The Directed Steiner Network Problem is Tractable for a Constant Number of Terminals

by

Jon Feldman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2000

Certified by . . . .
David R. Karger
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Directed Steiner Network Problem is Tractable for a Constant Number of Terminals

by

Jon Feldman

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

We consider the DIRECTED STEINER NETWORK problem, or the POINT-TO-POINT CONNECTION problem. Given a directed graph G and p pairs $\{(s_1, t_1), \ldots, (s_p, t_p)\}$ of nodes in the graph, one has to find the smallest subgraph H of G that contains paths from s_i to t_i for all i. The problem is NP-hard for general p, since the DIRECTED STEINER TREE problem is a special case. Until now, the complexity was unknown for constant $p \geq 3$. We prove that the problem is polynomially solvable if p is any constant number, even if nodes and edges in G are weighted and the goal is to minimize the total weight of the subgraph H. In addition, we give an efficient algorithm for the STRONGLY CONNECTED STEINER SUBGRAPH problem for any constant p, where given a directed graph and p nodes in the graph, one has to compute the smallest strongly connected subgraph containing the p nodes.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgments

I would like to thank Matthias Ruhl for working with me on this result. We would like to thank David Karger for helpful suggestions, and Andras Frank for asking about the 2-SCSS problem, which started our research on this topic. We also thank Marshall Bern, Yevgeniy Dodis, John Dunagan and Matt Levine for their comments.

Chapter 1

Introduction

The Steiner Problem is one of the classic problems of computational geometry and theoretical computer science, and is fundamental to the study of network design. The basic problem is very simple to state: given a set of points, what is the most efficient way to connect them? However, the problem is not as simple to solve. In fact, the different varieties of Steiner problems have generated volumes of work over the years.

The Euclidean Steiner Problem is the oldest problem in this family. In fact, Fermat (1601 - 1665) asked the question: given three points on the plane, find the point from which the sum of the distances to the three points is minimum. In 1934, Jarník and Kössler [JK34] first posed the Euclidean Steiner Problem: given n points in the plane, find the smallest network which connects them, using new points if you wish. Courant and Robbins [CR41] also posed the problem in their book "What is Mathematics?" and referred to it as the Steiner Problem, after the mathematician J. Steiner, who was interested in the problem. According to the survey by Hwang, Richards and Winter [HRW92], the popularity of this book was the reason that this family of network problems still bears his name.

We will focus our study on a more combinatorial version of Steiner's problem. In the first volume of the journal *Networks* in 1971, Hakimi [Hak71] gave this graph-theoretic version (independently posed by Levin [Lev71]):

STEINER TREE (ST): Given an undirected graph $G = (V, E)$, with weights on the edges, and a subset of *terminals* $T \subseteq V$, find a minimum weight connected subgraph H of G that contains the terminals.

It's not hard to see that this subgraph will form a tree; if there was a cycle, then one of the edges in the cycle could be removed, and all the terminals would still be connected. Thus this problem is referred to as the Steiner Tree problem in graphs. This formulation predated the theory of NP-completeness, but Hakimi's paper stated that instances of CLIQUE could be reduced to the Steiner Tree Problem. So, when the theory emerged, the Steiner Tree Problem in Graphs was among the first identified as NP-hard.

Hakimi also suggested a generalization to directed graphs:

Steiner Equivalent Network (SEN): Given a directed graph $G = (V, E)$, with weights on the edges, and a set of terminals $T \subseteq V$, find a minimum weight subgraph H of G that includes paths between every pair of terminals, provided such a path exists in G.

This is more general than the Steiner Tree Problem in undirected graphs, since an instance of ST can be modeled by an instance of SEN by turning the undirected edges into two directed edges with the same weight. However, this is not as general as it gets. In this thesis we address one of the most general Steiner problems, the Directed Steiner Network problem, also called the Point-to-Point Connection problem.

Directed Steiner Network (p-DSN): Given a directed graph $G = (V, E)$, with weights on the edges, and p pairs of nodes in the graph $\{(s_1, t_1), \ldots, (s_p, t_p)\}$, find the minimum weight subgraph H of G that contains paths from s_i to t_i for $1 \leq i \leq p$.

Clearly this problem is more general than Hakimi's Steiner Equivalent Network (SEN) problem. Given an instance G of SEN, we simply make every connected pair of nodes (s, t) in G into a terminal pair (s, t).

There are two parameters indicating the "size" of a p-DSN instance. The first parameter is n, the size of the input graph G. The second parameter is p, the number of terminal pairs. When p is allowed to an arbitrary quantity, this problem is NP-hard, since it is more general than Steiner Tree. The case $p = 2$ was solved in 1992 by Li, McCormick and Simchi-Levi [LMS92]. However, when p is a fixed quantity greater than two, the question of whether there exists a polynomial-time algorithm for p-DSN was open until now.

We can understand the difficulty of this question by looking at other parameterized problems. There are many parameterized problems that yield polynomial-time algorithms when the parameter is two, but become NP-hard when the parameter becomes three. One classic example is k-Satisfiability, where the parameter k is the number of literals in each clause of a boolean formula in conjunctive normal form [GJ79]. Another example is Chromatic Number, where the goal is to color each vertex of an input graph using k different colors, such that no two vertices of the same color are adjacent. This problem is solvable in polynomial time if $k = 2$, but NP-hard if $k \geq 3$ [GJ79]. On the other hand, there are some parameterized problems that yield simple polynomial-time algorithms when the parameter is any constant. A good example is k-CLIQUE, where looking for a constant-sized clique can be done by simply trying all possibilities. The p-DSN problem does not immediately fit into this category. Unlike Clique, when the parameter p is constant, the *solution* does not necessarily have constant size; thus a brute-force algorithm does not solve p-DSN.

In this thesis, we give the first polynomial-time algorithm for the case where p is any fixed constant, resolving this open question. This result is joint work with Matthias Ruhl [FR99].

1.1 Equivalent Digraphs and Strongly Connected Components

The Steiner Equivalent Network (SEN) problem posed by Hakimi has many connections to p-DSN. The most studied version of the problem is the case where $T = V$, where the problem becomes the following:

Minimum Equivalent Digraph (MED): Given a directed graph $G = (V, E)$, with weights on the edges, find a minimum weight subgraph H of G that includes paths between every pair of nodes, provided such a path exists in G.

Moyles and Thompson [MT69] show that in any instance of MED the input graph G can be decomposed as follows. Take every strongly connected component, and contract it to a single vertex; what you are left with is a directed acyclic graph (dag) G'. Now find the MED of G', and of each of the contracted components. The resulting edge set will be an MED of G.

Finding an MED of a dag is simple [Hsu75]; it is the set of edges (u, v) for which there is no other path from u to v. However, finding an MED of a strongly connected component is NP-hard [GJ79], thus most of the difficulty of the MED problem is contained in the special case where G is strongly connected. Khuller, Raghavachari and Young [KRY95] use this decomposition to give an polynomial-time approximation algorithm for MED with a performance guarantee of about 1.64 and a linear time algorithm with a performance guarantee of 1.75.

We find that an analogous decomposition also applies to the Directed Steiner Network problem, and this observation lies at the heart of our algorithm. As with MED, the difficulty of the p-DSN problem is contained within a special case of p-DSN for strongly connected components:

Strongly Connected Steiner Subgraph (p-SCSS): Given a directed graph $G = (V, E)$, and p terminal vertices $\{s_1, \ldots, s_p\}$ in V, find the smallest strongly connected subgraph H of G that contains $s_1, \ldots, s_p$.

In this thesis we also give the first polynomial-time algorithm for p-SCSS for any constant p, and use it as a subroutine in our algorithm for p-DSN.

1.2 Previous Results

Constant number of terminals When $p = 1$, there is only one terminal pair (s, t), so the solution is simply the shortest path from s to t. Finding a shortest path in a directed graph is well-known problem solvable in polynomial time [CLR92].

When $p = 2$, the problem becomes a bit more difficult, but yields a fairly simple solution. To motivate the solution for the general case, we present a solution for $p = 2$ in Chapter 2. This problem was first solved by Li, McCormick and Simchi-Levi

[LMS92], who called the problem the POINT-TO-POINT CONNECTION problem. The running time of their algorithm is $\mathcal{O}(n^5)$. They also state the case for all $p \geq 3$ as an interesting open question.

Natu and Fang [NF95] [NF97] improved the running time for the case where $p = 2$ first to $\mathcal{O}(n^4)$, and then to $\mathcal{O}(mn+n^2 \log n)$. They also present an algorithm for $p = 3$, and conjecture that a variant for their algorithm works for all constant p. In Appendix A.1 we provide what we believe to be a counterexample to the correctness of their algorithm for $p = 3$, and thus to their conjecture.

Arbitrary Terminals There has been some work on the approximability of p-DSN for arbitrary p. The best positive result obtained so far is by Charikar et al [CCC+98], who achieve an approximation ratio of $\mathcal{O}(p^{2/3} \log^{1/3} p)$ for any p. On the negative side, Dodis and Khanna [DK99] prove that p-DSN is $\Omega(2^{\log^{1-\varepsilon} p})$-hard; that is, unless $P = NP$, no algorithm exists for p-DSN that has an approximation ratio of $\Omega(2^{\log^{1-\varepsilon} p})$.

The method of Charikar et al [CCC+98] seems to be fundamentally different than the method we give for the fixed-parameter version. Their algorithm is based on greedily finding low-cost 'bunches' of vertices that connect a certain subset of the terminal pairs. A bunch has a very simple structure; it consists of a path from a node u to a node v, and shortest paths from each of the sources to u and from v to each of the terminals. In this thesis, we show much more complex structural properties of optimal solutions of p-DSN than the bunches described above. These properties also exist for arbitrary p (but unfortunately cannot easily be found in polynomial time). It is our hope that the ideas presented here can be used to obtain a better approximation ratio for an arbitrary number of terminals.

1.3 Related Problems

1.3.1 Steiner Tree

The original STEINER TREE problem in undirected graphs has received the most attention among combinatorial versions of Steiner's problem. The problem is NP-hard, so work has focused on obtaining approximation algorithms with good performance guarantees. Hwang, Richards and Winter [HRW92] give a good summary of this work.

The fixed-parameter version of this problem, where the number of terminals p is constant, is easily solvable in polynomial time. This result was also published in the first volume of the journal *Networks* in 1971, by Dreyfus and Wagner [DW71], and discovered independently by Levin [Lev71]. It turns out that a solution to to STEINER TREE can be described by the *topology* of the solution tree. By *topology*, we mean the relative order in which the paths between the terminals merge. After the topology of the tree is known, the solution can be obtained by shortest paths computations. When p is constant, we can try all possible topologies to obtain the optimal solution in polynomial time.

For general p, many different methods yield an approximation ratio of $2-f(n)$, for some function f of n ([HRW92], page 173). This was the best known approximation ratio until Zelikovsky [Zel93] broke through this barrier and gave an algorithm with a performance guarantee of $11/6 \approx 1.833$. He has since slowly been improving this ratio, with the latest result (with Robins) being a factor of $1 + \frac{\ln 3}{2} \approx 1.55$ [RZ00]. The problem is Max SNP-hard [BP89], which means that there is some ϵ for which there is no approximation algorithm with a ratio better than $1+\epsilon$, unless $P = NP$ [ALM+92].

1.3.2 Generalized Steiner Network

The Generalized Steiner Network problem has also received some attention in recent years, since it is solved by a natural application of the Primal-Dual method of approximation [GW95]:

Generalized Steiner Network (GSN): Given an undirected graph $G = (V, E)$, with weights on the edges, and connectivity requirements $r_{i,j}$ for each pair of nodes i and j, find a minimum weight subgraph H of G that includes $r_{i,j}$ disjoint paths between every pair of nodes.

The special case where $r_{i,j} \in \{0, 1\}$ is called the Generalized Steiner Tree problem. A direct application of the Primal-Dual method gives a 2-approximation to this problem [GW95], which was first discovered by Agrawal, Klein and Ravi [AKR95]. An approximation algorithm for Generalized Steiner Network with a performance guarantee of $2 \max r_{i,j}$ was given by Williamson et al [WGMV95].

1.3.3 Directed Steiner Tree

The Directed Steiner Tree problem is the simplest Steiner problem in directed graphs:

Directed Steiner Tree (DST): Given a directed graph $G = (V, E)$, with weights on the edges, a set of p terminals T and a root vertex r, find a minimum weight subgraph H of G that includes paths from every terminal to the root.

As in Steiner Tree, a simple brute-force search through tree topologies solves the fixed-parameter version of this problem (where the number of terminals p is fixed). This cannot be done for p-DSN, since the paths from the source terminals to their destinations may overlap repeatedly; this makes the number of possible topologies exponential, even though the number of terminals is constant.

The only known approximation algorithm for DST was for the case where G is acyclic [Zel97] until Charikar et al [CCC+98] achieved an approximation ratio of $\mathcal{O}(i(i-1)p^{1/i})$ for any constant i (the running time of their algorithm is $n^i p^{2i}$). Feige

[Fei98] shows that there is no approximation with a ratio better than $\ln p$ unless $NP \subseteq DTIME(n^{O(\log\log n)})$.

1.3.4 Disjoint Paths

The p-DSN problem becomes much harder if the p paths between the s_i and t_i are required to be edge-disjoint (or node-disjoint). Under that restriction, the problem is NP-complete already for $p = 2$ [FHW80]. More precisely, it is NP-hard even to determine whether *any* feasible solution H exists.

1.4 Applications

Algorithms for Steiner problems are fundamental tools used to design networks [MW84]. When many points in a network need to be connected, and there is a cost associated with connecting them, the minimum Steiner tree between the points represents the cheapest way of building the network. Examples of such networks include transportation, communication or shipping. Leung, Magnati and Singhal [LMS88] give some examples and heuristics.

Until recently, the directed version of Steiner's problem was of mostly theoretical interest, since networks were usually symmetric. However, with the increasing diversity of network links such as satellite and radio, link costs are becoming less symmetric [Ram96]. Therefore, the proper model for designing some types of networks is one where the underlying graph is directed [SRV97].

The problem of Multicast Tree generation is one example of the use of directed Steiner problems in network design that has received some attention recently [SRV97] [Ram96]. A Multicast Tree is used for point-to-multipoint communication in high-bandwidth applications. For example, in video-conferencing, a single source must be broadcast to many different destinations. The routes taken to each destination are encoded in the Multicast Tree. Finding a low-cost Multicast Tree makes for more efficient use of resources. Ramanathan [Ram96] uses the Directed Steiner Tree problem to find low-cost Multicast Trees.

1.5 Our Contribution

In this paper, we give an exact algorithm for p-DSN for any constant p with a running time of $\mathcal{O}(n^{4p})$, where $n = |V|$ and $m = |E|$. We also give an exact algorithm for p-SCSS for any constant p with a running time of $\mathcal{O}(n^{2p-1})$. In fact, for the unweighted case, and for sparse graphs ($m = \mathcal{O}(n \log n)$), we give slightly improved bounds. For details, see Appendix A.2.

For clarity, we present an algorithm for a version of p-DSN where there are no weights, and the goal is to minimize the number of *nodes* in the subgraph H. Section 4.3 shows how to generalize our algorithm to weights and edge-minimization through simple modifications.

Our algorithm for p-DSN can best be understood in terms of a game, where a player moves tokens around the graph. Initially, p tokens are placed on the starting nodes $s_1, \ldots, s_p$, one token per node. The player is then allowed to make certain types of moves with the tokens, and his goal is to perform a series of these moves to get the tokens to their respective destinations $t_1, \ldots, t_p$ (the token from s_1 to t_1, the token from s_2 to t_2, etc).

Every possible move has a cost associated with it: the number of nodes that are visited by the moving tokens. We define the moves carefully so that the lowest cost move sequence to get the tokens from $s_1, \ldots, s_p$ to $t_1, \ldots, t_p$ will only visit nodes from the optimal subgraph H, and will visit each node in H exactly once. The difficulty of the construction is to ensure that such a sequence exists for every optimal H. For $p = 2$ this is easy to do, since the two involved paths can only share vertices in a very restricted manner. However for $p \geq 3$ the relationships between the paths become significantly more complex. Critical to our argument is a structural lemma analyzing how these paths may overlap.

Overview In Chapter 2, we give a simple algorithm that solves p-SCSS for $p = 2$, while also defining the token game in more detail. We generalize this approach to any constant p and state the algorithm solving p-SCSS in Chapter 3, making use of a token game similar to the one described above. The correctness proof is given in Sections 3.4 and 3.5.

Using the algorithm for p-SCSS, we then in Chapter 4 give the algorithm for the p-DSN problem and prove its correctness. We conclude the paper by summarizing our results and discussing possible future research directions in Section 4.4.

Chapter 2

A Solution for 2-SCSS

We begin by solving unweighted 2-SCSS, the problem of finding a minimum (in terms of the number of nodes) strongly connected subgraph H of a graph $G = (V, E)$ that includes two specified nodes s and t. This is equivalent to finding the smallest H that contains paths from s to t and from t to s. Considering this simple problem allows us to introduce the notation and methodology used in the following chapters. The algorithm described here is similar to the one given by Natu and Fang [NF97].

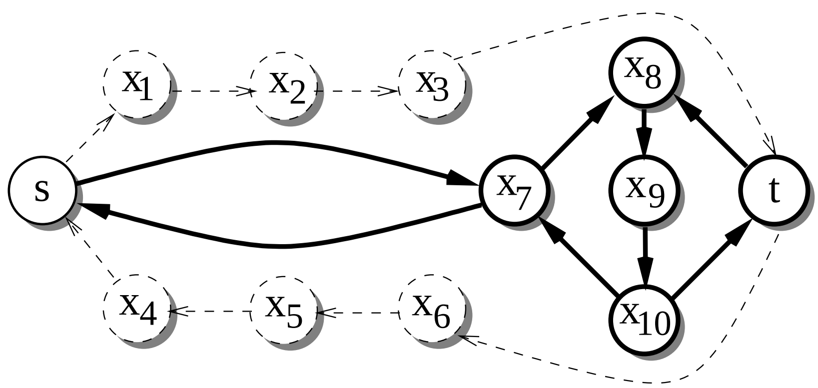

Figure 2-1: A sample graph for 2-SCSS, with terminals s and t. The optimal subgraph is in solid lines.

Figure 2-1 illustrates some of the difficulties of this problem. Let s, t be our terminals. The optimal subgraph consists of the six nodes $s, x_7, x_8, x_9, x_{10}, t$. The paths from s to t and t to s share vertex x_7, and share the vertex sequence $x_8 \to x_9 \to x_{10}$. Note that the optimal subgraph includes neither the shortest path from s to t, nor the shortest path from t to s.

2.1 The token game

To compute the optimal subgraph H, we will place two tokens, called f and b, on vertex s. We then move the tokens along edges, f moving forward along edges, and b moving backwards along edges, until they both reach t. Then the set of nodes visited during the sequence of moves will contain paths $s \rightsquigarrow t$ and $t \rightsquigarrow s$.

To find the smallest subgraph H containing those paths, we will charge for the moves. The cost of a move will be the number of new vertices entered by the tokens during that move. Clearly we could never find a move sequence that gets both tokens from s to t with cost lower than $|H|-1$, the size of the optimal solution H minus one (since we never charge for s). In fact, we will show that the lowest cost move sequence to get the tokens from s to t will have cost exactly $|H|-1$, and thus corresponds to an optimal solution.

The three kinds of moves we allow are given below. The notation $\langle x, y\rangle$ refers to the situation where token f is on vertex x, and token b is on vertex y. The expression "$\langle x_1, y_1\rangle \xrightarrow{c} \langle x_2, y_2\rangle$" means that it is legal to move token f from x_1 to x_2, and token b from y_1 to y_2 (at the same time), and that this move has cost c. We want to find a move sequence from $\langle s, s\rangle$ to $\langle t, t\rangle$ with minimal cost.

(i) *Token f moving forward:* For every edge $(u, v) \in E$ and all $x \in V$, we allow

(a) the move $\langle u, x\rangle \xrightarrow{1} \langle v, x\rangle$, and

(b) the move $\langle u, v\rangle \xrightarrow{0} \langle v, v\rangle$.

(ii) *Token b moving backward:* For every edge $(u, v) \in E$ and all $x \in V$, we allow

(a) the move $\langle x, v\rangle \xrightarrow{1} \langle x, u\rangle$, and

(b) the move $\langle u, v\rangle \xrightarrow{0} \langle u, u\rangle$.

(iii) *Tokens switching places:* For every pair of vertices $a, b \in V$ for which there is a path from a to b in G, we allow the move $\langle a, b\rangle \xrightarrow{c} \langle b, a\rangle$, where c is the length of the shortest path from a to b in G. By length we mean the number of vertices besides a and b on that path.

Type (i) and (ii) moves allow the tokens f and b to move forward along a single edge, and backward along an edge, respectively. Usually the cost is 1, accounting for the new vertex that the token visits. Only in the case where a token reaches a vertex with a token already on it, the cost is 0, since no 'new' vertices are visited.

Type (iii) moves allow the two tokens to switch places. We call this type of move a "flip", and say that the vertices on the shortest path from a to b are *implicitly* traversed by the tokens. The cost c of the move accounts for all of these vertices.

Let us return to the example in figure 2-1 to see how these moves are used. The lowest cost way to move both tokens from s to t is the following (we use subscripts to denote the type of the move).

$$\langle s,s\rangle \underset{(i)}{\xrightarrow{1}} \langle x_7,s\rangle \underset{(ii)}{\xrightarrow{0}} \langle x_7,x_7\rangle \underset{(i)}{\xrightarrow{1}} \langle x_8,x_7\rangle$$
$$\underset{(ii)}{\xrightarrow{1}} \langle x_8,x_{10}\rangle \underset{(iii)}{\xrightarrow{1}} \langle x_{10},x_8\rangle \underset{(ii)}{\xrightarrow{1}} \langle x_{10},t\rangle \underset{(i)}{\xrightarrow{0}} \langle t,t\rangle$$

The weight of this sequence is 5, which is $|H|-1$, and the nodes visited by the tokens are exactly the nodes in the optimal solution H.

2.2 The Algorithm

Let us phrase the preceding discussion in an algorithmic form. To compute H, we first construct a 'game-graph' $\widetilde{G}$. The nodes of the graph correspond to token positions $\langle x,y\rangle$, the edges to legal moves between positions. In our case, the nodes are just $V\times V$, and the edges are the ones given above as legal moves.

Finding H is done by computing a lowest cost path from $\langle s,s\rangle$ to $\langle t,t\rangle$ in $\widetilde{G}$. The graph H then consists of all the vertices from V that are mentioned along that path, including the vertices that are implied by type (iii) moves.

Running time: Clearly, this game-graph can be computed in polynomial time. First, we do an all-pairs-shortest-paths computation on G to obtain the information we need to set up the type (iii) moves. This takes time $\mathcal{O}(mn)$ using breadth-first search, where n and m are the number of vertices and edges in the original graph G. Then, we add to $\widetilde{G}$ all the edges corresponding to legal moves one at a time. The number of vertices in $\widetilde{G}$ is $\mathcal{O}(n^2)$. The number of type (i) and type (ii) edges in $\widetilde{G}$ is $\mathcal{O}(mn)$, since there is a type (i) and type (ii) move for every edge and vertex combination in G. The number of type (iii) edges is $\mathcal{O}(n^2)$ making the total time to compute $\widetilde{G}$ equal to $\mathcal{O}(mn)$.

Now we just have to perform a single-source shortest paths computation from $\langle s,s\rangle$ to obtain H. Since we now have weights on the type (iii) edges, we cannot use breadth-first search. For a simple implementation, we could use Dijkstra's algorithm (which has a running time of $\mathcal{O}(|\widetilde{V}|^2)$), and achieve a running time of $\mathcal{O}(n^4)$. Using Fibonacci heaps [FT87], which performs single-source shortest paths in time $\mathcal{O}(|\widetilde{E}|+|\widetilde{V}|\log|\widetilde{V}|)$, we achieve a running time of $\mathcal{O}(mn+n^2\log n)$.

As an aside, this algorithm can also be used to solve 2-DSN. Given a graph G and two node-pairs (s_1,t_1), (s_2,t_2), add two nodes s, t and edges $s\to s_1$, $t_1\to t$, $t\to s_2$, $t_2\to s$ to the graph and solve 2-SCSS for the two terminals s, t. The solution for this problem is also an optimal solution for the original 2-DSN problem (if we omit s and t). This leads to a running time of $\mathcal{O}(mn+n^2\log n)$ for 2-DSN, which is the same as the running time obtained by Natu and Fang [NF95].

In fact, we can do a little better for the unweighted case using more complicated shortest paths algorithms. For a more detailed analysis, see Appendix A.2.

2.3 Correctness

The proof that our algorithm actually solves 2-SCSS can be split into two claims. We provide a proof here that motivates the techniques used in the general case. An alternate proof can be found in [NF95, NF97].

Claim 2.3.1
If there is a legal move sequence from $\langle s, s\rangle$ to $\langle t, t\rangle$ with cost c, then there is a subgraph H of G of size $\leq c+1$ that contains paths $s \rightsquigarrow t$, and $t \rightsquigarrow s$.

If we follow a move sequence from $\langle s, s\rangle$ to $\langle t, t\rangle$, then f and b trace out paths $s \rightsquigarrow t$ and $t \rightsquigarrow s$, and this becomes our solution H. Moreover the tokens traverse at most $c+1$ vertices, since each vertex (except s) that we visit adds one to the cost of the move sequence.■

Claim 2.3.2
Let H^* be an optimal subgraph containing paths $s \rightsquigarrow t$ and $t \rightsquigarrow s$. Then there exists a move sequence from $\langle s, s\rangle$ to $\langle t, t\rangle$ with total cost $|H^*| - 1$.

This is the more difficult part of the correctness proof. We can prove it by actually constructing a move sequence $\langle s, s\rangle \rightsquigarrow \langle t, t\rangle$, using H^* as a 'map.'

When moving the tokens from $\langle s, s\rangle$ to $\langle t, t\rangle$, we 'pay' each time we reach a new vertex in H^*. In order to achieve total cost $|H^*| - 1$ we must make sure that we pay only once for each vertex in H^*. To ensure this, we enforce one rule: after a token moves off a vertex, no other token will ever move to that vertex again. We say that a vertex becomes 'dead' once a token moves from it, so that tokens are only allowed to move to vertices in H^* that are 'alive'. Note that the notion of dead and alive vertices is only used for the analysis, the algorithm itself never explicitly keeps track of them.

As we construct the move sequence, we maintain the invariant that there exist paths of 'alive' vertices from the position of token f to t, and from t to the position of token b. This will make sure that we do not have to violate our rule to proceed with the sequence. When we reach $\langle t, t\rangle$, we will have constructed a legal move sequence, and paid for each vertex in $|H^*|$ at most once. This will immediately imply the claim.

We construct the sequence in a greedy fashion. We start at position $\langle s, s\rangle$, and perform the following steps:

(a) **Move token f with type (i) moves.** Choose some path in H^* from the position of token f to t. We move token f forward along edges in that path (killing vertices along the way) using type (i) moves, until we reach t, or we reach a node x where moving token f would leave token b stranded; i.e., all paths of alive vertices from t to the position of token b go through x. We cannot move token f off of x, otherwise we would kill x and lose our invariant that there is a path of alive vertices from t to the position of token b (see figure 2-2).

(b) **Move token b with type (ii) moves.** Choose some path of alive vertices from t to the position of token b, and move the token b backwards along edges of that path towards t using type (ii) moves. We proceed until we reach t, x, or some node $y \neq x$ that would leave the token f stranded if we killed it (all paths from x to t go through y).

If the tokens are both on t, we are done. If the tokens are now on the same node x, go back to step (a). Note that in both these cases the last move of token b was free. If token b reached some node $y \neq x$, we have a 'deadlock' (see figure 2-2).

(c) **Resolve the deadlock with a flip: a type (iii) move.** To resolve the deadlock, we will use a type (iii) move. We know that all paths from x to t go through y, so there must be two disjoint simple paths P_{xy} and P_{yt} in H^* (see figure 2-2). Also, since all paths from t to y go through x, there must be a simple path P_{tx}. Path P_{tx} must also be disjoint from P_{xy}, since if it were not, token b would be able to get from y to t without going through x.

We apply the type (iii) move $\langle x, y\rangle \to \langle y, x\rangle$. The cost of the move is at most the size of P_{xy} (not including x and y) and we only kill nodes on P_{xy}. Since P_{xy} is disjoint from both P_{tx} and P_{yt}, we maintain our invariant that there are paths of 'alive' vertices from the position of token f to t and from t to the position of token b. Go back to step (a).

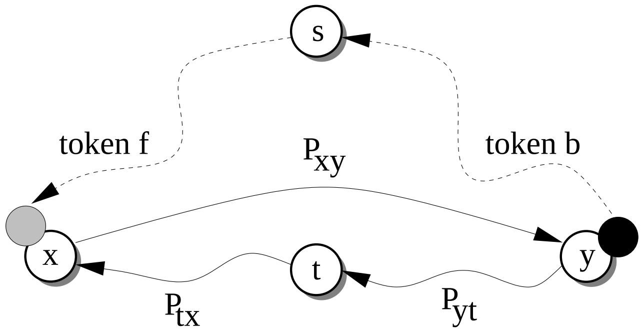

Figure 2-2: Deadlock in the construction of an optimal move sequence. Token f has moved forward onto node x, and token b has moved backward onto node y. All paths from x to t go through y, and al paths from t to y go through x. Therefore, moving either one of the tokens individually would leave the other token stranded. We resolve this deadlock using a flip: a type (iii) move exchanging the positions of the tokens.

We never move a token onto a dead vertex, and each step maintains the invariant that there are paths of alive vertices from the position of token f to t and from t to the position of token b. Therefore both tokens reach t, and we only pay once for each vertex we visit. Since we only visit vertices in H^*, and we don't pay for s, it must be the case that the cost of the move sequence is at most $|H^*| - 1$.■

This claim immediately implies that the shortest path in $\widetilde{G}$ will correspond to an optimal solution, since by Claim 2.3.1 and the minimality of H^* there are no paths in $\widetilde{G}$ on length less than $|H^*| - 1$.

The token movements for the $p = 2$ case essentially describe the way in which paths are shared in the optimal solution. Path sharing for $p \geq 3$ is more complex, so we will need a richer set of token moves, and a much more involved proof.

Chapter 3

Strongly Connected Steiner Subgraphs

In this chapter we give an algorithm for p-SCSS, which is a generalization of the algorithm for 2-SCSS given in the previous chapter.

Again we will use token movements to trace out the solution H. The way the tokens move is motivated by the following observation. Consider any strongly connected H containing $\{s_1, \ldots, s_p\}$. This H will contain paths from each $s_1, \ldots, s_{p-1}$ to s_p, and these paths can be chosen to form a tree rooted at s_p; we will call this tree the *forward tree*. The graph H will also contain paths from s_p to each $s_1, \ldots, s_{p-1}$, forming what we call the *backward tree*. Moreover, every H that is the union of two such trees is a feasible solution to our p-SCSS instance.

For ease of notation, we set $q := p - 1$ for the remainder of this chapter, and let $r := s_p$, as s_p plays the special role of 'root' in the two trees.

3.1 Token moves for p-SCSS

To trace out the two trees, we will have q "F-tokens" moving forward along edges in the forward tree from $\{s_1, \ldots, s_q\}$ to r, and q "B-tokens" moving backward along edges from $\{s_1, \ldots, s_q\}$ to r. Given a set of legal moves, we will again look for the lowest cost move sequence that moves all tokens to r. This will then correspond to the smallest subgraph containing paths $s_i \rightsquigarrow r$ and $r \rightsquigarrow s_i$ for all $i \leq q$, which is the graph we are looking for.

Since both sets of tokens trace out a tree, once two tokens of the same kind reach a vertex, they will travel the same way to the root. In that case, we will simply merge them into one token. It is therefore enough to describe the positions of the tokens by a pair of sets $\langle F, B \rangle$, where F and B are the sets of nodes currently occupied by the F- and B-tokens.

Again, we have three types of legal token moves. Type (i) moves correspond to F-tokens moving forward along an edge, and type (ii) moves correspond to B-tokens moving backward along an edge. We do not charge for entering a vertex if another token is already on it.

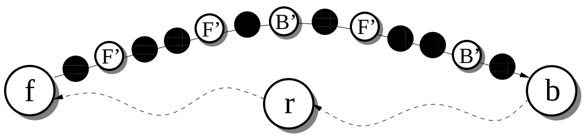

Figure 3-1: Flipping f and b, with tokens F' and B' that need to be "picked up." The black nodes are the set M.

Since there are are at most q tokens of each type, the possible token positions for a particular type are the subsets of V of size at most q. So, let $\mathcal{P}_q(V)$ be the set of subsets of V of size at most q.

(i) *Single moves for F-tokens:* For every edge $(u, v) \in E$, and all token sets $F \in \mathcal{P}_q(V)$, $B \in \mathcal{P}_q(V)$, such that $u \in F$, the following is a legal move:

$$\langle F, B\rangle \xrightarrow{c} \langle (F \setminus \{u\}) \cup \{v\}, B\rangle$$

where the cost c of the move is 1 if $v \notin F \cup B$, and 0 otherwise.

(ii) *Single moves for B-tokens:* For every edge $(u, v) \in E$, and all token sets $F \in \mathcal{P}_q(V)$, $B \in \mathcal{P}_q(V)$, such that $v \in B$, the following is a legal move:

$$\langle F, B\rangle \xrightarrow{c} \langle F, (B \setminus \{v\}) \cup \{u\}\rangle$$

where the cost c of the move is 1 if $u \notin F \cup B$, and 0 otherwise.

Type (iii) moves allow tokens to pass each other, similar to the type (iii) moves in the previous chapter, except that this time the "flip" is more complex (see figure 3-1). We have two 'outer' tokens, f and b, trying to pass each other. Between f and b there are other F-tokens moving forward and trying to pass b, and B-tokens moving backward and trying to pass f. These tokens, sitting on node sets F' and B', are 'picked up' during the flip.

(iii) *Flipping:* For every pair of vertices f, b, vertex sets F, B, $F' \subset F$, $B' \subset B$, such that:

- $f \in F \in \mathcal{P}_q(V)$
- $f \in B \in \mathcal{P}_q(V)$
- there is a path in G from $f \rightsquigarrow b$ going through all vertices in $F' \cup B'$

the following is a legal token move:

$$\langle F, B\rangle \xrightarrow{|M|} \langle (F \setminus \{f\} \setminus F') \cup \{b\}, (B \setminus \{b\} \setminus B') \cup \{f\}\rangle$$

where M is the set of vertices on a shortest path from f to b in G going through all vertices in $F' \cup B'$, besides f,b and the vertices in $F' \cup B'$.

3.2 The algorithm for p-SCSS

We can now state the algorithm for p-SCSS:

1. Construct a game-graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ from $G = (V, E)$. Set $\widetilde{V} := \mathcal{P}_q(V) \times \mathcal{P}_q(V)$, the possible positions of the token sets, and $\widetilde{E}$:= all legal token moves defined above.

2. Find a shortest path P in $\widetilde{G}$ from $\langle\{s_1, \ldots, s_q\}, \{s_1, \ldots, s_q\}\rangle$ to $\langle\{r\}, \{r\}\rangle$.

3. Let H be the union of $\{s_1, \ldots, s_q, r\}$ and all nodes given by P (including those in sets M for type (iii) moves).

The difficult part of constructing the game-graph $\widetilde{G}$ is computing the costs for the type (iii) moves that flip f and b. We need to know the size of shortest path between f to b in G going through all vertices in $F' \cup B'$. Note that we do not require this path to be simple. So, if we knew the order in which the vertices in F' and B' occurred on the path, computing the shortest path becomes easy: it is just the union of the shortest paths between consecutive nodes in that order. Since the number of tokens in $F' \cup B'$ is bounded by $2(q-1)$, which is a constant, we can simply try all possible permutations of the nodes in $F' \cup B'$.

The total running time of the algorithm is $\mathcal{O}(n^{2p-1})$. This can be improved slightly if the graph is sparse ($m = \mathcal{O}(n \log n)$). For more details on the running time, see appendix A.2.

3.3 Example

As an example we look at how the token game works on the graph in figure 3-2. Our terminals are s_1, s_2, s_3, s_4, s_5, and so we set s_5 to be the root vertex, and put a forward and backward token on each of the other terminals (figure 3-2(a)). In figures 3-2(b-i) we see the following move sequence:

$$
\begin{aligned}
&\langle\{s_1, s_2, s_3, s_4\}, \{s_1, s_2, s_3, s_4\}\rangle \\
\xrightarrow[(i)]{1}\ &\langle\{s_1, s_2, s_3, x_3\}, \{s_1, s_2, s_3, s_4\}\rangle \\
\xrightarrow[(ii)]{0}\ &\langle\{s_1, s_2, s_3, x_3\}, \{s_1, s_2, s_3\}\rangle \xrightarrow[(iii)]{2} \langle\{s_2\}, \{x_3\}\rangle \\
\xrightarrow[(i)]{1}\ &\langle\{x_5\}, \{x_3\}\rangle \xrightarrow[(ii)]{1} \langle\{x_5\}, \{x_4\}\rangle \xrightarrow[(iii)]{0} \langle\{x_4\}, \{x_5\}\rangle \\
\xrightarrow[(i)]{1}\ &\langle\{s_5\}, \{x_5\}\rangle \xrightarrow[(ii)]{0} \langle\{s_5\}, \{s_5\}\rangle.
\end{aligned}
$$

The total cost of the moves is 6, and therefore equal to $|H| - q = 10 - 4 = 6$, as expected. The solution $\{s_1, s_2, s_3, s_4, s_5, x_1, x_2, x_3, x_4, x_5\}$ (figure 3-2(j)) is made up

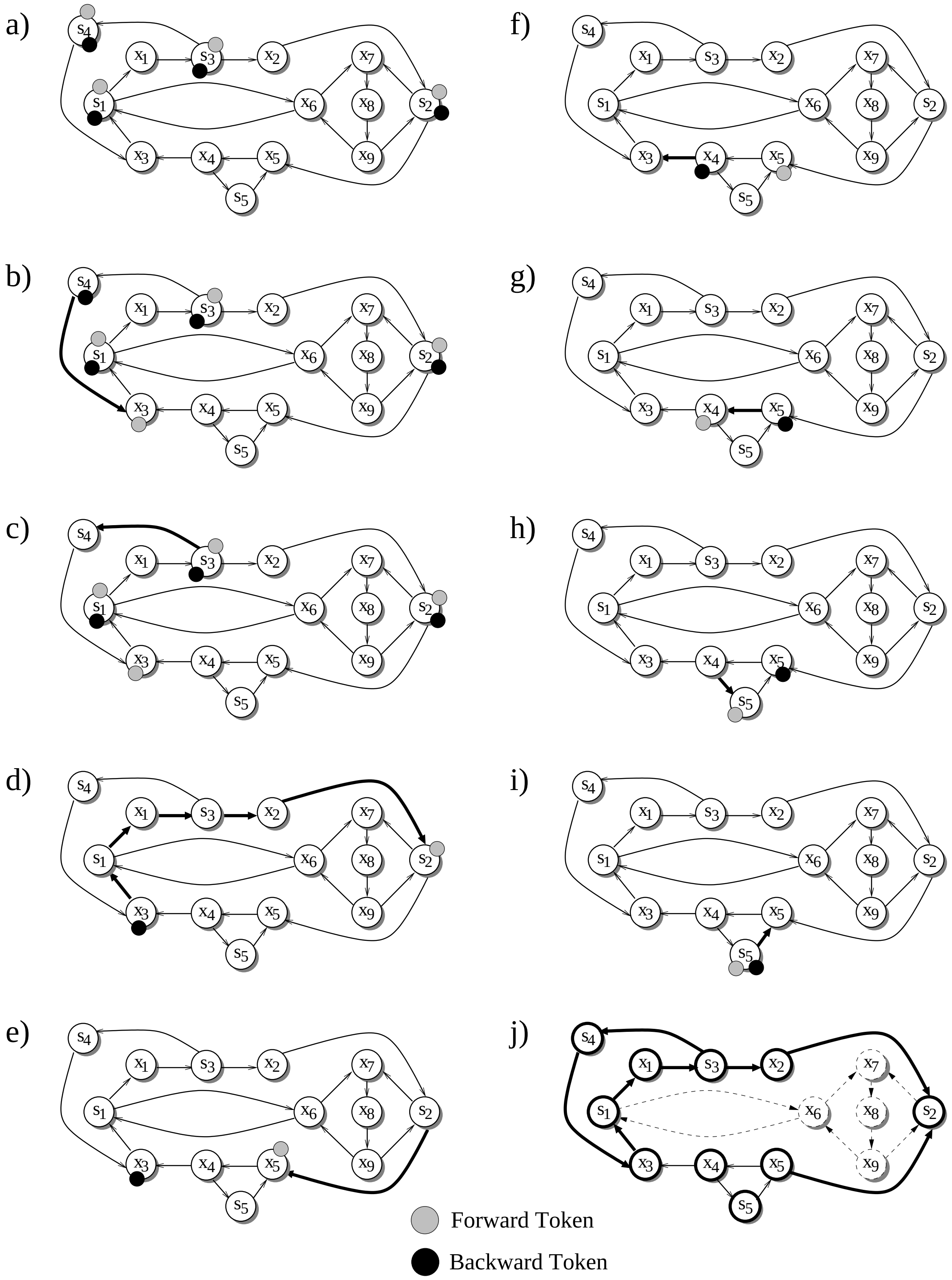

Figure 3-2: An example for the p-SCSS algorithm. Bold lines indicate the path of the last token move.

of the terminals $\{s_1, s_2, s_3, s_4, s_5\}$, the nodes $\{x_3, x_4, x_5\}$ mentioned in the sequence of moves, and the nodes $\{x_1, x_2\}$ in the set M for the first type (iii) move. This is optimal.

3.4 Correctness of the p-SCSS algorithm

The correctness proof for our p-SCSS algorithm can be split into the same two parts we used for 2-SCSS.

Lemma 3.4.1
Suppose there is a move sequence from $\langle\{s_1, \ldots, s_q\}, \{s_1, \ldots, s_q\}\rangle$ to $\langle\{r\}, \{r\}\rangle$ with total cost c. Then there exists a solution H to this p-SCSS instance of size $\leq c + q$. Moreover, given the move sequence, it is easy to construct such an H.

Proof: This follows directly from the definition of the moves. The cost of any move sequence is an upper bound on the number of vertices traversed by that sequence. Given the constructive nature of the moves, it is also easy to actually find H. ■

Together with the following, much more involved lemma, the correctness of the algorithm is proved.

Lemma 3.4.2
Suppose $H^* = (V^*, E^*)$ is any minimum cardinality feasible solution. Then there is a move sequence from $\langle\{s_1, \ldots, s_q\}, \{s_1, \ldots, s_q\}\rangle$ to $\langle\{r\}, \{r\}\rangle$ with weight equal to $|H^*| - q$.

Proof: To prove this lemma, we will effectively construct such a move sequence, where all intermediate positions of the tokens will be in H^*.

When moving the F- and B-tokens from $\{s_1, \ldots, s_q\}$ to r, we 'pay' each time we reach a new vertex. To account for each move, and achieve total cost $|H^*| - q$, we will use the same method as we used for 2-SCSS. We enforce the same rule: after a token moves off a vertex, no other token will ever move to that vertex again. As before, we say that a vertex becomes 'dead' once a token moves from it, so that tokens are only allowed to move to vertices in H^* that are 'alive'. This also makes sure that our move sequence will be finite, since no token can return to a vertex it has already visited.

We say that a token t *requires* a vertex $v \in V^*$ if all legal paths for t to get to r pass through v. By 'legal paths' we mean paths that are within H^*, go in the appropriate direction for the token t, and do not include any dead vertices. We will sometimes speak of tokens requiring tokens; in this case we mean that the first token requires the vertex on which the second token is sitting. Note that the requirement relation among tokens moving in the same direction is transitive, i.e. if f_1 requires f_2, and f_2 requires x, then f_1 also requires x.

We will construct our move sequence in a greedy fashion. That is, we will move tokens towards r using type (i) and (ii) moves, until each token sits on a vertex that is required by some other token to get to r. In this case we cannot apply any more type

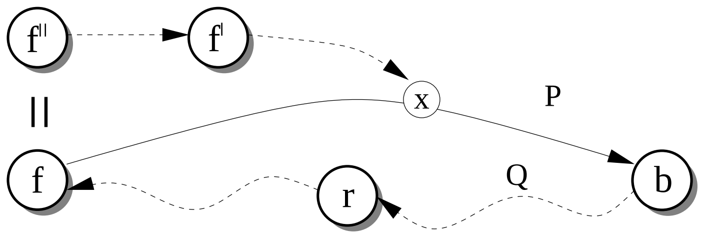

Figure 3-3: Showing Claim 3.4.4, that all tokens that require a vertex on P are on P themselves.

(i) or (ii) moves – doing so would leave another token stranded as it is not allowed to move onto the then dead vertex.

In this case we need to use a type (iii) move to resolve the deadlock. Showing that this is always possible is the core of the correctness proof, the 'flip lemma' shown in Section 3.5. To state this lemma and see how it implies the correctness of the algorithm, we have to introduce some additional notation.

Let the 'F_0-tokens' be the F-tokens that are not required by any other F-token. Similarly, let the 'B_0-tokens' be the B-tokens that are not required by any other B-token.

Lemma 3.4.3 (The Flip Lemma)
Suppose every token is required by some other token. Then there is an F_0-token f and a B_0-token b such that

- *f requires b, and no other F_0-token requires b,*
- *b requires f, and no other B_0-token requires f.* $\square$

We will prove this lemma in the next section. Let us now see how it concludes the proof of Lemma 3.4.2.

Let f and b be chosen according to the Flip Lemma. Fix any simple path from f to r that uses only live vertices. Call P the portion of this path between f and b, and Q the portion between b and r (see figure 3-3). By definition, P and Q are disjoint.

Claim 3.4.4
All tokens that require a vertex on P are on P themselves.

Proof: We prove the claim for F-tokens; a symmetric argument applies to B-tokens. Suppose some F-token $f' \neq f$ requires a vertex x on P. Every path $x \rightsquigarrow r$ must include b, otherwise f could move to x, and then to r, without visiting b. Therefore, f' also requires b (see figure 3-3). The token f' cannot be an F_0-token, since the Flip Lemma tells us that f is the only F_0-token that requires b. Note that due to transitivity, every F-token is either an F_0-token, or required by some F_0-token, so f' must be required by some F_0 token f''. By transitivity, f'' requires b, and so $f'' = f$, by the Flip Lemma.

Since $f = f''$ requires f', the token f' is either on P or Q. If f' is on Q, then x is also on Q, since f' requires x. This contradicts the fact that P and Q are disjoint, and so f' must be on P. ■

Let F' be the set of F-tokens that are on the path P, and B' be the set of B-tokens on P. We can apply a type (iii) move that switches f and b, and picks up F' and B' along the way. All vertices on P become dead. No token is stranded, since by Claim 3.4.4, all tokens that required a vertex on P were on P, and therefore were picked up by the flip.

We have shown, pending the Flip Lemma, that each step of the construction of our move sequence preserves paths of 'alive' vertices from the F-tokens to r and from r to the B-tokens, and never moves a token onto a dead vertex. This shows that we can always continue the construction of our move sequence until all tokens reach r, and that the cost of the move sequence will be no more than $|H^*| - q$. ■

3.5 The Flip Lemma

To complete the proof of correctness, it remains to show Lemma 3.4.3, the Flip Lemma. We prove the Flip Lemma by making a graph out of the requirement relationships between the F_0 and B_0 tokens. We show that there is a two-cycle in this graph, consisting of an F_0 and a B_0 token, that does not have any other incoming requirements. This proves the lemma.

During the discussion, keep in mind that transitivity only holds among requirements for the same type of token; if an F-token f requires a node (or a token) x, and some F-token f' requires f, then f' requires x. However, if some B-token b requires f, it is not necessarily the case that b requires x, since b is moving backwards along edges.

Proof of Lemma 3.4.3 (The Flip Lemma):

The requirement graph G_{req}. Let $G_{req} = (V_{req}, E_{req})$ be a new directed graph, whose nodes are the F_0 and B_0-tokens. The edges in E_{req} correspond to requirements: G_{req} has an edge $x \to y$ iff the token x requires the token y.

This graph has a lot of structure. First of all, G_{req} is bipartite, since no two F_0-tokens (and no two B_0-tokens) require each other (by the definition of an F_0- and B_0-token). For the remainder of the discussion, we will use f to denote a node on the F_0 side of the bipartition, and b to denote a node on the B_0 side.

By assumption (every token is required by some other token) and by definition (an F_0-token is not required by any F-token), we know that every F_0-token is required by at least one B-token. We know that either that B-token is a B_0-token, or there is another B_0-token that requires that B-token. Therefore, by transitivity for B-tokens, every F_0-token is required by at least one B_0-token. By symmetry, every B_0-token is required by at least one F_0-token. Therefore, every node in G_{req} has at least one incoming edge.

We want to find a two-cycle in G_{req} with no incoming edges, since two tokens in a such a cycle would require each other, but would be required by no other tokens, and the lemma would be proven. We can view G_{req} as a dag (directed acyclic graph) of strongly connected components, and sort the strongly connected components topologically. Let C be the first component in that ordering. This means that no token outside of C requires any token in C. Furthermore, C cannot consist of only one node, since then that token would be required by no other token, in contradiction to our assumption that every token is required by at least one token. If C contains exactly two nodes, we have found our two-cycle.

We will prove that C cannot consist of more than two nodes, which will imply the Flip Lemma, and the correctness of our algorithm. The proof rests on the observation that G_{req} satisfies a kind of transitivity, which we call the *projection* property. After showing his property, we prove the claim by contradiction by applying the projection property across the requirement graph.

Claim 3.5.1
Projection: *Suppose for three nodes f_1, f_2, b_1 ($f_1 \neq f_2$) in G_{req} we have edges $f_1 \to b_1$ and $b_1 \to f_2$ in G_{req}. Then the following holds: all nodes b that have an edge $b \to f_1$ also have an edge $b \to f_2$.*

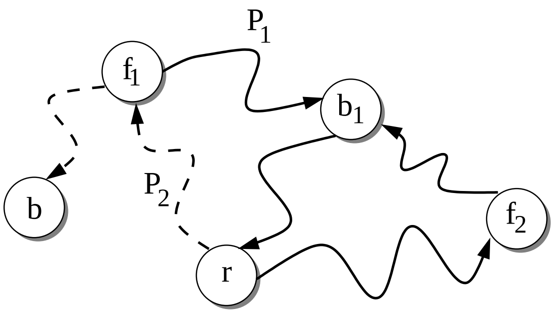

Figure 3-4: Proving the projection property in G_{req}. The solid lines are paths in H^* corresponding to edges $f_1 \to b_1$ and $b_1 \to f_2$ in G_{req}, the dashed line to the edge $b \to f_1$.

Proof: By definition of F_0, there is a legal path in H^* from f_1 to r avoiding f_2. Since f_1 requires b_1, this path goes through b_1. Therefore, there is a path P_1 from f_1 to b_1 avoiding f_2 (see figure 3-4).

Suppose that some node b in G_{req} has an edge $b \to f_1$. We show that $b \to f_2$ is also in G_{req} by contradiction. If $b \to f_2$ is not in the requirement graph, then there is a legal path in H^* from r to b avoiding f_2. Since b requires f_1, this path goes through f_1. Therefore, there is a path P_2 in H^* from r to f_1 avoiding f_2. Combining P_2 and P_1, we obtain a path from r to b_1 that does not visit f_2 in contradiction to $b_1 \to f_2$ being in G_{req}. ■

A symmetric property holds by exchanging f's and b's, i.e. for any triple f_1, b_1, b_2, if there are edges $b_1 \to f_1$ and $f_1 \to b_2$ in G_{req}, then for every node f in G_{req}, if there is an edge $f \to b_1$, then there must also be an edge $f \to b_2$.

The projection property has a profound effect on the structure of G_{req}. In fact, it shows that if there is a path in G_{req} from a node b to a node f, then there is an edge from b to f. To see this, consider the last four nodes on the path from b to f in G_{req}, $b'' \to f' \to b' \to f$. By projection, there must be an edge $b'' \to f$. This shortens the path by two. This argument can be continued up the path until it shows that there must be an edge $b \to f$.

A symmetric argument (using the symmetric projection property) shows that if there is a path in G_{req} from a node f to a node b, then there is an edge from f to b. We further conclude that every strongly connected component in G_{req} is a complete bipartite graph (every pair of nodes on opposite sides of the bipartition is connected in both directions). Now we are ready to finish the Flip Lemma.

Claim 3.5.2
No strongly connected component C of G_{req} has more than 2 nodes.

Proof: We prove the claim by contradiction. Assume that a strongly connected component C in G_{req} has at least three elements. Either there are two F_0 nodes in C, or two B_0 nodes in C. We assume there are two F_0 nodes (a symmetric argument shows the other case). Since C is a complete bipartite graph, there must be a complete bipartite subgraph of C consisting of nodes f_1,b_1,f_2.

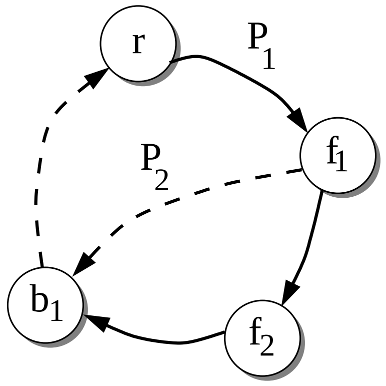

Figure 3-5: Components with more than 2 elements are impossible

We turn back to H^* to show that this structure cannot exist. Since b_1 requires both f_1 and f_2, there is a legal path in H^* from r to b_1 that visits both f_1 and f_2 (solid lines in figure 3-5). Without loss of generality assume that f_1 is the first node on that path, so that there is a path P_1 from r to f_1 that avoids f_2. Since f_1 requires b_1, but f_1 does not require f_2, there must also be a path P_2 from f_1 to b_1 that avoids f_2 (dashed lines in figure 3-5). Combining P_1 and P_2, we obtain a legal path in H^* from r to b_1 that avoids f_2, in contradiction to the fact that b_1 requires f_2.

This contradiction shows that C cannot have more than 2 nodes. ■

Chapter 4

The Directed Steiner Network problem

4.1 The Algorithm

In this chapter we show how to apply the algorithm developed in the previous chapter to solve DIRECTED STEINER NETWORK problem (p-DSN), for any constant p.

DIRECTED STEINER NETWORK (p-DSN) (unweighted, node-minimizing): Given a directed graph $G = (V, E)$, and p pairs of nodes in the graph $\{(s_1, t_1), \ldots, (s_p, t_p)\}$, find the subgraph H of G with the fewest number of nodes that contains paths from s_i to t_i for $1 \leq i \leq p$.

We use the same general model of a token game, but now we have tokens moving from each source s_i to its destination t_i. This time, we have no backwards moving tokens, and also tokens do *not* merge when they reach the same node. We describe the positions of the tokens by a p-tuple $\langle f_1, f_2, ..., f_p \rangle$. We have two kinds of moves for the tokens. The first kind of move allows a single token to move one step along an edge.

(i) For each edge (u, v) we include the moves $\langle —, u, — \rangle \xrightarrow{c} \langle —, v, — \rangle$, meaning that one token moves from u to v, and all others remain where they are. The cost c of the move is 0 if v already has a token on it, and 1 otherwise.

We also allow a group of tokens to move through a strongly connected component all at once. To see why this is useful, consider the optimal solution to p-DSN and contract every strongly connected component into a single node; the resulting graph is a dag (see figure 4-1). Each contracted component has at most p tokens entering, and at most p tokens exiting. We can compute the best way for some group of k tokens ($k \leq p$) to move from any k or fewer specific entrance points to any k or fewer specific exit points in a strongly connected component by solving an instance of $2k$-SCSS.

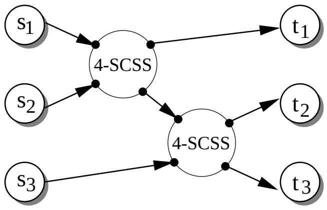

Figure 4-1: A solution to p-DSN is a dag of strongly connected components

(ii) For all $k \leq p$, and for every set of k node-pairs $\{(f_1, x_1), (f_2, x_2), \ldots, (f_k, x_k)\}$, for which there is a strongly connected subgraph of G containing all of the nodes in the pairs, we allow the move

$$\begin{aligned} &(\text{—}f_1\text{—}f_2\text{—}\ldots\text{—}f_k\text{—}) \\ \stackrel{c}{\rightarrow}\ &(\text{—}x_1\text{—}x_2\text{—}\ldots\text{—}x_k\text{—}). \end{aligned}$$

The cost c of this move is the size of the smallest strongly connected component containing the vertices $\{f_1, f_2, \ldots, f_k, x_1, x_2, \ldots, x_k\}$ minus the size of the set $\{f_1, \ldots, f_k\}$. We can use the the algorithm developed in Chapter 3 to compute this cost.

Similar in structure to our algorithm for p-SCSS in Chapter 3, the algorithm for p-DSN consists of the following steps.

1. Compute the game-graph $\mathcal{G}$, where the vertices in $\mathcal{G}$ are p-tuples of vertices in the input graph G, and edges are included for each legal token move.

2. Find the minimum-weight path P in $\mathcal{G}$ from $\langle s_1, \ldots, s_p\rangle$ to $\langle t_1, \ldots, t_p\rangle$.

3. Output the subgraph H of G induced by P, i.e. the subgraph containing

   - all vertices of G explicitly 'mentioned' by vertices in P, and
   - for all type (ii) moves used in P, all the vertices making up the smallest strongly connected component containing the f_i's and x_i's used to define that move.

4.2 Correctness

As for the previous algorithms, it is easy to see that for any move sequence from $\langle s_1, \ldots, s_p\rangle$ to $\langle t_1, \ldots, t_p\rangle$ of cost c, there is a feasible solution H of size at most $c + |\{s_1, \ldots, s_p\}|$. It is also easy to find this H, given the move sequence. The following lemma then implies the correctness of the algorithm.

Lemma 4.2.1
Let H^* be a minimum size subgraph of G that contains paths $s_i \rightsquigarrow t_i$ for all $i \in \{1, \ldots, p\}$. Then there is a legal sequence of token moves from $\langle s_1, \ldots, s_p \rangle$ to $\langle t_1, \ldots, t_p \rangle$ with cost $|H^*| - |\{s_1, \ldots, s_p\}|$.

Proof: We again do a constructive proof. We start with tokens $f_1, \ldots, f_p$ at $s_1, \ldots, s_p$, and move them to their respective destinations $t_1, \ldots, t_p$.

Regard each strongly connected component in H^* as a single node, and topologically sort this dag of strongly connected components. Let $C_1, \ldots, C_m$ be the resulting order of strongly connected components. We now consider each component in order, and move each token in the component either to its destination (if its destination is in the component), or to some later component in the ordering on a path to its destination. After doing so, all nodes in the component are dead. This ensures that we pay only once for every node.

For each component C_i containing some k tokens ($k \leq p$), we perform the following moves. We execute (a) and (b) if C_i consists of more than one node, and only (b) if C_i consists of a single node.

(a) We apply a type (ii) move. For each token f_ℓ in C_i we define a node x_ℓ in C_i to which it moves. For tokens f_ℓ whose destination t_ℓ is in C_i, we set x_ℓ to that destination. For all other tokens f_ℓ we choose any legal path to its destination t_ℓ and let x_ℓ be the last node of that path that is in C_i. Using a type (ii) move we simultaneously move all the tokens f_ℓ to their respective x_ℓ.

(b) We apply a type (i) move for each token f_ℓ in C_i that is not yet at its destination t_ℓ. We move along one edge of a path to t_ℓ into a new component C_j. ■

4.3 Weights and edges

The algorithms provided for p-DSN and p-SCSS can easily be modified to handle weighted nodes; just make the cost of a move the total weight of the unoccupied nodes entered during the move instead of just their number.

It is also easy to minimize the total edge weight in H. To do this, we make every vertex in G have weight 0, and replace every edge e by a new vertex having the weight of e. We connect this new vertex to the two vertices incident to e. Naturally, it is also possible to combine vertex weights and edge weights.

4.4 Conclusion

We have developed a polynomial time algorithm that computes the smallest subgraph containing paths between p pairs of nodes in a directed graph. It is an interesting question whether the tools developed to obtain this result can be used to construct improved approximation algorithms for arbitrary p, or for the closely related Directed Steiner Tree and Minimum Equivalent Digraph problems.

Also, it would be interesting to see further implications of the Flip Lemma. It seems that the Flip Lemma shows something deeper about directed graphs than what we've used it for here. Perhaps a more general statement of the structural properties of the Lemma could be formulated and used for other optimization problems on directed graphs.

Appendix A

Notes and Runtime analysis

A.1 Natu and Fang's algorithm for 3-DSN

In [NF97] Natu and Fang propose an algorithm for the 3-DSN problem. In this chapter we will briefly discuss their approach, and give a counterexample on which their algorithm apparently does not work correctly. They also conjecture that an extension of their algorithm solves p-DSN for $p > 3$. The same counterexample applies to this conjecture.

Their algorithm operates on edge-weighted graphs and minimizes the *total weight of edges* in H. To compute the optimal H, they use a 'divide-and-conquer' approach based on dynamic programming. Central to the design of the algorithm is their 'Optimal Decomposition Theorem' (p. 220 in [NF97]). It states that optimal solutions can be broken down into independent parts in the following manner.

Theorem A.1.1 (Optimal Decomposition Theorem)
Suppose H is the optimal subgraph for a 3-DSN *instance* $\{(s_1, t_1), (s_2, t_2), (s_3, t_3)\}$. *Then there is a partition of H into edge-disjoint subgraphs $H = H' \cup H''$, and three vertices a_1, a_2, a_3 in H such that:*

- *H', H'' contain at least one edge*
- *For all $i = 1, 2, 3$ either*
    - *H' contains a path $s_i \rightsquigarrow a_i$ and H'' contains a path $a_i \rightsquigarrow t_i$, or*
    - *H' contains a path $a_i \rightsquigarrow t_i$ and H'' contains a path $s_i \rightsquigarrow a_i$.* $\square$

The theorem as stated does not hold for the graph given in figure A-1. Note that the optimal H must contain all edges of the graph. Suppose that we split this graph into two non-empty edge disjoint subgraphs H' and H''. Then there must be a pair of consecutive edges that are not in the same subgraph.

Assume that, e.g, $s_1 \to t_2$ and $t_2 \to s_2$ are in different subgraphs. Since one of the subgraphs has to contain a path $s_3 \rightsquigarrow a_3$, and the other a path $a_3 \rightsquigarrow t_3$, we must have $a_3 = t_2$, and $\{s_3 \to t_1, t_1 \to s_1, s_1 \to t_2\}$ are all in the same sub-graph. But

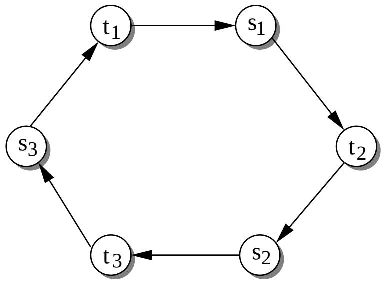

Figure A-1: Counterexample to the Optimal Decomposition Theorem

then the other subgraph contains none of the edges incident to s_1 or t_1, and therefore can contain neither a path $s_1 \rightsquigarrow a_1$ nor a path $a_1 \rightsquigarrow t_1$, and thus the theorem fails. For all other pairs of consecutive edges in the graph, essentially the same argument applies.

A.2 Runtime analysis

In this chapter we provide the running time analysis for our algorithms solving p-SCSS (from Chapter 3) and p-DSN (from Chapter 4).

The aim of this chapter is mainly to give an idea as to how the running time is distributed over the different parts of the algorithms (game-graph construction and shortest path computation).

A.2.1 The p-SCSS algorithm

The algorithm consists of two parts: the generation of the game-graph $\widetilde{G}$ from the input $G = (V, E)$, and the computation of a shortest path from $\langle\{s_1, \ldots, s_q\}, \{s_1, \ldots, s_q\}\rangle$ to $\langle\{r\}, \{r\}\rangle$ in $\widetilde{G}$.

The size of the game-graph $\widetilde{G}$. In the following n and m are always the number of vertices and edges, respectively, of the input graph G. We can assume $m \geq n - 1$, since if it was not, there would be some node disconnected from the graph which we could ignore. The number of vertices in the game-graph $\widetilde{G}$ is

$$|\mathcal{P}_q(V) \times \mathcal{P}_q(V)| = \left(\sum_{i=0}^{q} \binom{n}{i}\right)^2 = \mathcal{O}(n^{2q}).$$

The number of type (i) edges can be computed as follows. If we fix an edge $(u, v) \in E$, then there are $|\mathcal{P}_{q-1}(V \setminus \{u\})|$ choices for F, and $|\mathcal{P}_q(V)|$ choices for B, so the total number of type (i) edges is

$$m \cdot |\mathcal{P}_{q-1}(V \setminus \{u\})| \cdot |\mathcal{P}_q(V)| = \mathcal{O}(m \cdot n^{q-1} \cdot n^q) = \mathcal{O}(mn^{2q-1})$$

By symmetry, the number of type (ii) edges is the same.

For the type (iii) edges, we can also obtain an upper bound on their number by multiplying the number of choices for f and b ($\mathcal{O}(n)$ each), F and B ($\mathcal{O}(n^{q-1})$ each), and F' and B' ($\mathcal{O}(2^{q-1})$ each after choosing F and B). This yields a bound of $\mathcal{O}(n^{2q})$. Since $m \geq n-1$, the total number of edges is $\mathcal{O}(mn^{2q-1})$. The number of edges in $\widetilde{G}$ therefore is not much larger than the number of nodes.

Constructing the edge weights of the game-graph $\widetilde{G}$. Computing the edge weights takes constant time for type (i) and (ii) edges, but is slightly more expensive for type (iii) edges. We need to compute, for each type (iii) edge, the length of a shortest path from a node f to a node b going through at most $2q-2$ specific intermediate nodes (the ones in $F' \cup B'$). Since this path does not have to be simple, it will be the union of the shortest paths between the consecutive intermediate vertices. Since there are only a constant number of intermediate vertices, we can guess their order.

More formally, we run an all-pairs shortest paths algorithm on the input graph G; this takes time at most $\mathcal{O}(n^2 \log n + mn)$. Now, for each type (iii) edge, we go through all possible sequences in which the vertices in $F' \cup B'$ could appear on the path. For each sequence, we add together the shortest path distances for consecutive vertices to compute the total cost of the best path for that sequence. The shortest path length among all sequences is kept as the weight of the type (iii) edge. There are $\mathcal{O}((2q-2)!)$ possible sequences of the vertices in $F' \cup B'$, and adding together the distances takes time $\mathcal{O}(2q)$. So as long as p (and therefore q) is constant, the time to compute the weight of a type (iii) edge is also constant.

To summarize, we spend a constant amount of time to compute the weight of each edge in the graph, which leads to a total time of $\mathcal{O}(mn^{2q-1})$ for the game-graph construction – subsuming the time for the all-pairs shortest path computation. Note that this bound also holds for the case where the original graph has arbitrary weights.

Computing the shortest path in the game-graph $\widetilde{G}$. The second part of the algorithm is to compute a single-source shortest path query in the game-graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$. Since all known shortest paths algorithms must look at every edge, this step will dominate the running time of the algorithm. There are many different shortest paths algorithms out there [AMOB95], and the best one to use for a given graph depends on n, m and C, where C is the largest weight in the graph, if all weights are positive integers. For the case where the input graph G is weighted with positive real numbers, we want an algorithm that does not depend on C. Thus, the best choice is to use Fibonacci heaps [FT87], which performs single-source shortest paths in time $\mathcal{O}(|\widetilde{E}| + |\widetilde{V}| \log |\widetilde{V}|)$, and the total running time of our algorithm is:

$$\mathcal{O}(n^{2q} + mn^{2q-1} + n^{2q}\log n) = \mathcal{O}(mn^{2p-3} + n^{2p-2}\log n).$$

If the input graph has no weights, as in the case treated in this thesis, the weights in the game-graph will be positive integers bounded by $\mathcal{O}(n)$. So, we can use an algorithm of Ahuja et al. [AMOT90] that performs a shortest path query in time $\mathcal{O}(|\widetilde{E}| + |\widetilde{V}|\sqrt{\log(|\widetilde{V}|C)})$, where C is the largest weight in the game-graph. Therefore the running time for the unweighted p-SCSS algorithm is:

$$\mathcal{O}(mn^{2q-1} + n^{2q}\sqrt{\log n}) = \mathcal{O}(mn^{2p-3} + n^{2p-2}\sqrt{\log n}).$$

In both cases, if $m = \omega(n\log n)$, we don't have to worry about the second term, and hence the running time is just $\mathcal{O}(n^{2p-1})$.

A.2.2 The p-DSN algorithm

For this algorithm, the game-graph $\mathcal{G}$ consists of $\mathcal{O}(n^p)$ nodes, and can therefore have up to $\mathcal{O}(n^{2p})$ edges. This means that the final shortest path computation will take time at most $\mathcal{O}(n^{2p})$, using Dijkstra's algorithm. We are rough with this calculation since it turns out that for the p-DSN algorithm, the time to construct the game-graph actually overshadows this shortest-path computation.

The most time-consuming part of the game-graph construction is to determine the weights of the type (ii) edges. Obviously, it would be very inefficient to call our k-SCSS algorithm for every type (ii) edge in the game-graph. Fortunately, a simple observation makes it possible to avoid that.

Solving the instances of k-SCSS efficiently. Every instance of k-SCSS we wish to solve constructs a game-graph from the same underlying graph G. Furthermore, this game-graph does not depend on which vertices are terminals, but only on G and the number k. This is because the game-graph describes legal token moves, all independent of the starting positions of the tokens and their goal. Therefore, all instances of k-SCSS that operate on the same underlying graph G and have the same number of terminals use the same game-graph. Let us call this game-graph $\widetilde{G}_k$.

We need to solve instances of k-SCSS for all $k \leq 2p$. But, notice that $\widetilde{G}_k$ is a sub-graph of $\widetilde{G}_{2p}$ if $k \leq 2p$. Moreover, there are no edges from this sub-graph $\widetilde{G}_k$ to any other vertices in $\widetilde{G}_{2p}$, since all other vertices $\langle F, B\rangle$ in $\widetilde{G}_{2p}$ have more than k nodes in either F or B. So, we only need to construct a single game-graph $\widetilde{G}_{2p}$ to determine all the weights of the type (ii) edges. However, we need to perform several different shortest path computations on this graph for the different terminal sets.

Luckily, these computations all have something in common. Solving a k-SCSS instance requires computing a shortest path in $\widetilde{G}_{2p}$, to a node of the form $\langle\{r\},\{r\}\rangle$.

This suggests the following strategy: Run n single-destination shortest path computations, one for each possible destination $\langle \{r\}, \{r\} \rangle$ ($r \in V$). Now the weights of type (ii) edges can then be computed in constant time by looking up the appropriate shortest path length.

The number of nodes in $\widetilde{G}_{2p}$ is $\mathcal{O}(n^{4p-2})$, and the number of edges is $\mathcal{O}(mn^{4p-3})$. Using Fibonacci heaps to perform each of the n shortest-path computations, we obtain a running time of $\mathcal{O}(mn^{4p-2} + n^{4p-1} \log n)$. In the unweighted case, the maximum weight in $\widetilde{G}_{2p}$ is again $\mathcal{O}(n)$. So, we use the algorithm of Ahuja et al. [AMOT90], and obtain a running time of $\mathcal{O}(mn^{4p-2} + n^{4p-1}\sqrt{\log n})$.

Again, in both cases, if $m = \omega(n \log n)$, we don't have to worry about the second term, and hence the running time is just $\mathcal{O}(n^{4p})$.

Bibliography

[AKR95] Ajit Agrawal, Philip Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24(3):440–456, June 1995.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings 33rd IEEE Symposium on the Foundations of Computer Science*, pages 14–23, Los Angeles, CA, 1992. IEEE Computer Society.

[AMOB95] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, and Michael O. Ball. Network flows. *SIAM Review*, 37(1):115–??, March 1995.

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, April 1990.

[BP89] Marshall Bern and Paul Plassmann. The Steiner problem with edge lengths 1 and 2. *Information Processing Letters*, 32(4):171–176, September 1989.

[CCC+98] Moses Charikar, Chandra Chekuri, To-yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed Steiner problems. *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 192–200, 1998.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[CR41] Richard Courant and Herbert Robbins. *What Is Mathematics?* Oxford University Press, London, 1941.

[DK99] Yevgeniy Dodis and Sanjeev Khanna. Designing networks with bounded pairwise distance. *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC)*, pages 750–759, 1999.

[DW71] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1971.

[Fei98] Feige. A threshold of ln n for approximating set cover. *JACM: Journal of the ACM*, 45, 1998.

[FHW80] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[FR99] Jon Feldman and Matthias Ruhl. The directed steiner network problem is tractable for a constant number of terminals. *40th annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 299–308, 1999.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GW95] Goemans and Williamson. The primal-dual method for approximation algorithms and its application to network design problems. *D. Hochbaum, editor, Approximation Algorithms for NP-hard Problems, PWS Publishing, Boston*, pages 144–191, 1995.

[Hak71] S. L. Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1:113–133, 1971.

[HRW92] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem*. Number 53 in Annals of Discrete Mathematics. Elsevier Science Publishers B. V., Amsterdam, 1992.

[Hsu75] Harry T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *Journal of the ACM*, 22(1):11–16, January 1975.

[JK34] V. Jarník and O. Kössler. O minimálních grafech obsahujících n daných bodu. *Ĉas. Pêstování Mat.*, 63:223–235, 1934.

[KRY95] Samir Khuller, Balaji Raghavachari, and Neal Young. Approximating the minimum equivalent digraph. *SIAM Journal on Computing*, 24(4):859–872, August 1995.

[Lev71] A. Y. Levin. Algorithm for the shortest connection of a group of graph vertices. *Sov. Math. Dokl.*, 12:1477–1481, 1971.

[LMS88] J. M. Leung, T. L. Magnanti, and V. Singhal. Routing in point-to-point delivery systems. Technical Report OR-174-88, Massachusetts Institute of Technology, Operations Research Center, January 1988.

[LMS92] Chung-Lun Li, S. Thomas McCormick, and David Simchi-Levi. The point-to-point delivery and connection problems: complexity and algorithms. *Discrete Applied Mathematics*, 36(3):267–292, 1992.

[MT69] Dennis M. Moyles and Gerald L. Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *Journal of the ACM*, 16(3):455–460, July 1969.

[MW84] T. L. Magnanti and R. T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1):1–55, February 1984.

[NF95] Madan Natu and Shu-Cherng Fang. On the point-to-point connection problem. *Information Processing Letters*, 53(6):333–336, 1995.

[NF97] Madan Natu and Shu-Cherng Fang. The point-to-point connection problem – analysis and algorithms. *Discrete Applied Mathematics*, 78:207–226, 1997.

[Ram96] S. Ramanathan. Multicast tree generation in networks with asymmetric links. *IEEE/ACM Transactions on Networking*, 4(4):558–568, August 1996.

[RZ00] G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 770–779, 2000.

[SRV97] H. Salama, D. Reeves, and Y. Viniotis. Evaluation of multicast routing algorithms for real-time communication on high-speed networks. *IEEE Journal on Selected Areas in Communication*, February 1997.

[WGMV95] Williamson, Goemans, Mihail, and Vazirani. A primal-dual approximation algorithm for generalized steiner network problems. *COMBINAT: Combinatorica*, 15, 1995.

[Zel93] Alexander Zelikovsky. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica*, 9:463–470, 1993.

[Zel97] Alexander Zelikovsky. A series of approximation algorithms for the acyclic directed steiner tree problem. *Algorithmica*, 18(1):99–110, May 1997.