When trees collide: An approximation algorithm for the generalized Steiner Problem on Networks

Ajit Agrawal Philip Klein and R. Ravi

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-90-32
June 1994

When trees collide: An approximation algorithm for the generalized Steiner problem on networks.*

To appear in *SIAM Journal on Computing*

Ajit Agrawal[1] Philip Klein[1]
R. Ravi[1]
Brown University

Abstract

We give the first approximation algorithm for the *generalized network Steiner problem*, a problem in network design. An instance consists of a network with link-costs and, for each pair $\{i, j\}$ of nodes, an edge-connectivity requirement r_{ij}. The goal is to find a minimum-cost network using the available links and satisfying the requirements. Our algorithm outputs a solution whose cost is within $2\lceil \log_2(r+1) \rceil$ of optimal, where r is the highest requirement value.

In the course of proving the performance guarantee, we prove a combinatorial min-max approximate equality relating minimum-cost networks to maximum packings of certain kinds of cuts. As a consequence of the proof of this theorem, we obtain an approximation algorithm for optimally packing these cuts; we show that this algorithm has application to estimating the reliability of a probabilistic network.

AMS 1980 Subject Classification: 68R10, 68Q25
Keywords: Approximation algorithm, Network design, Steiner tree problem.

*A preliminary version of this paper appeared in the *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing* (1991), pp. 134-144.

[1]Research supported by NSF grant CCR-9012357, NSF grant CDA 8722809, ONR and DARPA contract N00014-83-K-0146 and ARPA Order No. 6320, Amendment 1 .

1 Introduction

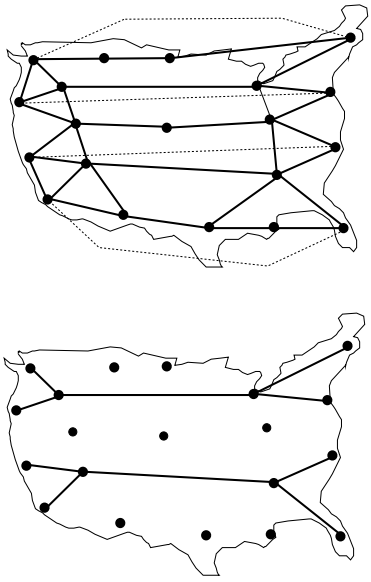

Figure 1: An instance of the unweighted network design problem and its solution. Solid edges correspond to unit-cost links; dotted edges connect site pairs.

Consider the following scenario. Client industries of a telephone company have requested commercial telephone connections between pairs of their offices in different cities. The telephone company must then install a network of fiber-optic telephone links that accommodates all the clients' requirements. That is, the network must contain a path using these links between every pair of cities specified by the clients. Given the cost of installing links between different cities, the company must now decide which links to install so as to minimize its cost.

We formalize the problem as follows. Let G be a graph with nonnegative edge-costs, and let R be a set of node-pairs (s_i, t_i). We call these pairs *site-pairs*, and we say the nodes s_i, t_i are *sites*. We call a subgraph H of G a *requirement join* if H contains a path between s_i and t_i for every requirement (s_i, t_i). We call the node-pairs *requirements* because they represent connectivity constraints that must be satisfied by the output subgraph. We abbreviate *requirement join* by *R-join* when we want to emphasize the

set R of requirements. The problem we consider in this paper is to find a minimum-cost R-join.

The problem faced by the telephone company can be directly formulated as a minimum-cost R-join problem. In this formulation, it is assumed that a link can be used simultaneously by many clients. This assumption is reasonable in light of the very high bandwidth of fiber-optic links.

Consider the special case of this problem in which there is a set T of *terminals*, and every pair of nodes in T needs to be connected. This special case is known in the literature as the *Steiner tree problem in networks*. This problem was one of the first seven problems shown NP-complete by Karp [19]. Given the range of its applications, it is not surprising that this problem has been well-studied. Many enumeration algorithms, heuristics [33, 44, 18], and approximation algorithms [6, 38, 25, 11, 31, 37, 29, 45] are known for the problem. Polynomial-time solutions for restricted classes of graphs are also known (see [42]).

However, none of the algorithms addresses the more general case in which each client can specify an arbitrary pair of cities. Note that in this general case the solution network need not be connected. Moreover, a minimum-cost Steiner tree solution can be arbitrarily costlier than a minimum-cost solution to this more general problem.

In this paper, we give the first approximation algorithm for the minimum-cost R-join problem.

Theorem 1.1 *There is a polynomial-time algorithm to find an R-join of cost at most $2 - 2/k$ times minimum, where k is the number of sites.*

1.1 The Generalized Steiner Problem in Networks

The algorithm of Theorem 1.1 is useful when the network to be constructed need not be connected. However, the algorithm is also useful, as a subroutine, even in designing connected networks. Namely, we consider a generalization of the minimum-cost R-join problem involving certain redundancy requirements.

Consider the scenario described above but where each client can specify that her pair of cities must be connected by some number of edge-disjoint paths, in order that the connection be less vulnerable to link failure. The goal is to design a network satisfying these specifications. The network is allowed to contain multiple links between the same pair of nodes; all such links have the same cost.

To model this situation, we allow more general requirements. The set R of requirements consists of triples (s_i, t_i, r_i), where r_i, the requirement value, is a positive integer. An *R-multijoin* is a multiset of the edges of G that contains r_i edge-disjoint paths from s_i to t_i, for every requirement (s_i, t_i, r_i). The cost of an R-multijoin is the sum of the costs of the edges in the multiset, counting multiplicities. Using our approximation algorithm for minimum-cost R-join, we obtain an approximation algorithm for minimum-cost R-multijoin.

Theorem 1.2 *There is a polynomial-time algorithm to find an R-multijoin of cost at most $(2 - 2/k)\lceil \log_2(r_{\max} + 1) \rceil$ times minimum, where $r_{\max}$ is the largest requirement value and k is the number of sites.*

This problem is called *multiterminal network synthesis* by Chien [7] and Gomory and Hu [13]. Gomory and Hu and later Sridhar and Chandrasekaran [35] address the synthesis problem for the special case where the input graph is the complete graph with all costs identical.

The problem is also essentially identical to the *generalized Steiner problem* as formulated by Krarup ([26], as cited in [41]). The problem is referred to as the design of minimum-cost *survivable networks* in the work of Steiglitz, Weiner, and Kleitman [36]. These researchers pose the problem of finding a subgraph of minimum cost satisfying given connectivity requirements. The problem we address differs in that we allow the solution to contain multiple copies of links appearing only once in the input graph G.

In this paper we will use the term *R-multijoin* whenever the requirements R include requirement values exceeding one, and will reserve the term *R-join* for the case when the requirement values are all one. The former case is addressed only in Subsection 3.3, where we show how to reduce the problem to the latter case. The remainder of the paper addresses only the latter case.

1.2 Packing cuts, with application to network reliability

Our results also have application to evaluating network reliability. Suppose the telephone company has an existing network and the same list of clients, each specifying a pair of cities. The company needs to determine how likely it is that random failure of communication links renders some of its clients' requirements unsatisfiable.[2] Assuming link failures are inde-

[2] A related problem—finding the minimum number of communication links that would need to fail for *all* requirements to be unsatisfiable—can be solved approximately, using techniques we have presented in an earlier paper [22].

pendent, determining the probability that the surviving links can serve all clients' requirements is a generalization of the $\#P$-complete problem [39] called Network Reliability. No approximation algorithms are known.

However, one powerful and useful heuristic for estimating two-terminal and k-terminal reliability [8, 9] can be directly generalized to handle the case of arbitrary pairs. The (generalized) heuristic consists in finding a large collection of edge-disjoint cuts in the network such that each cut separates at least one client's pair of cities. For a surviving network to be able to serve all clients' requirements, at least one edge in each cut must survive; thus such a cut-packing can be used to obtain a lower bound on the probability of catastrophic failure. Experience [9] with this heuristic in the cases of two-terminal and k-terminal reliability indicates that it is one of the best available.

One of the results of this paper is an algorithm for finding a nearly maximum collection of such cuts in an auxiliary network whose reliability is the same as that of the original network. We give more details in Section 3.

1.3 The combinatorial basis for our algorithms: a new approximate min-max equality

At the heart of our proofs of near-optimality is a combinatorial theorem that relates the R-join problem to the cut-packing problem in the case of unit edge-weights.

Theorem 1.3 *The minimum size of an R-join is approximately equal to one-half the maximum size of a collection of cuts, where each cut separates some site-pair, and no edge is in more than two cuts. By "approximately," we mean within a factor of $2 - 2/k$, where k is the number of sites.*

The proof of Theorem 1.3 is algorithmic, and is given in Section 5. We can formulate the two combinatorial quantities as the values of integer linear programs that are dual to one another. It follows from Theorem 1.3 that the fractional relaxations of these programs provide good approximations to both combinatorial quantities. Moreover, the factor of $2-2/k$ is existentially tight, as shown by the example of a k-cycle given by Goemans and Bertsimas [14].

2 Related work

2.1 The Steiner tree problem in networks

There have been volumes of work done on the Steiner tree problem in networks, including proposed solution methods, computational experiments, heuristics, probabilistic and worst-case analyses, and algorithms for special classes of graphs. Winter [41] and more recently Hwang and Richards [17] surveyed this body of work.

Karp [19] showed that the problem is NP-complete. Takahashi and Matsuyama [38], Kou, Markowsky and Berman [25], El-Arbi [11], Rayward-Smith [33], Aneja [2], and Wong [44] are among those who proposed heuristics. Among these, the heuristics that have been analyzed have a worst-case performance ratio of $2 - 2/k$, where k is the number of terminals that need to be connected (called Z-vertices in [41]). One algorithm, proposed by Plesnik [31] and by Sullivan [37], performs somewhat better. Recently Zelikovsky gave an approximation algorithm with a performance ratio of 11/6 [45]. Berman and Ramaiyer [6] have improved this to 16/9.

In computational experiments, these heuristics generally perform considerably better than the worst-case bounds. Jain [18] proposed an integer-program formulation of the Steiner tree problem in networks, and showed that for two random distributions of costs, the value of this integer program differed drastically from the value of its fractional relaxation.

2.2 The generalized Steiner problem in networks

The *generalized Steiner problem in networks*, as originally formulated by Krarup (see [42]), is as follows. The input consists of a graph with edge-costs, a subset Z of the vertices, and, for each pair of vertices $i, j \in Z$, a required edge-connectivity r_{ij}. The goal is to output a minimum-cost subnetwork satisfying the connectivity requirements. When the r_{ij}'s are allowed to be zero, we can clearly assume without loss of generality that Z consists of all the vertices of the graph.

Previous to our work, no approximation algorithms for the generalized Steiner problem were known. There have been papers on finding exact solutions and on algorithms for special classes of graphs [41, 42].

In the work of Goemans and Bertsimas, described below, and in our work, the edge-connectivity requirement is allowed to be satisfied in part by duplicating edges of the input graph. This corresponds to "buying" multiple communication links of the same cost and with the same endpoints.

2.3 Survivable networks

In recent work, Goemans and Bertsimas [14] considered a special case of the generalized Steiner problem in networks. Instead of arbitrary requirement values, the input includes an assignment of integers r_i to nodes. The goal is to find a minimum-cost network satisfying requirements $r_{ij} = \min(r_i, r_j)$. They propose a simple but powerful approach which involves solving a series of ordinary Steiner tree problems using a standard heuristic. They show that this approach yields solutions that are within a factor of $2\min(\log R, p)$ of optimal, where R is the maximum r_i and p is the number of distinct nonzero values r_i in the input. Moreover, they show that their analysis is tight in the worst case.

Goemans and Bertsimas restricted their attention to edge-connectivity requirements of the special form $r_{ij} = \min(r_i, r_j)$ in order that each subproblem have essentially the form of an (ungeneralized) Steiner tree problem. That enabled them to solve each subproblem approximately, using one of the known approximation algorithms for Steiner tree. By providing an approximation algorithm for the case of $r_{ij} \in \{0, 1\}$, we make it possible to handle requirements r_{ij} not of that special form.

2.4 Subsequent work

Building on our result, Goemans and Williamson [16] simplified and generalized our algorithm. They describe a framework in which to formulate and find approximately optimal solutions for many *constrained forest problems*, of which the minimum-cost R-join problem is an example. Their approximation algorithm uses an approach similar to ours, and achieves the same performance guarantee. Goemans and Williamson describe an implementation of their algorithm that runs in $O(n^2 \log n)$ time on graphs with n nodes.

In work building on that of Goemans and Williamson, we showed [32] how to obtain approximately optimal solutions to 2-edge-connected versions of the problems addressed in [16]. In these problems, one needs to achieve 2-connectivity without duplicating links. Finally, several subsequent papers [15, 23, 40] extended these methods to give approximation algorithms for the generalized Steiner problem without link duplication.

3 Background

An instance of the generalized Steiner problem consists of a graph G with edge-costs c, together with a collection $\{R_1, \ldots, R_b\}$ of *requirements*: each requirement R_i consists of a *site pair* $\{s_i, t_i\}$, a pair of nodes of G, and a *requirement value* r_i, a positive integer. A feasible network is a multiset N consisting of edges of G, such that for every requirement $R_i = (\{s_i, t_i\}, r_i)$, there are at least r_i edge-disjoint paths between s_i to t_i in the multigraph with edges N.

3.1 The unweighted case

To prove performance guarantees for our algorithm, we exploit an approximate duality between feasible networks and packings of cuts. Fix some instance of the generalized Steiner problem, where all costs and requirement values are 1. Thus the instance consists of a graph G and a collection of site pairs $\{s_i, t_i\}$. Let k denote the cardinality of the set of sites, i.e. the set of nodes appearing in site pairs. Note that the number of sites may be significantly smaller than the total number of nodes. A *feasible network* is a subgraph in which, for every site pair $\{s_i, t_i\}$, there is a path between s_i and t_i.

Let N be any feasible solution for this instance. Observe that if N is minimal, then it is just a forest. Let S be any subset of nodes of G such that for some site pair $\{s_i, t_i\}$, one of the sites is in S and one is not. In this case, the set of edges A with exactly one endpoint in S is called a *requirement cut*. There must be a path between s_i and t_i in N, so N intersects A in at least one edge. Thus we have

Lemma 3.1 *Every feasible network and every requirement cut have at least one edge in common.*

Suppose $A_1, \ldots, A_\beta$ are (not necessarily distinct) requirement cuts such that each edge of G occurs in at most two cuts. We call such a collection of cuts a 2-packing. Then we have the following easy lower bound on the minimum size of a network design.

Lemma 3.2 *The minimum size of a feasible network is at least one-half the maximum size of a 2-packing of requirement cuts.*

Proof: Let N be a feasible network and let $A_1, \ldots, A_\beta$ be a 2-packing consisting of β requirement cuts. We have

$$|N| \geq \sum_{e \in N} \frac{1}{2} |\{i : e \in A_i\}| = \frac{1}{2} \sum_{i=1}^{\beta} |A_i \cap N| \geq \frac{1}{2}\beta \tag{1}$$

because each $|A_i \cap N|$ is at least one. □

For comparison, Edmonds and Johnson [10] show that T-joins and T-cuts satisfy an analogous inequality, and, more importantly, they satisfy it with equality.

Instead of showing equality, we show approximate equality, to within a factor of $2(1 - 1/k)$. This is the content of Theorem 1.3.

Our proof of Theorem 1.3 is algorithmic. We give an algorithm that constructs a feasible network and a 2-packing, such that the first has size at most $(1 - 1/k)$ times the second. It follows that the feasible network is approximately minimum and the 2-packing is approximately maximum, to within a factor of $2(1 - 1/k)$.

The first step is to transform the original graph G_0 into a bipartite graph G by replacing each edge uv of G_0 with two edges ux and xv in series, where x is a new node. The resulting graph G has the following properties:

- Any minimal feasible network in G corresponds to a feasible network in G_0 of half the size.
- Any packing of edge-disjoint requirement cuts in G corresponds to a 2-packing of requirement cuts in G_0 of the same size.

Consequently, in order to prove Theorem 1.3 for G_0, it is sufficient to show the following for G:

> We can find a feasible network N and a packing of edge-disjoint requirement cuts $A_1, \ldots, A_\beta$ such that $N \leq 2(1 - 1/k)\beta$, where k is the total number of sites. (2)

We show (2) in Sections 4 and 5.

3.2 The weighted case

Now we consider the case in which the costs of edges may vary, but the requirement values are still all one. It turns out that, like Edmonds and

Johnson's theorem, Lemma 3.2 and Theorem 1.3 are self-refining. For non-negative integer edge-costs c, we simply replace each edge e by a path of length $c(e)$. We say a collection of requirement cuts is a $2c$-packing if each edge e appears at most $2c(e)$ times. Using this transformation, we obtain the following theorem from Theorem 1.3.

Theorem 3.3 *The minimum-cost of a feasible network is at least one-half the size of a $2c$-packing of requirement cuts, and at most $(1 - 1/k)$ times this size.*

To actually compute an approximately minimum feasible network, we use a more direct approach, which we describe in Subsection 4.2.1.

3.3 Arbitrary integral requirements

So far we have dealt with the case in which each site pair need only be connected in the final feasible network. As discussed in the introduction and Section 2, a client may also require that there be at least r_{ij} edge-disjoint paths between her pair of sites.[3] Thus the case dealt with up to now requires each r_{ij} to be either 0 or 1.

In order to obtain an approximation algorithm for this generalized problem from our algorithm for the case of 0–1 requirements, we make use of a heuristic technique due to Goemans and Bertsimas [14]. They propose a technique they call the *tree heuristic*, which consists essentially of decomposing a problem with many different requirement values into a series of simpler problems in which only two requirement values appear. As we mentioned in Section 2, they use the technique for solving only a special case of the generalized Steiner problem. In conjunction with our new algorithms for the 0–1 case, however, the technique can be easily adapted to apply to the general case.

Let the different values of r_{ij} be $0 = p_0 < p_1 < p_2 < \cdots < p_s$. For each $0 < d \leq s$, consider the transformed problem

$$r_{ij}^d = \begin{cases} p_d - p_{d-1} & \text{if } r_{ij} \geq p_d \\ 0 & \text{otherwise} \end{cases}$$

which is essentially $p_d - p_{d-1}$ copies of a 0–1 problem. Use a standard heuristic to find an approximately optimal solution, and combine the solutions to

[3] In this case, the feasbile network is allowed to use multiple copies of edges of the input graph; each copy of a given edge costs the same.

the s transformed problems to get a solution to the original problem. The resulting performance guarantee is $\Theta(s)$.[4] By using a similar approach, if each r_i is an integer b bits long, then the original problem can be decomposed into b problems, and the resulting performance bound is $2(1 - 1/k)b$. This is how we get the performance bound stated in Theorem 1.2.

The obvious question is whether one can do better than this. Goemans and Bertsimas can show that their analysis is tight, so another approach is needed, one that can deal simultaneously with widely varying requirement values.

3.4 Reliability estimation

In the introduction, we described a heuristic for estimation of network reliability in a probabilistic network. In order to use this heuristic effectively, we want to find a maximum collection of edge-disjoint requirement cuts. This problem is NP-complete for general graphs. Moreover, an approximation algorithm for this cut-packing problem would yield an approximation algorithm for maximum independent set [9], an unlikely outcome in view of recent results [3, 4, 12]. We instead show how to make use of a cut-packing in bipartite graphs. We apply the transformation described in Subsection 3.1 to turn an arbitrary graph into a bipartite graph with all sites on one side of the bipartition: replace an edge having failure probability $1 - p$ with two series edges each having failure probability $1 - \sqrt{p}$. We do not change the probability of reliability in carrying out this transformation, and we can apply the algorithm of Section 4 to find an approximately maximum set of edge-disjoint cuts in the resulting graph.

Thus we propose a four-step recipe for estimating network reliability. Transform the network into a bipartite network, find an approximately maximum cut-packing, compute for each cut the probability that at least one edge survives, and multiply these probabilities to get an upper bound on the probability that all clients can continue to communicate.

4 The algorithm

In this section, we describe an algorithm for finding a cut-packing and an R-join. In Subsection 4.2, we describe how to find a cut-packing in the case of

[4] More specifically, Goemans and Bertsimas show the performance bound is $2(1 - 1/k)(\sum_{d=1}^{s}(p_d - p_{d-1})/p_d)$.

unit edge-weights. In Subsection 4.2.1 we describe the modification needed to handle arbitrary edge-weights. The algorithm for finding an R-join is the same in the two cases.

4.1 Overview

We start by providing an overview of the algorithm for the case of unit edge-weights. The algorithm grows breadth-first search trees from the sites, accumulating cuts as it proceeds. The algorithm employs a notion of timesteps. At each timestep, each of the breadth-first trees grows by an additional level. Each tree grows until all the sites it contains have found their mates. When trees collide, they are merged. As the algorithm grows trees, it builds networks spanning the sites in each of these trees. Using a charging scheme, we show that the size of each network in a tree is about twice the number of cuts accumulated while growing the tree.

4.2 Finding a cut-packing

Assume the input graph has unit edge-weights; we briefly address the more general case at the end of this subsection. Let G be a bipartite graph with all sites on the same side of the bipartition. (We can obtain such a graph from an arbitrary graph as described in Subsection 3.1. All subsequent references to the "original graph" refer to G.) We are given a collection of site pairs $\{s_1, t_1\}, \{s_2, t_2\}, \ldots, \{s_b, t_b\}$. We refer to the nodes s_i, t_i as *sites*. We say that two sites in the same site pair are *mates* of each other.

The algorithm for constructing the cut-packing is quite intuitive. (A summary is given at the end of this subsection.) We grow disjoint breadth-first search trees from all sites s simultaneously. We call the edges connecting one level to the next in a breadth-first search tree a *level cut*. Each level cut in a breadth-first search tree rooted at s is a requirement cut because its edges separate s from its mate. Thus at each timestep, we accumulate one additional requirement cut for each tree being grown.

When multiple trees collide, we merge them into a single tree and continue growing from its boundary. Thus in general a tree may contain many sites. As soon as every site in a tree has its mate in the same tree, we can no longer guarantee that subsequent level cuts of the tree are requirement cuts, so we call the tree *inactive*, and we contract all its nodes into a single *supernode*. A tree that is still in the process of being grown is said to be *active*. The algorithm terminates when there are no active trees. At this

point, every site pair's two nodes are contained in the same tree. More precisely, since each tree has become inactive, and has hence been contracted to a supernode, there are no sites remaining in the graph.

Because of contractions, the graph on which we are working evolves during the course of the algorithm. We use G_t to denote the graph after t timesteps. When we refer to a graph, unless we explicitly call it the "original graph," we will mean the contracted graph G_t at a certain point t in the algorithm.

It is important to the analysis that all active trees grow at the same rate. The algorithm takes place over a series of timesteps. In each timestep, each active tree grows by one level. Thus after t timesteps, active trees that have not participated in any collisions all have radius t (as measured in the contracted graph G_t). More generally, let the *boundary* of a tree be the set of nodes at the most recent level of the tree. We have the following proposition

Propostion 4.1 *After t timesteps, each node in the boundary of an active tree is distance t from some site internal to the tree.*

In the initial bipartite graph, all the sites are on the same side of the bipartition. We show that this property continues to hold throughout the algorithm.

Lemma 4.2 *After t timesteps, the graph is still bipartite, with all sites on the same side of the bipartition.*

Proof: By induction on t. The basis $t = 0$ is trivial. We must show that the bipartition property described in the lemma is preserved by contractions. Suppose that G_{t-1} obeys the property, and that after t timesteps, some tree T has just become inactive and is about to be contracted. By Proposition 4.1, all the nodes in the boundary of T have distance t from some site. Hence they all belong in the same side of G_{t-1}'s bipartition. It follows that after the nodes of T are contracted to a single node, the bipartition property still holds. □

We can use Lemma 4.2 to show that all the cuts found by the algorithm are edge-disjoint.

Corollary 4.3 *No edge belongs to a level cut of more than one tree.*

Proof: By Proposition 4.1 and Lemma 4.2, all the nodes in boundaries of all active trees are in the same side of the bipartition of the graph. Hence no edge is incident to two active trees. □

Thus trees collide by reaching the same node in a given step. Below we summarize the cut-packing algorithm. In anticipation of the analysis of the algorithm, we "assign" cuts found to particular trees.

1 Initialize each site to be an active tree. Repeat the following steps until every tree is inactive.

2 Grow each tree by one level. Assign the corresponding level cut to the tree.

3 Contract each tree that has just become inactive.

4 Repeat

5 Take two distinct trees sharing a boundary node, and merge them into a single tree. (For the cut-packing algorithm, merging trees consists merely of taking the union of their nodes and of the cuts assigned to them.)

6 Until no more trees can be merged.

Because trees are merged immediately after they collide, we can claim the following: Just before the trees are grown, they are node-disjoint. Just after the trees are grown, they are *internally* node-disjoint: only their boundaries can share nodes. We make use of this property in the next subsection.

4.2.1 The cut-packing algorithm for weighted edges

The cut-packing algorithm in the case of weighted edges is only slightly more elaborate. We describe it in this subsection. The algorithm to find a feasible network based on the cut-packing, described in the next section, remains unchanged.

The key is to carry out many timesteps in a single iteration. It is useful to imagine that in each iteration, the growing trees continuously "consume" all their incident edges at the same rate until some edge is completely consumed, at which time things must be updated. We assume for simplicity that the edge-weights are integral. For each edge, we maintain a variable indicating how much of that edge remains to be consumed. To determine the amount λ by which to grow active trees in an iteration, we compute two minima:

$$\lambda_1 = \min_e \text{ amount of } e \text{ yet to be consumed} \tag{3}$$

where the min is over edges e that have one endpoint in an active tree, and

$$\lambda_2 = \min_e \text{ amount of } e \text{ yet to be consumed} \tag{4}$$

where the min is over edges e that have both endpoints in distinct active tree.

Finally, we let $\lambda = \min\{\lambda_1, \frac{1}{2}\lambda_2\}$.

To grow the trees by λ, we update the variables associated with edges: each edge having one endpoint in an active tree has its variable decreased by λ, and each edge having its endpoints in distinct active trees has its variable decreased by 2λ (because each such edge is being consumed from both sides). Then we execute steps 3 through 6 of the unweighted algorithm. It follows by definition of λ that at least one edge is wholly consumed in an iteration, hence at least one tree grows by at least one node. For a tree T, let t_T be the number of nodes in T. It follows that the potential function $\sum_T t_T -$ (number of trees) goes up by at least one in each iteration, and hence that the number of iterations is at most the number of nodes in the graph. Thus the cut-packing algorithm requires only polynomial time.[5]

4.3 The network-design algorithm

The basic approach to building a feasible network is also quite intuitive. For each tree, we maintain a connected network connecting together all sites in the tree. This is easy: start with each site being a network in itself, and, whenever trees merge, use simple paths to join up their two networks.

It is possible to show that for each tree, the size of a network for that tree is no more than twice the number of cuts assigned to the tree. Such an analysis, however, is insufficient: the networks formed in this way are not connected in the original graph, because of the contractions we have performed along the way. A path that contains a supernode is not in general a path in the original graph. Therefore, we must be more careful in joining networks, and must not forget to include edges between nodes within inactive trees. Note that such edges do not even appear in the contracted graph G_t.

We introduce some terminology to help us relate various contracted graphs to each other and to the original graph. We call a node a *real node* if it appears in the original graph, in order to distinguish such nodes from supernodes. If the tree T was contracted to form the supernode v, we say T *corresponds* to v, and vice versa. We say v *immediately encloses* v' if v is a supernode corresponding to a tree T containing v'. Note that each node is immediately enclosed by at most one node. A node enclosed by another

[5] Using a heap to organize the edges incident to each tree, one can implement the algorithm to run in $O(n^2 \log n)$ time [1]. Using a more sophisticated two-level heap structure, one can implement it in $O(n\sqrt{m}\log n)$ time [21].

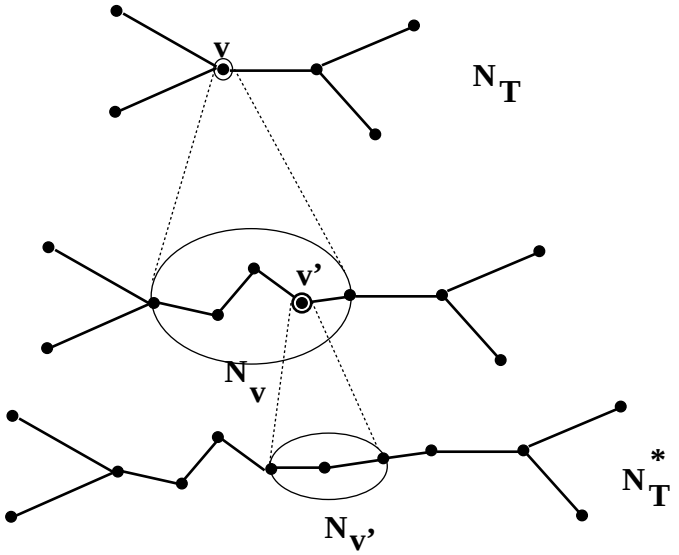

Figure 2: The network N_T corresponds in a natural way to a subgraph N_T^* of the original graph. To obtain N_T^* from N_T, replace each supernode v in N_T with the subgraph N_v, and recurse on the supernodes in N_v, like v' in the above figure.

node does not appear in the current graph, but we cannot simply forget about it since it continues to play a role in the algorithm.

We define the relation *encloses* to be the reflexive and transitive closure of the relation *immediately encloses*. That is, v encloses v' if by some series of contractions, v' was identified with other nodes to form v.

For an edge e incident to a node v in a contracted graph, there is a real node v' enclosed in v such that e is incident to v' in the original graph. We say that v' is the real node *by which e is incident to v*.

For each tree T, we maintain a *network* N_T, a subgraph of T. We maintain the following **site-inclusion invariant:**

> For each T, the network N_T includes all sites that are nodes of T.

We specifically mean to exclude those sites strictly enclosed by supernodes belonging to T. The site-inclusion invariant speaks only of those sites that are themselves nodes of T. If v is a supernode corresponding to an (inactive) tree T, we use T_v to denote T, and we use N_v to denote N_T. We say a node

is *free* if it is not contained in any network N_T.

Each network N_T corresponds in a natural way to a subgraph N_T^* of the original graph. Namely, to get N_T^* from N_T, replace each supernode v in N_T with the subgraph N_v, and recurse on the supernodes in N_v.

We want each network N_T to correspond to a connected subgraph in the original graph. We therefore maintain the following **connectivity invariant**.

> Each subgraph N_T^* is connected.

At any stage in the algorithm, the networks N_T induce a subgraph of the original graph, namely the subgraph induced by the edges in $\bigcup_T N_T$ where the union is over all trees active and inactive. Let us call this subgraph N. Note that each induced subgraph N_T^* is a subgraph of N.

We now observe that when the algorithm terminates, the invariants imply that N is indeed a feasible network—for each site pair $\{s_1, s_2\}$, there is a path in N between s_1 and s_2. Let T be the tree containing s_1. Once the algorithm terminates, T must be inactive, and hence contains s_1's mate s_2 as well. By the site-inclusion invariant, s_1 and s_2 are nodes of N_T. Since they are original nodes, they are also nodes of the induced subgraph N_T^*, which is a subgraph of N. Finally, by the connectivity invariant, N_T^* is connected, so the required path exists.

Now we give the algorithm for network design. We run the cut-packing algorithm of the last subsection and, whenever a "merge" of trees occurs, we update the N_T's in order to maintain the invariants. Initially, when every active tree T consists of a single site, N_T consists also of this site. For each tree T not yet formed, N_T is empty. Thus trivially the invariants hold initially.

In step 5 of the cut-packing algorithm, we merge a pair of distinct trees T_1 and T_2 sharing a common boundary node v. By simply taking the union of their networks N_{T_i}, we get a network that obeys the site-inclusion invariant. However, this network does not obey the connectivity invariant. We must therefore connect up these networks. To do this, we add paths from the common node v to each of the networks N_{T_i}. This involves some care when v is a supernode. However, in this description of the algorithm, we postpone discussion of this case until Subsection 4.4. Assume therefore that v is a real node. We call a procedure CONNECTTONETWORK(v, T_i) for $i = 1, 2$.

The goal of CONNECTTONETWORK(v, T) is to augment various networks $N_{T'}$ until v is connected to N_T^*. To do this, the procedure first finds a shortest path P_0 in T from v to a site in T, identifies the shortest initial subpath

P of P_0 that ends on a node of N_T, and adds the edges of P to N_T. We are not yet done; P does not necessarily correspond to a connected subgraph of the original graph because it may contain supernodes. Moreover, we have just added such supernodes u to N_T, so the networks N_u corresponding to these supernodes belong to N_T^*. In order to maintain the connectivity invariant, therefore, we must connect the networks N_u to N_T^*. We make these connections recursively using a procedure EXPANDPATH(u, P). This procedure expands P into a real path (i.e. a path in the original graph) by replacing each supernode u in the path with a subpath within T_u that connects a boundary node of T_u to u's network, goes through that network, and comes out again to the boundary of T_u. For technical reasons, EXPANDPATH does not replace the last node of P, so if this last node is a supernode, we use a recursive call to CONNECTTONETWORK to make this part of the path real. Making a path real using EXPANDPATH and CONNECTTONETWORK is illustrated in Figure 3.

Now we give the procedure for CONNECTTONETWORK(v, T). Once again, the basic idea is to find a short path P in T from v to the network N_T, then introduce additional edges to make P correspond to a real path, i.e. a path among the real nodes.

CONNECTTONETWORK(v, T)
Assumption: The node v is a real node enclosed by some node v_0 in the boundary of T.

C1 Let v_0 be the node in T that encloses v.

C2 Let P_0 be a shortest path in T from v_0 to a site s. Let v_r be the first node of P_0 belonging to N_T, and let P be the subpath of P_0 from v_0 to v_r.

C3 Add P to N_T.

C4 Call EXPANDPATH(v, P) to make a real path out of P, except possibly for the last connection.

C5 If the last node v_r in P is a real node, then stop.

C6 Else,

C7 Let T' be the (inactive) tree corresponding to the supernode v_r.

C8 Let v' be the real node by which the last edge of P is incident to v_r.

C9 Recursively call CONNECTTONETWORK(v', T').

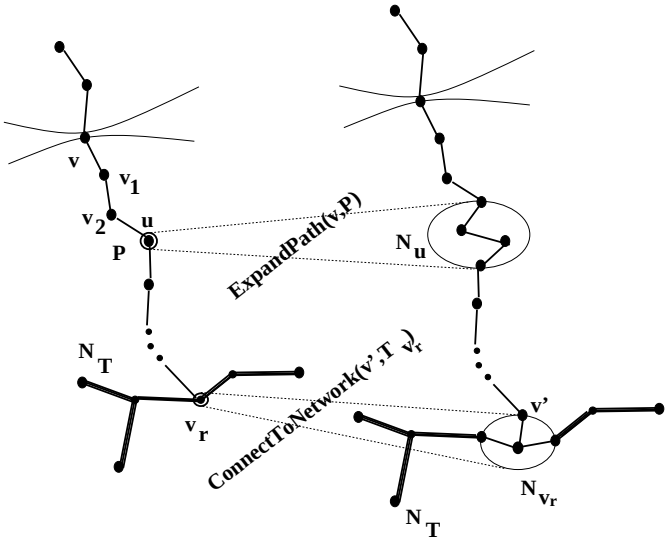

Figure 3: After identifying a path P of $v_1, \ldots, v_r$ to connect v to T, any supernode u in P is expanded by EXPANDPATH(v, P) into a connection via the network tree of u, N_u. If the last node in P, v_r, is a supernode, it is recursively expanded to a path in the original graph by identifying a vertex v' on its boundary and calling CONNECTTONETWORK(v', T_{v_r}) recursively to connect v' to the network N_{v_r}.

The procedure CONNECTTONETWORK uses a subprocedure EXPANDPATH(v, P) to make a real path out of P. For each node v of P except the last, if v is a supernode, we may have to add edges to N_v.

EXPANDPATH(v, P)
Assumption: P is a path in some tree T, whose first node encloses v, which is assumed to be a real node.

E1 Write $P = v_0 e_0 v_1 e_1 \ldots e_{r-1} v_r$.

E2 For $i := 0$ to $r - 1$ do

E3 Let v' be the real node by which e_i is incident to v_i.

E4 **Comment:** We must make a real path in N from v to v'.

E5 If v_i is a supernode then

E6 Let T be the tree corresponding to v_i.

E7 Call CONNECTTONETWORK(v, T).

E8 Call CONNECTTONETWORK(v', T).

E9 **Comment:** Now there is a real path from v to T's network to v'.

E10 Let v be the real node by which e_i is incident to v_{i+1}.

To prove that by using these procedures in the merge, we maintain the connectivity invariant, we would use induction to show the following two statements: The call CONNECTTONETWORK(v, T) introduces edges in N_T^* to connect the real node v to N_T^*. The call EXPANDPATH(v, P) introduces edges in the networks N_{v_i} (for each supernode $v_i \in P$ except the last) so that the edges of P are connected up in N.

4.4 Merging trees whose common node is a supernode

To complete the description of the algorithm, we consider the case in which the node at which trees collide is a supernode, rather than a real node. Let v be a supernode, and suppose trees $T_1, \ldots, T_k$ collide at v at time t. We describe how to merge these trees.

To initialize, let $T = T_1$. For $i = 2$ to k, we merge T_i into T as follows. Since v is on the boundary of T_i, Proposition 4.1 ensures a path of length t from v to a site of T_i. Let e be the first edge on this path, and let v_i be the real node by which e is incident to v. We then call CONNECTTONETWORK(v_i, T) and CONNECTTONETWORK(v_i, T_i). These calls establish a path going through v_i between the network of T and the network of T_i. We then let T be the resulting merged tree, i.e. $T := T \cup T_i$. This completes the merge of T_i into T.

The second invocation, CONNECTTONETWORK(v_i, T_i), needs some elaboration. As we shall see in the next section, the analysis of the algorithm requires that steps E7 and E8 of EXPANDPATH be executed at most once for a given tree T during the course of the algorithm. The first invocation of CONNECTTONETWORK(v_i, T) executes these two steps for the tree T_v corresponding to the supernode v. We must therefore avoid executing these steps in subsequent invocations of CONNECTTONETWORK. Fortunately, the choice of v_i enables us to avoid executing these steps, as we now explain.

Step C2 of CONNECTTONETWORK selects a path P_0 from the supernode v to a site in T_i. By choice of v_i, we can select the path P_0 so that its first edge is incident to the real node v_i in G. P is an initial subpath of P_0. Therefore, when we call EXPANDPATH(v_i, P) in step C4, we omit the iteration $i = 0$ in EXPANDPATH in which P's connection to v_i is made a real connection. This omission avoids re-execution of steps E7 and E8 of

EXPANDPATH on the tree T_v.

5 Proving the performance guarantee of the R-join algorithm

To prove (2) of Subsection 3.1, we shall show that the cost of the feasible network produced by the algorithm is small relative to the number of cuts produced.

At any point in the execution of the algorithm, the *age* of a tree is the number of timesteps the tree grew. Thus the age of an active tree is the current number of elapsed timesteps, while the age of an inactive tree is the number of timesteps that had elapsed when the tree became inactive. We denote the age of a tree T by $age(T)$. We define the *connect-cost* of a call to the subroutine CONNECTTONETWORK as the number of edges added to the network by the routine not including any calls to the routine EXPANDPATH. That is, the cost for a call is the number of edges added in step $C3$, plus the cost of the recursive call in C9. We recursively define the *height* of a node to be 0 if it is a real node and one more than the maximum height of any node it encloses if it is a supernode.

Lemma 5.1 *Steps E7 and E8 of* EXPANDPATH *are executed at most once for a given tree T through the course of the algorithm.*

Proof: Suppose we are about to begin the merging process for a given timestep. Through a series of calls to CONNECTTONETWORK, we build paths P that connect up some trees' networks. The key observation is that for every such path P, constructed in step C2 of CONNECTTONETWORK, every node of P except the last was previously free. (Recall that a *free* node is one that is not contained in any network N_T.) Moreover, since the edges of P are added to the network in step C3, such nodes are subsequently not free. Consequently, each node appears as a non-final node of a path P at most once during the course of the algorithm.

To complete the proof of the lemma, we need only add that a tree T for which Steps E7 and E8 of EXPANDPATH(v, P) are executed corresponds to a non-final node v_i of the path P. □

Lemma 5.2 *The connect-cost of a tree T is at most $age(T)$.*

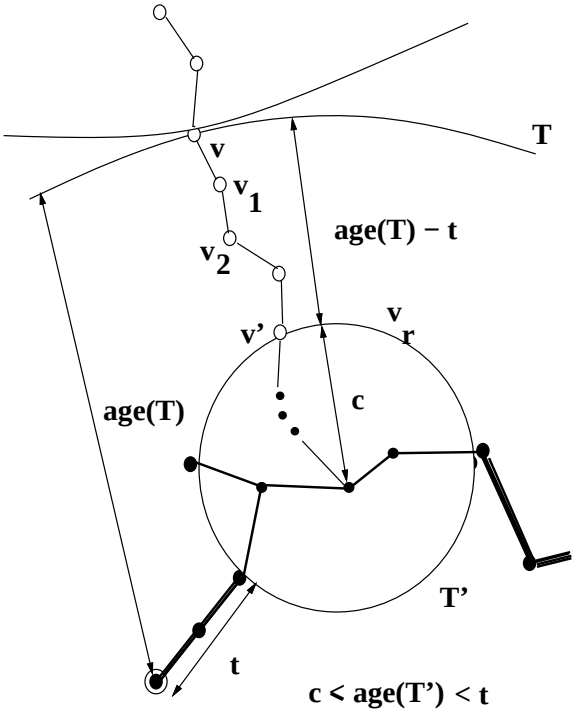

Figure 4: Proof of lemma 5.2 that the cost of a call to CONNECTTONETWORK in the construction of the solution for the tree T is at most $age(T)$.

Proof: We prove it by induction on the height of the nodes on the path from v to N_T. The statement trivially holds if all nodes have height 0, because by Proposition 4.1, v is at distance $age(T)$ from N_T.

Assume that the statement is true for nodes of height at most l. Let P be the path added in step C3, and let v_r be P's final node. By Proposition 4.1, P has at most $age(T)$ edges. Therefore, if v_r is a real node, we are through. Otherwise, v_r corresponds to an inactive tree T'. The proof for this case is illustrated in fig. 4. Let c be the cost of the recursive call CONNECTTONETWORK(v', T') in step C9. By the inductive hypothesis, c is at most $age(T')$, since no node in T' has height more than l. Suppose v_r was added to T after t timesteps. It follows that the number of edges in P is $age(T)-t$. Moreover, $age(T')$ is at most t, since T' was already inactive when v_r was added to T. Hence the total cost of the call to CONNECTTONETWORK

which is $|P| + c$, is at most $age(T) - t + age(T')$, which in turn is at most $age(T)$. □

Define the *expand-cost* of a tree T as the cost of the two calls CONNECTTONETWORK(v, T) and CONNECTTONETWORK(v', T) in steps steps E7 and E8. By Lemmas 5.1 and 5.2, the expand-cost of T is at most $2 \cdot age(T)$. Moreover, by the proof of Lemma 5.1, if the node v corresponding to T remains forever free, then these calls are never made, so the expand-cost of T is zero. We use $ExpandCost(T)$ to denote the expand-cost of T.

When trees $T_1, \ldots, T_r$ merge, the network N_T for the resulting tree T is constructed by taking the union the networks for the T_i's, and then making some calls to CONNECTTONETWORK. We recursively define the cost of T as the sum of the costs of the trees merged to form T, plus the costs of the calls to CONNECTTONETWORK. Thus the cost of a tree T is the number of edges added to create N_T^*, not including edges added in steps E7 and E8 of EXPANDPATH. We denote the cost of T by $Cost(T)$.

We will charge the cost of a tree against the number of cuts assigned to the tree. Recall from the cut-packing algorithm that in each timestep we grow each tree, and assign the corresponding level cut to the tree. Moreover, when trees are merged, their cuts are assigned to the resulting tree. We denote the number of cuts assigned to a tree by $CP(T)$.

Lemma 5.3 *After t timesteps have elapsed, the cost of a tree T is at most $2 \cdot CP(T) - 2 \cdot age(T)$.*

Proof: We shall prove this statement by induction on the number t of elapsed timesteps. When t is 0, the lemma holds trivially. Assume that the statement holds for t. During the $t + 1^{st}$ timestep, each active tree T, is grown by one level, so $CP(T)$ goes up by one, while its age also increases by one. So far, so good. Next, trees are merged. The additional cost incurred in merging $T_1, \ldots, T_r$ to form a tree T is the cost of $2(r-1)$ calls to CONNECTTONETWORK, each at cost at most $age(T)$ by Lemma 5.2. Hence the total cost of T is

$$2(r-1)age(T) + \sum_{i=1}^{r} Cost(T_i)$$

which, by the inductive hypothesis, is at at most $2(CP(T) - age(T))$. □

Now we can bound the size of the feasible network output by our algorithm. The size is the sum, over all inactive trees T of the cost of T plus the expand-cost of T. For any tree T whose node remains free, the expand-cost

is zero. Let us call a tree free if its corresponding supernode is free. Thus we have

$$\begin{aligned} & \text{size of feasible network} \\ &\leq \sum_T Cost(T) + ExpandCost(T) \\ &\leq \sum_{\text{free } T} Cost(T) + \sum_{\text{unfree } T} (Cost(T) + ExpandCost(T)) \\ &\leq 2(\sum_T CP(T) - \sum_{\text{free } T} age(T)) \end{aligned} \tag{5}$$

where the last inequality follows from Lemma 5.3 and our remarks about expand-cost.

Since $\sum_T CP(T)$ is the total number of cuts assigned by the cut-packing algorithm, we have proved a version of (2) with a factor of 2 instead of $2(1-1/k)$. To get the smaller factor, we prove a lower bound on the second sum in (5).

For a tree T, let k_T denote the number of sites that are nodes of T. Define $k_T^* = \sum\{k_{T'} : T \text{ encloses } T'\}$. Similarly, let $CP^*(T) = \sum\{CP(T') : T \text{ encloses } T'\}$.

Lemma 5.4 *For any tree T, $age(T)$ is at least $CP^*(T)/k_T^*$.*

Proof: The key observation is that for any tree T', $CP(T')$ is at most $k_{T'}$ times $age(T')$, since each of the $k(T')$ sites is assigned a maximum of one cut per timestep until $age(T')$ timesteps. If T' is enclosed by T, then $age(T')$ is at most $age(T)$, so we have

$$\begin{aligned} CP^*(T) &= \sum\{CP(T') : T \text{ encloses } T'\} \\ &\leq \sum\{k_{T'} : T \text{ encloses } T'\} age(T) = k_T^* age(T) \end{aligned}$$

□

We use Lemma 5.4 to get our lower bound on $\sum\{age(T) : T \text{ free}\}$. Let $k^* = \max\{k_T^* : T \text{ free}\}$. Then by Lemma 5.4, for each free tree T, $age(T) \geq CP^*(T)/k^*$. Since each tree is enclosed by some free tree, $\sum\{CP^*(T) : T \text{ free}\}$ is the total number CP of cuts assigned. Hence

$$\sum\{age(T) : T \text{ free}\} \geq CP/k^* \tag{6}$$

Substituting into (5) and replacing k^* by k, the total number of sites, gives (2) and completes the proof of Theorem 1.3.

6 Further directions for research

Two important variants of the basic Steiner tree problem in networks are the node Steiner problem [34], in which nodes are assigned costs, and the directed Steiner tree problem, in which the input graph is directed and one seeks a directed tree as the solution. It is an open problem to find good approximation algorithms for either problem. It was observed by Berman [5] that the set cover problem is reducible to the node Steiner problem via an approximation-preserving transformation. Khuller [20] made an analogous observation concerning the directed Steiner tree problem. In fact, Segev [34] gave an approximation-preserving reduction from the node Steiner problem to the directed Steiner problem. In view of Lund and Yannakakis' recent result showing that the set cover problem cannot be approximated by a factor smaller than logarithmic [28], it is natural to ask whether there are logarithmic-factor approximation algorithms for the node and directed Steiner problems. We have recently discovered such an algorithm for the former problem [24].

Acknowledgements

Thanks to Piotr Berman, Marshall Bern, Michel Goemans, Samir Khuller, Balaji Raghavachari, Arie Segev, and Richard Wong. Thanks also to the referees for a careful reading and helpful suggestions.

References

[1] A. Agrawal, P. Klein and R. Ravi, "When trees collide: an approximation algorithm for the generalized Steiner tree problem on networks," Technical Report CS-90-32, Brown University (1990).

[2] Y. P. Aneja, "An integer linear programming approach to the Steiner problem in graphs", *Networks*, vol. 10 (1980) 167-178.

[3] S. Arora and S. Safra, "Probabilistic checking of proofs: a new characterization of NP," *Proc., 33rd Symposium on Foundations of Computer Science* (1992), pp. 2-13.

[4] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, "Proof verification and hardness of approximation problems," *Proc., 33rd Symposium on Foundations of Computer Science* (1992), pp. 14-23.

[5] P. Berman, personal communication (1991).

[6] P. Berman and V. Ramaiyer, "Improved approximations for the Steiner tree problem," *Proc., 3rd Annual ACM-SIAM Symposium on Discrete Algorithms* (1992), pp. 325-334.

[7] R. T. Chien, "Synthesis of a communication net," IBM *J. Res. Develop.*, Vol. 3 (1960), pp. 311-320.

[8] C. J. Colbourn, *The Combinatorics of Network Reliability*, Oxford University Press (1987).

[9] C. J. Colbourn, "Edge-packings of graphs and network reliability", *J. Discrete Math., vol. 72* (1988) pp. 49-61.

[10] J. Edmonds, and E. L. Johnson, "Matching, Euler tours and the Chinese postman", *Math. Programming, vol. 5* (1973) pp. 88-124.

[11] C. El-Arbi, "Une heuristique pour le problem de l'arbre de Steiner", *R.A.I.R.O. Operations Research, vol. 12* (1978), pp. 207-212.

[12] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy, "Approximating clique is almost NP-complete," *Proc., 32nd Symposium on Foundations of Computer Science* (1991), pp. 2-12.

[13] R. E. Gomory and T. C. Hu, "Multi-terminal network flows," *J. Soc. Indust. Appl. Math.* Vol. 9, No. 4 (1961), pp. 551-570.

[14] M. X. Goemans, and D. J. Bertsimas, "Survivable networks, linear programming relaxations and the parsimonious property," *Mathematical Programming* 60 (1993), to appear.

[15] H. N. Gabow, M. X. Goemans, and D. P. Williamson, "An efficient approximation algorithm for the survivable network design problem," *Proceedings of the 3rd MPS Conference on Integer Programming and Combinatorial Optimization* (1993), pp. 57-74.

[16] M. X. Goemans and D. P. Williamson, "A general approximation technique for constrained forest problems," *Proc., 3rd Annual ACM-SIAM Symposium on Discrete Algorithms* (1992), pp. 307-315.

[17] F. K. Hwang and D. S. Richards, "Steiner tree problems," *Networks*, Vol. 22, No. 1, pp. 55-90 (1992).

[18] A. Jain, "Probabilistic analysis of an LP relaxation bound for the Steiner problem in networks", *Networks, vol. 19* (1989), pp. 793-801.

[19] R. M. Karp, "Reducibility among combinatorial problems", in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*. Plenum Press, New York (1972) pp. 85-103.

[20] S. Khuller, personal communication (1991).

[21] P. N. Klein, "A data structure for bicategories, with application to speeding up an approximation algorithm," submitted to *IPL* (1993).

[22] P. N. Klein, S. Rao, A. Agrawal, and R. Ravi, "An approximate max-flow min-cut relation for multicommodity flow, with applications," to appear in *Combinatorica*.

[23] M. X. Goemans, A. V. Goldberg, S. Plotkin, D. Shmoys, É. Tardos, and D. P. Williamson, "Improved approximation algorithms for network design problems," *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms* (1994), pp. 223-232.

[24] P. N. Klein and R. Ravi, "A nearly best-possible approximation algorithm for node-weighted Steiner trees," *Proceedings, 3rd Symposium on Integer Programming and Combinatorial Optimization* (1993), pp. 323-332.

[25] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for Steiner trees", *Acta Informatica, vol. 15* (1981), pp. 141-145.

[26] J. Krarup, "The generalized Steiner problem," Unpublished note (1978).

[27] L. Lóvasz, *An Algorithmic Theory of Numbers, Graphs and Convexity*, SIAM, Philadelphia (1986).

[28] C. Lund and M. Yannakakis, "On the hardness of approximating minimization problems," *Proc., 25th ACM Symposium on Theory of Computing* (1993), pp. 286-293.

[29] K. Mehlhorn, "A faster approximation algorithm for the Steiner problem in graphs," *Information Processing Letters, vol. 27(3)* (1988), pp. 125-128.

[30] M. Minoux, "Network synthesis and optimum network design problems: models, solution methods and applications," *Networks, vol. 19* (1989) pp. 313-360.

[31] J. Plesnik, "A bound for the Steiner tree problem in graphs," *Math. Slovaca, vol. 31* (1981) pp. 155-163.

[32] R. Ravi and P. N. Klein, "When cycles collapse: a general approximation technique for constrained 2-connectivity problems," *Proc., 3rd Symposium on Integer Programming and Combinatorial Optimization* (1993), pp. 39-55.

[33] V. J. Rayward-Smith, "The computation of nearly minimal Steiner trees in graphs," *Int. J. Math. Educ. Sci. Tech., vol. 14* (1983), pp. 15-23.

[34] A. Segev, "The node-weighted Steiner tree problem," *Networks, vol. 17* (1987) pp. 1-17.

[35] R. Sridhar and R. Chandrasekaran, "Integer solution to synthesis of communication network," Technical Report, The university of Texas at Dallas (1989).

[36] K. Steiglitz, P. Weiner, and D. J. Kleitman, "The design of minimum-cost survivable networks," *IEEE Trans. on Circuit Theory*, CT-16, 4 (1969) pp. 455-460.

[37] G. F. Sullivan, "Approximation algorithms for Steiner tree problems," Tech. Rep. 249, Dept. of Comp. Sci., Yale Univ. (1982).

[38] H. Takahashi, and A. Matsuyama, "An approximate solution for the Steiner problem in graphs," *Math. Japonica*, vol. 24 (1980), pp. 573-577.

[39] L. G. Valiant, "The complexity of enumeration and reliability problems", *Siam J. Comput.*, vol. 8 (1979), pp. 410-421.

[40] D. P. Williamson, M. X. Goemans, M. Mihail, and V. V. Vazirani, "A primal-dual approximation algorithm for generalized Steiner network problems," *Proceedings of the 25th ACM Symposium on Theory of Computing* (1993), pp. 708-717.

[41] P. Winter, "Steiner Problem in Networks: A Survey", *BIT 25* (1985), pp. 485-496.

[42] P. Winter, "Generalized Steiner Problem in Outerplanar graphs," *Networks* (1987), pp. 129-167.

[43] R. T. Wong, "Worst-case analysis of network design problem heuristics", *SIAM J. Alg. Disc. Math.*, vol. 1 (1980), pp. 51-63.

[44] R. T. Wong, "A dual ascent approach for Steiner tree problems on a directed graph", *Math. Program.*, vol. 28 (1984) pp. 271-287.

[45] A. Z. Zelikovsky, "The 11/6-approximation algorithm for the Steiner problem on networks," *Algorithmica*, 9, pp. 463-470.