Efficient Implementation of the Goldberg-Tarjan Minimum-Cost Flow Algorithm

Ursula Bünnagel Bernhard Korte Jens Vygen Research Institute for Discrete Mathematics, University of Bonn

dedicated to Masao Iri on the occasion of his 65th birthday

Abstract

The cost—scaling algorithm of Goldberg and Tarjan [9] is known to be one of the most efficient algorithms for minimum—cost flow problems. However, its efficiency in practice depends on many implementation aspects. Moreover, the inclusion of several heuristics improves its performance drastically. This paper addresses important implementation aspects and describes the most efficient heuristics. Experimental results also highlight the effect of combining several heuristics.

Keywords: Minimum-cost flows, Goldberg-Tarjan algorithm, efficient implementation

1 Introduction

We assume familiarity with network flow theory and use standard results and notation; see e.g. Ahuja, Magnanti, Orlin [1]. Here we cite only the most important ones. The minimum-cost flow problem is defined as follows: Given a digraph G with capacities $u: E(G) \to \mathbb{R}_+$, edge weights $c: E(G) \to \mathbb{R}$ and balance values $b: V(G) \to \mathbb{R}$, find a b-flow of minimum cost, i.e. a function $f: E(G) \to \mathbb{R}_+$ with $f(e) \leq u(e)$ for all $e \in E(G)$ and

$$\sum_{e \in \delta^+_G(v)} f(e) - \sum_{e \in \delta^-_G(v)} f(e) \ = \ b(v)$$

for all $v \in V(G)$ such that $\sum_{e \in E(G)} f(e)c(e)$ is minimum. Here $\delta_G^-(v)$ resp. $\delta_G^+(v)$ denotes the set of edges entering resp. leaving vertex v.

It is well–known that feasibility of a minimum–cost flow problem can be checked (and in the positive case a b–flow can be found) via a max–flow computation. The maximum flow problem is easier and can be solved much faster than a minimum–cost flow problem.

Given an instance (G, u, c, b) of the minimum—cost flow problem and a flow $f : E(G) \to \mathbb{R}_+$ with $f(e) \leq u(e)$ for all $e \in E(G)$, the residual capacity of an edge $e \in E(G)$ is defined by $u_f(e) := u(e) - f(e)$. Moreover, we consider a new edge e = (v, w) and define $u_f(e) := f(e)$ and c(e) := -c(e). The residual graph G_f consists of

all (old and new) edges with positive residual capacity. It is well known that a b-flow f is optimal (has minimum cost) if and only if no directed circuit in G_f has negative total weight.

Given a digraph G with edge weights $c: E(G) \to \mathbb{R}$, a feasible potential in (G,c) is a function $\pi: V(G) \to \mathbb{R}$ such that the reduced cost $c_{\pi}(e) := c(e) + \pi(v) - \pi(w)$ is nonnegative for each $e = (v, w) \in E(G)$. A feasible potential exists if and only if no directed circuit has negative total weight. This can be decided (and in the positive case a feasible potential can be found) by a single—source shortest path computation.

Given a flow $f: E(G) \to \mathbb{R}_+$ with $f(e) \leq u(e)$ and a potential $\pi: V(G) \to \mathbb{R}$, we define the admissable graph to consist of all edges of G_f which have negative reduced cost. We denote the admissable graph by G_f^{π} . If f is an optimal b-flow (for some b) and π is a feasible potential in G_f then G_f^{π} is acyclic.

2 The algorithm of Goldberg and Tarjan

A b-flow f is called ϵ -optimal if there is a potential $\pi: V(G) \to \mathbb{R}$ such that $c_{\pi}(e) \geq -\epsilon$ for all $e \in E(G_f)$. So 0-optimal is nothing but optimal. The cost-scaling algorithm of Goldberg and Tarjan [9] computes a b-flow which is ϵ -optimal, for some large ϵ . Then it reduces the value of ϵ in each iteration, preserving ϵ -optimality.

It is easy to see that any b-flow is ϵ -optimal if $\epsilon \ge \max_{e \in E(G)} |c(e)|$. Moreover, in the case of integral costs, any ϵ -optimal flow with $\epsilon < \frac{1}{n}$ is optimal. (For the easy proofs, see [9], [1] or [2].) So the global structure of the algorithm is as follows (α is a fixed parameter; it should be at least 2).

```
\begin{array}{ll} \mathbf{procedure} \ minimum\_cost\_flow(G,\ u,\ c,\ b) \\ \mathbf{begin} \\ \pi(v) := 0 \quad \forall \ v \in V(G) \ \text{and} \ \epsilon := \max_{e \in E(G)} |c(e)|; \\ \text{Determine a $b$-flow $f$. If no $b$-flow exists, } \mathbf{return}(null); \\ \mathbf{while} \ \epsilon \geq 1/n \ \mathbf{do} \\ \epsilon := \epsilon/\alpha; \\ refine(\epsilon,\ f,\ \pi); \\ \mathbf{end}; \\ \mathbf{return}(f); \\ \mathbf{end}; \end{array}
```

The subroutine *refine* proceeds as follows: First the maximum possible amount of flow is pushed along all edges with negative reduced cost. This results in an optimal b'-flow for some b'. A vertex is called *active* if its *excess*

$$e_f(v) := b(v) - \sum_{e \in \delta_G^+(v)} f(e) + \sum_{e \in \delta_G^-(v)} f(e)$$

is positive. As long as there is an active vertex, either a *relabel* or a *push* is performed at an active vertex.

```
procedure refine(\epsilon, f, \pi)
begin
   for e \in E(G_f^{\pi}) do push(e, u_f(e));
   while there exists an active vertex do
   begin
      choose an active vertex v;
      if G_f^{\pi} contains an edge e = (v, w) then
          push(e, min(u_f(e), e_f(v)));
          relabel(v);
   end;
end;
procedure push(e, \delta)
begin
   if e = (v, w) \in E(G) then
      increase f(e) by \delta;
      decrease f(\stackrel{\leftarrow}{e}) by \delta;
end;
procedure relabel(v)
begin
   replace \pi(v) by \max_{(v,w)\in E(G_f)}(\pi(w)-c(v,w)-\epsilon);
```

Each execution of refine uses $O(n^2)$ relabel and $O(n^2m)$ push operations, where n := |V(G)| and m := |E(G)|. Proofs are given by [9], [10], [1] and [2]. We remark that the theoretical running time of refine can be improved to $O(nm\log\frac{n^2}{m})$ by the use of dynamic trees.

For computing the initial b-flow, the max-flow algorithm of Goldberg and Tarjan [8] (see also [10]) is used. In all our experiments only a small portion of the running time was spent for this initial step.

3 Implementation aspects

A first decision when implementing the above algorithm concerns the residual graph: It is not necessary to store the residual graph explicitly. However, if one does without, one has to check all edges incident to an active vertex before one can relabel it. This turns out to be very expensive. Therefore we decided to store the residual graph explicitly, even though it has to be updated after each *push*.

The residual capacity is stored and updated at each edge. This makes it unnecessary to store the flow. In fact, the flow is not needed explicitly except when returning the output after termination of the algorithm. For each vertex v, its excess $e_f(v)$ and its potential $\pi(v)$ are stored. The reduced costs are not stored explicitly – it proved more efficient to recompute them whenever they are needed.

For each vertex, a list of leaving edges with positive residual capacity is available. We have a pointer which marks the current element of this list, for each active vertex. Whenever a push saturates the current edge (or when the algorithm ascertains that the current edge has nonnegative reduced cost), the pointer moves forward. This implies that a relabel can be performed when the pointer reaches the end of the list.

An important implementation issue is the choice of an active vertex. Since the number of active vertices is usually quite small, it is essential to keep a data structure which contains the set of active vertices. Simple data structures are a queue or a stack. A more complicated way is to store the active vertices in a topological order with respect to the admissable graph (which is acyclic). However, in practice the number of *push* and *relabel* operations seems to be slightly larger than with the simple queue implementation. Moreover, the selection of the first vertex in the topological order is more expensive – even when using the most efficient data structure, Fibonacci heaps [5]). These considerations led us to the decision that the active vertices are stored in a simple queue.

Note that this approach reduces the theoretical bound on the number of push operations to $O(n^3)$ per execution of refine ([9])

4 Heuristics

4.1 Price refinement

The procedure refine computes an ϵ -optimal b-flow. However, it happens quite often that this flow is already ϵ' -optimal for some $\epsilon' < \epsilon$. This might allow us to skip some iterations without executing refine. So two questions arise:

- 1. Given a b-flow f and an $\epsilon \geq 0$, find a potential $\pi : V(G) \to \mathbb{R}$ such that $c_{\pi}(e) \geq -\epsilon$ for all $e \in E(G_f)$, or decide that f is not ϵ -optimal.
- 2. Given a b-flow f, find the smallest $\epsilon \geq 0$ such that f is ϵ -optimal.

The first problem can be solved by a single–source shortest path computation in (G_f, c') , where c' is obtained from c by adding ϵ to each edge weight. Goldberg [6] suggested to do such a shortest path computation in each iteration, i.e. check whether the current b-flow is $\frac{\epsilon}{\alpha}$ -optimal, and call *refine* only in the negative case.

Another heuristic is implied by the solution to the second problem. This possibility has been mentioned, but not pursued further, by Goldberg and Tarjan [9] and Goldberg, Tardos and Tarjan [10]. Observe that f is ϵ -optimal if there is no directed circuit of negative total weight in (G_f, c') , where $c'(e) := c(e) + \epsilon$ for all $e \in E(G_f)$. Hence the smallest ϵ such that f is ϵ -optimal is $-\mu$, where μ is the minimum mean weight of a directed circuit in (G_f, c) .

Hence the second problem is nothing but a minimum mean cycle problem. This can be solved very efficiently by a parametric shortest path algorithm. A basic version has been described by Karp and Orlin [14] and Schneider and Schneider [16]; an improved version is due to Young, Tarjan and Orlin [18]. An efficient implementation of this algorithm is described by Bünnagel [2]. The worst case running time is $O(nm + n^2 \log n)$.

In our implementation of the minimum–cost flow algorithm, we solve the second problem by a parametric shortest path computation in each iteration. We find the minimum ϵ such that the current b-flow is ϵ -optimal, and we also find a suitable potential π . We then proceed with refine applied to f, π and $\frac{\epsilon}{\alpha}$.

This heuristic led to a significant improvement of the running time. Compared to just checking whether the flow is $\frac{\epsilon}{\alpha}$ -optimal by a simple shortest path computation, we observed an improvement of the running time by 40%.

Another question is how to choose the parameter α . Theoretical considerations suggest a value of 4 (see [2]), for computation results see Section 5.

4.2 Push-look-ahead

Another heuristic called push-look-ahead is due to Goldberg [6]. Its aim is to avoid pushing flow from a vertex v to a neighbour w if this flow is likely to be pushed back to v in a subsequent step. We modify the algorithm and push no more than

$$a_f(w) := \max \left\{ 0, -e_f(w) + \sum_{e \in \delta_{G_f^{\pi}}^+} u_f(e) \right\}$$

units of flow to vertex w. If this number is positive, it is the amount of flow which can be pushed away from w without relabeling.

It is necessary to extend the relabel procedure. If we push $a_f(w) < \min\{u_f(e), e_f(v)\}$ units from v to w, vertex w is marked as hyperactive. No further flow can be pushed into w until w has been relabeled. Moreover, v cannot be relabeled before w has been relabeled. However, if $a_f(w) > 0$, w will not be active after all possible pushs from w have been done. This shows that it is necessary to extend the relabel operation to hyperactive vertices, regardless of whether they are active or not.

Goldberg and Kharitonov [7] suggest to apply the *relabel* procedure as above to hyperactive vertices as well, with one difference: In the case when no residual edge leaves the relabelled vertex, the potential is reduced by ϵ . We implemented the following different relabel procedure:

```
procedure relabel(v)
begin
if e_f(v) = 0 then
decrease \pi(v) by \epsilon;
else
replace \pi(v) by \max_{(v,w) \in E(G_f)}(\pi(w) - c(v,w) - \epsilon);
end;
```

The reason for our choice of the extended *relabel* procedure is that this enabled us to bound the maximum absolute value of the potentials; see Bünnagel [2]. This can be important for numerical reasons.

During the examination of a hyperactive vertex v, another vertex w might become hyperactive. By putting the hyperactive vertices on a stack and always considering the most

recently added hyperactive vertex, they can be examined in the correct order. (Note that the admissable graph is acyclic.) The following recursive procedure reflects this:

```
procedure refine(\epsilon, f, \pi)
begin
  for e \in E(G_f^{\pi}) do push(e, u_f(e));
  while there exists an active vertex do
  begin
      choose an active vertex v;
      discharge(v);
  end;
end;
procedure discharge(v)
begin
   while e_f(v) > 0 or v is hyperactive do
      if G_f^{\pi} contains an edge e=(v,w) then
         push(e, \min(u_f(e), e_f(v), a_f(w)));
         if a_f(w) < min(u_f(e), e_f(v)) then
         begin
            \max w as hyperactive;
            discharge(w);
         end:
      end;
      else
      begin
         relabel(v);
         if v is hyperactive then
         begin
            \max v as non-hyperactive;
            return;
         end;
      end;
  end;
end:
```

The number of relabel operations applied to each vertex can still be bounded by $(\alpha + 1)n$. In practice the number of relabel operations remains about the same, while the number of push operations decreases by 40–60%. Moreover, the flow computed by the new refine procedure is often better, i.e. ϵ' -optimal for a smaller ϵ' . Hence the combination with the price refinement heuristic described in the previous section is very efficient.

4.3 Set-relabel

The third important heuristic (Goldberg [6], Goldberg and Kharitonov [7]) extends the relabel operation such that many vertices are relabeled in one step. Let S be a subset of

vertices containing all vertices with negative excess. Let the complement $V(G) \setminus S$ contain at least one active vertex, and suppose that there is no edge in the admissable graph from $V(G) \setminus S$ to S. Then the set–relabel operation is applicable: The potential of all vertices in $V(G) \setminus S$ can be reduced by ϵ .

It is easy to see that this operation does not destroy the ϵ -optimality of f. The theoretical bounds on the number of *push* and *relabel* operations (see above) do not change if set-relabel operations are applied, no matter how often (see e.g. [2] for a proof).

The set—relabel operation can of course be iterated:

```
procedure relabel\_global(\pi)
begin S:=\{v\in V(G): e_f(v)<0\};
while V(G)\setminus S contains an active vertex do
begin Add all vertices to S from which S can be reached in G_f^\pi;
if V(G)\setminus S contains an active vertex then \pi(v):=\pi(v)-\epsilon for all v\in V(G)\setminus S;
end;
```

This procedure can be implemented efficiently using a bucket structure. The buckets are indexed by nonnegative integers. A bucket number B(v) is assigned to each vertex v, denoting the bucket containing v, or ∞ if v is not yet contained in any bucket. At the end B(v) will be the number of set—relabel operations which are applied to vertex v.

One first puts all vertices with negative excess into bucket 0, all other buckets are empty. Then the buckets are scanned: For each bucket i (i = 0, 1, 2, ...) and each vertex v with B(v) = i and each edge $e = (w, v) \in G_f$ with $i + 1 + \left\lfloor \frac{c_{\pi}(e)}{\epsilon} \right\rfloor < B(w)$ vertex w is deleted from bucket B(w) and inserted into bucket $i + 1 + \left\lfloor \frac{c_{\pi}(e)}{\epsilon} \right\rfloor$. Let k be the maximum bucket index of an active vertex (the above procedure can be stopped when i = k). Then the potential of each vertex v is reduced by $\epsilon \cdot \min(B(v), k)$.

This yields an O(m) running time of the procedure $relabel_global$. If it is used after every n ordinary relabel operations, the worst–case running time of refine does not change. The number of set–relabel operations (the number k above) during $relabel_global$ is O(n). So the total number of set–relabel operations during refine is $O(n^2)$. Goldberg [6] and Goldberg and Kharitonov [7] claim that in fact each node participates in O(n) relabels and set–relabels per refine. However, this can only be true if there are only O(n) set–relabels per refine, because isolated vertices participate in each set–relabel.

Empirically, the set—relabel heuristic reduces the number of (local) relabel operations by a factor of almost 2 in most cases, even when combining it with the other heuristics (price refinement and push—look—ahead). The additional work for the procedure *relabel_global* is significant, but in most cases the overall running time is reduced (as the next section shows).

5 Experimental results

With the exception of Figure 8 the experimental results have been obtained on IBM RS/6000 Mod. 550 workstations. Two types of instances have been dealt with:

The first group has been generated by NETGEN, a tool developed by Klingman, Napier and Stutz [15] and used at the First DIMACS Implementation Challenge [13]. The generator first produces sources and sinks and a sparse network consisting of edges with high capacity, in order to ensure feasibility. It then adds edges randomly. The group consists of 35 networks with 200 up to 8000 verteices. They come from a set of 40 benchmark problems which are available together with the generator (ftp://dimacs.rutgers.edu). The other five problems are assignment problems.

The second group consists of problems arising in the placement phase in VLSI layout (see [17]). These networks are very sparse; the number of edges is only 2–3 times the number of vertices. The group consists of 570 networks; the number of vertices is between 10 and 1000.

Since there are only slight differences in the computational results between the two classes, we shall often restrict ourselves to the second one, simply because it contains more problems. The initial computation of a feasible b-flow with the max-flow algorithm of Goldberg and Tarjan [8] takes in most cases less than 1% (and in our tests never more than 10%) of the total running time.

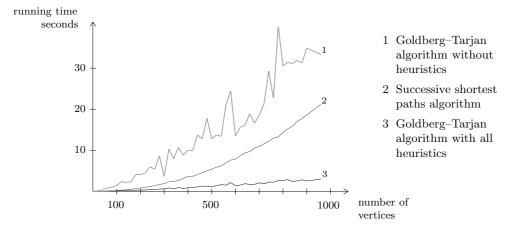


Figure 1: Running time of the Goldberg-Tarjan and the successive shortest paths algorithm (type 2 networks)

Figure 1 and 2 show the running time of the Goldberg–Tarjan and the successive shortest paths algorithm, another well–known minimum–cost flow algorithm due to Iri [11], Jewell [12], and Busacker and Gowen [3]; improved by Edmonds and Karp [4]. The underlying Dijkstra procedure is implemented in the fastest known way, with Fibonacci heaps. Two implementations of the Goldberg–Tarjan algorithm are compared – one without any heuristics, the other one with all heuristics described in Section 4: price refinement, push–look–ahead and set–relabel.

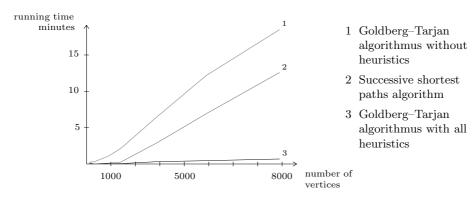


Figure 2: Running time of the Goldberg–Tarjan and the successive shortest shortest paths algorithm (type 1 networks)

The figures show that the implementation of the heuristics are essential to make the Goldberg–Tarjan algorithm competitive. With these heuristics it beats the successive shortest paths algorithm by a factor of seven on the second group and by a factor of ten on the first group of networks. However, we should mention that there are also instance types where the successive shortest paths algorithm is superior to the Goldberg–Tarjan algorithm.

Figure 3 shows the effect of each single heuristic on the running time and on the number of push and relabel operations (type 2 networks). Figure 3a) shows the basic algorithm, in b), c) and d) the effect of price refinement, set—relabel and push—look—ahead is shown. The running time of the basic algorithm is shown in all four parts for sake of comparison. Only local relabel operations are counted, so the set—relabel operations are not taken into account in (c).

The figure shows that each heuristic gives a substantial improvement of the number of push and relabel operations. The running time obviously depends mainly on the number of these operations. Moreover, the figure shows a tight correlation of the number of push and relabel operations. At first sight, this seems to be surprising, especially with the set–relabel heuristic, since about 70% of the running time is spent within this heuristic itself. However, the number of set–relabel operations depends on the number of ordinary relabel operations, since in our implementation the set–relabel is performed after every n relabel operations.

While price refinement and push-look-ahead yield a more or less uniform decrease of the number of operations and the running time, the set-relabel has the additional effect that the running time is strongly dependent on the number of vertices.

The push-look-ahead heuristic has a remarkable effect on the number of push operations: While this number is almost twice the number of relabel operations in the basic algorithm, the frequency of both operations becomes about equal when using push-look-ahead.

If one takes the number of set—relabel operations per vertex into account, the ratio of push—to relabel—operations changes completely (Figure 4). The number of relabel operations is then greater than the number of push operations in the case when no heuristic at all is used.

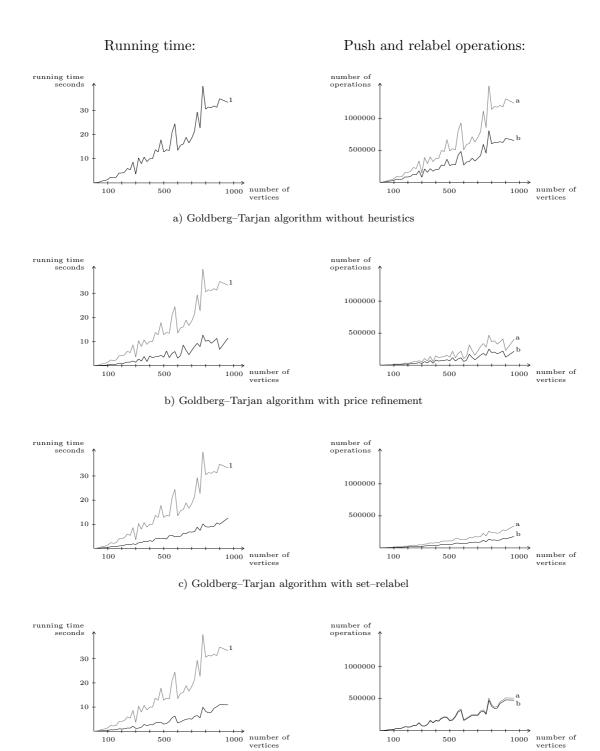


Figure 3: Effects of the heuristics on the running time of the Goldberg–Tarjan algorithm (type 2 networks)

d) Goldberg-Tarjan algorithm with push-look-ahead

a push operations

b relabel operations

1 Goldberg-Tarjan algorithm

without heuristics

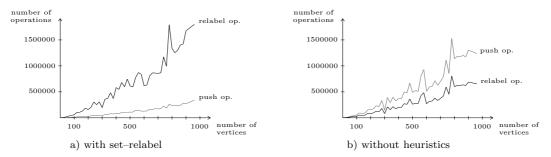


Figure 4: Push and relabel operations (including set—relabel operations) in the Goldberg— Tarjan algorithm (type 2 networks)

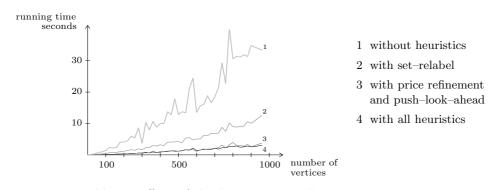


Figure 5: Non-additive effect of the heuristics on the running time

Figure 5 shows that the effects of the heuristics do not add up. With price refinement and push-look-ahead (3) one has about the same running time as with all heuristics. However, the additional use of the set-relabel heuristic can decrease the running time for larger instances ($n \ge 1000$). For smaller problems it is often better to do without.

Figure 6 might indicate that the set—relabel heuristic is the least important one. However, there are also instances where the removal of any of the heuristics increases the running time significantly.

Figure 7 shows the effect of the choice of the parameter α on the running time. This effect is quite significant, although it seems to depend on the problem structure. We found that $\alpha = 10$ is an appropriate choice.

Figure 8 shows the running time of our implementation with all heuristics for large graphs. They have been generated by NETGEN with the following parameters:

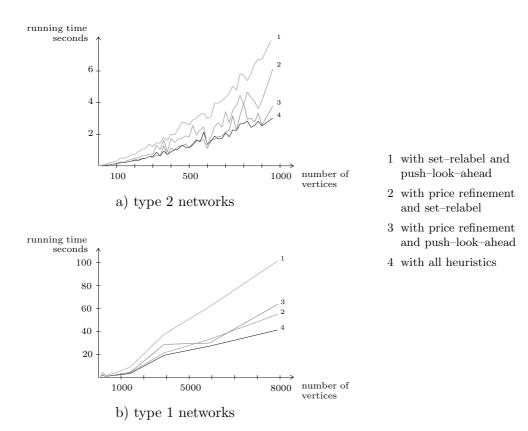
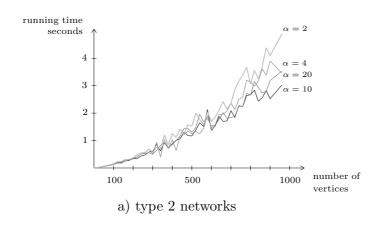


Figure 6: Effect of omitting one of the three heuristics on the running time

1.	initial value for the random number generator	large integer
2.	problem number for output documentation	
3.	number of vertices	n
4.	number of sources	$\frac{\frac{1}{3}n}{\frac{1}{3}n}$
5.	number of sinks	$\frac{1}{3}n$
6.	number of edges	10n
7.	minimal edge cost	-1000
8.	maximal edge cost	1000
9.	Total excess of the sources	$2000 \cdot \frac{1}{3}n$
10.	transshipment sources	0
11.	transshipment sinks	0
12.	percentage of the edges in the sparse network with maximal cost	100
13.	percentage of the edges in the sparse network with bounded capacity	100
14.	minimal edge capacity	1
15.	maximal edge capacity	1000

Parameters 12 and 13 concern the edges generated by NETGEN to ensure feasibility. Parameter 10 and 11 are irrelevant. The tests have been done on a IBM RS/6000 Mod. 590 which is faster than Mod. 550 by a factor of about 2–2.5. The numbers in Figure 8 show the average, minimum and maximum running time in seconds for five runs (with



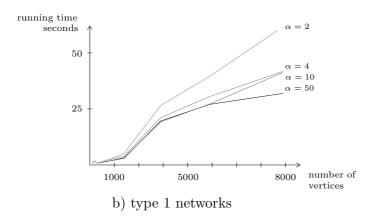


Figure 7: Effect of the parameter α on the running time

number of vertices	average	minimum	maximum
10000	59.94	59.30	61.59
30000	266.38	252.69	272.83
100000	1045.29	887.73	1111.31
300000	3945.05	3858.91	3996.14

Figure 8: Running time for large graphs

different initial values for the random number generator).

Since our implementation is part of a network optimization library used by several programs, robustness was even more important than efficiency. Nevertheless the figure shows that the running time grows only slightly faster than linear in the number of vertices. This is remarkable, especially when comparing the theoretical worst case running time. Moreover, the deviations from the average running time are small.

We do not claim that our implementation is the fastest existing one. Other minimum—cost flow algorithms, e.g. network—simplex methods, also perform very well. First experiments indicated that they can be faster on small instances, but the Goldberg—Tarjan algorithm

with our implementation seems to be superior on large instances. The (empirically) only slightly superlinearly growing running time (Figure 8) makes it especially attractive. A thorough comparison of all existing algorithms and implementations, as recommended by one of the referees, would be very interesting. However, this is beyond the scope of this paper whose aim is to demonstrate the value of several skillful heuristics for efficiently implementing the Goldberg–Tarjan algorithm. One of them, price–refinement based on minimum mean cycles, was, to our knowledge, never implemented before; its value is, however, significant.

References

- [1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin (1993) Network Flows. Prentice Hall, Engelwood Cliffs
- [2] U. Bünnagel (1998) Effiziente Implementierung von Netzwerkalgorithmen. Diploma thesis, University of Bonn
- [3] R.G. Busacker, and P.J. Gowen (1961) A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns. O.R.O. Technical Paper 15
- [4] J. Edmonds, and R.M. Karp (1972) Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. Journal of the ACM 19, 248-264
- [5] M.L. Fredman, and R.E. Tarjan (1987) Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM 34, 596-615
- [6] A.V. Goldberg (1992) An Efficient Implementation of a scaling Minimum-Cost Flow Algorithm. Departement of Computer Science, Stanford University, August 1992; NEC Research Institute, Princeton, October 1992; Journal of Algorithms 22 (1997), 1-29
- [7] A.V. Goldberg, and M. Kharitonov (1993) On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, 157-198
- [8] A.V. Goldberg, and R.E. Tarjan (1988) A New Approach to the Maximum Flow Problem. Journal of the ACM 35, 921-940
- [9] A.V. Goldberg, and R.E. Tarjan (1990) Finding Minimum–Cost Circulations by Successive Approximation. Mathematics of Operations Research 15, 430-466
- [10] A.V. Goldberg, É. Tardos, and R.E. Tarjan (1990) Network Flow Algorithms. In: B. Korte, L. Lovász, H. J. Prömel, A. Schrijver (eds.): Flows, Paths and VLSI Layout, Springer, Berlin, 101-164
- [11] M. Iri (1960) A New Method for Solving Transportation-Network Problems. Journal of the Operations Research Society of Japan 3, 27-87
- [12] W.S. Jewell (1958) Optimal Flow through Networks. Interim Technical Report 8, MIT

- [13] D.S. Johnson, and C.C. McGeoch (1993) Network Flows and Matching: First DI-MACS Implementation Challenge. AMS, Providence
- [14] R.M. Karp, and J.B. Orlin (1981): Parametric shortest paths with an application to cyclic staffing. Discrete Applied Mathematics 3, 37-45
- [15] D. Klingman, A. Napier, and J. Stutz (1974) A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. Managment Science 20, 814-821
- [16] H. Schneider, and M.H. Schneider (1987): Max-balancing weighted directed graphs. Madison, Baltimore
- [17] J. Vygen (1998) Algorithms for Detailed Placement of Standard Cells. Proceedings of the Conference Design, Automation and Test in Europe, IEEE, 321-324
- [18] N.E. Young, R.E. Tarjan, and J.B. Orlin (1991) Faster Parametric Shortest Path and Minimum Balance Algorithms. Networks 21, 205-221