Disjoint-Path Facility Location: Theory and Practice

Lee Breslau * Ilias Diakonikolas † Nick Duffield * Yu Gu ‡
MohammadTaghi Hajiaghayi * David S. Johnson * Howard Karloff *
Mauricio G. C. Resende * Subhabrata Sen *

Abstract

This paper is a theoretical and experimental study of two related facility location problems that emanated from networking. Suppose we are given a network modeled as a directed graph $G = (V, A)$, together with (not-necessarily-disjoint) subsets C and F of V, where C is a set of customer locations and F is a set of potential facility locations (and typically $C \subseteq F$). Our goal is to find a minimum sized subset $F' \subseteq F$ such that for every customer $c \in C$ there are two locations $f_1, f_2 \in F'$ such that traffic from c to f_1 and to f_2 is routed on disjoint paths (usually shortest paths) under the network's routing protocols.

Although we prove that this problem is impossible to approximate in the worst case even to within a factor of $2^{\log^{1-\epsilon} n}$ for any $\epsilon > 0$ (assuming no NP-complete language can be solved in quasipolynomial time), we show that the situation is much better in practice. We propose three algorithms that build solutions and determine lower bounds on the optimum solution, and evaluate them on several large real ISP topologies and on synthetic networks designed to reflect real-world LAN/WAN network structure. Our main algorithms are (1) an algorithm that performs multiple runs of a straightforward randomized greedy heuristic and returns the best result found, (2) a genetic algorithm that uses the greedy algorithm as a subroutine, and (3) a new "Double Hitting Set" algorithm. All three approaches perform surprising well, although, in practice, the most cost-effective approach is the multi-run greedy algorithm. This yields results that average within 0.7% of optimal for our synthetic instances and within 2.9% for our real-world instances, excluding the largest (and most realistic) one. For the latter instance, the other two algorithms come into their own, finding solutions that are more than three times better than those of the multi-start greedy approach.

In terms of our motivating monitoring application, where every customer location can be a facility location, the results are even better. Here the above Double Hitting Set solution is 90% better than the default solution which places a monitor at each customer location – such comparisons help justify the proposed alternative monitoring scheme of [8]. Our results also show that, on average for our real-world instances, we could save an additional 18% by choosing the (shortest path) routes ourselves, rather than taking the simpler approach of relying on the network to choose them for us.

1 Introduction

This paper studies two new facility location problems relevant to questions of Internet traffic monitoring and content distribution. These problems differ from their more standard predecessors in that each customer must be served by two facilities rather than one. In addition, the service routes must be vertex-disjoint.

More specifically, suppose we are given a network modeled as a directed graph $G = (V, A)$, together with possibly overlapping vertex subsets $C, F \subseteq V$, where C is the set of customer locations and F is the set of potential facility locations. Suppose in addition that for each pair (c, f), $c \in C$ and $f \in F$, we are given a set $P(c, f)$ of directed simple paths from c to f in G. These sets may be defined implicitly (such as the set of all shortest c-to-f paths), or by an explicit list of permitted paths. Indeed, the most important special case for applications is that in which $P(c, f)$ is the set of *all* shortest c-to-f paths. By convention, we require that if $c \in C \cap F$, then the zero-length path (c) is in $P(c, c)$, and, since the zero-length path is the only simple path from c to c, we in fact have $P(c, c) = \{(c)\}$, (c) denoting the length-0 path starting and ending at c.

*AT&T Labs – Research, 180 Park Avenue, Florham Park, NJ 07932.

†Computer Science Department, University of California, Soda Hall, Berkeley, CA 94720. This work was performed when the author was a student at Columbia University, and was partially supported by AT&T.

‡Amazon Web Services, 1200 12th Avenue South, Seattle, WA 98144. This work was performed when the author was a student at the University of Massachusetts, and was partially supported by AT&T.

Suppose $c \in C$ is a customer location and (f_1, f_2) is a pair of potential facility locations from F, where either $f_1 \neq f_2$ or $f_1 = f_2 = c$.

DEFINITION 1.1. *We say that* (f_1, f_2) covers c in a pathwise-disjoint fashion *if there exist paths* $p_1 \in P(c, f_1)$ *and* $p_2 \in P(c, f_2)$ *that have no common vertex except* c. *Such a pair* covers c in a setwise-disjoint fashion *if* no *path in* $P(c, f_1)$ *shares a vertex (other than* c*) with* any *path in* $P(c, f_2)$.

Note that if $c \in C \cap F$ and $P(c, c)$ contains the zero-length path from c to itself, then by definition the pair (c, c) will cover c in both fashions.

DEFINITION 1.2. *A subset* $F' \subseteq F$ *is called a* pathwise-disjoint *(respectively,* setwise-disjoint*)* cover *for* C *if for every* $c \in C$ *there is a pair* (f_1, f_2), $f_1, f_2 \in F'$, f_1 *and* f_2 *not necessarily distinct, such that* (f_1, f_2) *covers* c *in a pathwise-disjoint (respectively, setwise-disjoint) fashion.*

The two problems we study are defined as follows:

DEFINITION 1.3. *In* PATHWISE-DISJOINT FACILITY LOCATION (PDFL), *we are given* G, C, F, *and the sets* $P(c, f)$ *and asked to find a pathwise-disjoint cover of minimum size for* C, *if such a cover exists.* SETWISE-DISJOINT FACILITY LOCATION (SDFL) *is the same problem except that the cover must be setwise-disjoint.*

In this paper, we analyze the complexity of PDFL and SDFL and propose and test algorithms for them. A first observation is that both problems can be viewed as special cases of SET COVER BY PAIRS (SCP), first described in [9].

SET COVER BY PAIRS (SCP): Given a ground set U of elements, a set S of *cover objects*, and a set T of triples (u, s, t), where $u \in U$ and $s, t \in S$, find a minimum-cardinality covering subset $S' \subseteq S$ for U, where S' *covers* U if for each $u \in U$, there are $s, t \in S'$, possibly with $s = t$, such that $(u, s, t) \in T$.

PDFL and SDFL can be formulated as SCP by taking $U = C$, $S = F$, and $(c, f_1, f_2) \in T$ if and only if (f_1, f_2) covers c in a pathwise-disjoint (setwise-disjoint) fashion.

We prove, subject to a complexity assumption, that no polynomial-time algorithm can approximate SCP to within a factor which is $2^{\log^{1-\epsilon} n}$ for any $\epsilon > 0$. The best previous hardness bound for SCP was just SET COVER-hardness [9]. We will show that our two facility location problems cases are in the worst case just as difficult, even if the sets $P(c, f)$ are restricted to shortest paths. Nevertheless, we shall see that for the types of problems that arise from real-world networks, a variety of algorithms perform extraordinarily well in practice.

We tested three main algorithms. Each uses as a subroutine a standard randomized greedy heuristic (GREEDY) that actually solves the general SCP problem. The first of our main algorithms, GREEDY400, is the variant of GREEDY that performs 400 runs and returns the best solution found. The second is a genetic algorithm (GENETIC) that uses GREEDY as a subroutine. The third, *Double Hitting Set* (HH), exploits the graph structure and approximately solves SETWISE-DISJOINT FACILITY LOCATION when the sets $P(c, f)$ consist of all shortest paths—precisely the version of the problem that arises in our monitoring application. As a valuable side effect, it also can be used to derive a very good lower bound on the optimum in this case.

In addition, we formulated the derived SCP instances as mixed integer programs, which CPLEX was able to solve to optimality for all of our instances with $|F| \leq 150$, although, of course, running times grew dramatically with graph size.

A final algorithmic challenge was that of constructing those derived SCP instances. This involves exploiting shortest path graphs to determine the (often quite large) sets of relevant triples. The triples are required by all of our algorithms (even HH), and we needed some significant algorithmic ingenuity to prevent this computation from being a major bottleneck.

1.1 Outline. The remainder of the paper is organized as follows: In Section 2 we describe the two applications motivating our study. In Section 3 we present our complexity results. In Section 4 we describe the heuristics we implemented. Our test instances are described in Section 5, and our experiments and their results are summarized in Section 6.

1.2 Related Work. The only previous work on SET COVER BY PAIRS is, as far as we know, is that of Hassin and Segev [9], which is theoretical rather than experimental. That paper considers two applications that were significantly different from those introduced here, and, from a worst-case point of view, much easier to approximate. The paper also introduces a variant of the Greedy algorithm studied here for the general SCP problem and analyzes its worst-case behavior.

2 Applications

2.1 A First Application: Content Distribution. The SETWISE- AND PATHWISE-DISJOINT FACILITY LOCATION problems arise in a variety of networking contexts. Our primary motivation for studying them comes from a scheme proposed in [8] for active monitoring of

end-to-end network performance. However, both variants have a simple alternative motivation in terms of an idealized content distribution problem, which we shall use to help illustrate the definitions. Suppose we wish to distribute real-time data, such as television broadcasts, over a network that does not provide a rapid method for recovering from link or vertex failures. Suppose further that the service interruptions caused by such failures would be costly to us, and that we want our distribution process to be relatively robust against them. A common standard of robustness is immunity to any single vertex or link failure (as for instance might result from an accidental cable cut). To guarantee such resilience, we would need to place multiple copies of our data source in the network, but because of the costs of hosting such copies, we would like to minimize the number of such hosting sites that we deploy.

Pathwise-Disjoint Facility Location models this application as follows. The network $G = (V, A)$ is the underlying fiber network linking various sites. The set C of customer locations is the set of sites that are to receive the data stream. The set F of facility locations is the set of potential sites where the content can be hosted. The sets $P(c, f)$ correspond to the paths over which we can route the content from facility f to customer c, where considerations such as presence of optical switches, desire for low latency, etc., may restrict the set of paths allowed. Note that here we are talking about paths from the facilities to the customers rather than from customers to facilities, as was the case in our definitions. However, it is easy to see that the two versions of the problem are equivalent – simply reverse the arcs.

If we assume that link capacity is not an issue, then the pathwise-disjoint cover of minimum size for C represents the minimum-cost choice of hosting locations for our data, subject to the constraint that no single vertex or link failure can disconnect a (non-failed) customer from all the data sources.

Setwise-Disjoint Facility Location models the variant of this application in which we do not have control over the routing, but instead must rely on the network to do our routing for us. Many Internet Service Providers (ISP's) route packets within their networks using a shortest-path protocol such as OSPF or IS-IS. In such protocols, packets must be routed along shortest paths, where the weight (length) of an arc is set by the network managers so as to balance traffic and optimize other performance metrics. If there is more than one shortest path leaving a given router, then the traffic is split evenly between the alternatives. This can be of further help in balancing traffic, and so traffic engineers may specifically set weights that yield multiple shortest paths between key routers. The actual splitting is performed based on computing hash functions of entries in a packet's header (typically simply the destination IP address). These functions are themselves randomly chosen, are subject to change at short notice, and are typically not available to us. Thus when there are multiple shortest paths, although contemporaneous packets from a given router to the same destination are likely to follow the same path, the actual route chosen may not be readily predictable. All we know is that it must be a member of the set $P(c, f)$ of all shortest paths from c to f. This means that the only way to guarantee vertex-disjoint paths from a customer c to two facility locations f and f' is to restrict attention to pairs (f, f') such that the corresponding shortest path sets intersect only in c, and consequently our problem becomes a Setwise-Disjoint Facility Location problem for such shortest path sets.

2.2 A Second Application: Host Placement for End-to-End Monitoring. This is the application that motivated this paper. It is more realistic than our first, but also more complicated, and readers more interested in the algorithmic results can safely skip this section on a first reading.

Suppose we are an Internet Service Provider (ISP) and provide "virtual private network" (VPN) service to some of our customers. In such a service, we agree to send traffic between various locations specified by the customer, promising to provide a certain level of service on the connections, but not specifying the actual route the packets will take. (The actual routing will be done so as to optimize the utilization of our network, subject to the promised levels of service.) Our network is a digraph $G = (V, A)$, in which the vertices correspond to routers and the arcs to the links between routers. A key service quality metric is packet loss rate (the fraction of packets on a path that fail to reach their destination). Let $p(r_1, r_2)$ denote the probability that a packet sent from router r_1 to router r_2 will successfully arrive. Our goal is to obtain estimates for $p(r_i, r_j)$ for a collection of customer paths P_{r_i,r_j}. Note that, in contrast to our content distribution application, we here do not worry about links' failing (which would cause re-routing), but merely about their underperforming.

One way to measure the loss rate on the path in our network from router r_1 to router r_2 is to attach extra equipment to the routers, use the equipment at r_1 to send special measurement packets to r_2, and use the equipment at r_2 to count how many of the packets arrive. If N packets are sent and N' arrive, then N'/N should be a good estimate for $p(r_1, r_2)$, assuming N is sufficiently large. Unfortunately, the

process of authorizing, installing, and maintaining the extra equipment can be time-consuming and expensive. Thus, this scheme may not be practical in a large network with hundreds or thousands of distinct path endpoints. Moreover, it may be impossible to install the needed equipment at some of the endpoints due to lack of space or other concerns. For these reasons, the authors of [8] proposed an alternative scheme that can adapt to the fact that certain vertices cannot house the needed equipment, and also may yield a substantial reduction in the total amount of monitoring equipment needed.

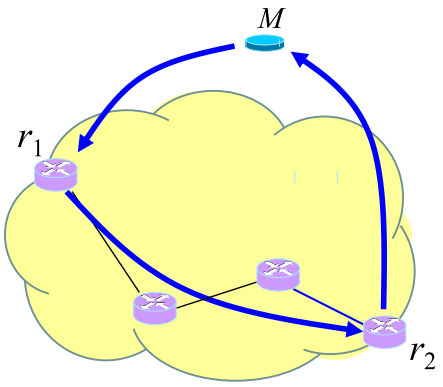

Figure 1: The centralized monitoring scheme of [8].

In this new scheme, all the monitoring is initiated from a single special measurement vertex M, as originally proposed in [2, 4]. See Figure 1. To measure loss on the path from vertex r_1 to vertex r_2, the equipment at M sends a packet on a circular path that first goes from M to r_1 (the *hop-on* path), then traverses the path from r_1 to r_2, and finally returns from r_2 to M (the *hop-off* path). Let us make the following assumptions:

1. Packets are only dropped by arcs, not by vertices. (This is a close approximation to reality in modern-day networks, where an arc models the system consisting of the physical wire/fiber connecting its endpoints, together with the line card at each of its ends.)

2. The three paths P_{M,r_1}, P_{r_1,r_2}, and $P_{r_2,M}$ are pairwise arc-disjoint. (As we shall show below, this will typically be true shortest-path routing.)

3. Loss rates on different arcs are independent of each other. (This is somewhat less realistic, but is approximately true except in heavily-loaded networks.)

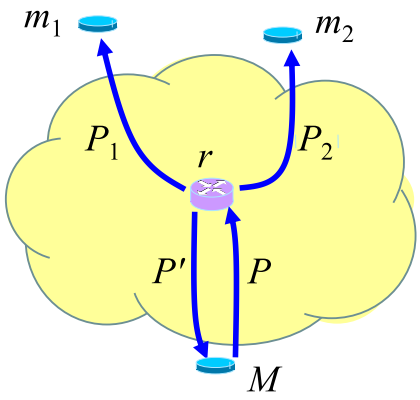

Figure 2: Scheme of [8] for measuring loss rate of hop-on and hop-off paths.

Then if N packets are sent on the circular path $P_{M,r_1,r_2,M}$, the expected number N' of packets successfully making the roundtrip will be $N' = Np(M,r_1)p(r_1,r_2)p(r_2,M)$. Thus if we measure N' and have good estimates for $p(M,r_1)$ and $p(r_2,M)$, we will have the estimate

$$p(r_1,r_2) = \frac{N'/N}{p(M,r_1)p(r_2,M)}\,.$$

Thus we have reduced the problem of measuring the loss rates for a collection of paths between arbitrary vertices to that of estimating the loss rates for the collection of hop-on and hop-off paths, all of which either begin or end at M.

In [2] it was proposed that these loss rates for a given path endpoint r be estimated by sending packets along an (M,r,M) circuit and, if, here, N packets were sent and N' received, concluding that $p(M,r) = p(r,M) = \sqrt{N'/N}$. Unfortunately, this assumes that Internet performance is symmetric, which it definitely is not. A quite accurate way to measure the loss rates would of course be to put equipment at both ends of each of the hop-on and hop-off paths, but this method would require installing equipment at just as many routers as in the original scheme for measuring the P_{r_1,r_2} paths directly – indeed at one more vertex, since now we need equipment at M. Setwise- and Pathwise-Disjoint Facility Location arise in the context of a tomographic method proposed by [8] for estimating loss rates in a potentially much more efficient fashion.

In terms of the facility location problems, the set C of "customer" vertices will consist of the endpoints of the original paths whose loss rates we wish to estimate. The set F of "facility locations" will be those vertices that are capable of hosting monitoring equipment,

which in this context we will call the (potential) monitoring vertices. Given a path endpoint r and a monitoring vertex m, the set $P(r, m)$ consists of all paths along which we may legally route packets from r to m in our network, as currently configured. (If r is itself a monitoring vertex, then $P(r, r)$ is the singleton set consisting only of the zero-length path from r to itself.)

Suppose r is a path endpoint and (m_1, m_2) is a pair of monitoring vertices that cover r in a pathwise-disjoint fashion. Then we claim (as justified below) that, by installing monitoring equipment at m_1 and m_2, we can estimate the loss rates $p(M, r)$ (for the hop-on path to r) and $p(r, M)$ (for the hop-off path from r). If r is itself a monitoring vertex, and $m_1 = m_2 = r$, then it is straightforward to measure both loss rates by sending packets between M and r and counting the number of them that successfully arrive at their destinations.

The key case is when m_1 and m_2 differ from r (and hence from each other). See Figure 2. Assuming we are allowed to specify the routing paths from r to m_1 and m_2, the fact that m_1 and m_2 cover r in a pathwise-disjoint fashion means that we can pick legal routing paths P_1 and P_2 from r to m_1 and m_2, respectively, that are vertex-disjoint except for r (and hence arc-disjoint). Moreover, assuming also that we have restricted ourselves to shortest paths, as are guaranteed by the most common routing protocols (OSPF and IS-IS), we also have that, under additional standard assumptions, the two paths P_1 and P_2 are arc-disjoint from the path P from M to r, which itself is arc-disjoint from the path P' from r to M.

This is a consequence of the following lemma, whose additional assumptions have to do with the arc weights used by OSPF and IS-IS in their shortest path computations. These weights are set by traffic engineers to help balance traffic loads and normally obey certain restrictions. First, they are positive integers. Second, in practice networks are typically *symmetric* directed graphs, in that the digraph contains an arc (a, b), then it must also contain arc (b, a). The weights w for such a digraph are themselves symmetric if for every arc (a, b), we have $w(a, b) = w(b, a)$. Typically, but not always, the weights are indeed symmetric.

LEMMA 2.1. *Suppose we are given a symmetric directed graph $G = (V, A)$, a weight function w on the arcs that is symmetric and positive, and three vertices a, b, c. If $P_{a,b}$ and $P_{b,c}$ are shortest-weight paths in this digraph from a to b and b to c, respectively, then they are arc-disjoint.* [Proof omitted.]

The basic idea of the technique of [8] for estimating the loss rate $p(M, r)$ using these paths is to send multicast packets from M to r along path P, replicate them at r, and then send the copies along paths P_1 and P_2 to m_1 and m_2, respectively. After this, m_1 and m_2 report back to M (using a guaranteed-delivery service such as TCP) as to which packets arrived. Based on this information, M estimates $p(M, r)$. The loss rate $p(r, M)$ can be estimated by sending packets along the (M, r, M) loop and counting the number that arrive back at M, using the fact that the loss rate for the loop should be $p(M, r)p(r, M)$. (We note that a result like Lemma 2.1 is needed if this method is to provide reliable estimates, a fact not observed in [8], which contained no such result.)

This scheme may require two monitoring hosts to measure the hop-on and hop-off rates for a path endpoint r, rather than the single one that would be required if we placed the monitoring equipment at vertex r itself. However, the scheme has the potential advantage that a given monitoring vertex can be reused to handle many different path endpoints. Thus there could be a substantial net overall savings in the total number of monitoring vertices used, and hence in equipment and operational cost.

As stated, the problem of finding a minimum sized set of monitoring vertices at which to place equipment so that we can estimate loss rates for all hop-on and hop-off paths is simply our original PATHWISE-DISJOINT FACILITY LOCATION problem. In practice, however, we will most-likely have to rely on the ISP's routing protocol (OSPF or IS-IS) to deliver our packets, and so, as with our the first application, will face the SETWISE-DISJOINT FACILITY LOCATION problem.

It should be noted that, in contrast to that first application, the necessity for *vertex*-disjoint paths from r to m_1 and m_2, rather than simply arc-disjoint paths, is less clear, since by the previous lemma we can only guarantee that these paths are arc-disjoint from the path from M to r. This is a meaningless distinction in the Setwise-Disjoint case, however, in light of the following lemma.

LEMMA 2.2. *Suppose $P(c, f)$ and $P(c, f')$ are the sets of all shortest paths from vertex c to vertices f and f', respectively, in a given digraph G. Then no path in $P(c, f)$ shares an arc with any path in $P(c, f')$ if and only if no path in $P(c, f)$ shares a vertex other than c with any path in $P(c, f')$.* [Proof omitted.]

A detailed description of the implementation of this scheme and the formulas used for estimating $p(M, r)$ and $p(r, M)$ is presented in [8].

3 Complexity

In this section we investigate the computational complexity of PATHWISE- and SETWISE-DISJOINT FACILITY LOCATION.

SCP is in general not only NP-hard, but also strongly inapproximable in the worst-case. Let $n = |U|$. In [9] it was observed that SCP is at least as hard to approximate as SET COVER, but we can prove much stronger inapproximability (albeit with a slightly stronger complexity assumption).

THEOREM 3.1. *If* NP $\not\subseteq$ DTIME($n^{O(\text{polylog}(n))}$), *no polynomial-time algorithm can be guaranteed to find a solution to SCP that is within a factor of* $2^{\log^{1-\epsilon} n}$ *of optimal for any* $\epsilon > 0$.

Proof of Theorem 3.1. The theorem follows via an approximation-preserving transformation from the MINREP problem of Kortsarz, who showed the above inapproximability bound to hold for MINREP [11].

In MINREP, we are given a bipartite graph $G = (V, E)$ with $n = 2kq$ vertices and partitions of the vertex sets on each side of the bipartite graph into k groups of q vertices, A_1 through A_k on the left and B_1 through B_k on the right. We are also given an integer K. We ask whether there is a subset $V' \subseteq V$ with $|V'| \leq K$ such that for every pair (A_i, B_j), $1 \leq i, j \leq k$, if G contains an edge between a vertex in A_i and one in B_j, then so does the subgraph induced by V'.

We transform MINREP to SCP by letting the items to be covered be the pairs (A_i, B_j) where G contains an edge between a member of A_i and a member of B_j. The set of covering objects is V, with the item (A_i, B_j) covered by all pairs $\{u, v\} \subseteq V$ in which $u \in A_i$, $v \in B_j$, and $(u, v) \in E$. There is then a one-to-one correspondence between SCP cover sets and subsets V' meeting the MINREP requirements, with the two sets having the same size. Hence any approximation guarantee for SCP implies an equally good guarantee for MINREP, and so SCP must be at least as hard to approximate. ■

In fact, the special cases PDFL and SDFL of SCP are in the worst case as hard to approximate as SCP itself, as a consequence of the following theorem. (The proofs of this and the next theorem will appear in the full version [3]):

THEOREM 3.2. *Even when restricted to instances in which* $C \subseteq F$,

1. *SDFL is at least as hard to approximate as SCP, even if each set* $P(w, u)$ *is the set of all shortest paths from* w *to* u.

2. *PDFL is at least as hard to approximate as SCP, even if each set* $P(w, u)$ *is an explicitly given subset of the set of all shortest paths from* w *to* u.

Theorem 3.2 leaves open the case of PDFL where the allowed paths include *all* shortest paths, not simply a selection of them, which we cannot yet prove is as hard as SCP in general. It is still $\Omega(\log n)$-hard to approximate, however, as a consequence of the following theorem:

THEOREM 3.3. *No polynomial-time approximation algorithm for PDFL can guarantee a solution that is* $c \log n$ *times optimum for any* $c < 1$ *unless* $P = NP$, *even if each set* $P(w, u)$ *is the set of* all *shortest paths from* w *to* u *and* $C \subseteq F$.

4 Algorithms

In this section we describe our algorithms for SETWISE- and PATHWISE-DISJOINT FACILITY LOCATION and our algorithms for constructing the required SCP triple sets for both problems.

4.1 Greedy Heuristics. The basic GREEDY heuristic that underlies all our algorithms has two phases. The first begins by initializing our cover $S' \subseteq S$ to the empty set. Then, as long as S' does not cover all $u \in U$, we find an $s \in S - S'$ such that $S' \cup \{s\}$ covers a maximum number of additional elements. If that maximum number is positive, then add s to S'. Otherwise, choose a *pair* $\{s, s'\} \subseteq S - S'$ that yields a maximum increase in the number of elements covered. If no pair yields an increase, then the instance is infeasible. The second phase *minimalizes* the cover: Test each object in the cover in turn to see if its removal would still leave a valid cover, and if so remove it. (The "Greedy" algorithm of [9] does not include a minimalization phase, and in its growth phase it considers pairs at every step, not just those in which no singleton helps. It is hence likely to be significantly slower and less effective in practice.)

Our implementation of GREEDY uses randomized tie-breaking in the growth phase. We allow two options for the minimalization phase: either consider facilities in the reverse of the order in which they were added to the cover, or consider them in random order. We also allow two options for starting the growth phase: either take the best pair, or take the best singleton. None of these options (or combinations of options) seems to dominate any of the others.

Here we test a multiple-run variant on GREEDY (GREEDY400), which performs 400 runs and returns the best solution found. In those 400 runs, we cycle through the four option combinations, using each for 100 runs. Multiple runs are feasible because we can implement the basic algorithm to run in linear time. (Technically the time is $O(|T| + |F|(|F| + |C|))$, where T is the set of triples, but for our instance classes $|T|$ tends to be the

dominant term.) The details of the implementation will be presented in the full paper.

4.2 Genetic Genetic algorithms are variants on local search that mimic the evolutionary process of survival of the fittest. Our genetic algorithm for SCP, called GENETIC in what follows, uses the "random key" evolutionary strategy proposed by Bean [1, 7]. In this approach, the "chromosomes" that do the evolving are not solutions themselves, but "random-key" vectors from which solutions can be derived.

Let the set of cover objects be $S = \{s_1, s_2, \ldots, s_k\}$. In GENETIC, each chromosome is a 0-1 vector $(gene_1, \ldots, gene_k)$ of length k. We derive a solution (cover) S' from such a chromosome as follows: Start with $S' = \{s_i : gene_i = 1,\ 1 \le i \le k\}$. If S' is already a cover, halt. Otherwise, complete S' to a cover using our basic GREEDY algorithm (without randomization or minimalization). The "fitness" of the chromosome is then the size of the resulting cover S'.

The overall genetic algorithm starts by creating a population of p randomly generated chromosomes, where each gene has an equal probability of being 0 or 1. (For our experiments, we set $p = \min\{300, |V|\}$.) The population then evolves in a sequence of generations. In each generation we start by computing the solution for each member of the population, yielding its fitness. The top 15% of the population by fitness (the "elite" members) automatically pass on to the next generation's population. In addition, to provide diversity, 10% of the next generation consists of randomly generated chromosomes, like those generated initially. For the remaining 75%, we repeat the following "crossover" construction: Pick a random member $(x_1, x_2, \ldots, x_k)$ of the top 15% of the current generation's population and a random member $(y_1, y_2, \ldots, y_k)$ of the remaining 85%. The "child" $(z_1, z_2, \ldots, z_k)$ of this pairing is determined as follows: For each i, independently, set $z_i = x_i$ with probability 70%, and otherwise set $z_i = y_i$.

This scheme insures that the best solution always survives into the next generation, where it may continue as champion or be dethroned by a better solution. Generations are continued until $\max\{p, 50\}$ have gone by without any improvement in the fitness (cardinality) of the best solution. We then take this final champion S' and "minimalize" it as in Phase 2 of GREEDY (although, in practice, this rarely helped for the final champion).

The parameter settings for this algorithm ($p = \min\{300, |V|\}$, $\max\{p, 50\}$ generations, top 15%, etc.) were based on preliminary experimentation and on intuition derived from applications of this approach to other problems. They appear to yield a good tradeoff between running time and quality of solution for our application, but we make no claim as to their optimality.

4.3 The Double Hitting Set Heuristic (HH) This section describes our Double Hitting Set Heuristic (HH). HH is the only one of our algorithms to explicitly exploit the structure of the digraph G, and the only one to come with an $O(\log n)$ performance "guarantee," "guarantee" in quotes because it only refers to covering the "t-good" customer nodes, "t-good" to be defined in a moment. (The key fact is that in many of our test instances, almost all customer nodes were t-good.) HH applies to the case in which the sets $P(c, f)$ of paths consist of all the shortest paths from c to f, as would be the case if they were generated by OSPF (as they are in our test instances and our main applications).

While OSPF can split traffic in the case of multiple shortest $c \to f$ paths, the amount of splitting that occurs in practice seems to be limited. To quantify this point, we need a definition.

DEFINITION 4.1. *We say a potential facility location f is* good for *customer vertex c if $f \neq c$ and all shortest $c \to f$ paths leave c via the same arc. Equivalently, if $wt(a, b)$ denotes the weight of arc (a, b) and $dist(a, b)$ denotes the distance from a to b, then a potential facility location f is good for customer vertex c if $f \neq c$ and $wt(c, v) + dist(v, f) > dist(c, f)$ for all out-neighbors v of c but one.*

If there is just one shortest $c \to f$ path, clearly f is good for c, but f may be good for c even when there are many $c \to f$ paths. Note that if f is good for c, then, under OSPF, there can be no splitting at c of the traffic from c to f, although splitting at later vertices in the path would be possible.

Algorithm HH attempts to construct a cover in which many of the customer vertices are covered by potential facility locations that are good for them.

DEFINITION 4.2. *Say that a customer vertex c is* t-good *if there are t or more good potential facility locations f for c (and* t-bad *otherwise), where t is a parameter of the algorithm.*

For our experiments, we ran the algorithm with both $t = \lfloor |F|/2 \rfloor$ and $t = 1$, and output the better result.

HH is designed to nearly optimally cover the t-good customers via a union $X \cup Y$, leaving the t-bad customers, if any, to be covered via the GREEDY algorithm.

The first step of the algorithm is to choose one good potential facility location for each t-good customer vertex c, without choosing too many vertices in total. For each such c, let S_c be the set of good potential facility locations for c, except that we add c to S_c if c is a potential facility location (which, by the definition of "good," is not good for itself). By the definition

of t-good we must have $|S_c| \geq t$. We want to choose a small set X of potential facility locations such that $X \cap S_c \neq \emptyset$ for all t-good customer vertices c. In other words, we want an X that hits (intersects) all such sets S_c. Finding a minimum-cardinality such X is the classic HITTING SET problem for the family $\{S_c | c$ is a t-good customer vertex$\}$.

The greedy algorithm for HITTING SET produces a feasible solution of size at most $1 + \ln n$ times the optimal, n being the number of sets in the instance [10, 12], which for our application is at most $|C|$. In practice, however, the greedy HITTING SET heuristic typically returns a set of size much closer than this to the optimal, often only a single-digit percent above optimal. So our first step is to construct a set X by applying this greedy hitting set algorithm to $\{S_c | c$ is a t-good customer vertex$\}$. In fact, the situation is even better here. We will see shortly that if $t = \lfloor |F|/2 \rfloor$, then a greedy hitting set has size $O(\log |C|)$.

Next, for each t-good customer vertex c, choose a potential facility location s_c which is in $X \cap S_c$, but if $c \in (X \cap S_c) \cap F$, definitely set $s_c = c$. It is easy to see that $s_c = c \Leftrightarrow c \in X$.

If $s_c \neq c$, then all shortest $c \rightarrow s_c$ paths leave c via the same arc; call this arc $e_c = (c, x_c)$, defining x_c implicitly. If, on the other hand, $s_c = c$, then $c \in X$ and c is therefore already covered by X; since we are going to cover the t-good nodes by $X \cup Y$ for some Y, we needn't worry further about covering such c's.

We now have, for each t-good c which is not in X, at least one chosen potential facility location $s_c \neq c$, but we need two. We find a second as follows. For each such customer vertex c, let F_c be the set of potential facility locations $f \neq c$ for which all shortest $c \rightarrow f$ paths *avoid* e_c, together with c if c is itself a potential facility node. (A vertex $u \in F - \{c\}$ is in F_c if and only if $wt(c, x_c) + dist(x_c, u) > dist(c, u)$.) If any set F_c is empty, then the instance is infeasible – clearly $c \notin F$ in this case, and for all potential facility locations f, there will be a shortest path from c to f that contains e_c and so no two such facilities can cover c in setwise-disjoint fashion. So suppose that for all c, F_c is not empty. The surprising fact is that for any $f \in F_c$, the pair $\{s_c, f\}$ covers c, an immediate consequence of Lemma 4.2 below.

Recall that for a t-good $c \in C$, $s_c \neq c \Leftrightarrow c \notin X$, so we may assume that $s_c \neq c$. The following lemma, proved below, will allow us to compute an extremely useful lower bound on the optimal cost.

LEMMA 4.1. *Any solution W that covers all t-good nodes must contain at least one facility node u in F_c for each t-good c with $s_c \neq c$.*

Thus any solution W must contain a hitting set Y for the family $\{F_c | c$ is t-good and $s_c \neq c\}$. So the next thing HH does is find one, again using the greedy HITTING SET heuristic. Fortunately, not only is the size of the optimal hitting set a lower bound on OPT, the set $X \cup Y$ is a *feasible* solution, for any feasible hitting set, as we show in Lemma 4.2 below. When $t = \lfloor |F|/2 \rfloor$, the resulting union $X \cup Y$ is a particularly good solution to the restricted problem, at least in comparison to the complexity results we saw in the last section for solutions to the complete SDFL problem.

Algorithm HH finishes by using the GREEDY algorithm to extend the solution $X \cup Y$ for the t-good nodes to a solution for the full SDFL instance. Since there are typically few t-bad nodes, this step typically adds few nodes. Finally, we minimalize the resulting solution (as in GREEDY).

This completes the description of HH.

PROOF OF LEMMA 4.1. If not, choose t-good c such that $s_c \neq c$ and a feasible W with $W \cap F_c = \emptyset$. Choose any $u \in W$; $u \notin F_c$. Hence $u \neq c$ ($u \in W$ implies that u is a facility node; were u equal to c, then c would be a facility node, c would be in F_c, and hence $u = c$ would be, too). Hence some shortest $c \rightarrow u$ path uses arc (c, x_c). Now x_c is in $Q(c, u)$, where $Q(a, b)$ denotes the set of vertices on *any* shortest $a \rightarrow b$ path. But this is true for all the u in W, and hence for any $i \neq j$ in W, x_c is in $Q(c, i)$ and $Q(c, j)$, so $\{i, j\}$ *doesn't* cover c (and $c \notin W$, so $\{c\}$ doesn't cover c, either). So W is infeasible, a contradiction. ■

Hence the size of any solution covering all t-good nodes is greater than or equal to the size of the smallest hitting set of the family $\{F_c | c$ is t-good and $s_c \neq c\} = \{F_c | c$ is t-good and $c \notin X\}$. Based on Lemma 4.1, we know that the size $H_{\min}$ of the smallest hitting set is a lower bound on the optimal solution size for an instance of SDFL. It is easy to set up an integer program for determining $H_{\min}$, and it turns out that CPLEX finds such problems much easier to solve than the full SDFL problem. Thus we have our code HH save this HITTING SET instance to a file, which in turn is fed to AMPL/CPLEX, which (quickly) solves it optimally. The LOWERBOUND figures in the Results section of this paper were computed in this way, taking the maximum for $t = 1$ and $t = \lfloor |F|/2 \rfloor$. (One could have used optimal AMPL/CPLEX-solved HITTING SET solutions instead of greedy solutions in implementing HH, but this would have made running the algorithm a more complex process and one that no longer had a worst-case polynomial-time bound. Moreover, given how good the HH solutions already were, it would probably not have been worth the effort.) Note that since the lower bound does not take t-bad customer vertices into account, it is

probably weak in cases that have many such nodes.

LEMMA 4.2. *Given X as above, any hitting set Y of $\{F_c | c$ is t-good and $s_c \neq c\}$ is such that $X \cup Y$ is a cover of all the t-good customer nodes c.*

Proof. Choose any t-good c. If $c \in X \cup Y$, then c is a facility node and hence $\{c\} \subseteq X \cup Y$ covers c. So we may assume that $c \notin X, c \notin Y$. Hence $s_c \neq c$. Since $s_c \neq c$, all of the $c \to s_c$ shortest paths use arc (c, x_c). The hitting set Y contains a vertex $f_c \in F_c$. Because $c \notin Y$ and $f_c \in Y$, $f_c \neq c$. Hence, because $f_c \in F_c - \{c\}$, by definition of F_c no shortest $c \to f_c$ path uses arc (c, x_c). We claim that *all* the $c \to f_c$ shortest paths actually avoid *all* the vertices (except c) on *all* the $c \to s_c$ shortest paths, and hence $\{s_c, f_c\} \subseteq X \cup Y$ covers c. (By the definitions of S_c and F_c, $f_c \neq s_c$.) This is true because if a $c \to s_c$ shortest path P_1 and a $c \to f_c$ shortest path P_2 shared a vertex other than c, let v be the first vertex (after c) on P_2 which is in P_1. Then the prefix from c to v of P_1 would have to have the same length as the prefix from c to v of P_2—this is the key point—and hence there would be a shortest $c \to f_c$ path which uses arc (c, x_c), a contradiction. Hence $X \cup Y$ covers all the t-good customer nodes. ■

Let OPT′ denote the optimal solution to the SDFL problem restricted to the t-good customer vertices. Lemma 4.3 is the performance "guarantee" of which we spoke.

LEMMA 4.3. *If $t = \lfloor |F|/2 \rfloor$, then $|X \cup Y|$ is $O((\log |C|) \cdot$ OPT′).*

Proof. Assume there is at least one t-good customer, as otherwise $|X \cup Y| = 0$ and the claim holds trivially. By Lemma 4.1, an optimal solution to the restricted problem must contain a hitting set for the sets F_c where c is a t-good customer vertex with $s_c \neq c$. Thus, by the results of [10, 12] we have $|Y| \leq (1 + \ln |C|)\text{OPT}'$.

We cannot use the same argument for $|X|$, since in constructing X we restricted attention to good potential facility locations, and OPT′ need not be so restricted. However, a different argument applies. By our choice of t, we know that for each t-good vertex c, $|S_c| \geq \lfloor |F|/2 \rfloor \geq |F|/3$. Therefore, $\sum_{c \text{ is } t\text{-good}} |F_c| \geq |C| \cdot |F|/3$, and so some potential facility location f must be in at least $|C|/3$ of the sets. Consequently, the greedy choice must hit at least that many sets. By essentially the same argument, it must hit at least 1/3 of the remaining unhit sets each time it makes a choice, and so must have completed constructing its hitting set X after at most $2 + \log_3 |C|$ steps. Since there is at least one t-good vertex, we must have OPT′ ≥ 1, and the Lemma follows. ■

4.4 Optimal Solutions Using Mixed Integer Programming. SET COVER BY PAIRS can be modeled as a simple mixed integer linear program (MIP), which allows us to leverage general purpose commercial software for solving such problems. Our MIP formulation has a Boolean variable x_s for each $s \in S$, where $x_s = 1$ if s is chosen for our cover. In addition, we have a real nonnegative variable $y_{s,s'}$ for each pair $\{s, s'\}$ of (not-necessarily distinct) elements of S, subject to the constraints that $y_{s,s'} \leq x_s$ and $y_{s,s'} \leq x_{s'}$. Note that these together imply that $y_{s,s'}$ can be positive only if both x_s and $x_{s'}$ equal 1 (i.e., are in the chosen cover). In addition we have the following constraints, one for each $u \in U$, in order to guarantee that the chosen set is a cover: $\sum_{s,s':(u,s,s') \in T} y_{s,s'} \geq 1$.

The goal of our MIP is to minimize $\sum_{s \in S} x_s$. Each feasible solution to the given MIP corresponds to a feasible solution $S' \subseteq S$ to the SCP and vice versa, with S' being the set of cover objects s for which $x_s = 1$.

We used AMPL™ to turn our SCP instances into MIP instances and called version 11.0 of CPLEX™ to solve them. This MIP approach proved practical for surprisingly large instances, enabling us to find optimal solutions to all but two of the instances in our test set with $|F| \leq 150$. Our detailed experimental results will be summarized in Section 6.

4.5 Triple Generation. In what follows, let n and m be the numbers of vertices and arcs in our graph G. For our applications, we may assume that $m \leq an$ for some relatively small constant a, given the structure of real-world data networks. We can also expect that the sizes of the sets C of customers and F of potential facility locations to be proportional to n.

A triple (c, f_1, f_2) is in the SCP instance corresponding to a given instance of SETWISE-DISJOINT FACILITY LOCATION if $c \in C$, $f_1, f_2 \in F$, and no shortest path from c to f_1 shares any vertex other than c with any shortest path from c to f_2. The naive way to test this would be to construct, for each f_i, the set S_i of vertices on shortest paths from c to f_i, and then testing whether $S_i \cap S_j = \{c\}$. These sets could conceivably be of size proportional to n, yielding a running time that could be proportional to $|C||F|^2 n$, so the best we can say about the running time of such an algorithm is that it is $O(n^4)$.

Fortunately, it is an easy observation that if $S_i \cap S_j$ contains some vertex other than c, then it contains a vertex that is a neighbor of c. Thus we may restrict attention to the sets N_i of vertices adjacent to c that are on shortest paths to f_i. To compute these sets we first construct a shortest path graph from c to all other vertices in time $O(m \log n)$, and then for each

neighbor v of c identify the sets N_i that contain it by an $O(m)$-time search of the tree. For each pair f_i, f_j, the intersection test can then be performed in time $O(outdegree(c))$, yielding an overall running time that is $O(|C|m \log n + m^2 + m|F|^2) = O(n^3)$ under our assumptions about the underlying graphs.

For the pathwise-disjoint problem, the naive algorithm is even worse, since for a given triple c, f_1, f_2, there may be exponentially many paths of potential length $\Omega(n)$ to compare. Here we must be more clever. We first observe that we can actually reduce the test to a simple network flow problem, obtained by adding new arcs from f_1 and f_2 to a new vertex t, giving all vertices capacity one, and asking whether there is a flow of value 2 from c to t. Next, we observe that this is a particularly easy flow problem, which can be solved in linear time by starting with a shortest path from c to t via f_1 and then looking for an augmenting path. Finally, we observe that, having fixed the shortest path to f_1 and constructed the residual graph, we can actually solve the network flow problems for all pairs (f_1, f_i) in parallel with a single breadth-first search. The overall running time thus becomes $O(|C|m \log n + |C||F|m)$, once again $O(n^3)$ under our assumptions.

5 Network Test Instances

We tested our algorithms on both synthetic and real-world instances. The two classes modeled different types of data networks and had distinctly different characteristics, enabling us to test the generality of our methods. In this section we will discuss each type of instance, and compare the structural properties of the resulting testbeds.

5.1 Synthetic LAN/WAN Instances The synthetic instances were designed to reflect the structure of large real-world local- and wide-area networks (LAN's and WAN's) and were sized so that we could study the scalability of our algorithms and the solutions they produce. They were generated via the following four-step process.

1. A transit-stub skeleton graph is generated using the Georgia Tech Internetwork Topology Models (GT-ITM) package [6]. We generated 10 graphs each for parameter settings that yielded $|V| = 26, 50, 100, 190, 220, 250, 300, 558$. (The value of $|V|$ is an indirect result of one's choice of the allowed input parameters.)

2. Traffic demand is generated between all pairs of vertices in the skeleton graph using a gravitational model (described below) with the shortest path metric.

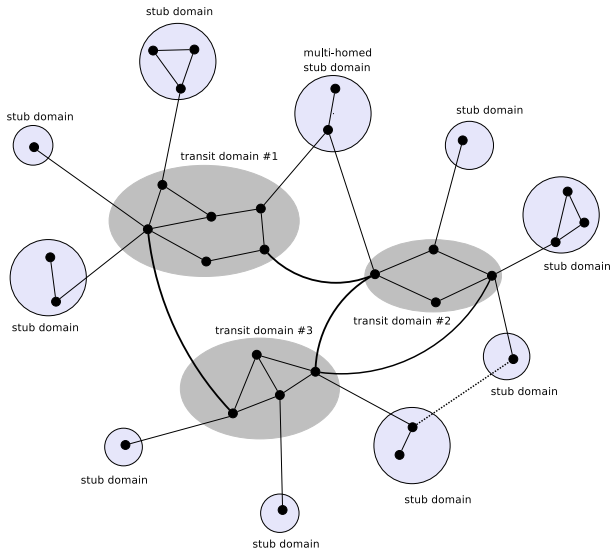

Figure 3: A transit stub network with three transit domains and ten stub domains. One stub domain homes in on transit domains 1 and 2. Two stub domains are linked by a stub to stub edge.

3. We determine link capacities and OSPF link weights such that all traffic can be routed on the resulting network using the OSPF routing protocol [5].

4. Given the desired numbers of customer and facility location vertices, the sets C and F are randomly generated, with $C \subseteq F$ to insure feasibility. Let (Cx,Fy) denote the set of instances with $|C| = \lceil |V|/x \rceil$ and $|F| = \lceil |V|/y \rceil$. For each graph we generated seven instances, one each of type (C1,F1), (C2,F2), (C4,F4), (C8,F8), (C2,F1), (C4,F1), and (C8,F1).

5.1.1 Transit-Stub Skeleton Graphs Transit-stub graphs [6] are hierarchical graphs made up of transit vertex components and stub vertex components. The stub node components can be thought of as access networks while the transit node components make up the backbone of the network. See Figure 3.

The GT-ITM package provides a parameterized method for generating such graphs randomly. It constructs undirected graphs, which we then view as directed graphs with arcs (u, v) and (v, u) in place of every edge $\{u, v\}$. The constructed graphs consist of T *transit domains*, each containing an average of N_T transit vertices and an average of E_T internal edges connecting them, with an average E_{TT} edges joining vertices of the domain to vertices in other domains. In addition, there are an average of S *stub domains* per transit vertex, each containing an average of N_S stub vertices and

$\|V\|$	$\|A\|$	T	N_T	S	N_S	E_T	E_{TT}	E_{ST}	E_S
26	69.4	1	3	2	8	2.3	0.0	2.0	7.5
50	182.2	1	2	3	8	1.0	0.0	4.0	41.0
100	354.4	1	4	3	8	4.2	0.0	3.0	41.2
190	569.2	2	5	3	6	6.6	2.0	3.0	23.6
220	720.4	2	5	3	7	5.4	2.0	3.0	30.9
250	870.0	2	5	3	8	6.6	2.0	3.0	39.1
300	1063.2	2	6	3	8	8.8	2.0	3.0	39.8
558	2344.4	3	6	3	10	8.4	4.0	3.0	60.1

Table 1: Measured parameters of generated transit-stub graphs.

E_S internal edges, along with an average of E_{ST} edges connecting vertices of the stub domain to transit vertices. In addition, the program adds additional edges if necessary to insure that the graph is connected, and on its own occasionally adds edges between different stub domains (this only happened once, for one of our 50-vertex graphs). See [6] for a description of the random process that produced networks of this form and guarantees that they are connected.

We generated 10 graphs each for eight different sets of parameters yielding increasing values for $|V|$. Note that the number of vertices is not an explicit parameter of the graph generation process, but rather is a function of the values of the other parameters, which helps explain the fact that the values of $|V|$ do not increase uniformly. The package also does not allow for direct input of E_T and E_S, but instead asks for the probability that any pair of vertices in a transit domain/stub is an edge. For all our graphs we set these probabilities to 0.6 and 0.42, respectively. In Table 1 we present the measured averages for the parameters for each value of $|V|$. Note that each edge $\{u, v\}$ in the generated graph is represented by the two directed arcs (u, v) and (v, u) in the derived network.

5.1.2 Traffic Demands The traffic demands are created via a randomized "gravitational" method. We first generate random numbers $o(v), d(v) \in [0, 1]$ for each vertex v, and, for each pair of vertices (u, v), compute $dist(u, v)$, the shortest length (in edges) of a path from u to v. Let $Dmax$ be the largest of these distances. Then, for any ordered pair (u, v) of distinct vertices, we choose a random number $r \in [0, 1]$ and set the traffic demand from u to v to be

$$r \cdot e^{-\left(\frac{dist(u,v)}{2Dmax}\right)} \cdot o(u) \cdot d(v) \; .$$

5.1.3 OSPF Routes Given the skeleton graph and the traffic demands, we apply the algorithm of [5] that computes link capacities and corresponding OSPF weights under which the traffic can be efficiently routed. The OSPF weights then become the basis for determining the shortest (i.e., lowest-weight) paths between all pairs of vertices, with the set of paths for the pair (u, v) being represented implicitly by the graph consisting of the union of all the edges contained in the paths.

5.1.4 Customers and Potential Facility Locations For each of our skeleton graphs, we generated seven networks, differing in their choices of the sets F and C of potential facility locations and of customer vertices. Our synthetic instance testbed thus contains 70 different networks for each value of $|V|$, for a total of 560 networks.

5.2 Real-World ISP Instances The real-world instances in our testbed were derived from five proprietary Tier-1 Internet Service Provider (ISP) backbone networks and used actual OSPF weights. The networks ranged in size from a little more than 100 routers to nearly 1,000, each with between $3.5|V|$ and $4.3|V|$ edges (similar to the range for our synthetic instances). We shall denote them by R100a, R100b, R200, R500, and R1000, where the number following the R is the number of routers, rounded to the nearest multiple of 100.

We constructed 16 instances from each of the four smaller topologies, starting with the case in which $F = C = V$. The other instances also had $F = V$, but C was a random sample of roughly 1/2, 1/4, or 1/8 of the vertices. For each combination of a topology from the set R100a, R100b, R200, and R500 and a sample size, we generated five distinct instances, based on different random choices for C.

For the largest network, R1000, where we have more detailed information about the roles played by the routers, we constructed an instance which took those roles into account: access routers served as the customer vertices and the potential facility locations consisted of the access and aggregation routers (modeling the situation in which measurement hosts could not be connected directly to backbone routers).

5.3 Instance Properties Our synthetic instances reflect the structure of real-world LANs and WANs, and consist of many, relatively small 2-connected components, the largest averaging only 18% of the vertices (12% when $|V| > 50$). In contrast, the Tier-1 ISP instances consist of one very large 2-connected component, containing over 90% of the vertices on average, with small 2-connected components hanging off of it. This results in the ISP instances yielding substantially more triples than the synthetic ones of the same size. For example, R500 has roughly four times as many triples as the (C1,F1) instances with $|V| = 558$.

The choice of arc weights also has an effect on our instances. The optimized weights of our syn-

thetic instances lead to relatively few shortest-path ties, and so the pathwise-disjoint instances average only 5% more triples than the corresponding setwise-disjoint instances. The weights in the ISP instances were constrained by a variety of factors in addition to traffic balancing, and yield far more ties. As a result, the pathwise-disjoint instances on average have roughly 50% more triples. If, in our synthetic instances, we set all the edge weights equal, rather than optimizing them, we get even more ties and more extreme results. The number of setwise-disjoint triples drops by about 10% on average, but the number of pathwise-disjoint triples averages 80% above that (and 45% above the number of such triples when weights are optimized). We also observe that the number of triples grows worse than quadratically with the number of vertices for each synthetic class (Cx,Fy), and there is at least a quadratic blow-up for our ISP instances, with the number of triples for R1000 approaching 50 million.

6 Summary of Experimental Results

This section evaluates our algorithms in three areas.

- *Accuracy:* How accurate were the approximations provided by each of the algorithms GREEDY, GENETIC and HH? When OPT is known, we compare to OPT, and otherwise to the HH-derived LOWERBOUND.
- *Execution Time:* How fast are the algorithms? The times we report were for runs on an SGI shared multiprocessor machine with 32 1.5 Ghz Itanium (IA-65) processors and 256 gigabytes of memory. (Each algorithm run used only a single processor.)
- *Solution Quality (Cost Reduction):* The first two measures address the quality of the algorithms. This one addresses what they tell us about the applications. Even an optimal solution will not be enough to justify the proposed monitoring scheme of [8] if it does not provide a significant savings over the default solution that simply takes *all* customers (which for our instances are themselves facility locations) and is likely to yield more reliable measurements. We also consider how much further improvement is obtainable when going from the setwise-disjoint to the pathwise-disjoint versions of the problem.

6.0.1 Accuracy for Synthetic Instances In Table 2, we present the average values of the lower bounds computed by our HH heuristic for our 56 classes of synthetic instances. This provides an idea of how solution values vary depending on class and value for $|V|$. The table also reports how many of the 10 instances of each type that our MIP code was able to solve within a 24-hour time bound. For $|V| = 190$, we eventually solved the four missing (C4,F1) and (C8,F1) instances, although the longest (C4,F1) took slightly more than a week. Note, however, that 341 of the 370 solved instances took less than 10 minutes. For those class/size combinations in which MIP solved all 10 instances, we also present the average amount by which the optimal solution exceeded LOWERBOUND.

The HH lower bounds are very good. As shown in the table, the average values of OPT − LOWERBOUND never exceed 0.8 in any of the cases with $|V| < 558$. This can still result in fairly large percentage differences in those cases where OPT is small (16% for (C8,F8) when $|V| = 26$), but for all cases in which $|C| > 15$, the percentage gap is less than 3.8%. Also, as shown in the table, the lower bound is often optimal even

Class	26	50	100	190	220	250	300	558
Average Values of LOWERBOUND								
(C1,F1)	8.7	10.7	24.0	53.8	59.2	60.7	71.1	87.5
(C2,F1)	5.8	7.5	17.1	38.2	39.5	42.8	50.0	67.7
(C4,F1)	4.0	6.1	11.7	26.8	28.2	29.5	36.3	52.2
(C8,F1)	3.1	3.9	7.9	16.1	19.1	20.5	24.5	38.7
(C2,F2)	5.9	7.8	17.7	39.2	43.5	45.9	53.6	70.4
(C4,F4)	3.8	5.6	11.9	26.0	30.7	32.2	37.4	57.8
(C8,F8)	2.6	3.9	7.8	16.4	19.9	20.2	25.2	40.6
Number of Instances Solved by MIP								
(C1,F1)	10	10	10	-	-	-	-	-
(C2,F1)	10	10	10	-	-	-	-	-
(C4,F1)	10	10	10	7	-	-	-	-
(C8,F1)	10	10	10	9	-	-	-	-
(C2,F2)	10	10	10	10	10	10	10	-
(C4,F4)	10	10	10	10	10	10	10	10
(C8,F8)	10	10	10	10	10	10	10	10
Average Value of OPT−LOWERBOUND								
(C1,F1)	0.0	0.0	0.1	-	-	-	-	-
(C2,F1)	0.0	0.0	0.0	-	-	-	-	-
(C4,F1)	0.0	0.0	0.0	0.0	-	-	-	-
(C8,F1)	0.0	0.2	0.1	0.0	-	-	-	-
(C2,F2)	0.2	0.1	0.0	0.4	0.4	0.5	0.5	-
(C4,F4)	0.4	0.5	0.4	0.6	0.6	0.2	0.7	2.1
(C8,F8)	0.5	0.7	0.5	0.6	0.2	0.8	0.8	1.4
Number of instances, of 10, for which LOWERBOUND is known to equal OPT								
(C1,F1)	10	10	10	2	8	4	2	0
(C2,F1)	10	9	10	9	9	6	5	1
(C4,F1)	10	10	10	10	10	6	6	1
(C8,F1)	10	10	10	10	9	8	10	3
(C2,F2)	10	10	10	10	10	10	10	1
(C4,F4)	10	10	10	10	10	10	10	10
(C8,F8)	10	10	10	10	10	10	10	10

Table 2: LOWERBOUND versus OPT for synthetic instances.

for the larger values of $|V|$ where MIP fails. We are able to conclude this whenever one of our heuristics finds a solution that matches the lower bound, which happened for 110 of our instances for which MIP failed, bringing the total number of instances for which optima are known to 480 of 560. The gap between LowerBound and the best solution known begins to widen for $|V| = 558$, although we do not at this point know how to apportion the blame between the quality of LowerBound or of our heuristics. In what follows, our standard of comparison for accuracy will be the true optimum when it is known, and LowerBound only if the true optimum is not known. We shall call this our "lower bound" as opposed to LowerBound.

The strength of LowerBound might seem surprising, given that it corresponds to the size of just one of the two disjoint hitting sets that are put together by the HH heuristic. However, the first hitting set tends to be quite small, and recall that the HH solution can be significantly smaller than the sum of the sizes of the two hitting sets, given the final minimalization pass that deletes redundant vertices from the cover. However, the quality of the bound does depend on there being a high proportion of t-good customer vertices, and the average percentage of such customers that are t-good when $t = \lfloor |F|/2 \rfloor$ is dropping as $|V|$ increases for these instances (from 97.7% to 96.0% as we go from $|V| = 300$ to 558).

For instances of the type studied here, all of our heuristics come very close to our lower bounds. This is in contrast to our complexity results that suggest that no polynomial-time heuristic can do well against a malicious instance designer. Leading the pack, Greedy400 found the optimal solution for all the instances where it is known, and found the best solutions we know on all of the other instances, averaging 0.76% above the lower bound for all instances with $|V| \geq 100$, with no individual class average excess exceeding 2.41%. (We ignore results for our instances with $|V| < 100$, as those seem too easy. Each is solved by MIP in 15 seconds or less, and Greedy400 takes a fraction of a second and always found that optimal.) The corresponding percentages for basic Greedy, based on taking the median result from the 400 runs of Greedy400, were 0.90% and 2.97%. For Genetic, based on just one run, they were 0.86% and 2.74% and for HH, they were 1.36% and 4.00%. In all three cases, the worst results were for class (C1,F1) with $|V| = 558$. None of these three latter algorithms dominate any of the others, although Genetic can be viewed as marginally the best, while HH is the worst – Genetic finds solutions better than those of Greedy 17 times, and is only beaten 9 times. In comparison to HH, its score is 68 to 7. Greedy beats HH 65 to 7.

A similarly mixed story holds for the corresponding pathwise-disjoint instances, where once again Greedy400 finds the optimal solution whenever it is known, and is never beaten by any of the other heuristics. Note that although the theory justifying HH only applies to the setwise-disjoint case, it still can be run on pathwise-disjoint instances, and still produces reasonable solutions. It no longer produces valid lower bounds, however, so for these instances we cannot evaluate our heuristics' excess over optimal except on those instances where MIP succeeded in a feasible amount of time.

The results for Greedy400, although they potentially measure the tail of a distribution, are actually quite repeatable here. For none of the 560 synthetic instances did the value of the "best-of-400" runs occur in fewer than 19 of the runs. If we assume this distribution mirrors the actual distribution, then the probability of failing to find that best is no more than $(381/400)^{400} < 10^{-8}$. We cannot be as confident in our results for Genetic, since there, because of the running times involved, we rely on the value of a single run. We shall, however, investigate this issue in the full paper.

6.0.2 Execution Times When we began this study, our main concern was to evaluate the proposed monitoring scheme, and hence our emphasis was on measuring solution quality, with less concern for running time. Consequently, none of our implementations (which at that time did not include Greedy400)) was particularly efficient. This remains the case for HH and, in part, for Genetic. We have, however, reimplemented Greedy. Each of our heuristics was originally implemented by a different co-author, and involved slightly different tie-breaking rules in the implementation of the underlying greedy heuristic. The three implementations consequently did not yield entirely consistent results when their code was adapted to just run the greedy heuristic. The current version of Greedy was implemented to explore these differences, and consequently was designed with efficiency more in mind. Moreover, typically 95% or more of its running time is spent in reading the input and setting up its initial data structures, making the performance of multiple runs on the same instance quite cost effective and leading us to add Greedy400 to our algorithmic mix.

See Table 3, which lists our algorithms' average running times for our largest synthetic instances, those with $|V| = 300$ and $|V| = 558$, and all our seven instance classes. In the full paper, we will report on re-implemented versions Genetic and HH using the new Greedy implementation. We expect HH to have a time competitive with single-run-Greedy. Based on preliminary experiments, re-implementing Genetic

Class	Triple Gen	Greedy 1	Greedy 400	HH	Genetic
	$\|V\| = 300$				
(C1,F1)	1.38	.74	15.80	12.86	6281.6
(C2,F1)	0.68	.35	2.79	7.00	2159.3
(C4,F1)	0.36	.17	.94	4.49	800.7
(C8,F1)	0.19	.09	.13	3.64	303.8
(C2,F2)	0.22	.10	.12	4.97	233.0
(C4,F4)	0.05	.02	.02	3.22	7.9
(C8,F8)	0.02	.00	.00	3.11	0.4
	$\|V\| = 558$				
(C1,F1)	7.47	4.07	70.96	65.18	77528.6
(C2,F1)	3.69	2.14	32.13	30.54	34866.0
(C4,F1)	1.92	1.10	17.46	15.48	6422.0
(C8,F1)	0.93	.46	6.17	8.06	1612.6
(C2,F2)	1.14	.54	10.08	11.00	4445.8
(C4,F4)	0.22	.08	2.01	4.32	173.2
(C8,F8)	0.06	.02	.89	3.71	5.5

Table 3: Average running times in seconds on a 1.5 Ghz Itanium processor for Setwise-Disjoint instances. Triple generation for the Pathwise-Disjoint instances takes roughly twice as long as for Setwise-Disjoint ones, with the ratio growing slightly with $|V|$.

should also speed it up significantly, although the improvement may not be so dramatic. Note, however, that even with the current implementations, our results suggest that Greedy400 and HH are both fast enough to be practical even for much larger instances than studied here, and our triple generation code will not add a substantial overhead.

6.0.3 Cost Reduction. In this section we consider the savings our heuristics can provide over simply choosing the cover consisting of the set C of all customer vertices (always a feasible solution since our instances all have $C \subseteq F$, and the default solution for network path monitoring if we do not use our proposed monitoring scheme). For a given class these savings do not appear to vary strongly with $|V|$. Therefore, we can simplify things and report only a single average improvement for each class, as shown in Table 4. Here we present the average percentage reduction in cover size produced by Greedy400, as well as the reduction that could be obtained if we could find covers matching our best lower bounds for all instances. As can be seen, the savings are substantial. The smallest average savings generated by (setwise-disjoint) Greedy400 is roughly 32% and the largest is 75%. In addition, the table shows the further reduction we could obtain if, instead of considering the setwise-disjoint version of the problem, we used the pathwise-disjoint solutions. The improvements are more moderate, but the latter does improve on the former

Class	Set-Disjoint Greedy400	UB	Path-Disjoint Greedy400	Δ
(C1,F1)	75.1	75.3	76.1	4.9
(C2,F1)	65.3	65.4	66.2	3.4
(C4,F1)	50.9	51.5	51.7	1.8
(C8,F1)	35.2	35.4	35.6	0.7
(C2,F2)	63.2	63.3	64.4	3.9
(C4,F4)	48.3	48.3	49.4	2.3
(C8,F8)	32.1	32.1	32.9	1.1

Table 4: Percentage reduction in cover size compared to $|C|$. "UB" means the best possible improvement in the setwise-disjoint case, given our lower bound on the optimal solution for that case. "Δ" means the percentage by which Greedy400's pathwise-disjoint solution improves on its setwise-disjoint solution.

for 241 of our 560 instances, and for each class the average beats that for the setwise-disjoint upper bound. The overall average improvement is 3.2%, which works out to almost saving a full facility. A further .07% improvement can be obtained if we are willing to settle for arc- rather than vertex-disjoint paths, which might make sense in our monitoring application, where all the relevant lemmas would still apply. Arc-disjointness provided an improvement on 10 of our 560 instances.

6.1 Results for Real-World ISP Topologies. The results for our 65 instances based on actual ISP backbone topologies were similar to those for our synthetic instances, although, as we shall see, there were a few significant differences. Our MIP code once again was able to solve all our instances with $|F| < 150$, in this case the ones based on R100a and R100b, with a median running time of 16 seconds and a maximum of 10 minutes. We were also able to solve all the (C4,F1) instances based on R200, although the maximum here was 7 hours, and we solved four of the five (C2,F1) R200 instances, in times ranging form 2 hours to 2.5 days.

For the instances MIP solved, the gap between the optimal and the HH-based LowerBound was larger than for our synthetic instances. It averaged 5.0% versus 1.7% for our synthetic instances with $|V| \geq 100$ (10.6% for the solved R200 instances). For none of the 27 real-world instances left unsolved by MIP did we find a solution that matched LowerBound, as opposed to 110 out of 190 for our synthetic instances. This decline in the quality of LowerBound is possibly because there were significantly fewer $\lfloor |F|/2 \rfloor$-good customers than there were for our synthetic instances, presumably because the real OSPF weights generated more shortest path ties. Overall the average was 93.0% versus 97.7% for our synthetic instances, and, for the instances with approximately 500 vertices, 89.9% versus 96.0%. For R1000 it was just 42.2%.

As with our synthetic instances, GREEDY400 found the optimal solution for all instances that MIP solved. It also continued to be robust, never finding its best solution value fewer than 7 times. However, it no longer found the best solution on *all* instances, being beaten by GENETIC on one R500 instance and by HH on the R1000 instance, which was too large for our initial GENETIC implementation to handle. In addition, it averaged 2.97% above our best lower bound, as opposed to 0.76% for our synthetic instances, although, as suggested above, this may be more the fault of LOWERBOUND. And it did beat GENETIC on 9 instances and HH on 12. (GENETIC outclassed HH 12 to 7.)

One can, however, surmise that GREEDY400 may be running out of gas as $|V|$ increases. For the one R500 instance where it was beaten by GENETIC, we can find that better solution with high probability if we replace GREEDY400 by GREEDY10,000. This would increase the running time from 3 to 69 minutes, as compared to 16.5 hours taken by the original GENETIC implementation. A re-implemented GENETIC may be much more competitive, however, since the number of calls it makes to GREEDY will probably be significantly smaller than 10,000. And for R1000, our most realistic real-world instance, GREEDY400 is totally outclassed. HH finds a solution of size 90 in 33 minutes, whereas GREEDY400, which takes roughly the same time, finds only a solution of size 318. Indeed, all GREEDY solutions we have ever found, over more than 10,000 runs, are bigger than 300. (In preliminary experiments, a re-implemented GENETIC algorithm also finds solutions of size 90. This takes 27 hours, a time which should be substantially reduced once we optimize the implementation's parameters, but the resulting running time is unlikely to be competitive with a similarly re-implemented HH, which would likely take less than a minute.)

As to cost reduction, our heuristics provided even greater savings in the numbers of facilities used than we saw for our synthetic instances. For the cases in which all vertices are customers ($C = V$), GREEDY400 produces average savings of 89% over the naive solution of using all customer vertices, versus 75% for the analogous synthetic instances. (The savings is 93% for the HH solution of R1000.) For the cases in which half the vertices were customers, the savings averaged 84% versus 65%. Even in the cases in which only about 1/8 of the vertices were customer vertices, the savings averaged 73% as compared to 35% for our synthetic instances.

Also in contrast to our synthetic instances, here we obtain substantial improvements if we switch to the pathwise-disjoint version of the problem. Whereas the average improvement we saw for such a switch in the synthetic case was about 3.2%, it was 18.0% for our real-world instances, excluding R1000. For R1000, the improvement is 61%, from 90 facilities to 35, obtained using both GREEDY400 and HH. Here settling for arc-disjoint paths only yielded an improvement on one of our 65 instances, and that by just a single facility.

References

[1] J. Bean. Genetics and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6:154–160, 1994.

[2] L. Breslau, C. Chase, N. Duffield, B. Fenner, Y. Mao, and S. Sen. Vmscope – a virtual multicast VPN performance monitor. *ACM SIGCOMM Workshop on Internet Network Management (INM)*, 2006.

[3] L. Breslau, I. Diakonikolas, N. Duffield, Y. Gu, M. Hajiaghayi, D. S. Johnson, H. Karloff, M. G. C. Resende, and S. Sen. Disjoint-path facility location: Theory and practice, 2010. Preliminary draft of journal version, available at `http://www.research.att.com/~dsj/papers/full-monitor.pdf`.

[4] H. Burch and C. Chase. Monitoring link delays with one measurement host. *SIGMETRICS Performance Evaluation Review*, 33(3):10–17, 2005.

[5] L. Buriol, M. Resende, and M. Thorup. Survivable IP network design with OSPF routing. *Networks*, 49(1):51–64, 2007.

[6] K. Calvert, M. Doar, and E. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, 35(6), 1997.

[7] J. F. Gonçalves and M. G. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 2010. Published online 27 August 2010. DOI: 10.1007/s10732-010-9143-1.

[8] Y. Gu, L. Breslau, N. Duffield, and S. Sen. GRE encapsulated multicast probing: A scalable technique for measuring one-way loss. In *IEEE Infocom 2008: The 27th Conf. on Comput. Comm.*, pages 1651–1659. IEEE Communications Society, New York, NY, 2008. A slightly earlier version of this paper is available at `http://www.research.att.com/~duffield/papers/GBDS-vmscope.pdf`.

[9] R. Hassin and D. Segev. The set cover with pairs problem. In *FSTTCS 2005: Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*, pages 164–176, Berlin, 2005. Springer-Verlag.

[10] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, 9:256–278, 1974.

[11] G. Kortsarz. On the hardness of approximating spanners. *Algorithmica*, 30:432–450, 2001.

[12] L. Lovász. On the ratio of optimal integral and fractional covers. *Disc. Math.*, 13:383–s 390, 1975.