Experimental evaluation of algorithmic solutions for the maximum generalised network flow problem*

Tomasz Radzik† King's College London

Shengxiang Yang‡ University of Leicester

December 2001

Abstract

The maximum generalised network flow problem is to maximise the net flow into a specified node in a network with capacities and gain-loss factors associated with edges. In practice, input instances of this problem are usually solved using general-purpose linear programming codes, but this may change because a number of specialised combinatorial generalised-flow algorithms have been recently proposed. To complement the known theoretical analyses of these algorithms, we develop their implementations and investigate their actual performance. We focus in this study on Goldfarb, Jin and Orlin's excess-scaling algorithm and Tardos and Wayne's push-relabel algorithm. We develop variants of these algorithms to improve their practical efficiency. We compare the performance of our implementations with implementations of simple, but non-polynomial, combinatorial algorithms proposed by Onaga and Truemper, and with performance of CPLEX, a commercial general-purpose linear programming package.

Keywords: Network optimisation, Network flow algorithms, Generalised flow, Experimental evaluation.

1 Introduction

In this paper we present our work on developing efficient implementations of recently proposed polynomial-time combinatorial algorithms for the *maximum generalised flow problem*. This problem generalises the maximum network flow problem in the following way. Each edge e in the underlying network has a gain factor $\gamma(e)$ associated with it, and if x units of flow enter edge e, then $x \cdot \gamma(e)$ units arrive at the other end. Each node has a specified amount of one common commodity, called the *(initial) excess* at this node. The objective is to design a flow which carries these node excesses through the network to one distinguished node, the *sink*. The designed flow should maximise the amount of the commodity arriving at the sink and should not violate the capacities of edges. The maximum generalised flow problem models some optimisation problems arising in manufacturing, transportation and financial analysis. The gain factors may represent the changes of the amount of the commodity caused by physical transformations (for example, evaporation or deterioration during transportations, or dissipation of energy during transmission) or administrative transformations (for example, currency

*This work was supported by the EPSRC grant GR/L81468. It was first presented at the *17th International Symposium on Mathematical Programming*, Atlanta, GA, USA, August 2000. It is also available as Technical Report TR-01-09, Department of Computer Science, King's College London, U.K.

†Department of Computer Science, King's College London, Strand, London WC2R 2LS, U.K.
E-mail: radzik@dcs.kcl.ac.uk.

‡Most of this work was done when this author was at the Department of Computer Science, King's College London. Current address: Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester LE1 7RH, U.K. E-mail: s.yang@mcs.le.ac.uk.

exchanges). A comprehensive discussion of various applications of the generalised flow problem can be found in [1], [3], and [12].

The maximum generalised flow problem is a special case of *linear programming*, so it can be solved by any of the known general-purpose linear programming methods. The best asymptotic worst-case time bound on computing maximum generalised flows using this approach is the $O(m^{1.5}n^2 \log B)$ bound of Kapoor and Vaidya's algorithm [9, 19] which is based on Karmarkar's interior-point method. Here n is the number of nodes, m is the number of edges, and B is the largest integer in the representations of the capacities and gain factors of edges and the excesses at nodes, assuming that these numbers are given as ratios of two integers. From the practical point of view, one can solve instances of the maximum generalised flow problem as linear programs using, for example, the commercial general-purpose linear-programming package CPLEX [10].

A number of specialised "combinatorial" maximum generalised flow algorithms have been proposed during the last decade. A combinatorial algorithm for the maximum generalised flow problem exploits the combinatorial structures of the underlying network and the flows in this network, and often uses as subroutines combinatorial algorithms for simpler network problems, such as the shortest paths problem, the maximum (non-generalised) flow problem, and the minimum-cost (non-generalised) flow problem. Two simple combinatorial maximum generalised flow algorithms are due to Onaga [13] and Truemper [18]. Onaga's algorithm uses shortest-path computations while Truemper's algorithm uses maximum flow computations, but both algorithms may need in the worst-case exponentially many iterations.

Goldberg, Plotkin, and Tardos [4] designed the first two polynomial-time combinatorial algorithms for the maximum generalised flow problem, which use shortest-path computations and minimum-cost (non-generalised) flow computations. The running times of the theoretically faster of those two algorithms is $O(n^2m(m + n \log n) \log n \log B)$. Improved versions of Goldberg, Plotkin, and Tardos' algorithms were proposed by Goldfarb and Jin [7] and Radzik [16]. Goldfarb, Jin and Orlin [8] presented two simple excess-scaling algorithms which run in $O(m^2(m + n \log n) \log B)$ time. Their simplicity made them probably the first practical polynomial-time combinatorial algorithms for the maximum generalised flow problem. Tardos and Wayne [17] (see also [20]) proposed a polynomial-time variant of Truemper's algorithm and an adaptation of Goldberg and Tarjan's push-relabel method for the minimum-cost (non-generalised) flow problem [6].

The aim of the work summarised in this paper is to investigate whether the recently proposed polynomial-time combinatorial algorithms for the maximum generalised network flow problem can lead to practically efficient implementations. We decided to focus on Goldfarb, Jin and Orlin's [8] excess-scaling algorithm and Tardos and Wayne's [17] push-relabel algorithm, because of their relative simplicity and because the minimum-cost flow algorithms which these algorithms generalise (Orlin's capacity-scaling algorithm [15] and Goldberg and Tarjan's push-relabel algorithm [6], respectively) were demonstrated to be practically efficient. Our first straightforward implementations were disappointingly slow, so in order to obtain competitive implementations, we had to develop modifications and new variants of those algorithms.

We compare the performance of our implementations with implementations of Onaga's and Truemper's algorithms and with performance of the linear-programming package CPLEX. The randomly generated input networks which we use in our experiments are loosely based on possible applications of the maximum generalised flow problem in the quantitative financial analysis. In our experiments our fastest variant of Goldfarb, Jin and Orlin's algorithm is usually considerably faster than the implementations of Onaga's and Truemper's algorithms. CPLEX performs better than our implementations, but on large networks its dominance is small enough to believe that combinatorial generalised network flow algorithms, most likely in combination with the ideas underlying general-purpose linear programming methods, will soon lead to the fastest implementations for the maximum generalised network flow problem.

In the next section we introduce the terminology of the generalised flows and show the linear-programming and the combinatorial optimality conditions for maximum generalised flows. In Section 3 we describe Onaga's and Truemper's algorithms. In Section 4 we describe the framework of Goldfarb, Jin and Orlin's excess-scaling algorithm and presents within this framework their original algorithm and its three variants proposed in this paper. In Section 5 we describe Tardos and Wayne [17] push-relabel method and our implementation of this method. In Sections 6 we briefly describe the generators of networks which we use in our experiments, and in Section 7 we present a few results from our experiments. In the final Section 8 we mention some promising directions in further improving implementations of combinatorial maximum generalised flow algorithms.

2 Definitions and Preliminaries

The input

An input instance of the *maximum generalised flow problem* is an *asymmetric generalised flow network* $\vec{G} = (V, \vec{E}, t, e, u, \gamma)$ with the following properties.

- $(V, \vec{E})$ is a directed graph with a set of nodes V and an asymmetric set of edges $\vec{E}$; that is, if $(v, w) \in \vec{E}$, then $(w, v) \notin \vec{E}$.
- $t \in V$ is the *sink*, the destination of flow.
- $e : V \longrightarrow R_{\geq}$ is an (*initial*) *excess* (or *supply*) *function.* We assume that $e(t) = 0$.
- $u : \vec{E} \longrightarrow R_{\geq}$ is an *edge capacity function.*
- $\gamma : \vec{E} \longrightarrow R_{>}$ is an *edge gain* (or *gain/loss*) *function.* For an edge $e \in \vec{E}$, $\gamma(e)$ is called the *gain* (or *gain/loss*) factor of e.

Symbols $R_{\geq}$ and $R_{>}$ stand for the nonnegative and the positive real numbers, respectively. The meaning of the gain function γ is that if x units of flow enter an edge (v, w) at node v, then $x \cdot \gamma(v, w)$ units arrive at node w. The assumption that the set of edges is asymmetric is made for notational convenience and without loss of generality. We also assume that for each node in $v \in V$, there is a path from v to the sink t consisting of positive-capacity edges. We denote the number of nodes by n and the number of edges by m. If we state running time bounds using also a parameter B, then we assume that all input numbers (that is, the capacities and gain factors of edges and the excesses of nodes) are given as fractional numbers with denominators and enumerators not greater than B.

A flow and the optimisation objective

A *flow* in network $\vec{G}$ is a nonnegative function $f : \vec{E} \longrightarrow R_{\geq}$ which satisfies the *capacity constraints*: for each edge $(v, w) \in \vec{E}$, $f(v, w)$, the amount of flow outgoing from node v along this edge, cannot be greater than the capacity of this edge; and the *flow conservation constraints*: the net flow outgoing from each node v cannot be greater than the initial excess at this node. The objective of the maximum generalised flow problem is to find a flow which maximises the net flow incoming into the sink t. Such a flow is called a *maximum* (*generalised*) *flow*. Formally, the maximum generalised flow problem can be expressed as the linear program (P) with decision variables $f(v, w)$, for each $(v, w) \in \vec{E}$. Constraints (2) and (3) are the flow conservation constraints and the capacity constraints, respectively. In (2), the first sum is the flow outgoing from node v, the second one is the flow incoming into node v, and their difference is the *net flow outgoing from node v*. The *net flow incoming to a node* is equal to the inverse of the net flow outgoing from this node.

(P)

$$\text{maximise:} \quad \sum_{(z,t)\in\vec{E}} \gamma(z,t) f(z,t) - \sum_{(t,x)\in\vec{E}} f(t,x); \tag{1}$$

$$\text{subject to:} \quad \sum_{(v,x)\in\vec{E}} f(v,x) - \sum_{(z,v)\in\vec{E}} \gamma(z,v) f(z,v) \;\leq\; e(v), \quad \text{for each } v \in V - \{t\}; \tag{2}$$

$$f(v,w) \;\leq\; u(v,w), \quad \text{for each } (v,w) \in \vec{E}; \tag{3}$$

$$f(v,w) \;\geq\; 0, \quad \text{for each } (v,w) \in \vec{E}.$$

The dual problem

Let $\mu(v)$, for $v \in V - \{t\}$, and $\lambda(v,w)$, for $(v,w) \in \vec{E}$, be the dual variables associated with constraints (2) and (3) of linear program (P). The dual problem of problem (P) has the following formulation, simplified by introduction of a constant $\mu(t) = 1$.

$$
\text{(D)} \qquad \begin{array}{rll}
\text{minimise:} & \displaystyle\sum_{v \in V} e(v)\mu(v) \;+ \sum_{(v,w)\in\vec{E}} u(v,w)\lambda(v,w); & \\
\text{subject to:} & \mu(v) - \gamma(v,w)\mu(w) + \lambda(v,w) \ge 0, & \text{for each } (v,w) \in \vec{E}; \\
& \mu(v) \ge 0, & \text{for each } v \in V; \\
& \lambda(v,w) \ge 0, & \text{each } (v,w) \in \vec{E}; \\
& \mu(t) = 1. &
\end{array}
$$

A non-negative vector $\mu : V \longrightarrow R_{\ge}$, with $\mu(t) = 1$, is called a *(node) labeling* of network $\vec{G}$. It is clear from the formulation of problem (D) that for any node labeling of $\vec{G}$, there exists a vector λ such that the pair (μ, λ) is a feasible solution of problem (D). If we fix node labeling μ, then the objective function of the dual problem (D) is minimised by setting

$$
\lambda_\mu(v,w) \;=\; \max\{0,\, \gamma(v,w)\mu(w) - \mu(v)\}, \quad \text{for each } (v,w) \in \vec{E}. \tag{4}
$$

Therefore we will refer only to μ as the dual vector, assuming that vector λ is as defined in (4).

A technical issue

For notational convenience, flow networks are often used in a special form with symmetric sets of edges. In the context of the maximum generalised flow problem, a *symmetric flow network* is a network $H = (V, E, t, e, u, \gamma)$ where the attributes have the same meaning as for the asymmetric network $\vec{G}$ with the difference that the set of edges E is symmetric, that is, if $(v,w) \in E$, then also $(w,v) \in E$, and for each $(v,w) \in E$, $\gamma(w,v) = 1/\gamma(v,w)$. For such a network, the maximum generalised flow problem is formulated in the following way.

$$
(\mathrm{P}_s) \qquad \begin{array}{rll}
\text{maximise:} & \displaystyle\sum_{(z,t)\in E} \gamma(z,t) f(z,t); & \\
\text{subject to:} & f(w,v) \;=\; -\gamma(v,w) f(v,w), & \text{for each } (w,v) \in E; \qquad (5) \\
& \displaystyle\sum_{(v,x)\in E} f(v,x) \;\le\; e(v), & \text{for each } v \in V - \{t\}; \qquad (6) \\
& f(v,w) \;\le\; u(v,w), & \text{for each } (v,w) \in E.
\end{array}
$$

Only the positive edge flows are actual flows. Symmetric edges and Condition (5), which is sometimes referred to as the *skew symmetry of a flow function*, are used to simplify the notions of residual paths and residual networks defined below. In the case of an asymmetric network, we have to refer to "sending flow along an edge (v,w)," as well as to "reversing some of the flow sent before along (v,w)." In the case of a symmetric network, we can refer only to "sending flow along an edge," either (v,w) or its symmetric (w,v).

For an asymmetric input network $\vec{G} = (V, \vec{E}, t, e, u, \gamma)$, let G denote the symmetric network (V, E, t, e, u, γ), where E is obtained from $\vec{E}$ by adding to $\vec{E}$ the symmetric edge (w,v) for each edge $(v,w) \in \vec{E}$, and setting $u(w,v) = 0$ and $\gamma(w,v) = 1/\gamma(v,w)$. For a flow f in network G, that is, for a feasible vector $f : E \longrightarrow R$ of problem (P_s), the corresponding flow in network $\vec{G}$ is obtained simply by restricting f from set E to set $\vec{E}$. Conversely, for a flow f in network $\vec{G}$, the corresponding flow in network G is obtained by extending f from $\vec{E}$ to E according to (5). It can be easily checked that this natural correspondence shows that problems (P) and (P$_s$) are equivalent. For example, the right-hand sides of (2) and (6) are different formulas for the same quantity, the net flow from a node v:

$$\begin{aligned}\sum_{(v,x)\in\vec{E}} f(v,x) - \sum_{(z,v)\in\vec{E}} \gamma(z,v)f(z,v) &= \sum_{(v,x)\in E, f(v,x)>0} f(v,x) + \sum_{(z,v)\in E, f(z,v)>0} -\gamma(z,v)f(z,v) \\ &= \sum_{(v,x)\in E, f(v,x)>0} f(v,x) + \sum_{(v,z)\in E, f(v,z)<0} f(v,z) = \sum_{(v,x)\in E} f(v,x).\end{aligned}$$

We will use either network $\vec{G}$ and problem (P) or network G and problem (P$_s$), whichever is more convenient for a particular purpose. The reason for introducing first asymmetric networks is to have a more natural notion of flow and a straightforward derivation of the dual problem.

Residual network

Let f be a flow in network G. The *residual capacities* u_f of edges and the *residual excesses* e_f of nodes are defined in the following way.

$$\begin{aligned} u_f(v,w) &= u(v,w) - f(v,w), && \text{for each } (v,w)\in E; \\ e_f(v) &= e(v) - \sum_{(v,w)\in E} f(v,w), && \text{for each } v \in V. \end{aligned}$$

Note that $e_f(t)$ is the value of the objective function for flow f: the net flow incoming into the sink t (we assume that $e(t) = 0$). If $e_f(v) > 0$ for a node $v \in V_f$, then we say that v is a node with (residual) excess. A *residual edge* is an edge in E with positive residual capacity and a *residual path* (residual cycle) is a path (cycle) which consists only of residual edges. Let V_f denote the set of nodes which can reach the sink t along residual paths, and let $E_f = E \cap (V_f \times V_f)$ be the set of the edges between the nodes in V_f. The *residual network* is the (symmetric) generalised flow network $G_f = (V_f, E_f, t, e_f, u_f, \gamma)$, with functions e_f, u_f, and γ restricted to sets V_f and E_f.

If h is a flow in the residual network G_f, then $f + h$ is a flow in network G, which creates the excess $e_f(t) + e_h(t)$ at the sink t. (Assume that $h(v,w) = 0$ for each $(v,w) \in E - E_f$.) A process of computing a flow h in the residual network G_f and adding it to the current flow f is often called an *augmentation of the current flow*. If h is a maximum flow in the residual network G_f, then $f + h$ is a maximum flow in network G.

Let $e_{\text{opt}}(t)$ denote the maximum possible net flow into the sink $e_f(t)$ over all flows f in network G. We say that a flow f in network G is ξ-*optimal*, if $e_f(t)(1+\xi) \geq e_{\text{opt}}(t)$.

Flow generating cycles

The gain $\gamma(P)$ of a path or cycle P is equal to the product of the gains of the edges on P. If we send x units of flow from a node v along a residual path P to the sink, then the residual excess at the sink increases by $x \cdot \gamma(P)$ units. A *flow generating cycle* is a residual cycle with gain greater than 1. If we send x units of flow around a cycle Γ from a node v, then $x \cdot \gamma(P)$ units come back to v. Thus if Γ is a flow generating cycle, then by sending flow around Γ we can create (or increase) the excess at at least one node on this cycle. This additional excess may be sent to the sink, so it may contribute to the optimal net flow into the sink.

Let f be a flow in network G and let h be a flow in the residual network G_f such that the residual network G_{f+h} of the combined flow $f' = f + h$ does not have flow generating cycles. We say that such a flow h *cancels*, or *saturates, (all) flow generating cycles* in the residual network G_f, and the computation of such a flow h is called *canceling*, or *saturating (all) flow generating cycles*.

Relabeled network

Let f and μ be a flow and a positive node labeling in network G, respectively. The *relabeled residual capacities*, the *relabeled gain factors*, and the *relabeled residual excesses* are defined as:

$$\begin{aligned} u_{f,\mu}(v,w) &= u_f(v,w)\mu(v), \\ \gamma_\mu(v,w) &= \gamma(v,w)\mu(w)/\mu(v), \\ e_{f,\mu}(v) &= e_f(v)\mu(v). \end{aligned}$$

The residual network G_f and the *relabeled residual network* $G_{f,\mu} = (V_f, E_f, t, e_{f,\mu}, u_{f,\mu}, \gamma_\mu)$ are equivalent instances of the maximum generalised flow problem. For a flow h in network G_f, the corresponding flow h_μ in network $G_{f,\mu}$ is such that

$$h_\mu(v,w) = h(v,w)\mu(v), \quad \text{for each } (v,w) \in E_f.$$

We actually consider functions h and h_μ as different ways of expressing the same flow; h expresses that flow in terms of network G_f, while h_μ expresses it in terms of the equivalent network $G_{f,\mu}$.

Observe that for a path P from a node v to a node w, $\gamma_\mu(P) = \gamma(P)\mu(w)/\mu(v)$, and for a cycle Γ, $\gamma_\mu(\Gamma) = \gamma(\Gamma)$. If there exists a positive node labeling of a residual network G_f such that the relabeled gain of each residual edge in G_f is at most 1, then clearly network G_f does not have any flow generating cycles. Such a node labeling is called a *proper (node) labeling*. Conversely, if a residual network G_f does not have any flow generating cycles, then there exists a proper labeling of G_f. For example, if network G_f does not have any flow generating cycle, then the node labeling μ of G_f such that $\mu(v)$ is equal to the highest gain of a residual path from $v \in V_f$ to the sink t in network G_f is well defined and is proper. This labeling is called the *canonical (node) labeling*. If a network does not have any flow generating cycle, then we call it a *non-gain network*.

A *highest-gain tree* in a non-gain network G_f is a subset of residual edges of G_f which form a tree rooted at the sink t with edges directed towards the root, such that the path in this tree from a node $v \in V_f$ to the root t is a highest-gain residual path from v to t in G_f. The canonical labeling of a non-gain network G_f and a highest-gain tree can be computed by a shortest-path algorithm by setting the weight of an edge e to $-\log(\gamma(e))$. Using the Bellman-Ford-Moore single-source shortest-paths algorithm, the computation of the canonical labeling and a highest-gain tree takes $O(mn)$ time. If we have a proper labeling (as often happens in maximum generalised flow algorithms), then this computation can be done in $O(m+n\log n)$ time using Dijkstra's single-source shortest paths algorithm with Fibonacci heaps [2].

Complementary slackness conditions and other optimality conditions

Let f and μ be feasible solutions of the primal problem (P) and the dual problem (D), respectively. The following conditions are the linear-programming *complementary slackness conditions* for f and μ.

1. For each $(v,w) \in \vec{E}$, if $f(v,w) > 0$, then $\gamma(v,w)\mu(w) \geq \mu(v)$.
2. For each $(v,w) \in \vec{E}$, if $f(v,w) < u(v,w)$, then $\gamma(v,w)\mu(w) \leq \mu(v)$.
3. For each $v \in V \setminus \{t\}$, if $\mu(v) > 0$, then $e_f(v) = 0$.

These three conditions correspond to the three cases of a positive variable in one problem implying that the corresponding bound in the other problem must be tight. Condition (2) is an equivalent formulation of the condition that if $\lambda(v,w) > 0$, then $f(v,w) = u(v,w)$.

In the theorem below we state together "linear-programming" and "combinatorial" optimality conditions for maximum generalised flows. The "combinatorial" Condition (2) is due to Onaga [14].

Theorem 1. *If f is a flow in network G, then the following four conditions are equivalent.*

0. Flow f is a maximum flow.

1. *There exist a node labeling μ of G such that the feasible solutions f and μ of the primal and the dual problems (P) and (D), satisfy the complementary slackness conditions 1–3.*

2. *(a) There is no node with positive residual excess in V_f, and*

   (b) there is no flow generating cycle in the residual network G_f.

3. *(a) There is no node with positive residual excess in V_f, and*

   (b) there exists a proper node labeling of the residual network G_f.

Proof.
(0) $\Leftrightarrow$ (1). This is the linear-programming complementary slackness optimality condition.

(2) $\Leftrightarrow$ (3). This equivalence is discussed above in section "Relabeled network."

(0) $\Rightarrow$ (2). If not (2a) or not (2b), then we would be able to send more flow into the sink.

(2) $\Rightarrow$ (1). Compute the canonical labeling μ of the residual network G_f, extend it to a labeling of network G by setting $\mu(v) = 0$ for each $v \in V - V_f$, and check that vectors f and μ satisfy the complementary slackness conditions 1–3. ■

The algorithms considered in this paper converge to optimal solutions by progressively coming ever closer to satisfying the optimality condition 3. Onaga's and Truemper's algorithms described in Section 3 and Goldfarb, Jin and Orlin's excess-scaling algorithm and its variants described in Section 4 maintain a proper node labeling of the current residual network G_f, and keep reducing the residual excesses at the nodes in G_f. Tardos and Wayne's push-relabel algorithm discussed in Section 5 also keeps reducing the residual excesses at the nodes in network G_f, but maintains only an "approximately proper" labeling: for some small $\beta > 0$, $\gamma_\mu(e) \le 1 + \beta$ for each residual edge e in network G_f. If a maximum generalised flow algorithm may be viewed as being based on the optimality condition 3, then the relabeled residual excess in the residual network of the current flow may indicate the closeness of this flow to an optimal one, as described in next paragraph.

Let f be a flow in network G such that the residual network G_f is a non-gain network and let μ be a proper labeling of G_f. Define the *total relabeled residual excess* as

$$\textsc{TotResEx}_{f,\mu} = \sum_{v \in V_f \setminus \{t\}} e_{f,\mu}(v).$$

The flow decomposition theorem for generalised flows (see, for example, [4] or [16] for details) implies that

$$\textsc{TotResEx}_{f,\mu} \ge e_{\mathrm{opt}}(t) - e_f(t). \tag{7}$$

This inequality implies that if

$$\textsc{TotResEx}_{f,\mu} \le \xi \cdot e_f(t), \tag{8}$$

then $e_f(t)(1 + \xi) \ge e_{\mathrm{opt}}(t)$, so flow f is ξ-optimal.

3 Onaga's and Truemper's algorithms

The first and the simplest combinatorial algorithm for the maximum generalised flow problem was proposed by Onaga [13] and can be described in the following way. The algorithm starts with a non-gain residual network G_f and iteratively augments the current flow by sending flow from a node $v \in V_f \setminus \{t\}$ with positive residual excess to the sink t along a highest gain path P (see Figure 1). The amount of flow sent in the current iteration from node v is such that either the residual excess at node v becomes zero or at least one edge on path P becomes saturated. By using always only the highest-gain residual paths, the algorithm maintains the invariant that there are no flow generating cycles in the residual network. The computation terminates when $V_f \setminus \{t\}$ does not contain any node with positive residual excess, or, if we need only an approximate solution, when $\textsc{TotResEx}_{f,\mu}$

{ INPUT: a generalised flow network G }
$f \leftarrow$ a flow which cancels all flow generating cycles in G;
while there is a node $v \in V_f \setminus \{t\}$ with positive residual excess **do**
  1: compute a highest gain path P from v to t in G_f;
  2: update flow f by sending flow from v to t along P
    (transferring as much of the excess from v to t as the edge capacities on P allow);
  { invariant: there are no flow generating cycles in G_f }
end_while
{ OUTPUT: optimal flow f in G }

Figure 1: Onaga's algorithm (code HighestPath).

{ INPUT: a generalised flow network G }
$f \leftarrow$ a flow which cancels all flow generating cycles in G;
while there is a node $v \in V_f \setminus \{t\}$ with positive residual excess **do**
  1: $\mu \leftarrow$ the canonical labeling of G_f;
  2: $h \leftarrow$ a maximum flow h in $G_{f,\mu}$ from the nodes with excesses to t
    using only edges with (relabeled) gains equal to 1;
  3: $f \leftarrow f + h$;
  { invariant: there are no flow generating cycles in G_f }
end_while
{ OUTPUT: optimal flow f in G }

Figure 2: Truemper's algorithm (code MaxFlow).

decreases below the desired approximation level. In the former case, Condition 2 of Theorem 1 implies that the computed flow is optimal. In the latter case, if we use the termination condition (8), then the computed flow is ξ-optimal.

The running time of each iteration of Onaga's algorithm is dominated by the computation of a highest gain path from the selected node v to the sink. This computation is done by computing the canonical labeling μ and a highest-gain tree in the current residual (non-gain) network G_f. After updating the flow along a highest-gain path from v to t, the labeling μ remains a proper labeling, since the relabeled gain of each new residual edge is equal to 1. Therefore the canonical labeling and a highest-gain tree at the beginning of each iteration (except possibly the first iteration) can be done using Dijkstra's shortest-path computation.

Truemper [18] proposed an algorithm which also augments the current flow using only the highest-gain paths (so it maintains the invariant that there are no flow generating cycles in the residual network), but in each iteration all highest-gain paths to the sink are used. The pseudocode of Truemper's algorithm is shown in Figure 2. Let f be the current flow and let μ be the canonical labeling of the residual network G_f. The highest-gain paths to the sink are the paths to the sink which consist only of edges with relabeled gains γ_μ equal to 1. Truemper's algorithm computes in each iteration a maximum flow in network $G_{f,\mu}$ from the nodes with positive residual excesses to the sink using only the edges with relabeled gains γ_μ equal to 1. This is a standard (non-generalised) maximum flow computation. The remarks above regarding the termination conditions in Onaga's algorithm and the optimality of computed flows apply also to Truemper's algorithm.

In the worst case, both Onaga's and Truemper's algorithms may have to perform exponentially many iterations. Actually, in the model of computation which assumes that data can be not only fractional but arbitrary real numbers, Onaga's algorithm may not terminate in finite time.

```
{ INPUT: a generalised flow network G and a number ξ > 0 }
f ← a flow which cancels all flow generating cycles in G;
μ ← the canonical labeling in G_f;
while  TotResEx_{f,μ} > ξ · e_f(t)  do
   Δ ← TotResEx_{f,μ}/(2(m + n));
   Phase(Δ);
   { no flow generating cycles in G_f, μ is the canonical labeling of G_f,
     TotResEx_{f,μ} < (m + n)Δ }
end_while
{ OUTPUT: ξ-optimal flow f in G }.

Phase(Δ):
   { no flow generating cycles in G_f, μ – the canonical labeling of G_f }
   while  there exists v ∈ V_f \ {t} such that e_{f,μ}(v) ≥ Δ  do
       UpdateFlow;   { a sequence of EdgeFlow operations to decrease
                       ⌊TotResEx_{f,μ}/Δ⌋ by at least 1 }
       compute the canonical labeling μ and a highest-gain tree in G_f;
   end_while
   for  each (v, w) ∈ E  do  LinkFlow([v, w], v, e_{f,μ}(v, w)).
```

Figure 3: The general framework of Goldfarb, Jin and Orlin's excess scaling algorithm.

4 Excess-scaling algorithms

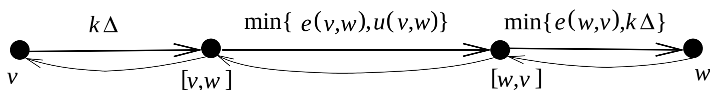
Goldfarb, Jin and Orlin [8] proposed two excess-scaling algorithms. In our paper we consider the first of those two algorithms and its variants, which have the following overall structure (see Figure 3). First all flow generating cycles are canceled, that is, a flow f in an input network G is computed such that the residual network G_f is a non-gain network. Throughout all subsequent computation, which consist of a sequence of *scaling phases* controlled by the *scaling parameter* Δ, the residual network always remains a non-gain network.

During one phase the value of the scaling parameter is fixed at

$$\Delta \;=\; \frac{\textsc{TotResEx}_{f',\mu'}}{2(n+m)}, \tag{9}$$

where f' is the flow in network G at the beginning of the phase and μ' is the canonical labeling of the residual network $G_{f'}$. The computation performed during one phase is a sequence of applications of operation UpdateFlow and re-calculations of the highest-gain tree and the canonical labeling of the residual network. Operation UpdateFlow sends residual node excesses towards the sink t along edges of the current highest-gain tree using operation EdgeFlow. Operation EdgeFlow(v, w, k) tries to send $k\Delta$ units of flow along a tree edge (v, w), where k is a positive integer. The current phase continues for as long as there is a node in the residual network with the residual excess at least Δ. When the phase ends, the value of the scaling parameter Δ is re-computed and the next phase begins. The original Goldfarb, Jin and Orlin's algorithm and its three variants which we propose in this paper differ in the ways operation UpdateFlow sends the residual node excesses in the current highest-gain tree. Before describing the details of the different variants of operation UpdateFlow, we first discuss the convergence of the overall method, the crucial underlying idea of "storing" small excesses at the edges, and the details of operation EdgeFlow.

Let f', f'' and μ', μ'' denote the flows and the canonical labelings at the beginning and at the end of one phase. The computation performed during this phase has the property that the total relabeled residual excess $\textsc{TotResEx}_{f'',\mu''}$ at the end of the phase is less than $(n+m)\Delta$, that is, less than half of the total relabeled residual excess $\textsc{TotResEx}_{f',\mu'}$ at the beginning of the phase; see (9). Thus the value of the scaling parameter Δ decreases geometrically from phase to phase. There



EdgeFlow(v, w, k):
  { send $k\Delta$ units of flow along edge (v, w) from v towards w;
    pre-conditions: $\gamma_\mu(v, w) = 1$, $e_{f,\mu}(v) \geq \Delta$, k is a positive integer, $k\Delta \leq e_{f,\mu}(v)$ }
  LinkFlow$(v, [v, w], k\Delta)$;
  LinkFlow$([v, w], [w, v], \min\{e_{f,\mu}(v, w), u_{f,\mu}(v, w)\})$;
  LinkFlow$(v, [v, w], \min\{e_{f,\mu}(w, v), k\Delta\})$.

LinkFlow(x, y, δ):
  { $\gamma_\mu(x, y) = 1$; $\delta \leq \min\{e_{f,\mu}(x), u_{f,\mu}(x, y)\}$ }
  $e_{f,\mu}(x) \leftarrow e_{f,\mu}(x) - \delta$; $e_{f,\mu}(y) \leftarrow e_{f,\mu}(y) + \delta$;
  $f_\mu(x, y) \leftarrow f_\mu(x, y) + \delta$; $f_\mu(y, x) \leftarrow f_\mu(y, x) - \delta$.

Figure 4: Sending flow along one edge.

are two main options for the termination condition, which correspond to two possible definitions of an approximate flow. The computation may terminate when the total relabeled residual excess TotResEx$_{f,\mu}$ decreases below ξ times its initial value (the value at the beginning of the first phase), or when it decreases below ξ times the current accumulated excess at the sink t, where $\xi > 0$ is the input parameter indicating the allowed approximation of the computed solution. If the former termination condition is used, the computation terminates in $\log(1/\xi)$ phases. In Figure 3 and in our implementations we use the latter stopping condition, which ensures that the final flow is ξ-optimal.

We discuss now how the residual node excesses are sent towards the sink t during the execution of operation UpdateFlow. Denote the number of full Δ-unit portions of residual excess at the nodes in $V_f \setminus \{t\}$ by

$$\text{TotResEx}_{f,\mu,\Delta} \quad = \sum_{v \in V_f \setminus \{t\}} \lfloor e_{f,\mu}(v)/\Delta \rfloor.$$

Operation UpdateFlow sends residual node excesses towards the sink t along the edges of the current highest-gain tree, having an underlying aim of decreasing TotResEx$_{f,\mu,\Delta}$ by at least 1 (to ensure progress of computation). This aim could be easily achieved if the residual capacities of all tree edges were at least Δ, because sending Δ units of flow from a node $v \in V \setminus \{t\}$ to the sink t decreases TotResEx$_{f,\mu,\Delta}$ by exactly 1. Thus the question is how to handle the edges which have small residual capacities. Goldfarb, Jin and Orlin [8] proposed a very elegant solution, which is based on storing some small excesses at the edges and can be described in the following way. Imagine that each edge (v, w) is split into three *links* by introducing two new nodes $[v, w]$ and $[v, w]$; see Figure 4. Links $(v, [v, w])$ and $([w, v], w)$ have infinite capacity in both directions and the gain factors equal to 1. The middle link $([v, w], [w, v])$ takes the capacity and the gain factor of the edge (v, w). The excesses at $[v, w]$ and $[v, w]$ are initially zero. It is easy to see that such transformation does not change in any essential way the task of sending excesses through the network to maximise the total amount reaching the sink. Nodes $[v, w]$ and $[v, w]$ are introduced to store small excesses.

The computation of operation UpdateFlow is a sequence of operations EdgeFlow(v, w, k) applied to some edges (v, w) of the current highest-gain tree. The formal pre-conditions of this operation are: $\gamma_\mu(v, w) = 1$, $e_{f,\mu}(v) \geq \Delta$, and k is a positive integer such that $k\Delta \leq e_{f,\mu}(v)$. This operation sends $k\Delta$ units of flow from a node v into an edge (v, w) in the following way. First $k\Delta$ units of flow are sent from v to $[v, w]$. Then flow of value equal to the minimum of $e_{f,\mu}(v, w)$, the current excess at $[v, w]$, and $u_{f,\mu}(v, w)$, the current residual capacity of edge (v, w) (and of link $([v, w], [w, v])$ as well) is sent from $[v, w]$ to $[w, v]$. Finally, flow of value equal to the minimum of $e_{f,\mu}(w, v)$, the current excess at $[w, v]$, and $k\Delta$ is sent from $[w, v]$ to w. The details of operation EdgeFlow(v, w, k) are shown in Figure 4. To clarify the meaning of the attribute "current" which we use here, we note that, for example, the current excess at $[v, w]$ after sending $k\Delta$ units of flow from v to $[v, w]$ is equal to $e_{f,\mu}(v, w) = e_{f',\mu}(v, w) + k\Delta$, where f' is the flow right before the application of operation EdgeFlow(v, w, k) and μ is the current node labeling, which does not change during the execution of operation UpdateFlow. Note also that we simplify notation by writing $e_{f,\mu}(w, v)$ instead of $e_{f,\mu}([w, v])$.

At the end of each phase the excesses accumulated at the added nodes are sent to the original nodes: the excess from $[v, w]$ is sent to node v and the excess from $[w, v]$ is sent to node w (see the last line in the pseudocode Phase(Δ) in Figure 3). To be able to claim that each phase decreases substantially the total residual excess, we have to ensure that only small excesses can accumulate at the added nodes. Goldfarb, Jin and Orlin [8] send always exactly Δ units of flow from a node v into an edge (v, w), that is, in our terminology, they use operation EdgeFlow$(v, w, 1)$, and they prove that such operation preserves the following condition.

$$e_{f,\mu}(v, w) + e_{f,\mu}(w, v) < \Delta. \tag{10}$$

Observe that Condition (10) is also preserved by the re-calculation of the canonical labeling, because the new canonical labels can be only the same or smaller than the previous ones, so the relabeled node excesses either remain the same or decrease. Since Condition (10) is true for every edge (v, w) at the beginning of the phase (the excesses at the added nodes are zero), this condition is an invariant of the computation performed during one phase, so it holds in particular also at the end of the phase. Thus at the end of the phase, the residual excess at each node $v \in V_f \setminus \{t\}$ is less than Δ and the residual excess stored on each edge is less than Δ, so the total residual excess is less than $(n + m)\Delta$, as claimed above.

We implemented Goldfarb, Jin and Orlin algorithm, but the initial experiments showed that using solely the operation of sending only Δ units of flow along one edge results in a slow and non-competitive code. We have obtained substantially faster implementations by sending excesses equal to multiplicities of Δ. The following lemma shows values of k which ensure that operation EdgeFlow(v, w, k) preserves Condition (10). This lemma and its proof generalise Lemma 4 and its proof presented in [8]. The lemma says that operation EdgeFlow(v, w, k) preserves also an additional condition (12). One can easily verify that Condition (12) holds for each edge at the beginning of a phase and is preserved by re-calculations of the canonical labeling. Thus both Conditions (10) and (12) are invariants throughout the whole computation of one phase, provided that for each application of operation EdgeFlow(v, w, k) the pre-conditions of this operation and the bound (11) on k are satisfied. The importance of Invariant (12) lies in ensuring that if $e_{f,\mu}(v) \geq \Delta$ and $u_{f,\mu}(v, w) > 0$, then the upper bound (11) on k is always at least 1.

Lemma 2. *Let f and μ denote the current flow and labeling. If nodes $v, w \in V_f$ and an integer k are such that $\gamma_\mu(v, w) = 1$, $e_{f,\mu}(v) \geq \Delta$ and $1 \leq k \leq \lfloor e_{f,\mu}(v)/\Delta \rfloor$ (that is, the pre-conditions of operation* EdgeFlow(v, w, k) *are satisfied), and additionally*

$$k \;\leq\; \lceil (u_{f,\mu(v,w)} - e_{f,\mu}(v, w))/\Delta \rceil, \tag{11}$$

then operation EdgeFlow(v, w, k) *preserves Condition (10) and the following Condition (12):*

$$\begin{array}{lll} u_{f,\mu}(v, w) > 0 & \Rightarrow & e_{f,\mu}(v, w) < u_{f,\mu}(v, w), \quad \textit{and} \\ u_{f,\mu}(w, v) > 0 & \Rightarrow & e_{f,\mu}(w, v) < u_{f,\mu}(w, v). \end{array} \tag{12}$$

Proof. Let v, w, and k be as required by the lemma, and assume that Conditions (10) and (12) hold before the execution of operation EdgeFlow(v, w, k). Let f' and f'' be the flows before and after the execution of this operation, respectively. The computation begins with sending $k\Delta$ units of flow from v to $[v, w]$. The amount of flow sent subsequently from $[v, w]$ to $[w, v]$, and further on to w, depends on the relative value of the residual capacity $u_{f',\mu}(v, w)$. Consider two cases.

If $k\Delta + e_{f',\mu}(v, w) \leq u_{f',\mu}(v, w)$, then $k\Delta + e_{f',\mu}(v, w)$ units of flow are sent from $[v, w]$ to $[w, v]$, and $k\Delta$ units of flow are sent from $[w, v]$ to w. Thus $e_{f'',\mu}(v, w) = 0$ and $e_{f'',\mu}(w, v) = e_{f',\mu}(w, v) +$

$e_{f',\mu}(v,w) < \Delta$, so Condition (10) holds after the operation. Since $u_{f'',\mu}(w,v) \geq k\Delta + e_{f',\mu}(v,w) \geq \Delta > e_{f'',\mu}(w,v)$, Condition (12) holds as well.

If $k\Delta + e_{f',\mu}(v,w) > u_{f',\mu}(v,w)$, then $u_{f',\mu}(v,w)$ units of flow are sent from $[v,w]$ to $[w,v]$, saturating edge (v,w). Thus $e_{f'',\mu}(v,w) = e_{f',\mu}(v,w) + k\Delta - u_{f',\mu}(v,w)$. Consider two sub-cases. If $k\Delta \leq e_{f',\mu}(w,v) + u_{f',\mu}(v,w)$, then $k\Delta$ units are sent from $[w,v]$ to w. In this sub-case, $e_{f'',\mu}(w,v) = e_{f',\mu}(w,v) + u_{f',\mu}(v,w) - k\Delta$, so $e_{f'',\mu}(v,w) + e_{f'',\mu}(w,v) = e_{f',\mu}(v,w) + e_{f',\mu}(w,v) < \Delta$, which means that Condition (10) holds after the operation. We also have $u_{f'',\mu}(v,w) = 0$ and $e_{f'',\mu}(w,v) = e_{f',\mu}(w,v) + u_{f',\mu}(v,w) - k\Delta < u_{f',\mu}(v,w) \leq u_{f'',\mu}(w,v)$, so Condition (12) holds as well.

If $k\Delta > e_{f',\mu}(w,v) + u_{f',\mu}(v,w)$, then all $e_{f',\mu}(w,v) + u_{f',\mu}(v,w)$ units of flow are sent from $[w,v]$ to w. Thus $e_{f'',\mu}(w,v) = 0$ and $e_{f'',\mu}(v,w) + e_{f'',\mu}(w,v) = e_{f'',\mu}(v,w) = e_{f',\mu}(v,w) + k\Delta - u_{f',\mu}(v,w) < \Delta$, where the last inequality follows from (11). This means that Condition (10) holds after the operation, and since $u_{f'',\mu}(v,w) = 0$ and $e_{f'',\mu}(w,v) = 0$, then Condition (12) holds as well. ■

We describe below four variants of operation UPDATEFLOW, which define the algorithm proposed by Goldfarb, Jin and Orlin [8] and its three variants. Each variant of operation UPDATEFLOW is dominated by $O(n)$ applications of operation EDGEFLOW, and decreases $\text{TOTRESEX}_{f,\mu,\Delta}$ at least by 1. The re-calculation of the canonical labeling and the highest-gain tree performed after each application of operation UPDATEFLOW may decrease further $\text{TOTRESEX}_{f,\mu,\Delta}$, but it cannot increase this quantity. At the beginning of a phase,

$$\text{TOTRESEX}_{f,\mu,\Delta} \;\leq\; \text{TOTRESEX}_{f,\mu}/\Delta \;=\; 2(n+m).$$

This means that the number of applications of operation UPDATEFLOW in one phase is at most $2(n+m)$, so the running time of one phase is at most $2(n+m)(O(n) + O(m + n\log n)) = O(m(m + n\log n))$. (Operation UPDATEFLOW changes the flow only on edges which have the relabeled gain equal to 1, so the current labeling remains proper and the new canonical labeling and the new highest-gain tree can be computed using Dijkstra's shortest-paths algorithm.) In the worst case, all variants of operation UPDATEFLOW have the same performance: they reduce $\text{TOTRESEX}_{f,\mu,\Delta}$ only by 1 and they have the same worst-case asymptotic time bound. Thus all variants of Goldfarb, Jin and Orlin's algorithm presented in this paper have the same worst-case asymptotic time bounds, but their actual performances are considerably different. The idea underlying the proposal of new variants of the original algorithm was to maximise the usage of the current highest-gain tree, since the computation of the new one is a relatively costly procedure.

To show that a particular variant of operation UPDATEFLOW decreases $\text{TOTRESEX}_{f,\mu,\Delta}$ at least by 1, we use the following two straightforward lemmas.

Lemma 3. *If the pre-conditions of operation* EDGEFLOW(v,w,k) *are satisfied, then this operation does not increase* $\text{TOTRESEX}_{f,\mu,\Delta}$.

Proof. This operation sends $k\Delta$ units of flow from node v and at most $k\Delta$ units of flow into node w (the difference stays at the intermediate nodes $[v,w]$ and $[w,v]$). This means that $\lfloor e_{f,\mu}(v)/\Delta \rfloor$ decreases by k, $\lfloor e_{f,\mu}(w)/\Delta \rfloor$ does not increase by more than k and $\lfloor e_{f,\mu}(x)/\Delta \rfloor$ remains the same for each $x \in V \setminus \{v,w\}$, so $\text{TOTRESEX}_{f,\mu,\Delta}$ cannot increase. ■

Lemma 4. *If the pre-conditions of operation* EDGEFLOW(v,w,k) *are satisfied, then this operation decreases* $\text{TOTRESEX}_{f,\mu,\Delta}$ *by k, if $w = t$ or $e_{f,\mu}(w) < \Delta$ right after the execution of this operation.*

Proof. In both cases $\lfloor e_{f,\mu}(v)/\Delta \rfloor$ decreases by k. If $w \neq t$ and right after the execution of this operation $e_{f,\mu}(w) < \Delta$, then $\lfloor e_{f,\mu}(w)/\Delta \rfloor$ does not change. ■

We describe now the four variants of Goldfarb, Jin and Orlin's algorithm, defined by four variants of operation UPDATEFLOW. Each variant of operation UPDATEFLOW decreases $\text{TOTRESEX}_{f,\mu,\Delta}$ at least by 1 (as required for the claim that there are only $O(n+m)$ applications of operation UPDATEFLOW during each phase) because it performs at least one operation EDGEFLOW(v,w,k) of the type specified in Lemma 4 for $k \geq 1$. One can easily verify using Lemma 2 that Condition (10) is an invariant of each variant of operation UPDATEFLOW (we leave this to the reader), so whichever variant is used,

UPDATEFLOW0:
  $v \leftarrow$ a node in $v \in V_f \setminus \{t\}$ such that $e_{f,\mu}(v) \geq \Delta$;
  PATHFLOW$(v, 1)$.

PATHFLOW(v, k):
  $r \leftarrow v$;
  repeat
    EDGEFLOW$(r, \mathit{next}(r), k)$;
    $r \leftarrow \mathit{next}(r)$;
  until $r = t$ or $e_{f,\mu}(r) < \Delta$.

Figure 5: Operation UPDATEFLOW0 used by Goldfarb, Jin and Orlin's excess scaling algorithm [8] (algorithm EXSC0).

UPDATEFLOW1:
  $v \leftarrow$ a node in $v \in V_f \setminus \{t\}$ such that $e_{f,\mu}(v) \geq \Delta$;
  PATHFLOW$(v, 1)$;
  $c \leftarrow$ the residual capacity of the highest-gain path from v to t;
  $k \leftarrow \lfloor \min\{e_{f,\mu}(v),\, c\}/\Delta \rfloor$;
  if $k \geq 1$ **then** PATHFLOW(v, k).

Figure 6: Operation UPDATEFLOW1 used by algorithm EXSC1.

the total relabeled residual excess is reduced at least by half during each scaling phase, as discussed above.

Algorithm EXSC0

Algorithm EXSC0 is exactly the algorithm proposed by Goldfarb, Jin and Orlin [8]. The pseudocode of operation UPDATEFLOW0, the variant of operation UPDATEFLOW used by this algorithm, is shown in Figure 5. Operation UPDATEFLOW0 selects an arbitrary node v which has the residual excess at least Δ and sends Δ units of flow from v towards the sink t along the path in the current highest-gain tree, applying operation EDGEFLOW$(r, \mathit{next}(r), 1)$ to the consecutive nodes r on this path. Node $\mathit{next}(r)$ is the parent of a node r in the current highest-gain tree. The computation ends when the sink t has been reached or the residual excess at the current node is less than Δ. Operation UPDATEFLOW0 decreases TOTRESEX$_{f,\mu,\Delta}$ at least by 1 because the last application of operation EDGEFLOW is of the type specified in Lemma 4 with $k = 1$.

Algorithm EXSC1

Our first variant of Goldfarb, Jin and Orlin's algorithm is obvious and straightforward, and should actually be considered as the natural way of implementing this algorithm. The tree path from the selected node v to the sink t may have a large residual capacity, which allows sending a multiplicity of Δ units in one go. Operation UPDATEFLOW1 used by algorithm EXSC1 first sends Δ units of flow from the selected node v towards the sink t along the tree path in the same way as operation UPDATEFLOW0 does, with the only difference that the remaining residual capacity c of the used path is also computed. If $c > 0$, then Δ units of flow have been sent from node v to the sink t. If $k = \lfloor \min\{e_{f,\mu}(v),\, c\}/\Delta \rfloor \geq 1$, then there is the second pass over the same path which sends $k\Delta$ units of flow from node v to the sink t. The pseudocode of operation UPDATEFLOW1 is shown in Figure 6. One can check that there is always one or two applications of operation EDGEFLOW of the type specified in Lemma 4 (the last application of operation EDGEFLOW in each pass), so operation UPDATEFLOW1 decreases TOTRESEX$_{f,\mu,\Delta}$ at least by 1.

```
UPDATEFLOW2:
    v ← a node in v ∈ V_f \ {t} such that e_{f,μ}(v) ≥ Δ;
    r ← v;
    repeat
        EDGEFLOW(r, next(r), 1);
        if u_{f,μ}(r, next(r)) ≥ Δ and e_{f,μ}(r) ≥ Δ then
            EDGEFLOW(r, next(r), ⌊min{u_{f,μ}(r, next(r)), e_{f,μ}(r)}/Δ⌋);
        r ← next(r);
    until r = t or e_{f,μ}(r) < Δ.
```

Figure 7: Operation UPDATEFLOW2 used by algorithm EXSC2.

```
UPDATEFLOW3:
    L ← list of the nodes in V_f \ {t} in a tree order according to the current highest-gain tree;
    while L is not empty do
        v ← first node in L;  remove v from L;
        if e_{f,μ}(v) ≥ Δ then
            EDGEFLOW(v, next(v), min{⌊e_{f,μ}(r)/Δ⌋, ⌈(u_{f,μ}(r, w) − e_{f,μ}(r, w))/Δ⌉}).
```

Figure 8: Operation UPDATEFLOW3 used by algorithm EXSC3.

Algorithm EXSC2

Similarly as the previous two variants of operation UPDATEFLOW, the variant UPDATEFLOW2, used by algorithm EXSC2, selects an arbitrary node $v \in V_f \setminus \{t\}$ which has the residual excess at least Δ and sends off flow from v towards the sink t along the tree path. The difference is that previously the amount of flow sent off from each consecutive node r of the path was the same (Δ units in operation UPDATEFLOW0 and $k\Delta$ units, for some integer $k \geq 1$, in operation UPDATEFLOW1), while operation UPDATEFLOW2 sends from a node r into edge $(r, \mathit{next}(r))$ first Δ units of flow and then additional $\Delta \cdot \lfloor \min\{u_{f,\mu}(r, \mathit{next}(r)), e_{f,\mu}(r)\}/\Delta \rfloor$) units of flow. Thus different amount of flow may be sent into different edges of the path. The amount of flow sent into an edge $(r, \mathit{next}(r))$ is relative to the residual excess at node r and the residual capacity of this edge. The pseudocode of operation UPDATEFLOW2 is shown in Figure 7. The one or the two applications of operation EDGEFLOW to the last edge are of the type specified in Lemma 4, so operation UPDATEFLOW2 decreases $\mathrm{TOTRESEX}_{f,\mu,\Delta}$ at least by 1.

We should note that this heuristic, which sends into consecutive edges of the path (almost) as much flow as possible, does not always lead to faster actual performance. As the results of our experiments presented in Section 7 show, code EXSC2 performs significantly better than code EXSC1 on networks of one of the two classes of input networks which we have experimented with, but it performs somewhat worse on networks of the other class.

Algorithm EXSC3

Operation UPDATEFLOW3, used by algorithm EXSC3, is our final variant of operation UPDATEFLOW. It takes the most advantage of the current highest-gain tree and has performed best of all variants in all our experiments. Operation UPDATEFLOW3 first puts the nodes of the set $V_f \setminus \{t\}$ into a list L in an order defined by the current highest-gain tree: for each $r \in V_f \setminus \{t\}$, r is in L before $\mathit{next}(r)$. Next the nodes are taken one by one from L, and if the residual excess at the current node v is at least Δ, then flow is send into edge $(v, \mathit{next}(v))$ by applying operation $\mathrm{EDGEFLOW}(v, \mathit{next}(v), k)$ with the largest possible k allowed by Lemma 2. The details of operation UPDATEFLOW3 are shown in Figure 8.

Operation UPDATEFLOW3 uses all edges of the current highest-gain tree, while all three previous variants of operation UPDATEFLOW use only the edges of one path of this tree. The last application

```
{ INPUT: a generalised flow network G and a number ξ > 0 }
f ← zero flow in G;
k ← 0;
loop
   k ← k + 1;
   cancel all flow generating cycles in G_f and compute the canonical labeling μ;
   if TotResEx ≤ ξ e_f(t) then terminate;
   Phase(ε_k);      { ε_k is the value of the rounding parameter for the k-th phase }
end_loop
{ OUTPUT: ξ-optimal flow f }.

Phase(ε):
   { G_f does not have flow generating cycles, μ is the canonical labeling in G_f }
   b ← (1 + ε)^{1/n};
   let γ̃_μ(v, w) be γ_μ(v, w) rounded down to integer power of b, for each residual edge (v, w) in G_f
   while exists v ∈ G_f with positive excess do
      if exists an admissible edge (v, w) { a residual edge in G_f with γ̃_μ(v, w) > 1 } then
         push(v, w);     { update f by sending min{e_f(v), u_f(v, w)} units of flow along (v, w) }
      else
         relabel(v);    { μ(v) ← μ(v)/b^{1/n} }
   end_while
   { [e_opt(t) − e_{f''}(t)] ≤ [ε/(1 + ε)] · [e_opt(t) − e_{f'}(t)],
     where f' and f'' are the flows at the beginning and at the end of the computation }
```

Figure 9: Tardos and Wayne's algorithm PushRelabel.

of operation EdgeFlow is of the type specified in Lemma 4, so operation UpdateFlow3 decreases $\text{TotResEx}_{f,\mu,\Delta}$ at least by 1. However, unlike the previous variants, one operation UpdateFlow3 may perform many operations EdgeFlow of the type specified in Lemma 4, so may substantially decrease $\text{TotResEx}_{f,\mu,\Delta}$. This seems to be the reason why algorithm ExSc3 outperforms the other variants in our experiments.

5 Push-relabel algorithm

Tardos and Wayne [17] (see also [20]) proposed an algorithm for the maximum generalised flow problem based on Goldberg and Tarjan's push-relabel method for the minimum-cost (non-generalised) flow problem [6]. The pseudocode of Tardos and Wayne's algorithm, which we call in this paper algorithm PushRelabel, is shown in Figure 9. The computation of this algorithm is a sequence of applications of procedure Phase(ϵ), where ϵ is a parameter used in the calculations of rounding gain factors of edges.

At the beginning of the computation of procedure Phase(ϵ), the current flow in network G is such that the residual network G_f is a non-gain network and the canonical labeling μ of network G_f has been computed. The computation starts with rounding down the relabeled gain factors γ_μ of the residual edges to integer powers of $b = (1+\epsilon)^{1/n}$. Let $\tilde{\gamma}_\mu$ denote the rounded gain factors. A node $v \in V_f \setminus \{t\}$ is an *active node*, if it has positive residual excess. An edge (v,w) is an *admissible edge*, if it is a residual edge and $\tilde{\gamma}_\mu(v,w) > 1$. The way the flow and the labeling is updated is analogous to Goldberg and Tarjan's push-relabel method for non-generalised flows: keep sending flow from active nodes along admissible edges, and if there is no admissible edge outgoing from an active node v, decrease its label $\mu(v)$ to increase the gain factors of the outgoing edges. More specifically, procedure Phase(ϵ) uses two operations, push(v,w) and relabel(v). Operation push(v,w) applies if a node v is active and an edge (v,w) is admissible, and it sends $\min\{e_f(v), u_f(v,w)\}$ units of flow along this edge. Operation RELABEL(v) applies if a node v is active and there are no admissible edges outgoing from v, and it decreases the label $\mu(v)$ by factor $b^{1/n}$ (so the gain factors of the edges outgoing from node v increase by factor $b^{1/n}$). Procedure PHASE(ϵ) repeatedly selects an active node and applies either operation PUSH(v, w), if there is an admissible edge (v, w), or operation RELABEL(v) otherwise. The computation terminates when no active node is left in the residual network G_f.

Tardos and Wayne [17, 20] show that the running time of procedure PHASE(ϵ) is $\tilde{O}(mn^3\epsilon^{-1}\log B)$[1], and if f' and f'' denote the flows at the beginning and at the end of the computation, then

$$e_{\text{opt}}(t) - e_{f''}(t) \;\leq\; \frac{\epsilon}{1+\epsilon}(e_{\text{opt}}(t) - e_{f'}(t)). \tag{13}$$

Since the rounded gain factors are used, the flow is not necessary sent along the highest-gain paths, so flow-generating cycles may be created. Therefore, before the computation proceeds to the next phase, we have to cancel all flow-generating cycles. Goldberg, Plotkin and Tardos [4] show that this can be done in $\tilde{O}(mn^2\log B)$ time by an adaptation of Goldberg and Tarjan's algorithm for the minimum-cost (non-generalised) flow problem which repeatedly cancels minimum mean-cost cycles [5].

The property (13) implies that the flow f at the end of the q-th iteration of algorithm PUSHRELABEL is such that

$$e_{\text{opt}}(t) - e_f(t) \;\leq\; \frac{\epsilon_1\epsilon_2\cdots\epsilon_q}{(1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_q)}\, e_{\text{opt}}(t),$$

where ϵ_i is the actual parameter of the i-th application of procedure PHASE. Hence this flow is ξ-optimal for

$$\xi \;=\; \frac{\epsilon_1\epsilon_2\cdots\epsilon_q}{(1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_q) - \epsilon_1\epsilon_2\cdots\epsilon_q}. \tag{14}$$

Tardos and Wayne [17, 20] repeatedly apply procedure PHASE with the same $\epsilon = 1/2$ to obtain the best asymptotic worst-case time bound for computing ξ-optimal flows. In this case, (14) implies that $O(\log\xi^{-1})$ phases suffice, so the total running time is $\tilde{O}(mn^3\log B\log\xi^{-1})$. Our experiments showed that the computation of PHASE(1/2) is too slow to give good performance of algorithm PUSHRELABEL. When we used instead $\epsilon = \Theta(n)$, then the performance considerably improved. Such values of parameter ϵ increase the worst-case asymptotic bound on the number of applications of procedure PHASE needed to obtain a ξ-optimal flow by an $O(n)$ factor to $O(n\log\xi^{-1})$, but they decrease the worst-case asymptotic bound on the running time of procedure PHASE only by an $O(\log n)$ factor. In our experiments, however, the trade-off goes the other way: the running time of one phase considerably decreased, while usually only two or three calls to procedure PHASE were sufficient to obtain 10^{-5}-optimal flows.

Our implementation and algorithm PUSHRELABEL differs from the algorithm presented by Tardos and Wayne [17, 20] also in the following way. We use the rounded gain factors only to identify admissible edges, but we always keep updating the current flow in the original network G_f, that is, using the original exact gain factors. Tardos and Wayne update the flow in the rounded network, and at the end of phase interpret the updates in the original network. When all updates are made in the original network, then more flow is sent to the sink in each phase, resulting in a better convergence of the computation. One can check that the analysis of the PUSHRELABEL algorithm presented by Tardos and Wayne in [17] and [20] remains valid for this modification. Since we have not developed yet a fast code for canceling all flow generating cycles, this computation is perform in our code PUSHRELABEL by a CPLEX optimiser.

6 Generators of generalised flow networks

We developed, and used in our tests and experiments, two simple generators of generalised flow networks, which are loosely based on possible applications of the maximum generalised flow problem in the quantitative financial analysis. To simplify our initial experiments, the generated networks do not have flow generating cycles. In our experiments we used uniform and various non-uniform distributions.

[1] Notation $\tilde{O}()$ hides a factor polynomial in $\log n$.

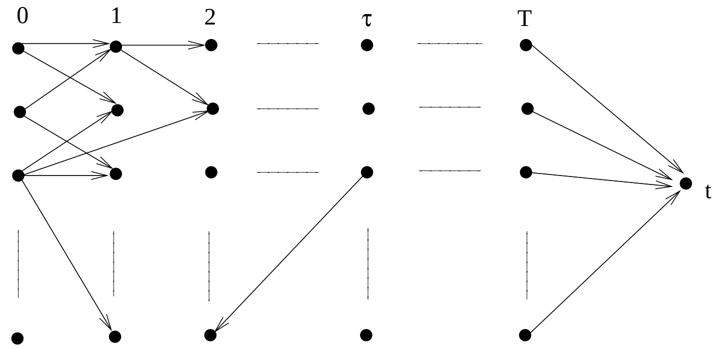


Figure 10: The structure of networks generated by the input generator LAYERSX.

Layered network: a multiperiod portfolio management model

Generator LAYERS and its variant LAYERSX have parameters K, T, D, $\gamma_{\min}$, $\gamma_{\max}$, $u_{\min}$ and $u_{\max}$, and create a network with the set of nodes

$$V = \{[b, \tau] : b = 1, 2, \ldots, K; \tau = 0, 1, \ldots, T\} \cup \{t\}.$$

A node $[b, \tau]$ represents an asset b at time τ. An edge $e = ([b', \tau], [b'', \tau + 1])$ with gain $\gamma(e)$ and capacity $u(e)$ represents possibility of exchanging $x \leq u(e)$ units of asset b' available at time τ into $\gamma(e) \cdot x$ units of asset b'', which will be available at the next time $\tau + 1$. Each node $[b, T]$ is connected by an edge to the sink t. Generator LAYERS connects each node $[b, \tau]$, $\tau < T$, to D randomly chosen nodes at the next time point $\tau + 1$. Generator LAYERSX connects each node $[b, \tau]$, $\tau \leq T$, to D randomly chosen nodes other than the nodes at the same time point τ. A *forward edge* from a time point τ to a time point $\tau' > \tau$ represents exchanging assets, with delayed availability of the purchased asset if $\tau' > \tau + 1$. A *back edge* $e = ([b', \tau'], [b'', \tau''])$ to a time point $\tau'' < \tau'$ represents borrowing. The flow value x on such an edge means that we get $x \cdot \gamma(e)$ units of asset b'' at the time point τ'' but we have to return x units of asset b' at the later time τ'. Figure 10 shows the structure of networks generated by generator LAYERSX.

The gains and the capacities of the edges are randomly selected from the intervals $[\gamma_{\min}, \gamma_{\max}]$ and $[u_{\min}, u_{\max}]$, respectively. Only nodes $[b, 0]$ have positive initial excesses, which are randomly selected from the interval $[Du_{\min}, Du_{\max}]$. The maximisation of the flow into the sink t models the maximisation of the total value of the assets held at the final time point T. One should expect that in the multiperiod portfolio model the gain factors of edges should be close to 1, assuming that the input data comes in a normalised form (that is, an initial labeling of nodes is given which normalises the units of the assests), so in our experiments we usually set parameters $\gamma_{\min}$ and $\gamma_{\max}$ close to 1; say 0.9 and 1.1 in generator LAYERS and 0.9 and 1.0 in generator LAYERSX (to avoid flow generating cycles).

Grid of cliques: exchanging and transferring currencies

The generator GRIDOFCLIQUES has parameters K, Q, $\gamma_{\min}$, $\gamma_{\max}$, $u_{\min}$, $u_{\max}$, $e_{\min}$, and $e_{\max}$, and creates a network with the set of nodes

$$V = \{[c, q] : c = 1, 2, \ldots, K;\ q = 1, 2, \ldots, Q\}.$$

The sink t is an arbitrarily chosen node in V. A node $[c, q]$ represents a currency c at a market q. For each $q = 1, 2, \ldots, Q$, the nodes $\{[c, q] : c = 1, 2, \ldots, K\}$ form a (directed) clique; and for each $c = 1, 2, \ldots, K$, the nodes $\{[c, q] : q = 1, 2, \ldots, Q\}$ form a clique. An edge $([c', q], [c'', q])$ represents possibility of exchanging currency c' for other currency c'' within the same market q. An edge $e = ([c, q'], [c, q''])$ represents possibility of transferring the same currency c from market q' to other market q'', and $1 - \gamma(e)$ is the cost of such transfer per unit of the currency. The capacities and

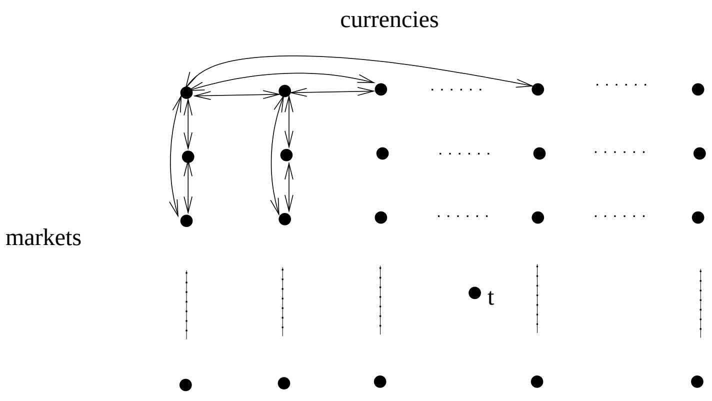


Figure 11: The structure of networks generated by the input generator GridOfCliques.

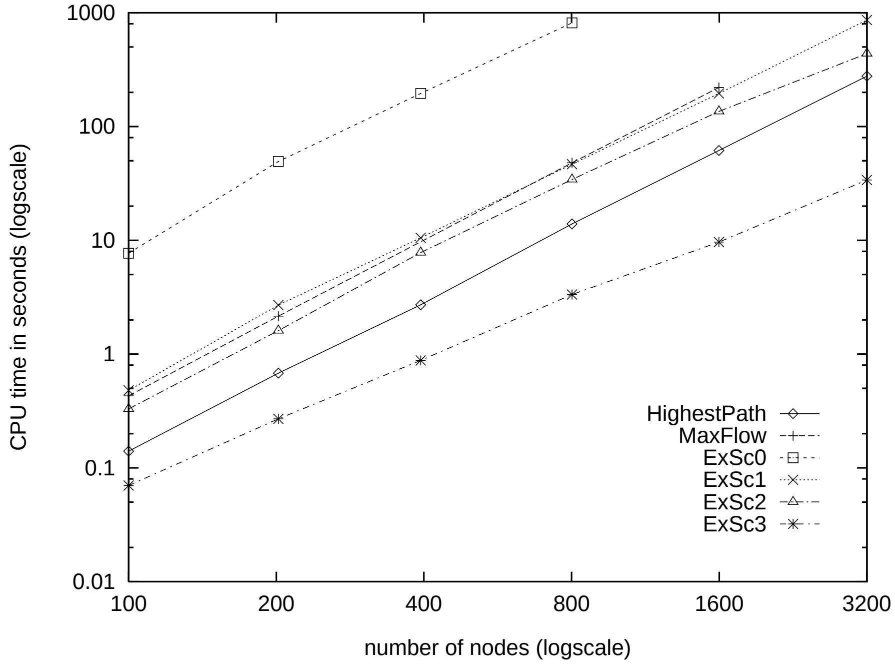
gain factors are randomly selected as in generator LayersX. The initial node excesses are randomly chosen from the interval $[e_{\min}, e_{\max}]$. The maximisation of the flow into the sink $t = [c_t, q_t]$ is meant to model the computation of an optimal way of transferring and exchanging various currencies held at various markets into the currency c_t at the market q_t. Figure 11 shows the structure of networks generated by generator GridOfCliques.

7 Experiments

We present in this section results of some initial experiments with our implementations. All implementations were developed in C++ (gnu g++ version 2.8) using LEDA, version 4.0, a C++ library of data types and algorithms [11]. We conducted all experiments on Sun UltraSPARC II (2×300 MHz, 512 MB, Unix SunOS 5.5.1). Each running time shown in the tables is the average time in seconds of 5 runs on networks of the same type and size, that is, networks generated by the same input generator, with the same parameters, but with different random seeds.

Figures 12, 13 and 14 compare the implementations of different versions of Goldfarb, Jin and Orlin's algorithm described in Section 4 and implementations of Onaga's and Truemper's algorithms (codes HighestPath and MaxFlow, respectively) on networks generated by input generators Layers, LayersX and GridOfCliques. All codes were run until 10^{-5}-optimal solutions were computed. Figures 12 and 13 show that our successive variants of Goldfarb, Jin and Orlin's algorithm lead to progressively better performance on the Layers and LayersX networks, and the final variant ExSc3 is considerably faster than code HighestPath, the implementation of Onaga's algorithm. Figure 14 shows that codes ExSc3 and HighestPath are the fastest on networks GridOfCliques, but the differences in performance of all codes are less dramatic and, contrary to our expectations, heuristic ExSc2 does not improve on heuristic ExSc1.

Figure 15 compares the performance of code ExSc3, our fastest variant of Goldfarb, Jin and Orlin's algorithm, and code PushRelabel against the performance of the commercial linear-programming package CPLEX [10]. CPLEX has three basic optimisers: primal simplex, dual simplex and barrier. The barrier optimiser is based on the interior-point method. We include in the tables only the average running times of the primal and the dual simplex optimisers because the barrier optimiser was always the slowest. The running times of codes ExSc3 and PushRelabel shown in the tables are for computing 10^{-5}-optimal solution, while the running times of the CPLEX optimisers are for computing exact solutions. We monitored the computation of the primal simplex optimiser, which generates a sequence of progressively improved primal feasible solutions, and observed that the time needed to obtain a 10^{-5}-optimal solution was usually equal to 80%-90% of the time needed to reach the final optimal solution. The tables in Figure 15 show that codes ExSc3 and PushRelabel perform worse than CPLEX, but the dominance of CPLEX should not be considered overwhelming, especially if we take into account possibilities for further improvements of our implementations (see Section 8).



nodes/edges	HighestPath	MaxFlow	ExSc0	ExSc1	ExSc2	ExSc3
100/400	0.11	0.34	7.69	0.48	0.33	0.07
200/800	0.54	1.73	49.22	2.70	1.61	0.27
400/1600	2.17	8.52	195.10	10.54	7.80	0.88
800/3200	11.16	34.42	815.46	46.68	34.23	3.34
1600/6400	49.32	176.24	-	195.66	135.76	9.67
3200/12800	221.86	-	-	862.73	437.79	33.94

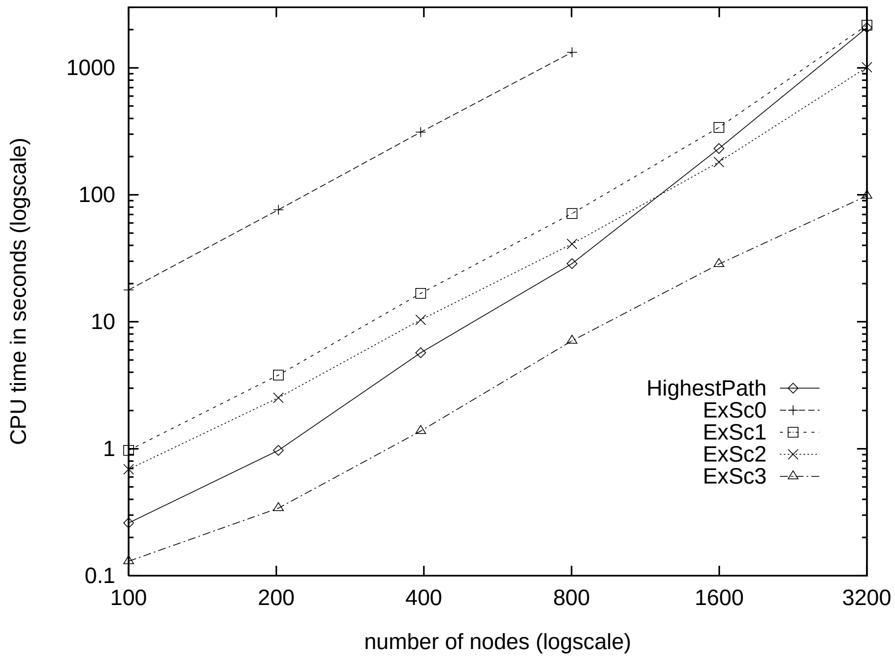
Figure 12: Running times on inputs created by the generator Layers.

Table (c) shows that on the LayersX networks the running time of our code ExSc3 grows, with the growth of the size of networks, somewhat slower than the running times of the CPLEX optimisers. Thus, even without any further improvements, code ExSc3 may be faster than the CPLEX optimisers on networks a few hundred times larger than the largest networks included in Table (c). (The limits of our current system platform makes it difficult to extend experiments to such large networks.)

8 Conclusion

We have developed implementations for the maximum generalised network flow problem based on recently proposed polynomial-time combinatorial algorithms: Goldfarb, Jin and Orlin's [8] excess scaling algorithm (codes ExSc) and Tardos and Wayne's [17] push-relabel algorithm (code PushRelabel). Since our initial implementations which closely followed the original descriptions of these algorithms turned out to be very slow, it was necessary to investigate the details of design and analysis of the algorithms to come up with promising modifications. The performance of our final codes ExSc3 and PushRelabel comes close enough to the performance of CPLEX to postulate that combinatorial generalised network flow algorithms, in combination with the ideas underlying general-purpose linear programming methods, will soon lead to practically fast optimisers for the maximum generalised network flow problem.

The running time of codes ExSc is overwhelmingly dominated by the running time of the computations of the highest-gain trees (Dijkstra's single-source shortest-path computations). Thus an



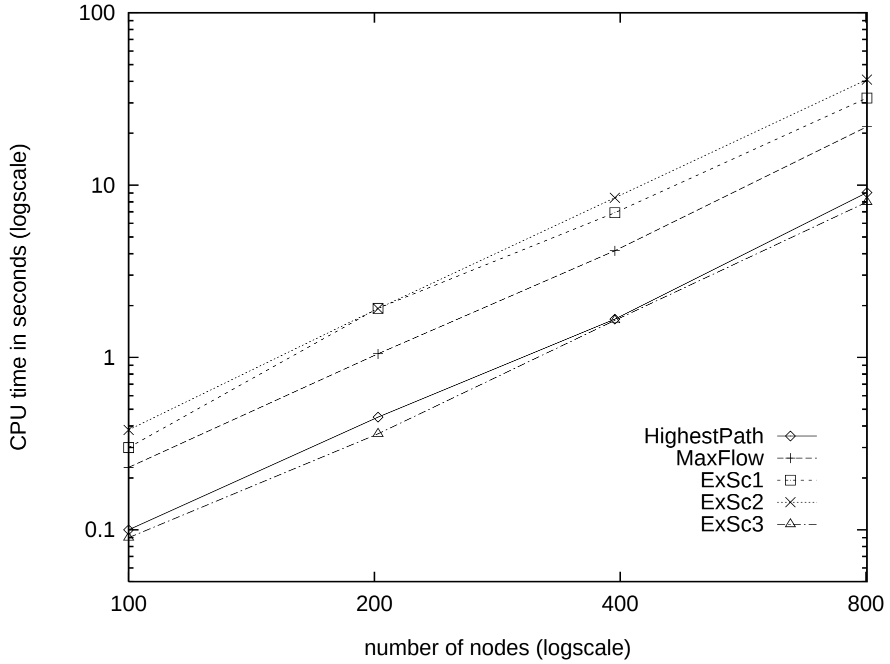
nodes/edges	HighestPath	ExSc0	ExSc1	ExSc2	ExSc3
100/500	0.21	14.26	0.97	0.69	0.13
200/1000	0.78	61.03	3.80	2.52	0.34
400/2000	4.56	249.86	16.72	10.36	1.38
800/4000	22.95	1061.11	71.29	41.02	7.08
1600/8000	185.50	-	339.45	181.22	28.46
3200/16000	1659.97	-	2169.16	1009.39	98.24

Figure 13: Running times on inputs created by the generator LayersX. (MaxFlow code gives very high running times even for small networks generated by LayersX.)

obvious direction in speeding-up these codes is to replace the computation of the new highest-gain tree from scratch with a suitable practically fast update of the previous tree. Another direction is to come up with some mechanism of identifying "inactive" edges during the computation, that is, the edges which will never again, or at least not for a long time, have their flow values updated. In our experiments most of the edges quickly become inactive, so removing such edges from the network may considerably speed-up the computation.

The push-relabel method seems to offer more room for improvement. Our code PushRelabel implements only the basic push-relabel strategy which maintains the active nodes in a FIFO queue, while there exist a number of different variants and heuristics for the push-relabel method developed for non-generalised network flows. For example, it has been demonstrated that in the context of non-generalised flows, periodical global re-calculation of the node labels may considerably improve the actual running times. One should expect similar improvements from an analogous heuristic in the context of maximum generalised flows.

The cumbersome part of the push-relabel method for the maximum generalised flow problem is the canceling of all flow generating cycles, which is required before each phase. This computation does not have an analogue in the push-relabel method for the minimum-cost non-generalised flow problem. We use a CPLEX optimiser for this computation in our code PushRelabel. One could contemplate using only approximate canceling of flow generating cycles, following the way Radzik [16] and Tardos and Wayne [17] used such computation in their variants of Goldberg, Plotkin and Tardos'



nodes/edges	HighestPath	MaxFlow	ExSc0	ExSc1	ExSc2	ExSc3
100/1000	0.08	0.18	38.16	0.30	0.38	0.09
200/3000	0.36	0.84	341.18	1.93	1.92	0.36
400/8000	1.34	3.33	-	6.91	8.44	1.64
800/23000	7.24	17.47	-	32.00	40.93	7.98

Figure 14: Running times on inputs created by the generator GridOfCliques.

fat-path algorithm [4]. However, details and a theoretical analysis of a possible push-relabel maximum generalised flow algorithm which uses only approximate canceling of flow generating cycles are yet to be worked out.

The push-relabel method for the minimum-cost non-generalised flow problem removes the negative cycles (the "flow generating" cycles in that context) by simply saturating all edges which have negative reduced costs. Analogously, we could get rid of all flow generating cycles by saturating all edges which have relabeled gains greater than 1. This method creates negative node excesses and therefore does not fit into Tardos and Wayne's push-relabel framework for the maximum generalised flow problem presented in [17]. We believe, however, that the push-relabel framework can be extended to handle negative node excesses, and such an extension should lead to improved practical performance.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, algorithms, and applications.* Prentice Hall, 1993.

[2] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[3] F. Glover, J. Hultz, D. Klingman, and J. Stutz. Generalized networks: A fundamental computer-based planning tool. *Management Science*, 24(12), August 1978.

[4] A. V. Goldberg, S. A. Plotkin, and É. Tardos. Combinatorial Algorithms for the Generalized Circulation Problem. *Math. Oper. Res.*, 16(2):351–381, 1991.

(a)

nodes/edges	ExSc3	PushRelabel	CPLEX-p	CPLEX-d
100/500	0.11	0.06	0.03	0.04
200/1000	0.37	0.19	0.08	0.14
400/2000	1.13	0.78	0.22	0.39
800/4000	5.78	2.68	0.72	1.54
1600/8000	27.64	16.80	2.35	5.00
3200/16000	122.82	102.37	31.32	16.00

(b)

nodes/edges	ExSc3	PushRelabel	CPLEX-p	CPLEX-d
100/1000	0.08	0.09	0.03	0.06
200/3000	0.21	0.27	0.11	0.27
400/8000	1.45	2.06	0.26	1.37
800/23000	7.16	11.95	1.89	9.73

(c)

nodes/edges	ExSc3	PushRelabel	CPLEX-p	CPLEX-d
6000/20000	105	32	35	29
9000/30000	193	51	57	68
12000/40000	287	91	111	120
15000/50000	389	145	145	167
18000/60000	523	311	210	279

Figure 15: Comparison of codes ExSc3 and PushRelabel with CPLEX. ExSc3 and PushRelabel compute 10^{-5}-optimal flows. CPLEX computes optimal flows. (a) Small LayersX networks. (b) GridOfCliquesnetworks. (c) Large LayersX networks.

[5] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. Assoc. Comput. Mach.*, 36:388–397, 1989.

[6] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15:430–466, 1990.

[7] D. Goldfarb and Z. Jin. A faster combinatorial algorithm for the generalized circulation problem. *Math. Oper. Res.*, 21:529–539, 1996.

[8] D. Goldfarb, Z. Jin, and J. Orlin. Polynomial-time highest-gain augmenting path algorithms for the generalized circulation problem. *Math. Oper. Res.*, 22:793–802, 1997.

[9] S. Kapoor and P. M. Vaidya. Speeding up Karmarkar's algorithm for multicommodity flows. *Math. Programming*, 73(1):111–127, 1996.

[10] ILOG CPLEX Division, Incline Village, NV. *ILOG CPLEX 6.5, User's Manual.* March 1999.

[11] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometrical Computing.* Cambridge University Press, 1999.

[12] S. M. Murray. *An interior point approach to the generalized flow problem with costs and related problems.* PhD thesis, Stanford Univ., August 1992.

[13] K. Onaga. Dynamic Programming of Optimum Flows in Lossy Communication Nets. *IEEE Trans. Circuit Theory*, 13:308–327, 1966.

[14] K. Onaga. Optimal Flows in General Communication Networks. *J. Franklin Inst.*, 283, 1967.

[15] J. B. Orlin. A Faster Strongly Polynomial Algorithm for the Minimum Cost Flow Problem. *Oper. Res.*, 41:338–350, 1993.

[16] T. Radzik. Faster algorithms for the generalized network flow problem. *Math. Oper. Res.*, 23(1):69–100, 1998.

[17] É. Tardos and K.D. Wayne. Simple generalized maximum flow algorithms. In *Proc. 6th International Conference on Integer Programming and Combinatorial Optimization*, pages 310–324, 1998.

[18] K. Truemper. On Max Flows with Gains and Pure Min-Cost Flows. *SIAM J. Appl. Math.*, 32:450–456, 1977.

[19] P. M. Vaidya. Speeding up Linear Programming Using Fast Matrix Multiplication. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 332–337, 1989.

[20] K.D. Wayne. *Generalized maximum flow algorithms.* PhD thesis, Cornell University, January 1999.