A Faster Primal Network Simplex Algorithm

C. Aggarwal, H. Kaplan, J. Orlin and R. Tarjan

OR 315-96 March 1996

A faster primal network simplex algorithm

Charu C. Aggarwal, MIT
Haim Kaplan, Princeton University
James B. Orlin, MIT
Robert E. Tarjan, Princeton University

March 4, 1996

Abstract

We present a faster implementation of the polynomial time primal simplex algorithm due to Orlin [23]. His algorithm requires $O(nm \cdot \min\{log(nC), m \cdot \log n\})$ pivots and $O(n^2 m \cdot \min\{\log nC, m \cdot \log n\})$ time. The bottleneck operations in his algorithm are performing the relabeling operations on nodes, selecting entering arcs for pivots, and performing the pivots. We show how to speed up these operations so as to yield an algorithm whose running time is $O(nm \cdot \log n)$ per scaling phase. We show how to extend the dynamic-tree data-structure in order to implement these algorithms. The extension may possibly have other applications as well.

Keywords: Network Flows, Simplex algorithm, polynomial time, premultipliers.

1 Introduction

The minimum cost flow problem is a well known one in the area of network optimization. Many important network problems, such as the shortest path problem, the maximum flow problem, and the assignment problem are special cases of it. The problem has applications in a large number of fields, such as manufacturing, transportation, logistics, telecommunications, and scheduling. A comprehensive survey of such applications may be found in the book by Ahuja, Magnanti, and Orlin [2].

Let $G = (N, A)$ be a network with n nodes and m arcs. Each arc $(i, j) \in A$ has a cost of c_{ij} and a capacity of u_{ij}. Associated with each node i is a supply value $b(i)$. In the event that the costs are integral, we let C denote the maximum absolute arc cost in the network; otherwise we assume $C = \infty$.

The minimum cost flow problem is a special case of the linear programming problem, and is hence solvable in polynomial time. Because of its special combinatorial structure, a number of fast algorithms have been found for the minimum cost flow problem. The current fastest algorithms for the problem are due to Ahuja, Goldberg, Orlin, and Tarjan [1], Goldberg and Tarjan [13], and Orlin [22]. The network simplex algorithm is the choice in practice because of its excellent empirical performance, even though many of the natural pivoting rules take an exponential number of pivots in the worst case, as proved by Zadeh [31]. The first polynomial time primal simplex algorithm has been found by Orlin [23]. The number of pivots required by this method is $O(nm \cdot \log nC)$ or $O(nm^2 \cdot \log n)$, whichever is smaller. However, the bottleneck operations in this method is the time to perform the relabelings ($O(n^2m)$ per scaling phase) and the time to implement the pivots ($O(n)$ per pivot). In this paper, we show how to speed up these bottlenecks, so that the running time of the new algorithm is $O(nm \cdot \log n \cdot \log nC)$ which is within a $O(\log n)$ factor of the best cost scaling algorithm due to Goldberg and Tarjan. ([13])

The simplex method has also been implemented on a number of important special cases of the network flow problem. There exist polynomial time network simplex algorithms for the assignment problem (see, for example, Ahuja and Orlin [3], Akgul [5], Hung [18], Orlin [21], and Roohy-Laleh [25]), the shortest-path problem (see, for example, Ahuja and Orlin [3], Akgul [4], Dial, Glover, Karney, and Klingman [10], Goldfarb, Hao, and Kai [17], and Orlin [21]), and the maximum flow problem (see, for example, Goldfarb and Hao [14], [15], and Goldberg, Grigoriadis, and Tarjan [12].) Polynomial time dual network simplex algorithms for the minimum cost flow problem are due to Orlin [20], and Orlin, Plotkin, and Tardos [24] [6]. Some non-monotonic primal network simplex algorithms for the minimum cost flow problem also exist [16] and [30]. These algorithms do not necessarily cause the objective function to be non-decreasing during pivots. As a result, they are not primal network simplex rules in the usual sense.

Consider a directed graph $G = (N, A)$ with node set N and arc set A. A *directed path* is a sequence of arcs $(i_1, i_2), (i_2, i_3), (i_3, i_4), \ldots, (i_{k-1}, i_k)$, such that $(i_j, i_{j+1}) \in A \ \forall j \in \{1, \ldots, k-1\}$, $i_r \neq i_s \ \forall\, r \neq s; r, s \in \{1, \ldots, k\}$. An *undirected path* is defined in the same way as a directed path except that the orientation of the arcs along the path is inconsequential. A *directed cycle* is defined in a similar way to a directed path, except that in this case the start node i_1 and the end-node i_k must be the same. For the purpose of this paper, whenever we refer to a path or a cycle, we are referring to a directed path or cycle respectively. The cost of a path P is the sum of the costs on the arcs in the path P and is denoted by $c(P)$. The cost of a cycle W is the sum of the costs of the arcs in the cycle W, and is denoted by $c(W)$.

Given a flow x, it is possible to construct a *residual network* $G(x)$ which is a function of the original graph $G = (N, A)$, and the flow x. For each arc $(i, j) \in A$ we add arcs $(i, j)'$ and $(j, i)'$

in the residual network $G(x)$ having *residual capacities* $r_{ij} = u_{ij} - x_{ij}$ and $r_{ji} = x_{ij}$ respectively. The costs of the arcs $(i,j)'$ and $(j,i)'$ in the residual network are c_{ij} and $-c_{ij}$ respectively. (Thus, it may be possible that the residual network may contain mutiple parallel arcs between two nodes as a result of addition of the residual arcs corresponding to oppositely directed arcs in the original network.) Thus, the residual network $G(x)$ with respect to the flow x is defined to be the network $(N, A(x))$, where $A(x)$ consists of the set of all arcs with positive residual capacity.

Associated with each node i we have a ***dual multiplier*** or ***dual node potential*** which is denoted by $\pi(i)$. The *reduced cost* of an arc with respect to a set of dual mutipliers π is equal to $c_{ij} - \pi(i) + \pi(j)$, and is denoted by c^{π}_{ij}. The reduced cost of a path P is the sum of the reduced costs of the arcs on the path, and is denoted by $c^{\pi}(P)$. A flow x is said to be ϵ-optimal with respect to the node multipliers π, if for all unsaturated arcs (i,j) with $x_{ij} < u_{ij}$, $c^{\pi}_{ij} \geq -\epsilon$, and for all arcs (i,j) with $x_{ij} > 0$, $c^{\pi}_{ij} \leq \epsilon$. The premultiplier method is essentially a cost scaling algorithm, which starts off with an ϵ-optimal flow for $\epsilon = nC$. The method scales ϵ by a factor of 2 in each phase until ϵ is strictly less than $1/n$. The idea of ϵ-scaling was first developed by Goldberg and Tarjan in [13].

We also need to define the predecessor indices for a rooted tree T with root r. The predecessor index of a node i is denoted by ***pred(i)***, and is the first node on the unique tree path P_{ir} from node i to the root r. The predecessor index of the root node r is ***null***. An arc (i,j) in a rooted tree T is defined to be ***upward pointing***, if $j = pred(i)$. Otherwise (i,j) is said to be ***downward pointing***.

This paper is organized as follows. In the next section, we discuss the premultiplier method due to Orlin [23]. In Section 3, we describe the various operations possible using the dynamic tree data structure. The details of the implementation of the data structure are discussed in Section 4. In Section 5, we show how to use this data structure to implement the algorithm in $O(nm \cdot \log n \cdot \min\{m \cdot \log n, log(nC)\})$ time. Finally, in Section 6, we present a brief conclusion and summary.

2 The Premultiplier Method

In this section, we shall present Orlin's polynomial time primal simplex algorithm [23] for the minimum cost flow problem. The *premultiplier method* derives its name from the fact that it uses the concept of ***premultipliers***, rather than the usual dual variables called ***simplex-multipliers***. Premultipliers are maintained in such a way that the reduced costs of the tree arcs are not necessarily zero. The algorithm maintains a rooted simplex tree at each iteration. A node of the tree is designated as the *p-root*. The p-root may change from iteration to iteration. This is quite different from the usual simplex algorithm, in which any node may be chosen as the root for the purpose of a simplex pivot. The concept of *p-root* is very useful in maintaining these premultipliers. We shall now explain, how the premultipliers are defined and maintained.

A critical assumption on which Orlin's algorithm depends is that of non-degeneracy. This assumption is easy to accomplish without loss of generality. One can increase the supply of node 1 by ϵ for some strictly positive but small value of ϵ, and decrease the supply of every other node by $\epsilon/(n-1)$. Under the assumption that ϵ is sufficiently small, it can be shown that any optimal spanning tree solution for the perturbed problem is also optimal for the original problem, and every spanning tree flow is non-degenerate. Under the assumption that all supplies, demands and capacities are integral, one can choose $\epsilon = (n-1)/n$. We refer the reader to Orlin [23] for a detailed description of the perturbation approach which is also called *strong feasibility*. The latter concept was developed independently by Cunninghan [9] and Barr, Glover, and Klingman [7].

Since the simplex tree is nondegenerate, we may assume that whenever the simplex tree $T(x)$ contains the arc (i,j), it also contains the arc (j,i), and vice-versa.

A set of node numbers π are defined to be premultipliers with respect to a rooted tree T, if and only if for every tree-arc $(i,j) \in T$, such that if $j = pred(i)$, $c^{\pi}_{ij} \leq 0$, and if $i = pred(j)$, $c^{\pi}_{ij} \geq 0$. In other words, all upward pointing arcs in the tree have a reduced cost which is non-positive, and all downward pointing arcs have a reduced cost which is non-negative.

Consider the forest F, which is formed by all the arcs (i,j) in the simplex tree $T(x)$, which satisfy $c^{\pi}_{ij} = 0$. We shall refer to F as the *zero-forest.*

The zero forest consists of a number of *zero-subtrees.* Among these subtrees the subtree which contains the *p-root* is referred to as the *primary zero-subtree.*

A node is eligible, if it is a node of the primary zero-subtree. Equivalently, a node is eligible, if and only if the unique directed path from the node to *p-root* contains only arcs with zero reduced cost.

An arc (i,j) in the residual network $G(x) = (N, A(x))$ is defined to be *ϵ-eligible*, if i is eligible, $(i,j) \in A(x) - T(x)$ and $c^{\pi}_{ij} \leq -\epsilon$.

The important property of ϵ-eligible arcs is that the basic cycle which is formed by these arcs has reduced cost which is less than or equal to $-\epsilon$. This result has been established in [23]. We shall summarize this result in the following theorem.

Lemma 1 *The cost of the basic cycle which is formed by an ϵ-eligible arc is at most $-\epsilon$.*

Proof: Omitted. See [23]. □

The algorithm proceeds in a number of cost scaling phases. In the ϵ-scaling phase, the premultipliers are maintained in such a way that all reduced costs in the residual network are greater than or equal to $-\epsilon$. Thus, in this case, we define a flow x in the residual network $G(x) = (N, A(x))$ to be ϵ-optimal when for each arc $(i,j) \in A(x)$, $c^{\pi}_{ij} \geq -\epsilon$. In each phase ϵ is scaled by a factor of 2, until ϵ is less than $1/n$. It was established by Bertsekas [8], that when ϵ is less than $1/n$, the flow is optimal.

Because of the non-degeneracy assumption and Lemma 1, if the premultiplier method proceeds in such a way, so that the arc which is pivoted in during the ϵ-scaling phase is $\epsilon/4$-eligible, then the objective function will improve in each pivot. In a simplex pivot, the incoming arc (i,j) satisfies $c^{\pi}_{ij} < 0$. Unlike the simplex algorithm, the dual premultipliers may not necessarily be updated after each pivot. All that is required is that the node potentials remain as premultipliers. so as to make the reduced costs of such tree arcs equal to zero so long as all arcs with negative reduced costs in the tree point towards the *p-root.* Hence it is necessary to verify that after each pivot the premultiplier property is maintained. In order to maintain the premultiplier property, we find a new *p-root* so that all tree arcs continue to satisfy the premultiplier property.

If $(v,w) \in G(x)$ is the leaving arc (assume that (v,w) has the same orientation as the augmentation in the pivot cycle.) then v is chosen to be the new *p-root.* It can be proved [23] that such a pivot maintains the premultiplier property of the tree with respect to the new *p-root.* The result is summarized in the following theorem:

Lemma 2 *Let $G(x) = (N, A(x))$ be the current residual network. Suppose that $(k,l) \in A(x)$ is an eligible entering arc, and that $(v,w) \in A(x)$ is the leaving arc in a simplex pivot. Then, the new simplex tree is eligible with respect to the node v as the new* p-root.

```
Scaling Pre-Multiplier Method
begin
 let $x$ be any feasible spanning tree flow;
 let $\pi$ be the vector of simplex multipliers;
 $\epsilon := \max\{-c^{\pi}_{ij} : c^{\pi}_{ij} \leq 0, (i,j) \in A(x)\}$;
 while $x$ is not optimal do
 begin
   Improve-Approximation($x, \epsilon, \pi$)
   $\epsilon := \max\{-c^{\pi}_{ij} : c^{\pi}_{ij} < 0\}$;
 end
end
```

Figure 1: The scaling premultiplier method

Proof: Omitted. See [23]. □

The description of this algorithm is slightly different from that given in Orlin [23]. During the ϵ-scaling phase, only those arcs which are $\epsilon/4$-eligible are considered candidates for pivoting in. In order to find $\epsilon/4$-eligible arcs we maintain a data structure called the *CurrentArc* data structure. The Current Arc data structure is maintained in the form of an array whose length is equal to the number of nodes. For each node i, *CurrentArc(i)* is either ϕ or it points to some arc emanating from node i. Just before each simplex pivot we update the value of *Current Arc(i)* for each eligible node i so that it points to an arc emanating from node i which is $\epsilon/4$ eligible. In the event that no arc emanating from node i is $\epsilon/4$-eligible, then $CurrentArc(i) = \phi$, and we do not scan node i until its premultiplier increases to the next higher multiple of $\epsilon/4$ units. At that stage the value of $CurrentArc(\cdot)$ of that node is reset to the value of the first arc in the adjacency list for that node. ($FirstArc(i)$) (Hence, the arc list of a node is scanned only when its premultiplier is a multiple of $\epsilon/4$.) Since the premultipplier of any node increases by at most $3n \cdot \epsilon/2$ [23] this ensures that the arc list for each node is scanned at most $O(n)$ times. During each simplex pivot, we try to find some eligible node i such that $CurrentArc(i) \neq \phi$. In the event that no such eligible node exists, the premultipliers of all nodes are increased by the smallest amount which either changes the premultiplier of some node to a multiple of $\epsilon/4$ (hence making it candidate for scanning) or decreases the reduced cost of some basic arc emanating from the eligible set S to zero units. This will also ensure that the in each iteration either a new node can be scanned for $\epsilon/4$-eligible arcs or the set of eligible nodes increases, while at the same time maintaining the premultiplier property. The ϵ-scaling phase terminates the first time we try to relabel the premultipliers so that after the relabeling, each and every node would have been relabeled at least once. At this stage, the flow is $\epsilon/2$-optimal. For a detailed proof of this result, the reader is advised to refer to Orlin [23].

Theorem 1 *At the termination of the ϵ-scaling phase the flow is $\epsilon/2$-optimal.*

Since ϵ can be cut by a factor of 2 in each scaling phase, there will be a total of $O(log(nC))$ scaling phases. By using the method of arc-fixing suggested by Tardos [28], the number of scaling phases can be shown to be $m \cdot \log n$. We shall also reproduce some of the results which are present in Orlin's paper on the premultiplier method. The reader is assumed familiar with the results in [23]. We review some of the results of that paper as needed.

```
Procedure Improve-Approximation($x, \epsilon, \pi$);
begin
  $N^* = N$;
  { $N^*$ is the set of nodes which have been relabeled at least once}
  while $N^* \neq \phi$ do
  begin
    for each eligible node $i$ do UpdateCurrentArc(i);
    if there is an eligible node $i$ with CurrentArc($i$)$\neq \phi$ then
    begin
      $j$:=CurrentArc($i$);
      SimplexPivot($i, j$);
    end;
    else Modify-$\epsilon$-premultipliers;
  end
end
```

Figure 2: The Improve-Approximation Procedure

```
Procedure UpdateCurrentArc($i$)
begin
  while CurrentArc($i$) is not $\epsilon/4$-eligible and CurrentArc($i$) $\neq \phi$
    do CurrentArc($i$):=Next(CurrentArc($i$));
end
```

Figure 3: Updating the Current Arc

```
Procedure Modify-$\epsilon$-premultipliers
begin
  $S$:= { eligible nodes };
  $N^* := N^* - S$;
  if $N^* = \phi$ then terminate Improve-Approximation($x, \epsilon, \pi$);
  $\Delta_1 := \min\{-c_{ij}^{\pi} : (i,j) \in T, i \notin S, j \in S\}$;
  $\Delta_2 := \min\{\delta : \delta > 0$ and $\pi_i + \delta = 0 \bmod \epsilon/4$ for some eligible node $i$ }
  Increase $\pi$ by $\Delta := \min\{\Delta_1, \Delta_2\}$ for each node $i \in S$;
  for each eligible node $j$, if $\pi_j = 0 \bmod \epsilon/4$ then CurrentArc($j$):=FirstArc($j$);
end
```

Figure 4: Updating the premultipliers

Theorem 2 *The scaling premultiplier method requires at most $3n \cdot m$ pivots per scaling phase and requires $O(min\{m \cdot log(n), log(nC)\})$ scaling phases. The subroutine Modify-ϵ-premultipliers is executed $O(n \cdot m)$ times per scaling phase. Each scaling phase requires $O(n^2 \cdot m)$ time.*

The details of the primal network simplex algorithm are illustrated in Figures 1, 2, 3, and 4.

3 The dynamic tree data structure

We use two different dynamic tree data structures for the purpose of this algorithm. One keeps track of the residual arc capacities and the other keeps track of the node premultipliers. The first structure which keeps track of the residual capacities, is an extension of the original Sleator-Tarjan dynamic tree data structure (see, for example, [27]) which allows each tree arc to have two capacities, one in each direction. Thus, associated with each arc (i, j) in the dynamic tree, we have two numbers, which are $g(i, j)$ and $g(j, i)$. This data structure supports the following operations:

make-tree(i): Create a one-node tree with node i.

cut(i): Break the tree containing node i into two by deleting the arc joining i and its parent. Node i must not be a root node.

link(i, j, x, y): Combine the trees containing i and j, by making i the parent of j. Define $g(i, j) = x$, and $g(j, i) = y$. Before the link operation i and j must be in different trees, and i is the root of the tree to which it belongs.

change-value(i, Δ): Add real number Δ to $g(j, parent(j))$, and subtract Δ from $g(parent(j), j)$ for every non-root ancestor j of i.

evert(i): Reroot the tree containing the node i. The new root is i.

find-parent(i) : Return $parent(i)$. Return null if i is the tree root.

find-value(i) : Return $g(i, parent(i))$. If i is a tree root, then return infinity.

find-min-value(i): Return the ancestor j of i, which is such that the value of $g(j, parent(j))$ is as small as possible.

The contribution of this paper is another separate data-structure which is used to represent the node pre-multipliers. This structure is used to represent the zero-forest as a dynamic tree. We define such trees as the *premultiplier based trees* since they are used to keep track of the dual premultipliers. Each node i in these trees has a value which is called $value(i)$. In addition to the standard link, cut and evert operations, we need to support the following operations in these trees:

find-node-value(i): Return *value(i)*.

add-value(i, δ): If i is a tree root, add δ to the value of every node in the tree with root i. Node i must be a tree root for this to apply.

find-min-node(i): Return a descendent j of i which has the minimum value.

change-node-value(i, l): Change the value of node i to l. In order to implement these operations, we use the dynamic tree data structure described in [?], except that we store the values in difference form: for a node i in the virtual tree (see [12]), if i is a root, we store *value(i)* at i. If i is not a root, then we store $value(i) - value(VirtualParent(i))$. This allows computation of $value(i)$ buy summing values up the path to the virtual tree root, and performance of *addvalue*$(\cdot)$ by adding δ to the virtual tree root. A second value must be stored in each node to help perform the *find-min-node* operations. The details are straightforward; see [26], [27], and [29].

4 Dynamic trees implementation of the premultiplier algorithm

In this section, we shall show how to reduce the running time to $O(\log n)$ per pivot. We shall primarily concentrate on the details of finding entering arcs for pivots, and updating the premultipliers of the nodes in the eligible set. The details for updating flow values using dynamic trees can be found in [26], [27], and [29]. In order to find the eligible arcs for pivots we shall maintain two values associated with each node i, which are BASIC-SCAN(i) and NEXT-SCAN(i) respectively. These values are defined as follows:

NEXT-SCAN(i): It is the absolute difference between the current pre-multiplier of the node and the next higher multiple of $\epsilon/4$.

BASIC-SCAN(i): It is the minimum amount by which we need to increase the pre-multiplier of node i before the reduced cost of some tree-arc which is currently not in the zero-forest becomes 0. Note that it is possible for the BASIC-SCAN(i) to be infinity.

In order to efficiently maintain NEXT-SCAN(i) and BASIC-SCAN(i), we maintain three copies of the premultiplier based dynamic tree structure. The first keeps track of the node premultipliers (*value(i)*=$\pi(i)$), the second keeps track of the values NEXT-SCAN$(\cdot)$ (*value(i)*=NEXT-SCAN(i)), and the third keeps track of the values BASIC-SCAN$(\cdot)$ (*value(i)*=BASIC-SCAN(i)). We shall now proceed to describe how to implement the different steps of the algorithm.

In the event that the pre-multipliers of the zero-subtree containing the *p-root* are to be updated by Δ, we use the *addvalue(p-root, Δ)* operation on the copy of the dynamic tree which maintains the pre-multipliers. At the same time, the BASIC-SCAN values and the NEXT-SCAN values need to be increased by $-\Delta$. This can again be done using the *add-value* operation. While updating the premultipliers we also need to find the minimum amount Δ by which to increase the premultipliers of the nodes before either the pre-multiplier of some node becomes divisible by $\epsilon/4$, or the reduced cost of some tree arc which is currently not in the zero forest goes to zero. In order to do this we evaluate the minimum value of BASIC-SCAN$(\cdot)$ as well as as the minimum value of NEXT-SCAN$(\cdot)$ over all nodes in the primary subtree. Let these values be Δ_1 and Δ_2 respectively. Note that the *find-min-node* operation needs to be used in order to find these values. Thus Δ_1 denotes the minimum value by which the premultipliers of the nodes in the primary zero-subtree need to increase before the reduced cose of some tree-arc which is not currently in the zero-forest becomes 0. Similarly Δ_2 represents the minimum amount by which the premultiplier of a node in the primary zero-subtree has to increase before it becomes divisible by $\epsilon/4$. We can then update the premultipliers of all nodes in the primary subtree by $\Delta = \min\{\Delta_1, \Delta_2\}$.

Note that whenever the *link(j, i ·, ·)* operation is performed on the dynamic tree representation of the zero-forest, we need to update the value of BASIC-SCAN(i) using the *change-node-value* operation. The new value of BASIC-SCAN(i) is the minimum among the reduced costs of all tree arcs emanating from node i which have a positive reduced cost. We can do this by maintaining a heap at each node i which is called *basic-heap(i)*. This heap at node i maintains the set of arcs (i,j) which satisfy $\{(i,j) \in T, c^{\pi}_{ij} > 0\}$. In the heap, the value which we maintain is the offset from the minimum value of c^{π}_{ij} in the heap. Thus, if (k,l) be the arc in the heap with the minimum reduced cost, then the value stored for any arc (i,j) in the heap is equal to $c^{\pi}_{ij} - c^{\pi}_{kl}$. This ensures that the heap structure does not change when all the premultiplieres in the primary subtree are updated by the same amount. This is because the reduced cost of all the arcs in the heap change by the same amount, when the premultipliers in the zero-subtree are updated by the same amount. Note that updating the pre-multipliers of all nodes in the primary-subtree does not change the heap unless the reduced cost of a tree-arc in some heap (say *basic-heap(i)*) goes to zero, in which case it needs to be deleted from *basic-heap(i)*. Similarly, a cut operation deletes an arc from the basis, which may delete an arc from *basic-heap(i)*. A similar technique can be used in order to deal with cut-operations. Also, the addition of an $\epsilon/4$-eligible arc (i,j) to the basic-tree during a simplex pivot will cause *basic-heap(j)* to change. The arc (j,i) needs to be inserted into *basic-heap(j)* in this case.

We shall now proceed to analyze the time-complexity of each step of this method.

- **Time for performing the relabels:** Since each set of relabels in the primary subtree requires $O(\log n)$ time, and there are at most $O(nm)$ such sets of relabeling operations in each scaling phase, it follows that the time for relabeling is $O(nm \cdot \log n)$ per scaling phase.

- **Time for performing the pivots:** Since each pivot requires $O(\log n)$ time, the total time for performing the pivot operations is $O(nm \cdot \log n)$.

- **Time required for heap operations:** The number of *add-heap(.)* operations is bounded above by the number of simplex pivots which is $O(nm)$. The number of *delete-heap* operations can be charged to the number of *add-heap* operations, which is again $O(nm)$. Thus the total time for the heap operations is $O(nm \cdot \log n)$. *Find-min(·)* is called after a "cut" or "add" or basic arc going to zero. Each of these is bounded by $O(n \cdot m)$ per scaling phase (Orlin [23]). Thus, the total time for the *Find-min(·)* operations is given by $O(nm \cdot \log n)$.

- **Time for performing the arc scans:** It has already been established in [23] that the time for performing arc scans during a scaling phase is at most $O(nm)$.

- **Time for performing the *link* and *cut* operations:** The *number* of *link(.)* and *cut(.)* operations on the forest F is proportional to the number of simplex pivots. (The same result holds for the tree T.) Thus, the total time required of these operations is $O(nm \cdot \log(n))$.

Summing up the time required for all these operations we obtain the desired result.

Theorem 3 *The dynamic tree implementation of the premultiplier method requires $O(nm \cdot \log n)$ time per scaling phase.*

5 Conclusions and Summary

In this paper we presented an implementation of the primal simplex method which is both strongly polynomial, and also achieves a running time which is competitive with the best cost-scaling algorithm of Goldberg and Tarjan except for very dense graphs in which case it is at most $O(\log n)$ times slower. The extension of the dynamic tree data structure for this method is of independent interest in itself, and may possibly have other applications.

References

[1] Ahuja R. K., Goldberg A. V., Orlin J. B., and Tarjan R. E. Finding minimum-cost flows by double-scaling. *Mathematical Programming.* 53, 243-266.

[2] Ahuja R. K., Magnanti T. L. and Orlin J. B. Network Flows:Theory Algorithms and Applications. *Prentice Hall, Englewood Cliffs.* 1993.

[3] Ahuja R. K. and Orlin J. B. 1992. The scaling network simplex algorithm. *Operations Research.* 40 (1992), Supplement 1, S5-S13.

[4] Akgul M. Shortest Path and Simplex Method. Research Report, Department of Computer Science and Operations Research, North Carolina State University, Raleigh, N. C. (1985).

[5] Akgul M. A Genuinely Polynomial Primal Simplex Algorithm for the Assignment Problem. Research Report, Department of Computer Science and Operations Research, North Carolina State University, Raleigh, N. C. (1985).

[6] Armstrong R. and Jin Z. A strongly polynomial dual network simplex algorithm. *Mathematical Programming.* to appear.

[7] Barr R., Glover F., and Klingman D. The Alternating Path Basis Algorithm for the Assignment Problem. *Mathematical Programming.* 12, (1977) 1-13.

[8] Bertsekas D. P. A distributed algorithm for the assignment problem. Working Paper, Laboratory for Information and Decision Sciences, MIT, Cambridge, MA.

[9] Cunningham, W. H. 1976. A Network Simplex Method. *Mathematical Programming.* 11, 105-116.

[10] Dial R., Glover F., Karney D., and Klingman D. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks* 9, 215-248.

[11] Galil Z., and Naamad A. An $O(EV\log^2 V)$ algorithm for the maximal flow problem. *Journal of Comput. System Sci.* 21 (1980), pp. 203-217.

[12] Goldberg A. V., Grigoriadis M. D., and Tarjan R. E. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Mathematical Programming.* 50 (1991) 277-290.

[13] Goldberg A. V., and Tarjan R. E. Solving Minimum Cost Flow Problem by Successive Approximation. *Mathematics of Operations Research.* (1990) 15, 430-466.

[14] Goldfarb D., and Hao J. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Mathematical Programming* 47, (1990) 353-365.

[15] Goldfarb D. and Hao J. On strongly polynomial variants of the Network Simplex Algorithm for the maximum flow problem. *Operations Research Letters.* 10, (1991) 383-387.

[16] Goldfarb D., and Hao J. Polynomial Simplex Algorithms for the Minimum Cost Flow Problem. *Algorithmica.* 8, (1992), 145-160.

[17] Goldfarb D., Hao J., and Kai S. Efficient Shortest Path Simplex Algorithms. *Operations Research.* 38,(1990) 624-628.

[18] Hung M. S. A Polynomial Simplex Method for the Assignment Problem. Operations Research 31, (1983) 595-600.

[19] Kennington J. L., and Helgason R. V. *Algorithms for Network Programming.* 1980. Wiley-Interscience, N. Y.

[20] Orlin J. B. Genuinely Polynomial Simplex and Non-Simplex Algorithms for the Minimum Cost Flow Problem. Technical Report No. 1615-84, Sloan School of Management, MIT, Cambridge, MA.

[21] Orlin J. B. On the Simplex Algorithm for Networks and Generalized Networks. *Mathematical Programming Study.* 24, 1985, 166-178.

[22] Orlin J. B. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Operations Research.* 41, 1993, 338-350.

[23] Orlin J. B. A polynomial time primal simplex algorithm. *Unpublished Manuscript.*

[24] Orlin J. B., Plotkin S. A., and Tardos E. Polynomial Dual Network Simplex Algorithms. *Mathematical programming.* 60, 255-276.

[25] Roohy-Laleh E. *Improvements in the Theoretical Efficiency of the Network Simplex Method.* Unpublished Ph. D. Dissertation, Carleton University, Ottawa, Canada.

[26] Sleator D. D. and Tarjan R. E. A data structure for dynamic trees. *Journal of Computer and System Sciences.* 26, (1983) 362-391.

[27] Sleator D. D. and Tarjan R. E. Self-adjusting binary search trees. *Journal of the Association of Computing Machinery.* 32, (1985) 652-686.

[28] Tardos E. 1985. A strongly polynomial minimum cost circulation algorithm. *Combinatorica.* 5, 247-255.

[29] Tarjan R. E. *Data Structures and Network Algorithms.* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983).

[30] Tarjan R. E. 1991. Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem. *Mathematics of Operations Research.* 16, 272-291.

[31] Zadeh, N. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming.* 5, 255-266.