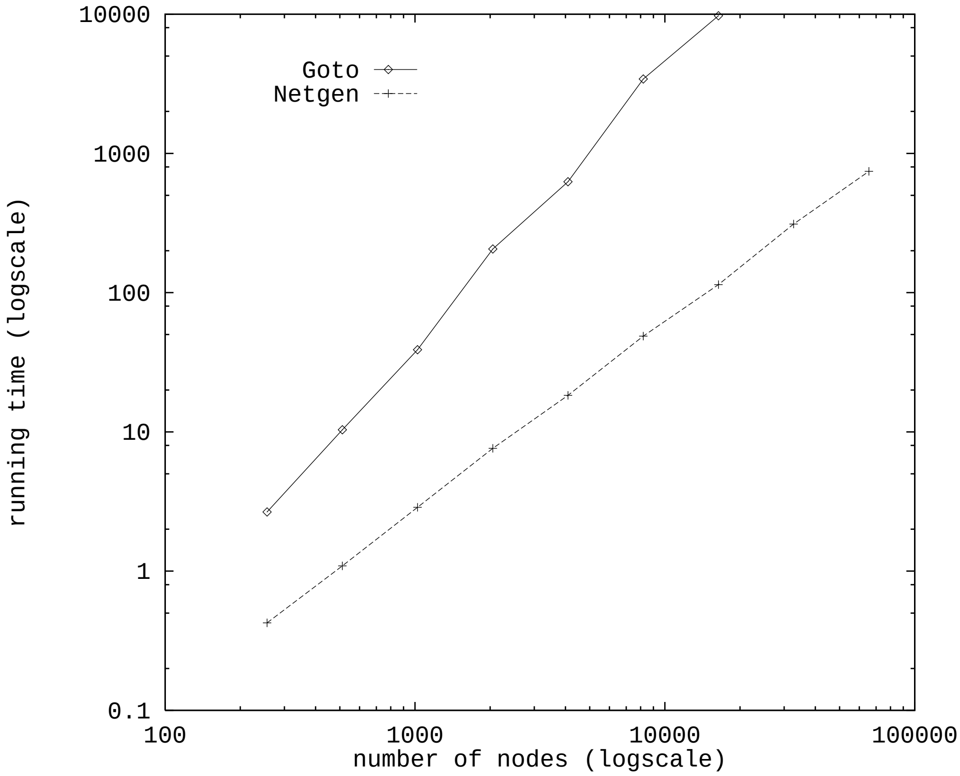
7. Effectiveness of Heuristics

In this section we examine the effect of heuristics on the algorithm's running time.

Our implementation spends most of its time doing *refine*. The percentage of the computation time spent in *refine* increases with the problem size, exceeding 95% on the larger problems we experimented with. The running time of *refine* depends on the number of *push* and *relabel* operations performed as well as on the speed of selecting and applying the operations. Heuristic improvements decrease both the number of operations and time per operation.

Figure 19 gives running times as well as operation counts for SPUR on problems from several classes we studied. We count the number of *refines* and the number of *pushes* and *relabels* per refine. One interesting observation here is that most of the time the number of *push* operations is smaller than the number of *relabel* operations. Since the latter operation is computationally more expensive, the number of relabels is the bottleneck for our implementation. In contrast, the theoretical worst-case bottleneck of the method is the number of pushes.

Figure 19 also gives the data for SPUR with all the heuristics turned off. Comparison of the two sets of data shows that the speedup due to heuristics is different for different problem types. For the problems in the Figure 19 the speedup varies from the factor of 2.88 on the GOTO-I instances to the factor of 6.5 on the NETGEN-HI instances. Reduction in the number of refines, reduction in the number of pushes and relabels per refine, and faster operation times all contribute to the speedup.



Problem size (nodes)	SPUR running time		
	GOTO-8 family	NETGEN-HI family	Ratio
256	2.662	0.425	6.26
512	10.358	1.092	9.49
1024	38.904	2.875	13.53
2048	205.767	7.613	27.03
4096	626.221	18.300	34.22
8192	3420.925	48.750	70.17
16384	9728.400	114.096	85.27
32768	timeout	310.858	?
65536	timeout	743.075	?

FIGURE 18. Average running times (in seconds) of SPUR on GOTO-8 and NETGEN-HI families.

Category	with heuristics	w/out heuristics	gain factor
Time (seconds)	626.221	1964.104	3.14
Number of Refines	7.750	12.000	1.55
Relabelings/Refine	1995966.125	2129125.000	1.07
Pushes/Refine	1724047.000	4268074.500	2.48

GOTO-8 examples with 4096 nodes and 32768 edges.

Category	with heuristics	w/out heuristics	gain factor
Time (seconds)	676.196	1947.979	2.88
Number of Refines	7.750	11.000	1.42
Relabelings/Refine	534075.062	482254.875	0.90
Pushes/Refine	535869.750	1111533.125	2.07

GOTO-I examples with 2048 nodes and 92682 edges.

Category	with heuristics	w/out heuristics	gain factor
Time (seconds)	743.075	4833.667	6.50
Number of Refines	9.000	13.000	1.44
Relabelings/Refine	3143826.500	4828418.500	1.54
Pushes/Refine	1883396.875	8488667.000	4.51

NETGEN-HI examples with 65536 nodes and 525830 edges.

Category	with heuristics	w/out heuristics	gain factor
Time (seconds)	751.346	4064.438	5.41
Number of Refines	7.000	13.000	1.86
Relabelings/Refine	6707662.500	9974990.000	1.49
Pushes/Refine	4905041.500	19180974.000	3.91

GRID-SQUARE examples with 32763 nodes and 65522 edges.

FIGURE 19. Effects of heuristics on SPUR's average running times and operation counts.

The only exception is the number of relabels per refine on the GOTO-I instances, where the number of relabels per refine in the full implementation is somewhat greater than in the implementation without heuristics.

Figures 20 – 22 summarize the effects of individual heuristics on the algorithm's performance. Note that the heuristics are interdependent: the speedup for the code that uses all the heuristics is less then the product of the speedups for individual heuristics.

As Figure 20 shows, early termination detection reduces the number of iterations of the algorithm. The corresponding improvement in the running time, however, is not directly proportional to the reduction in the number of iterations: sometimes it is noticeably more, sometimes noticeably less. This is probably due to the fact that the network structure at the end of the execution of the implementation without heuristics (when the flow is already optimal) is different from the structure at the beginning of the execution.

Arc fixing is aimed at reducing time per operation. As Figure 21 shows, the heuristic succeeds in this respect. An unexpected side-effect of this heuristic is reduction in the number of *push* and *relabel* operations on GRID-SQUARE examples. (On other examples in the table, the number of these operations does not change much.)

The lookahead heuristic is designed to reduce the number of *push* operations. As Figure 22 suggests, this heuristic also reduces the number of *relabel* operations, although by a smaller factor.

Figure 23 compares running times of the implementations with and without heuristics on the GRID-SQUARE family. This includes the implementation with no heuristics, the implementation with all heuristics, and three implementations wich a single heuristic each.

Figures 24, 25, and 26 compare the number of refines and the number of relabels and pushes per refine for the implementation with all the heuristics and the implementation with no heuristics. There is no significant asymptotic decrease in the number of relabels and pushes per refine due to heuristics. The speedup due to heuristics also does not grow significantly.

8. Conclusions and Further Research

Our implementation works well on a wide class of problems. The heuristics we use are important for the efficiency of our implementation. Although we studied several heuristics in the context of several variants of the push-relabel method, many more remain to be studied. Several directions for further study are suggested in Section 4.

The following example illustrates why SPUR is relatively slow on some classes of networks such as the GRID-LONG family. Consider a path of k nodes going from left to right. A unit of flow excess is located at the left end of the path and a unit

Category	early termination	no heuristics	gain factor
Time (seconds)	1373.833	1964.104	1.43
Number of Refines	7.750	12.000	1.55
Relabelings/Refine	2275481.250	2129125.000	0.94
Pushes/Refine	4830547.000	4268074.500	0.88

GOTO-8 examples with 4096 nodes and 32768 edges.

Category	early termination	no heuristics	gain factor
Time (seconds)	1620.292	1916.387	1.18
Number of Refines	8.000	11.000	1.38
Relabelings/Refine	556442.625	482254.875	0.87
Pushes/Refine	1364173.500	1111533.125	0.81

GOTO-I examples with 2048 nodes and 92682 edges.

Category	early termination	no heuristics	gain factor
Time (seconds)	2266.725	4833.667	2.13
Number of Refines	9.000	13.000	1.44
Relabelings/Refine	3139502.750	4828418.500	1.54
Pushes/Refine	5626424.000	8488667.000	1.51

NETGEN-HI examples with 65536 nodes and 525830 edges.

Category	early termination	no heuristics	gain factor
Time (seconds)	2451.208	4064.438	1.66
Number of Refines	7.500	13.000	1.73
Relabelings/Refine	10064268.000	9974990.000	0.99
Pushes/Refine	19821288.000	19180974.000	0.97

GRID-SQUARE examples with 32763 nodes and 65522 edges.

FIGURE 20. Effects of early termination on SPUR's average running times and operation counts.

Category	arc fixing	no heuristics	gain factor
Time (seconds)	985.912	1964.104	1.99
Number of Refines	12.000	12.000	1.00
Relabelings/Refine	2222590.750	2129125.000	0.96
Pushes/Refine	4429898.000	4268074.500	0.96

GOTO-8 examples with 4096 nodes and 32768 edges.

Category	arc fixing	no heuristics	gain factor
Time (seconds)	759.421	1916.387	2.52
Number of Refines	11.000	11.000	1.00
Relabelings/Refine	485542.094	482254.875	0.99
Pushes/Refine	1130211.625	1111533.125	0.98

GOTO-I examples with 2048 nodes and 92682 edges.

Category	arc fixing	no heuristics	gain factor
Time (seconds)	1665.158	4833.667	2.90
Number of Refines	13.000	13.000	1.00
Relabelings/Refine	4980536.000	4828418.500	0.97
Pushes/Refine	8754921.000	8488667.000	0.97

NETGEN-HI examples with 65536 nodes and 525830 edges.

Category	arc fixing	no heuristics	gain factor
Time (seconds)	2103.892	4064.438	1.93
Number of Refines	13.000	13.000	1.00
Relabelings/Refine	7264765.500	9974990.000	1.37
Pushes/Refine	14034406.000	19180974.000	1.37

GRID-SQUARE examples with 32763 nodes and 65522 edges.

FIGURE 21. Effects of arc fixing on SPUR's average running times and operation counts.

Category	lookahead	no heuristics	gain factor
Time (seconds)	1235.654	1964.104	1.59
Number of Refines	12.000	12.000	1.00
Relabelings/Refine	1539412.375	2129125.000	1.38
Pushes/Refine	1271935.500	4268074.500	3.36

GOTO-8 examples with 4096 nodes and 32768 edges.

Category	lookahead	no heuristics	gain factor
Time (seconds)	1684.221	1916.387	1.14
Number of Refines	11.000	11.000	1.00
Relabelings/Refine	439438.469	482254.875	1.10
Pushes/Refine	405540.375	1111533.125	2.74

GOTO-I examples with 2048 nodes and 92682 edges.

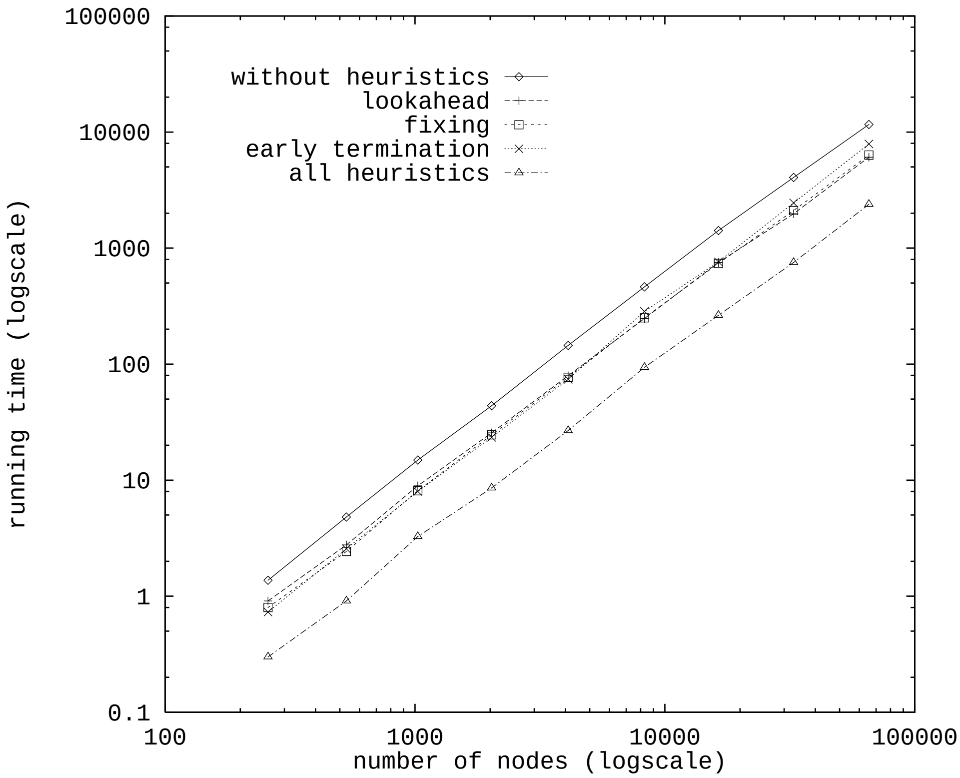
Category	lookahead	no heuristics	gain factor
Time (seconds)	3836.388	4833.667	1.26
Number of Refines	13.000	13.000	1.00
Relabelings/Refine	4615173.000	4828418.500	1.05
Pushes/Refine	2675020.250	8488667.000	3.17

NETGEN-HI examples with 65536 nodes and 525830 edges.

Category	lookahead	no heuristics	gain factor
Time (seconds)	1975.516	4064.438	2.06
Number of Refines	13.000	13.000	1.00
Relabelings/Refine	7177376.000	9974990.000	1.39
Pushes/Refine	5123041.000	19180974.000	3.74

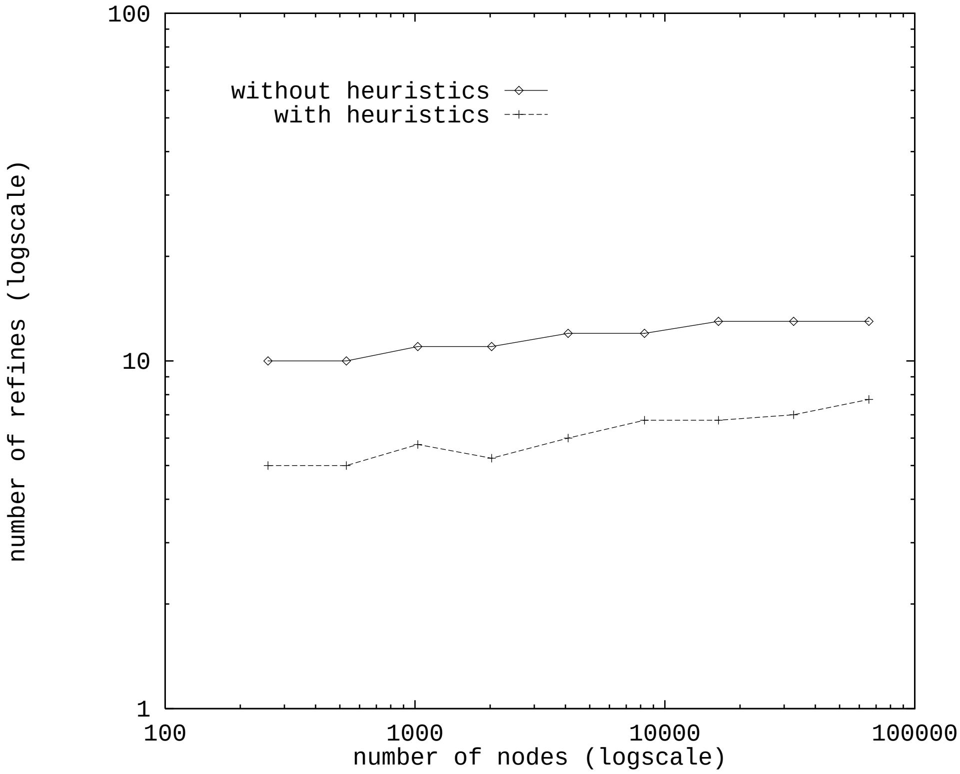
GRID-SQUARE examples with 32763 nodes and 65522 edges.

FIGURE 22. Effects of lookahead on SPUR's average running times and operation counts.



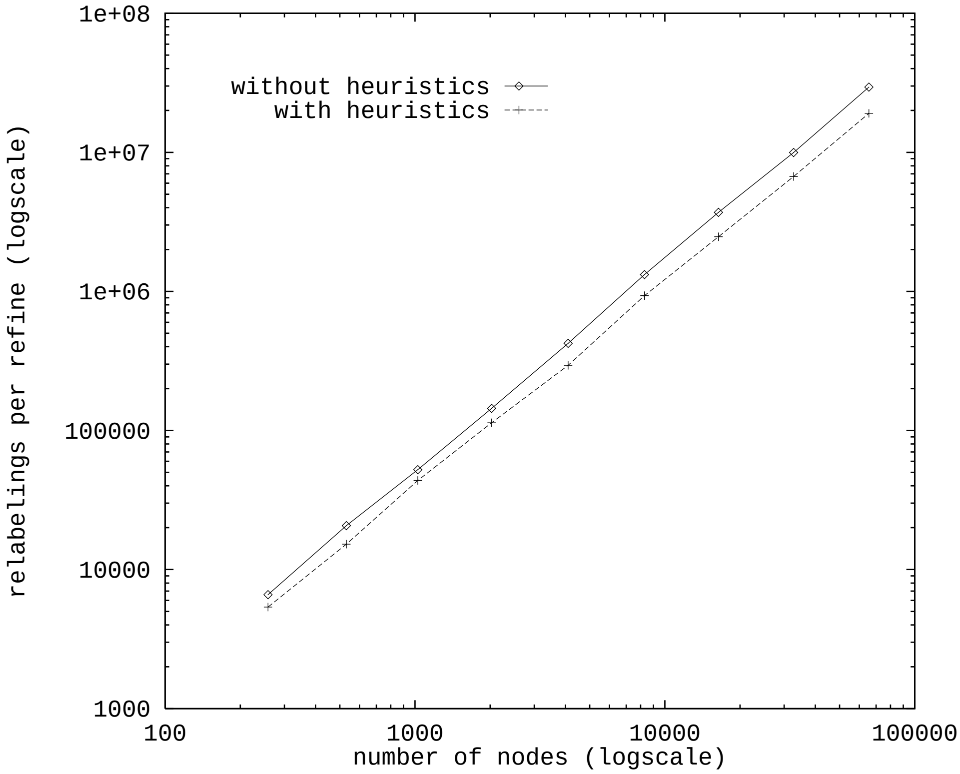
Problem size		Running Time					
		early			all	without	gain
nodes	edges	termination	fixing	lookahead	heuristics	heuristics	factor
258	512	0.73	0.80	0.91	0.30	1.37	4.55
531	1058	2.56	2.43	2.77	0.91	4.81	5.27
1026	2048	8.00	8.10	8.94	3.27	14.90	4.56
2027	4050	23.44	24.58	25.58	8.57	43.75	5.10
4098	8192	73.92	76.89	79.39	26.77	144.82	5.41
8283	16562	284.52	250.29	247.05	93.73	462.63	4.94
16386	32768	760.00	738.65	751.90	263.55	1414.87	5.37
32763	65522	2451.21	2103.89	1975.52	751.35	4064.44	5.41
65538	131072	7898.13	6344.48	6070.79	2385.33	11621.89	4.87

FIGURE 23. Effects of individual and combined heuristics on SPUR's running time on the Grid-Square family.



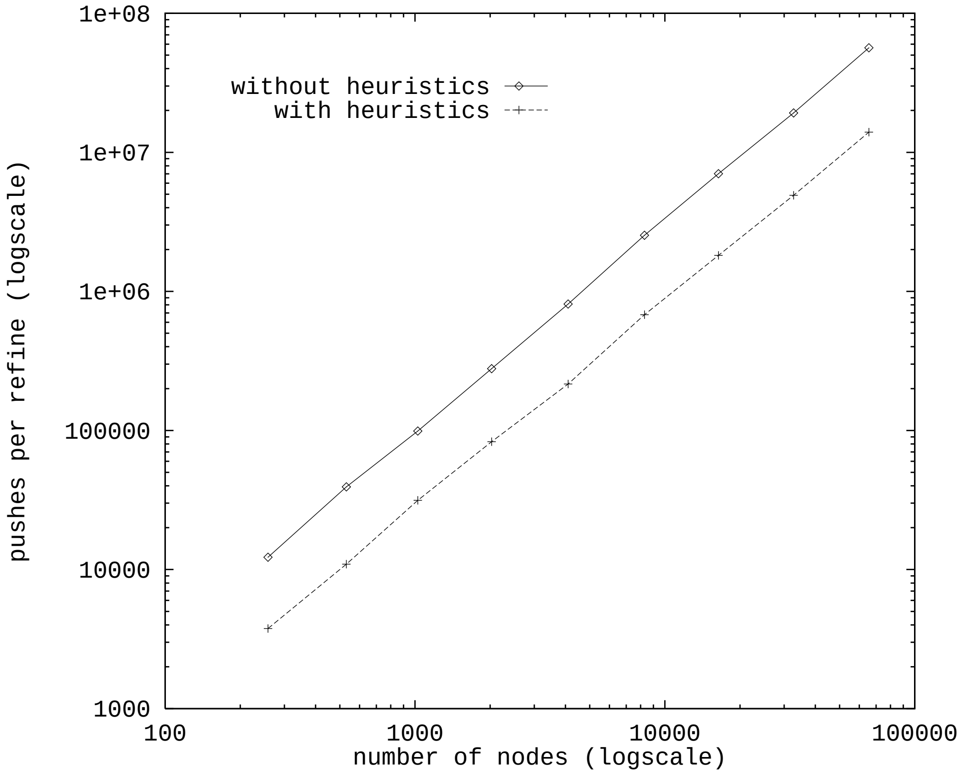
Problem size		Number of Refines		
		with	without	gain
nodes	edges	heuristics	heuristics	factor
258	512	5.000	10.000	2.00
531	1058	5.000	10.000	2.00
1026	2048	5.750	11.000	1.91
2027	4050	5.250	11.000	2.10
4098	8192	6.000	12.000	2.00
8283	16562	6.750	12.000	1.78
16386	32768	6.750	13.000	1.93
32763	65522	7.000	13.000	1.86
65538	131072	7.750	13.000	1.68

FIGURE 24. Effects of heuristics on the average number of refines done by SPUR on the Grid-Square family.



Problem size		Relabelings per Refine		
nodes	edges	with heuristics	without heuristics	gain factor
258	512	5366.500	6595.500	1.23
531	1058	15213.500	20686.775	1.36
1026	2048	43678.522	52273.727	1.20
2027	4050	113666.524	144165.636	1.27
4098	8192	294052.458	422782.938	1.44
8283	16562	931970.481	1319870.875	1.42
16386	32768	2474598.148	3700223.077	1.50
32763	65522	6707662.357	9974990.173	1.49
65538	131072	19079652.419	29456771.904	1.54

FIGURE 25. Effects of heuristics on the average number of relabels per each refine done by SPUR on the Grid-Square family.



Problem size		Pushes per Refine		
		with	without	gain
nodes	edges	heuristics	heuristics	factor
258	512	3766.100	12258.800	3.26
531	1058	10931.800	39324.400	3.60
1026	2048	31478.304	99049.045	3.15
2027	4050	83044.333	278346.864	3.35
4098	8192	216022.292	810585.167	3.75
8283	16562	679902.630	2535172.292	3.73
16386	32768	1815986.481	7017152.000	3.86
32763	65522	4905041.643	19180973.788	3.91
65538	131072	13972798.419	56414415.288	4.04

FIGURE 26. Effects of heuristics on the average number of pushes per each refine done by SPUR on the Grid-Square family.

of deficit at the right end. Suppose all arcs are directed from left to right and have capacities of one and reduced costs of 2ϵ. The number of relabels that occur during an execution of *refine* on such a network is $\Omega(k^2)$. Based on this example, one can imagine a variety of subnetworks that cause quadratic behavior of the algorithm. We call such kind of structure a *hill*.

Note that price update heuristics discussed in Section 4.3 are very effective in eliminating hills if applied at the right time. Some of these heuristics, however, may create potential hills.

Recently, the first author developed another implementation of the cost-scaling push-relabel method. This implementation uses price update and price refinement heuristics based on the *set-relabel* operation described in Section 4 and the scaling shortest path algorithm of [**13**]. The new implementation is superior to SPUR, and on many problem families the improvement is asymptotic. This implementation does not seem to suffer from the hill phenomena.

Performance of the new implementation on GRIDGRAPH examples depends very little on the grid shape. The implementation is faster then SPUR even on GRID-WIDE examples, on which SPUR works best. The improvement on the GRID-LONG examples is dramatic. On GRID-LONG examples with 2^{16} nodes, the average running time of the current code is 215 seconds, compared to 12578 seconds for SPUR — a factor of 58.5 improvement. The new code is 3.2 times faster then RELAX on these examples.

The new implementation is also substantially faster on the GOTO examples. On GOTO-8 examples with 2^{14} nodes, the average running time is 405 seconds, compared to 9728 seconds for SPUR — a factor of 24 improvement. (See Figure 7.) The new code is 224 times faster then RELAX on these examples.

The performance of the new implementation may improve even further. The results of the work related to this implementation will be reported in an upcoming paper.

Our comparison of SPUR and RELAX shows that neither code dominates. Limited experiments with a network simplex codes NETFLO [**23**] and RNET [**18**] indicate that the same can be said for all of these codes. The new implementation of the scaling method mentioned above, however, is superior to the other codes on most families of problems. Further improvement may be possible using some of the ideas suggested in this paper.

APPENDIX: the GOTO Generator

In this section we discuss the GOTO generator, which was developed by the first author. GOTO stands for **G**rid **O**n **TO**rus. We would like to note that although the initial stage of the example generation does use the concept of a grid embeded on torus, the output of the generator is much more complicated. The goal of this generator is to produce hard instances of the capacitated transshipment problem,

#	name	notation	restrictions
1	number of nodes	n	$n \geq 15$
2	number of arcs	m	$6n \leq m \leq n^{\frac{5}{3}}$
3	maximum capacity	*max-cap*	*max-cap* ≥ 8
4	maximum cost	*max-cost*	*max-cost* ≥ 8
5	seed		

FIGURE 27. GOTO generator parameters. All parameters are integers.

and in fact the problems produced by the generator are harder then those produced by other generators we experimented with.

The generator works as follows. Imagine an X by Y torus with a grid embedded on it. Each node of the torus with coordinates (a, b) is connected to nodes $(a + 1 \, mod \, X, b), \ldots, (a + x\text{-}degree \, mod \, X, b)$ in x direction and to nodes $(a, b + 1 \, mod \, Y), \ldots, (a, b + y\text{-}degree \, mod \, Y)$ in y direction.

The capacity of an x direction arc is chosen uniformly at random from the interval $[1, \lceil \frac{max\text{-}cap}{\alpha^{i-1}} \rceil]$, where i is the x direction distance from the tail to the head of the arc. Here *max-cap* is an input parameter and $alpha = max\text{-}cap^{\frac{1}{x\text{-}degree+2}}$. The cost of an x direction arc is chosen uniformly at random from the interval $[0, max\text{-}cost]$, where *max-cost* is an input parameter.

The capacity of an y direction arc is chosen uniformly at random from the interval $[1, max\text{-}cap]$. The cost of an y direction arc is chosen uniformly at random from the interval $[0, 8]$.

Then the arcs between nodes with x coordinates X and 1 are deleted, and the demand D is set to be equal to the sum of capacities of the deleted arcs. Node demands are set to zero, except for the *source node* $(1, 1)$ and the *sink node* (X, Y); the former node has a demand of $-D$ (supply of D), and the latter has a demand of D.

To assure feasibility, a hamiltonian path is added from the source to the sink. This path traverses nodes in the lexicographical order of their coordinates. The capacity of the arcs on this path is set to D, and the cost of the arcs is set to $\frac{max\text{-}cost}{Y}$. Note that capacity of the arcs on the path can exceed *max-cap*.

To make the generator easy to use, we reduced the number of input parameters as much as possible. The generator takes five parameters, described in Figure 27. The generator computes the values of X, Y, *x-degree*, and *y-degree* from the number of nodes n and the number of arcs m so that X is approximately $n^{\frac{2}{3}}$, Y is approximately $n^{\frac{1}{3}}$, We *x-degree* and *y-degree* so that $y\text{-}degree^2$ is approximately *y-degree*, and the sum of the two degrees is as large as the number of available arcs allows.

Because of the integrality, we usually have $X \times Y < n$, so some of the nodes are not used in the construction described above. We call these nodes *extra nodes*. Similarly, we have *extra* arcs. We set *x-degree* and *y-degree* to make sure that there are enough extra arcs for the construction below. The extra nodes are connected in a path from the source to the sink by arcs of small capacity and large cost. The remaining extra arcs are used to connect extra nodes to the regular nodes. The direction, head node, and destination node of these arcs are chosen uniformly at random from the appropriate set.

The examples produced by GOTO have the following properties. On one hand, the range of costs and capacities of paths from the source to the sink is large. On the other hand, for any given path, there are usually many paths whose costs and capacities are close to those of the given path. A similar statement holds for the cuts between the source and the sink. We believe that these properties account for the difficulty of the problems generated by GOTO.

Acknowledgment

This work was done as a part of the First DIMACS International Algorithm Implementation Challenge, organized by Mike Grigoriadis, David Johnson, Cathy McGeoch, Clyde Monma, and Bob Tarjan. We would like to thank Bob Bland, Joseph Cheriyan, David Jensen, and Laśzlo Ladańyi for helpful discussions and assistance, and for the permission to run experiments on Cornell computers. We are also thankful to Robert Kennedy and Andrew Purshottam for comments on a draft of this paper, and to the International Computer Science Institute for providing the second author with office and computing support.

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. 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).
4. D. P. Bertsekas and P. Tseng. Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems. *Oper. Res.*, 36:93–114, 1988.
5. R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladańyi. Personal communication. 1991.
6. R. G. Bland and D. L. Jensen. On the Computational Behavior of a Polynomial-Time Network Flow Algorithm. *Math. Prog.*, 54:1–41, 1992.
7. B. V. Cherkassky. Personal communication. 1991.
8. 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.
9. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.
10. 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.
11. A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.
12. 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).

13. A. V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. Technical Report STAN-CS-92-1429, Department of Computer Science, Stanford University, 1992.
14. 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.
15. 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.
16. 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.
17. M. D. Grigoriadis. An Efficient Implementation of the Network Simplex Method. *Math. Prog. Study*, 26:83–111, 1986.
18. M. D. Grigoriadis. Personal communication. 1988.
19. R. Hassin. The Minimum-Cost Flow Problem: A Unifying Approach to Dual Algorithms and a New Tree-Search Algorithm. *Math. Prog.*, 25:228–239, 1983.
20. 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.
21. R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Math.*, 23:309–311, 1978.
22. R. M. Karp and J. B. Orlin. Parametric Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Math.*, 3:37–45, 1981.
23. J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming.* John Wiley and Sons, 1980.
24. 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.
25. T. Leong, P. Shor, and C. Stein. Implementation of a Combinatorial Multicommodity Flow Algorithm. The First DIMACS International Implementation Challenge, 1991.
26. M. G. C. Resende and G. Veiga. Computational Investigation of an Interior Point Linear Programming Algorithm for Minimum Cost Network Flows. 1991.
27. H. Röck. Scaling Techniques for Minimal Cost Network Flows. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 181–191. Carl Hansen, Münich, 1980.
28. N. Schienker and R. E. Tarjan. Personal communication. 1990.
29. É. Tardos. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, 5(3):247–255, 1985.
30. J. Wein. Personal communication. 1990.

Andrew V. Goldberg, Computer Science Department, Stanford University, Stanford, CA 94305.
E-mail address: goldberg@cs.stanford.edu

Michael Kharitonov, Computer Science Department, Stanford University, Stanford, CA 94305.
E-mail address: misha@theory.stanford.edu