A Faster Algorithm for the Inverse Spanning Tree Problem

Ravindra K. Ahuja
Department of Industrial & Systems Engineering
University of Florida,
Gainesville, FL 32611, USA
ahuja@ufl.edu

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

(February 14, 1998; Revised May 27, 1999)

A Faster Algorithm for the Inverse Spanning Tree Problem

Ravindra K. Ahuja and James B. Orlin

ABSTRACT

In this paper, we consider the inverse spanning tree problem. Given an undirected graph $G^0 = (N^0, A^0)$ with n nodes, m arcs, 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 specially structured graph (which we call the *path graph*) that contains m nodes and as many as (m-n+1)(n-1) = O(nm) arcs. We first transform the bipartite 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 develop an $O(n^2 \log n)$ time algorithm. This improves the previous $O(n^3)$ time algorithm due to Sokkalingam, Ahuja and Orlin [1999].

1. INTRODUCTION

In this paper, we study the inverse spanning tree problem. Inverse optimization is a relatively new area of research within the operations research community. Some references on inverse optimization include the following: Burton and Toint [1992 and 1994], Burton, Pulleyblank, and Toint [1997], Cai and Li [1995], Cai, Yang, and Li [1996], Xu and Zhang [1995], Yang and Zhang [1996], Yang, Zhang, and Ma [1997], Zhang and Cai [1998], Zhang, Liu, and Ma [1996], Zhang, Ma, and Cai [1995], and Ahuja and Orlin [1998a, 1998b]. This paper improves upon the previous algorithm of Sokkalingam, Ahuja and Orlin [1999] for the inverse spanning tree problem.

We first give some network notation. 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 c denote the arc cost vector. Let $n = |N^0|$ and $m = |A^0|$. We assume that $N^0 = \{1, 2, \ldots, n\}$ and $A^0 = \{a_1, a_2, \ldots, a_m\}$. 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]. In this paper, we use the network notation such as tree, spanning tree, matching, rooted tree, path, and directed path, as in the book of Ahuja, Magnanti and Orlin [1993]. We represent a path as a sequence of nodes 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 $i_2, i_3, \ldots, i_{k-1}$ as *internal nodes* in the path i_1-i_2-i_3- ... - i_k. Alternatively, we may also represent a path as a sequence of arcs a_1-a_2-a_3- ... - a_k with the implicit assumption that the consecutive arcs in the path have a common endpoint.

Let T^0 be a spanning tree of G^0. We assume that the arcs are indexed so that $T^0 = \{a_1, a_2, \ldots, a_{n-1}\}$. We refer to the arcs in T^0 as *tree arcs* and arcs not in T^0 as *nontree arcs*. 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 graph (which we call 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 show that we can solve this minimum cost flow in $O(n^3)$ time. We next use its special structure more effectively and develop an $O(n^2 \log n)$ time. This approach yields an $O(n^2 \log n)$ algorithm for the inverse spanning tree problem and improves the previous $O(n^3)$ time algorithm due to Sokkalingam, Ahuja and Orlin [1999].

2. FORMULATING THE INVERSE SPANNING TREE PROBLEM

In this section, we show that the inverse spanning tree problem can be transformed to a bipartite node weighted matching problem on a graph. 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. 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 $i = 1, 2, \ldots, (n-1)$, and $\alpha_j \ge 0$ for each $j = n, n+1, \ldots, m$. This observation allows us to formulate the inverse spanning tree problem as follows:

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

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,

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

subject to

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

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

where the graph $G' = (N', A') = (N_1' \cup N_2', A)$ is a bipartite graph defined with respect to the tree T^0 in the following manner. 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, $A' = \{(i, j) : a_i \in \mathbf{P}[a_j], 1 \le i \le n-1, \text{ and } n \le j \le m\}$. We refer to the graph G' as the *path graph.* Observe that the path graph contains m nodes and as many as (m-n+1)(n-1) = O(nm) arcs.

The formulation (3) is a linear programming problem. We will now take the dual of (3). 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:

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

subject to

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

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

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

In taking the dual, we have taken the liberty of replacing what should appear as (4c) by its negative. Notice that (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 $j \in N_2'$. (In a node weighted matching problem, we want to find a matching such that the sum of the weights of the matched nodes is maximum.) 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

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. This will lead to two nodes in G with label i for each $a_i \in T^0$; however, it will be clear from context which of these nodes is being referred to.

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 $\mu = \sum_{j \in T^0} c_j$. We will henceforth denote the cost of any arc (i, j) as c_{ij} and the capacity of the arc (i, j) as u_{ij}. 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 integral flow x in G satisfying cx = cx + m. Conversely, for every integral feasible flow x in G, there exists a feasible matching x' with cx = cx + m.*

Proof. Consider a feasible matching x′ 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 $c'x' = x'_{ij} = \sum_{\{j \in N_2'(M)\}} c_j - \sum_{\{i \in N_1'(M)\}} c_i$. We obtain an integral 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 an integral 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 structures. 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 $A(j) = \{i : i \in N_1 \text{ and } (i, j) \in A\}$. Thus, A(j) is the set of nodes adjacent to node $j \in N_2$. 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 network G and 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. 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 $c_q = min\{c_j : j \hat{I} R_2(x)\}$, 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 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 only $O(n^2)$ time rather than in O(nm) 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 $j \hat{I} N_2$ 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 in N_2 is not reachable from node s, then it can be easily proved that it will not be reachable in subsequent stages. (One can easily prove more generally that as more iterations are performed, no new nodes are added to $R_1(x)$ but its nodes may be deleted.) 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 node weighted matching;
begin
    let σ denote an ordering of the nodes in N2 in the nondecreasing order of cj's;
    x : = 0;
    compute R1(x) ⊆ N1;
    label all nodes in R1(x) and unlabel all other nodes in N1;
    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[p]-q-t;
            update x and G(x);
            compute R1(x), and S[i] for each node i ∈ R1(x);
            mark all nodes in R1(x) as labeled and all other nodes in N1 as unlabeled;
        end;
    end;
    x is an optimal flow in the network G;
end;
```

Figure 1. The node weighted matching algorithm on path graphs.

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); compute $R_1(x)$ and the path S[i] for each $i \in R_1(x)$ (Operation 4); and labels nodes in N_1 (Operation 5). 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 $|N_1| = n-1$. 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 $R_1(x)$ and paths S[i] for all $i \in R_1(x)$. 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 $s\text{-}i_0\text{-}j_1\text{-}i_1\text{-}j_2\text{-}i_2\text{-} \ldots \text{-}j_k\text{-}i_k$, where each of the arcs (j_1, i_1), (j_2, i_2), ..., (j_k, i_k) is a reversal of a matched arc in x. Hence all the nodes $j_1, j_2, \ldots, j_k$ 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 $M_2(x) \leq (n-1)$, 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 $R_1(x)$, the nodes reachable from node s, it also finds the directed paths to those nodes which it stores in the form of predecessor indices. Operation 4 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 on path graphs, 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, Ahuja and Orlin [1999] for the inverse spanning tree problem. The algorithm given in Figure 1 can also be implemented in $O(n^2 \log n)$ time using the dynamic tree data structure due to Sleator and Tarjan [1983]. However, the dynamic tree data structure has large computational overhead and is difficult to implement. In the next section, we describe another $O(n^2 \log n)$ algorithm that is simpler and is easier to implement. In fact, our improved implementation of the node weighted matching algorithm is the same as the one described above, except that it is carried out on a transformed network.

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

In this section, we develop an $O(n^2 \log n)$ implementation of the node weighted matching problem developed in Section 3. We first present some notation.

Notation and Definitions

We will 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. 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 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 desc(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 desc(i) ≥ ½ desc(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 one of its successors, then this arc will be the node's leftmost arc. We denote by H the set of heavy arcs in T^0 and by 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. We define the *root* of a heavy path as the node on the path closest to node 1. We refer to a subpath of a heavy path as a *heavy subpath*. We illustrate the definitions of heavy arcs and heavy paths

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.

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] (see, also, Tarjan [1983]). The following property is immediate.

Property 3. *The set H of heavy arcs defines a collection of a node-disjoint heavy paths.*

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 Q[i]. We can efficiently identify the predecessor path Q[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 may be expressed as a sequence of heavy and light arcs, where heavy subpaths alternate with light arcs. The following result due to Sleator and Tarjan [1983] states that a predecessor path will have O(log n) light arcs and, hence, O(log n) heavy subpaths.

Property 4. *A predecessor path contains O(log n) light arcs and O(log n) heavy subpaths.*

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.

Defining 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 node weighted matching 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} can be as large as n and the running time of the algorithm becomes $O(n^3)$. 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 right 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 in T^0 between the two endpoints of the arc a_j. Observe that $\mathbf{P}[a_j] = (Q[tail[j]] \cup Q[head[j]]) - (Q[tail[j]] \cap Q[head[j]])$. We call the node where the two predecessor paths Q[tail[j]] and Q[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 Property 4, $\mathbf{P}[a_j]$ contains O(log n) light arcs, but may contain as many as (n –1) heavy arcs (if T^0 is a path). We thus need to handle heavy arcs carefully. Property 4 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. $\mathbf{P}[a_j]$ contains O(log n) Type 1 subpaths and 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 a 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 except the arc (i, i). It follows from Property 4 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.

Handling Type 1 Subpaths

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 directed 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. (Notice that each node h = p, p+1, … , q is a left node.) Each arc in Figure 3 has zero cost and infinite capacity.

Suppose that the tree path $\mathbf{P}[a_j]$ 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 $(\overline{l}, j)$. 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, $p \le h \le l$, 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 $\mathbf{P}[a_j]$ to be equivalently represented by a single arc in $\overline{G}$. Since any $\mathbf{P}[a_j]$ 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.

Handling Type 2 Subpaths

Consider again the heavy path $a_p, a_{p+1}, \ldots, a_q$ 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 $a_p, a_{p+1}, \ldots, a_q$ and can terminate at any of the arcs $a_p, a_{p+1}, \ldots, a_q$. 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 up to q-p+2 additional nodes to $\overline{G}$ and construct a binary tree $\mathbf{T}[p, q]$ 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 $\mathbf{T}[p, q]$, we let $D[i] = \{j : p \le j \le q$ and j is a descendant of node i$\}$; that is, the set of descendants of node i that are also the leaf nodes of $\mathbf{T}[p, q]$. Observe that D[i] is an interval of consecutive integers; let α_i be the first integer in this interval and β_i is the last integer in the interval. Then, $D[i] = [\alpha_i, \beta_i]$. For example, in Figure 4, D[B] = [7, 14] and D[F] = [15, 18].

Now consider $\mathbf{P}[a_j]$. Suppose that it contains a heavy Type 2 subpath $\mathbf{S}$ of the heavy path $a_p, a_{p+1}, \ldots, a_q$, and $\mathbf{S} = \{a_k, a_{k+1}, \ldots, a_l\}$ with $p < k \le l \le q$. (Recall that any heavy path or subpath consists of consecutively numbered arcs.) We can thus alternatively represent the set $\mathbf{S}$ by [k, *l*]. We call a node i in the binary tree $\mathbf{T}[p, q]$ *maximal* with respect to $\mathbf{S}$ 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 $\mathbf{S} = [11, 17]$, 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 $\mathbf{T}[p, q]$ by $\mathbf{Path}[k, l]$. We call a set $\mathbf{C}$ of nodes in the binary tree $\mathbf{T}[p, q]$ to be a *cover* of $\mathbf{S}$ if $[k, l] = \bigcup_{i \in \mathbf{C}} [\alpha_i, \beta_i]$. The set of maximal nodes of $\mathbf{S}$ forms a cover of $\mathbf{S}$. We denote it by $\mathbf{C}[k, l]$ and call it the maximal cover of $\mathbf{S}$. For example, $\mathbf{C}[11, 17] = \{E, L, 17\}$ is the maximal cover of $\mathbf{S} = [11, 17]$. 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 (r, j) for every $r \in \mathbf{C}[k, l]$. 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 $|\mathbf{C}[k, l]| = O(\log n)$ and we can determine it in O(log n) time.

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

determine **C**[k, *l*]. We consider each node i in **Path**[k, *l*] as well as the children of each node of **Path**[k, *l*] and check each to see if it is a maximal node of [k, *l*]; if so, we add i to **C**[k, *l*]. This method can 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}, … , 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 ∈ **C**[k, *l*] to the network $\overline{G}$. Suppose i ∈ N_1 and j ∈ 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 ≤ j ≤ 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 procedure assumes that we start with the network G where we do not add arcs from nodes in N_1 to the nodes in N_2. We next add the constructs shown in Figures 3 and 4 needed to handle Type 1 and Type 2 subpaths. We refer to the network at this stage by $\overline{G}$. The procedure as it proceeds add more arcs 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 five possibilities to consider, which we show in Figure 6. The case (a) occurs when both the arcs in **P**[a_j] incident to apex[j] are light arcs; cases (b) and (c) occur when one of the arcs in **P**[a_j] incident to apex[j] is a heavy arc and apex[j] is connected to its predecessor by a light arcs; and cases (d) and (e) occur when one of the arcs in **P**[a_j] incident to apex[j] is a heavy arc and apex[j] is connected to its predecessor by a heavy arc. In case (a), we do not get any Type 1 or Type 2 subpath. In cases (b) and (c), we get a Type 1 subpath. In cases (d) and (e), we get a Type 2 subpath. Depending upon the case, the procedure adds appropriate arcs to $\overline{G}$. 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 Property 4, there are O(log n) light arcs or heavy subpaths in **P**[a_j] and at most one Type 2 heavy subpath.

```
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 (Case (a))
    else if β = root[α] then add the arc (ᾱ, j) to Ḡ (Case (b))
        else if α = root[β] then add the arc (β̄, j) to Ḡ (Case (c))
            else if depth[α] > depth[β] then
                    compute the set C[k, l] and add the arc (i, j) to Ḡ for every i ∈ C[k, l] (Case (d))
                else compute the set C[k, l] and add the arc (i, j) to Ḡ for every i ∈ C[k, l] (Case (e));
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 on path graphs, 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 $\overline{N}_1$ contains all 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 offer our sincere thanks to the referees for their insightful comments. In particular, the detailed comments by one of the referees helped us greatly to improve the presentation. We gratefully

acknowledge support from the Office of Naval Research under contract ONR N00014-98-1-0317 as well as a grant from the United Parcel Service.

REFERENCES

Ahuja, R. K., and J. B. Orlin. 1998a. Inverse Optimization. Working Paper, Sloan School of Management, MIT, Cambridge, MA. *Submitted for publication.*

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

Burton, D., B. Pulleyblank, and Ph. L. Toint. 1997. The inverse shortest paths problem with upper bounds on shortest paths costs. In *Network Optimization*, edited by P. Pardalos, D. W. Hearn, and W. H. Hager, *Lecture Notes in Economics and Mathematical Systems*, Volume 450, pp. 156-171.

Burton, D., and Ph. L. Toint. 1992. On an instance of the inverse shortest paths problem. *Mathematical Programming* **53**, 45-61.

Burton, D., and Ph. L. Toint. 1994. On the use of an inverse shortest paths algorithm for recovering linearly correlated costs. *Mathematical Programming* **63**, 1-22.

Cai, M., and Y. Li. 1995. Inverse matroid intersection problem. Research Report, Institute of System Science, Academia Sinica, Beijing, China. To appear *in ZOR-Mathematical Methods of Operations Research*.

Cai, M., X. Yang, and Y. Li. 1996. Inverse polymatroidal flow problem. Research Report, Institute of System Science, Academia Sinica, Beijing, China.

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. 1999. Solving inverse spanning tree problems through network flow techniques. *Operations Research* 47, 291-298.

R. E. Tarjan. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA.

Xu, S., and J. Zhang. 1995. An inverse problem of the weighted shortest path problem. *Japanese Journal of Industrial and Applied Mathematics* **12**, 47-59.

Yang, C., and J. Zhang. 1996. Inverse maximum capacity path with upper bound contraints. To appear in *OR Spektrum*.

Yang, C., J. Zhang, and Z. Ma. 1997. Inverse maximum flow and minimum cut problem. *Optimization* **40**, 147-170.

Zhang, J., and M. C. Cai. 1998. Inverse problem of minimum cuts, *Mathematical Methods of Operations Research* **47**, No. 1.

Zhang, J. and Z. Liu, 1996. Calculating some inverse linear programming problems, *Journal of Computational and Applied Mathematics* **72**, 261-273.

Zhang, J. Z. Liu, and Z. Ma. 1996. On the inverse problem of minimum spanning tree with partition constraints. *Mathematical Methods of Operations Research* **44**, 171-188.

Zhang, J., Z. Ma, and C. Yang. 1995. A column generation method for inverse shortest path problems, *ZOR-Mathematical Methods for Operations Research* **41**, 347-358.

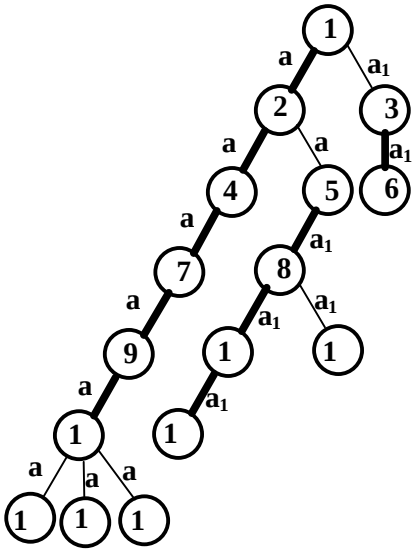

Figure 2. The initial tree

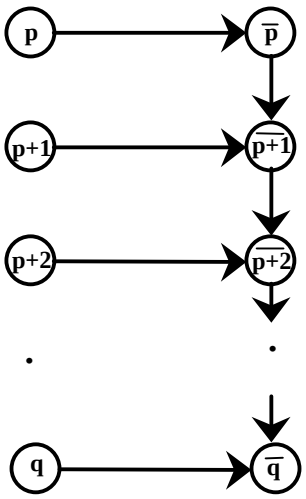

Figure 3. Construct for handling Type 1

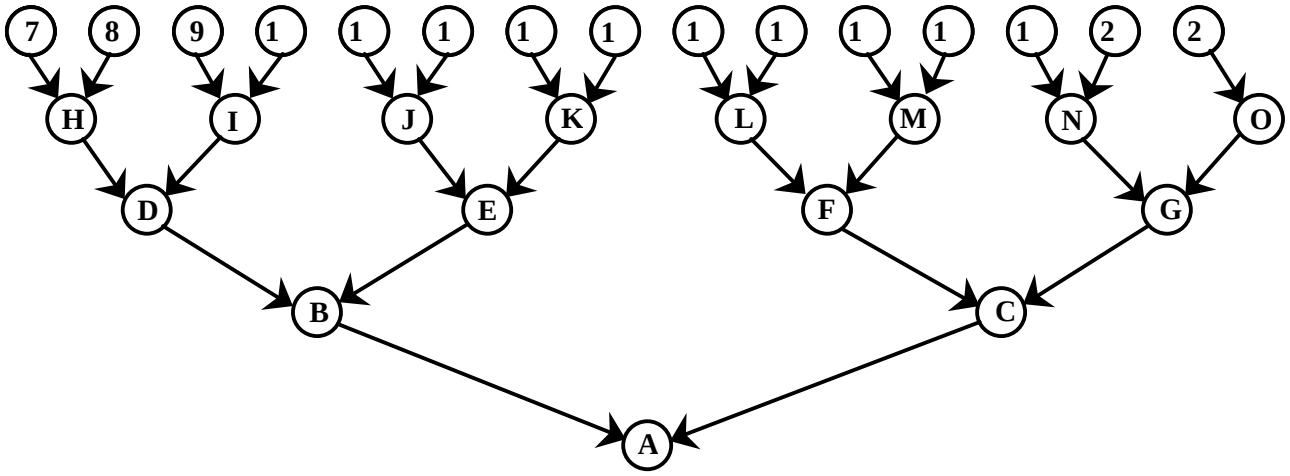

Figure 4. The construct for handling Type 2

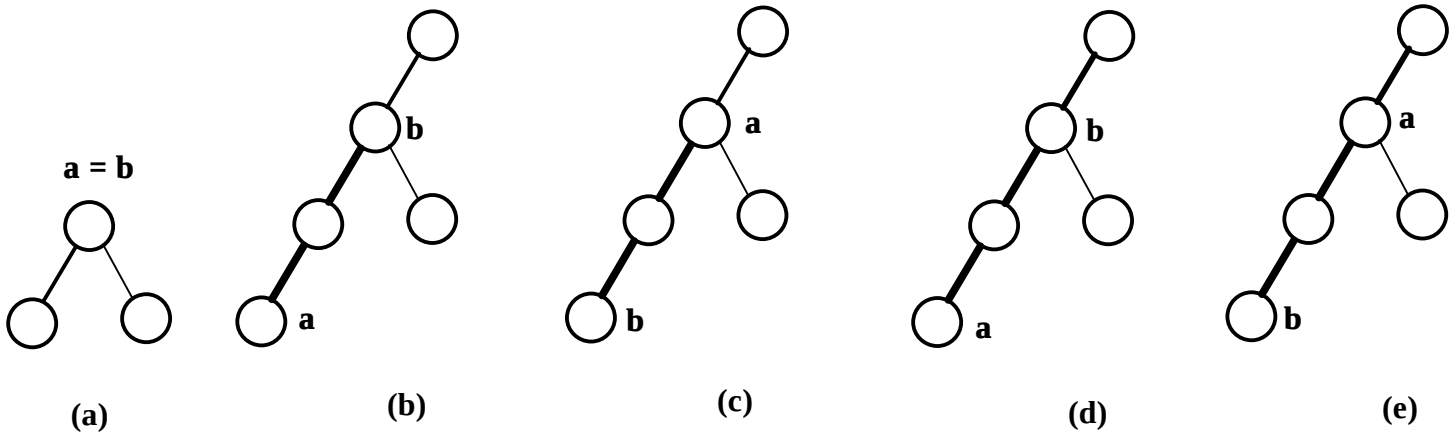

Figure 6. The five termination conditions.