Experiments with a network design algorithm using ϵ-approximate linear programs*

Daniel Bienstock

Bellcore and Columbia University
Morristown, NJ 07960 New York, NY 10027

December, 1996 (revised May, 1998)

Abstract

We describe an upper-bound algorithm for multicommodity network design problems that relies on new results for approximately solving certain linear programs, and on the greedy heuristic for set-covering problems.

1 Introduction.

Network design problems are mixed-integer programs that have the following broad structure. Given a graph, and a set of "demands" – positive amounts to be routed between pairs of vertices – capacity must be added to the edges and/or vertices of the graph, in discrete amounts, and at minimum cost, so that a feasible routing is possible.

Problem of this form are increasingly important in telecommunications applications, because of the great expense inherent in maintaining and upgrading metropolitan networks. A wide variety of special cases have been studied. For example, one may be constrained to using a fixed family of paths to carry out the routing, or to using a single path for each demand, or to using integral flows. The precise manner in which capacity can be added can also lead to fairly different problems. Finally, we may not have a graph at all, but a hypergraph, with the hyperedges used both in the routing and capacity installation. It is nearly impossible to provide a comprehensive list of references on network design problems, but those given below (and the references therein) should be adequate in terms of the focus of this paper.

By far most of the work has focused on the special case of the problem where routing is done using fractional flows, and capacity is added to the edges of the graph (see e.g. [2], [3], [7], [8], [26], [27], [28], [30], [35]). We note that variants of this basic problem could potentially be extremely intractable from a combinatorial viewpoint. For example, restricting flows to taking integral values gives rise to disjoint-path problems. But perhaps due to the nature of the practical instances (the demands are probably "large" – it does not make sense to expand the capacity of a network otherwise) even some of these more complex models have effectively behaved like the fractional flow model (see, for example, [11]).

The mixed-integer programs arising from the simplest flow model can be quite challenging. As many researchers have observed,

*This research was partially funded by NSF grants NCR-93-0175 and DDM-90-57665, and was partially carried out while the author was on sabbatical at Bellcore.

- the duality gap between the mixed-integer optimum and the initial LP-relaxation can be very large,
- the polyhedral structure of the problem is complex, and separating the simplest facet-defining inequalities is already difficult, and
- the linear programs to be solved, although relatively small, can become very difficult, especially after cutting planes have been added.

To this list one may add another item, that no effective general-purpose heuristics are known. Altogether, this makes the problems fairly intractable. For example, consider problem 1FC2 studied in [7]. This problem arose from a network with 16 nodes and 98 edges (typical size for a real application). The cutting-plane algorithm in [7] reduced the duality gap from over 250% to just over 7% in a few hours of computing. Starting from the extended formulation, the problem was solved to optimality by J. Eckstein [15] (also see [16]) using a parallel branch-and-bound code that required a wall-clock time of 10 hours running on a CM-5 (64 Sparc10 processors). Recently, S. Ceria and G. Pataki [6] were able to solve this problem, using the extended formulation, in approximately three hours (Sun Ultra 167). To this end they used their code MIPO to generate cuts, followed by a call to CPLEX branch and bound. On the other hand, without the polyhedral cuts they were unable to narrow the duality gap to less than 25%, even after many hours of computing. From this experience we may conclude that indeed the polyhedral cuts we use are extremely useful, in fact essential towards the solution of the problem, even at the expense of making the linear programs more difficult.

The question of why these linear programs are difficult, is, of course, of great interest, but we are unable to say much about this, other than our and our colleagues' ([6], [15], [38]) empirical observation that this is indeed the case, i.e. the LPs can be difficult enough even before adding the cuts, and the addition of the cuts makes them worse. Perhaps this may be due to the density of the cuts, which typically might contain dozens of nonzeroes. It has also been empirically observed [5] that combinatorial problems tend to exhibit large amounts of degeneracy, i.e. optimal extreme points are supported by too many facet-defining inequalities. Finally, there is the multicommodity flow component of the problem, which has long been viewed as a potential source of difficulties. The hardness of these linear programs has spurred research on alternative formulations for the network design problem. See, for example, [37] and [8] for an approach where the problem is projected to the space of the capacity variables – one consequence of this approach is that the *integer* program may well become more difficult, however, because the polyhedral structure becomes more complex.

We are interested in developing a robust, generic algorithm with application to many types of network design problems. Towards this end, we have started work on the simplest version: here we have a directed graph, demands are routed using fractional flows, and capacity is added in integral amounts to the edges of the graph. Note that the demands themselves are directed. A formal mixed-integer formulation is given in Section 2. This problem was studied in [8], with minor variations in [3], and it underlies problems studied in [7], [26], [27], [28] and [30]. In spite of its simplicity, this problem, denoted **CAP** in the sequel, arises in practical applications and can be quite difficult. Our experience with this model and the more complex one in [7] as well as that of other researchers makes it clear that decreasing the LP solution time is an important priority. One way to accomplish this goal would be to solve the linear programs only to a rough approximation: given that we are facing nontrivial duality gaps, working with near-optimal, near-feasible LP solutions should be sufficient to detect violated inequalities. In principle, a cutting-plane algorithm could be run in this manner until we have a have a reasonably tight lower bound. These points can be expanded as follows:

As pointed out by a referee, there is not much theory to explain how to use approximate LP solutions to find high-quality violated inequalities. While this is true in the case of general polyhedra, in our case the nature of the cutting-planes we are dealing with and the very large duality gaps we encounter facilitates the separation of approximate solution.

To put it differently, the class of inequalities that we will (approximately) separate are of the form $ax \geq b$ where a is nonnegative; and furthermore the x variables are unbounded. Hence a cutting-plane algorithm using ϵ-feasible solutions (in terms of **relative** infeasibilities), will, at termination, correctly claim to have solved the optimization problem over the convex hull of these constraints $ax \geq b$ with zero infeasibility and relative optimality error $\frac{1}{1-\epsilon} - 1 \approx \epsilon$ for ϵ small. All that would remain in would be to round our ϵ-optimal fractional solution, this task presumably being facilitated by our improved representation of the underlying integral polyhedron.

To put this in another perspective, let us consider the basic "cutset" inequalities (described in more detail below)

$$x(\delta(U)) \quad \geq \quad \lceil d(U) \rceil \tag{1}$$

where U is a subset of nodes in a graph, $\delta(U)$ is the set of edges with exactly one end in U, and $d(U)$ is the sum of demands with exactly one end in U. In other words, this constraint states that we must install enough capacity in the edges of the cut $\delta(U)$. Consider an example where, say, $d(U) = 2.1$. It is quite likely that the optimal solution to the LP-relaxation, x^*, will in fact satisfy $x^*(\delta(U)) \approx 2.1$. As an empirical observation we can state that many cuts $\delta(U)$ will be found where this situation arises. Thus, even if we only solve the LP-relaxation to a relative accuracy of 1%, we should still be able to detect violation of the inequality $x(\delta(U)) \geq 3$, and after "reoptimizing", even if we can only guarantee a relative infeasibility of 1%, the infeasibility in this inequality will be an order of magnitude smaller than before. Previous experience by other researchers and the author indicates that separating over the cutset inequalities alone improves the value of the LP-relaxation by as much as 30 % – hence, it must indeed be the case that these inequalities are (initially) highly violated.

In recent years, several researchers have developed approximate LP-solving algorithms that would seem to fulfill the role we seek. References [18], [34] discuss Lagrangian relaxation methods for linear programs whose constraints are made up of two components: a set of "linking constraints" and a "subproblem" (or subproblems). Under appropriate assumptions, given $\epsilon > 0$, these algorithms find an ϵ-optimal and ϵ-feasible solution for the combined linear program, while solving a polynomially bounded number of linear programs over the subproblem constraints [We note that we are referring to *relative*, i.e. *scaled*, feasibility and optimality].

Here, the bound is polynomial in the size of the problem and in ϵ^{-2}. Also see [19], [33]. An early form of these algorithms appears in [36], where it was developed with a specific multicommodity flow problem in mind. In fact, multicommodity flow applications are prominently discussed in all the references, and as far as we can tell, all implementations of this methodology have so far been for multicommodity flow problems. This is relevant to us since the network design problem has a multicommodity component, although it is not clear that this is the primary cause for the difficulty of the LPs: they can become significantly more difficult after adding valid inequalities involving the capacity variables. We will discuss the ϵ-approximation algorithms in more detail later.

The main contribution of this paper is an implementation of a cutting-plane algorithm for problem **CAP**, and an upper bound heuristic for it, both of which rely on an implementation of the ϵ-approximation methodology specialized for LP-relaxations of CAP. To a very rough degree, the cutting component of this algorithm can be viewed as a version of the algorithm in [8] with all calls to a simplex solver replaced with calls to an ϵ-approximation algorithm – however, because of the very different behavior of the two LP-solvers, there are some significant differences. In addition, the primary focus is on obtaining good estimates of problem values.

The initial experience using our algorithm on real-life problems appears good. We quickly obtain reasonably tight upper bounds for the problems in [8]. In this regard, our approximation algorithm is clearly superior to (our) previous "exact" implementation, and it does not appear easy to convert this exact implementation into a comparably fast upper-bound heuristic. We simply do not know of another method that can reliably perform with comparable performance. In the case of a new, large (200 node) problem that cannot be approached with a simplex- (or barrier-) based algorithm, we again obtain a fairly good bound.

A byproduct of using ϵ-approximation is that our "solutions" will have small infeasibilities. On the theoretical side, we feel that as long as this is treated in a mathematically precise manner, we are on safe ground. On the practical side, network design is frequently used in *volume studies*, where many different potential demand patterns are analyzed. Here it is important to obtain good estimates quickly, and as long as the infeasibilities are small there should not be a difficulty, since the exact demand values are not known. The merits of this paper rest on these assumptions.

There is a deeper side to this issue that is worth pointing out. Consider the performance of a commercial branch-and-bound code. Such a code will use various tolerance parameters, typically small numbers: a feasibility tolerance, an optimality tolerance and an integrality tolerance. However, even if these tolerances are small, such a code may fail to find the optimal solution, even while claiming otherwise. The reason for this is that a node may be fathomed as "infeasible" even though this may not "really" be the case, and because we are dealing with an integer program, the "true" value of the problem could well turn out to be *quite* different from the reported value at termination. As far as we know, commercial branch-and-bound codes do not include any provisions that guarantee that this problem will not occur. The only way we can see to avoid it would be to use exact arithmetic at various critical points of the algorithm. We are only aware of such a strategy in the work of [1], but not in commercial codes. Yet such codes are widely used. In fact, typically the tolerances can be relaxed, precisely so that "approximate" solutions can be quickly found.

What is more, a critic of our approach could point out that the value of the optimization problem, subject to ϵ-"relaxed" constraints, *might* be smaller than the value of the original problem. The answer to this, is that this piece of information is irrelevant, unless the relaxed problem can be quickly solved, which is highly unlikely. Our algorithm produces a solution whose feasibility error, with regards to the *actual* problem we are interested in, is very small in practical terms, while at the same time having cost not too far from the optimum, and, further, it produces such a solution much faster than by any other method we know. Finally, although from a purist point of view our algorithm is a heuristic, its crucial components rest on solidly grounded mathematical constructs.

However, our experience is not perfect and it points out shortcomings of the ϵ-approximate methodology. Based on the results in this paper and in [22], it appears that a hybrid algorithm combining ϵ-approximation and the simplex method may be the most effective solution, in particular in the context of a cutting-plane algorithm using polyhedral cuts. In general, when dealing with either multicommodity flow problems or LPs of the type we face in network design

- For problems of small to medium size (networks with up to 50 nodes, say), a properly formulated linear program will only have a few tens of thousands of variables, and the simplex method using primal steepest edge pivoting is clearly superior as a pure LP-solver over the ϵ-approximation techniques. This is true even when ϵ is relatively large, while on the other hand we use simplex to solve the LP to standard precision. When dealing with very large problems (also see [19]) the ϵ-approximation techniques indeed appear useful.
- Further theoretical and implementation work is needed to make these techniques truly robust and fast.
- In particular, interleaving the Lagrangian iterations with periodic simplex-like steps to "clean-up" the current iterate may well be beneficial, as well as a final crossover-type step to a pure-simplex method. This last stage may well be critical in terms of effective separation of violated inequalities when using finer tolerances (e.g. $\epsilon = 10^{-5}$). We have not implemented this crossover step, and the "clean-up" step only in embryonic form.

2 The simple network design problem.

We briefly return to problem CAP described in Section 1. A mixed-integer formulation is as follows:

$$\min \quad \sum_{ij} c_{ij} x_{ij}$$

$$\textbf{(CAP)} \qquad \text{s.t.}$$

$$\sum_{(j,i)} f_{k,ji} - \sum_{(i,j)} f_{k,ij} \quad = d_i^k \quad \forall k, i \tag{2}$$

$$\sum_k f_{k,ij} - x_{ij} \quad \leq 0 \quad \forall (i,j) \tag{3}$$

$$x_{ij} \in Z^+ \ \forall (i,j), \ \ f_{k,ij} \quad \geq 0 \quad \forall k, (i,j) \tag{4}$$

Here $x_{ij} \in Z^+$ is the capacity added to edge (i,j), and $f_{k,ij}$ is the flow of commodity k in edge (i,j). Thus (2) are flow-conservation constraints, where d_i^k is the demand for commodity k at node i (more details on this below), and (3) is a capacity constraint. In this and in more complex flow models of network design, there are several alternative versions of "commodities." It appears that the most common in the literature is that where each demand corresponds to a separate commodity. It is also possible to aggregate demands by source (or destination). The resulting formulation is much more compact and results in substantially easier linear programs. Moreover, as discussed below, the cutting planes that are used in computations are valid for both versions. Consequently, in our opinion, the source-aggregated formulation is superior for CAP, and we will use it in this paper.

Research by many authors has shown that the polyhedral structure of this problem can be very complex. Most of the combinatorial inequalities concern multi-cuts, or partitions of the vertices (see the above references). These inequalities can be considered generalizations of the partition inequalities for the Steiner tree problem. The simplest inequality concerns a simple cut in the graph. Given a set of vertices U, let $V-U$ denote its complement, and let $d(U, V-U)$ be the sum of demands going from U to $V-U$. The *cutset inequality* then states:

$$\sum_{i \in U, j \in V-U} x_{ij} \quad \geq \quad \lceil d(U, V-U) \rceil \tag{5}$$

Possibly [26] is the earliest reference for these inequalities. For the particular problem at hand, the cutset inequality is facet-defining when $d(U, V-U)$ is fractional and both U and $V-U$ induce strongly connected subgraphs. Separating cutset inequalities is NP-hard. More precisely: given a pair x, f satisfying (2), (3) and (4) it is NP-complete to decide whether all cutset inequalities are satisfied (unpublished result). Barahona [3] has developed a max-cut based algorithm for separating these inequalities (or rather, the version of these inequalities arising in his model) and otherwise various heuristics have been proposed by other authors.

Another class of inequalities that has been widely used is that of three-partition inequalities. Let U^i, i= $1,2,3$ be a partition of the vertices into three classes, and let $d(U^1, U^2, U^3)$ denote the sum of all demands with source in U^1 and destination outside U^1 and all demands from U^2 to U^3. Then

$$\sum_{i \in U^1, j \in U^2} x_{ij} + \sum_{i \in U^1, j \in U^3} x_{ij} + \sum_{i \in U^2, j \in U^3} x_{ij} \quad \geq \quad \lceil d(U^1, U^2, U^3) \rceil \tag{6}$$

is valid and facet-defining under appropriate conditions. Finally MIR-type inequalities that include terms $f_{k,ij}$ have also been studied (Günlük [21] has reported some positive experience with inequalities of this type).

Reference [8] describes two cutting-plane algorithms for CAP with good computational experience. In particular, one that uses the formulation given above, employed two heuristics to separate cutset inequalities, and used these as an engine for separation of three-partition inequalities.

The first separation heuristic is enumeration-based: in the case of typical-size networks (roughly with 30 nodes) as in [8] this heuristic initially enumerates all facet-defining cutset inequalities and stores them in an appropriate data structure. This data structure is periodically scanned by the algorithm to detect violations.

The other heuristic is based on the idea that edges that are "tight" for the variable upper-bound inequality (3) are likely to be in cuts for which the cutset inequality is also tight. Thus the heuristic assigns, to any edge (i,j) a positive length w_{ij} depending on its congestion. Next the heuristic computes the set of nodes within distance $\delta > 0$ of any given node, for appropriately chosen δ. These δ-neighborhoods are used for testing the cutset inequalities.

The resulting algorithm in [8], together with branch-and-bound, was successful at solving optimally or near-optimally several families of real-life problems, in approximately one hour on a Sparc 10/51.

We briefly mention the competing algorithm studied in [8]. This algorithm operates by working with the x variables only. Since the projection of the *continuous* relaxation of CAP to x-space is already complex enough (it requires exponentially many inequalities) this algorithm always uses a larger, simpler polyhedron, with cutting planes being used both to enforce integrality and to achieve feasibility. This algorithm has the advantage of facing (presumably) easier LPs, but it has the disadvantage of most likely facing a far more complex convex hull of integer solutions than that of formulation CAP. In the testing provided in [8] the multicommodity approach was typically five times faster than the projection approach. We conservatively interpret this result to mean that the multicommodity approach is at least competitive with the projection approach.

3 Approximate methods for linear programs.

In recent years a great deal of research has been focused on the approximate solution of linear programs with a special structure. We will describe the results in condensed form here. Suppose we are interested in finding a feasible solution to a system of the form,

$$AX \leq b \tag{7}$$
$$X \in P \tag{8}$$

where A has nonnegative entries and b has positive entries, and (to fix ideas) P is a polyhedron over which it is "easy" to solve linear programs.

We will denote the rows of A by $a_1, a_2, \cdots$. For $\epsilon > 0$ the algorithms in [18], [19], [34] find, if it exists, a vector $X \in P$ such that $AX \leq (1+\epsilon)b$ by solving a number of linear programs over P bounded by a polynomial on the number of rows and columns of A and ϵ^{-1}. Reference [18] also assumes that there is a point $x \in P$ that is "sufficiently interior" to the constraints (7), while on the other hand [34] includes another factor in the complexity bound, the *width* of P w.r.t. (7), defined by $\max_{X \in P} \max_i \{a_i X / b_i\}$. Here we are dealing with a *packing* problem; when (7) is replaced by $AX \geq b$ (a *covering* problem), and also under more general assumptions, a similar methodology can be used. In general, the constraints corresponding to A are called the "linking constraints."

Briefly, the algorithms iterate through a sequence X^1, $X^2, \cdots$ of points in P. As a surrogate for feasibility, the algorithms use a *potential function* of the form

$$\Phi(X) = \sum_i e^{\alpha(\frac{a_i X}{b_i} - 1)} \tag{9}$$

where $\alpha > 0$ and the sum is over the rows of A.

Let X^k be an iterate, and let δ^* be an optimal solution to the optimization problem

$$\textbf{(DIR)} \qquad \begin{aligned} &\min \quad [\nabla\Phi(X^k)]^t \delta \\ &\text{s.t.} \\ &\delta \in P - X^k \end{aligned} \tag{10}$$

where $P - X^k$ is the set P translated by X^k. Then $X^{k+1} = X^k + \lambda\delta^*$, i.e. δ^* is used as a step direction and $\lambda > 0$ is the stepsize, chosen either through a line-minimization (to minimize potential) or using a prescribed value. With α appropriately chosen (in particular, of the order of ϵ^{-1}) it can be shown that the potential function decreases fast enough to guarantee polynomial-time convergence to an ϵ-feasible point. As far as we can tell, the best complexity bound on the number of iterations is $O(\epsilon^{-2})$ (and low-degree polynomial on the other parameters), attained by the algorithm in [19].

Note that

$$[\nabla\Phi(X^k)]^t = v^t A \quad \text{where} \tag{11}$$

$$v_i = \frac{\alpha}{b_i} e^{\alpha(\frac{a_i X^k}{b_i} - 1)}, \tag{12}$$

and consequently we can view the overall algorithm as a Lagrangian relaxation procedure, with multipliers v being used in iteration k. Further, in the sense that the algorithm proceeds by trying to decrease the potential function, it behaves as a steepest descent method. Finally, the -1 term in (9) has no effect, but we prefer to state the potential function this way to stress that it penalizes relative violations.

The basic framework just described can be used for optimization problems. One way to do this is to include the objective function as another constraint in the system $AX \leq b$, using a target value for its right-hand side, and then iterate (using binary search) through several target values. Reference [19] discusses a method of bypassing this binary search. Also see [33].

The standard application of this methodology has been to pure (continuous) multicommodity flow problems. Here the role of the packing constraints (7) is played by the joint capacity constraints

$$\sum_k f_{k,ij} \leq u_{ij} \tag{13}$$

where $f_{k,ij}$ is the flow of commodity k in edge (i,j), and u_{ij} is the capacity of this edge, and the polyhedron P is described by the flow conservation equations. Thus the direction-finding linear program breaks up into a separate problem for each commodity. Depending on the precise variant of multicommodity flow, these problems are either min cost flow or shortest path problems. In fact, the earliest version of the exponential penalty function method is described in [36], where it was specifically developed for the *max concurrent flow problem*, a special case of the multicommodity flow problem where the commodities must be routed so as to minimize the maximum violation of the joint capacity constraints (13).

3.1 Computational experience.

As far as we can tell, to date all computational experiments using the ϵ-approximate methodology have been with multicommodity flow problems. Here we will briefly comment on the results in [24], [19], [22] and [17] (also see [10]). The results may be summarized as follows: the methodology seems useful when applied to very large problems; otherwise the simplex method appears superior as a stand-alone solver. For medium-size and large problems the methodology is useful as an elaborate "crash" heuristic.

In [24] the authors describe possibly the first implementation of the ϵ-approximation algorithm to the maximum concurrent flow problem (route to minimize the maximum violation). The results in this paper showed that the approximation technique had some promise, and that a high degree of sophistication would be needed to get very good performance.

In [19] the authors describe a sophisticated implementation of the ϵ-approximation algorithm to some special multicommodity flow problems created by a problem generator. The implementation is solid and relies on mature codes to carry out several of the tasks (such as subproblem optimization and line-search). On some quite large instances (the largest, **PDS80**, has some 426,000 columns) the algorithm soundly defeated OSL. In fact, on the largest problems OSL could not really be used, and although the ϵ-approximation algorithm did not in fact terminate, it obtained an nearly feasible solution (thus providing an upper bound on the value). Problem PDS80 required approximately 1 hour 43 minutes on an RS6000/550.

It is clear that on these problems the methodology is obtaining useful results not directly possible using standard LP codes. However, can we somehow make simplex competitive? To handle these problems, the algorithm in [22] proceeded as follows. First, the ϵ-approximation stage was run a very small number of iterations with ϵ greater than 0.01. The algorithm then switched to a simplex stage (using CPLEX 3.0), which required several iterations of column pricing and re-optimization. The resulting code essentially matched the performance of that in [19]. For example, on PDS80 the code took approximately 1 hour 58 minutes on a Sparc 10/40 to obtain a slightly better solution to that found in [18], and a similar behavior was observed with the other problem instances. In fact, it turned out that the simplex calls in [22] were running out of memory; thus a small increase in memory would have considerably sped up the code.

Finally, we mention the implementation in [17]. The code presented here approximately solves general multicommodity flow problems. On randomly generated problems, even extremely large ones, this code performs impressively well.

In summary, it appears that the ϵ-approximation methodology can be useful, especially when applied to very large problems, and, in our opinion, when used as a starting step for a standard linear programming code. We stress that no such "crossover" is used in the algorithm reported in this paper.

There are several areas where in our experience the methodology can be improved.

- The very good experience in [18] with multicommodity flows notwithstanding, can the algorithms reliably and quickly handle values of ϵ much smaller than 0.01, especially when dealing with more general linear programs? In terms of a practical application, a final output with accuracy 0.01 would be sufficient, but to be truly useful in a cutting-plane algorithm that uses polyhedral cuts, the methodology should be able to handle $\epsilon = 10^{-4}$.
- The methodology tends to produce fairly interior points. For example, in multicommodity flow problems it tends to produce flow-carrying cycles. This probably makes the algorithm run slower and is also undesirable in the cutting-plane setting. In the multicommodity flow case it is easy to correct this problem (and our implementation does) but it is not clear how to do so in general, while avoiding linear algebra.
- As noted before, the algorithms essentially prescribe steepest-descent steps steps. In the traditional literature, methods of this type have had poor theoretical and practical behavior. This is a fundamental difficulty. One can easily construct multicommodity flow examples where the Hessian matrix (see [25]) of the potential function is arbitrarily bad, potentially leading to *zigzagging*. Another potential bad effect is that of *jamming*: here a sequence of very short steps converges to a point that is not a minimizer. Note that the ϵ-approximation algorithms may take very short steps. For example, in [33] the steps could be as short as ϵ^{-2}. This could lead to numerical difficulties. When the specified step size is very small, it may be unwise to use it, and the algorithm has effectively jammed.

4 An implementation of the ϵ-approximate LP solver as specialized to the network design problem

Here we describe our implementation of the ϵ-approximation algorithm, as specialized to the (LP-relaxation of) formulation CAP plus some number of cutting-planes.

The cutting planes we use are all cutset inequalities or three-partition inequalities. Consequently, the linear programs to be handled have three types of constraints: the flow conservation constraints for each commodity (2), the variable upper-bound inequalities (3) and the facet-defining inequalities

$$Mx \quad \geq \quad r, \tag{14}$$

where M is a $\{0,1\}$-matrix and r is an integral vector, as well as the variable bounds.

Before giving a precise description of our implementation, it is worth pointing out some salient facts.

If we restrict the linear program to the x variables we have a covering-type problem (with linking constraints (14) and trivial subproblem) and if we restrict it to the f variables we have a packing-type problem (with linking constraints (3)); this is a feasibility system for a multicommodity flow problem where the direction-finding subproblem consists of a shortest-path problem for each demand. Based on this, one could presumably devise an algorithm that alternatively approximately solves covering and packing problems, or perhaps alternates between small numbers of iterations for either system. We did not implement this approach.

In our implementation we simultaneously use the x and f variables. Here the linking constraints are (14) and (3). All other constraints (i.e. the flow conservation constraints and the variable bounds) are viewed as constituting a single subproblem.

More precisely, the algorithm employs a parameter $\theta > 0$. All linking constraints with relative infeasibility greater than $-\theta$ are considered *active*. The remainder are simply called *feasible* or *inactive*. In our implementation we used $\theta = 10^{-3}$. **Only** the active constraints are used to construct the objective for the direction-finding subproblem. Sometimes, as a byproduct, a step is restricted to the x or the f variables (i.e. the direction-finding problem outputs a direction step that is zero along some of the coordinates) and in particular to some of the commodities only. The pure-routing part of our LPs did not seem to be the computational bottleneck, and we did not employ more sophisticated commodity-choice rules. Also recall that the generic ϵ-approximation approach, as described in the previous section, either approximately solves a system of linear inequalities, or proves its infeasibility. An infeasibility proof that considers only a subset of the **constraints** is a valid infeasibility proof in any case, but may be much faster to obtain.

The variable upper-bound inequalities (3) present a difficulty because of the zero right-hand side. Ideally, we would like to include these inequalities directly in the potential function (9). Doing so would probably not work because it will tend to overpenalize their violation. Inequalities (3) must be *scaled* so that in a sense we are dealing with relative infeasibilities. The default choice of scales that we use is the following: at iteration t inequality (3) corresponding to edge (i, j) is scaled down by the value $\max(1, D, x_{ij}^t)$, where D is the sum of all demands, and x_{ij}^t is the current value of x_{ij}. However, we will also consider LPs where the x variables are fixed (i.e., pure routing problems) and here the scale will be $\max(1, x_{ij})$.

Finally, as stated before, in our formulation for CAP all commodities are aggregated by source. However, the direction-finding subproblem decomposes into a separate shortest-path problem for each demand.

4.0.1 Major structure of the approximate LP-solver

As is standard, the algorithm has a feasibility stage and an optimality stage. As in [34] (also see [19]), both stages are divided into improvement phases. During improvement phase t the algorithm is trying to achieve accuracy ϵ_t. The first phase uses $\epsilon_t = 0.1$ and each successive phase reduces ϵ_t by half until we reach the desired value of ϵ. Each phase performs at most a fixed number of iterations of the ϵ-approximation method. During phase t the parameter α in the potential function (9) is of the form γ/ϵ_t. Ideally the constant γ should depend (weakly) on the number of active constraints; after some testing we settled on $\gamma = 0.2$. Please refer to the Appendix for pseudocode for a typical improvement phase.

The feasibility stage. In this stage the algorithm seeks to achieve ϵ-feasibility. In order to find a low-cost solution as well we modified the objective function to the direction-finding subproblem DIR. Recall definition (9) of the potential function, where the $X^h = (x^h, f^h)$ is the solution at iteration h. The objective function of DIR is modified from $[\nabla\Phi(X^h)]^t\delta$ to

$$[c + \sigma\nabla\Phi(X^h)]^t\delta, \tag{15}$$

where c is the original objective function to the network design problem CAP (nonzero only for x variables), and

$$\sigma \;=\; \frac{c^t x^h}{\sum_i b_i e^{\alpha(\frac{a_i^t X^h}{b_i} - 1)}}, \tag{16}$$

the sum being taken over **all** linking constraints i that are currently active (recall that this means relative violation at least -10^{-3}). Thus the objective is trading off cost vs. penalized right-hand side. This is similar to the subproblem in [34] except for the choice of σ. Finally, during the feasibility stage we use $\epsilon = 10^{-4}$ and each improvement phase (fixed ϵ_t) is performed for at most 1000 iterations. In terms of the pseudocode in the Appendix, this is a call to LOOP(10^{-4}, 1000).

The optimality stage. In this stage the algorithm iterates through several objective target values, by adding a packing-type budget constraint $\sum_{ij} c_{ij}x_{ij} \le T$ (as in [34]) and searching for an ϵ-feasible solution to the overall system. The algorithm iterates through values of T by running a biased version of bisection search. If $[T^1, T^2]$ is the current range of target values, the next value of T is $0.7T^1 + 0.3T^2$. Ideally, this approach should mostly produce infeasible problems (values of T that are too low), and the infeasibility of each problem should be detected early, until we get to a value of T close to the optimum value. In other words, ideally this approach would converge to optimality at the same time as it converges to feasibility. Note: [19] describes a method for avoiding the explicit bisection search. Also see [20].

In addition, the cost coefficients c_{ij} are scaled at the start so that the maximum cost has value 1. In practical instances of CAP (and certainly in the harder ones) it is usually the case that the optimal values of the x_{ij} are small integers (say, ≤ 5). When the costs are scaled we can therefore expect to be dealing with targets T that are essentially bounded, or more precisely, at worst of the order of the number of edges in the graph. In the direction-finding subproblem we also use the objective as in (15), except that the budget constraint also contributes to the Lagrangian $\nabla\Phi(X^h)$.

Finally, during the optimization stage we use, for each objective target, $\epsilon = 8 \times 10^{-3}$, as a pessimistic surrogate for our real objective, which is to achieve $\epsilon = 0.01$. We also run each improvement phase for at most 200 iterations. Thus, this is a call to LOOP(8×10^{-3}, 200);

There are two key items that are needed in a successful implementation: the choice of a stepsize, and the handling of zero-stepsize situations.

Choosing the stepsize. After solving the direction-finding subproblem DIR, the stepsize λ is chosen as follows.

The standard approach (see e.g. [19]) would be to choose λ to minimize potential. However, given the combinatorial nature of the problem and of the constraints (14) we have experimented with choosing λ to minimize the maximum infeasibility, whenever possible.

Thus, the stepsize is determined by solving a problem of the form,

$$\min\nolimits_{t \ge 0} f(t), \tag{17}$$

where

$$f(t) \;=\; \max_i \{a_i t + b_i\}, \tag{18}$$

for appropriate values a_i, b_i, $1 \le i \le m$. Hence we have to solve a linear program on two variables. $O(m)$-time algorithms for this problem are known ([14], [29]), however they are somewhat complex. Instead, we employed bisection search to solve (17), to absolute accuracy of 10^{-7}.

Handling jams. A *jam* occurs when the stepsize is too small (in our implementation, smaller than 10^{-7}) or when the step direction increases the maximum violation. Jams are handled by attempting the following fixups, in that order.

1. If the jam occurs during a feasibility stage, then the step direction is recomputed by solving DIR without the c term in (15); i.e. by solving the standard version of DIR (c.f. (15)). In this case the jam is broken.
2. Otherwise, we try one of several heuristics to modify the step direction and break the jam. The main such heuristic consists of temporarily defining the set of active inequalities to be those whose (relative) violation is at least 0.995 times the maximum violation, and rerun DIR.
3. If the jam persists we *abort*.

Here "aborting" means that we terminate the current improvement phase of the algorithm. Frequently, simply moving to the next value ϵ_{t+1} will break the jam. If the jam occurs during the last phase, then the current system is declared ϵ-infeasible. If this happens during the last iteration of the bisection search for the objective target, then the overall algorithm stops.

Notice that by default the algorithm does not try to prove ϵ-approximation of the solution it produces, and potentially it may fail to achieve this, but in our experience it worked well. The main focus in the implementation was to make it work well with the rest of the network design algorithm, in particular, the ability to quickly take small numbers of steps, restricted to a subset of the variables and constraints. Table 1 shows computational results with LP-relaxations of problems from [8] that we will consider later. These linear programs have roughly 1500-1900 variables and 400-600 constraints (In terms of the source-aggregated LP-formulation. This has no bearing on the ϵ-approximation algorithm, other than on the data structures we use). Note: recall that the runs reported here used $\epsilon = 8 \times 10^{-3}$.

Problem	Exact value	Approx. value	error (%)
MS1	2753.4	2767	0.49
MS2	2791.5	2804	0.44
MS3	2829.5	2851	0.76
MS4	3067.9	3074	0.20
MS5	2997.6	2977	-0.69
MS6	2391.2	2399	0.33

Table 1: Exact vs. approximate LP solutions

The table shows that the desired accuracy was achieved. However, the same problems were solved to optimality by CPLEX in about one-third of the time. This should not be surprising, in view of the results in Section 3 concerning small problems. Later we will see results with a much larger problem where the approximate method is far superior to simplex.

For completeness, in the Appendix we include pseudo-code to describe the key component of our implementation: the loop, where, for a given objective target value T, we either try to obtain an ϵ-feasible solution to the system consisting of the budget constraint $\sum_{ij} c_{ij} x_{ij} \leq T$, any facet-defining inequalities that are currently in the formulation, the variable upper-bound inequalities, and the flow conservation equations, or else prove that this system is infeasible, as well as each improvement phase (i.e. fixed ϵ_t) as described above.

5 An algorithm for CAP based on ϵ-approximation

Next we will describe a new algorithm for CAP. This algorithm has

1. an initial approximate LP-solving step,
2. a cutting-plane step, and
3. a rounding-branching-cleanup step to obtain upper bounds.

All these phases employ the heuristic implementation of the ϵ-approximation algorithm given in the previous section to handle the linear programs. For brevity, we will refer to our overall algorithm as **APPROX**.

As noted in section 3.1 it is likely that the simplex method will outperform the ϵ-approximation method for linear programs arising from networks of typical size. However, this is not necessarily the case when re-optimizing a problem, for example after adding cuts. Morevoer, the lightweight nature of the ϵ-approximation algorithm makes it easy to use heuristically and in restricted form, for example to run a small number of iterations or restricted to a submatrix of the constraints. Finally, we are interested in handling much larger problems. As noted before, potentially a hybrid LP-solver would be best. In this paper we are primarily interested in stress-testing the ϵ-approximation approach.

5.1 The initial step.

The first step in the algorithm is to magnify all demands by a small factor. After some testing, we chose a factor of 1.02. The rationale behind this is as follows. Because of the way that we handle the linear programs, we expect that inequalities (3) will have small violations –ideally, 1% to 2%. [At a certain point in our upper bound heuristic, we will restore all demands to their true values. This will have the effect of making the violations of (3) even smaller, and also possibly allowing us to decrease some of the x_{ij}.]

We found this magnification step to be very important, but in principle it can substantially increase the value of the mixed-integer program.

After magnifying the demands, we add to the formulation all degree inequalities (cutset inequalities corresponding to singletons) and "solve" the LP relaxation as detailed above: the optimality stage uses $\epsilon = 8 \times 10^{-3}$ and the feasibility stage uses $\epsilon = 10^{-4}$. Our objective is to try to achieve overall accuracy of the order of 0.01.

5.2 The cutting-plane step.

This consists of running at most $five$ iterations of cutting and resolving the linear program. An inequality was added to the formulation if its violation was at least 5%. The reason for these choices is simple: our implementation of ϵ-approximation algorithm cannot adequately handle weakly violated inequalities. Moreover, we found that after just a few rounds of cutting, the new violations we could detect were very small (typically smaller than 1%). Instead, our algorithm handles inequalities with small violations during the upper-bound stage.

As in [8] we have applied two procedures for separating facet-defining cutset inequalities. The first simply enumerates all such inequalities at the start and maintains them in a list thereafter; despite the obvious lack of appeal of such a procedure for problems of typical size this actually runs quite fast (roughly 2 minutes). The second procedure is based on the observation that a cut consisting of edges that are nearly tight for inequality (3) is likely to give rise to a violated cutset inequality. We will define this procedure in more detail later, but it basically assigns lengths to edges based on the tightness of (3) and evaluates the cutset inequality on sets of nodes within a given distance δ of each source.

Having detected violation of a cutset inequality corresponding to a set U, we also test the cutset inequality for all sets obtained by adding to U and removing from U a singleton. Finally, we also test all three-partition inequalities corresponding to U, a singleton, and the remaining nodes.

In our experiments, using either procedure yielded large numbers of violated inequalities. However many of these were weakly violated and were discarded. On the problems corresponding to Table 1 (27 nodes) the enumeration procedure was somewhat more successful, at least in terms of the final output of APPROX. Typically the cutting step finished with some 100-200 cuts in the formulation.

5.3 The upper-bound stage.

This stage consists of four steps,

1. Refinement of fractional solution,
2. Perturbation (rounding) of fractional solution,
3. A further round of cutting, and
4. Branching and clean-up.

The final output is a vector satisfying the properties: (a) all x_{ij} are integral, (b) no flow is routed on edges (i, j) with $x_{ij} = 0$, (c) all demands are routed, but (d) some variable upper-bound inequalities (3) may have small violations.

Refinement step. In preparation for rounding and branching, we first refine the current solution vector to obtain a more accurate solution. Recall that the ϵ-appromation algorithm is run with $\epsilon = 8 \times 10^{-3}$ during the cutting-plane stage. Now we run it once, with $\epsilon = 10^{-3}$. This accuracy may not be achieved, of course, but the intention is to "push" variables closer to bounds (for example, to make small but positive values even smaller).

We also identify and eliminate flow-carrying cycles, as discussed before.

Rounding and cutting steps. Now let x^* be the fractional capacity vector we have after the more accurate optimization. Clearly $\lceil x^* \rceil$ would be feasible with the current flows, but it could be very expensive. Instead we should be able to round up some components of x^* and round down the rest, and this could be done using standard branch-and-bound. If we use ϵ-approximation this step could become costly because the linear programs probably ought to be solved to high accuracy. For example, when branching down on a variable x_{ij}, its corresponding capacity constraint might become highly violated. The nature of the ϵ-approximation algorithm is such that it tries to "even out" or "spread around" infeasibilities, potentially resulting in a slow run.

Instead, we proceed as follows. Let the capacity vector y be defined by the following procedure:

1. Suppose $x^*_{ij} < 1$. If $x^*_{ij} < 0.01$ then $y_{ij} = 0$. If $x^*ij > 0.6$ then $y_{ij} = 1.0$. Otherwise $y_{ij} = x^*_{ij}$.
2. Suppose ose $x^*_{ij} \geq 1$. If $\lceil x^*_{ij} \rceil - x^*_{ij} < 0.04$ set $y_{ij} = \lceil x^*_{ij} \rceil$, otherwise $y_{ij} = \lfloor x^*_{ij} \rfloor$.

Typically, y will be highly infeasible (together with the current flow) and superoptimal, as well as fractional. One may think of y as simply being obtained by rounding down x^*. We wish to perturb y to an integral, nearly feasible vector of cost similar to the cost of y, while avoiding the solution of linear programs.

To do this, we first run one iteration of the cut separation procedure, as described above, applied to y. This will result in a fairly large number of cuts that can be added to the formulation. Denote the resulting system by

$$Ax \geq b \tag{19}$$

At this point we –temporarily– view system (19) as a surrogate for determining feasibility of a capacity vector. This is related to the approach used in [37] and in one of the algorithms in [8]. Indeed, the cutset and three-partition inequalities without rounding of the right-hand-side are special cases of the *metric* inequalities that define the projection of (fractional) CAP to the space of the x variables [32]. In addition, recall that at the start of APPROX we magnified the demands by 2%. Clearly we have that $\lceil x^*_{ij} \rceil$ is a feasible solution to (19). Ideally, we would next look for a low-cost integral vector w which is feasible for (19) and also satisfies

$$\lfloor x^* \rfloor \leq w \leq \lceil x^* \rceil. \tag{20}$$

Instead, we relax this somewhat. For each edge (i, j) let n_{ij} be the smallest integer larger than y_{ij}. We then search for a low-cost vector z with

$$z_{ij} = 0 \quad \text{or} \quad z_{ij} = n_{ij} - y_{ij} \tag{21}$$

for all (i, j), such that $y + z$ is a feasible solution to (19). Notice that since A is a $\{0, 1\}$-matrix, searching for z is similar to solving a set-covering problem.

Branching and clean-up. To find z we could use one of the classical greedy heuristics for $0, 1$-set covering (see [13], also see [12] for a different approach). These heuristics start with all variables set to zero; and then iterate by choosing the variable that minimizes the ratio of cost to the total number of new rows covered by setting that variable to 1. This variable is set to 1 and the process repeats until all rows are covered.

Such heuristics can have poor practical behavior and instead, we use a greedy-like heuristic (described next) as the basis for branching: the variable picked by the heuristic is instead used to branch.

When branching down on a variable x_{ij}, the algorithm sets $z_{ij} = 0$. When branching up on x_{ij}, it chooses $z_{ij} = n_{ij} - y_{ij}$. Branching is fathomed when a node is feasible, or guaranteed to be suboptimal, or all variables have already been branched on (and the node is still infeasible).

The branching variable we use is chosen by the following rule. For simplicity we describe how the rule works at the root node. We choose an edge (i, j) such that the ratio

$$\frac{c_{ij}(n_{ij} - y_{ij})}{F_{ij} + (1/20)F_{ij}^2} \tag{22}$$

is minimized, where n_{ij} is as before and F_{ij} is the total decrease in infeasibility in (19) obtained by increasing x_{ij} from y_{ij} to n_{ij}. We have found that having a quadratic term in the denominator above helps, in a way this crudely approximates the potential function.

When the branching algorithm does find a vector z such that $x = y + z$ is feasible for (19), it may still be the case that x, together with the flows we had at the start of branching, violates some of the capacity (i.e. variable upper-bound) inequalities (3). Also, x may have some fractional components. To handle this situation, the algorithm performs a heuristic "clean-up." The core procedure is the following.

The first step is to remove the 2% magnification of the demands imposed at the start of APPROX. Then we decrease to the nearest smaller integer any x_{ij} if this does not violate the corresponding inequality (3). Next, say that an edge (i, j) is *bad* if its inequality (3) has relative violation greater than 0.02. The bad edges are sorted in decreasing order of the total amount of flow they carry. We enumerate every edge (i, j), in this order, and if the violation of its inequality (3) is greater than 0.02, we increase x_{ij} by one unit. We then call the ϵ-approximate LP-solver as described in Section 4 running 1000 iterations with $\epsilon = 10^{-3}$-approximation algorithm with the x variables fixed at their current values. In terms of the Appendix, this is essentially a call to IMPROVEPHASE(10^{-3}, 1000), that in addition does not allow the x variables to change. Finally, we round up all fractional components of x.

At this point, if the average violation exceeds 0.035, the solution is *rejected*. Otherwise, it is accepted and it may provide a new upper bound.

5.4 Computational experiments

We will focus on the more demanding problems studied in [8], and on a much larger problem. The harder problems from [8] are interesting because they are difficult but not very large. Potentially APPROX is at a disadvantage relative to a simplex-based method because of the relatively small size of these problems. The network and cost data for these problems is real, and some of the demands are somewhat artificial. Results on the other problems from [8] are summarized at the end of this section. On all of these problems we used the basic cutset inequalities and the three-partition inequalities.

For convenience, we include below results from [8]. These problems are all on the same Norwegian network, with 27 nodes, 102 edges, 67 demands and varying numbers of sources. Thus, these problems have a 102 general integer variables and roughly 1900 variables in total, and some 500 constraints. In what follows, we will refer to the algorithm in [8] as the *exact* algorithm, even though it is not.

Prob.	lower	cut lower	best lower	upper	GAP (%)	Time (sec)
MS1	2753.4	2925.7	2933.2	2976.3	1.45	3600
MS2	2791.5	2943.1	2955.3	2978.2	0.77	3600
MS3	2829.5	2969.4	2978.9	2978.9	0.00	3600
MS4	3067.9	3225.9	3232.3	3256.8	0.75	3600
MS5	2997.6	3169.6	3174.8	3246.6	2.21	3600
MS6	2391.2	2563.4	2566.9	2592.4	0.98	3600

Table 2: Performance of exact algorithm

Here the time is in seconds, on a Sparc 10/51, using CPLEX 3.0. The columns headed "best lower" and "upper" indicate the bounds obtained by branch-and-bound, following cutting, and the column headed "GAP" indicates the gap between the two bounds. The column headed "lower" shows the values of the initial LP-relaxations (with single-node cutset inequalities) and the column headed "cut lower" shows the eventual lower bound proved by the exact method.

The following table shows the performance of APPROX on MS1-6. Here we used the enumeration procedure to generate cutset inequalities. This gave better results than the shortest-path based procedure but cost 129 seconds. The branching algorithm used depth-first branching, and was terminated after examining 25,000 nodes.

Problem	upper bound	opt. error (%)	ave. infeas.	num. infeas.	Time (sec)	cuts (I)	cuts (II)
MS1	3098.5	5.64	9.70×10^{-3}	35	129+110.0	59	444
MS2	3157.6	6.85	2.44×10^{-2}	33	129+19.7	107	21
MS3	3238.1	8.70	4.97×10^{-3}	22	129+227.6	166	629
MS4	3479.1	7.64	5.01×10^{-3}	35	129+128.6	67	507
MS5	3399.4	7.07	1.07×10^{-2}	32	129+151.2	102	830
MS6	2783.3	8.43	1.05×10^{-2}	42	129+108.9	117	825

Table 3: Performance of APPROX

In this table, "opt. error" is the percentage gap between the upper bound produced by APPROX and the best lower bound obtained by the exact algorithm, "ave. infeas." is the average relative infeasibility of the solution, averaged over all infeasible constraints (3), whose number is "num. infeas," "cuts (I)" is the number of cuts in the formulation at the end of the cutting-plane algorithm, and "cuts (II)" is the number of cuts added by the upper-bound heuristic. The first term in the "time" column is the time spent enumerating cutsets, and the second term is the time spent by APPROX proper. These times are on a 200 MHz PentiumPro running Solaris 2.5, on which APPROX runs twice as fast as on a Sparc 10/51.

Next we address two issues concerning Table 3: (a) the time comparison with the exact algorithm, and (b) the infeasibility of the solutions.

As stated above, the running times of APPROX should be magnified by a factor of two to compare with the exact algorithm. Further, of the 3600 seconds spent by the exact algorithm to obtain the stated bounds, 2050 were spent on pure cutting and 1550 on branching. After 3000 seconds, the exact algorithm already had a better bound than the one produced by APPROX, but of course, it had no bounds until the cutting step was completed. These time limits were crafted after some careful fine-tuning and it would not be easy to simply change them without downgrading the overall performance.

However, our (and other researchers') experience would indicate that simply terminating the cutting stage very early will lead to a very poorly performing algorithm, in terms of getting good bounds. To some degree this is an unfair remark, since at least part of the point in using "exact" cutting-plane methods is to obtain tight **lower** bounds. Nevertheless, beyond any doubt, if the exact method is only allowed to run for a few minutes, most likely it will not produce a formulation significantly better than the initial formulation, and as a result, in our estimation it will not be competitive with APPROX, from the point of view of getting reasonably good **upper** bounds. Part of the issue here is that, as a reader might notice, when running APPROX we are never really fully "solving" or "re-solving" a linear program, and that the cutting algorithm and the approximate LP-solver on the one hand, and the rounding heuristic and the approximate LP-solver on the other, are in each case woven together. In our opinion, this is the main reason why we are able to obtain quicker running times using APPROX. In summary, we see no way to cram the exact algorithm into a 300 second run-time limit and at the same time expect to get a competitive upper bound.

Of course, by radically changing the exact algorithm, it might be potentially possible to turn it into an upper-bound generating heuristic: in fact, one could use simplex calls to try to mimic APPROX, although not every single step could easily (or quickly) be replicated, in particular the flow rerouting step in out upper bound heuristic described before. This hypothetical algorithm *might* be comparable to APPROX on problems of size similar to those in Table 3, but this is pure guesswork. Given that as a standalone LP-solver simplex is vastly superior to our implementation of the ϵ-approximation methods on problems of this size, and given the maturity of CPLEX relative to that of APPROX, we view this result as positive evidence for our overall approach. Later we will see how on a larger problem APPROX becomes far superior.

Another alternative, suggested by a referee, would be the implementation of our exact algorithm using an interior point method (i.e. a barrier code) instead of a simplex solver. Such an implementation would be a separate research project (e.g. handling hot starts, branching, etc) unto itself, and beyond the scope of this paper. However, what we are able to say is that on LPs arising from networks of a size similar to those described above (say, 20 to 30 nodes, resulting in approximately one thousand columns and constraints) the (steepest-edge) simplex method runs four to five times faster than state-of-the art barrier methods. For larger problems (say, over 100 nodes) the reverse happens, assuming that memory is not a limitation. For example, on a 200-node, 1828-edge example described in section 5.4.1 the barrier method is several orders of magnitude faster than any simplex variant. However, requiring more than 9000 seconds, it is not what we want, in any case – we should add that little tailing-off effect was observed while running the barrier code. More on this in section 5.4.1.

A final issue to consider is whether the problems in Table 3 are somehow "easy." We would argue that the numbers in Table 2 show otherwise. In any case, consider problem **MS5**. Running this problem (without any cutting planes) with CPLEX 5.0 on a 167 MHz SUN Ultra, using strong branching and steepest-edge pivoting, we obtained the following results: until over 9700 seconds had elapsed, the best upper bound

CPLEX had obtained had value 3504.42 (i.e., some 3% worse than APPROX's 280-second bound on a slower machine), and the lower bound was 3033.08 (i.e. a more than 15% gap). After 75,000 seconds, the best upper bound was 3346.28 and the lower bound was 3059.48, a gap of about 9%. The run was terminated after 154,000 seconds with the gap still at 9%.

Now we turn to the infeasibilities. From the point of view of the practical application, these are rather small. In fact the infeasibility of the solutions is overstated in Table 3, simply because the pure-routing part of our ϵ-approximation algorithm cannot do better in the allotted time. One could engage in the following experiment: given the capacity vector x output by APPROX, fix it, and find a routing with small infeasibilities using the simplex method, a far easier linear program. The table below illustrates some of the results.

Problem	Version 1	Version 2	Version 3
MS1	2.45×10^{-3}	8.13×10^{-3}	3.94×10^{-2}
MS2	7.28×10^{-4}	4.76×10^{-3}	1.64×10^{-2}
MS3	0	0	0
MS4	0	0	0
MS5	4.84×10^{-4}	5.10×10^{-3}	1.04×10^{-2}
MS6	8.33×10^{-4}	4.45×10^{-3}	1.52×10^{-2}

Table 4: True infeasibilities of solutions in Table 3

In this table we describe three versions of infeasibility. In all of them we route the commodities so as to maintain feasibility of all capacity inequalities (3), but allowing some of the demand not to be routed. In Version 1, the routing minimizes the total amount that goes unrouted, and we report the ratio of this to the sum of all demands. In Version 2 we aggregate commodities by source, and we minimize the maximum fraction of any commodity that goes unrouted. In Version 3, the strictest, we do not aggregate demands and minimize the maximum fraction of any demand that goes unrouted. Notice that except for the MS1, Version 3 case, all the amounts are quite small, and even in that case only a few of the demands are affected.

To complete this section, in tables 5 and 6 we give the performance of APPROX on all remaining problem instances in [8]. On all these problems APPROX ran quickly, taking no more than 30 seconds after cut enumeration. In these tables, "exact" is the lower bound on value of the problem produced by the exact algorithm, "approx" is the value of the solution produced by APPROX, "error" is the gap between "approx" and "exact", "Etime" is the running time of the exact algorithm and "Atime" that of APPROX. (after the 129-second enumeration procedure).

First we have problems with very sparse demands.

Problem	NW1	NW2	SS1	SS2	SS3	SS4	SS5	SS6
exact	323.5	654.7	298.3	626.4	497.4	897.9	803.2	1003.1
approx	*	*	310.1	658.0	531.7	946.3	832.8	1071.0
error (%)	-	-	3.96	5.04	6.90	5.39	3.69	6.78
Etime (sec.)	49	1775	7	6	7	31	27	7
Atime (sec.)	< 30	< 30	< 30	< 30	< 30	< 30	< 30	< 30

Table 5: Sparse problems, * = all solutions discarded

On these problems the exact method is clearly superior, as it solved all of them to optimality in a few seconds, sometimes without any branching. Notice the poor performance in problems NW1 and NW2. There is a weakness in the way our algorithm handles sparse demand problems where the demands are also small. For example, problem NW1 has 18 demands, the largest of which has value 0.032 (this is probably somewhat artificial, given the application). Thus the sum of demands is smaller than 1.0, and problem CAP reduces to that of finding a subgraph containing a path between the source and destination of each demand. This is a pure $0, 1$-problem, in which the upper-bound heuristic in APPROX may not be the tool of choice.

The next set of problems are dense or very dense. These problems were all solved by APPROX with infeasibilities smaller than 10^{-3}.

Problem	NY1	NY2	FD1	FD2	FD3	FD4	FD5	FD6
exact	24804	78855	29549	29518	98133	97873	58909	58692
approx	25944	74388	29432	29321	101151	101191	61265	61077
error (%)	4.60	-5.66	-0.40	-0.67	3.08	3.39	4.00	4.06
Etime (sec.)	13	65	3600	3600	3600	3600	3600	3600
Atime (sec.)	< 30	< 30	< 30	< 30	< 30	< 30	< 30	< 30

Table 6: Denser problems

Problems NY1 and NY2 are on a smaller network than the Norwegian network and were solved to optimality by the exact method. The other problems are on the Norwegian network; the exact method was unable to solve any of them to optimality in one hour and on the row with label "exact" we have the best lower bound that was obtained by the exact method.

In summary, the very sparse problems and the two smaller dense problems were handled much better by the exact algorithm than by our approximation method, while the in the larger dense problems the approximation algorithm appears superior.

5.4.1 A very large problem.

Here we discuss a much more difficult problem instance, with all data real. The network in question has 200 nodes and 1828 edges. There are 1082 demands with 148 sources; the average demand value is 2.675.

We had to make some modifications to APPROX in order to handle this problem. First, all tolerances were somewhat relaxed. We also used the shortest-path procedure to separate cutset inequalities, implemented so as to favor "large" cuts (cuts separating large sets of nodes). At most 800 cuts were allowed in the formulation at any time.

The branching algorithm was parallelized so as to enumerate, evaluate and clean-up several nodes concurrently. Recall that the standard form of APPROX uses depth-first search. In the parallel version nodes are chosen in three different ways: newest node (i.e. depth-first search), oldest node (essentially breadth-first search) and by choosing a node of minimum "weight", where the weight depends on cost and infeasibility.

The resulting code ran in 9863 seconds on a 4x166 MHz Pentium machine, obtaining a solution of value 2.528×10^6 and average infeasibility 2.599×10^{-2}. The LP-relaxation (as before, with degree inequalities added) has value 2.352×10^6, hence our solution has optimality error at most 7.48%. Further, an infeasibility analysis as in Table 4 yields errors 1.53×10^{-4}, 3.85×10^{-3} and 4.07×10^{-3}.

It is difficult to say whether this problem is combinatorially intractable or not, modulo its large size. Our solution had average x_{ij} (over positive values) roughly 5.2, and thus it is unlikely that a one-unit error is being made on the average. Also, the demands are not so large that the problem is easy to round.

In any case, the LP-relaxation, with 272,372 columns and 31,576 rows, and almost one million nonzeros, is clearly quite difficult. CPLEX 5.0 (barrier) solved it in approximately 9500 seconds on a 167MHz SUN Ultra The simplex variants of CPLEX 5.0 all took more than one day to solve this problem. In contrast, in some 20 minutes APPROX found a 7%-optimal, 1%-feasible solution to the LP-relaxation. It is doubtful that without major modifications the exact algorithm can compete with APPROX on problems of this size. As a final note, we add that in current work with a new implementation of an ϵ-approximate LP-solver for *general* problems, using CPLEX 4.0 as a black box to solve the direction-finding subproblems, we have been able to solve this LP to 1% tolerance in about 2 minutes (167 SUN Ultra). CPLEX 6.0 (barrier) can solve it in about 2000 seconds on a 300 MHZ SUN Ultra, and CPLEX 6.0 (dual) takes about 4800 seconds on the same machine [4].

Appendix – core of ϵ-optimizer

For convenience, we represent all linking inequalities in the form $aX \geq b$, where $X = (x, f)$. Let D = the sum of all demands. First we describe an improvement phase (i.e. we are trying to achieve ϵ_t feasibility starting from something slightly worse).

IMPROVEPHASE(ϵ_t, N){

Let $\alpha = \gamma/\epsilon_t$.

while(max violation $\geq \epsilon_t$, and for at most N iterations){

/** compute violations, penalties and Lagrange multipliers **/

For each *active* linking constraint h, compute s_h, its scaled violation, $\pi_h = e^{\alpha s_h}$ and $v_h = \frac{\alpha}{b_h}\pi_h$.

Compute step direction for x-variables: For each variable x_{ij}, let $C_{ij} = c_{ij} + \sigma \sum v_h a_{h,ij}$, (summed over the active constraints). [here σ is as in (16) (c.f. (15))]. **If $C_{ij} < 0$, set $\delta^x_{ij} = D - x_{ij}$. Otherwise, set $\delta^x_{ij} = -x_{ij}$.**

Compute step direction for f-variables: For each commodity k, with source node $s(k)$ {

For each edge (i, j), set $\delta^f_{k,ij} = -f_{k,ij}$.

For each node $u \neq s(k)$, with positive demand $demand_{s(k),u}$ of commodity k, compute a shortest path from $s(k)$ to u, where the length of edge (i, j) is the exponential penalty associated with the variable upper-bound inequality (3) for (i, j), if active, zero otherwise. For each edge (i, j) **in the path**, reset $\delta^f_{k,ij} \leftarrow \delta^f_{k,ij} + demand_{s(k),u}$.

}

Step length: Find $0 \leq \lambda \leq 1$ such that $(x, f) + \lambda(\delta^x, \delta^f)$ minimizes the maximum relative violation, over all linking inequalities.

If $(\lambda = 0)$ {

Detect any directed cycle C with positive flows of any fixed commodity k, and decrease the flows of $f_{k,ij}$, $(i, j) \in C$, by their minimum.

Redefine the active inequalities to be those with relative violation at least 0.995 times the maximum, recompute the Lagrangean, the step direction and λ. If λ is still zero, **RETURN**.

}

}

Now we can describe the main loop.

LOOP(ϵ, N){

Initialization. Let $X = (x, f)$ be the current solution vector. Set $t = 0$, $\epsilon_0 = 0.1$, $\gamma = 0.2$.

while($\epsilon_t \geq \epsilon$){

IMPROVEPHASE(ϵ_t, N)

Set $\epsilon_{t+1} \leftarrow \epsilon_t/2$, and $t \leftarrow t + 1$.

}

}

Upon termination of this loop, if the maximum infeasibility is greater than ϵ, the system is deemed infeasible.

Acknowledgement. The author wishes to thank two anonymous referees for remarks that lead to improvements in the presentation of this paper.

References

[1] D. Applegate, R. Bixby, V. Chvátal and W. Cook, Solving the traveling salesman problem. In preparation (1998).

[2] A. Balakrishnan, T. Magnanti and R. Wong, A dual-ascent procedure for large-scale uncapacitated network design, *Oper. Res.* **37** (1989) 716-740.

[3] F. Barahona, Network design using cut inequalities, *SIAM J. Opt.* **6** (1996) 823-837.

[4] R. Bixby, personal communication.

[5] W. Cunningham, personal communication.

[6] S. Ceria and G. Pataki, personal communication.

[7] D. Bienstock and O. Günlük, Capacitated Network Design - Polyhedral Structure and Computation, *INFORMS J. on Computing* **8** (1996) 243-260.

[8] D. Bienstock, S. Chopra, O. Günlük and C. Tsai, Minimum Cost Capacity Installation for Multicommodity Network Flows (1995) to appear, *Math. Programming* **81** (1998), 177-199.

[9] D. Bienstock, Computational study of a family of mixed-integer quadratic programming problems, *Math. Programming* **74** (1996) 121-140.

[10] J.M. Borger, T.S. Kang and P.N. Klein, Approximating concurrent flow with unit demands and capacities: an implementation *DIMACS Series in Discrete mathematics and Theoretical Computer Science* **12** (1993) 371-385.

[11] B. Brockmüller, O. Günlük and L.A. Wolsey, Designing private line networks – polyhedral structure and computation, manuscript (1996).

[12] S. Ceria, P. Nobili and A. Sassano, A Lagrangian-based heuristic for large-scale set covering problems (1995) to appear, *Math. Programming* **81** (1998), 215-228.

[13] V. Chvátal, A greedy heuristic for the set-covering problem, *Math. Oper. Research* **4** (1975) 233-235.

[14] M.E. Dyer, Linear time algorithms for two- and three-variable linear programs, *SIAM Journal on Computing* **13** 31-45, 1984.

[15] J. Eckstein, personal communication, 1994.

[16] J. Eckstein, Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5, *SIAM Journal on Optimization* **4** 794-814, 1994.

[17] A. Goldberg, J.D. Oldham, S. Plotkin and C. Stein, An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flows, to appear, IPCO '98. (also: Technical report STAN-CS-TR-97-1600, Stanford University).

[18] M.D. Grigoriadis and L.G. Khachiyan, Fast approximation schemes for convex programs with many blocks and couping constraints, *SIAM Journal on Optimization* **4** (1994) 86-107.

[19] M.D. Grigoriadis and L.G. Khachiyan, An exponential-function reduction method for block-angular convex programs, *Networks* **26** (1995) 59-68.

[20] M.D. Grigoriadis and L.G. Khachiyan, Approximate minimum-cost multicommodity flows in $O(\epsilon^{-2}KNM)$ time, *Mathematical Programming* **75** (1996), 477-482.

[21] O. Günlük, personal communication (1996).

[22] Y. Jang, Development and Implementation of Heuristic Algorithms for Multicommodity Flow Problems, Ph.D. dissertation, Columbia University (1996).

[23] T. Leighton, F. Makedon, S. Plokin, C. Stein, É. Tardos, and S. Tragoudas, Fast approximation algorithms for multicommodity flow problems, In *Proceedings of the 23th Annual ACM Symposium on Theory of Computing,* (1991), 101-111.

[24] T. Leong, P. Shor and C. Stein, Implementation of a Combinatorial Multicommodity Flow Algorithm, DIMACS Series in Discrete mathematics and Theoretical Computer Science **12** (1993), 387-405.

[25] D.G. Luenberger, Linear and Nonlinear Programming. Addison-Wesley (1984).

[26] T. Magnanti, P. Mirchandani and R. Vachani, Modeling and solving the capacitated network loading problem, Working Paper OR 239-91, MIT (1991).

[27] T. Magnanti, P. Mirchandani and R. Vachani, The convex hull of two core capacitated network design problems, *Mathematical Programming* **60** (1993) 233-250.

[28] T. Magnanti, P. Mirchandani and R. Vachani, Modeling and Solving the Two Facility Capacitated Network Loading Problem, *Oper. Res.* **43** (1995), 142-157.

[29] M. Megiddo, Linear time algorithm for linear programming in R^3 and related problems, *SIAM Journal on Computing* **12** 759-776 (1983).

[30] P. Mirchandani, Projections of the capacitated network loading problem, (1992) manuscript (U. of Pittsburgh).

[31] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, New York (1988).

[32] K. Onaga and O. Kakusho, On Feasibility Conditions of Multicommodity Flows in Networks, *IEEE Transactions on Circuit Theory*, CT-18, No. 4, pp. 425-429.

[33] S. Plotkin and D. Karger, Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows, In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing,* (1995), 18-25.

[34] S. Plotkin, D.B. Shmoys and E. Tardos, Fast approximation algorithms for fractional packing and covering problems, *Proc. 32nd Annual IEEE Symp. On Foundations of Computer Science,* (1991), 495-504.

[35] Y. Pochet and L.A. Wolsey, Network design with divisible capacities: Aggregated flow and knapsack subproblems, *Proceedings IPCO2* (1992) 150-164, Pittsburgh.

[36] F. Shahrokhi and D.W. Matula, On solving large maximum concurrent flow problems, In *Proceedings of the ACM Computer Conference, ACM, New York,* (1987), 205-209.

[37] M. Stoer and G. Dahl, A Polyhedral Approach to Multicommodity Survivable Network Design, *Technical Report*, University of Oslo, Department of Informatics, (1995).

[38] L. Wolsey, personal communication.