The Directed Steiner Network problem is tractable for a constant number of terminals

Jon Feldman[*] Matthias Ruhl[†]

MIT Laboratory for Computer Science
Cambridge, MA 02139, USA

Abstract

We consider the DIRECTED STEINER NETWORK *problem, also called the* POINT-TO-POINT CONNECTION *problem, where 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.*

1. Introduction

In this paper 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 $G = (V,E)$, and p pairs of nodes in the graph $\{(s_1,t_1),\ldots,(s_p,t_p)\}$, find the smallest subgraph H of G that contains paths from s_i to t_i for $1 \leq i \leq p$.

According to how 'smallest' is defined, there are several variations of this problem. In this paper, 'smallest' will mean 'minimum number of nodes'. Other possibilities are 'minimum number of edges' or 'smallest cost' if G is a graph with node and edge costs. We extend our results to these variations at the end of the paper.

The DSN problem occurs naturally when designing networks delivering goods from sources to destinations, where an underlying network is present, but its services have to be paid for. For example, the graph G could be a set of internet routers, where edges are connections between routers. Suppose that a bank needs to send secure data over the network from sources s_i to destinations t_i. But to transmit their data securely, the routers used in the transmissions have to be upgraded, which is expensive. The bank naturally wants to minimize the number of routers to upgrade.

Just like the original STEINER TREE problem [6], DSN is **NP**-complete if the number of pairs p is part of the input. For constant p, on the other hand, its complexity was mostly unknown so far. The case $p = 1$ is just a shortest path query, and for $p = 2$ the problem was solved in 1992 by Li, McCormick and Simchi-Levi [8]. They state the case $p \geq 3$ as an open problem.

Our Contribution In this paper, we give a polynomial time algorithm for any constant p, and therefore resolve this open problem. More precisely, the running time is $O(mn^{4p-2} + n^{4p-1} \log n)$, where $n = |V|$ and $m = |E|$.

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 visit exactly the nodes of the optimal subgraph H. 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

[*]E-Mail: `jonfeld@theory.lcs.mit.edu`
[†]E-Mail: `ruhl@theory.lcs.mit.edu`

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.

We find that most of the difficulty of p-DSN is contained in the special case when $t_i = s_{i+1}$ for $1 \leq i < p$, and $t_p = s_1$. It is not hard to see that every optimal solution H to this special case must be a strongly connected subgraph. This problem is therefore equivalent to the STRONGLY CONNECTED STEINER SUBGRAPH problem, defined as follows.

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

We give an algorithm for p-SCSS that runs in time $O(mn^{2p-3} + n^{2p-2} \log n)$, for any constant p, which makes use of a token game similar to the one mentioned above.

Related Work There are many related Steiner problems that are well-studied, most of them for undirected graphs. For a monograph on the subject, see [7].

The only previously known polynomial-time algorithm for p-DSN with constant p, except for the trivial case $p = 1$, was the one given by Li, McCormick and Simchi-Levi [8] for $p = 2$. The running time of their algorithm is $O(n^5)$. Natu and Fang in [9] and [10] improved this running time first to $O(n^4)$, and then to $O(mn + n^2 \log n)$. In [10] they also present an algorithm for $p = 3$, and conjecture that a variant for their algorithm works for all constant p. In Appendix A we provide what we believe to be a counterexample to the correctness of their algorithm for $p = 3$, and thus to their conjecture.

There is strong evidence that p-DSN is not fixed-parameter tractable, i.e. there is no algorithm with a running time of $O(n^k)$ for some constant k independent of the parameter p. This follows from results proved for the 'W-hierarchy' defined by Downey and Fellows [3], where it was shown that DIRECTED STEINER TREE is $W[2]$-hard.

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 $p = 2$ [4]. More precisely, it is NP-hard even to determine whether *any* feasible solution H exists.

Other recent work has centered on the approximability of p-DSN for general p. The best positive result obtained so far is by Charikar et al [1], who achieve an approximation ratio of $O(p^{2/3} \log^{1/3} p)$ for any p. They also give an approximation algorithm for p-SCSS for general p that achieves an approximation ratio of $2i(i-1)p^{1/i}$ and runs in time $O(n^i p^{2i})$. On the negative side, Dodis and Khanna [2] prove that p-DSN is $\Omega(2^{\log^{1-\varepsilon} p})$-hard.

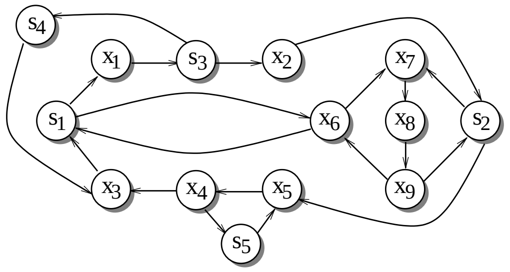

Figure 1. A sample graph

Overview In section 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 section 3. The correctness proof is given in sections 4 and 5.

Using the algorithm for p-SCSS, we then in section 6 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 7.

2. A Solution for 2-SCSS

We begin by solving 2-SCSS, the problem of finding a minimum strongly connected subgraph H of a graph $G = (V, E)$ that includes two specified nodes s_1 and s_2. This is equivalent to finding the smallest H that contains paths from s_1 to s_2 and from s_2 to s_1. Considering this simple problem allows us to introduce the notation and methodology used in the following sections. The algorithm described here is similar to the one given by Natu and Fang [10].

Figure 1 illustrates some of the difficulties of this problem. Let s_1, s_2 be our terminals. The optimal subgraph consists of the six nodes $s_1, x_6, x_7, x_8, x_9, s_2$. The paths from s_1 to s_2 and s_2 to s_1 share vertex x_6, and share the vertex sequence $x_7 \to x_8 \to x_9$. Note that the optimal subgraph includes neither the shortest path from s_1 to s_2, nor the shortest path from s_2 to s_1.

2.1. The token game

To compute the optimal subgraph H, we will place two tokens, called f and b, on vertex s_1. We then move the tokens along edges, f moving forward along edges, and b moving backwards along edges, until they both reach s_2.

Then the set of nodes visited during the sequence of moves will contain paths $s_1 \rightsquigarrow s_2$ and $s_2 \rightsquigarrow s_1$.

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. The lowest cost move sequence to get the tokens from s_1 to s_2 then corresponds to the 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 $\langle s_1,s_1\rangle$ to $\langle s_2,s_2\rangle$ 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 1 to see how these moves are used. The lowest cost way to move both tokens from s_1 to s_2 is the following (we use subscripts to denote the type of the move).

$$\langle s_1,s_1\rangle \xrightarrow[(i)]{1} \langle x_6,s_1\rangle \xrightarrow[(ii)]{0} \langle x_6,x_6\rangle \xrightarrow[(i)]{1} \langle x_7,x_6\rangle$$
$$\xrightarrow[(ii)]{1} \langle x_7,x_9\rangle \xrightarrow[(iii)]{1} \langle x_9,x_7\rangle \xrightarrow[(ii)]{1} \langle x_9,s_2\rangle \xrightarrow[(i)]{0} \langle s_2,s_2\rangle$$

The weight of this sequence is 5, which is $|H|-1$. The difference by one is due to the fact that we never pay for entering s_1.

2.2. The Algorithm

Let us phrase the preceeding 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. Clearly, this game-graph can be computed in polynomial time.

Finding H is done by computing a lowest cost path from $\langle s_1,s_1\rangle$ to $\langle s_2,s_2\rangle$ 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.

2.3. Correctness

The proof that our algorithm actually solves 2-SCSS can be split into two claims. We just provide the essential ideas behind the proof, and refer the reader to section 4 for the general case, or to [9, 10] for an alternative proof for the $p = 2$ case.

Claim 2.1
If there is a legal move sequence from $\langle s_1,s_1\rangle$ to $\langle s_2,s_2\rangle$ with cost c, then there is a subgraph H of G of size $\leq c+1$ that contains paths $s_1 \rightsquigarrow s_2$, and $s_2 \rightsquigarrow s_1$.

This is easy to see. If we follow a move sequence from $\langle s_1,s_1\rangle$ to $\langle s_2,s_2\rangle$, then f and b trace out paths $s_1 \rightsquigarrow s_2$ and $s_2 \rightsquigarrow s_1$. Moreover the tokens traverse at most $c+1$ vertices, since we pay for each vertex (except s_1) that we visit.

Claim 2.2
Let H^* be an optimal subgraph containing paths $s_1 \rightsquigarrow s_2$ and $s_2 \rightsquigarrow s_1$. Then there exists a move sequence from $\langle s_1,s_1\rangle$ to $\langle s_2,s_2\rangle$ 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 $\langle s_1,s_1\rangle \rightsquigarrow \langle s_2,s_2\rangle$, that visits every vertex in H^* only once. The key idea here is that if we fix two paths $s_1 \rightsquigarrow s_2$ and $s_2 \rightsquigarrow s_1$ in H^*, then wlog, they share vertices only in a very restricted manner. They may share several disjoint sequences of vertices, but these sequences occur in reverse order on the two paths (see figure 2). This is because if two segments occur in the same order, they can be merged by making the paths the same between the two segments.

So to construct the move sequence, we move both tokens using type (i) and (ii) moves until they reach a shared segment on the paths. In figure 2, token f will reach vertex x,

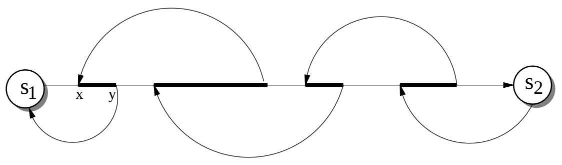

Figure 2. Paths $s_1 \rightsquigarrow s_2$ and $s_2 \rightsquigarrow s_1$ sharing sequences of vertices. The straight horizontal line from s_1 to s_2 gives the path $s_1 \rightsquigarrow s_2$, the round segments are part of $s_2 \to s_1$. The bold lines are sequences shared by both paths. They occur in opposite order on the two paths.

and token b – moving backwards – will reach vertex y. Now we can apply a type (iii) move to exchange the two tokens, and count the vertices in the shared segment only once. We can then continue to use type (i) and (ii) moves until we hit the next shared segment, and so on, until both tokens reach s_2.

The token movements for $p \geq 3$ will be much more involved, since the paths can share vertices in more complex ways.

3. Strongly Connected Steiner Subgraphs

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

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 $\{s_1,\dots,s_p\}$. This H will contain paths from each $s_1,\dots,s_{p-1}$ 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 $s_1,\dots,s_{p-1}$, 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. Note that for 2-SCSS these two trees were just single paths.

For ease of notation, we set $q := p-1$ for the remainder of this section and the next section, 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 $\{s_1,\dots,s_q\}$ to r, and q "B-tokens" moving backward along edges from $\{s_1,\dots,s_q\}$ 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.

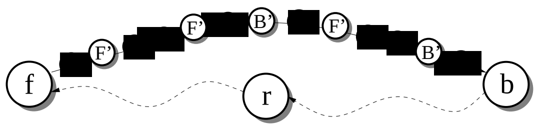

Figure 3. Flipping f and b, with tokens F' and B' that need to be "picked up." The black nodes are the set M.

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.

For any set S, let $\mathcal{P}_k(S)$ be the set of subsets of S of size at most k.

(i) *Single moves for F-tokens:* For every edge $(u,v) \in E$, and all token sets $F \in \mathcal{P}_{q-1}(V \setminus \{u\})$, $B \in \mathcal{P}_q(V)$, the following is a legal move:

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

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 $F \in \mathcal{P}_q(V)$, $B \in \mathcal{P}_{q-1}(V \setminus \{v\})$, the following is a legal move:

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

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 section, except that this time the "flip" is more complex (see figure 3). 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:

- there is a path in G from $f \rightsquigarrow b$ going through all vertices in $F' \cup B'$
- $F \in \mathcal{P}_{q-1}(V \setminus \{f,b\})$
- $B \in \mathcal{P}_{q-1}(V \setminus \{f,b\})$

the following is a legal token move:

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

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 $\langle \{s_1,\dots,s_q\}, \{s_1,\dots,s_q\}\rangle$ to $\langle \{r\}, \{r\}\rangle$.
3. Let H be the union of $\{s_1,\dots,s_q,r\}$ 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 do not require that the shortest path from f to b going through all vertices in $F' \cup B'$ be simple. Since the number of tokens in $F' \cup B'$ is bounded by $2(q-1)$, which is a constant, we can compute this path in polynomial time by simply trying all possible sequences of the nodes in $F' \cup B'$, and computing shortest paths along the sequence. For more details on the running time, see appendix B.

3.3. Example

As an example we look at how the algorithm works on the graph in figure 1, where now our terminals are s_1,s_2,s_3,s_4,s_5. The optimal solution is the node set $\{s_1,s_2,s_3,s_4,s_5,x_1,x_2,x_3,x_4,x_5\}$. The following is a sequence of lowest cost moves for this graph:

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

The total cost of the moves is 6, and therefore equal to $|H| - q = 10 - 4 = 6$, as expected. The solution is made up 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.

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 4.1
Suppose there is a move sequence from $\langle \{s_1,\dots,s_q\}, \{s_1,\dots,s_q\}\rangle$ 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 4.2
Suppose $H^* = (V^*, E^*)$ is any minimum cardinality feasible solution. Then there is a move sequence from $\langle \{s_1,\dots,s_q\}, \{s_1,\dots,s_q\}\rangle$ 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 $\{s_1,\dots,s_q\}$ to r, we 'pay' each time we reach a new vertex. In order to achieve total cost $|H^*| - q$ we must make sure that we pay only once for each vertex. 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 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. Note that the notion of dead and alive vertices is only used for the analysis, the algorithm itself never explicitly keeps track of them.

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 needed by some other token to get to r. In this case we cannot apply any more type (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 5. To state this lemma and see how it implies the correctness of the algorithm, we have to introduce some additional notation.

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.

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 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 4.2.

Let f and b be chosen according to the Flip Lemma. Fix any path P from f to b that uses only live vertices. For all vertices x on the path P, every path $x \rightsquigarrow r$ must include b, otherwise f could move to x, and then to r, without visiting b.

Suppose some F-token $f' \neq f$ requires a vertex on P, and therefore by transitivity also requires b. 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. The token f' must therefore be on P. In summary, all F-tokens are either on P, or do not require any vertex on P. By symmetry, the same applies to B-tokens.

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, but no token is stranded.

This proves that we can always continue the construction of our move sequence until all tokens reach r. $\blacksquare$

5. The Flip Lemma

Proof of Lemma 4.3 (The Flip Lemma): 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 $x \rightarrow y$ iff the token x requires the token y.

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, 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. Thus, every node in G_{req} has at least one incoming edge. G_{req} is also bipartite, since no two F_0-tokens (and no two B_0-tokens) require each other.

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, these tokens require each other, but are required by no other tokens, and the lemma is proven.

In the following we prove that C cannot consist of more than two nodes.

Claim 5.1
No strongly connected component C of G_{req} has more than 2 nodes.

Proof: The proof rests on the observation that G_{req} satisfies a kind of transitivity property. Suppose for three nodes f_1, f_2, b_1 ($f_1 \neq f_2$) in G_{req} we have edges $f_1 \rightarrow b_1$ and $b_1 \rightarrow f_2$ in G_{req}. Then the following holds: all nodes b that have an edge $b \rightarrow f_1$ also have an edge $b \rightarrow f_2$.

This is not hard to see. By definition of F_0, there is a legal path in H^* from f_1 to r avoiding f_2, and since f_1 requires b_1, there is a path P_1 from f_1 to b_1 avoiding f_2 (see figure

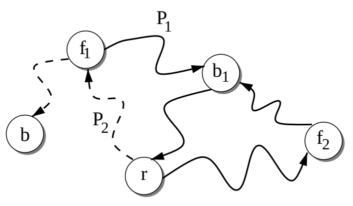

Figure 4. Proving transitivity in G_{req}. The solid lines are paths in H^* corresponding to edges $f_1 \rightarrow b_1$ and $b_1 \rightarrow f_2$ in G_{req}, the dashed line to the edge $b \rightarrow f_1$.

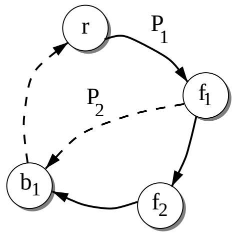

Figure 5. Components with more than 2 elements are impossible

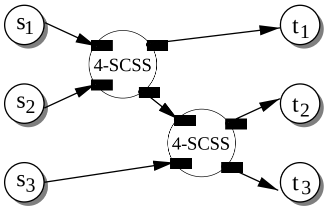

Figure 6. A solution to p-DSN is a dag of strongly connected components

4). Now assume that $b \to f_1$ is in G_{req}. If $b \to f_2$ is not in the requirement graph, then there is also a legal path P_2 in H^* from r to f_1 avoiding f_2, since b requires f_1. 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 argument 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 F_0-token f, if there is an edge $f \to b_1$, then there must also be an edge $f \to b_2$.

We now prove the claim by contradiction. Assume that a strongly connected component C in G_{req} has at least three elements $f_1, \dots, f_k, b_1, \dots, b_\ell$ ($k, \ell \geq 1$). For every pair f_i, b_j there is a path from f_i to b_j in G_{req}. Applying our transitivity observation along the path we conclude that the edge $f_i \to b_j$ must actually be in G_{req}. By symmetry, G_{req} also contains the edges $b_j \to f_i$ for all i, j.

Since $k + \ell \geq 3$, one of k and ℓ must be at least 2. Assume $k \geq 2$ (the case $\ell \geq 2$ is handled in the same manner). Then the token b_1 requires all f_i's. Therefore there is a legal path in H^* from r to b_1 that visits all f_i's (solid lines in figure 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 the token on node 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 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 assumption that b_1 requires all f_i's.

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

6. The Directed Steiner Network problem

6.1. The Algorithm

In this section we show how to apply the algorithm developed in the previous sections to solve the DIRECTED STEINER NETWORK problem (p-DSN), for any constant 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 $\langle \text{---} u \text{---} \rangle \xrightarrow{c} \langle \text{---} v \text{---} \rangle$, 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 6). 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 specific entrance points to any k specific exit points in a strongly connected component by solving an instance of $2k$-SCSS.

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

$$(\text{---} f_1 \text{---} f_2 \text{---} \dots \text{---} f_k \text{---})$$
$$\xrightarrow{c} (\text{---} x_1 \text{---} x_2 \text{---} \dots \text{---} x_k \text{---}).$$

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

Similar in structure to our algorithm for p-SCSS in section 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.

6.2. 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 6.1
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 component after it in the ordering. 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. ■

6.3. 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.

7. 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 problem. Another open question is whether these techniques can be used to obtain new results for other network design problems.

Acknowledgments

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.

References

[1] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed Steiner problems. *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 192–200, 1998.

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

[3] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[4] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

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

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

[7] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*. Number 53 in Annals of Discrete Mathematics. Elsevier Science Publishers B. V., Amsterdam, 1992.

[8] C.-L. Li, S. T. McCormick, and D. Simchi-Levi. The point-to-point delivery and connection problems: complexity and algorithms. *Discrete Applied Mathematics*, 36(3):267–292, 1992.

[9] M. Natu and S.-C. Fang. On the point-to-point connection problem. *Information Processing Letters*, 53(6):333–336, 1995.

[10] M. Natu and S.-C. Fang. The point-to-point connection problem – analysis and algorithms. *Discrete Applied Mathematics*, 78:207–226, 1997.

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

In [10] Natu and Fang propose an algorithm for the 3-DSN problem, provide a correctness proof, and conjecture that an extension of their algorithm solves p-DSN for $p > 3$. In this section we will briefly discuss their approach, and give a counterexample on which their algorithm apparently does not work correctly.

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 [10]). It states that optimal solutions can be broken down into independent parts in the following manner.

Theorem A.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 $s_i \rightsquigarrow a_i$.* □

The theorem as stated does not hold for the graph given in figure 7. 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.

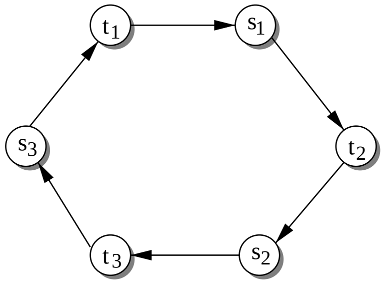

Figure 7. Counterexample to the Optimal Decomposition Theorem

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 subgraph. But 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.

B. Runtime analysis

In this section, we provide the running time analysis for our algorithms solving p-SCSS (from section 3) and p-DSN (from section 6).

The aim of this section 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).

It was not our goal to produce optimal algorithms, but rather to keep them simple to explain.

B.1. The p-SCSS algorithm

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

Let us first compute the size of $\widetilde{G}$. In the following n and m are always the number of vertices and edges, respectively, of the input graph G.

The number of vertices in the game-graph $\widetilde{G}$ is

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

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

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

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 ($O(n)$ each), F and B ($O(n^{q-1})$ each), and F' and B' ($O(2^{q-1})$ each after choosing F and B). This yields a bound of $O(n^{2q})$.

The number of edges in $\widetilde{G}$ therefore is not much larger than the number of nodes. Thus, edges should be stored as lists for each vertex, and not in an adjacency matrix.

Computing the edge weights takes constant time for type (i) and (ii) edges, but is slightly more expensive for type (iii) edges. It can be done with reasonable efficiency by first running an all-pairs shortest paths algorithm on the input graph G; this takes time at most $O(n^2 \log n + mn)$. Computing a shortest path from a node f to a node b visiting nodes in $F' \cup B'$ can now be done in time $O((2q-2)!)$ by going through all possible sequences in which the vertices in $F' \cup B'$ could appear on the path. So as long as p (and therefore q) is constant, this time is constant.

To summarize, we spend a constant amount of time to compute each of the edges in the graph, which leads to a total time of $O(n^{2q} + mn^{2q-1})$ for the game-graph construction – subsuming the time for the all-pairs shortest path computation.

The second part of the algorithm is to compute a shortest path query in the game-game $\widetilde{G} = (\widetilde{V}, \widetilde{E})$. Using Fibonacci heaps [5] this can be done in time $O(|\widetilde{E}| + |\widetilde{V}| \log |\widetilde{V}|)$, which is

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

Since computing the shortest path takes more time that constructing the graph, this also is the total running time of the algorithm.

B.2. The p-DSN algorithm

For this algorithm, the game-graph $\mathcal{G}$ consists of $O(n^p)$ nodes, and can have up to $O(n^{2p})$ edges. This means that the final shortest path computation will take time at most $O(n^{2p})$. It turns out that for this 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. First, note that the game-graph $\widetilde{G}$ constructed for an instance of k-SCSS does not depend on the source and terminal vertices s_i, t_i, but only on the underlying graph G and the number k. Let us call this game-graph $\widetilde{G}_k$. It is also true 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 $\widetilde{G}_{2p}$.

Solving a k-SCSS instance requires computing a shortest path in $\widetilde{G}_k$, or, equivalently, in $\widetilde{G}_{2p}$, to a node of the form $\langle \{r\}, \{r\} \rangle$. This suggests the following strategy: We can solve all these problems at the same time by running n single destination shortest path algorithms, one for each destination $\langle \{r\}, \{r\} \rangle$ $(r \in V)$. The weights of type (ii) edges can then be computed in constant time by looking up the appropriate shortest path length.

The running time for all n single destination shortest path queries is $O(mn^{4p-2} + n^{4p-1} \log n)$, which therefore is the total running time of the algorithm.

As an aside, there is a simpler way to solve 2-DSN than using our algorithm: 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. It is not hard to see that 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 an improved running time of $O(mn + n^2 \log n)$, which is the same as the running time obtained by Natu and Fang [9].