A Fast Algorithm for the Bipartite Node Weighted Matching Problem on Path Graphs with Application to the Inverse Spanning Tree Problem

by

Ravindra K. Ahuja
James B. Orlin

SWP# 4006 February 1998

A Fast Algorithm for the Bipartite Node Weighted Matching Problem on Path Graphs with Application to the Inverse Spanning Tree Problem

Ravindra K. Ahuja*
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

James B. Orlin
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

(Revised February 14, 1998)

* On leave from Indian Institute of Technology, Kanpur 208 016, INDIA.

A Fast Algorithm for the Bipartite Node Weighted Matching Problem with Application to the Inverse Spanning Tree Problem

Ravindra K. Ahuja* and James B. Orlin**

ABSTRACT

In this paper, we consider the bipartite node weighted matching problem on a special class of graphs, called *path graphs*, and develop a highly efficient algorithm for solving it. This matching problem arose while solving the inverse spanning tree problem defined as follows. Given an undirected graph $G^0 = (N^0, A^0)$ with n nodes, m arcs, and an arc cost vector c, and a spanning tree T^0, the inverse spanning tree problem is to perturb the arc cost vector c to a vector d so that T^0 is a minimum spanning tree with respect to the cost vector d and the cost of perturbation given by $|d - c| = \sum_{(i,j)\in A} |d_{ij} - c_{ij}|$ is minimum. We show that the dual of the inverse spanning tree problem is a bipartite node weighted matching problem on a path graph that contains m nodes and as many as (m-n+1)(n-1) = O(nm) arcs. We first transform the node weighted matching problem into a specially structured minimum cost flow problem and use its special structure to develop an $O(n^3)$ algorithm. We next use its special structure more effectively and use a more sophisticated transformation to solve the node weighted matching problem in $O(n^2 \log n)$ time. This approach also yields an $O(n^2 \log n)$ algorithm for the inverse spanning tree problem and improves the previous $O(n^3)$ algorithm.

* Sloan School of Management, MIT, Cambridge, MA 02139, USA; On leave from Indian Institute of Technology, Kanpur 208 016, INDIA.

** Sloan School of Management, MIT, Cambridge, MA 02139, USA.

1. INTRODUCTION

In this paper, we study the bipartite node weighted matching problem on a special class of graphs, called *path graphs*, and consider its application to the inverse spanning tree problem. Let $G^0 = (N^0, A^0)$ be a connected undirected network consisting of the node set N^0 and the arc set A^0. Let $n = |N^0|$ and $m = |A^0|$. We assume that $N^0 = \{1, 2, ... , n\}$ and $A^0 = \{a_1, a_2, ..., a_m\}$. We denote by tail[j] and head[j] the two endpoints of the arc a_j. Since each arc a_j is undirected, we can make any of its endpoints as tail[j] and the other endpoint as head[j]. Let T^0 be a spanning tree of G^0. We assume that the arcs are indexed so that $T^0 = \{a_1, a_2, ... , a_{n-1}\}$. We refer to the arcs in T^0 as *tree arcs* and arcs not in T^0 as *nontree arcs*. In the spanning tree T^0, there is a unique path between any two nodes; we denote by $\mathbf{P}[a_j]$ the set of tree arcs contained between the two endpoints of the nontree arc a_j. In this paper, we use the network notation such as tree, spanning tree, matching, rooted tree, path, directed path, as in the book of Ahuja, Magnanti and Orlin [1993]. We represent a path as i_1-i_2-i_3- ... - i_k with the implicit understanding that all the arcs (i_1, i_2), (i_2, i_3), ... , (i_{k-1}, i_k) are present in the network. We refer to the nodes i_2, i_3, ... , i_{k-1} as the *internal nodes* in the path i_1-i_2-i_3- ... - i_k. We first define the problems studied in this paper.

Bipartite Node Weighted Matching Problem. Let $\mathbf{G} = (V, E)$ be a bipartite graph with $V = V_1 \cup V_2$, $E \subseteq V_1 x V_2$ and node weights given by the vector w. Let M be an (arc) matching in $\mathbf{G}$, and let M^V denote the set of matched nodes in M, that is, the set of nodes in V that are incident to some arc in M. The bipartite node weighted matching problem is to identify a matching M for which $\sum_{i \in M^V} w_i$ is minimum.

Path Graph. A path graph is a bipartite graph defined with respect to an undirected graph G^0 and a spanning tree T^0 in G^0. As above, we assume that that $T^0 = \{a_1, a_2, ... , a_{n-1}\}$. We denote the path graph with respect to G^0 and T^0 by $G' = (N', A')$ and define it as follows. The node set $N' = N_1' \cup N_2'$ satisfies $N_1' = \{1, 2, ..., n-1\}$ and $N_2' = \{n, n+1, ..., m\}$ and the arc set A′ is obtained by considering each nontree arc a_j one by one and adding the arc (i, j) for each $a_i \in \mathbf{P}[a_j]$; that is, $A' = \{(i, j) : a_i \in \mathbf{P}[a_j], 1 \le i \le n\text{-}1, \text{and } n \le$

$j \leq m\}$. Observe that the path graph contains m nodes and as many as (m-n+1)(n-1) = O(nm) arcs.

Inverse Spanning Tree Problem. The data of the inverse spanning tree problem consists of a spanning tree T^0 of G^0 and an arc cost vector c with c_j denoting the cost of the arc a_j. The objective in the inverse spanning tree problem is to find an arc cost vector d such that T^0 is the minimum cost spanning tree with respect to d and such that $\sum_{j=1}^{n} |d_j - c_j|$ is minimum.

We show in this paper that the dual of the inverse spanning tree problem is a bipartite node weighted matching problem on a path graph that contains m nodes and as many as (m-n+1)(n-1) = O(nm) arcs. This allows us to solve the inverse spanning tree problem using any algorithm for the bipartite node weighted matching problem. The node weighted matching problem is a matroid and hence a greedy algorithm can be used to obtain its optimal solution (see, for example, Papadimitriou and Steiglitz [1982]). We however concentrate on developing faster algorithms in this paper. We first transform the node weighted matching problem into a specially structured minimum cost flow problem where all arcs have unit or infinite capacity and only the arcs entering the sink node have nonzero costs. We show that we can solve this minimum cost flow problem by solving a sequence of graph reachability problems and use its special structure to develop an $O(n^3)$ algorithm.

We next focus on developing an even faster algorithm. Our minimum cost flow formulation of the inverse spanning tree problem has O(nm) arcs, which for completely dense networks becomes $O(n^3)$. Since any minimum cost flow algorithm must look at each arc at least once, it appears difficult to improve the running time of the minimum cost flow algorithm unless we can represent the minimum cost flow problem more compactly. We next present an equivalent formulation of the minimum cost flow problem that uses only O(m log n) arcs. This formulation allows the node weighted matching problem to be solved in $O(n^2 \log n)$ time. This approach yields an $O(n^2 \log n)$ algorithm for the inverse spanning tree problem, which improves the previous best time bound of $O(n^3)$ for solving this problem due to Sokkalingam, Ahuja and Orlin [1996].

2. APPLICATION TO THE INVERSE SPANNING TREE PROBLEM

In this section, we show the application of the bipartite node weighted matching problem to inverse optimization. Inverse optimization is a relatively new area of research within the operations research community. Ahuja and Orlin [1998a, 1998b, 1998c] describe several applications of inverse optimization, and describe solution techniques for solving the general inverse optimization problem, inverse linear programming problem, and inverse network flow problems. Sokkalingam, Ahuja and Orlin [1996] have studied the inverse spanning tree problems.

In this section, we show that the inverse spanning tree problem can be transformed to a bipartite node weighted matching problem on the path graph. Consider the graph $G^0 = (N^0, A^0)$ where T^0 is a specified spanning tree. Recall that the objective in the inverse spanning tree problem is to perturb the arc cost vector c to d such that T^0 is the minimum cost spanning tree with respect to d and such that $\sum_{j=1}^{m} |d_j - c_j|$ is minimum. It is well known (see, for example, Ahuja, Magnanti and Orlin [1993]) that T^0 is a minimum spanning tree with respect to the arc cost vector d if and only if it satisfies the following optimality conditions:

$$d_i \le d_j \text{ for each arc } a_i \in \mathbf{P}[a_j] \text{ and for each } j = n, n+1, \ldots, m. \tag{1}$$

Now observe from (1) that increasing the cost of tree arcs and decreasing the cost of nontree arcs does not take the tree T^0 closer to satisfying the optimality conditions. This observation implies that there exists an optimal cost vector d such that $d = c + \alpha$, $\alpha_i \le 0$ for each i = 1, 2, ..., (n-1), and $\alpha_j \ge 0$ for each j = n, n+1, ..., m. This observation allows us to formulate the inverse spanning tree problem as follows:

$$\text{Minimize } \sum_{j=n}^{m} \alpha_j \ - \ \sum_{i=1}^{n-1} \alpha_j \tag{2a}$$

subject to

$$c_i + \alpha_i \le c_j + \alpha_j \text{ for each } a_i \in \mathbf{P}[a_j] \text{ and for each } j = n, n+1, \ldots, m, \tag{2b}$$

$$\alpha_i \le 0 \text{ for each } i = 1 \text{ to } (n-1), \text{ and } \alpha_j \ge 0 \text{ for each } j = n, n+1, \ldots, m. \tag{2c}$$

or, equivalently,

$$\text{Maximize} \quad \sum_{i=1}^{n-1} \alpha_j \ - \ \sum_{j=n}^{m} \alpha_j \tag{3a}$$

subject to

$$\alpha_i - \alpha_j \leq c_j - c_i \quad \text{for each } (i, j) \in A', \tag{3b}$$

$$\alpha_i \leq 0 \text{ for each node } i \in N_1' \text{ and } \alpha_j \geq 0 \text{ for each node } j \in N_2', \tag{3c}$$

where the graph $G' = (N_1' \cup N_2', A)$ is the path graph defined with respect to the tree T^0. Recall from Section 1 that each node $i \in N_1'$ corresponds to the tree arc a_i and each node $j \in N_2'$ corresponds to the nontree arc a_j. If we associate a dual variable x_{ij} with the arc (i, j) in (3b), then the dual of (3) can be stated as follows:

$$\text{Minimize} \sum_{(i,j) \in A'} (c_j - c_i)\, x_{ij} = \sum_{j \in N_2'} c_j \Big(\sum_{\{i:(i,j) \in A'\}} x_{ij} \Big) - \sum_{i \in N_1'} c_i \Big(\sum_{\{j:(i,j) \in A'\}} x_{ij} \Big) \tag{4a}$$

subject to

$$\sum_{\{j:(i,j) \in A'\}} x_{ij} \leq 1 \qquad \text{for each node } i \in N_1', \tag{4b}$$

$$\sum_{\{i:(i,j) \in A'\}} x_{ij} \leq 1 \qquad \text{for each node } j \in N_2', \tag{4c}$$

$$x_{ij} \geq 0 \qquad \text{for each arc } (i, j) \in A'. \tag{4d}$$

Clearly, (4) is a mathematical formulation of the bipartite node weighted matching problem on the path graph, where we associate a weight of $-c_i$ with any node $i \in N_1'$ and a weight of c_j for each node $j \in N_2'$. For every matching M of G', we may represent M by its vector x defined as $x_{ij} = 1$ for every arc $(i, j) \in M$ and $x_{ij} = 0$ for every arc $(i, j) \notin M$. We will also refer to x as a matching.

3. AN $O(n^3)$ ALGORITHM

In this section, we describe a simple $O(n^3)$ algorithm to solve the inverse spanning tree problem. An improvement of this algorithm using a more sophisticated data structure will be presented in the next section. Our algorithm first transforms the bipartite node weighted matching problem to the minimum cost flow problem. It then uses the successive shortest path algorithm to solve the minimum cost flow problem and uses its special structure to speed up various computations. In the subsequent discussion, we shall refer to the bipartite node weighted matching problem on the path graph more briefly as the node weighted matching problem.

We now describe the transformation of the node weighted matching problem in $G' = (N', A')$ to a bipartite minimum cost flow problem in a network which we represent by $G = (N, A)$. This minimum cost flow problem has the following nodes: (i) a set N_1 of (n-1) *left* nodes, one left node i corresponding to each arc $a_i \in T^0$; (ii) a set N_2 of m *right* nodes, one right node j corresponding to each arc $a_j \in A^0$ including arcs of T^0; (iii) a source node s; and (iv) a sink node t. The minimum cost flow problem has the following arcs: (i) a *source arc* (s, i) from the source node s to every left node i; (ii) *a sink arc* (j, t) from each right node j to the sink node t; (iii) an arc (i, i) from every left node i to the corresponding right node i (this arc corresponds to a slack variable); and (iv) an arc (i, j) from a left node i to the right node j for every arc (i, j) in the path graph G′. We define the supply/demand vector b as follows: $b(s) = -b(t) = (n-1)$, and $b(i) = 0$ for every other node i. In the network G, we set the capacity of each source and sink arc to 1, and set the capacity of each remaining arc to infinity. Finally, we set the cost of each sink arc (j, t) to c_j and the cost of all other arcs to zero. Let $\mu = \sum_{j \in T^0} c_j$. The following result establishes a connection between the node weighted matching problem in G' and the minimum cost flow problem in G.

Lemma 1. *For every feasible matching x′ in the network G′, there exists a feasible flow x in G satisfying* $cx = c'x' + \mu$. *Conversely, for every feasible flow x in G, there exists a feasible matching x′ with* $cx = c'x' + \mu$.

Proof. Consider a feasible matching in G′. Let $N_1'(M)$ and $N_1'(U)$ respectively, denote the sets of matched and unmatched nodes in N_1'. Similarly, let $N_2'(M)$ and $N_2'(U)$, respectively, denote the sets of matched and unmatched nodes in N_2'. Observe that in G′ the contribution of a matched arc (i, j) to the objective function (4a) is $c_j - c_i$. Notice that $c'x' = \sum_{\{(i,j) \in N'\}} (c_j - c_i)\, x'_{ij} = \sum_{\{j \in N_2'(M)\}} c_j - \sum_{\{i \in N_1'(M)\}} c_i$. We obtain the flow x in the network G corresponding to the matching x′ as follows. We send one unit of flow along the path s-i-j-t for every arc (i, j) that satisfies $x'_{ij} = 1$, and one unit of flow along the path s-i-i-t for every node $i \in N_1'(U)$. Observe that $cx = \sum_{\{j \in N_2'(M)\}} c_j + \sum_{\{i \in N_1'(U)\}} c_i$. Then, $cx - c'x' = \sum_{\{i \in N_1'\}} c_i = \mu$. Hence, $cx = c'x' + \mu$, completing the proof of one part of the theorem. To prove the converse result, let x be a flow in G. We

obtain a matching x' from x in the following manner: we let $x'_{ij} = 1$ if $x_{ij} = 1$, $i \neq s$, $j \neq t$, and $i \neq j$; otherwise $x'_{ij} = 0$. The proof that $cx = c'x' + \mu$ is similar to the proof of the first part. ♦

The minimum cost flow problem in the network G satisfies the following properties: (i) each source and sink arc in the network has a unit capacity; (ii) there are (n-1) source arcs; and (iii) all arcs other than the sink arcs have zero cost of flow. These properties allow us to solve the minimum cost flow problem in $O(n^3)$ time using simple data structure. We first need to define some additional notation. Consider the network $G = (N, A)$, where $N = \{s, t\} \cup N_1 \cup N_2$. For any node $j \in N_2$, we let $A(j) = \{i : i \in N_1$ and $(i, j) \in A\}$. Thus, A(j) is the set of incoming arcs at node j. Let x be a feasible flow in the network G. With respect to the flow x, we call an arc $(i, j) \in N_1 \times N_2$ a *matched arc* if $x_{ij} = 1$. If $x_{ij} = 1$, then nodes i and j are both said to be *matched.* Let $M_2(x)$ denote the set of matched nodes in N_2 with respect to the flow x.

We define the residual network G(x) with respect to the flow x as follows. We replace each arc $(i, j) \in A$ by two arcs (i, j) and (j, i). The arc (i, j) has cost c_{ij} and residual capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j, i) has cost $c_{ji} = - c_{ij}$ and residual capacity $r_{ji} = x_{ij}$. The residual network consists only of arcs with positive residual capacity. In the residual network G(x), let $R_1(x)$ and $R_2(x)$, respectively, denote the sets of nodes in N_1 and N_2 which are reachable from node s (that is, have directed paths from node s). We can determine the sets $R_1(x)$ and $R_2(x)$ using a graph search algorithm. The search time is proportional to the number of arcs in the residual network G(x), which in our case is O(nm).

We use the successive shortest path (minimum cost flow) algorithm to solve the minimum cost flow problem in the network G. The successive shortest path algorithm is a well-known algorithm to solve the minimum cost flow problem (see, for example, Ahuja, Magnanti and Orlin [1993]). This algorithm starts with $x = 0$ and proceeds by augmenting flow along shortest (directed) paths from node s to node t in the residual network G(x). Observe that any directed path from node s to node t will contain exactly one sink arc, and the cost of the path will equal the cost of the sink arc. Consequently, the shortest path in G(x) will contain the smallest cost sink arc among all sink arcs emanating from nodes in $R_2(x)$. We state this result as a property.

Property 1. *Let $c_q = min\{c_j : j \in R_2(x)\}$, and let P[q] be any directed path from node s to node q. Then the directed path P[q]-t is a shortest path in G(x) from node s to node t.*

Our algorithm for the node weighted matching problem uses Property 1 but not directly. Its direct use requires computing the set $R_2(x)$ which takes O(nm) time, since the network G contains O(nm) arcs. We will show how we can identify a shortest path in G(x) from node s to node t in $O(n^2)$ time. Our algorithm instead determines $R_1(x)$ and for each node $i \in R_1(x)$ determines a directed path from node s to node i which we represent by S(i). Assuming that $R_1(x)$ has been determined, the following result allows us to determine the reachability of any node $j \in N_2$.

Property 2. *There is a directed path from node s to a node $j \in N_2$ if and only if $R_1(x) \cap A(j)$ is nonempty.*

Our algorithm for the node weighted matching problem first orders the nodes in N_2 in the nondecreasing order of the costs c_j's. Let the vector σ denote the resulting node ordering, that is, $c_{\sigma(1)} \leq c_{\sigma(2)} \leq .... \leq c_{\sigma(m)}$. The algorithm then examines nodes in this order and uses Property 2 to determine the first unmatched node q that is reachable from the source node s. The node order ensures that Property 1 is satisfied and the shortest augmenting path from node s to node t passes through node q; subsequently, the algorithm augments a unit flow along this path. If an unmatched node is not reachable from node s, then it can be easily proved that it will not be reachable in subsequent stages. Thus the algorithm need not reexamine any node in N_2. Figure 1 gives an algorithmic description of the inverse node weighted matching algorithm.

```
algorithm inverse spanning tree;
begin
    let σ denote an ordering of the nodes in N_2 in the nondecreasing order of c_j's;
    x : = 0;
    compute R_1(x) ⊆ N_1;
    label all nodes in R_1(x) and unlabel all other nodes in N_1;
    for j := 1 to m do
    begin
        q : = σ[j];
        if there is a labeled node in A(q) then
        begin
            select a labeled node p in A[q];
            augment one unit of flow in the path s-S[p]-q-t;
            update x and G(x);
            compute R_1(x);
            mark all nodes in R_1(x) as labeled and all other nodes in N_1 as unlabeled;
        end;
    end;
    x is an optimal flow in the network G;
end;
```

Figure 1. The node weighted matching algorithm.

We next study the worst-case complexity of the node weighted matching algorithm. Let d_{max} denote the maximum indegree of a node $j \in N_2$. It follows that $d_{max} \geq |A(j)|$ for each $j \in N_2$. In the worst-case, d_{max} can be as large as n-1, but it may be much smaller as well. We will determine the running time of the algorithm in terms of d_{max}. The algorithm takes O(m log m) = O(m log n) time to determine the node ordering σ. An iteration of the for loop examines each arc in A(q) to find a labeled node p (Operation 1). In case it succeeds in finding a labeled node, then it augments one unit of flow along the shortest path (Operation 2); updates x and G(x) (Operation 3); computes $R_1(x)$ (Operation 4) and labels nodes in N_1 (Operation 5). Clearly, Operation 1 takes $O(d_{max})$ time per node in N_2 and $O(m\ d_{max})$ overall. Operations 2 through 5 are performed whenever an augmentation takes place. There will be exactly (n-1) augmentations because an augmentation saturates a source arc, there are (n-1) source arcs, and eventually each source arc will be saturated. An augmentation contains at most 2n+2 nodes because its internal nodes alternate between nodes in N_1 and N_2 and $|N_1| = n$. Consequently, Operations 2 and 3 require O(n) time per iteration and $O(n^2)$ overall.

We next focus on Operation 4 that involves computation of $R_1(x)$. Let $M_2(x)$ denote the set of matched nodes in N_2 with respect to x. Any directed path from node s to a node p in N_1 in G(x) is of the form $s\text{-}i_0\text{-}j_1\text{-}i_1\text{-}j_2\text{-}i_2\text{-} \dots \text{-}j_k\text{-}i_k$, where each of the arcs $(j_1, i_1), (j_2, i_2), \dots, (j_k, i_k)$ is a reversal of a matched arc in x. Hence all the nodes $j_1, j_2, \dots, j_k$ are matched nodes in x. In other words, any directed path in G(x) from node s to a node p in N_1 must have all arcs incident to nodes in $M_2(x)$ (except the first arc which is a source arc). This observation allows us to compute $R_1(x)$ by applying the graph search algorithm to a smaller subgraph $G^s = (N^s, A^s)$ defined as follows: $N^s = \{s\} \cup N_1 \cup M_2(x)$ and $A^s = \{(s, i) : i \in N_1\} \cup \{(i, j) \text{ in } G(x) : i \in M_2(x) \text{ or } j \in M_2(x)\}$. Since $M_2(x) \le (n-1)$, we can construct $G^s(x)$ in $O(n\ d_{max})$ time and run the graph search algorithm to find all nodes reachable from node s in the same time. A graph search algorithm not only finds the nodes reachable from node s, it also finds the directed paths to those nodes which it stores in the form of predecessor indices. Hence Operation 4, which consists of computing $R_1(x)$, takes $O(n\ d_{max})$ time per iteration and $O(n^2\ d_{max})$ time overall. After computing $R_1(x)$, we label nodes in N_1 in O(n) time. We summarize our discussion with the following result.

Theorem 1. *The node weighted matching algorithm solves the node weighted matching problem, and hence the inverse spanning tree problem in $O(n^2\ d_{max})$ time, where d_{max} is the maximum indegree of any node in N_2.*

Since $d_{max} = O(n)$, we immediately get a bound of $O(n^3)$ for both the problems. This time bound matches the time bound of the algorithm by Sokkalingam et al. [1996] for the inverse spanning tree problem. However, the algorithm given here uses simpler data structure and is easier to implement. An additional advantage of this algorithm is that its running time can be further improved to $O(n^2 \log n)$ by using some additional transformations. We study this speedup in the next section.

4. AN $O(n^2 \log n)$ ALGORITHM

In this section, we develop an improved implementation of the $O(n^3)$ algorithm for the node weighted matching problem developed in Section 3. A fundamental bottleneck operation of the minimum cost flow formulation given earlier is that each

node in N_2 may have as many as n-1 incoming arcs. Thus the minimum cost flow network may contain as many as $\Omega(nm)$ such arcs. If the original spanning tree problem is defined on a dense network, this bound becomes $\Omega(n^3)$. Since any minimum cost flow algorithm must look at each arc at least once, we obtain a lower bound of $\Omega(n^3)$ for dense minimum cost flow problems obtained through this formulation. Our improvement described in this section in based on a transformation that reduces the number of arcs in the minimum cost flow formulation from O(nm) to O(m log n) and improves the running time of the algorithm from $O(n^3)$ to $O(n^2 \log n)$.

Notation and Definitions

We now introduce some additional notation. We will henceforth conceive of the spanning tree T^0 as a tree rooted at node 1. We visualize the tree T^0 as if it is hanging down from node 1. We use the notation that arcs in the tree denote the predecessor-successor relationship with the node closer to the root being the predecessor of the node farther from the root. A node in the tree can have multiple successors but each non-root node has exactly one predecessor. We denote the predecessor of node i by pred(i) and follow the convention that pred(1) = 0. We define the descendants of a node i to be the set of all nodes belonging to the subtree of T^0 rooted at node i, that is, containing node i, its successors, successors of its successors, and so on. We denote by d(i) the number of descendants of node i. We assume without any loss of generality that for any node its child with the maximum number of descendants is its leftmost child.

Consider a tree arc (i, j) with j = pred(i). As per Sleator and Tarjan [1983], we call an arc (i, j) *heavy* if $d(i) \geq \frac{1}{2} d(j)$, that is, node i contains at least half of the descendants of node j. An arc which is not heavy is called a *light* arc. Notice that since the descendant set of nodes includes node i, node i will have at most one heavy arc going to one of its successors. If a node has a heavy arc directed to its successor, then this arc will be node's leftmost arc. We denote by $\mathcal{H}$ the set of heavy arcs in T^0 and by $\mathcal{L}$ the set of light arcs in T^0. We define *a heavy path* as a path in T^0 consisting entirely of heavy arcs and which is maximal in the sense that none of its supersets satisfies this property. We define the *root* of a heavy path as the node on the path closest to node 1. We refer to a subset of a heavy path as a *heavy subpath*. We illustrate the definition of heavy arcs using the numerical example given in Figure 2. In the figure, we show the light arcs by thin lines and heavy arcs by thick lines. The tree has three heavy paths: 1-2-4-7-9-12, 5-8-10-13, and 3-6, with roots as 1, 5, and 3, respectively.

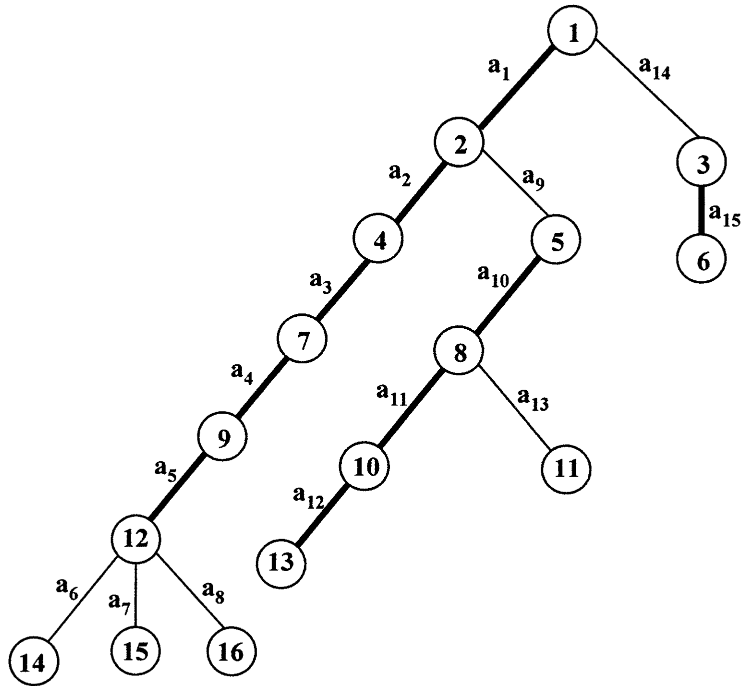

Figure 2. The initial tree T^0.

We point out that our definitions of the heavy arcs have been adapted from the dynamic tree data structure due to Sleator and Tarjan [1983]. The two properties we prove next, Property 3 and Lemma 2, can also be found in the same reference.

Property 3. *The set $\mathcal{H}$ of heavy arcs comprises of a collection of a node-disjoint heavy subpaths.*

Proof. Any node in T^0 is incident to at most two heavy arcs - one going towards to its successors and one to its predecessor. This fact together with the acyclicity of the tree T^0 implies the property. ♦

Each node i in the tree T^0 has a unique path to the root node which we call the *predecessor path* and denote it by P[i]. We can efficiently identify the predecessor path P[i] by tracing the predecessor indices starting at node i. Now consider a predecessor path from any node k to the root node. This path is a sequence of heavy and light arcs, where heavy subpaths alternate with light arcs. The following lemma shows that a predecessor path will have O(log n) light arcs and, hence, O(log n) heavy subpaths.

Lemma 2. *A predecessor path can have at most O(log n) light arcs and at most O(log n) heavy subpaths.*

Proof. Consider the predecessor path of node k in T^0 starting at node k and ending at the root node. Observe that as we move along the predecessor path, the number of descendants of the nodes encountered in the path strictly increase. Further, whenever we encounter a light arc (i, j) with j = pred(i), then the number of descendants are at least doubled (because $d(i) < \frac{1}{2} d(j)$). Since the tree T^0 has n nodes, we can have at most $\lfloor \log n \rfloor$ light arcs. Since each heavy subpath lies in between two light arcs (except the subpaths at the beginning and at the end of the predecessor path), there can be at most $\lfloor \log n \rfloor + 1$ heavy subpaths, completing the proof of the lemma. ♦

We will assume in the subsequent discussion that arcs in the tree T^0 are numbered so that all arcs in each heavy path are consecutively numbered. We accomplish this by performing a depth-first search of the tree T^0 and numbering the arcs in the order they are examined. While performing the search, we follow the convention that arcs corresponding to the children of each node are examined from left to right. (The tree in Figure 2 shows such an ordering of arcs.) This convention together with the fact that any

heavy arc is a leftmost arc implies that arcs will be renumbered in a manner so that arcs in each heavy path (or, subpath) are consecutive.

Type 1 and Type 2 Subpaths

We are now in a position to describe the basic idea behind our improvement. The running time of the inverse spanning tree algorithm described in the previous section is $O(n^2 d_{max})$, where d_{max} is the maximum indegree of any node in N_2. For the minimum cost flow formulation described earlier, d_{max} could be as large as n and the running time of the algorithm is $O(n^3)$. In the new equivalent formulation described in this section, $d_{max} = O(\log n)$, and the running time becomes $O(n^2 \log n)$.

Consider any nontree arc a_j in the original graph. In the previous formulation, a node $j \in N_2$ has an incoming arc from every node $i \in N_1$ if arc $a_i \in \mathbf{P}[a_j]$. Recall that $\mathbf{P}[a_j]$ is the set of all tree arcs between the two endpoints of the arc a_j. Observe that $\mathbf{P}[a_j] = P[tail[j]] \cup P[head[j]] - P[tail[j]] \cap P[head[j]]$. We call the node where the two predecessor paths P[tail[j]] and P[head[j]] meet as the *apex* of the path $\mathbf{P}[a_j]$ and denote it by *apex[j]*.

The set $\mathbf{P}[a_j]$ may contain light as well as heavy arcs. By Lemma 2, $\mathbf{P}[a_j]$ contains O(log n) light arcs, but may contain as many as (n –1 – log n) heavy arcs. We thus need to handle heavy arcs carefully. Lemma 2 also demonstrates that $\mathbf{P}[a_j]$ contains O(log n) heavy subpaths and each such heavy subpath is a part of a heavy path. Each heavy subpath in $\mathbf{P}[a_j]$ is one of the following two types: it contains the root of the heavy path (*Type 1 subpath*) or it does not contain the root of the heavy path *(Type 2 subpath*). For example, for the tree shown in Figure 2, if a_j = (12, 13) then $\mathbf{P}[a_j]$ contains one Type 1 subpath a_{10}-a_{11}-a_{12} and one Type 2 subpath a_2-a_3-a_4-a_5. It is easy to observe that $\mathbf{P}[a_j]$ can contain many Type 1 subpaths, but it can have at most one Type 2 subpath. If $\mathbf{P}[a_j]$ contains a Type 2 subpath, then this subpath contains apex[j]. Our new formulation defines a transformation to represent heavy subpaths in a manner so that each Type 1 subpath in $\mathbf{P}[a_j]$ contributes only one incoming arc to node j, and the Type 2 subpath in $\mathbf{P}[a_j]$ contributes O(log n) arcs. After these transformations, the total number of incoming arcs at node j are O(log n), which will lead to the necessary speedup.

We will denote the network corresponding to the new formulation by $\overline{G}$ = $(\overline{N}, \overline{A})$. We will explain later how to construct $\overline{G}$ efficiently. For now, we will explain the topological structure of $\overline{G}$. To construct it, we start with the graph G = (N, A) where we delete all arcs emanating from each left node $i \in N_1$ if a_i is a heavy arc; we however keep the arc (i, i). It follows from Lemma 2 that each right node in $\overline{G}$ has O(log n) incoming arcs at this stage. We have, however, modified the minimum cost flow problem because we have eliminated the incoming arcs in $\mathbf{P}(a_j)$ corresponding to arcs in the heavy subpaths. Observe that the arcs we have deleted had zero cost and infinite capacity. We next show how to add arcs and nodes to the network $\overline{G}$ so that the minimum cost flow problem in $\overline{G}$ is equivalent to the minimum cost flow problem in G. We will show that by adding O(n) nodes and O(m log n) arcs, we can ensure that for each arc (i, j) in G with the left node i and right node j, there is a directed path, say path[i, j], from node i to node j in $\overline{G}$ of zero cost and infinite capacity. Moreover, if arc (i, j) is not in G, then we will not create any path from node i to node j in $\overline{G}$. Using this property, any flow in G may be transformed into a flow in $\overline{G}$ as follows: for every x_{ij} units of flow sent on any arc (i, j) in G, we send x_{ij} units of flow on path[i, j] from node i to node j in $\overline{G}$. The converse result is also true. Given any flow x in $\overline{G}$, we first decompose the flow into unit flows along paths from node s to node t. For every unit of flow sent along the path s-path[i, j]-t in $\overline{A}$, we send a unit flow along the path s-i-j-t in A. This establishes one-to-one correspondence between flows in the networks G and $\overline{G}$ both of which have the same cost.

In the subsequent discussion, we describe in detail the method used to represent heavy subpaths in our transformation.

<u>Handling Type 1 Subpaths</u>

Consider a heavy path a_p-a_{p+1}- ... - a_q, with a_p as the root of this heavy path. For this heavy path, any heavy subpath of Type 1 will include exactly one of the following path segments: a_p, a_p-a_{p+1}, a_p-a_{p+1}-a_{p+2}, ... , a_p-a_{p+1}-a_{p+2}-...-a_q. We can handle these possibilities using the transformation given in Figure 3, where we expand the network $\overline{G}$ by adding the nodes $(\overline{p}, \overline{p+1}, ..., \overline{q})$ and adding the arcs $(h, \overline{h})$ for each h = p, p+1, ... , q, and the arcs $(\overline{h-1}, \overline{h})$ for each h = p+1, p+2, ... , q. Each arc in Figure 3 has zero cost and infinite capacity.

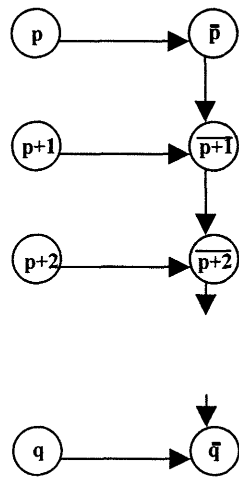

Figure 3. Construct for handling Type 1 subpaths.

Suppose that the tree path **P**[a_j] contains the Type 1 subpath a_p-a_{p+1}- ... - a_l for some l, $p \le l \le q$. The minimum cost flow formulation in G contains the arcs (p, j), (p+1, j), ... , (l, j). But in the new formulation, we will only add the arc ($\bar{l}$, j). It follows from our construct, as illustrated in Figure 3, that there is a path from node i to node j in $\overline{G}$ for each i = p, p+1, ... , l. Consequently, for each arc (h, j) in G, $l \le h \le p$, there is a directed path of the same cost and capacity in $\overline{G}$.

We introduce the construct described above in $\overline{G}$ for every heavy path in T^0. This construct allows each heavy subpath of Type 1 in any **P**[a_j] to be equivalently represented by a single arc in $\overline{G}$. Subsequent to this construction, there is a path in $\overline{G}$ from a node $i \in N_1$ to a node $j \in N_2$ if and only if there is an arc $(i, j) \in G$ and a_i is not a Type 2 subpath of **P**[a_j]. To summarize, the above construction allows us to represent each heavy subpath of Type 1 in **P**[a_j] by a single arc. Since any **P**[a_j] can contain at most O(log n) heavy subpaths of Type 1, $\overline{G}$ will have at most O(log n) incoming arcs on any node j after Type 1 heavy subpaths have been considered.

<u>**Handling Type 2 Subpaths**</u>

Consider again the heavy path a_p, a_{p+1}, ... , a_q with a_p as the root of the heavy path. For this heavy path, any subpath of Type 2 can start at any of the arcs a_p, a_{p+1}, ... , a_q and can terminate at any of the arcs a_p, a_{p+1}, ... , a_q. There are $\Omega((q-p+1)^2)$ such possibilities and our transformation should be able to handle all of them. We define a construct which will allow all of these possibilities by adding O(q-p+1) nodes to $\overline{G}$ and increasing the indegree of a node in N_2 by O(log n). First, we introduce more notation.

We insert q-p+1 additional nodes to $\overline{G}$ and construct a binary tree **T**[p, q] with nodes p, p+1, ... , q, as the leaf nodes of the binary tree; each arc in this binary tree has zero cost and infinite capacity. Figure 4 shows the construct for the heavy path 7-8-9- ... -21. We denote by *parent[i]* as the parent of node i in the binary tree. We refer to the two children of a same parent as *siblings*. For each node i in the binary tree, we let **D**[i] = $\{j : p \le j \le q$ and j is a descendant of node i$\}$. Observe that we denote by **D**(i) the set of descendants of node i that are also the leaf nodes of **T**[p, q]. Observe that **D**(i) is an interval of consecutive integers and can be compactly represented as $[\alpha_i, \beta_i]$, where α_i is

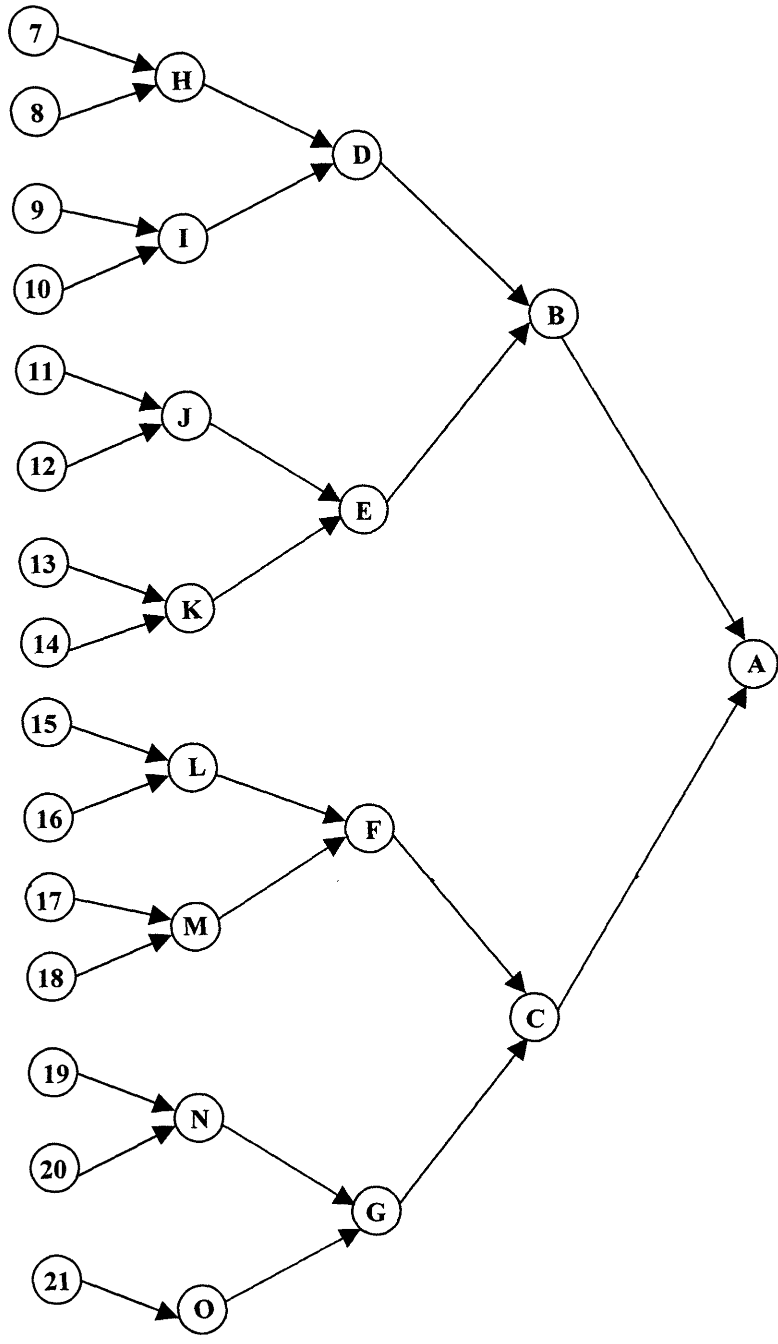

Figure 4. Construct for handling Type 2 subpaths.

the first integer and β_i is the last integer in the interval. For example, in Figure 4, **D**(B) = [7, 14] and **D**(F) = [15, 18].

Now consider **P**[a_j]. Suppose that it contains a heavy Type 2 subpath **S** of the heavy path $a_p, a_{p+1}, \dots, a_q$, where S = {$a_k, a_{k+1}, \dots, a_l$} with $p < k \le l \le q$. (Recall that any heavy path and any heavy subpath consists of consecutively numbered arcs.) We can thus alternatively represent the set **S** by [k, l]. We call a node i in the binary tree **T**[p, q] *maximal* with respect to **S** if $[\alpha_i, \beta_i] \subseteq [k, l]$ but $[\alpha_j, \beta_j] \not\subset [k, l]$ for j = parent[i]. For example, in Figure 4, if **S** = [11, 17], then the nodes E, L, and 17 are maximal while the remaining nodes are not. We denote the unique path from node k to node l in the binary tree **T**[p, q] by **P**[k, l]. We call a set of nodes in the binary tree tree **T**[p, q] to be a *cover* of **S** if $[k, l] = \bigcup_{i \in C[k,l]} [\alpha_i, \beta_i]$. The set of maximal nodes of **S** forms a cover of **S**. We denote it by **C**[k, l] and call it the maximal cover of **S**. For example, **C**[k, l] = {E, L, 17} is the maximal cover of **S** = [11, 17]. Recall that graph G contains an arc (i, j) for every node $i \in [k, l]$. But in the graph $\overline{G}$, we will add an arc (i, j) for every $j \in$ **C**[k, l]. It is easy to see that for each such arc (i, j) in G, there is a corresponding directed path from node i to node j in $\overline{G}$ with the same cost and same capacity. We will now show that |**C**[k, l]| = O(log n) and we can determine it in O(log n) time.

It is easy to verify that a cover is the maximal cover of **S** if and only if it does not contain two siblings. This result yields the following iterative method to determine **C**[k, l]. We start with **C**′[k, l] = {k, k+1, ... , l} and if **C**′[k, l] contains two siblings we replace them by their parent. We repeat this process until **C**′[k, l] has no siblings. Finally, we terminate with **C**′[k, l] = **C**[k, l]. Moreover, each node of **C**[k, l] is either a node in **P**[k, l] or a child of a node in **P**[k, l]. This result implies that there are only O(log n) nodes qualified to be in the set **C**[k, l] and yields the following more efficient algorithm to determine **C**[k, l]. We consider each node i in **P**[k, l] as well as the children of each node of **P**[k, l] and check each to see if it is a maximal node of [k, l], we add i to **C**[k, l]. This method may be implemented in O(log n) time.

To summarize, we handle Type 2 subpaths in the following manner. For each heavy subpath $a_p, a_{p+1}, \dots, a_q$ in T^0, we introduce the construct of a binary tree as shown in Figure 4. (We point out that this construct is a superimposition over the construct shown in Figure 3.) If some **P**[a_j] contains a Type 2 subpath **S**, then we determine its maximal cover **C**[k, l] and add the arc (i, j) for each $i \in$ **C**[k, l] to the network $\overline{G}$.

Suppose $i \in N_1$ and $j \in N_2$. Then, there is an arc from node i to node j in G if and only if there is a path from node i to node j in $\overline{G}$.

Determining Type 1 and Type 2 Subpaths

We will now describe a method to determine all Type 1 and Type 2 subpaths for any **P**[a_j], $n \le j \le m$. Notice that **P**[a_j] may contain as many as (n-1) arcs and if we scan all arcs in it while identifying all the Type 1 and Type 2 subpaths, then it would take a total of O(nm) time and would constitute the bottleneck operation in the algorithm. We will show how we can determine these subpaths for any **P**[a_j] in O(log n) time. To do so, we would need two additional indices for each node in the tree T^0, namely, depth[i] and root[i]. The index depth[i] gives the depth of node i in the tree T^0, that is, the number of arcs in the predecessor path from node i to node 1. We define root[i] = i if (i, pred[i]) is a light arc; otherwise, it is the root of the heavy path containing node i. For the tree T^0, these indices can be determined in a total of O(n) time.

We give in Figure 5, the procedure to determine the light arcs and heavy subpaths in any **P**[a_j] and add the corresponding arcs to the network $\overline{G}$. The algorithm assumes that we start with the network $\overline{G}$ = G, where we have not added arcs from nodes in N_1 to the nodes in N_2. We also assume that the constructs shown in Figures 3 and 4 needed to handle Type 1 and Type 2 subpaths have already been added to $\overline{G}$. The while loop in the procedure traces the predecessor indices starting at the endpoints of the arc a_j. For each light arc a_r encountered it adds the arc (r, j). For each heavy Type 1 subpath encountered, it identifies the corresponding heavy subpath using the root indices, adds an appropriate arc to $\overline{G}$ and moves to the root of the heavy path. At the termination of the while loop, there are three possibilities which we show in Figure 6: (i) $\alpha = \beta$ (as in Figure 6(a)); (ii) l = root[k] (as in Figure 6(b)); and (iii) $l \neq$ root[k] (as in Figure 6(c)). In case (i), there is no additional heavy subpath to consider; in case (ii), there is an additional subpath of Type 1 to consider; and in case (iii), there is an additional subpath of Type 2 to consider. The algorithm then adds the corresponding arcs to the network $\overline{G}$. The running time of this procedure is O(log n) since it devotes O(1) time per light arc or per Type 1 heavy subpath, and O(log n) time for a Type 2 heavy subpath. By Lemma 2 there are O(log n) light arcs or heavy subpaths in **P**[a_j] and at most one Type 2 heavy subpath.

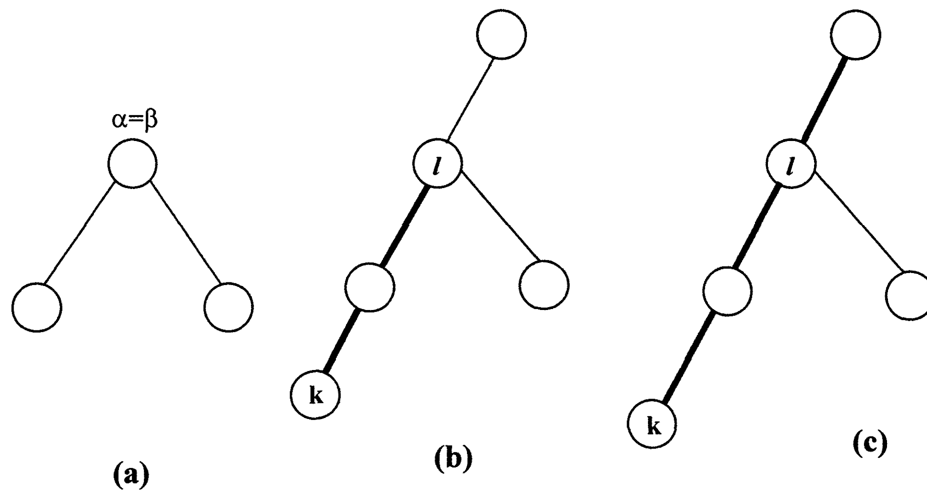

Figure 6. The three termination conditions.

```
procedure determine–subpaths(T^0, a_j);
begin
    α := tail[j];
    β := head[j];
    while root[α] ≠ root[β] do
        if depth[root[α]] < depth[root[β]] then scan(α) else scan(β);
    if α = β then return;
    if root[α] = α then k := β and l := α else k := α and l := β;
    if l = root[k] then add the arc (k̄, j) to Ḡ
    else compute the set C[k, l] and add the arc (i, j) to Ḡ for every i ∈ C[k, l];
end;

procedure scan(h);
begin
    let a_r := (h, pred[h]);
    if a_r is a light arc then add the arc (r, j) to Ḡ and set h := pred[h];
    if a_r is a heavy arc then add the arc (r̄, j) to Ḡ and set h := root[h];
end;
```

Figure 5. Adding arcs to $\overline{G}$ corresponding to heavy subpaths in P[a_j].

Worst-Case Complexity

To solve the node weighted matching problem, we solve the minimum cost flow problem in $\overline{G} = (\overline{N}, \overline{A})$ using a minor modification of the algorithm described in Figure 1. The modification consists of replacing the set N_1 by the set $\overline{N}_1$, where $\overline{N}_1$ consists of the nodes in N_1 plus all the additional nodes added by the constructs shown in Figures 3 and 4. We also replace $R_1(x)$ by $\overline{R}_1(x)$, where $\overline{R}_1(x)$ denotes all the nodes in $\overline{N}_1$ that are reachable from node s in the residual network $\overline{G}(x)$ with respect to the flow x. Since $|\overline{N}_1|$ is O(n), these changes do not affect the worst-case complexity of the algorithm, which remains as $O(n^2 d_{max})$. But since d_{max} = O(log n), the running time of the algorithm improves to $O(n^2 \log n)$. Hence the following theorem.

Theorem 2. *The improved node weighted matching algorithm solves the bipartite node weighted matching problem on the path graph, and hence the inverse spanning tree problem, in $O(n^2 \log n)$ time.*

ACKNOWLEDGEMENTS

We gratefully acknowledge support from the Office of Naval Research under contract ONR N00014-96-1-0051 as well as the grant from the United Parcel Service.

REFERENCES

Ahuja, R. K., and J. B. Orlin. 1998a. Inverse optimization, Part I: General problem and linear programming. Working Paper, Sloan School of Management, MIT, Cambridge, MA.

Ahuja, R. K., and J. B. Orlin. 1998b. Inverse optimization, Part II: Network flow problems. Working Paper, Sloan School of Management, MIT, Cambridge, MA.

Ahuja, R. K., and J. B. Orlin. 1998c. Combinatorial algorithms for inverse network flow problems. Working Paper, Sloan School of Management, MIT, Cambridge, MA.

Papadimitriou, C. H., and K. Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Inc., Englewood Cliffs, NJ.

Sleator, D. D., and R. E. Tarjan. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences* **24**, 362-391.

Sokkalingam, P. T., R. K. Ahuja, and J. B. Orlin. 1996. Solving inverse spanning tree problems through network flow techniques. Working Paper, Sloan School of Management, MIT, Cambridge, MA To appear in *Operations Research*.