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

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 $2 \lg R$ 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.

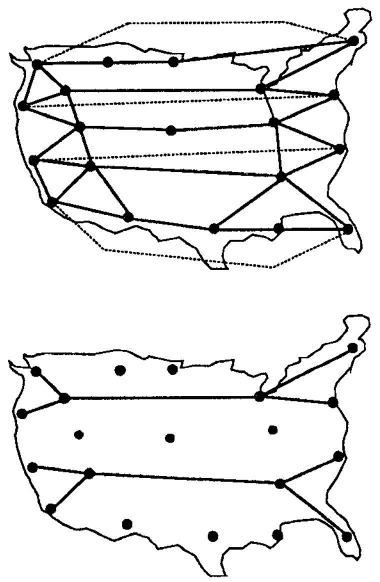
Figure 1: An instance of the unweighted network design problem and its solution. Solid edges correspond to unit-cost links; dotted edges connect site pairs.

1 Introduction: On designing a network

Here's the scenario: You're in the business of providing communication channels to your customers. You have a set of clients with communication requirements; each client has specified a pair of cities between which the client must have communication capabilities. In front of you is the AT&T[2] price list, which gives prices for constructing communication links between various cities. Each link built has essentially unbounded capacity. Your job is to select a minimum-cost collection of communication links that can accommodate all your clients' communication requirements. The network you must construct need not be connected; all that is needed is that every client's pair of cities be connected through your network.

Unfortunately, your job's NP-complete. Even the very special case in which each client wants to connect some city to New York is the dreaded Network Steiner Tree Problem, one of Karp's original list of NP-complete problems [7]. Since you only have polynomial time to spare, you are forced to turn to approximation algorithms. Consulting the literature [1, 4, 9, 12, 13, 15, 17, 23], you find a variety of good approximation algorithms for the Steiner tree problem in networks. Each of these algorithms constructs a nearly minimum-cost tree connecting together a given set of cities.

[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 .

[2]Disclaimer: The authors are in no way connected to AT&T.

However, none of the algorithms addresses the more general case in which each client can specify an arbitrary pair of cities, and the output 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 this network design problem. Our algorithm is guaranteed to output a network that is within a factor of $2-2/k$ of optimal, where k is the number of cities specified by clients. It generalizes the known approximation algorithms for the Network Steiner Tree Problem, in that its worst-case performance on instances of this special case is asymptotically at least as good as those of other known algorithms.

Theorem 1.1 *There is an $O(m \log m)$ algorithm for finding a network of cost at most $2-2/k$ times optimal, where k is the number of cities specified by clients, and m is the number of edges in the input graph.*

In fact, we can generalize our result to include costs on nodes as well as edges. To our knowledge, no approximation algorithms that allow node-weighted graphs were known, even for the Steiner tree special case, but they fit easily into our framework. We can handle hypergraphs with weighted hyperedges as well, through a simple transformation to the node-weighted case.

The Generalized Steiner Problem in Networks

In fact, the algorithm can be generalized to handle a problem involving certain redundancy requirements. 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 network is then allowed to use multiple links connecting a pair of nodes. This problem is the generalized Steiner problem in networks.

Theorem 1.2 *There is an $O(m \log m \log R)$ algorithm for finding a network of cost at most $(2-2/k)\lceil \log_2(R+1)\rceil$ times optimal, subject to arbitrary edge-connectivity requirements r_{ij}, where R is the largest requirement value.*

Packing cuts, with application to network reliability

Well, our algorithm's just made your old job obsolete. So you turn to a new job: evaluating network reliability. You are presented with an existing network and the same list of clients, each specifying a pair of cities. Now you would like to determine how likely it is that random failure of communication links renders some of your clients' requirements unsatisfiable.[2] Assuming link failures are independent, determining the probability that the surviving links can serve all clients' requirements is a generalization of the notorious $\#P$-complete problem [19] called Network Reliability. No approximation algorithms are known.

However, one powerful and useful heuristic for estimating two-terminal and k-terminal reliability [2] 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 [2] 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.

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 network design problem to the cut-packing problem.

Theorem 1.3 *The minimum size of a network design is approximately equal to one-half the maximum size of a collection of cuts, where each cut separates some client's cities, 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 client-specified cities.*

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 relaxation of these programs provides a good approximation to both combinatorial quantities. Moreover, the factor of $2-2/k$ is existentially tight for the example of a k-cycle given by Goemans and Bertsimas [5].

2 Related work

2.1 The Steiner tree problem in networks

There has been volumes of work done on the Steiner tree problem in networks, including proposed solution methods, computational experiments, heuristics, probabilistic

[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 [8].

and worst-case analyses, and algorithms for special classes of graphs. Winter [20] recently surveyed this body of work.

Karp [7] showed that the problem is NP-complete. Takahashi and Matsuyama [17], Kou et al. [9], El-Arbi [4], Rayward-Smith [13], Aneja [1], and Wong [23] 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 cities that need to be connected up (called Z-vertices in [20]). One algorithm, proposed by Plesnik [12] and by Sullivan [15], performs somewhat better. In computational experiments, these heuristics generally perform considerably better than the worst-case bound predicts. Jain [6] 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. Segev [14] formulated the node-weighted Steiner problem, and proposed branch-and-bound techniques for its solution. We know of no approximation algorithms for this problem.

2.2 The generalized Steiner problem in networks

The *generalized Steiner problem in networks*, as originally formulated by Krarup (see [21]), 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.

To our knowledge, no approximation algorithms for the generalized Steiner problem are known. There have been papers addressing finding an exact solution and algorithms for special classes of graphs [21, 20]

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 very recent work, Goemans and Bertsimas [5] considered a special case of the generalized Steiner problem in networks, which they called *the survivable network design problem*. Instead of arbitrary edge-connectivity requirements, the input includes an assignment of integers r_i to nodes. The goal is to find a minimum-cost network satisfying edge-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. Thus this restriction was a result of the lack of a known approximation algorithm for the case of $r_{ij} \in \{0, 1\}$. We remedy this lack.

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 solution, which we call a network design, 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 a duality between network design and packing 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 $\{s_i, t_i\}$. We will denote the cardinality of the set of sites by k.

Let N be any network design 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 network design 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 network design is at least one-half the maximum size of a 2-packing of requirement cuts.*

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

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

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

For comparison, Edmonds and Johnson [3] 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 network design and a 2-packing, such that the first has size at most $(1 - 1/k)$ times the second. It follows that the network design is approximately minimum and the 2-packing is approximately maximum, to within a factor of $2(1 - 1/k)$. The algorithm takes time $O(m\alpha(m, m))$, where m is the number of edges in the input graph G.

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

- Any minimal network design in G corresponds to a network design 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 network design 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 nonnegative 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 3.1 *The minimum-cost of a network design 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 network design, we don't use this refinement technique but instead use a more direct approach, and thereby achieve the time bound stated in Theorem 1.1. In fact, this algorithm proves Theorem 3.1 for rational (non-integral) costs c as well, where we define "fractional" packings in the usual way (see, e.g. [10]). By considering fractional packings of node-cuts instead of edge-cuts, we can prove a node-weighted version of Theorem 3.1, and give an algorithm for the node-weighted case of the network design problem. In Section 6, we briefly allude to some of the issues arising in implementing these algorithms.

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 network design. 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 [5]. 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 problem. Use a standard heuristic to find an approximately optimal solution, and combine the solutions to 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

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

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

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, a problem that has steadfastly resisted approximation [2]. However, we can 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 don't 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 a network design. 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.

4.1 Overview

We grow breadth-first search trees from the sites, accumulating cuts as we go. 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 we grow trees, we build networks spanning the trees' sites. 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

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 4.1 *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 4.1 to show that all the cuts found by the algorithm are edge-disjoint.

Corollary 4.1 *No edge belongs to a level cut of more than one tree.*

Proof: By Proposition 4.1 and Lemma 4.1, 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 union-ing their nodes and 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.

4.3 The network design algorithm

The basic approach to building a network design 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

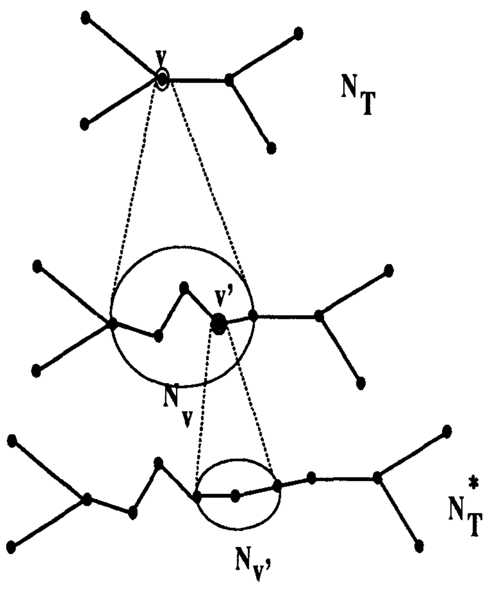

Figure 2: The network N_T corresponds in a natural way to a subgraph N_T* of the original graph. To obtain N_T* from N_T, replace each supernode v in N_T with the subgraph N_v, and recurse on the supernodes in N_v, like v' in the above figure.

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. We say v ***encloses*** v' if either $v = v'$ or v immediately encloses some node v'' that encloses v'. 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. In the site-inlusion invariant only speaks of 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 a valid network design—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 unioning 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 omit discussion of this case. 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 v to N_T, so the networks N_v corresponding to these supernodes belong to N_T^*. In order to maintain the connectivity invariant, therefore, we must connect the networks N_v to N_T^*. We make these connections recursively using a procedure EXPANDPATH(v, P).

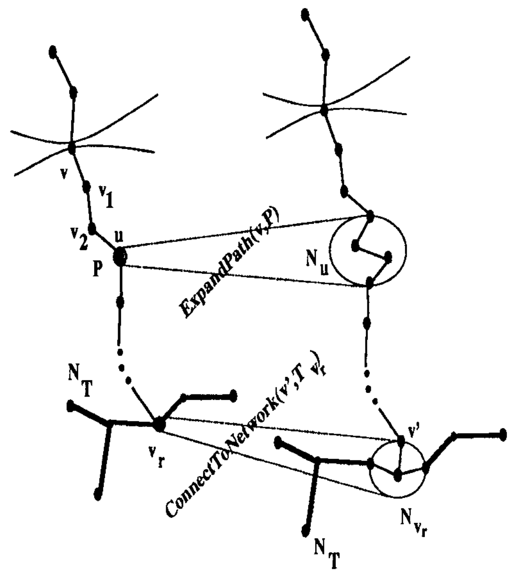

Figure 3: After identifying a path P of $v_1, \ldots, v_r$ to connect v to T, any supernode u in P is expanded by ExpandPath(v,P) into a connection via the network design tree of u, N_u. If the last node in P, v_r, is a supernode, it is recursively expanded to a path in the original graph by identifying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

This procedure expands P into a real path (i.e. a path in the original graph) by replacing each supernode v in the path with a subpath within T_v that connects a boundary node of T_v to v's network, goes through that network, and comes out again to the boundary of T_v. 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 fig. 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$(v', T')$.
```

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 that node is a supernode, we may have to add edges to N_v and, recursively.

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

5 The proof

To prove (2) of Subsection 3.1, we shall show that the cost of the network design 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 $age(T)$.

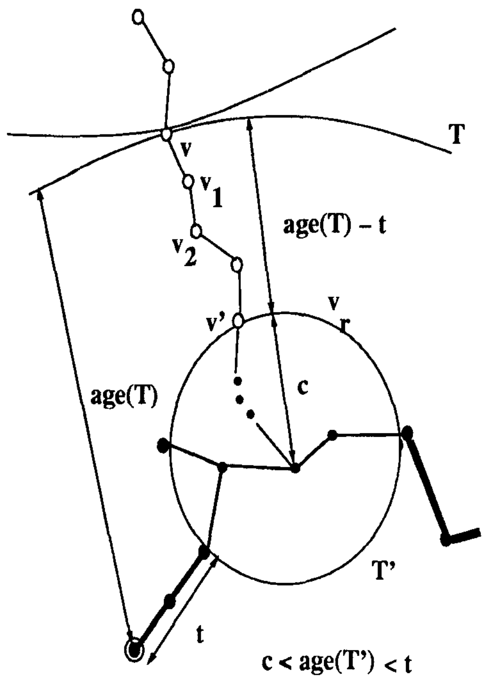

Figure 4: Proof of lemma 5.2 that the cost of a call to ConnectToNetwork in the construction of the Network Design solution for the tree T is at most $Age(T)$.

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. 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 $age(T)$.*

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 $2.age(T)$. 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 union-ing 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 $2CP(T) - 2 \cdot age(T)$.*

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 network design output by our algorithm. The size is just the sum, over all inactive trees T when the algorithm terminates, 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

size of network design

$$\begin{aligned}
&\leq \sum_T Cost(T) + ExpandCost(T) \\
&\leq \sum_{\text{free } T} Cost(T) + \sum_{\text{unfree } T} (Cost(T) + ExpandCost(T)) \\
&\leq 2(\sum_T CP(T) - \sum_{\text{free } T} age(T)) \qquad (3)
\end{aligned}$$

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 (3).

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

$$\sum\{age(T) : T \text{ free}\} \geq CP/k^* \qquad (4)$$

Substituting into (3) and replacing k^* by k, the total number of sites, gives (2) and completes the proof of the theorem.

6 Implementation issues

In this section, we touch on some of the issues arising in obtaining algorithms for the unweighted and weighted cases. Implementation of the unweighted algorithm uses union-find data structures. Implementation of the weighted algorithm is similar, but replaces one of the union-find data structures with a mergeable heap data structure.

6.1 The unweighted case

First we describe the implementation of the cut-packing algorithm, then we show how data structures created during cut-packing can be used for network design.

For purposes of the algorithm, it is convenient to copy the input graph, replacing each edge with two oppositely directed arcs. We will be removing arcs from the copy as the algorithm progresses. A top-level tree or node is one not contained in any tree. It is also convenient to represent top-level real nodes, top-level supernodes, and active trees uniformly, so each is considered as if it were a supernode. Thus for each we maintain an *arc set* consisting of its outgoing arcs, a *node set* consisting of all its enclosed real nodes, and an *active site set* consisting of all its enclosed active sites. Initially, the only nodes are real nodes. A real node's arc set consists of its outgoing arcs, its node set consists of itself, and its site either consists of itself, if it is a site, or is empty otherwise. Since we maintain these sets only for top-level nodes or trees, and since every real node is in exactly one top-level node or tree, these sets remain disjoint throughout the algorithm.

A timestep consists of growing each active tree (a tree with a nonempty active site set) by one step. We consider the active trees in any order. Growing a tree consists of the following substeps. Scan the tree's outgoing arcs in order to find the set U of real nodes at distance 1 from the tree, and remove these scanned arcs from the graph. Next, use the FIND operation on the nodes in U to find the set W of top-level nodes or trees containing nodes of U. That is, if U contains some real node contained in either a supernode or active tree, then W will contain that supernode or active tree. This substep includes the case where the tree being grown has collided with another tree grown previously during the same timestep.

Next, augment the node set associated with the tree being grown by the union of the node sets of elements of W. Then construct the arc set of the active tree by taking the union of the arc sets of elements of W. Finally, modify the active site set of the tree being grown by taking the symmetric difference with the active site sets of all the elements of W. If the symmetric difference turns out to be empty, designate the tree being grown as *inactive*.

The number of operations in the above cut-packing algorithm is proportional to the number of arcs, so the time required is $O(m\alpha(m))$, where m is the number of edges, using Tarjan's analysis of union-find [18].

While the above cut-packing algorithm is progressing, it should construct two data structures to enable the network design to take place. First, the algorithm builds a *enclose tree* to record the enclosing relation among nodes and trees. The enclose tree contains one leaf for every real node and one internal vertex for every supernode and every tree, such that v is an ancestor of w if and only if v encloses w. Second, every time the cut-packing algorithm grows a tree, the top-level nodes newly added to the tree are assigned *arc pointers* to the arcs by which they were reached. Thus by using these pointers, one can traverse a shortest path from a node in a tree to a site in the same tree.

These two data structures enable the first two steps of CONNECTTONETWORK(v, T) to be implemented so that the total time required for network design is $O(m\alpha(m))$. We add some cryptic remarks in justification of this claim, and postpone a careful description to the final paper. Given a real node v and a tree T enclosing v, one can trace up the enclose tree to find the node v_0 in T that encloses v. Namely, v_0 is the child of T that is an ancestor of v. During the trace, one determines all the intermediate ancestors of v; this sequence of supernodes is stored temporarily with v_0 to enable EXPANDPATH to recurse. Once v_0 has been found, its arc pointer can be used to find the first arc in a shortest path P_0 from v_0 to a site. That arc points to a real node v' on which this procedure is repeated, and so on.

6.2 The weighted case

The basic algorithm to handle the weighted case is essentially the same as that for the unweighted case, except that Dijkstra's shortest-path algorithm is used in place of breadth-first search. It is useful to imagine that in the cut-packing algorithm for the weighted case, each growing tree continuously "consumes" all its outgoing arcs at the same rate until some edge is completely consumed, at which time things must be updated. The implementation can be modified accordingly by substituting mergeable heaps, e.g. binomial trees, for the arc sets. Note that, in constrast to Dijkstra's algorithm, the heaps contains arcs rather that nodes. The key associated with an arc is the amount of the corresponding edge yet to be consumed, plus the current timestep.

We must make one additional modification. Because an edge can be consumed from both ends at the same time (usually from different trees), we must partition a tree's outgoing arcs into two heaps, one for the arcs whose edges are being consumed from only one end, and one for the arcs whose edges are being consumed from both ends. The latter arcs are essentially being consumed twice as fast, so in determining the next outgoing edge to be completely consumed, we compare the minimum key in the first heap to half the minimum key in the second heap. To maintain these two heaps, when some tree begins to consume an arc, it checks whether the oppositely oriented arc is already being consumed. If so, it notifies the tree consuming that arc, and the arcs are moved into the appropriate heaps.

7 Remarks

We described the algorithm above for integral edge weights. The algorithm extends naturally to include node weights. When node and edge weights are allowed, a set of nodes and edges separating a requirement form a requirement cut. An assignment of nonnegative rationals to such cuts constitutes a "fractional" cut-packing; a c-packing in this case is a packing in which the sum of the rationals assigned to any edge or node is at most its cost. The notion of fractional cut-packing lets us extend our results to rational node and edge weights. Moreover, by reducing the problem in a hypergraph to the problem in a graph with node and edge weights, we can obtain similar guarantees for the generalized Steiner problem in hypergraphs.

It remains an open question if finding a maximum $2c$-packing of cuts in a graph is NP-complete. Obtaining an approximation algorithm for the directed version of the generalized Steiner problem remains an important open problem.

Acknowledgements

Thanks to Marshall Berne, Michel Goemans, Arie Segev, and Richard Wong.

References

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

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

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

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

[5] M. X. Goemans, and D. J. Bertsimas, "Survivable Networks, Linear Programming Relaxations and the Parsimonious Property", OR 216-90, Center for Operations Research, MIT (1990).

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

[7] 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.

[8] P. N. Klein, A. Agrawal, R. Ravi, and S. Rao, "Approximation through multicommodity flow", *31st Annual Symp. on Foundations of Comp. Sci.*, (1990), pp. 726-737.

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

[10] L. Lóvasz, "An algorithmic theory of numbers, graphs and convexity". SIAM, Philadelphia (1986).

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

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

[13] 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.

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

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

[16] H. Suzuki, T. Akama, and T. Nishizeki, "Finding Steiner forests in planar graphs", *1st Ann. ACM-SIAM Symp. on Disc. Alg.* (1990), pp. 444-453.

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

[18] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm", *J. Assoc. Comput. Mach., vol. 22* (1975), pp. 215-225.

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

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

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

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

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