EFFICIENT ALGORITHMS FOR THE INVERSE SPANNING-TREE PROBLEM

DORIT S. HOCHBAUM

Department of Industrial Engineering and Operations Research, Walter A. Haas School of Business, University of California, Berkeley, California 94720, hochbaum@ieor.berkeley.edu

The inverse spanning-tree problem is to modify edge weights in a graph so that a given tree T is a minimum spanning tree. The objective is to minimize the cost of the deviation. The cost of deviation is typically a convex function. We propose algorithms here that are faster than all known algorithms for the problem. Our algorithm's run time for any convex inverse spanning-tree problem is $O(nm\log^2 n)$ for a graph on n nodes and m edges plus the time required to find the minima of the n convex deviation functions. This not only improves on the complexity of previously known algorithms for the general problem, but also for algorithms devised for special cases. For the case when the objective is weighted absolute deviation, Sokkalingam et al. (1999) devised an algorithm with run time $O(n^2 m\log(nC))$ for C the maximum edge weight. For the sum of absolute deviations our algorithm runs in time $O(n^2\log n)$, matching the run time of a recent (Ahuja and Orlin 2000) improvement for this case. A new algorithm for bipartite matching in path graphs is reported here with complexity of $O(n^{1.5}\log n)$. This leads to a second algorithm for the sum of absolute deviations running in $O(n^{1.5}\log n\log C)$ steps.

Received August 2001; revision received October 2002; accepted November 2002.
Subject classifications: Programming: integer, algorithms, optimization in integers. Network/graphs: flow algorithms, parametric minimum cut. Mathematics: convexity, convex optimization problem.
Area of review: Optimization.

1. INTRODUCTION

The inverse spanning-tree problem is a problem in the class of "inverse optimization" problems. Inverse problem theory is a subject extensively studied in the context of tomographic studies, seismic wave propagation, and in a wide range of statistical inference with priors problems (Tarantola 1987, Barlow et al. 1972). In an inverse optimization problem a candidate-feasible solution is given, and the goal is to modify the cost parameters so that the given solution is optimal. The objective function is to modify the costs so as to minimize the penalty incurred by the modification of the cost parameters. In the inverse spanning-tree problem there is a given spanning tree T in an edge-weighted graph. The problem is to modify the edge weights so that the given tree is a minimum spanning tree and so that the cost of the deviation from the original weights is minimum. The penalty cost function is typically a convex function of the magnitude of the deviation.

The inverse minimum spanning-tree problem is defined on an undirected graph $G^T = (N', E)$ with edge weights c_e for all $e \in E$ and a spanning tree T. We let $|N'| = n$ and $|E| = m$. A subgraph (N', T) for $T \subseteq E$ is said to be a spanning tree if it is connected and acyclic.

A necessary and sufficient condition for a tree to be a minimum spanning tree is that each out-of-tree edge $i \in E\backslash T$ must have a weight c_i greater than or equal to each of the weights of the edges in the tree on the unique path between its endpoints.

Definition 1. For a spanning tree T let $\{a, b\} \in E\backslash T$ be an out-of-tree edge. Then, the unique path $P_{ab}(T)$ between a and b in T is said to be the *path induced* by $\{a, b\}$.

If T is not a minimum spanning tree, then the rank order constraints, $c_i \geqslant c_j$ for $i \in E\backslash T$, $j \in P_i(T)$, are violated for at least one pair (i, j). To ensure that the constraints are satisfied, the edge weights must be modified. The penalties for deviating from the given weights of the edges are convex functions of the type $f_e(x_e - c_e)$ for $e \in E$ that are minimized for $x_e = c_e$. Let $l = \min_{j\in E} c_j$ and $u = \max_{j\in E} c_j$. Our formulation of the inverse spanning-tree problem (IST) is thus:

$$\text{(IST)}\qquad \text{Min } \sum_{j\in E} f_j(x_j - c_j)$$
$$\text{subject to } x_i \geqslant x_j \quad \forall\, i \in E\backslash T,\ j \in P_i(T),$$
$$l \leqslant x_j \leqslant u \quad \forall\, j \in E.$$

We restrict the discussion to the problem in integer variables. The technique, with slight modifications, is also shown to be applicable to the continuous optimization problem, in §7.

The number of variables in this formulation is m and the number of constraints is the sum of the induced paths lengths, $\sum_{i\in E\backslash T} |P_i(T)|$, which is $O(mn)$. We refer hereafter to the number of variables and constraints in the IST formulation as N, M, respectively.

1.1. Prior Research

Prior research in the field of inverse optimization problems has addressed, in terms of solution approaches, only a limited collection of penalty functions. These are restricted to the norms L_1 (sum of absolute deviations), L_2 (quadratic deviations), L_∞ (minimizing maximum absolute deviation), and a variant of L_1, the weighted sum of absolute deviations.

The most efficient algorithms for IST were developed recently by Sokkalingam et al. (1999) and by Ahuja and Orlin (2000). Sokkalingam et al. (1999) studied the problem for three specific convex penalty functions: sum of absolute deviations where $f_i(x_i) = |x_i - c_i|$, weighted sum of absolute deviations where $f_i(x_i) = w_i|x_i - c_i|$, and maximum absolute deviation that seeks to minimize $\max_{i\in E} |x_i - c_i|$. The run times for these three cases are $O(n^3)$, $O(n^2 m \log(nC))$, and $O(n^2)$, respectively. Ahuja and Orlin (2000) further improved the algorithm for the objective function of the sum of absolute deviations, L_1, improving the complexity from $O(n^3)$ to $O(n^2 \log n)$. This improvement was achieved by considering a modified path graph, which we call the *reformulation.* This reformulation, with some adjustments, is important in achieving faster run times for the algorithms reported here as well.

Other efficient algorithms for the problem have been devised for problems that generalize IST. These algorithms were not devised specifically for the convex IST, but rather for the convex dual of minimum cost network flow, (DMCNF). In this dual formulation the dual variables of the flow balance constraints, x_j, and the dual variables of the capacity constraints, z_{ij}, in a graph $G' = (V', A')$ appear with convex cost functions $f_j(\,)$ and $e_{ij}(\,)$ in the objective:

$$\text{(DMCNF)} \qquad \text{Min} \sum_{j\in V'} f_j(x_j) + \sum_{(i,j)\in A'} e_{ij}(z_{ij})$$

$$\text{subject to } x_i - x_j \leqslant z_{ij} \quad \text{for } (i,j)\in A',$$

$$u_j \geqslant x_j \geqslant l_j, \quad j \in V',$$

$$z_{ij} \geqslant 0, \quad (i,j)\in A'.$$

Obviously, IST is a DMCNF problem with $z_{ij} = 0$.

Two different polynomial time algorithms were devised for the convex dual of minimum cost network flow in articles by Ahuja et al. (1999a, b). The algorithm reported in Ahuja et al. (1999a) has a running time of $O(MN \log(N^2/M) \log(NU))$, where $N = |V'|$, $M = |A'|$, and $U = \max_{j\in V'}\{u_j - l_j\}$. The algorithm reported in Ahuja et al. (1999b) has the running time of solving the minimum s,t-cut problem $\log U$ times. We refer to these algorithms as AHOa and AHOb, respectively.

When $z_{ij} = 0$, the minimum s,t-cut problem in the AHOb algorithm is solved on a graph with $O(N)$ nodes and $O(M)$ arcs for a total run time of $O(MN \log(N^2/M) \log U)$ using, e.g., the algorithm of Goldberg and Tarjan (1988) as the minimum cut algorithm.

Part of the contribution here is to consider the properties of the graph in which the minimum cut[1] is solved and derive better run times for this algorithm. Denoting by $T(n, m, L)$ the complexity of solving the minimum cut problem on a graph with n nodes, m arcs, and maximum *label* or distance from source to sink L, the complexity of the AHOb algorithm is $O(\log U \cdot T(N, M, L))$.

The convex inverse spanning-tree problem is also a special case of the convex cost-closure problem (CCC) studied by Hochbaum and Queyranne (2003). For a set of variables V' and a set of ordered pairs A', the formulation of the problem is

$$\text{(CCC)} \qquad \text{Min} \sum_{j\in V'} f_j(x_j)$$

$$\text{subject to } x_i - x_j \geqslant 0 \quad \forall (i,j)\in A',$$

$$l_j \leqslant x_j \leqslant u_j, \quad j\in V'.$$

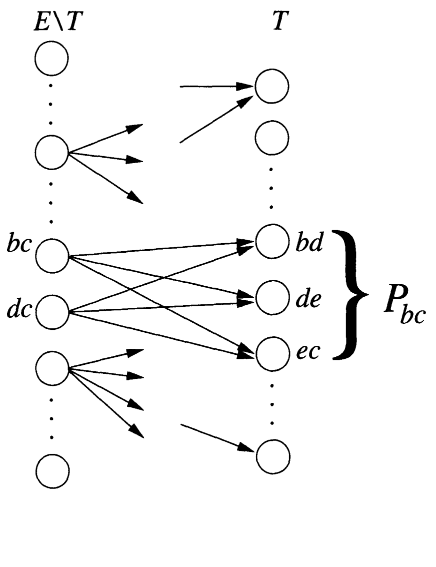
The algorithm of Hochbaum and Queyranne has the complexity of a (parametric) minimum cut on a graph associated with the formulation, $T^p(n', m', L)$, plus the complexity of finding the minima of the convex functions that form the penalty functions, $O(T^p(n', m', L) + n' \log U)$. Note that finding the minima of n weighted deviation functions or of quadratic deviation functions can be accomplished in $O(n)$. The most efficient algorithms reported here for the convex IST problem use the CCC algorithm of Hochbaum and Queyranne (2003).

1.2. Main Contributions

The formulation of IST introduced here is different from the one previously used in Sokkalingam et al. (1999) and Ahuja and Orlin (2000). That formulation is a dual of the assignment problem. We believe that our formulation is particularly amenable to efficient algorithms in that it makes the role of the minimum cut structure of the problem transparent. Our results exploit the special structure of the graph associated with the formulation. That graph, called the *path graph,* is bipartite with $|T|$ and $|E\backslash T|$ nodes in the two parts of the bipartition, respectively, and an edge between each node representing an out-of-tree edge j and the nodes representing in-tree edges on the induced path $P_j(T)$ (see Figure 1 for illustration). We use the fact that the graph is bipartite and then reformulate the problem with fewer constraints to obtain further improvements. Our major contributions here are:

1. For any convex deviation function an algorithm of complexity $O(mn \log n \log(n/\log n) + n \log C)$. This algorithm improves on the run time of the algorithm of Ahuja

Figure 1. The path graph for the tree in Figure 2.



et al. (1999a), $O(m^2 n \log(m/n) \log nC)$. The algorithm here is, furthermore, a uniform algorithm regardless of the type of objective function.

2. For the weighted absolute deviation function or for quadratic deviation function (as well as other easily minimized convex functions), the run time of the algorithm is $O(mn \log n \log(n/\log n))$. It matches the bound of Ahuja and Orlin (2000) for unit weight deviations (L_1 norm) of $O(n^2 \log n)$. A second algorithm we propose for the L_1 norm has complexity of $O(n^{1.5} \log n \log C)$, which is the best run time when $C < 2^{\sqrt{n}}$.

3. For the min max deviation case (L_∞ norm), the complexity is $O(m \log n)$. This improves on the $O(n^2)$ bound of Sokkalingam et al. (1999) for nondense graphs.

4. We introduce a new algorithm for bipartite matching in path graphs of complexity $O(n^{1.5} \log n)$. This improves by a factor of $\sqrt{n}$ on the result of Ahuja and Orlin (2000) for node-weighted bipartite matching on path graphs when the node weights are all equal.

In Table 1 we show the complexity of algorithms solving the problem here and in prior papers. Run times preceded by a * appeared previously in the literature, whereas the others are developed here. The best run times are indicated with the symbol $\rhd$. The results on each line of the table indicate the reference to the algorithm adapted. On the right column the algorithms are used with the reformulation. All run times reported are for solving the problem in integer variables. Solving the problem in continuous variables with accuracy ϵ is equivalent to solving in integers on a grid of size ϵ. The only required adjustment is to replace C, in the run times that include it, by C/ϵ. (This issue is discussed in detail in §7.)

The structure of the paper is as follows. We first describe the closure graph associated with the formulation of IST. We then describe the reformulation that has only $O(m \log n)$ constraints and $O(m)$ variables and establish a couple of important properties bounding the maximum distance label in the graph associated with the reformulation and characterizing the paths in the residual graph. These properties are summarized in lemmas that are used in the implementation described in later sections. We then describe the algorithms using the cut-based algorithm of Ahuja et al. (1999b) and the related subroutine solving minimum cut on "almost bipartite" graphs. We detail the necessary adaptations and properties needed to improve the run time complexity for the general convex problem and for the L_1 norm. We show how to adapt a variant of Dinic's (1970) algorithm for the L_1 norm and consequently generate an algorithm that solves the maximum bipartite matching in path graphs in $O(n^{1.5} \log n)$. Next, the algorithm of Hochbaum and Queyranne (2003) for CCC is sketched with the description of the adaptations to the special structure of the IST path graph. We introduce in §6.2 a new algorithm for the L_∞ norm, based on the reformulation. Finally, we comment about the problem in continuous variables and about convex quadratic penalty functions.

2. PRELIMINARIES

The inverse spanning-tree problem is defined for an undirected graph $G^T = (N', E)$ with edge weights c_e for all

Table 1. Complexity of algorithms for the inverse spanning-tree problem.

Penalty Function	Best Results and Their Adaptations	Reformulation Improvements
Convex	$^*O(mn \log n \log nC) = O(m^2 n \log n \log nC)$ AHOa	
	$O(\log C \cdot T_B(n, m)) = O(\log C \cdot mn^2)^1$ AHOb	$O(\log C \cdot mn \log n \log(n/\log n))^3$
	$O(T_B^p(n, m) + n \log C) = O(mn^2 + n \log C)^2$ HQ	$\rhd O(mn \log n \log(n/\log n) + n \log C)^3$
Weighted absolute deviation	$^*O(mn^2 \log nC)$ SAO	
	$O(mn^2 \log C)^1$ AHOb	$O(mn \log n \log(n/\log n) \log C)$
	$O(mn^2)$ HQ	$\rhd O(mn \log n \log(n/\log n))$
Absolute deviation	$^*O(n^3)$ SAO	$\overset{*}{\rhd} O(n^2 \log n)$ AO
		$O(n^2 \log n)$ HQ
	$O(n^{2.5} \log C)^4$ AHOb	$\rhd O(n^{1.5} \log n \log C)$
Maximum absolute deviation	$\overset{*}{\rhd} O(n^2)$ SAO	$\rhd O(m \log n)$

Notes. $T_B(n_1, n_2)$ denotes the complexity of solving the minimum s, t-cut problem on a bipartite network with n_1 and n_2 nodes on each side of the bipartition. $T_B^p(n_1, n_2)$ denotes the complexity of solving the parametric minimum s, t-cut problem on a bipartite network with n_1 and n_2 nodes on each side of the bipartition. The algorithm of Hochbaum and Queyranne (2003) is denoted by HQ, the algorithm of Sokkalingam et al. (1999) is denoted by SAO. The algorithm of Ahuja and Orlin (2000) is denoted by AO.

[1]Although the associated graph is not bipartite, it is "almost bipartite". As shown in §5, a slight adaptation of bipartite network-flow algorithms such as that of Gusfield et al. (1987) or the FIFO push-relabel of Goldberg and Tarjan (1988) give the stated run time.

[2]This is based on the parametric implementation of FIFO push-relabel in bipartite graphs (Goldberg and Tarjan 1988, Gallo et al. 1989).

[3]Using Goldberg and Tarjan's algorithm with dynamic trees for a graph with maximum label value n and $m \log n$ arcs (Goldberg and Tarjan 1988).

[4]Using Dinic's algorithm in "almost" simple "almost" bipartite networks with maximum flow value $\leqslant n$ (§5).

$e \in E$ and a spanning tree T. The graph G^T with a spanning tree T has an associated digraph, G, corresponding to the CCC problem. The graph G has one node for each variable and one arc (i, j) for each constraint $x_i \geqslant x_j$. This directed graph $G = (V, A)$ is referred to as the *closure graph.* The number of nodes in this graph is denoted by $|V| = N$ and the number of edges by $|A| = M$.

Let (B, D) be the collection of arcs with tails at B and heads at D. The corresponding sum of capacities of these arcs is denoted by $C(B, D)$, $C(B, D) = \sum_{i\in B, j\in D} c_{ij}$, where c_{ij} is the capacity of arc (i, j).

An s, t-graph is a directed graph that contains a source node s and a sink node t. The minimum cut (or s, t-cut) problem is to partition the graph into two subsets S and $\overline{S}$ so that $s \in S$, $t \in \overline{S}$ and $C(S, \overline{S})$ is minimum.

In a node-weighted graph with the weight of node j equal to w_j, we let the weight of a subset of nodes $D \subseteq V$ be denoted by $w(D) = \sum_{j\in D} w_j$.

A path from v_1 to v_2 in $G = (V, A)$ is an ordered sequence $[v_1, u_{i_1}, u_{i_2}, \ldots, u_{i_k}, v_2]$ where (v_1, u_{i_1}), (u_{i_k}, v_2), $(u_{i_{j-1}}, u_{i_j})$ are all arcs in A for $j = 2, \ldots, k$. The nodes $u_{i_1}, u_{i_2}, \ldots, u_{i_k}$ are said to be the *internal* nodes of the path. An s, t-path is a path from s to t in an s, t-graph.

The concept of *residual graph* is essential for maximum flow and minimum cut algorithms. For a graph $G = (V, A)$ with arc capacities constraints $l_{ij} \leqslant f_{ij} \leqslant u_{ij}$ and a feasible flow f, the residual graph with respect to f is $G^r = (V, A^r)$ with $(i, j) \in A^r$ if either

$$f_{ij} < u_{ij}, \quad (i, j) \in A, \text{ or}$$
$$f_{ji} > l_{ji}, \quad (j, i) \in A.$$

An augmenting path is an s, t-path in the residual graph. Given a graph with feasible flow, f, that flow is maximum if and only if the residual graph contains no augmenting path. If there is an augmenting path, then the flow can be incremented by the bottleneck residual capacity of the augmenting path.

The *residual flow* is the maximum flow in a residual graph.

We define the *throughput* of a node to be the maximum amount of flow that can be sent through that node. The throughput is bounded by the minimum between the total capacity of the incoming and outgoing arcs; i.e., $\min\{\sum_{(i,k)\in A} c_{ik}, \sum_{(k,j)\in A} c_{kj}\}$.

The maximum length of a shortest path in the residual graph is significant in determining the complexity of a number of algorithms for minimum cut (and maximum flow). In the residual graph an arc can appear in the opposite direction to the direction it appears in the original network. The *maximum label* of a node in the residual graph is the maximum length of a simple shortest path over all possible orientations of the arcs in the graph.

A bipartite graph on the bipartition $N_1 \cup N_2$ is denoted by $G = (N_1 \cup N_2, A)$ with $|N_1| = n_1$ and $|N_2| = n_2$. A bipartite graph is said to be unbalanced if $n_2 \gg n_1$ or $n_1 \gg n_2$. A network is bipartite if it is a bipartite s, t-graph with a source s and a sink t, where s is adjacent only to nodes in N_1 and t is adjacent only to nodes in N_2.

A set of nodes $D \subseteq V$ in a directed graph $G = (V, A)$ is said to be *closed* if all predecessor nodes of D are also included in D; i.e., if $j \in D$ and $(i, j) \in A$, then $i \in D$. Equivalently, D is said to be closed if there are no incoming arcs into D.

3. THE CLOSURE GRAPH ASSOCIATED WITH A FORMULATION OF IST

Consider the minimum closure problem, a special case of CCC when the range of the variables is restricted to binary variables. The minimum closure problem is solved by finding a minimum s, t-cut problem on a certain graph which will be called the *associated graph* of the CCC problem. Because IST is a CCC problem, the graph that defines its binary version minimum closure problem is its associated graph.

(Minimum Closure)

$$\begin{aligned} &\text{Min} \quad \sum_{j\in V} w_j \cdot x_j \\ &\text{subject to} \quad x_i - x_j \geqslant 0 \quad \forall (i, j) \in A, \\ &\qquad\qquad 0 \leqslant x_j \leqslant 1 \quad \text{integer } j \in V. \end{aligned}$$

The minimum closure problem is to find a closed set D so that $w(D)$ is of minimum weight.

Picard (1976) demonstrated that the minimum closure problem can be solved using a minimum cut procedure on the associated graph. The sink set of a minimum cut is a minimum closed set. The associated graph is an s, t-graph that has a node j for each variable x_j and an arc of infinite capacity (i, j) for each constraint $x_i \geqslant x_j$. If the weight of the variable w_j is positive, then node j has an arc from the source into it with capacity w_j. If the node has weight w_j, which is negative, then there is an arc from j to t with capacity $-w_j$. Let V^+ be the set of nodes with positive weights, and V^- the set of nodes with negative weights.

To see that the sink set of a minimum cut is a minimum closed set, consider any finite s, t-cut in the graph that partitions the set of nodes to two subsets $\{s\} \cup S$ and $\{t\} \cup \overline{S}$. It is easy to see that $\overline{S}$ is a closed set because there are no infinite capacity arcs from S to $\overline{S}$ (else the cut is not finite).

Let a cut $(\{s\} \cup S, \overline{S} \cup \{t\})$ be finite:

$$\begin{aligned} &\min_{\overline{S}\subseteq V}[C(\{s\} \cup S, \overline{S} \cup \{t\})] \\ &\quad = \min_{\overline{S}\subseteq V} \sum_{j\in\overline{S}\cap V^+} w_j + \sum_{j\in S\cap V^-} (-w_j) \\ &\quad = \min_{\overline{S}\subseteq V} \sum_{j\in\overline{S}\cap V^+} w_j - \left(\sum_{i\in V^-} w_i - \sum_{i\in\overline{S}\cap V^-} w_i\right) \\ &\quad = \min_{\overline{S}\subseteq V} \sum_{j\in\overline{S}} w_j - w(V^-). \end{aligned}$$

In the last expression the term $w(V^-)$ is a constant. Thus, the closed set $\overline{S}$ of minimum weight is also the sink set of a minimum cut and vice versa—the sink set of a minimum cut (which has to be finite) also minimizes the weight of the closure.

Similarly, we map the IST's graph G^T to the *associated graph* G: The set of variables and nodes of the digraph G, V, corresponds to the set of edges E of G^T. For each $i \in E\backslash T$ and $j \in P_i(T)$ there is an arc $(i, j) \in A$. The graph $G = (V, A)$ is bipartite with $V = N_1 \cup N_2$ where $N_1 = E\backslash T$ and $N_2 = T$. Note that such a bipartite graph is typically unbalanced for graphs that are not sparse, with $n-1 = |N_2| < |N_1| = m-n+1$. We adapt the terminology of Sokkalingam et al. (1999), referring to such graphs as *path graphs.* In Figure 1 we show part of the path graph associated with the tree given in Figure 2 showing the path induced by the out-of-tree edge (b, c), P_{bc}.

Even though the graph G is bipartite, it is not obvious that the associated s, t-graph on which the minimum cut problem is solved is also bipartite. In order for that to happen, all nodes in N_1 must have weights that are nonnegative (nonpositive) and all nodes in N_2 must have weights that are nonpositive (nonnegative). (The sign of each set does not matter as long as these are opposites.) We show next that this is indeed the case for IST.

So far we have not commented on how the weights w_j derive from the convex objective. While these depend on the algorithm and whether the problem is on integer or continuous variable, the weights in every case are some form of derivatives of $f_j()$ or subgradients—finite differences of the form $[f_j(x+h) - f_j(x)]/h$. Recall that the functions $f_j()$ are convex with a minimum value 0 at c_j for all $j \in E$ and, thus, monotone nonincreasing for $x \leqslant c_j$ and monotone nondecreasing for $x \geqslant c_j$. We next show that the values of x_i for $i \in T$ can be restricted to $x_i \leqslant c_i$ and the values of x_j for $j \in E\backslash T$ can be restricted to $x_j \geqslant c_j$.

Lemma 3.1. *There exists an optimal solution* $\mathbf{x}^*$ *so that for each* $i \in T$, $x_i^* \leqslant c_i$ *and for each* $j \in E\backslash T$, $x_j^* \geqslant c_j$.

Proof. Suppose that $x_i^* > c_i$ for some $i \in T$. The function $f_i(x_i - c_i)$ is monotone nondecreasing for $x_i^* > c_i$, thus setting $x_i^* = c_i$ can only decrease the value of the optimal solution and retain the feasibility of all constraints involving x_i which are of the form $x_p \geqslant x_i$.

Similarly, if $x_j^* < c_j$ for some $j \in E\backslash T$, the increase of the value of x_j^* to c_j retains feasibility of all constraints involving x_j and can only reduce the value of the function $f_j(x_j)$. □

As a consequence of the lemma, the formulation can either incorporate the modified upper and lower bounds on the variables or, alternatively, redefine the functions $f_i()$ for $i \in T$:

$$f_i(x_i) = \begin{cases} f_i(x_i) & \text{if } x_i \leqslant c_i, \\ 0 & \text{if } x_i \geqslant c_i. \end{cases}$$

Thus, the subgradient or derivative of $f_i()$, if it exists, is nonpositive. Similarly, for $j \in E\backslash T$:

$$f_j(x_j) = \begin{cases} f_j(x_j) & \text{if } x_j \geqslant c_j, \\ 0 & \text{if } x_j \leqslant c_j. \end{cases}$$

Thus, the subgradient or derivative of $f_j()$, if it exists, is nonnegative.

In these ranges the finite differences of the convex functions are nonnegative and nonpositive, respectively. Therefore, $w_j \geqslant 0$ for $j \in N_1$ and $w_i \leqslant 0$ for $i \in N_2$ and the associated s, t-graph for the IST problem is bipartite.

4. THE REFORMULATION AND CONSTRUCTION OF THE AUXILIARY GRAPH

The straightforward formulation of IST has

$$M = \sum_{j \in E\backslash T} |P_j(T)|$$

constraints. Because path $P_j(T)$ can be $O(n)$ in length, the number of arcs M in the associated bipartite graph can be as large as $O(mn)$. We present here an alternative formulation with $O(m \log n)$ constraints and maximum label in the associated graph $O(n)$. Note that the reformulation is to be used only if $M > m \log n$.

As motivation for the construction used next, consider the case where the IST graph is a complete bipartite graph (also known as biclique) with all arcs present. Then, although many out-of-tree edges correspond to paths of length $O(n)$, the formulation can be drastically reduced in size.

All out-of-tree edges $x^{[2]}$ that induce paths of length 2 are set as before, with two inequalities. However, now each path of length 3, $x^{[3]}$, can be represented with two inequalities only, one constraint that sets it to be larger than or equal to the weight of the path of length 2 it contains, $x^{[3]} \geqslant x^{[2]}$, and the other for the third edge on the induced path. For any path of length k, the appropriate inequalities are captured with only two inequalities, one with respect to a path of length $k-1$ contained in the k-path and the second an inequality with the remaining single edge.

Note that in this example the number of variables/edges has not changed, but the number of inequalities is only twice the number of edges. The corresponding graph thus has m nodes and $O(m)$ edges. However, the associated graph is no longer bipartite—there are arcs between the nodes of $E\backslash T$.

Suppose we now have an associated graph which is not complete. Then, we could add all the "missing" edges as auxiliary variables to reduce the number of inequalities in the formulation. This approach entails adding edges that were not originally in the graph, for a total of $O(n^2)$ edges/variables and twice as many $O(n^2)$ inequalities. We call the edges added *auxiliary edges* and the resulting closure graph the *auxiliary graph.* The auxiliary variables have no costs and the associated penalty function is zero. Although the number of arcs (inequalities) in the auxiliary graph is relatively small, the number of nodes can be quite significantly larger than in the original associated graph.

For relatively sparse graphs this approach is not effective. For such graphs we show another construction, introduced by Ahuja and Orlin (2000), that adds only $O(n)$ auxiliary edges and reduces the number of inequalities to no more than $O(\log n)$ per variable/edge for a total of $O(m)$ variables and $O(m \log n)$ constraints. A crucial property of the associated graph is that the maximum label is still only $O(n)$ even though the graph is no longer bipartite.

Ahuja and Orlin showed how to solve the inverse spanning tree with the absolute deviation function, $\sum_{j\in E}|x_j - c_j|$. They reduced the problem to a form of a node-weighted bipartite matching problem and proposed a construction that modifies the graph and reduces the number of arcs. We found that their construction is useful in the general case of IST and used it here to improve the run time of the algorithms for several cases. For the sake of completeness we sketch the construction that is using the dynamic tree data structure of Sleator and Tarjan (1983). Our construction is not identical to that of Ahuja and Orlin (2000)—it is a minor variant that has the property proved in Lemma 4.3 that is not shared by their construction.

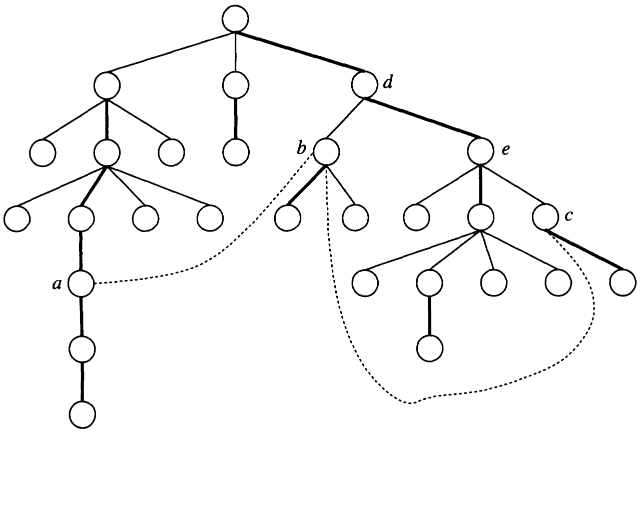
We use the common terminology for a rooted tree where an ancestor of a node is any node on the path between the node and the root, and the descendant of a node i is any node for which i is an ancestor. A parent of a node is an immediate (adjacent) ancestor and a child of a node is an immediate descendent.

We root the tree T at an arbitrarily selected node, say node 1. We then partition the edges of the tree into heavy and light edges, where a heavy edge leads from a node to its descendant that carries at least half of the nodes in the induced subtree, and light edges are all the others. A consecutive sequence of heavy edges encountered on a path from a node to the root is called a *heavy path.* A rooted tree depicting heavy edges in thick lines is given in Figure 2. The out-of-tree edges are described with dotted lines.

Lemma 4.1 (Sleator and Tarjan 1983). *On any path from a node to the root there are no more than* $\log n$ *light edges and no more than* $\log n$ *heavy paths.*

Proof. Each time that the path from a node to the root traverses a light edge, then the number of descendants of the node reached is doubled. Therefore, there cannot be more than $\log n$ light edges on the path. Because any pair of consecutive heavy paths is separated by at least one light edge, then the number of heavy paths traversed is at most $\log n$ as well. □

Figure 2. The heavy and light edges in the tree.



Consider the path induced by an out-of-tree edge $[a, b]$. The path induced is formed of two sections: The two sections of the paths from a and from b to their lowest common ancestor. (One of these two sections can be empty if a is an ancestor of b or vice versa.) The common ancestor that separates the induced path into two sections is called the *apex* of the induced path. As an example, consider the induced paths $P_{[a,b]}$ and $P_{[b,c]}$ in Figure 2 which have the root of the tree and node d as apex, respectively.

Define a *root of a heavy path* to be the node on the heavy path closest to the root of the tree.

Consider the section of the induced path $P_{[a,b]}$ (we omit (T) from the notation as T will remain fixed throughout this discussion) from a to the apex of the path. This section traverses light edges and heavy paths where each heavy path included is traversed from some internal point on the heavy path (which could be the endpoint of the heavy path) to the root of the heavy path. There is at most one heavy path, the one adjacent to the apex, that is not traversed all the way to its root, but rather between two internal points of the path. For example, in Figure 2, $P_{[b,c]}$ traverses the section $[e, d]$ of a heavy path where both e and b are internal nodes to this heavy path.

We now present the set of inequalities that will ensure that $x_{[a,b]}$ assumes a value at least as large as all edges along $P_{[a,b]}$. Each new auxiliary variable introduced will be indicated with the superscript corresponding to the section of the path it represents (that is, it is greater or equal to all edge variables for the edges on the path).

Let a generic heavy path be $[v_1, \dots, v_r]$ with root v_r and length b. Let $x^{[k,r]}$ be an auxiliary variable associated with the section of the path $[v_k, \dots, v_r]$. We partition the edges of $[v_1, \dots, v_r]$ into sections of length 2 each (starting with r). Each section of length 2 has an auxiliary variable defined by two inequalities:

$$x^{[k,k+1,k+2]} \geqslant x_{[k,k+1]},$$
$$x^{[k,k+1,k+2]} \geqslant x_{[k+1,k+2]}.$$

Then, recursively partition the path of length p into sections of length 2^q for $q = 2, \dots, \log p$:

$$x^{[v_1,\dots,v_{2^q}]} \geqslant x^{[v_1,\dots,v_{2^{q-1}}]},$$
$$x^{[v_1,\dots,v_{2^q}]} \geqslant x^{[v_{2^{q-1}+1},\dots,v_{2^q}]}.$$

For a heavy path of length p we add at most $p/2$ auxiliary variables corresponding to sections of length 2; $p/4$ auxiliary variables corresponding to sections of length 4; and $p/2^q$ auxiliary variables corresponding to sections of length 2^q, for $q = 2, \dots, \log p$. The total number of variables added per heavy path is no more than $\sum_{q=1}^{\log p}(p/2^q) < p$. Thus, the total number of auxiliary variables added in this fashion is no more than n.

All heavy paths, except possibly the one adjacent to the apex, are traversed from some internal point to the root r. To represent these sections of heavy paths compactly, we define a collection of $p - 1$ auxiliary variables for each heavy path of length p, $x^{[k,r]}$ for $k = v_1, \ldots, v_p$ (there are p edges in the path and thus $p+1$ nodes including the node r) as follows: We take the longest power-of-2-long section contained in $[k, r]$, s_1, and then the longest section contained in the remainder of the path, s_2, and so on. It is easy to see that there are altogether no more than $O(\log|p-k|)$ sections required to partition and cover the path. For each of these sections s_p we add the inequality

$$x^{[k,r]} \geqslant x^{s_p}.$$

Each variable $x^{[k,r]}$ thus requires up to $\log(p-k)$ such inequalities that correspond to the binary representation of its length. The total number of variables $x^{[k,r]}$ is no more than n for all the heavy paths in the graph.

Now, with the possible exception of the heavy path adjacent to the apex, we represent the set of inequalities for $P_{[a,b]}$ as inequalities for each light edge on the path e,

$$x_{[a,b]} \geqslant x_e,$$

and for each heavy path on the induced path traversed from an internal point k to its root r as

$$x_{[a,b]} \geqslant x^{[k,r]}.$$

Altogether, this requires $O(\log n)$ inequalities.

We finally address the representation of the heavy path $[v_1, \ldots, v_{k_1}, \ldots, v_{k_2}, \ldots, v_r]$ for which the section that is adjacent to the apex of $P_{a,b}$ is traversed between two internal points, $[v_{k_1}, \ldots, v_{k_2}]$. Here, we do not introduce a new auxiliary variable corresponding to these sections of heavy paths, as there could be as many as $O(m)$ such sections, one for each induced path. Instead, we take the longest power-of-2-long section contained in this path, s_1, and then the longest section contained in the remainder of the path, s_2, and so on. It is easy to see that there are altogether no more than $O(\log|k_2-k_1|)$ sections required to partition and cover the path. For each of these sections s_p we write the inequality

$$x_{[a,b]} \geqslant x^{s_p}.$$

Note that there is only one internal nodes section for each induced path, so these inequalities add only $O(\log n)$ inequalities to the representation of each induced path.

Because all heavy paths have no more than n edges jointly, the number of auxiliary variables representing disjoint sections of length 2 is at most $n/2$ and the number of auxiliary variables representing disjoint sections of length 2^q is at most $n/2^q$. The total number of auxiliary variables corresponding to these sections for all paths is at most n.

To summarize, to enforce that the value of $x_{[a,b]}$ is greater or equal to the weight of all edges along the induced path, the reformulation includes:

$O(\log n)$ inequalities for heavy paths traversed from an internal point to their root,

$O(\log n)$ inequalities for light edges, and

$O(\log n)$ inequalities for at most one heavy path adjacent to the apex that is not traversed all the way to its root.

We thus have a total of $O(\log n)$ inequalities for each out-of-tree edge, $O(\log n)$ inequalities for each heavy path section $[k, r]$, and $O(1)$ inequalities for all other auxiliary variables. The total number of auxiliary variables is $O(n)$ and the total number of inequalities is $O(m \log n)$. This is instead of the $O(\sum_{i\in E\setminus T} |P_i(T)|)$ inequalities in the original formulation of IST.

Consider the associated graph with the new formulation. Let the set of auxiliary variables be denoted by N_3. N_1 and N_2 are defined as before and are both independent sets with $m-n+1$ and $n-1$ nodes, respectively. Let the auxiliary closure graph be $(N_1 \cup N_2 \cup N_3, \tilde{A})$ with n_1 nodes adjacent to the source, n_2 nodes adjacent to the sink, and n_3 transshipment nodes adjacent to neither source nor sink. Next, we show that maximum length of a simple path from source to sink is $O(n)$ at most.

Lemma 4.2. *If the set of nodes N_1 is independent in the graph $(N_1 \cup N_2 \cup N_3, \tilde{A})$, then any path to sink is of length $O(n_2 + n_3)$ at most.*

Proof. Any path visits nodes of $N_2 \cup N_3$ at most once. Between two consecutive visits of nodes of N_1, a path must visit a node of $N_2 \cup N_3$. Therefore, the length of a path cannot exceed $2(n_2 + n_3) + 1$. □

Consequently, the maximum distance label in the reformulation graph is $O(n)$ as required.

Lemma 4.3. *In a residual graph defined for a feasible flow on the graph $(N_1 \cup N_2 \cup N_3, \tilde{A})$, the shortest path from a node of N_1 to N_2 or from N_2 to N_1 is of length $O(\log n)$ at most.*

Proof. Every node of N_1 is either adjacent to a node of N_2 (if the latter represents a light edge) or to some auxiliary node representing a heavy path section. All heavy path sections are represented by "binary representation" of depth not exceeding $O(\log n)$. Thus, the number of nodes in the path between a node of N_1 and N_2 is $O(\log n)$. □

5. ADAPTING THE CUT-BASED ALGORITHM FOR DMCNF

The cut-based algorithm of Ahuja et al. (1999b) for DMCNF is based on the proximity-scaling framework for convex separable optimization of Hochbaum and Shanthikumar (1990). Briefly, this entails the piecewise linear approximation of the convex objective with "small" number of breakpoints on a grid of size s that is determined by the relevant *proximity theorem*. The algorithm repeats solving a scaled problem in binary variables for progressively smaller grid size. The problem solved at each iteration is a minimum closure problem, as shown next.

Let the variables be contained in the range $[l, u]$ and $C = u - l$. The scaled problem for a scaling unit s and $k = \lceil C/s \rceil$ is defined as follows: Each variable x_i is replaced by k binary variables $x_i^{(p)}$, $p = 1, \ldots, k$ such that $x_i = \sum_{p=1}^{k} x_i^{(p)}$.

Each binary variable is assigned a weight that is its incremental contribution to the objective function according to the following recursive procedure:

$$w_j^{(0)} = f_j(l),$$
$$w_j^{(p)} = f_j(l+ps) - f_j(l+(p-1)s) \quad \text{for } p = 1, \dots, k.$$

The following (s-IST) is a piecewise linear approximation of the problem on a grid of unit size s. The objective differs from that of IST by the constant $\sum_{j\in E} w_j^{(0)}$:

$$\begin{aligned}
(s\text{-IST}) \quad \text{Min} \quad & \sum_{j\in E}\sum_{p=1}^{k} w_j^{(p)} x_j^{(p)} \\
\text{subject to} \quad & x_i^{(p)} \geqslant x_j^{(p)} \quad \text{for } i \in E\backslash T,\ j \in P_i(T) \\
& \quad \text{and } p = 1, \dots, k, \\
& x_j^{(p)} \leqslant x_j^{(p-1)} \quad \text{for } j \in E \text{ and } p = 1, \dots, k, \\
& x_j^{(p)} \text{ binary} \quad \text{for } j \in E \text{ and } p = 1, \dots, k.
\end{aligned}$$

Note that s-IST is a minimum closure problem and denote its optimal solution by $x^{(s)}$. The proximity theorem for the distance between an optimal solution x^* to IST and the scaled problem's optimal solution $x^{(s)}$ is at most one scaled unit. This is the "strong proximity theorem" of Ahuja et al. (1999b) for homogeneous constraints,

$$\|x^* - x^{(s)}\|_\infty \leqslant s.$$

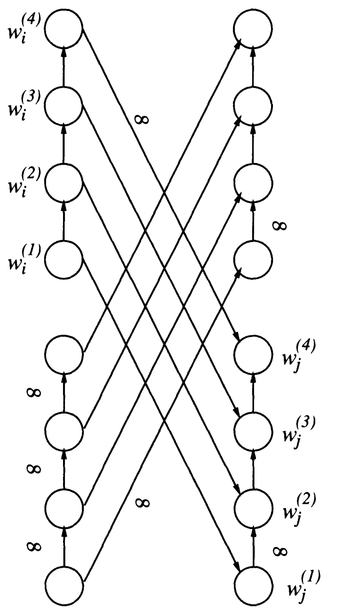
The proximity-scaling procedure starts with an interval of length $4s$ and $k = 4$. In each iteration the optimal solution of s-IST, $x^{(s)}$, serves as a center to a contracted interval containing the optimal solution $[x^{(s)} - s, x^{(s)} + s]$. The length of this interval is $2s$, half the length of the interval in which the variables were restricted in the problem s-IST. Using a new scaling unit of $s/2$, the problem $s/2$-IST has again no more than 4 binary variables per variable x_i. The scaled problem is repeatedly solved, each time for half as large a value of s, until $s \leqslant 1$ (in which case it is set to 1). The solution to 1-IST is the optimal solution to the problem. Thus, the proximity-scaling procedure calls for $O(\log C)$ solutions of a scaled problem s-IST.

The corresponding closure graph construction for s-IST has a 4-node sequence for each node in the associated graph, each representing a value in the range 1, 2, 3, 4. There is an arc of infinite capacity between a node representing value $v-1$ to a node representing value v, as illustrated in Figure 3. We call such graphs that are derived by replacing nodes in a bipartite graph by subgraphs of bounded size, "almost bipartite" graphs.

The substitution of each node in the associated bipartite graph by a "chain" of 4 nodes implies that the structure of an associated graph as a bipartite graph is not preserved. It is thus not obvious that algorithms that are particularly efficient for bipartite graphs (Ahuja et al. 1994) would still be compatible with the almost bipartite associated graph.

The efficiency of most algorithms for bipartite graphs is derived from the length of the distance of a path between any node and the sink. This distance is a bound on the node labels in the push-relabel algorithm (Goldberg and Tarjan 1988) and the pseudoflow algorithm (Hochbaum 1997). In Dinic's (1970) algorithm, the number of stages is bounded, where at each stage there are augmentations along shortest paths of prescribed length in the residual graph. We show that in "almost bipartite" graphs the distance label is still no more than twice the smallest side of the bipartition.

Figure 3. The "almost bipartite" graph.



Let each node in a bipartite graph (N_1, N_2, A) be substituted by an arbitrary subgraph with the number of nodes bounded by q. Denote the resulting "almost bipartite" graph by $(\tilde{N}_1, \tilde{N}_2, \tilde{A})$. Other than arcs within each subgraph, there are only arcs between nodes that are in subgraphs of $\tilde{N}_1$ and nodes that are in subgraphs of $\tilde{N}_2$. In the case of the graph associated with s-IST, the subgraphs are chains of 4 nodes and $q = 4$.

LEMMA 5.1. *The longest-path distance between source and sink in a residual graph of* $(\tilde{N}_1, \tilde{N}_2, \tilde{A})$ *for any flow* f *is bounded by* $2q\min\{n_1, n_2\}$.

PROOF. Let $n_2 \leqslant n_1$. Every residual path between source and sink alternates between subgraphs of N_1 and subgraphs of N_2. Between any two visits to $\tilde{N}_1$ there is at least one node of $\tilde{N}_2$. Each visit to $\tilde{N}_1$ includes a sequence of q nodes at most. Therefore, a path cannot include more than qn_2 visits of subgraphs of N_1 and of subgraphs of N_2. The length of any path to the sink t is therefore bounded by $2qn_2$. □

We conclude that the running time of solving each scaled problem is $O(n_2^2 n_1)$, the same as the running time for a bipartite graph as in the algorithms of Gusfield et al. (1987) or as the FIFO implementation of the push-relabel algorithm (Goldberg and Tarjan 1988). The total run time required to solve IST with the proximity-scaling algorithm is thus $O(n^2 m \log C)$.

Now consider solving the minimum cut problem s-IST on the reformulation graph. In that graph, the maximum distance label is $O(n)$, as proved in Lemma 4.2, and the number of arcs is $O(m \log n)$. In this case it is possible to implement the dynamic trees push-relabel algorithm in run time $O(mn \log n \log(n/\log n))$. This leads to a total run time of $O(mn \log n \log(n/\log n) \log C)$.

The running times of these algorithms for the convex IST problem are always dominated by the respective ones generated from adapting the algorithm for CCC of Hochbaum and Queyranne (2003). For the L_1 norm, however, the cut-based algorithm using an implementation of Dinic's (1970) algorithm yields an algorithm that is fastest for $C < 2^{\sqrt{n}}$, as we show next.

5.1. The Absolute Deviation Objective Function

Let the "almost bipartite" graph associated with s-IST be $(\widetilde{N}_1, \widetilde{N}_2, \widetilde{A})$. For the objective function of absolute deviations, $\sum_{e\in E} |x_e - c_e|$, the differences $w_j^{(p)}$ are all one scaling unit s and $-s$ for $j \in N_1$ and $j \in N_2$, respectively. Thus, the capacities of all arcs adjacent to source and sink in the associated network are 1.

For convenience, we prefer to work with the "reverse graph" (V_1, V_2, A^R) which is derived from the graph $(\widetilde{N}_1, \widetilde{N}_2, \widetilde{A})$ by setting $V_1 = \widetilde{N}_2$, $V_2 = \widetilde{N}_1$ reversing the directions of all arcs set with same capacities and reversing the roles of the source and sink. The maximum flow and minimum cut in the reverse graph are equal to the maximum flow and minimum cut in the original graph. In this reverse graph the set adjacent to source is of size $4n$ and each node has an incoming arc from the source of capacity 1.

Dinic's algorithm (1970) can be implemented particularly efficiently for simple networks. These are networks where each node's throughput is 1. That is, each node has either a single incoming arc of capacity 1 or a single outgoing arc of capacity 1. For simple networks, Even and Tarjan (1975) showed that Dinic's algorithm can be implemented to run in time $O(\sqrt{n'}m')$, where n', m' are the number of nodes and arcs, respectively, in the simple network. Although our network is not simple we can nevertheless mimic some features of the procedure for s-IST and achieve the run time of $O(n^{2.5})$ or, with the reformulation, $O(n^{1.5} \log n)$.

Dinic's algorithm works in stages, where at stage l there is an augmentation of flow along all shortest paths of length l until the flow is blocking (meaning until there is no further augmentation along a path of length l or, equivalently, in a forward direction only). The residual graph constructed at each stage when the shortest path is of length l is a graph with l layers of nodes. It is referred to as the *l-layered network*. In general, the number of stages does not exceed the number of nodes. Here the number of stages is no more than $O(|\widetilde{N}_2|)$, or $O(|V_1|)$, because this is a bound on the maximum distance to the sink, as was shown in Lemma 5.1.

We replace the residual graph G^r by a construction of the V_1-residual graph, $G^r_{V_1}$. This is a residual graph that involves, in addition to source and sink, only nodes of V_1, and it has the property that the maximum flow in the V_1-residual graph $G^r_{V_1}$ is of the same value and corresponds to a maximum residual flow in G^r.

The V_1-residual graph is defined as follows. Given the graph (V_1, V_2, A), a feasible flow f, and the residual graph G^r, the graph $G^r_{V_1}$ is obtained from G^r by contracting the vertices of V_2. This is equivalent to adding arcs between any pair of nodes u_1, u_2 in $\{s, t, V_1\}$ that have a directed path between them in G^r, $[u_1, u_{i_1}, \ldots, u_{i_k}, u_2]$, where all the internal nodes of the path $u_{i_1}, \ldots, u_{i_k}$ are in V_2. Note that in our case $k \leqslant 4$ as any path alternates between nodes of a 4-chain in V_1 and nodes of a 4-chain in V_2. It follows that any path from source to sink in $G^r_{V_1}$ is at least 1/5 of the length of a corresponding path in G^r.

Lemma 5.2. *In an l-layered network of the residual graph $G^r_{V_1}$ for the almost bipartite graph (V_1, V_2, A), the maximum residual flow is at most $O(n/l)$.*

Proof. The l-layered network includes only nodes of V_1 in l layers, $V_1^{(1)}, V_1^{(2)}, \ldots, V_1^{(l)}$. Each node in V_1 has throughput of at most 4. (More precisely, each chain of 4 nodes has throughput of, at most, 4.) The flow in this layered network is thus at most

$$4 \cdot \min_{p=1,\ldots,l} |V_1^{(p)}| \leqslant 4 \cdot \frac{|V_1|}{l} = O\left(\frac{n}{l}\right). \quad \square$$

The implementation of the algorithm works as follows. In a first phase we find the blocking flow in the l-layered network until $l \geqslant \sqrt{n}$. This requires up to $\sqrt{n}$ consecutive stages. When done running these stages, the remaining residual flow is no more than $O(\sqrt{n})$, as proved in the lemma above. We then apply a second phase in which the maximum flow is found using an augmenting-paths algorithm.

Consider the complexity of a single stage consisting of generating the layered network and pushing the blocking augmenting flows through. First, consider what it takes to generate the residual graph $G^r_{V_1}$ and the layered network. A node of a 4-chain of V_2 can be internal to a path between two nodes of V_1 in the residual graph only if at least one of the nodes of the 4-chain has incoming flow from a node of V_1. Consequently, there could be at most $O(|V_1|)$ such nodes of V_2. Thus, the residual graph $G^r_{V_1}$ has at most $O(n^2)$ arcs that can be searched using Breadth-First-Search in $O(n^2)$ time.

As for the complexity of finding the blocking flow in each stage, we will push one unit of flow at a time. (The usual implementation is to identify the node of minimum throughput and push/pull that amount to the sink and from the source.) Because each node has throughput $\leqslant 4$ it can be processed at most 4 times before it is eliminated. Finding one flow augmentation in the layered network is $O(l)$, and whenever a node is eliminated all arcs adjacent to it are removed as well. Thus, the complexity of finding the blocking flow in a stage is $O(n^2)$ and the removal of arcs when nodes are eliminated is $O(n^2)$ as well. Thus, all the $O(\sqrt{n})$ stages are completed in $O(n^{2.5})$ steps.

In the second phase of the algorithm the remaining $O(\sqrt{n})$ units of flow are found by augmenting the flow up to $O(\sqrt{n})$ times. Finding augmenting paths is accomplished by searching the residual graph for a path from source to sink. Maintaining and updating the residual path involves $O(n)$ steps for the update of each node along the path along which the flow was augmented. The search for an augmenting path is $O(n^2)$ at most. Thus, the second phase of the algorithm also has complexity $O(n^{2.5})$, and therefore the complexity of the algorithm for finding maximum flow (and minimum cut) on the "almost bipartite" graph is $O(n^{2.5})$. The total complexity of solving the IST problem with this implementation of the maximum flow algorithm used at each scaling step is $O(n^{2.5}\log C)$.

5.1.1. The Absolute Deviation Case with the Reformulation. The reformulation of the associated graph called the *auxiliary graph* is described in §4. For our purposes here it is important to note several properties of the auxiliary graph. A set of nodes V_3 that serves as intermediary nodes between V_1 and V_2 is added to the graph (V_1, V_2, A). The number of nodes in V_3 is $O(n)$, and the maximum degree of each node in the auxiliary graph is $O(\log n)$. The maximum distance label in the auxiliary graph is $O(n)$ and the distance from nodes of V_1 to nodes of V_2 that are their neighbors in the associated graph does not exceed $O(\log n)$, and vice versa. Therefore, in the residual auxiliary graph the distance from one node of V_1 to another is at most $O(\log n)$.

We now construct a layered network where each layer has only nodes of V_1, and two consecutive layers are separated by nodes of $V_3 \cup V_2$ that are internal to some path between two nodes of V_1. As before, there are at most $O(n)$ nodes of V_2 that are internal to any path between nodes of V_1. The construction of the l-layered network for each stage thus takes $O(n\log n)$ steps, and for all $\sqrt{n}$ stages it is $O(n\sqrt{n}\log n)$.

Although some nodes have throughput larger than 1, we push flow from the first layer of V_1 nodes 1 unit at a time along a path to sink. Each node on the path gets its throughput reduced by 1 unit and each residual path along the path gets its capacity reduced by 1 unit. If a node's throughput has become zero, it is eliminated along with its adjacent arcs. The complexity of a push of a single unit is $O(l\log n)$ because two adjacent layers are separated by paths of length not exceeding $\log n$. Throughout a single stage, at most $O(n\log n)$ residual arcs are eliminated. Because l-layered networks and stages are used until $l \geqslant \sqrt{n}$ and the total flow in all stages is $O(n)$, the complexity of pushing the flow through the layered networks in the first phase is at most $O(n\sqrt{n}\log n)$.

Finally, finding each augmenting path in the residual graph and updating the residual graph is at most $O(n\log n)$. Because there are up to $O(\sqrt{n})$ units of residual flow, the total complexity is $O(n^{1.5}\log n)$. This implies in particular that finding the maximum bipartite matching in path graphs or their extension to almost bipartite path graphs can be accomplished in $O(n^{1.5}\log n)$. This is faster than the algorithm of Ahuja and Orlin (2000) for this problem, $O(n^2\log n)$, although the latter works also for node-weighted bipartite matching in path graphs.

For the IST problem with the L_1 norm, this leads to a $O(n^{1.5}\log n\log C)$ algorithm which is more efficient than other algorithms for $C < 2^{\sqrt{n}}$.

6. USING THE CONVEX COST CLOSURE ALGORITHM

The convex closure algorithm of Hochbaum and Queyranne (2003) works in two steps, where in step 1 a parametric minimum cut problem is solved and in step 2 the values of the variables are determined. The outcome of the first step is a partition of the set of variables into $V_1 \cup \cdots \cup V_p$ and the interval $[l, u]$ into a collection of p subintervals, $(a_{k-1}, a_k]$ for $k = 1, \ldots, p$, where $a_0 = l$ and $a_p = u$. The property of an optimal solution is that all variables in the same subset V_k assume an identical value which falls in the corresponding interval $(a_{k-1}, a_k]$.

In the first step, the CCC problem is reduced to its binary counterpart—the minimum closure problem. The key to this reduction is in the threshold theorem, Theorem 6.1. The threshold theorem makes use of the derivatives of the functions $f_j(\,)$ denoted by $f_j'(\,)$. If the functions are not differentiable, then the discrete equivalent of the derivative, $f_j'(\lambda) = f_j(\lambda+1) - f_j(\lambda)$ is used. This definition will be adjusted for the problem on continuous variables discussed in §7.

THEOREM 6.1 (HOCHBAUM AND QUEYRANNE 2003). *Let $w_i = f_i'(\lambda)$ be the weight assigned to node i, $i = 1, \ldots, n$ in a minimum closure problem defined on the partial order graph $G = (V, A)$. Let S^* be the minimal minimum weight closed set in this graph. Then, an optimal solution $\mathbf{x}^*$ to the convex cost closure problem satisfies $x_i^* > \lambda$ if $i \in S^*$ and $x_i^* \leqslant \lambda$ if $i \in \bar{S}^*$.*

One obvious method of using the threshold theorem for solving CCC is to perform a search by calling for the solution of the minimum closure problem for all integer values of λ in the interval (l, u). A more careful analysis shows that phase 1 can be solved more efficiently.

Let G_λ be the graph associated with the minimum closure problem with weights $w_i = f_i'(\lambda)$. Denote the source set of a minimum cut in the graph G_λ by S_λ. Consider varying the value of λ in the interval $[l, u]$. As the value of λ increases, the sink set becomes smaller and contained in the previous sink sets corresponding to smaller values of λ. Specifically, for some $\lambda \leqslant l$, $S_\lambda = \{s\}$ and for some $\lambda \geqslant u$, $S_\lambda = V \cup \{s\}$. We call each value of λ where S_λ strictly increases a *node-shifting breakpoint*. For $\lambda_1 < \cdots < \lambda_p$, the set of all node-shifting breakpoints, we get a corresponding nested collection of source sets:

$$\{s\} = S_{\lambda_1} \subset S_{\lambda_2} \subset \cdots \subset S_{\lambda_p} = \{s\} \cup V.$$

We are interested only in those parameter values where the sink set of the cut is strictly reduced in size, $\lambda_1 < \lambda_2 < \cdots < \lambda_p$. From the threshold theorem it follows that if $j \in S_k - S_{k-1}$, then the optimal solution of x_j lies in the interval $(\lambda_{k-1}, \lambda_k]$. So, to identify all these intervals it is sufficient to find all breakpoint values of λ. The breakpoints where the cut is changed can be generated by a parametric cut procedure.

Suppose we have a capacitated network where the capacities are functions of a given parameter. The source-adjacent arcs have capacities that are monotone nondecreasing with the parameter value, and the sink-adjacent arcs have capacities that are monotone nonincreasing with the parameter value. A form of a *complete* parametric analysis is to find all the breakpoints in the parameter values where the cut changes the source set. Obviously, there are at most n breakpoints. Gallo et al. (1989) showed for linear monotone functions how to find all the breakpoints in the time of a single minimum cut for the push-relabel algorithm. Hochbaum (1997) showed that a complete parametric analysis can be conducted using the pseudoflow algorithm with the complexity of a single minimum cut. (To date these are the only two types of algorithms for which the strong complexity result of accomplishing a complete parametric analysis in the complexity of a single run is valid.)

The algorithm solving CCC, summarized formally below, makes calls to a procedure called *parametric* which is a parametric minimum cut algorithm identifying all the breakpoints in the specified interval. The procedure *parametric* $(f'_j(), j = 1, \ldots, n, l, u)$ has nodes of positive weight connected to source with capacity equal to the weight and nodes of negative weight connected to the sink with capacity equal to the absolute value of the weight. The capacity of an arc (s, j) is thus $\max\{0, f'_j(\lambda)\}$ and the capacity of an arc (i, t) is $\min\{0, f'_i(\lambda)\}$. These derivatives are monotone nondecreasing in λ. Thus the source adjacent capacities as a function of λ are monotone nondecreasing and the sink adjacent capacities are monotone nonincreasing.

Procedure Convex Closure $(G, f_j, j = 1, \ldots, n, [l, u])$

Step 1. Call **parametric** $(f'_j(\ ), j = 1, \ldots, n, l, u)$. Let the output be a set of up to n breakpoints $\lambda_1, \lambda_2, \ldots, \lambda_p$ and the corresponding sets of source sets of minimum cuts $S_1 \subset S_2 \cdots \subset S_p$.

Step 2. Output the optimal solution $\mathbf{x}^*$ where for $j \in S_k - S_{k-1}$, $x^*_j = \lambda_{k-1} + 1$.

The complexities of the algorithms of Gallo et al. and Hochbaum are finding a single minimum cut plus an additional run time when the parametric functions are general monotone. Although this is not discussed explicitly in Gallo et al. (the description there is only for linear monotone functions) that run time is $O(n \log U)$. We further show that the additional run time is equivalent to minimizing sums of subsets of the convex functions $f_j(\ )$.

At each iteration of *parametric* there is a search for breakpoints in an interval $[\lambda_1, \lambda_2]$ and the minimum cuts resulting from setting the parameter value to λ_1 and λ_2 have been computed. By comparing the cuts' source sets $S_{\lambda_1} \subseteq S_{\lambda_2}$ it is easy to determine whether the two cuts are identical. If not, there is a breakpoint in the interval.

One way of identifying the breakpoint is selecting a median λ^* in the interval and proceeding recursively on $[\lambda^*, \lambda_2]$ and $[\lambda_1, \lambda^*]$. This results in additional run time of $O(n \log U)$ since each time a median point is selected, up to $O(n)$ capacities are adjusted.

An alternative approach is to observe that if there is a *single* breakpoint in the interval $[\lambda_1, \lambda_2]$ then it is the value λ^* where the cut capacities as functions of λ intersect, $C_{\lambda_1}(\lambda^*) = C_{\lambda_2}(\lambda^*)$.

The following statements hold:

1. $C_{\lambda_1}(\lambda) - C_{\lambda_2}(\lambda)$ is a monotone nondecreasing function of λ.

2. If there are two or more breakpoints in the interval $[\lambda_1, \lambda_2]$, then λ^* "separates" them in the sense that $[\lambda^*, \lambda_2]$ and $[\lambda_1, \lambda^*]$ each contain at least one breakpoint. (This is a corollary of 1.)

Therefore, finding λ^* is required at most n times.

3. The complexity of finding λ^* is the same as the complexity of finding that the sum of derivatives of convex functions is equal to a constant K (or finding the minimum of a sum of convex functions plus the linear function $K(\lambda)$ as we see below).

We provide a proof of 1 and 3: Let $(\{s\} \cup S_1, \{t\} \cup T_1)$, $(\{s\} \cup S_2, \{t\} \cup T_2)$ be the cuts corresponding to λ_1 and λ_2 respectively.

$$\begin{aligned} &C_{\lambda_1}(\lambda) - C_{\lambda_2}(\lambda) \\ &\quad = C(S_1, T_1) - C(S_2, T_2) \\ &\qquad + C(\{s\}, T_1)(\lambda) - C(\{s\}, T_2)(\lambda) \\ &\qquad + C(S_1, \{t\})(\lambda) - C(S_2, \{t\})(\lambda) \\ &\quad = K_{12} + C(\{s\}, T_1 \setminus T_2)(\lambda) - C(S_2 \setminus S_1, \{t\})(\lambda), \end{aligned}$$

$K_{1,2}$ is a constant independent of λ. $C(\{s\}, T_1 \setminus T_2)(\lambda)$ is a monotone nondecreasing function, and $C(S_2 \setminus S_1, \{t\})(\lambda)$ is a monotone nonincreasing function. Therefore the difference between these two terms is monotone nondecreasing. Thus we proved 1.

To prove 3 we find the intersection of the two functions,

$$\begin{aligned} 0 &= C_{\lambda_1}(\lambda) - C_{\lambda_2}(\lambda) \\ &= K_{1,2} + C(\{s\}, T_1 \setminus T_2)(\lambda) - C(S_2 \setminus S_1, \{t\})(\lambda) \\ &= K_{1,2} + \sum_{j \in T_1 \setminus T_2} \max\{0, f'_j(\lambda)\} + \sum_{i \in S_2 \setminus S_1} \min\{0, f'_i(\lambda)\}. \end{aligned}$$

We now note that $T_1 \setminus T_2 = S_2 \setminus S_1$; therefore, this sum is $K_{1,2} + \sum_{i \in S_2 \setminus S_1} f'_i(\lambda)$. Thus λ^* that solve this equation is also the minimum argument of the sum of convex functions, $\sum_{i \in S_2 \setminus S_1} f_i(\lambda) + K_{1,2}\lambda$.

We call the search for λ^* in *parametric*, the *λ^*-step*.

Because the associated graph is a bipartite graph, we use here variants of the push-relabel algorithm in the parametric minimum cut that have particularly efficient running times for bipartite graphs of the type we are concerned with. Such algorithms are analyzed in a paper by Ahuja et al. (1994). We select the FIFO implementation of the push-relabel algorithm which runs in time $O(n_1^2 n_2)$ on a bipartite graph with n_1 and n_2 nodes in the bipartition where $n_1 < n_2$. Our bipartite graph has n and m nodes, and thus the largest distance label is at most $2n$. The running time using the FIFO variant is thus $O(n^2 m)$ for the inverse spanning-tree problem. The complexity of λ^*-step depends on the convex functions. For the general convex function defined on a variable in an interval of width U, one can find its integer minimum using binary search in time $O(\log U)$. For the quadratic penalty function, weighted absolute deviation and absolute deviation objective function, this time is instead $O(n)$ for all $O(n)$ functions. The total run time is thus $O(n^2 m + n\log C)$ for the general convex problem and $O(n^2 m)$ for the easily minimized convex objectives.

With the reformulation presented in §4, the complexity of step 1 is improved. The reformulation has an associated graph that is no longer bipartite, with $O(n)$ additional nodes, but still with the maximum node label bounded by $O(n)$. The maximum degree of a node in this graph is $O(\log n)$. Thus, if we use the pseudoflow algorithm in parametric, its complexity is the product of the maximum label times the number of arcs and a logarithmic factor for a total of $O(mn\log^2 n)$. With some adaptation, following the methods of Ahuja et al. (1994), the parametric push-relabel can be used as well with complexity $O(mn\log n\log(n/\log n))$.

6.1. The Absolute Deviation Problem

Consider the path graph in Figure 1 when the objective function is the sum of absolute deviations $\sum_{j\in E}|x_j - c_j|$. The functions, as proved in Lemma 3.1, are in fact $\max\{x_j - c_j, 0\}$ for $j \in E\backslash T$ and $\min\{x_j - c_j, 0\}$ for $j \in T$. Therefore, the derivatives defining the weights of the nodes in the parametric graph are, for an out-of-tree edge node j, 0 in the range $(-\infty, c_j]$ and then 1 in the range (c_j, ∞). For an in-tree edge node the derivative is -1 in the range $(-\infty, c_j]$ and 0 in the range (c_j, ∞).

Consider the modifications in the parametric graph G_λ as the value of λ increases from l to u. Let the indices of the nodes in the associated graph be ordered so that $c_1 < c_2 < \cdots < c_m$. When $\lambda \in (c_{i-1}, c_i]$, then nodes $i, i+1, \ldots, m$ are isolated in the sense that the arc connecting them to the source or the sink has capacity 0 and thus can be removed from the graph. As the value of λ increases to c_i, node i joins the graph G_λ. It is therefore simpler to consider the change in the minimum cut, if any, as one node at a time joins the parametric graph. The breakpoints are all in the set $\{c_1, c_2, \ldots, c_m\}$. The value c_i is a breakpoint if the addition of node i increases the value of the flow by 1 unit.

Consider the auxiliary graph of the reformulation. Again, it will be convenient to consider the reverse graph (V_1, V_3, V_2, A), where V_1 are nodes adjacent to the source representing the $O(n)$ edges of T, V_2 are the $O(m)$ nodes adjacent to the sink, and V_3 are the $O(n)$ added intermediary nodes between V_1 and V_2 in the reformulation. The algorithm will maintain the reachability status of all nodes (each node that has a residual path from the source reaching it is labeled as reachable). Suppose a node of V_1 is added, then updating the status of nodes newly reachable from the source and exploring whether the newly added node, and thus the source, can reach an unsaturated node of V_2 requires $O(n\log n)$ work at most. If a node of V_2 is added, then it is sufficient to check whether any of its in-neighbors is reachable from source. If so, then there is an augmenting path ending at that node. Because the indegree of a node in V_2 is at most $O(\log n)$, the complexity of this step is $O(\log n)$.

To summarize, if the added node is $i \in V_1$, then the complexity is $O(n\log n)$, and for $|V_1|$ nodes the complexity is $O(n^2\log n)$, accounting for the addition of all nodes of V_1. If the added node is $i \in V_2$, then the complexity is $O(\log n)$, and for all $|V_2|$ nodes the complexity is $O(m\log n)$.

The total complexity is thus $O((n^2 + m)\log n)$, which is $O(n^2\log n)$.

6.2. The Maximum Deviation Problem

We now address the IST problem with the L_∞ norm. Sokkalingam et al. (1999) showed that the optimal value is $\delta/2$ where $\delta = \max\{0, \max_{i\in T\, j\in E\backslash T}(c_i - c_j)\}$. This quantity can be calculated in a straightforward manner in $O(mn)$ steps. Sokkalingam et al. (1999) showed how to compute δ in $O(n^2)$ steps. We use the reformulation to demonstrate that this computation can be done in $O(m\log n)$ steps.

In the auxiliary graph $(V_1, V_3, V_2, \tilde{A})$ we compute recursively for each node of $V_1 \cup V_3$, the maximum value of c_i among all predecessors of the node. The computation starts by labeling all nodes i of V_1 with their weights c_i. For each node of V_3 that has all its predecessors labeled, assign it the maximum label among all its predecessors. The complexity of this process is the number of arcs (adjacency relations) visited, which is $O(n\log n)$. Now, for each node j of V_2, compare the value of c_j to the label of each of its no more than $\log n$ neighbors in $V_1 \cup V_3$. Finding the value of δ is thus dominated by this computation, which requires at most $O(m\log n)$ comparisons. When the graph is not dense, the run time of $O(m\log n)$ is more efficient than $O(n^2)$.

7. COMPLEXITY ISSUES IN CONVEX MINIMIZATION

Nonlinear and nonquadratic optimization problems with linear constraints were proved impossible to solve in strongly polynomial time in a complexity model of the arithmetic operations, comparisons, and the rounding operation (Hochbaum 1994). That implies that even convex minimization over a bounded interval may not be solved in strongly polynomial time. Instead, the complexity has to depend either on the convex function analytic description (or variability) or on the size of the interval. Therefore, all algorithms for convex IST must have the $\log C$ factor in the run time expression.

Integer minimization of a convex function on an interval of length U can be trivially accomplished using binary search in $O(\log U)$. In the λ^*-step of the CCC algorithm there are up to n convex functions minimizations over bounded intervals; thus, the complexity of that stage is $O(n \log C)$.

Now, consider the problem of convex minimization in real variables. The solution to such a problem may be irrational and impossible to represent with finite accuracy. In Hochbaum and Shanthikumar (1990) we introduced for that purpose the ϵ-accuracy complexity model. That complexity model has the solution to the continuous problem given with ϵ-accuracy on a grid of size ϵ or with a number of significant bits that is $O(\log(1/\epsilon))$. The negative result on strong polynomiality translates to requiring run time which is at least polynomial in $\log(U/\epsilon)$ for an ϵ-accurate solution.

The complexity of λ^*-step in the CCC algorithm is $O(n \log C)$ for the integer-valued solution, or a complexity of $O(n \log(C/\epsilon))$ for the continuous solution of ϵ accuracy. The definition of the subgradient when we solve the problem in continuous variable with ϵ accuracy is $f'(x) = [f(x+\epsilon) - f(x)] \cdot \epsilon$. With the cut-based algorithm the solving of the scaled problem s-IST proceeds until $s \leqslant \epsilon$. This increases the number of calls to the solution of s-IST to $O(\log(C/\epsilon))$.

For simple functions such as weighted absolute deviation functions, or quadratic functions, λ^*-step can be implemented in $O(1)$ for each of the interval convex minimization problems. Next, we discuss further the complexity of the problem with the quadratic objective function.

7.1. The Quadratic Convex Closure Problem

The negative result proved in Hochbaum (1994) on the impossibility of solving constrained nonlinear problems in strongly polynomial time is not applicable to the quadratic case. Thus, it may be possible to solve constrained quadratic optimization problems in strongly polynomial time, yet very few constrained quadratic optimization problems are known to be solvable in strongly polynomial time. For instance, it is not known how to solve the minimum quadratic convex cost network flow problem in strongly polynomial time. For the convex quadratic IST problem, our result adds to the limited repertoire of quadratic problems solved in strongly polynomial time.

In the quadratic case, the CCC algorithm is implemented to run in strongly polynomial time. This is easily achieved because finding the minima in the λ^*-step of the algorithm amounts to solving a linear equation in one variable. Therefore, the run time required for finding all the minima is $O(n)$, and thus the overall run time of the algorithm is dominated by the complexity of a single minimum cut. Note that among the other algorithms that apply to the quadratic IST none solves the problem in strongly polynomial time.

ENDNOTE

1. All minimum cut problems mentioned here are in fact the minimum s,t-cut problem, which calls for a partition of the graph to S and $\overline{S}$ so that S contains a specific source node s and $\overline{S}$ contains t, and the total capacity of the arcs between S and $\overline{S}$ is minimized.

ACKNOWLEDGMENTS

This research was supported in part by NSF award Nos. DMI-0085690 and DMI-0084857 and by UC-smart award. Special thanks to Jim Orlin for the inspiration for writing this paper and providing the pointer to reference Ahuja and Orlin (2000). Our discussion on the properties of the reformulation provided the essential ingredients in the algorithms described here. Extra thanks to Ilan Adler who, while converting espresso to theorems, provide statement 1 for the parametric algorithm.

REFERENCES

Ahuja, R. K., J. B. Orlin. 2000. A faster algorithm for the inverse spanning tree problem. *J. Algorithms* **34** 177–193.

——, D. S. Hochbaum, J. B. Orlin. 1999a. Solving the convex cost integer dual network flow problem. G. Cornuejols, R. E. Burkard, G. J. Woeginger, eds. *Proc. IPCO'99. Lecture Notes in Computer Science*, Vol. 1610, 31–44. *Management Sci.* Forthcoming.

——, ——, ——. 1999b. A cut based algorithm for the convex dual of the minimum cost network flow problem. Manuscript, University of California, Berkeley, CA.

——, J. B. Orlin, C. Stein, R. E. Tarjan. 1994. Improved algorithms for bipartite network flow. *SIAM J. Comput.* **23** 906–933.

Barlow, R. E., D. J. Bartholomew, J. M. Bremer, H. D. Brunk. 1972. *Statistical Inference Under Order Restrictions*. Wiley, New York.

Dinic, E. A. 1970. Algorithm for solution of a problem of maximal flow in a network with power estimation. *Soviet Math. Dokl.* **11** 1277–1280.

Even, S., R. E. Tarjan. 1975. Network flow and testing graph connectivity. *SIAM J. Comput.* **4** 507–518.

Gallo, G., M. D. Grigoriadis, R. E. Tarjan. 1989. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.* **18** 30–55.

Goldberg, A. V., R. E. Tarjan. 1988. A new approach to the maximum flow problem. *J. ACM* **35** 921–940.

Gusfield, D., C. Martel, D. Fernandez-Baca. 1987. Fast algorithms for bipartite network flow. *SIAM J. Comput.* **16** 237–251.

Hochbaum, D. S. 1994. Lower and upper bounds for allocation problems. *Math. Oper. Res.* **19** 390–409.

——. 1997. The pseudoflow algorithm for the maximum flow problem. Manuscript, University of California, Berkeley, CA.

——, M. Queyranne. 2003. The convex cost closure problem. *SIAM J. Discrete Math.* **16** 192–207.

——, J. G. Shanthikumar. 1990. Convex separable optimization is not much harder than linear optimization. *J. ACM* **37** 843–862.

Picard, J. C. 1976. Maximal closure of a graph and applications to combinatorial problems. *Management Sci.* **22** 1268–1272.

Sleator, D. D., R. E. Tarjan. 1983. A data structure for dynamic trees. *J. Comput. Systems Sci.* **24** 362–391.

Sokkalingam, P. T., R. Ahuja, J. B. Orlin. 1999. Solving inverse spanning tree problems through network flow techniques. *Oper. Res.* **47** 291–298.

Tarantola, A. 1987. *Inverse Problem Theory: Methods for Data Fitting and Model Parameter Estimation*. Elsevier, Amsterdam, The Netherlands.