Eötvös Loránd University

Faculty of Science

Julianna Bor

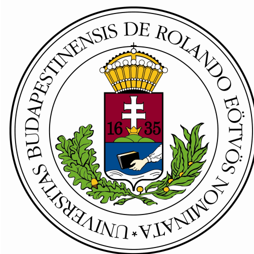
Strongly polynomial algorithms for minimum cost network flow problems

BSc Thesis

Advisor:

Alpár Jüttner

Department of Operations Research



Budapest, 2016

Acknowledgements

I am grateful to my supervisor, Alpár Jüttner for introducing me to this topic and helping me throughout my work. I would also like to thank Alexander Bor and Gábor Salamon for language editing.

Contents

Introduction 4

1 Minimum cost network flow problem 6

1.1 Terminology and Definitions 6

1.2 Basic Theorems 7

2 Network simplex algorithms 8

2.1 Primal network simplex algorithm 9

2.2 Dual network simplex algorithm 12

3 Strongly polynomial algorithms 16

3.1 Strongly polynomial primal network simplex algorithm 16

3.1.1 Structure and properties 16

3.1.2 Complexity analysis 20

3.2 Strongly polynomial dual network simplex algorithm 24

3.2.1 Structure and properties 24

3.2.2 Complexity analysis 29

4 Summary 31

References 32

Introduction

The minimum cost network flow problem (MCNFP) plays a crucial role within the area of network optimization. Its applications appear in a wide range of fields, including physics, computer networking, engineering, and transportations. This problem is usually solved by applying the network simplex algorithm due to its simplicity and speed. Although the network simplex algorithm works excellent in practice the most commonly used pivot rules make an exponential number of pivots in the worst case.

The first strongly polynomial algorithm for the MCNFP was developed by Tardos in 1985 [1], and later Orlin in 1993 published an algorithm with a better time complexity [2].

Creating a specialization of the network simplex algorithm that runs in polynomial time has been proved to be a difficult problem. In this thesis I demonstrate, inter alia, the structure and properties of the first strongly polynomial time primal network simplex algorithm for minimum cost flows developed by Orlin in 1995 [3].

Nowadays, several methods are known to reach strongly polynomial running time for the MCNFP. There are two types of algorithms: *cycle cancelling* and *cut cancelling* which are used depending upon whether we work on the primal or the dual problem, respectively. The cycle cancelling algorithm searches for negative cost cycles in the residual network and sends as much flow around that cycles as possible thereby cancelling them. Meanwhile the cut cancelling algorithm is the cycle cancelling algorithms dual counterpart. For example Goldberg and Tarjan developed a minimum mean-cost cycle cancelling algorithm [4], which uses the cycle with the minimum $(c(W)/|W|)$ value to decrease the objective function. Using Karp's method the minimum mean cost cycle can be found in strongly polynomial time. Goldberg and Tarjan use this as a subroutine, and their algorithm finds a minimum cost flow within $\mathcal{O}(nm^2 \log n)$ iterations.

A widely used technique to increase the efficiency of an algorithm is the *scaling technique*. An essential part of this technique is to find an attribution that becomes fixed at some point during the algorithm ensuring the finiteness.

This paper describes a primal and a dual specialization of the network simplex

algorithm developed by James B. Orlin [3], Ronald D. Amstrong and Zhiying Jin [5], respectively.

Section 1 introduces the problem and mentions a couple basic theorems strongly related to this topic. Then the second section presents the main structure of both the primal and the dual algorithms. Finally, Section 3 describes the cost-scaling versions of the algorithms and shows that the primal algorithm runs in $\mathcal{O}(n^2m^2 \log n)$ time and the dual runs in $\mathcal{O}(mn(m + \log n) \log n)$ time.

1 Minimum cost network flow problem

1.1 Terminology and Definitions

Let $G = (V, E)$ be a directed graph with n nodes and m arcs. Each arc $(i, j) \in E$ has a lower bound l_{ij} and an upper bound u_{ij} on the flow running through it. c is the cost function $c : E \to \overline{\mathbb{R}}$, and b is the demand function $b : V \to \mathbb{R}$ with

$$\sum_{i \in V} b(i) = 0$$

The minimum cost flow problem can be stated as the following linear programming problem:

$$\min \sum_{(i,j) \in E} c_{ij} x_{ij}$$

subject to

$$\sum_{(i,j) \in E} x_{ij} - \sum_{(j,i) \in E} x_{ij} = b(i) \quad \forall i \in V$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in E$$

In the following algorithms, we will assume that $l \equiv 0$, $u \equiv +\infty$ and $0 \leq c$. This assumption can be made without loss of generality.

Lemma 1.1.1. *The linear program mentioned above can be transformed to the following form:*

$$\min \sum_{(i,j) \in E} c_{ij} x_{ij} \quad \textit{where} \quad 0 \leq c$$

subject to

$$\sum_{(i,j) \in E} x_{ij} - \sum_{(j,i) \in E} x_{ji} = b(i) \quad \forall i \in V \tag{P}$$

$$0 \leq x_{ij} \quad \forall (i,j) \in E$$

Proof. First, note that the criterion $0 \leq c$ can be easily satisfied. All we have to do is to reverse the arcs with negative costs, since the reversed arc's cost will be negative of the original cost.

For the $l \equiv 0$ assumption, we have to shift the lower bound as well as the upper bound with l_{ij} for each arc $(i, j) \in E$ so the new bounds will be: $l_{ij} \leftarrow 0$,

$u_{ij} \leftarrow u_{ij} - l_{ij}$. To keep to the original problem we also have to change the demand function:

$$b(i) \leftarrow b(i) - \sum_{(i,j)\in E} u_{ij} + \sum_{(j,i)\in E} u_{ij}$$

Now, if x is a feasible solution to the transformed problem then $x + u$ is a feasible solution to the original one.

Our last job is to relax the upper bounds. We will take in extra arcs and nodes for that as follows: let (i, j) be an original arc, instead of (i, j) we use the arcs $(i, 1), (2, 1), (2, j)$ where 1 and 2 mark the new nodes and we define $b(1) = -u_{ij}$ and $b(2) = u_{ij}$. The lower bounds stay at 0 and there are no upper bounds. Suppose that $0 \leq x_{i1}$ flow goes through $(i, 1)$. To satisfy the $b(1) = -u_{ij}$ demand (or supply) x_{21} must be $0 \leq x_{21} = u_{ij} - x_{i1}$ which means $x_{i1} \leq u_{ij}$. And to satisfy the $b(2) = u_{ij}$ demand (or supply) x_{2j} must be $0 \leq x_{2j} = u_{ij} - (u_{ij} - x_{i1}) = x_{i1}$. □

Definition 1.1.2. We call an algorithm strongly polynomial if both statements below are true:

1. the algorithm is a polynomial space algorithm
2. assuming that the elementary arithmetic operations take a unit step to perform regardless of the size of the input, it performs a polynomial number of operations (polynomial in the number of input numbers)

We are going to discuss two algorithms below. The first one is a primal network simplex algorithm and the second one is a dual network simplex algorithm. Both algorithms are strongly polynomial.

1.2 Basic Theorems

The algorithms refer to the following theorems many times:

Theorem 1.1 (Duality Theorem)**.** *Let us take the following primal problem:*

$$\max\{(c_0x_0 + c_1x_1) \mid Px_0 + Ax_1 = b_0,\ Qx_0 + Bx_1 \leq b_1,\ 0 \leq x_1\}$$

and the dual problem associated with it:

$$\min\{(y_0b_0 + y_1b_1) \mid y_0P + y_1Q = c_0,\ y_0A + y_1B \geq c_1,\ y_1 \geq 0\}$$

Suppose that neither polyhedrons are empty. The maximum value of the primal optimization problem is equal to the minimum value of the dual optimization problem. That is, if x^* is an optimal solution to the primal problem and y^* to the dual problem then $cx^* = y^*b$.

Theorem 1.2. *Suppose that the P polyhedron defined with the*

$$\{Px_0 + Ax_1 = b_0,\ Qx_0 + Bx_1 \leq b_1,\ 0 \leq x_1\}$$

system is not empty and $\{cx = c_0x_0 + c_1x_1 \mid x \in P\}$ is upper bounded. Then x^* is an optimal solution to the primal problem P if and only if there exists an $y^* = y_0^* + y_1^*$ vector which satisfies the dual problem:

$$y_0^*P + y_1^*Q = c_0,\ y_0^*A + y_1^*B \geq c_1,\ y_1^* \geq 0$$

And for a pair (x^*, y^*) the complementary slackness conditions are fulfilled. That is,

$$x_1^*(i) > 0 \implies y_0^*A_i + y_1^*B_i = c_1$$
$$y_1^*(i) > 0 \implies Q^i x_0^* + B^i x_1^* = b_1$$

Overview of the ensuing sections Section 2 describes a simplified version of the algorithms in order to understand the basic structure of them better. These run in pseudo polynomial time. Then we are going to improve the algorithms to achieve strongly polynomial running time. For this purpose in Section 3 we discuss the scaling variant of them.

2 Network simplex algorithms

This section presents the main structure of the algorithms. We start with the description of the primal simplex algorithm, then we continue with the explanation of the dual simplex algorithm.

2.1 Primal network simplex algorithm

As alluded to before, we use the form mentioned in Lemma 1.1.1. In the original articles the algorithms work on the capacitated problems which allows a practical implementation because the tree has n arcs instead of $n+2m$ arcs as in my interpretation. However, using the uncapacitated graph simplifies the description.

Non-degeneracy assumption We will assume that there are no 0 tree arcs in a feasible solution. This assumption can be made without loss of generality using a perturbation technique (see, for example [6]).

Definition 2.1.1. Auxiliary graph $G(x)$: for a given feasible solution x let $G(x)$ denote the set of all arcs of G and the reversed arcs of the non-zero arcs.

Definition 2.1.2. Let $\pi \in \mathbb{R}^n$ be a vector of node potentials. For each π, the **reduced cost** is: $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$. We call an x solution ε**-optimal** with respect to π if $c^{\pi}_{ij} \geq -\varepsilon \ \forall (i,j) \in G(x)$. (If x is 0-optimal then it is optimal.)

It is well known that $c(W) = c^{\pi}(W)$ for any directed cycle W, therefore minimizing the function $c^{\pi}x$ gives us the same result as minimizing cx.

A vector x is called a **feasible solution** if it satisfies the constraints (P) written in Lemma 1.1.1. To have an easily found initial basic feasible solution we add some artificial arcs: $(1,j)$ and their reversal where $j \neq 1$. The costs of these arcs are set to $c_{1j} = (n+1) * C$ where $C = max\ c_{ik}$. Thus, an optimal basic feasible solution will not use these arcs unless the problem is unsolvable.

We can give a basic feasible solution with tree T given that x is zero for the rest of the arcs. Furthermore, x is not optimal $\Leftrightarrow \exists$ basic cycle W with $c(W) < 0$.

In the following, we describe the premultiplier algorithm of James B. Orlin [3].

Definition 2.1.3. $T(v)$ is a spanning in-tree with route v. So every arc in $T(v)$ is directed towards v.

Definition 2.1.4. $\pi \in \mathbb{R}^n$ is a vector of premultipliers for tree T if there exists a node v for which $c^{\pi}_{ij} \leq 0$ for every arc $(i,j) \in T(v)$.

The following lemma is straightforward to proof.

Lemma 2.1.5. *Suppose that π is a vector of premultipliers with respect to node v. π is a vector of premultipliers with respect to node i if and only if $c^{\pi}_{kj} = 0$ for every arc on the path from i to v in $T(v)$.*

Definition 2.1.6 (Eligible nodes and arcs)**.** Suppose that π is a vector of premultipliers for tree T. We call node i **eligible** if π is a vector of premultipliers with respect to $T(i)$ as well. And (i, j) is an **eligible arc** if i is an eligible node and $c^{\pi}_{ij} < 0$.

Lemma 2.1.7. *Let T be a tree and π is a vector of premultipliers with respect to $T(v)$. Then the basic cycle induced by any eligible arc has a negative cost.*

Proof. Let (i, j) be an eligible arc and W is the basic cycle induced by (i, j). Let t be the closest node to the root v on the path from j to i in the tree. The path from j to t in W remains unchanged hence the reduced cost of these arcs is non-positive. On the path from t to i every arc is reversed. Since (i, j) is an eligible arc i is an eligible node by definition, so every arcs had a zero reduced cost there. Furthermore $c^{\pi}_{ij} < 0$ also by definition. That is, the reduced cost of W is strictly less than zero and $c\,(W) = c^{\pi}(W) < 0$. □

Now, we describe the **premultiplier algorithm** by Orlin [3].

begin
  choose an initial basic feasible solution x and a vector of premultipliers π with respect to $T(v)$.
  while x is not optimal **do**
    if there is an eligible arc **then**
    begin
      choose an eligible arc (k, l);
      Call procedure SIMPLEX PIVOT(k, l);
    end;
    else
      Call procedure MODIFY-PREMULTIPLIERS;
end;

procedure SIMPLEX PIVOT(k, l);

begin

W is the basic directed cycle induced by (k, l) in $G(x)$

$\delta := \{\min x_{ji} \mid (i,j) \in W \wedge (j,i) \in G\}$

send δ units of flow around W;

let (p, q) be the arc pivoted out;

reset the root of the tree to be p;

end;

procedure MODIFY-PREMULTIPLIERS;

begin

let S denote the set of eligible nodes;

$Q := \{(i,j) \in T(v) \mid i \notin S,\ j \in S\}$;

$\Delta := \{\min -c_{ij}^{\pi} \mid (i,j) \in Q\}$;

for each node $j \in S$, increase π_j by Δ;

end;

During the procedure SIMPLEX PIVOT(k, l), the arc (k, l) is pivoted in. Because of the non-degeneracy assumption there will be only one arc (p, q) with $x_{qp} = \delta$. Sending δ units of flow around W means that for every arc $(i, j) \in W$

1. if $x_{ij} \in G : x_{ij} \leftarrow x_{ij} + \delta$
2. if $x_{ji} \in G : x_{ji} \leftarrow x_{ji} - \delta$

Note that after modifying the flow it maintains the feasibility since δ is the smallest value we can decrease the flow with on the reversed arcs. After the update $x_{pq} = 0$ so (p, q) is the leaving arc. The subroutine also changes the root node to be p.

During the procedure MODIFY-PREMULTIPLIERS, the number of eligible nodes strictly increases. Observe that for every arc $(i, j) \in Q$ c_{ij}^{π} must be negative, otherwise j would be in S. After adding Δ to each π_j in S, at least one arc's reduced cost becomes zero in Q. If (k, l) is an arc whose reduced cost became zero, then k is a new member of S. Therefore, the number of eligible nodes have increased.

Lemma 2.1.8. *The premultiplier algorithm maintains a vector of premultipliers at every step.*

Proof. We are proving the lemma by induction on the number of steps made by the algorithm. We have a vector of premultipliers for the initial basic feasible solution by

assumption.

In the subroutine MODIFY-PREMULTIPLIERS, $T(v)$ remains unchanged and the update only affects arcs which enter or leave S. One can easily see that there cannot be any arcs leaving S in $T(v)$. So, the affected arcs are in Q. Due to the definition of Δ the reduced cost of the arcs in Q stay non-positive.

During the subroutine SIMPLEX PIVOT(k,l), we choose the root node to be p. (k,l) is an eligible arc so π is also a vector of premultipliers with respect to $T(k)$. Let T' be the updated tree: $T' = T \setminus (p,q) \cup (k,l)$. The orientation of the arcs in $T'(p) \setminus (k,l)$ is the same as in $T(k)$ and $c^{\pi}_{kl} < 0$. Thus, π is a vector of premultipliers with respect to $T'(p)$. □

Remark 2.1.9. Optimality conditions T is an optimal basis if and only if the cost of each basic cycle is non-negative.

Theorem 2.1. *The premultiplier algorithm solves the minimum cost flow problem in a finite number of iterations.*

Proof. In every non-degenerate pivot, the objective function is strictly increasing – which directly follows from our non-degeneracy assumption – hence we do not return to a former basic tree T. And in every call of the MODIFY-PREMULTIPLIERS, the number of eligible nodes is strictly increasing so the algorithm performs a non-degenerate pivot within n iterations. Since there is a finite number of basis, the algorithm stops after a finite step. □

2.2 Dual network simplex algorithm

In the following, I am describing a dual network simplex algorithm for the same problem below.

Definition 2.2.1. x is called a **feasible flow** if $0 \leq x$.

Definition 2.2.2. imbalance at node j: $e_x(j) := b(j) + \sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ji}$

We say that j has an excess or a deficit depending upon whether $e_x(v) > 0$ or $e_x(v) < 0$, respectively.

Let T be a spanning tree. $\pi \in \mathbb{R}^n$ is the associated dual solution. As suggested by Theorem 1.2, T is called **dual feasible** if $\pi(i) - \pi(j) \leq c_{ij} \; \forall (i,j) \in E$ and from the complementary slackness conditions we have that $x > 0 \implies \pi(i) - \pi(j) = c_{ij}$.

Similarly to the previous algorithm, artificial arcs are added to have an initial basic structure: $(0,j) \; \forall j \in V$ where 0 is a new artificial node, $b(0) = 0$ and $c_{0j} = 0$. So $\pi \equiv 0$ is a dual feasible solution and $x_{0j} = -b(j) \;$ if $\; b(j) \leq 0$ otherwise $x_{kl} = 0$ is a feasible flow.

For any spanning tree T, assuming that $\pi(0) = 0$ and $\pi(i) - \pi(j) = c_{ij} \; \forall (i,j) \in T$, π is fully determined. Since starting from the leaves we can always figure out π for the next node. Each non-tree arc has a flow equal to 0.

Note that the imbalances for the initial flow are $e_x(j) \geq 0$ for every original node j and $e_x(0) \leq 0$ for the artificial node (root node). In this algorithm we are going to push flow to the root - without generating any deficit in the original nodes - until it has no deficit.

Definition 2.2.3. T is a spanning tree. There are two types of arcs:

downward arc - an arc directed away from the root

upward arc - an arc directed towards the root

Definition 2.2.4. T is called strongly dual feasible if $x(i,j) > 0$ for every upward arc. (So a positive amount of flow can be pushed down to the leaves.)

Note that our initial spanning tree is strongly dual feasible.

For a leaving arc we are going to choose a downward arc $(i,j) \in T$ which satisfies: $x_{ij} = 0 \wedge e_x(j) > 0$.

After leaving an arc out of the tree it falls apart into two sub-trees. T^R is the one with the root node in it and T^N is the other part of the tree.

To choose an entering arc first we have to calculate $\theta = \min\{c^\pi_{vw} \mid x_{vw} = 0, \;\; v \in T^N, \;\; w \in T^R\}$. An arc $(p,q) : c^\pi_{pq} = \theta$ will be chosen as the entering arc in order to maintain the dual feasibility after the updates.

Now we describe the **algorithm** by Ronald D. Amstrong and Zhiying Jin [5].

Let $T = \{(0,v) \mid v \in V\}$ be the initial strongly dual feasible basis, $\pi \equiv 0$ is the associated dual solution and $x_{0j} = -b(j) \;$ if $\; b(j) \leq 0$ otherwise $x_{kl} = 0$ is the initial feasible flow.

begin $(*)$

  $S_{leav} := \{(i,j) \in E \mid x_{ij} = 0,\ \ e_x(j) > 0, (i,j)$is a downward arc$\}$;

  if S_{leav} is empty **then**

   STOP: x is optimal;

  else

   choose an arc $(g,h) \in S_{leav}$;

   begin $(**)$

    $S_{ent} := \{(i,j) \in E \mid x_{ij} = 0,\ \ i \in T^N,\ \ j \in T^R\}$;

    if S_{ent} is empty **then**

     STOP: the problem is infeasible;

    else

     if (g,h) is the first leaving arc in this pivot **then**

      $\theta_j := \min\{c^{\pi}_{ij} \mid x_{ij} = 0, i \in T^N\}\ \ \forall j \in T^R$

     else

      $\theta_j := \min\left\{\left(c^{\pi}_{ij} \mid x_{ij} = 0, i \in T^N \setminus T^{N'}\right), \left(\theta^{'}_j - \theta^{'}\right)\right\}\ \ \forall j \in T^R$

     $\theta := \min \theta_j$

     choose an entering arc $(p,q) \in S_{ent} : c^{\pi}_{pq} = \theta$;

     Set $\pi(v) \leftarrow \pi(v) + \theta\ \ \forall v \in T^N$;

     begin

      search for a leaving arc on the path from q to the root;

      if there is no leaving arc on that path **then**

       $T \leftarrow T \setminus \{$leaving arcs$\} \cup \{$entering arcs$\}$;

       procedure PUSH-FLOW(T);

       go to $(*)$

      else let (g,h) be the first leaving arc on the path **then** go to $(**)$;

     end;

   end;

end;

procedure PUSH-FLOW(T);

begin

 for $v \in V$ in a reverse BFS order **do**

 begin

  $w :=$ the node after v on the path from v to the root;

if (v, w) is an upward arc **then**
push $e_x(v)$ from v to w;
else
push $\min\{e_x(v), x_{wv}\}$ from v to w;
end;
end;

Remark 2.2.5. Note that we do not change the tree until a positive amount of flow can be pushed to the root. When we cannot send any flow to the root T^N is strictly increasing so a positive amount of flow will be sent to the root within n steps. The minimum reduced cost can be chosen in $\mathcal{O}(n \log n)$ time per block pivot using a Fibonacci-heap . The tree update and the update of the dual variables takes $\mathcal{O}(n)$ time. During a block-pivot we calculate the reduced cost at most once for each arc. So the total running time of a pivot is $\mathcal{O}(m + n \log n)$.

Lemma 2.2.6. *The algorithm maintains a strongly dual feasible tree at each step.*

Proof. The lemma is proved by induction on the number of steps in which procedure PUSH-FLOW is called. The initial tree is strongly dual feasible since there is no upward arc in it. Suppose that at some point (g, h) is the leaving arc and (p, q) is the subsequent entering arc. By the induction hypothesis T was strongly dual feasible before the tree update that means a positive amount of flow could have been pushed down from g to p. The orientation of the arcs on that path will change after the tree update hence we are able to push $\alpha > 0$ amount of flow from g to q during the procedure PUSH-FLOW. Therefore, after the tree update we can send at least α amount of flow down from q to g. Furthermore, the orientation of the arcs on the path from q to the root will not change. Although, some flow may have been pushed from q to the root during the PUSH-FLOW procedure it can only increase x on the upward arcs. □

Theorem 2.2. *The algorithm solves the minimum cost flow problem in a finite number of iterations.*

Proof. Since after each block pivot the total excess of the original nodes is strictly decreasing we cannot return to any former basis. There is a finite number of basis so the algorithm ends after a finite number of block pivots. □

3 Strongly polynomial algorithms

This section describes the scaling version of both algorithms we expounded earlier. The main idea is that arcs will become fixed at some point. In the first algorithm, they stay permanently non-basic which means they are not in any feasible basis during the current or any subsequent scaling phase. While in the second algorithm they stay permanently basic.

3.1 Strongly polynomial primal network simplex algorithm

As mentioned above, this section discusses the scaling version of the primal network simplex algorithm described in Subsection 2.1.

3.1.1 Structure and properties

Definition 3.1.1. Let N^* denote the subset of nodes whose multipliers have yet to change during the ε-scaling phase.

Definition 3.1.2. Suppose that π is a vector of premultipliers with respect to x. We say that π is a vector of **ε-premultiplers** if $c^{\pi}_{ij} \geq -\varepsilon \ \ \forall (i,j) \in G(x)$.

Definition 3.1.3. We call a node i **awake** if $i \in N^*$ or $\pi_i = k * \varepsilon/4$ for some $k \in \mathbb{Z}$ otherwise it is called **asleep**.

Definition 3.1.4. An arc (i,j) for which $x_{ij} = 0$ is called **admissible** if i is an eligible and awake node and $c^{\pi}_{ij} < -\varepsilon/4$.

The following algorithm is the scaling version of the premultiplier algorihtm described in Section 2.1. It is called the **scaling-premultiplier algorithm** [3];

begin

choose an initial basic feasible solution x and a vector of premultipliers π with respect to $T(v)$;

while x is not optimal **do**

begin

IMPROVE-APPROXIMATION(x, ε, π);

$\varepsilon := \max\{|c^{\pi}_{ij}| \mid c_{ij} < 0$ and $(i,j) \in G(x)\}$;
end;
end;

procedure IMPROVE-APPROXIMATION(x, ε, π);
begin
$N^* := N$;
while $N^* \neq \emptyset$ **do**
begin
if there is an admissible arc **do**
begin
select an admissible arc (k, l);
SIMPLEX-PIVOT(k, l);
end;
else MODIFY-ε-PREMULTIPLIERS;
end;
end;

procedure MODIFY-ε-PREMULTIPLIERS;
begin
let S be the set of eligible nodes;
$N^* := N \setminus S$;
if $N^* = \emptyset$ **then** STOP IMPROVE-APPROXIMATION(x, ε, π);
$\Delta_1 := \min\{-c^{\pi}_{ij} \mid (i,j) \in T(v), i \notin S, j \in S\}$;
$\Delta_2 := \min\{(\varepsilon/4 - \pi_i) \mod \varepsilon/4 \mid i \in S\}$;
$\Delta := \min\{\Delta_1, \Delta_2\}$;
Set $\pi_i \leftarrow \pi_i + \Delta\ \forall i \in S$;
end;

In order to guarantee an amortized average time of $\mathcal{O}(n)$ per scaling phase, we use a structure called *current arc data structure*. In this structure, each node has a current arc and before the algorithm starts we establish an order of the arcs which remains unchanged during the whole algorithm. Initially, First-Arc(i) will be the current arc for node i. At first, the algorithm maintains a depth first search starting

at the root node to find eligible and awake nodes. If there is an eligible and awake node, then the algorithm will check whether the current arc is admissible. If not, it labels the next arc in the arc list as current arc. When we reach the last arc for a particular node i, the current arc is set to $\emptyset$. It stays $\emptyset$ until i becomes awake (def 3.1.3). When i wakes up, we set the current arc to be First-Arc(i). Thus, an arc (i, j) is scanned as many times as i becomes awake which is $\mathcal{O}(n)$ per scaling phase (we are going to prove that later). So the total running time scanning the arc list for admissible arcs is $\mathcal{O}(nm)$ per scaling phase. Furthermore, searching for eligible and awake nodes for which the current arc $\neq \emptyset$ takes $\mathcal{O}(n)$ time following by a pivot or update of the dual variables.

The procedure SIMPLEX-PIVOT(k, l) is the one we described in the premultiplier algorithm earlier.

During the procedure MODIFY-ε-PREMULTIPLIERS, either a node becomes eligible or a node wakes up depending upon whether $\Delta = \Delta_1$ or $\Delta = \Delta_2$, respectively.

Lemma 3.1.5. *Suppose that π^{curr} is a vector of ε-premultipliers obtained during an ε-scaling phase. Let π' be the vector of premultipliers immediately prior to the most recent execution of* MODIFY-ε-PREMULTIPLIERS *at which time i was both eligible and awake. Then* $0 < \pi^{curr} - \pi' \leq \varepsilon/4$.

Proof. Let π^0 be the vector of premultipliers at the beginning of the scaling phase. First of all, note that i was both eligible and awake at the first time when π_i^0 was increased so $\pi^{'}$ is well-defined.

If $\pi_i^{curr} - \pi_i^0 \leq \varepsilon/4$ then the lemma is evident since $\pi_i^0 \leq \pi_i^{'} \leq \pi_i^{curr}$.

So, let us assume that $\pi_i^{curr} - \pi_i^0 > \varepsilon/4$. Let α be the largest multiple of $\varepsilon/4$ which is strictly less then π_i^{curr}. Then $\pi_i^{curr} - \alpha \leq \varepsilon/4$ and $\pi_i^0 < \alpha < \pi_i^{curr}$. We show that $\pi_i^{'} = \alpha$. Consider the first iteration at which π_i was increased to a value at least α, thus $\pi_i \geq \alpha$. By the definition of Δ_2, $\pi_i = \alpha$ at this time, therefore i is both eligible and awake. The next iteration at which π_i is an integer multiple of $\varepsilon/4$ that is, i wakes up again is when $\pi_i = \alpha + \varepsilon/4$ but $\pi_i^{curr} \leq \alpha + \varepsilon/4$. □

Lemma 3.1.6. *Suppose that x is a feasible flow and π is a vector of ε-premultipliers obtained during an ε-scaling phase. If $i \notin N^*$ then for $(i, j) \in G(x)$: $c_{ij}^{\pi} \geq -\varepsilon/2$.*

Proof. Let π' be the vector defined above in Lemma 3.1.5.

1. If (i, j) was a tree-arc during the iteration at which π' was the vector of premultipliers, then i and j were both eligible since a tree-arc cannot leave the set of eligible nodes. Thus, $c^{\pi}_{ij} = 0$ from that iteration.

2. If (i, j) was an inadmissible non-tree arc then $c^{\pi'}_{ij} > -\varepsilon/4$. We know that

$$c^{\pi}_{ij} = c^{\pi'}_{ij} + (\pi_j - \pi'_j) - (\pi_i - \pi'_i)$$

   By Lemma 3.1.5: $\pi_i - \pi'_i \leq \varepsilon/4$. And $\pi_j - {\pi_j}' \geq 0$. So in this case we have that

$$c^{\pi}_{ij} \geq -\varepsilon/4 + 0 - \varepsilon/4 = -\varepsilon/2$$

3. If $(i, j) \notin G(X')$ (where x' is the flow related to π') then (j, i) had to enter the basis or else (i, j) would not be in $G(x)$. Let π" be the vector of premultipliers when (j, i) entered the basis. We have $c^{\pi"}_{ji} < -\varepsilon/4$ that implies $c^{\pi"}_{ij} = -c^{\pi"}_{ji} > \varepsilon/4$ and

$$c^{\pi}_{ij} = c^{\pi"}_{ij} + (\pi_j - \pi_j") - (\pi_i - \pi_i")$$

   We also know that $\pi' \leq \pi" \leq \pi$ therefore

$$c^{\pi}_{ij} \geq \varepsilon/4 + 0 - \varepsilon/4 = 0$$

□

Corrolary 3.1.7. If $N^* = \emptyset$ then π is a vector of $\varepsilon/2$-premultipliers with respect to x.

Lemma 3.1.8. *Let x and x' denote any two different non-degenerate basic feasible flows. For any pair of nodes i and j there is a directed path from i to j in $G(x)$ for which the reversed path from j to i is in $G(X')$.*

Proof. We prove by induction on the distance between i and j in tree T corresponding to the basic feasible flow x. First, assume that $d_T(i, j) = 1$ so either (i, j) or (j, i) is in T. Suppose that $(i, j) \in T$ if $x'_{ij} \neq 0$ then $(j, i) \in G(X')$ and the lemma is proved. Now, consider the case when $x'_{ij} = 0$. $x' - x$ can be decomposed into flows around directed cycles by flow decomposition. Equivalently, we can get x' from x by pushing positive amount of flow around directed cycles in $G(x)$. Since $x_{ij} > x'_{ij}$ there is a directed cycle containing (j, i) in $G(x)$. Let us take the path P from i to j in that

cycle. P is in $G(x)$ as we have already seen and there is a positive amount of flow going through P by the time when x' is the flow so the reversed path is in $G(X')$.

Now, suppose that $d_T(i,j) = D$. Let k be the node preceding j in the path from i to j in T. So $d_T(i,k) = D-1$ and $d_T(k,j) = 1$ by the induction hypothesis we have a path P_1 from i to k and another path P_2 from k to j in $G(x)$ which satisfy the conditions. Let us concatenate P_1 and P_2 into a directed walk and then leave out the directed cycles, if there any. This path meets the criteria. □

Lemma 3.1.9. *During the ε-scaling phase π_j is increased by at most $3/2n\varepsilon$ units.*

Proof. Let x denote the basic flow and π the vector of premultipliers at the beginning of the ε-scaling phase. Suppose that π' is the premultiplier sometime during the ε-scaling phase and x' is the related flow. As the algorithm will stop immediately when N^* becomes empty, there is a node $k \in N^*$ and for that node $\pi_k = \pi'_k$. By Lemma 3.1.8 there is a path $P \in G(x)$ from k to j which reversal P^{-1} is in $G(X')$. We assume that node k is the only node in N^* along this the path. (If it were not, we could take the last node $k^* \in N^* \cap P$ and replace the path with the subpath from k^* to j.) We know that

$$c^{\pi}(P) = c(P) - \pi_k + \pi_j \geq -(n-1)\varepsilon$$

Moreover, since there is no arc emanating from N^* in P^{-1} from Lemma 3.1.6 we have

$$c^{\pi'}(P^{-1}) = c(P^{-1}) - \pi'_j + \pi'_k \geq -(n-1)\varepsilon/2$$

Which implies

$$\underbrace{c(P) + c(P^{-1})}_{=0} + \underbrace{\pi'_k - \pi_k}_{=0} + \pi_j - \pi'_j \geq -(n-1)3/2\varepsilon$$

Rearranging the inequality we get $\pi_j + (n-1)3/2\varepsilon \geq \pi'_j$, completing the proof. □

3.1.2 Complexity analysis

Lemma 3.1.10. *The algorithm performs at most $6nm$ pivots per scaling phase.*

Proof. Let π and π' denote the premultipliers at some iteration when either (i,j) or (j,i) is pivoted in for the first and for the second time, respectively. Let us assume that $\pi'_i - \pi_i + \pi'_j - \pi_j \geq \varepsilon/2$. This assumption and Lemma 3.1.9 imply that there are at most $6n$ iterations at which (i,j) or (j,i) is pivoted in.

Now, we are going to prove the assumption. Suppose that (i,j) is the entering arc by the time π is the vector of permultipliers.

Case I. The arc pivoted in next time was (j,i). We have that $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j \leq -\varepsilon/4$ and $c^{\pi'}_{ji} \leq -\varepsilon/4 \Leftrightarrow c^{\pi'}_{ij} = c_{ij} - \pi'_i + \pi'_j \geq \varepsilon/4$. It follows that

$$\varepsilon/2 \leq (\pi'_j - \pi_j) + (\pi_i - \pi'_i) = (\pi'_j - \pi_j) - (\pi'_i - \pi_i) \leq \pi'_j - \pi_j$$

Case II. The arc pivoted in next time was (i,j). This requires that between π and π', there must have been an iteration π^{out} at which (j,i) was pivoted out. Since, when (i,j) was pivoted in for the first time, a positive amount of flow has been sent from i to j and (i,j) can only be considered as an entering arc once that flow is sent back from j to i that is, when (j,i) is pivoted out. In this case, we know that $c^{\pi^{out}}_{ji} \leq 0 \Leftrightarrow c^{\pi^{out}}_{ij} \geq 0$ since $(j,i) \in T(v)$ and π^{out} is a vector of premultipliers. Thus, we get that

1. $c_{ij} - \pi_i + \pi_j \leq -\varepsilon/4$
2. $0 \leq c_{ij} - \pi^{out}_i + \pi^{out}_j$
3. $c_{ij} - \pi'_i + \pi'_j \leq -\varepsilon/4$
4. $\pi \leq \pi^{out} \leq \pi'$

$$(1) + (2) \implies \varepsilon/4 \leq \underbrace{c_{ij} - c_{ij}}_{=0} + \underbrace{\pi_i - \pi^{out}_i}_{\leq 0 \text{ using}(4)} + \pi^{out}_j - \pi_j \leq \pi^{out}_j - \pi_j \leq \pi'_j - \pi_j$$

$$(2) + (3) \implies \varepsilon/4 \leq \underbrace{c_{ij} - c_{ij}}_{=0} + \underbrace{\pi^{out}_j - \pi'_j}_{\leq 0 \text{ using}(4)} + \pi'_i - \pi^{out}_i \leq \pi'_i - \pi^{out}_i \leq \pi'_i - \pi_i$$

Summing the two inequalities above we get $\varepsilon/2 \leq \pi'_i - \pi_i + \pi'_j - \pi_j$, completing the proof. □

Lemma 3.1.11. *The scaling permultiplier algorithm terminates in $\mathcal{O}(m \log n)$ scaling phases with an optimal flow.*

Proof. First, we bound the number of scaling phases at which a pivot takes place. As I mentioned at the beginning of this section we are going to use an arc relaxation technique.

Definition 3.1.12 (permanently non-basic arcs)**.** An arc (i, j) is permanently non-basic if it is not in any feasible basis in the ε-scaling phase or in any other subsequent scaling phase.

We show that after each scaling phase in which a pivot takes place at least one arc becomes permanently non-basic within $\mathcal{O}(4 + 2\log n)$ iterations. Since an arc can become permanently non-basic at most once, the total number of scaling phases with a pivot is $\mathcal{O}(4m + 2m\log n)$.

By Lemma 3.1.9, π_j is increased by at most $3/2n\varepsilon$ units during an ε-scaling phase and by Lemma 3.1.6 ε decreases by a factor of at least 2 in two consecutive scaling phases. So, the total increase in π_j during the ε-scaling phase and all subsequent scaling phase is:

$$3n\varepsilon(1/2 + 1/4 \ldots) \leq 3n\varepsilon$$

Therefore, if there is an arc (i, j) during the ε-scaling phase for which $c_{ij}^{\pi} > 3n\varepsilon$ then this arc will not be in any subsequent feasible basis since its reduced cost stays non-negative.

Suppose that (i, j) is pivoted in during the ε-scaling phase and W is the basic cycle induced by (i, j). Then $c^{\pi}(W) = c(W) < -\varepsilon/4$. Now, let ε' be the scale factor $(4 + 2\log n)$ scaling phase later and π' is the vector of premultipliers at the beginning of the ε'-scaling phase. We know that

$$\varepsilon' < \varepsilon/2^{4+2\log n} = \varepsilon/16n^2 \Leftrightarrow -\varepsilon/4 < -\varepsilon' 4n^2$$

This implies $c^{\pi'}(W) = c(W) < -\varepsilon/4 < -\varepsilon' 4n^2$. Since there are at most n arcs in W, there is at least one arc (k, l) for which $c_{kl}^{\pi'} < -4n\varepsilon'$. Note that π' is a vector of ε'-premultipliers hence satisfies $c_{pq}^{\pi'} \geq -\varepsilon'$ $\forall (p, q) \in G(X')$. Thus, (k, l) is not represented in $G(X')$ but then $(l, k) \in G(X')$ and $c_{lk}^{\pi'} \geq 4n\varepsilon' \geq 3n\varepsilon'$.

Now we bound the number of scaling phases at which no pivot takes place. We show that there cannot be two consecutive scaling phases at which no pivot takes place. Suppose that no pivot takes place in the ε-scaling phase. Then the procedure MODIFY-ε-PREMULTIPLIERS called consecutively several times until every

node becomes eligible. Therefore, at the end of this scaling phase every tree arc has 0 reduced cost. At the beginning of the subsequent scaling phase, $\varepsilon := \max\{|c_{ij}^\pi| \mid c_{ij} < 0 \text{ and } (i,j) \in G(x)\}$, so there is an arc (k,l) for which $c_{kl}^\pi = -\varepsilon$. Note that every node is both awake and eligible at the beginning of this particular scaling phase that is why (k,l) is an admissible arc.

Thus, the total number of scaling phases is at most twice the number of scaling phases at which a pivot takes place, completing the proof. □

Remark 3.1.13. If the data is integer, then there is an easier way to bound the number of scaling phases which may as well be a better estimation. In this special case, there is at most $\mathcal{O}(\log nC)$ scaling phases where C denotes the maximum absolute arc cost.

The initial value of ε is $\mathcal{O}(nC)$, because we added artificial arcs with $(nC+1)$ costs. ε is decreased by a factor of at least 2 in two consecutive scaling phases by Lemma 3.1.6. So within $\mathcal{O}(\log nC)$ scaling phases $\varepsilon' < 1/n$. Let π' denote the ε'-premultipliers at the beginning of the ε'-scaling phase.
So $c_{ij}^{\pi'} \geq -\varepsilon' > -1/n \ \forall (i,j) \in G(X')$. Take any directed cycle $W \in G(X')$. We have $c^{\pi'}(W) = c(W) > -1$ and $c(W)$ is integer therefore $c(W) \geq 0$, completing the proof.

Lemma 3.1.14. *The procedure* MODIFY-ε-PERMULTIPLIERS *is called at most* $\mathcal{O}(nm)$ *times per scaling phase.*

Proof. In the procedure MODIFY-ε-PREMULTIPLIERS either a node awakes or a node becomes eligible. The former case can happen at most $6n$ times per node according to Lemma 3.1.9 so $\mathcal{O}(n^2)$ times per scaling phase altogether.

In the latter case an arc's reduced cost becomes 0 whose reduced cost was not 0 before the procedure. We call that cancelling an arc. Let (i,j) denote an arc which is pivoted in so $c_{ij}^\pi < -\varepsilon/4$ and stays negative until either (i,j) is pivoted out or cancelled during the procedure MODIFY-ε-PREMULTIPLIERS. If (i,j) is cancelled then its reduced cost stays 0 until (i,j) is pivoted out. So (i,j) is cancelled at most as many times as it is pivoted in which is $6n$ times by Lemma 3.1.10 per scaling phase. Therefore, the total number of cancelling is $\mathcal{O}(nm)$ per scaling phase. □

Theorem 3.1. *The scaling premultiplier algorithm solves a minimum cost network simplex problem in* $\mathcal{O}(m \log n)$ *scaling phases, each of which has* $\mathcal{O}(nm)$ *pivots. Furthermore, the running time per scaling phase is* $\mathcal{O}(n^2 m)$.

Proof. The theorem directly follows from the lemmata above.

1. The number of scaling phases is $\mathcal{O}(m \log n)$ by Lemma 3.1.11.
2. The procedure MODIFY-ε-PREMULTIPLIERS is called at most $\mathcal{O}(nm)$ times per scaling phase by Lemma 3.1.14
3. Scanning nodes in N takes $\mathcal{O}(n)$ time per MODIFY-ε-PREMULTIPLIERS
4. There are $\mathcal{O}(nm)$ pivots per scaling phase by Lemma 3.1.10.
5. Updating the spanning tree and sending flow around takes $\mathcal{O}(n)$ time per pivot
6. Finding an admissible arc if there is one takes $\mathcal{O}(n)$ time per pivot + the time to update current arcs
7. The update of the current arcs takes $\mathcal{O}(nm)$ time per scaling phase (as we discussed it right after the algorithm)

□

3.2 Strongly polynomial dual network simplex algorithm

In this subsection, we are going to discuss the scaling version of the dual network simplex algorithm described in Subsection 2.2.

3.2.1 Structure and properties

Overview of the algorithm: we are trying to push ε units of flow from a node v with $e_x(v) \geq \varepsilon$ to the root. So, we are going to choose a downward arc (i, j) with $x_{ij} = 0$ and $e_x(j) \geq \varepsilon$ as a leaving arc.

Suppose that (p, q) is the current entering arc. If the imbalance at some node k along the path from q to the root is less then ε after we pushed flow to k, then we will not continue the flow augmentation from there.

In order to facilitate calculating the number of times this can happen per arc, we also define a weaker attribute:

Definition 3.2.1. Suppose that we are trying to push ε amount of flow from j to i. We say that (i, j) *restricts* flow if $x_{ij} < \varepsilon$, so we can only reduce x_{ij} with less then ε. And we say that (i, j) is ε*-obstructs* flow augmentation if the imbalance at node i is less then ε after we pushed flow from j to i.

Note that: (i, j) ε-obstructs flow augmentation $\implies$ (i, j) restricts flow.

Lemma 3.2.2. *During an ε-scaling phase each arc can ε-obstruct flow augmentation at most once.*

Proof. Suppose that (i, j) is a basic arc at the beginning of the ε-scaling phase. Since there are no upper bounds on the arcs, (i, j) can only restrict flow augmentation when we push flow from j to i and before that $x_{ij} < \varepsilon$. After this step $x_{ij} = 0$. Therefore, x_{ij} will be a multiple of ε through out the current scaling phase.

Now let (i, j) denote a non-basic arc which means that $x_{ij} = 0$ at the beginning of the ε-scaling phase. So, (i, j) cannot restrict flow during this scaling phase.

Since ε-obstruction is a stronger attribute than flow restriction the lemma follows. □

Definition 3.2.3. T is called ε-strongly dual feasible (ε-SDF) if $x_{ij} + e_x(i) \geq \varepsilon$ for every upward arc.

In case of an ε-SDF tree, when we attempt to push ε amount of flow down from j to i (where $e_x(j) \geq \varepsilon$) either

1. $x_{ij} \geq \varepsilon$ so we set $x_{ij} \leftarrow x_{ij} - \varepsilon$ and $e_x(i) \leftarrow e_x(i) + \varepsilon$ or
2. $curr = x_{ij} < \varepsilon$ so we set $x_{ij} \leftarrow 0$ and $e_x(i) \leftarrow e_x(i) + curr$

Either way $e_x(i) \geq \varepsilon$ after the update, so we are able to push down ε-amount of flow from the root to the leaves.

Similarly to the primal simplex algorithm, the purpose is to relax some arcs. In this algorithm, relaxing an arc (i, j) means that $x_{ij} > 0$ and the value of x_{ij} is "big enough" so it stays positive during the current scaling phase and any subsequent scaling phases. Note that a relaxed arc cannot be non-basic and it cannot ε-obstruct flow augmentation either. After at most n relaxation the algorithm terminates.

At the beginning of each ε-scaling phase, the algorithm checks if there exists any arc that can be relaxed. In this algorithm, we relax an arc (i, j) if $x_{ij} \geq 6m\varepsilon$. We prove the correctness of this claim later in Lemma 3.2.9.

procedure RELAX ARC (ε,*relax*)
begin
  for each arc (i, j) **do**
  begin
    if $x_{ij} \geq 6m\varepsilon$ and (i, j) is not relaxed **then**
      relax (i, j)
      Set *relax* $\leftarrow 1$;
  end;
end;

A pivot contains the following series of steps: We choose a downward arc (i, j) with $e_x(j) \geq \varepsilon$ as the leaving arc and an arc (p, q) with $p \in T^N, q \in T^R$ as the entering arc. Suppose that T is an ε-SDF tree hence ε amount of flow can be pushed from j to q. Now, we verify weather ε amount of flow can also be pushed from q to the root. Only a downward arc (v, w) with $x_{vw} + e_x(v) < \varepsilon$ can ε-obstruct the flow. Let (v, w) be the first downward arc on the path from q to the root which would ε-obstruct the flow. If $x_{vw} = 0$ then (v, w) will be the next leaving arc. If $x_{vw} \neq 0$ or there is no arc which would ε-obstruct the flow on that path, we will update the dual variables and the tree.

This way after the tree update, we are able to send ε amount of flow from j (the first leaving arc) to the root or from j to w. For that we apply the procedure below.

procedure PUSH-ε-FLOW(T, ε, v)
begin
  while $e_x(v) \geq \varepsilon$ **do**
  begin
    let $w := pred(v)$;
    if (w, v) is a downward arc **then**
      begin
        Set $\alpha := \min\{x_{wv}, \varepsilon\}$

$e_x(v) \leftarrow e_x(v) - \alpha$, $x_{wv} \leftarrow x_{wv} - \alpha$, $e_x(w) \leftarrow e_x(w) + \alpha$;

end;

else //(v, w) is an upward arc

begin

$e_x(v) \leftarrow e_x(v) - \varepsilon$, $x_{vw} \leftarrow x_{vw} + \varepsilon$, $e_x(w) \leftarrow e_x(w) + \varepsilon$;

end;

Set $v \leftarrow w$;

end;

end;

In the following, we show the scaling version of the dual simplex algorithm described in Section 2.2.

algorithm [5]

Set $T = \{(0, v) \mid v \in V\}$, $\pi \equiv 0$;

$x_{0j} = -b(j)$ if $b(j) \leq 0$ otherwise $x_{kl} = 0$;

Set $\varepsilon' \leftarrow \max\{e_x(v) \mid v \in V\}$

I.

begin

Call procedure PUSH-FLOW(T);

Set $\varepsilon \leftarrow \max\{e_x(v) \mid v \in V\}$

$\varepsilon \leftarrow \min\{\varepsilon, \varepsilon'\}$;

if $\varepsilon == 0$ **then**

STOP; the current solution is optimal

Set *relax* $\leftarrow 0$;

while there is a node j with $e_x(j) \geq \varepsilon$ **do**

begin

$i := pred(j)$;

if (i, j) is a downward arc and $x_{ij} = 0$ **then**

perform a pivot as described before;

Call procedure PUSH-ε-FLOW(T, ε, j);

while *relax* $= 0$ **do**

II.

begin

```
            Set ε ← ε/2;
            Call procedure RELAX-ARC(ε, relax)
            while there is a node j with e_x(j) ≥ ε do
            begin
               i := pred(j);
               if (i, j) is a downward arc and x_ij = 0 then
                  perform a pivot as described before;
               Call procedure PUSH-ε-FLOW(T, ε, j);
            end;
         end;
      Set ε' ← ε/2;
      go to (I.);
   end;
```

An iteration starts at the line marked with the **I.** or **II.** symbol. (We also refer to an iteration as a scaling phase.)

The following lemma is self-evident.

Lemma 3.2.4. *If ε denotes the scaling factor at the beginning of the k-th iteration and ε' is the scaling factor at the beginning of the $(k+1)$-th iteration. Then $\varepsilon' < \varepsilon/2$.*

Lemma 3.2.5. *The algorithm maintains an ε-SDF tree after each call of procedure* PUSH-ε-FLOW.

Proof. Since there are no upward arcs in the initial basic tree T, it is ε-SDF. Thus, the basic tree is ε-SDF right before the first call of procedure PUSH-ε-FLOW(T, ε, j).

We assume that before each call of the procedure the current basic tree is ε-SDF. After each call of procedure PUSH-ε-FLOW(T, ε, j) every upward arc (k, l) on the predecessor path from j to the root satisfies $x_{kl} + e_x(k) \geq \varepsilon$. If it is on the subpath from j to the last entering arc then, since $e_x(j) \geq \varepsilon$ before the execution and there are no flow obstruction along that subpath the arc meets the condition. If it is on the subpath from the last entering arc to the root then it was an upward arc before and sending ε amount of flow through it cannot change the value of $x_{kl} + e_x(k)$.

Each upward arc (v, w) beside that path fulfils the ε-SDF condition. If it has not change its direction then it is still ε-SDF since it was before. If it changes its

direction due to the pivot then it was along the path from j to one of the leaving arcs (k,l) (logically (k,l) was not the last leaving arc before the tree update since we assumed that it is not on the current predecessor path from j to the root). Therefore, during the pivot we verified that sending ε amount of flow from w to v would not ε-obstruct the flow.

□

Lemma 3.2.6. *A leaving arc will always exist during the iterations marked with I.*

Proof. After the procedure PUSH-FLOW(T) either $e_x(v) = 0 \quad \forall v \in V$ or $\varepsilon \leq \max\{e_x(v) \mid v \in V\}$ so there will be a downward arc (i,j) with $x_{ij} = 0$ and $e_x(j) \geq \varepsilon$. □

3.2.2 Complexity analysis

Our last job is to estimate the running time of the algorithm above.

Lemma 3.2.7. *At the beginning of each iteration the total excess is less than $2n\varepsilon$ where ε is the scaling factor at the beginning of the iteration.*

Proof. At the beginning of the first iteration $\varepsilon = \max\{b(v) \mid v \in V\}$. So the total excess is

$$\sum_{v:b(v)>0} b(v) \leq n\varepsilon$$

Before every other iteration marked with **II.** $\varepsilon = \varepsilon'/2$ where ε' is the previous scaling factor. We know that at the end of the previous iteration $e_x(j) < \varepsilon' \, \forall j \in V$. So the total excess is less than $n\varepsilon' = n2\varepsilon$.

Before every iteration marked with **I.** either $\varepsilon = \max\{b(v) \mid v \in V\}$ or $\varepsilon = \varepsilon'/2$. Both cases were covered above. □

Lemma 3.2.8. *The total amount of flow pushed over any arc during an ε-scaling phase is less then $3m\varepsilon$.*

Proof. After each call of procedure PUSH-ε-FLOW either ε amount of flow is sent to the root or there is an arc that ε-obstructs the flow. We call the formal case *normal execution* and the latter case *abnormal execution.*

By Lemma 3.2.7 there are at most $2n$ normal executions and by Lemma 3.2.2 there are at most m abnormal executions during an ε-scaling phase since each flow can ε-obstruct the flow at most once per scaling phase. If the procedure PUSH-FLOW is called then $\varepsilon = \max\{e_x(v) \,|\, v \in V\}$. It can be called at most once per scaling phase and it may result in ε amount of flow going through an arc. So the total amount of flow pushed through an arc is $(2n + m + 1)\varepsilon < 3m\varepsilon$. □

Now, we are ready to prove that our criteria to relax an arc is correct.

Lemma 3.2.9. *If $x_{ij} \geq 6m\varepsilon$ at the beginning of some scaling phase then $x_{ij} > 0$ holds in any subsequent scaling phases.*

Proof. From Lemma 3.2.8 and 3.2.4 the total amount of flow pushed over (i, j) is $3m(1 + 1/2 + \ldots)\varepsilon < 6m\varepsilon$. □

Lemma 3.2.10. *The iteration marked with **II.** is called at most* $\log n$ *times consecutively.*

Proof. A leaving arc always exists in **I** by Lemma 3.2.6. Let (i, j) denote that leaving arc. The total excess of $T^N_{\varepsilon_0}$ (recall that T^N is the sub-tree which does not contain the root) is at least ε_0 since $e_x(j) \geq \varepsilon_0$. Let $\varepsilon_1, \varepsilon_2 \ldots \varepsilon_L$ denote the scaling factors in the next L scaling phases. From Lemma 3.2.7 at the time when ε_L is the scaling factor the total excess of all nodes is less then $2n\varepsilon_L$ hence at least $\varepsilon_0 - 2n\varepsilon_L$ amount of flow has been pushed out of $T^N_{\varepsilon_0}$. That means there is at least one arc (k, l) for which $x_{kl} = 0$ by the time when (i, j) was the leaving arc and for which $x_{kl} \geq (\varepsilon_0 - 2n\varepsilon_L)/m$ L iterations later. If $L = \mathcal{O}(\log n)$ then $\varepsilon_0 = 2^{\mathcal{O}(\log n)}\varepsilon_L$ and:

$$(\varepsilon_0 - 2n\varepsilon_L)/m \geq 6m\varepsilon_L$$

$$2^{\mathcal{O}(\log n)}\varepsilon_L - 2n\varepsilon_L \geq 6m^2\varepsilon_L$$

$$n^c \geq 6m^2 + 2n$$

Which is true for some constant c. Therefore x_{kl} is becoming relaxed at the and of the scaling phase marked with L. So the *relax* variable is set to 1. □

In the following two lemmas, I give a bound on the number of pivots during the algorithm.

Lemma 3.2.11. *There are at most $\mathcal{O}(nm \log n)$ pivots where an ε-obstruction takes place.*

Proof. After every iteration marked with **I**, part **II** is called at most $\log n$ times until an arc is relaxed. So the total number of scaling phases is at most $n(\log n + 1)$.

In a scaling phase an arc can ε-obstruct the flow at most once by Lemma 3.2.2. Thus, the total number of pivots with ε-obstruction is $\mathcal{O}(mn \log n)$. □

Lemma 3.2.12. *There are at most $\mathcal{O}(n^2 \log n)$ pivots without an obstruction.*

Proof. If there are no arc that would ε-obstruct the flow then ε amount of flow is sent to the root. The total excess is less then $2n\varepsilon$ per scaling phase, so there are at most $2n$ pivots without an obstruction per scaling phase. And the number of scaling phases during the algorithm is at most $n(\log n + 1)$. Thus, the total number of pivots where ε amount of flow is sent to the root is $\mathcal{O}(n^2 \log n)$. □

Theorem 3.2. *The algorithm solves a minimum cost network simplex problem in $\mathcal{O}(mn \log n)$ pivots. The running time per pivot is $\mathcal{O}(m + \log n)$.*

Proof. From Lemma 3.2.11 and 3.2.12 the total number of pivots is $\mathcal{O}(mn \log n)$. And a pivot takes $\mathcal{O}(m + n \log n)$ time by remark 2.2.5.

The procedure RELAX ARC takes $\mathcal{O}(m)$ time while the procedures PUSH-FLOW and PUSH-ε-FLOW run in $\mathcal{O}(n)$ time. So these procedures do not increase the running time mentioned above. □

4 Summary

The goal of this paper was to demonstrate the crucial role of the scaling technique in the area of the network optimization by presenting two strongly polynomial versions of network simplex algorithm. Applying this technique results in strongly polynomial running time, which constitutes a profound efficiency increase.

References

[1] E. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5:247–255, 1985.

[2] J.B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41:338–350, 1993.

[3] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78:109–129, 1997.

[4] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36:873–886, 1989.

[5] Zhiying Jin Ronald D. Amstrong. A new strongly polynomial dual network simplex algorithm. *Mathematical Programming*, 78:131–148, 1997.

[6] J.B. Orlin. On the simplex algorithm for networks and generalized networks. *Mathematical Programming Study*, 24:166–178, 1985.