APPROXIMATION ALGORITHMS FOR THE MULTI-LEVEL FACILITY LOCATION PROBLEM

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Nathan John Edwards

January 2001

APPROXIMATION ALGORITHMS FOR THE MULTI-LEVEL FACILITY LOCATION PROBLEM

Nathan John Edwards, Ph.D.

Cornell University 2001

The multi-level generalization of the facility location problem has been conspicuous in its absence from recent results improving the approximation constant for the facility location problem. The facility location problem assigns clients to facilities, paying a service cost for each assignment and a fixed cost for each opened facility. The optimal solution must find a balance between the service costs and the facilities used. The multi-level facility location problem is an important generalization that arises in such contexts as multi-level inventory hierarchies and network design. The optimal solution to this problem seeks a balance between the cost of routing demand via a facility of each level and the cost of installing the facilities used. We will present a number of techniques that make it possible to leverage the recent results for single-level problems to find good solutions for multi-level instances.

Biographical Sketch

Nathan John Edwards was born on July 26, 1970 in Perth, Australia. He entered the University of Western Australia in 1988, graduating in 1990 with a Bachelor of Science in Mathematics and Computer Science. He stayed for a fourth year, completing a Bachelor of Science with First Class Honors in 1991. After gradution from UWA, Nathan took a job with Telecom Research Laboratories in Melbourne, Australia. In the Network Analysis Section at TRL, Nathan learned a great deal about telecommunications and software engineering but eventually began to yearn for the opportunity to pursue academic research. Seeking not only education but also adventure, Nathan enrolled, in August 1994, in the Ph.D. program in the Field of Operations Research at Cornell University, in Ithaca, NY. Upon graduation from Cornell, Nathan will join Celera Genomics in Rockville, MD.

To Mum & Dad

Acknowledgements

As I look back over the course of my graduate studies, I am deeply grateful to the many people who have each, in their own way, touched my life during my time at Cornell.

I wish to thank my advisor and mentor, David Shmoys, for his advice, encouragement, patience, and confidence in me.

I also thank Sid Resnick and Tom Coleman for serving as members of my committee.

I am grateful for financial support during my first three years at Cornell courtesy of a Hackett Studentship from the University of Western Australia.

My undergraduate honors advisor "C.J." Goh provided my first chance to do research, planting a seed that grew into a Ph.D. I thank him for his foresight.

I have made some lasting friendships during my time at Cornell. Thanks to David Williams, Ed Chan and Semyon Kruglyak, for being true-blue mates.

To the Rolfe family, Jim and Margaret, Casey and Caitlin, my heartfelt thanks for making me part of your family.

I thank the members of my graduate class, Greta Pangborn, Ed Chan, Fabian Chudak, Mark Huber, Semyon Kruglyak, and Paulo Zanjacomo, who have shared my highs and lows, good times and bad.

Kathy Caggiano deserves special mention for her gracious hospitality, providing me with a place to stay during the last few weeks of my time in Ithaca.

To the members of the field hockey club at Cornell, particularly Andre Sequin, and Dwayne and Amanda Kirk, thanks for some great times, on and off the field.

Finally, I thank my Mum and Dad, for their ongoing prayers, encouragement, and love.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Mathematical Programs

Chapter 1

Introduction

Operations research is the study of problems motivated by and abstracted from real-world business decisions. Many of these problems concern minimizing the cost of providing some service by the judicious selection of one course of action out of a set of alternatives. Combinatorial optimization provides a way to model these problems by requiring the selection of an answer from a discrete set of possibilities. A combinatorial optimization model of a real-world business decision rarely answers the business question outright, but it does provide excellent prototype answers, valuable insight into the structure and properties of good answers, and building blocks from which practical algorithms can be built.

The abstract combinatorial optimization problems distilled from real-world business decisions are usually very difficult to solve, despite capturing only the essential elements of the real decision. The academic community measures the difficulty of an abstract problem by the time required to solve the most difficult instances of the problem as a function of the size of the instance. Problems of similar difficulty are then grouped into classes. Problems that can be solved in time polynomial in the size of the problem, for even the most difficult instances, are considered easy. Hard problems are those for which no such polynomial-time algorithm is known, which means that we have no algorithm to solve these problems that is qualitatively better than the explicit enumeration of all possible solutions. Garey and Johnson [GJ79] popularized the term *NP-hard* for this class of hard problems. Almost all of the combinatorial optimization abstractions motivated by real-world business decisions, with a few notable exceptions, are *NP*-hard. In the words of Garey and Johnson, this means "I can't find an efficient algorithm, but neither can all these famous people." ([GJ79], p.3)

The well known shortest path problem is in the class of easy problems. Since it can be solved quickly, it is used in practical settings where the running time is very important. It can also be used as a subroutine to help solve more difficult problems. We will see a number of examples of this throughout this thesis.

The network design problem is an example of a *NP*-hard combinatorial optimization problem. Given a communications network consisting only of terminals, the network design problem asks what communications links must be installed so that all pairs of communicating terminals have enough bandwidth, redundancy, or bitrate on some route through the resulting network. Other *NP*-hard problems try to balance two types of cost associated with a resource, a fixed installation cost and a per use cost. The *NP*-hard multi-level facility location problem contains aspects of both of these types of *NP*-hard problems.

Since we do not know how to solve NP-hard problems to optimality in polynomial time, we must consider alternative approaches. We will evaluate the alternatives in light of their practicality for solving problems motivated by real-world business decisions.

The real-world business decision context provides us with some room to maneuver that our theoretic notions of optimality and easy versus hard do not. The business context does not require an optimal solution, just a very good one. The business context does not care whether or not an algorithm produces good solutions on the hardest instances, only that it produces very good solutions on today's instances. On the other hand, the business context *does* care whether or not an algorithm can be run on very large instances, and therefore memory footprint and running time are critical considerations.

The first alternative to computing optimal solutions in polynomial time is to construct algorithms that take exponential time for the most difficult instances, but that find the optimum fast enough for typical instances. In practice, the most common examples of this approach are the branch-and-bound and branch-and-cut algorithms for solving integer programs. Modern implementations of these algorithms are significantly faster than explicit enumeration on typical instances of hard problems, particularly when tuned to take advantage of the problem specific polyhedral structure of the integer programming formulation. However, when an instance proves too difficult to solve quickly and the algorithm is stopped before proving optimality, we may be left with no solution at all, or a solution of questionable quality.

Heuristics provide a second alternative. Heuristics are usually designed to take advantage of common sense intuition about a problem. Heuristics usually produce excellent solutions very quickly, provided the instances being considered happen to agree with the relevant intuition. Unfortunately, typical instances from a particular context often fail to cooperate. Practically, we run many heuristics, each based on different intuition, on the same problem instance and take the best solution, hoping that not all the heuristics will do badly on the instance at hand.

The third alternative is approximation algorithms. Approximation algorithms relax the optimality requirement, instead guaranteeing a solution within some factor of optimal in polynomial time. Approximation algorithms guarantee a reasonable running time and a good solution on even the most difficult instances. Approximation algorithms typically run more quickly than exponential-time algorithms in practice, having sacrificed optimality. On the other hand, approximation algorithms are usually slower than heuristic approaches, but guarantee a good solution for instances that fall outside the domain of the available heuristics. In practice, approximation algorithms almost always produce much better solutions than their guarantees for the most difficult instances suggest, making them good, robust heuristics. The solutions from approximation algorithms are sometimes used as backups when integer programming or heuristic approaches fail, or as good starting solutions for heuristics that make incremental improvements.

Vašek Chvátal summed up these alternatives at the 2000 Symposium on Discrete Algorithms with the quip (paraphrased), "Optimal, fast, guaranteed - choose any two."

This thesis will concentrate on approximation algorithms for the *NP*-hard multi-level facility location problem.

1.1 Approximation Algorithms

Approximation algorithms guarantee, in polynomial time, a solution to a problem of objective function value within some factor of the value of the optimal solution, for each instance of a problem. We say an algorithm is a ρ-approximation algorithm if, for a minimization problem, the algorithm produces a feasible solution of value at most ρ times the optimal value. The factor ρ is called the approximation guarantee, and the solution is called a ρ-approximation. We will make a distinction between approximation algorithms, which run in polynomial time, and approximation techniques, for which we have no such running-time bound.

Approximation algorithms for *NP*-hard combinatorial optimization problems have proven to be a particularly successful way to generate excellent solutions for problems long considered intractable in practice. The recent book edited by Hochbaum [Hoc97] provides a comprehensive survey of the application of approximation algorithms to *NP*-hard problems.

In addition to their practical utility, approximation algorithms also provide a way to further classify *NP*-hard problems. Intuitively, an *NP*-hard approximation algorithm that has a constant approximation algorithm is considered easier than a *NP*-hard approximation for which no constant approximation algorithm is known. For some *NP*-hard problems, we can show there is a family of approximation algorithms achieving an approximation guarantee of ρ for all $\rho > 1$. Problems for which such a family of approximation algorithms exist are said to have a *polynomial time approximation scheme*, or *PTAS*. For other problems, it is possible to show that no PTAS exists, unless our class of easy problems is the same as our class of hard problems. Problems with this property are called *MAX-SNP*-hard, see [AL97] for a survey. A problem is shown to be *MAX-SNP*-hard by demonstrating that if an approximation algorithm with a specific constant approximation guarantee ρ exists, then some other *NP*-hard problem can be solved in polynomial time. *MAX-SNP*-hard problems are considered more difficult than problems for which a PTAS is known.

1.2 Facility Location

The *facility location problem* concerns the optimal number and location of facilities needed to provide some service to a set of clients. The optimal solution must balance two types of costs — the cost of establishing a facility at a particular location, and the total cost of providing service to each of the clients from one of the opened facilities.

In its simplest form, each opened facility may serve an unlimited number of clients, in which case the problem is called *uncapacitated*. If each opened facility can serve only a limited number of clients the problem is called *capacitated*.

A restriction is often placed on the nature of the cost of providing service to a client from a potential facility site. The service costs are said to be *metric* if the cost of providing service to a client from a facility is proportional to some notion of the distance between them.

This thesis will concentrate on the *uncapacitated metric facility location problem*, also called the *location-allocation problem* and the *simple plant location problem* in the operations research literature.

Applications

The facility location problem has been applied in a large number of settings. Traditionally, the facility location problem has been applied to the problem of locating warehouses or factories to service retail outlets, as evidenced by the terminology used to describe the problem. Examples of can be found in the work of Stollsteimer [Sto61] & [Sto63], Kuehn and Hamburger [KH63], and Manne [Man64]. The facility location problem has since found applications in many other settings, including the location of fire stations or hospitals in a city; locating base stations in cellular and wireless networks; star topology network design; and clustering of documents for information retrieval. An extensive survey can be found in the textbook edited by Mirchandani and Francis [MF90], in particular the chapter by Cornuéjols, Nemhauser, and Wolsey [CNW90].

Problem Statement

The input to the uncapacitated metric facility location problem consists of a set of facilities F, a set of clients D, a facility cost f_i for each $i \in F$, and a metric that defines service costs c_{ij} for each $i, j \in F \cup D$. Feasible solutions must assign each client to a facility. The set of facilities used by the clients in a feasible solution shall be denoted I throughout. A feasible solution is charged the sum of the facility costs of the facilities used, plus the total of the service costs of the clients' assignments. An optimal solution to the uncapacitated facility location problem is a feasible solution of minimum cost.

In fact, the set of facilities used implicitly gives a complete description of a feasible solution. Suppose we are given a set of facilities I. By assigning each client to the facility that minimizes the service cost paid, we have a feasible solution that dominates all feasible solutions that install exactly the facilities of I.

We will be concerned with the *metric* version of the uncapacitated facility location problem. Metric instances have service costs satisfying the following conditions:

$$
\begin{array}{lll}
c_{ij} \geq 0 & \text{for all } i, j \in F \cup D, & \text{[Non-Negativity]} \\
c_{ij} = c_{ji} & \text{for all } i, j \in F \cup D, & \text{[Symmetry]} \\
c_{ii'} \leq c_{ij} + c_{ji'} & \text{for all } i, i', j \in F \cup D. & \text{[Triangle Inequality]}
\end{array}
$$

Integer Programming Formulation

The uncapacitated facility location problem is modeled by the integer programming formulation (IP_{FL}). The 0-1 variables y_i represent whether or not a facility i is used, while the 0-1 variables x_{ij} represent whether or not client j has been assigned to facility i. The constraint (1.2) enforces the condition that each client is assigned to

$$
\begin{aligned}
(\mathrm{IP_{FL}}) \qquad \min \quad & \sum_{i\in F} f_i y_i + \sum_{i\in F}\sum_{j\in D} c_{ij}x_{ij} && && (1.1)\\
\text{s.t.} \quad & \sum_{i\in F} x_{ij} = 1, && \text{for each } j \in D, && (1.2)\\
& x_{ij} \le y_i, && \text{for each } i \in F, j \in D, && (1.3)\\
& x_{ij} \in \{0,1\}, && \text{for each } i \in F, j \in D, && (1.4)\\
& y_i \in \{0,1\}, && \text{for each } i \in F. && (1.5)
\end{aligned}
$$

some facility location, while the constraint (1.3) guarantees that if a facility is used by some client, then it is paid for.

Previous Work

Until quite recently, no constant approximation algorithm for the uncapacitated facility location problem was known. Hochbaum [Hoc82] and Lin and Vitter [LV92b] found algorithms with a $O(\log |D|)$ approximation guarantee for instances with arbitrary service costs, but it was not until Shmoys, Tardos and Aardal considered the special case of metric service costs [STA97] that the first constant approximation algorithm was discovered.

The constant approximation algorithm of Shmoys, Tardos and Aardal solves the linear programming relaxation of ($\mathrm{IP_{FL}}$) and rounds the fractional solution to an integer solution. This algorithm achieves a approximation constant of $\approx$ 3.16. Soon after, Chudak, working with Shmoys, was able to improve the rounding algorithm, see [Chu98a], [Chu98b] and [CS], achieving a significant reduction in the approximation constant to ≈ 1.736.

The linear program solution rounding approach is a powerful approximation algorithm design technique, often producing excellent approximation guarantees. However, this approach requires the solution of the linear relaxation of ($\mathrm{IP}_{\mathrm{FL}}$), which can be very time consuming for large instances. The first combinatorial constant approximation algorithm for the facility location problem is due to Korupolu, Plaxton and Rajaraman [KPR98]. This result demonstrated that a local search heuristic often used by practitioners can be shown to be a $(5+\varepsilon)$-approximation algorithm. Prior to the work of Korupolu, Plaxton and Rajaraman, Guha and Khuller demonstrated the power of local search heuristics [GK98] with polynomially many instances the linear relaxation of ($\mathrm{IP}_{\mathrm{FL}}$), an application of the rounding procedure of Shmoys, Tardos and Aardal, and a greedy local search heuristic, achieving an approximation constant of ≈ 2.414.

Also around this time, a second combinatorial approach, based on the primal-dual algorithm design technique of Goemans and Williamson [GW95], was shown to achieve an approximation constant of 3 for this problem. This algorithm, due to Jain and Vazirani [JV99] uses the linear programming relaxation of ($\mathrm{IP}_{\mathrm{FL}}$) and its dual to guide the construction of its combinatorial solution and to argue about its quality.

Most recently, a straightforward new local search scheme was shown by Charikar and Guha [CG99] to achieve an approximation constant of 3. When combined with cost scaling and a modified version of the primal-dual algorithm of Jain and

Vazirani, this algorithm yields a ≈ 1.853-approximation algorithm, without having to solve any linear programs. When further combined with the rounding algorithm of Chudak and Shmoys, the resulting hybrid achieves the current best approximation constant of ≈ 1.726, though this hybrid algorithm must solve a polynomial number of instances of the linear programming relaxation of $(\mathrm{IP}_{\mathrm{FL}})$.

We point out that the facility location problem, even in its metric form, is *MAX-SNP*-hard. A result of Guha and Khuller [GK98] combined with a result of Sviridenko show that we cannot hope to achieve an approximation constant less than ≈ 1.463 unless $P = NP$.

To provide a foundation for applying these algorithmic techniques to the multi-level facility location problem, we briefly describe the most recent application of each of these techniques to the facility location problem. We refer the reader to the relevant papers for the details.

Linear Program Solution Rounding

While the linear program solution rounding algorithm of Chudak and Shmoys is straightforward to state, its analysis is not. See [Chu98a], [Chu98b] and [CS] for the details.

The algorithm of Chudak and Shmoys first solves the linear programming relaxation $(\mathrm{LP}_{\mathrm{FL}})$ of $(\mathrm{IP}_{\mathrm{FL}})$ formed when (1.4) and (1.5) are replaced with non-negativity constraints. We denote the optimal solution of $(\mathrm{LP}_{\mathrm{FL}})$ by (x, y).

$$
\begin{aligned}
(\mathrm{LP_{FL}}) \qquad \min \quad & \sum_{i \in F} f_i y_i + \sum_{i \in F} \sum_{j \in D} c_{ij} x_{ij} && && (1.6)\\
\text{s.t.} \quad & \sum_{i \in F} x_{ij} = 1, && \text{for each } j \in D, && (1.7)\\
& x_{ij} \le y_i, && \text{for each } i \in F, j \in D, && (1.8)\\
& x_{ij} \ge 0, && \text{for each } i \in F, j \in D, && (1.9)\\
& y_i \ge 0, && \text{for each } i \in F. && (1.10)
\end{aligned}
$$

$$
\begin{aligned}
(\mathrm{DP_{FL}}) \qquad \max \quad & \sum_{j \in D} v_j && && (1.11)\\
\text{s.t.} \quad & v_j - w_{ij} \le c_{ij}, && \text{for each } i \in F, j \in D, && (1.12)\\
& \sum_{j \in D} w_{ij} \le f_i, && \text{for each } i \in F, && (1.13)\\
& w_{ij} \ge 0, && \text{for each } i \in F, j \in D. && (1.14)
\end{aligned}
$$

The algorithm also makes use of the optimal solution to the dual of ($\mathrm{LP}_{\mathrm{FL}}$), denoted ($\mathrm{DP}_{\mathrm{FL}}$), with optimal solution (v, w). Typically, the optimal dual solution is obtained in the process of solving the primal problem ($\mathrm{LP}_{\mathrm{FL}}$).

To state the rounding algorithm of Chudak and Shmoys, we need a few definitions. The set of facilities near a client j with respect to our optimal primal solution is defined as $N(j) = \{i : x_{ij} > 0\}$. The fractional assignment cost of client j, paid by the optimal fractional solution, is $C_j = \sum_{i \in F} c_{ij} x_{ij}$. The rounding algorithm of Chudak and Shmoys uses $N(j)$ and C_j in Algorithm 1.1: FindClusters to compute a clustering of the clients and facilities that will guide the rounding step. The cluster centered at some client j contains all the facilities of $N(j)$, and all clients j' that have a facility of $N(j)$ in $N(j')$.

In the rounding step Algorithm 1.2: LPRounding, each of the cluster centers $j \in S$ will randomly select a facility from $N(j)$. Each facility R not considered by some cluster center is then considered for selection. Note that LPRounding, as presented, assumes (x, y) satisfies a property called completeness, see [Chu98a] and [CS] for details.

The construction of clusters around special clients, introduced by Shmoys, Tardos and Aardal [STA97], is the key idea of this algorithm. These special clients, the cluster centers, are chosen to ensure they are good proxies for the other clients in the cluster. The facility selections are made based solely on the preferences of the cluster centers. The cluster center selection criteria make the facility decisions appropriate for the other clients in each cluster. Note that this is only true for

Algorithm 1.1 FINDCLUSTERS

$S \leftarrow \emptyset, J \leftarrow D, R \leftarrow F.$ {Initialize.}
while $J \neq \emptyset$ **do**
  Select $j \in J$ with minimum $C_j + v_j$. {j is a new cluster center.}
  $S \leftarrow S \cup \{j\}$. {Add j to set of cluster centers.}
  $R \leftarrow R \setminus N(j)$. {Remove facilities in j's cluster from R.}
  $J \leftarrow J \setminus \{j' \in J : N(j') \cap N(j) \neq \emptyset\}$. {Remove clients in j's cluster from J.}

Algorithm 1.2 LPROUNDING

for all $j \in S$ **do** {j is a cluster center.}
  Choose one $i \in N(j)$ with probability x_{ij}, $I \leftarrow I \cup \{i\}$.
  {Randomly choose one facility from j's cluster.}
for all $i \in R$ **do** {Facility i is not in a cluster.}
  $I \leftarrow I \cup \{i\}$ independently with probability y_i. {Consider i for selection.}
Assign each client to the closest facility of I.

metric instances! The key use of the metric assumption is formalized in Lemma 1 for later reference.

Lemma 1. *Suppose the service costs* c_{ij}, $i, j \in F \cup D$ *are metric. If* $i, i' \in F$ *and* $j, j' \in D$, *then* $c_{ij'} \leq c_{ij} + c_{i'j} + c_{i'j'}$.

Proof. See Figure 1.1 for a picture proof of this simple application of the triangle inequality. □

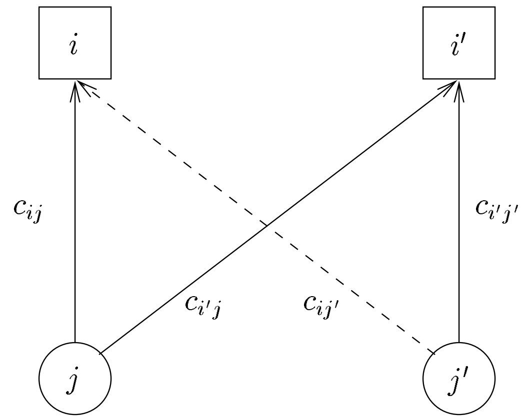

Figure 1.1: Picture proof of Lemma 1

Coupled with judicious selection of our cluster centers j, Lemma 1 makes it possible to bound the service cost that j' pays when it is assigned to i instead of its preferred i'.

Local Improvement

The local improvement heuristic of Charikar and Guha can be stated with varying degrees of sophistication; we present only the simplest version here. See [CG99] for more details.

Given a sufficiently good initial solution, Charikar and Guha show that by repeatedly choosing a facility at random and including it in our solution if doing so improves the total cost, we will, after a polynomial number of iterations, obtain a solution of cost at most a factor of 3 more than the optimal cost.

Algorithm 1.3: ComputeGain computes an lower bound on the amount that the total cost of the current solution would improve if a facility i was added to our

Algorithm 1.3 ComputeGain($i \in F$)

$A \leftarrow \{j \in D : c_{ij} < c_{\sigma(j)j}\}$. {Clients in A should be reassigned.}

for all $i_0 \in I$ **do**

  if $f_{i_0} > \sum_{j \in D(i_0) \setminus A} (c_{ij} - c_{\sigma(j)j})$ **then** {Should we remove i_0 from I?}

    $A \leftarrow A \cup D(i_0)$. {Clients in $D(i_0)$ should be reassigned}

    $R \leftarrow R \cup \{i_0\}$. {and i_0 removed from I.}

return $\sum_{i_0 \in R} f_{i_0} - \sum_{j \in A} (c_{ij} - c_{\sigma(j)j})$.

current solution I. Let $\sigma(j)$ be the facility to which client j is currently assigned and let $D(i)$ be the set of clients currently assigned to facility i. ComputeGain first checks whether any client j is closer to i than to $\sigma(j)$. ComputeGain then checks whether any node i_0 of I can be removed, even though we must reassign the clients $D(i_0)$ to i. Note that ComputeGain can, in fact, recommend the removal of a number of facilities from I and that we may consider the "inclusion" of a facility that is already in I.

The local search algorithm starts with the initial solution formed by assigning each client to its closest facility and setting I to the facilities used. Algorithm 1.4: LocalImprovement then repeatedly chooses a facility location i from F uniformly at random and calls $g \leftarrow$ ComputeGain(i) to evaluate it. If the randomly chosen node is already in I, then if g is positive we proceed with the proposed change. If the randomly chosen node is not in I, then we must have $g > f_i$ in order for the change to be beneficial.

Algorithm 1.4 LOCALIMPROVEMENT

1: Initialize I with solution of minimum service cost.
2: **for** $O(|F| \log |F|)$ iterations **do** {Sufficient in expectation.}
3: Choose $i \in F$ uniformly at random; $g \leftarrow$ COMPUTEGAIN(i).
4: **if** $(i \in I$ **and** $g > 0)$ **or** $(i \notin I$ **and** $g > f_i)$ **then** {Enough gain?}
5: $\sigma(j) \leftarrow i$ for all $j \in A$; $D(i) \leftarrow D(i) \cup A$. {Reassign clients of A.}
6: $I \leftarrow I \setminus R$; $D(i_0) \leftarrow \emptyset$ for all $i_0 \in R$. {Remove facilities of R from I.}
7: Assign each client to the closest facility of I.

While the notion of clustering around special clients is absent in the proof of correctness of this algorithm, nevertheless, the critical use of the metric assumption on the service costs appears, as it did for the rounding algorithm of Chudak and Shmoys, as shown in Lemma 1.

Primal-Dual Schema

The application of the primal-dual technique of Goemans and Williamson [GW95] to the facility location problem by Jain and Vazirani [JV99] extends the technique to a new class of problems. Previously, the primal-dual technique had only been applied to a class of network design problems with constraints modeled by covering cuts by edges. The primal-dual algorithm of Jain and Vazirani uses the linear program $(\mathrm{LP}_{\mathrm{FL}})$ and its dual $(\mathrm{DP}_{\mathrm{FL}})$ to guide a combinatorial solution construction. This approach avoids solving the linear program but makes it possible to use the theoretical tools of linear programming duality anyway.

The algorithm is composed of an expansion phase, which constructs a feasible dual solution (v, w), followed by a rationalization phase, which forms a corresponding primal solution. During the expansion phase, each client j uniformly increases the amount v_j it is prepared to pay to be assigned to a facility. The amount each client promises to pay goes first to pay the service cost between the client and a facility, and then contributes towards the shared facility cost. Once enough contributions towards the cost of a facility have been promised, each client promising to contribute to a facility is assigned to it. In the rationalization phase, we must make sure that the clients actually pay for only one facility, despite promising to contribute to a number of facilities. Ensuring this means that some clients must be connected to a backup facility, but the metric assumption guarantees this does not increase the cost of the solution too much.

The expansion phase Algorithm 1.5: PRIMALDUALEXPANSION can be thought of as a series of events occurring over time, as each client increases the amount it is prepared to pay to be assigned to a facility. The event list $\mathcal{E}$ contains all pending events. The first type of event CONTRIBUTE(i, j) occurs when the dual constraint (1.12) becomes tight for some client-facility pair (i, j) because j has promised to pay at least the service cost between j and i. From this point on, j is promising to contribute towards the facility cost of i. The second type of event PAIDFOR(i) occurs when the facility cost of some facility i is paid for by the contributions from the clients. This corresponds to the dual constraint (1.13). We keep track of $F(j)$, the facilities that client j promises to help pay for; $D(i)$, the clients promising to

Algorithm 1.5 PRIMALDUALEXPANSION

$t_0 \leftarrow 0$; $F(j) \leftarrow \emptyset$ for each $j \in D$; $D(i) \leftarrow \emptyset, n_i \leftarrow 0, r_i \leftarrow f_i$ for each $i \in F$.
Add event CONTRIBUTE(i, j) to $\mathcal{E}$ at time c_{ij}, for each $i \in F, j \in D$.
while some client is not yet assigned **do**
  Pop next event and time t_1 from $\mathcal{E}$; $r_i \leftarrow r_i - (t_1 - t_0) * n_i$ for each $i \in F$.
  if next event is CONTRIBUTE(i, j) and j is not yet assigned **then**
    if $i \notin I$ **then**
      $F(j) \leftarrow F(j) \cup \{i\}$; $D(i) \leftarrow D(i) \cup \{j\}$; $n_i \leftarrow n_i + 1$.
      Add or reset event PAIDFOR(i) in $\mathcal{E}$ to time $t_1 + r_i/n_i$.
    else if $i \in I$ **then**
      Client j is assigned to facility location i.
      $n_{i'} \leftarrow n_{i'} - 1$ for each $i' \in F(j) \setminus I$.
      Reset PAIDFOR(i') in $\mathcal{E}$ to time $t_1 + r_{i'}/n_{i'}$ for each $i' \in F(j) \setminus I$.
  else if next event is PAIDFOR(i) **then**
    $I \leftarrow I \cup \{i\}$.
    for all unassigned $j \in D(i)$ **do**
      Client j is assigned to facility location i.
      $n_{i'} \leftarrow n_{i'} - 1$ for each $i' \in F(j) \setminus I$.
      Reset PAIDFOR(i') in $\mathcal{E}$ to time $t_1 + r_{i'}/n_{i'}$ for each $i' \in F(j) \setminus I$.
  $t_0 \leftarrow t_1$.

Algorithm 1.6 PrimalDualRationalization

1: **for all** $i \in I$ in order added if not yet removed **do**

2: $R \leftarrow \emptyset$.

3: **for all** j such that $i \in F(j)$ **do** {Clients j promising to help pay for i.}

4: $R \leftarrow F(j)$. {Delete all other facilities j promised to help pay for ...}

5: $R \leftarrow R \setminus \{i\}$. {... but not i.}

6: Reassign all clients assigned to a facility of R to i.

7: $I \leftarrow I \setminus R$.

8: Assign each client to the closest facility of I.

help pay for i; n_i, the number of clients currently promising to help pay for i; and r_i, the facility cost of facility i currently remaining to be paid.

The rationalization phase Algorithm 1.6: PrimalDualRationalization considers the facilities in the order that they were added to I. If a client promised to contribute towards the cost of two or more facilities, only the first one is kept. Any client assigned to one of the deleted facilities is reassigned to the facility responsible for deleting its assigned facility.

Like the linear program solution rounding algorithm of Chudak and Shmoys [CS], this algorithm forms clusters. However, unlike the algorithm of Chudak and Shmoys, the clusters are centered at facilities, the facilities remaining in I after the rationalization phase. Nevertheless, as before, Lemma 1 codifies the use of the metric assumption in the analysis of this algorithm. Lemma 1 ensures that the reassignment step of PrimalDualRationalization increases the service cost for the reassigned clients by at most a factor of 3.

1.3 Multi-Level Facility Location

The *k-level facility location problem* generalizes the facility location problem. The multi-level version requires that each client be assigned to a sequence of k facilities, each of which belongs to a distinct class or level. As with the facility location problem of Section 1.2, the solution to a k-level facility location problem must balance the cost of routing the clients through their assigned sequence of facilities, and the cost of the facilities used.

The multi-level version of the facility location problem may be posed capacitated or uncapacitated, metric or non-metric, just as the facility location problem may be. We will again concentrate on the *uncapacitated metric k-level facility location problem*.

Applications

The multi-level facility location problem has been traditionally applied to the location of warehouses and distribution centers in a hierarchical goods distribution network or supply chain. See Kaufman, van den Eede, and Hansen [KvdEH77]; Van Roy and Erlenkotter [VE82]; Tcha and Lee [TL84]; de Barros [dB95]; and Aardal, Labbé, Leung and Queyranne [ALLQ96] for examples. The work in this thesis was originally motivated by the access network design problem, first posed by Andrews and Zhang [AZ98]. The access network design problem must connect a set of clients to a central exchange or hub, installing communications links to carry the clients' traffic. Each communications link has a fixed cost that must be paid regardless of how many clients use it and a cost based on the number of clients using it. The multi-level facility location problem must also trade fixed cost against per use cost in a network routing setting, but instead places fixed costs on nodes rather than edges and imposes a rigid structure on the network being constructed.

Problem Statement

The input to the uncapacitated metric k-level facility location problem consists of a set of facilities F partitioned into k levels, denoted $F_1, \ldots, F_k$, a set of clients D, a facility cost f_i for each $i \in F$, and a metric that defines service costs c_{ij} for each $i, j \in F \cup D$. Feasible solutions must assign each client to a sequence of k facilities, one from each level $F_k, \ldots, F_1$, in order. The set of facilities used by the clients in a feasible solution shall be denoted I throughout. A feasible solution is charged the sum of the facility costs of the facilities used, plus the total of the service costs of the clients' assignments. Each client's service cost is the sum of service cost from itself to the first facility of its sequence, plus the service cost between successive facilities of its sequence. An optimal solution to the uncapacitated metric k-level facility location is a feasible solution of minimum cost.

It will be convenient to formally define the graph G implicit in this problem statement. G is a directed graph defined on the nodes $F \cup D$. The edges E of G are of the form $i_l i_{l-1}$ for $i_l \in F_l, i_{l-1} \in F_{l-1}, i = 2, \ldots, k$ or $j i_k$ for $j \in D, i_k \in F_k$.

For each $ij \in E$, we define the service cost of the edge ij to be c_{ij}. The assignment of a client $j \in D$ to a valid sequence of facilities can now be represented as the assignment of j to a path p in G from j to one of the top level facilities F_1. We will switch between these two characterizations of valid facility sequences at will.

As with the facility location problem, the set of facilities used implicitly gives a complete description of a feasible solution to the multi-level facility location problem. Suppose we are given a set I of facilities containing at least one facility location of each level. We shall denote the subgraph of G induced by removing the facilities not in I by G_I. Each client j is then assigned to the shortest path, with respect to service cost, from j to one of the top level facilities of $F_1 \cap I$ in G_I. This feasible solution dominates all feasible solutions that install exactly the facilities of I.

Integer Programming Formulation

The uncapacitated multi-level facility location problem is modeled by the integer programming formulation ($\mathrm{IP}_{\mathrm{MLFL}}$). We define the set of all valid sequences of facilities by $P = F_k \times \cdots \times F_1$ and the service cost of client j's assignment to sequence $p = (i_k, \ldots, i_1)$ by $c_{pj} = c_{ji_k} + c_{i_k i_{k-1}} + \cdots + c_{i_2 i_1}$. The 0-1 variable y_i represents whether or not a facility i is used and the 0-1 variable x_{pj} represents whether or not a client j is assigned to the path p.

Constraint (1.16) ensures that every client is assigned to a path while constraint (1.17) guarantees that any facility on a path used by some client is paid for.

$$
\begin{aligned}
(\mathrm{IP}_{\mathrm{MLFL}}) \qquad & \min \quad \sum_{i \in F} f_i y_i + \sum_{p \in P} \sum_{j \in D} c_{pj} x_{pj} && && (1.15)\\
& \text{s.t.} \quad \sum_{p \in P} x_{pj} = 1, && \text{for each } j \in D, && (1.16)\\
& \qquad \sum_{p \ni i} x_{pj} \le y_i, && \text{for each } i \in F, j \in D, && (1.17)\\
& \qquad x_{pj} \in \{0,1\}, && \text{for each } p \in P, j \in D, && (1.18)\\
& \qquad y_i \in \{0,1\}, && \text{for each } i \in F. && (1.19)
\end{aligned}
$$

In fact, constraint (1.17) can be replaced by the weaker

$$
x_{pj} \le y_i, \qquad \text{for each } i \in p, p \in P, j \in D \tag{1.20}
$$

and $(\mathrm{LP}_{\mathrm{MLFL}})$ will still model the multi-level facility location problem. The stronger constraint is critical, however, for the linear programming relaxation. We will elaborate on this in due course.

Previous Work

The uncapacitated metric multi-level facility location problem has not seen the frenetic progress enjoyed by the facility location problem in the last couple of years. Nevertheless, the ≈ 3.16-approximation result of Shmoys, Tardos and Aardal [STA97] for the facility location problem was also shown by Shmoys, Tardos and Aardal to apply to the 2-level facility location problem. This result was recently extended by Aardal, Chudak and Shmoys [ACS99] to a 3-approximation algorithm

$$
\begin{aligned}
(\mathrm{LP}_{\mathrm{MLFL}}) \qquad & \min \quad \sum_{i \in F} f_i y_i + \sum_{p \in P} \sum_{j \in D} c_{pj} x_{pj} && && (1.21)\\
& \text{s.t.} \quad \sum_{p \in P} x_{pj} = 1, && \text{for each } j \in D, && (1.22)\\
& \qquad \sum_{p \ni i} x_{pj} \leq y_i, && \text{for each } i \in F, j \in D, && (1.23)\\
& \qquad x_{pj} \geq 0, && \text{for each } p \in P, j \in D, && (1.24)\\
& \qquad y_i \geq 0, && \text{for each } i \in F. && (1.25)
\end{aligned}
$$

for the k-level facility location problem, when k is part of the input to the problem. The multi-level facility location algorithm of Aardal, Chudak and Shmoys is able to use some of the improvements made by Chudak and Shmoys to the algorithm of Shmoys, Tardos and Aardal to achieve their approximation constant.

Since the multi-level facility location problem is a generalization of the facility location problem, we can conclude that the multi-level version is *MAX-SNP*-hard, and that we cannot hope to achieve an approximation constant less than ≈ 1.463, unless $P = NP$.

The algorithm of Aardal, Chudak and Shmoys uses the same clustering technique used in the algorithms of Chudak and Shmoys and Shmoys, Tardos and Aardal. After solving the linear programming relaxation of $(\mathrm{IP}_{\mathrm{MLFL}})$, special clients are chosen to be cluster centers. The cluster centers randomly select a suitable path

$$
\begin{aligned}
(\mathrm{DP_{MLFL}}) \qquad \max \quad & \sum_{j \in D} v_j && && (1.26)\\
\text{s.t.} \quad & v_j - \sum_{i \in p} w_{ij} \le c_{pj}, && \text{for each } p \in P, j \in D, && (1.27)\\
& \sum_{j \in D} w_{ij} \le f_i, && \text{for each } i \in F, && (1.28)\\
& w_{ij} \ge 0, && \text{for each } i \in F, j \in D. && (1.29)
\end{aligned}
$$

without considering the clients in their clusters, and finally each client in a cluster is assigned to the path of its cluster center.

The linear programming relaxation of ($\mathrm{IP_{MLFL}}$) is constructed by replacing (1.18) and (1.19) with non-negativity constraints.

On first inspection, it is not clear that ($\mathrm{LP_{MLFL}}$) can be solved in polynomial time for arbitrary k, as it has an exponential number of variables. However, since the stronger constraint (1.23) is used instead of (1.20), there are polynomially many constraints. Consequently, the dual linear program ($\mathrm{DP_{MLFL}}$) of ($\mathrm{LP_{MLFL}}$) has polynomially many variables.

The dual linear program can be solved in polynomial time using the ellipsoid algorithm (see [Kar84] and [GLS88]), assuming a polynomial-time separation algorithm exists. A separation algorithm for ($\mathrm{DP_{MLFL}}$), given (v, w), must determine if (v, w) is feasible for ($\mathrm{DP_{MLFL}}$). If not, it must prove infeasibility by exhibiting a violated constraint. The separation algorithm can check each of the constraints (1.28) in polynomial time, but checking (1.27) for a violated constraint is more involved.

The constraints (1.27) can be rewritten as

$$v_j \leq \sum_{e \in p} c_e + \sum_{i \in p} w_{ij}, \qquad \text{for each } p \in P, j \in D. \tag{1.30}$$

We define modified edge costs $c'_e = c_e + w_{ij}$ for $e = i'i$. Then the right hand side of (1.30) is the path length of p with respect to these modified edge costs. Therefore, the separation algorithm must check, for client j, whether or not there is a path p from j to some top level node of F_1 that costs less than v_j with respect to c'. If the shortest path with respect to c' has cost at least v_j, then no such path exists. On the other hand, if the shortest path costs less than v_j, then (1.30) is violated for this path-client pair. Therefore, we can check all the constraints (1.27) by running a shortest path algorithm for each client.

Note that this is not only of theoretical interest. This separation algorithm is not only finding violated constraints for the dual, but is also finding primal variables that must be added in order to prove optimality. Therefore, we can use this separation algorithm as part of a column generation scheme, in order to solve (LP_{MLFL}) in practice. Chapter 4 discusses the solution of (LP_{MLFL}), in practice, in greater depth.

Having solved (DP_{MLFL}) to obtain an optimal dual solution (v, w) and therefore constructing an optimal primal solution (x, y) in the process, the rounding algorithm of Aardal, Chudak and Shmoys [ACS99] must round this fractional solution to an integer one. The set of paths used by a client in the optimal primal solution is

Algorithm 1.7 FINDCLUSTERS

$S \leftarrow \emptyset, J \leftarrow D.$ {Initialize.}
while $J \neq \emptyset$ **do**
  Select $j \in J$ with minimum $C_j + v_j$. {j is a new cluster center.}
  $S \leftarrow S \cup \{j\}$. {Add j to set of cluster centers.}
  $J \leftarrow J \setminus \{j' \in J : I(j') \cap I(j) \neq \emptyset\}$.
  {Remove clients using any facility in j's cluster from J.}

Algorithm 1.8 LPROUNDING

for all $j \in S$ **do** {j is a cluster center.}
  Choose one $p \in P(j)$ with probability x_{pj}, $I \leftarrow I \cup p$.
  {Randomly choose one facility from j's cluster.}
Assign each client to its shortest path in G_I.

denoted $P(j) = \{p : x_{pj} > 0\}$ and the fractional service cost of a client's fractional assignment is, as before, denoted $C_j = \sum_{p \in P} c_{pj} x_{pj}$. We also define the set of facilities used by a client in the optimal primal solution $I(j) = \{i : i \in p, p \in P(j)\}$. As with the single-level case, the rounding algorithm consists of a clustering step, Algorithm 1.7: FINDCLUSTERS, and a random selection step, Algorithm 1.8: LPROUNDING.

Notice that the rounding step does not round in facilities that are not in a cluster, as the single-level version does. In fact, this detail is primarily responsible for the much more respectable approximation constant achieved by the algorithm of Chudak and Shmoys [CS] for the facility location problem.

As before, this algorithm's solution is constructed around a cluster center. Each cluster center chooses a path to which each client in that cluster is assigned. This results in solutions with clearly non-optimal structure. The paths selected by the cluster centers must be disjoint, due to the definition of $I(j)$ and Step 5 of FIND-CLUSTERS, and make no use of the multi-level structure of the problem. In Chapter 2 we show that a 3-approximation is the best we can do if we restrict ourselves to these types of solutions.

Chapter 2

Reductions from Multi-Level Facility Location to Facility Location

The gap between the performance guarantees of Section 1.2 for the facility location problem and Section 1.3 for the multi-level facility location problem suggest that it might be possible to improve the performance guarantee for the multi-level case. A polynomial-time approximation-preserving reduction from the multi-level facility location problem to a facility location problem would make it possible to apply the results of Section 1.2 and obtain these superior performance guarantees for the multi-level problem.

Thus far, a reduction that is both polynomial-time (or equivalently, generates a single-level problem of polynomial-size) *and* approximation-preserving has been elusive. This chapter will describe two reductions. The first reduction is approximation-preserving but results in exponentially-sized single-level problems, while the second reduction produces polynomially-sized single-level problems but does not preserve the approximation constant.

2.1 Steiner Branchings

The multi-level facility location problem can be thought of as a minimum-cost flow problem with respect to the service costs, plus fixed charges. The constraint (1.16) in x is just the path formulation of the flow polytope for one unit of flow from each client to one of the facilities of F_1. In fact, if we temporarily add to G a sink facility i_0 and edges ii_0 with service cost 0, for each $i \in F_1$, then the x variables can be thought of as a minimum-cost unit flow from each client to the sink i_0. Our intuition about minimum-cost flow suggests that once two flows meet at some facility, they should then follow the same path to the sink.

Definition 1. *A feasible solution to the multi-level facility location problem satisfies the* branching property *if the clients' paths define a subgraph of G with out-degree at most 1.*

Theorem 2. *Any feasible solution to the multi-level facility location problem can be transformed, in polynomial time, into a solution that costs no more and also satisfies the branching property.*

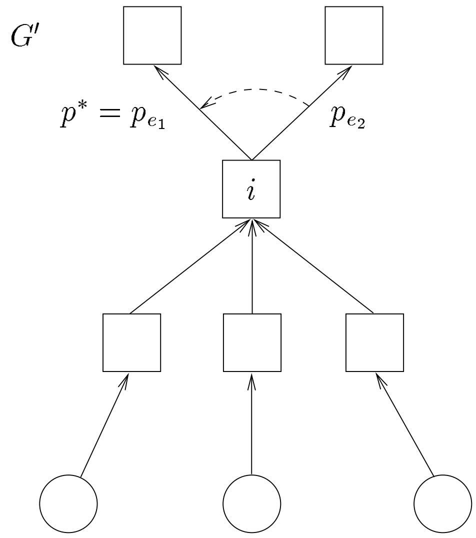

Figure 2.1: Branching property transformation operation.

Proof. Let G' be the subgraph of G defined by the clients' paths. Since exactly one path leaves each client, the out-degree of the clients in G' is 1. Therefore, any node of G' that violates the branching property must be a facility. Let $i \in F_l$ be a facility of G' with out-degree strictly greater than 1 where the level l is minimal over all such possibilities. If there are no facilities with out-degree greater than 1, then the solution satisfies the branching property.

Denote the out-edges of i in G' by E'_i. Each of these edges are in G' because they were used by at least one path-client pair. Let D_e be the set of clients using the edge $e \in E'_i$. Note that all of the clients in D_e use the same sub-path once facility i and edge e is reached, by our choice of i. Denote this sub-path for the clients of D_e by p_e.

We will select the subpath p^* from $\{p_e : e \in E_i'\}$ for which the service cost of the partial path c_{p_e} is minimum. Note that p^* need not be unique. We form a new solution as follows. For each client j of $D_e, p_e \neq p^*$, the path p used by j is replaced by the path p' that follows p from j to i and then switches to subpath p^*.

We note that the service cost of the new solution is no more than the original service cost, by the selection of p^*. Since each of the facilities on p^* was already open, the facility cost of the new solution is no more than the original facility cost. In fact, if we remove from the new solution any facility vertices that are not used, the facility cost of the new solution may in fact be less than the original facility cost.

We now have one fewer facility with out-degree greater than 1. The process needs to be repeated at most $|F|$ times before there are no longer any such facilities, and our solution satisfies the branching property. $\square$

In fact, subgraphs satisfying the branching property has been extensively studied in the mathematical programming literature, see Nemhauser and Wolsey [NW88].

Definition 2. *A rooted spanning tree of a directed graph is an* in-branching *if*

a. it provides a directed path from each node to the root, and

b. all nodes except the root have out-degree 1.

Given variables y_e on the edges of a directed graph $G = (V, E)$, the branching polytope describes the convex hull of r-rooted in-branchings:

$$\sum_{e \in \text{out}(C)} y_e \geq 1, \quad \text{for each } C \subseteq V \setminus \{r\}, \tag{2.1}$$

$$\sum_{e \in \text{out}(i)} y_e \leq 1, \quad \text{for each } i \in V \setminus \{r\}, \tag{2.2}$$

where $\text{out}(C)$ is the set of edges leaving the set C. Constraint (2.1) enforces the first condition of the definition by requiring that there is at least one edge crossing any cut that separates the nodes of G from the root. Notice that this is merely the cut formulation of the single-sink flow polytope, with the root as the sink. Constraint (2.2) requires the out-degree of any node except the root be at most 1, which, in combination with constraint (2.1), enforces the second condition of the definition. The branching polytope is special because it is the convex hull of the intersection of two *matriods*, special combinatorial structures with nice properties. We will not elaborate on these properties in detail here, see Nemhauser and Wolsey for this. Nevertheless, it should be clear how close this branching polytope comes to modeling the problem we are actually interested in, the multi-level facility location problem.

Unfortunately, the edges used by a feasible solution to the multi-level facility location problem *do not* form an in-branching because they do not form a directed path from *every* node of G to the root i_0. Instead, we require only that there is a directed path from the clients to the root.

Definition 3. *A rooted tree of a directed graph is a* Steiner in-branching *on the nodes of S if*

a. *it defines a directed path from each node of S to the root, and*

b. *all nodes have out-degree at most 1.*

A Steiner in-branching on the nodes of S, rooted at r, is said to span $S \cup \{r\}$.

The distinction here between Steiner in-branchings and (spanning) in-branchings is subtle but crucial. In particular, the set of Steiner in-branchings cannot be represented as the intersection of two matriods, as the set of in-branchings can; and it is *NP*-hard to optimize over the set of Steiner in-branchings, unlike optimizing over the set of in-branchings. In addition, various powerful algorithmic tools, such as the Edmonds decomposition, see [Edm72] and [GM95], that can be used in the context of in-branchings are not available in the Steiner in-branching context.

Corollary 3. *There exists an optimal solution to the multi-level facility location problem for which the subgraph of G defined by the clients' paths consists of disjoint Steiner in-branchings rooted at facilities of F_1.*

Proof. Given any optimal solution to the multi-level facility location problem, Theorem 2 shows that we can construct an optimal solution that satisfies the branching property. Define I_1 to be the facilities of F_1 used in this optimal solution. For each $i \in I_1$, consider the subgraph B_i of G defined by the clients' paths that terminate at i.

Since our optimal solution satisfies the branching property, B_i is a Steiner in-branching on the vertices of G, rooted at i, spanning i and the clients whose paths terminate at i.

Finally, suppose that two distinct Steiner in-branching subgraphs $B_i, B_{i'}$ for $i, i' \in I_1$ share some facility vertex of G. Then by the branching property, B_i and $B_{i'}$ must be rooted at the same vertex and hence cannot be distinct. □

An alternate, more direct proof of this result provides useful intuition.

Proof. We consider a variant of G, denoted $\widetilde{G}$, in which all the edges of G are reversed and a source facility i_0, plus edges $i_0 i$ with service cost 0, for each $i \in F_1$, are added.

Given the set I of facility locations used in our optimal solution, we have already established that we can construct an equivalent optimal solution by assigning each client j to the shortest path, with respect to service cost, in G_I from j to a top level node $i_1 \in F_1$. We now observe that these shortest paths, reversed, are also shortest paths in $\widetilde{G}_I$ from the top level nodes F_1 to each client. Therefore, we can construct a further equivalent optimal solution by computing the shortest path tree rooted at i_0 in $\widetilde{G}_{I \cup \{i_0\}}$ and discarding paths that do not terminate at clients. The structure of this shortest path tree, when mapped back to G, results in an optimal solution that satisfies the branching property. □

Corollary 3 shows that we can reduce the multi-level facility location problem to the facility location problem by constructing virtual "super-facilities," each of which represents a Steiner in-branching of G. Of course, there are exponentially many of these virtual super-facilities. Nevertheless, we will show that the reduction is approximation-preserving and provides a way to apply the algorithms of Section 1.2 to our multi-level instances.

Given an instance $\mathcal{I} = (F_1, \ldots, F_k, D, f, c)$ of the multi-level facility location problem we construct an instance $\widehat{\mathcal{I}} = (\widehat{F}, \widehat{D}, \widehat{f}, \widehat{c})$ of the facility location problem. For every Steiner in-branching B of G that spans exactly one node of F_1 and some non-empty subset of F_k, we create a virtual super-facility $\hat{\imath}_B \in \widehat{F}$. For each client $j \in D$, we create a client $\hat{\jmath} \in \widehat{D}$. For each $\hat{\imath} = \hat{\imath}_B \in \widehat{F}$ we define $\widehat{f}_{\hat{\imath}} = \sum_{i \in B} f_i$, and we construct $\widehat{c}_{\hat{\imath}\hat{\jmath}}$ for each $\hat{\imath} \in \widehat{F}$ and $\hat{\jmath} \in \widehat{D}$ as follows. Assigning $\hat{\jmath}$ to $\hat{\imath} = \hat{\imath}_B$ incurs the same service cost as assigning j to the cheapest path represented by the Steiner in-branching B. Consider the subpaths $p_1, \ldots, p_m$ from each vertex $i_1, \ldots, i_m$ of $F_k \cap B$ to the root of B, each of which represents one possible routing for the client j. We choose the subpath $p_{(B,j)}$ and hence the facility $i_{(B,j)}$ of F_k that results in the minimum service cost for the assignment of $\hat{\jmath}$ to $\hat{\imath}$. Formally,

$$\widehat{c}_{\hat{\imath}\hat{\jmath}} \quad = \quad c_{i_{(B,j)}j} + c_{p_{(B,j)}} \quad = \quad \min\{c_{i_l j} + c_{p_l} : l = 1, \ldots, m\}. \tag{2.3}$$

Theorem 4. *Let $\mathcal{I}$ be an instance of the multi-level facility location problem and let $\widehat{\mathcal{I}}$ be the Steiner in-branching reduction single-level problem derived from $\mathcal{I}$. Given any solution to $\widehat{\mathcal{I}}$, we can construct a solution for $\mathcal{I}$ of equal or lesser cost.*

Proof. For each virtual super-facility $\hat{\imath}_B$ used in the solution for $\widehat{\mathcal{I}}$, we include the facilities $i \in B$ in our solution for $\mathcal{I}$. Each client is assigned the path in G that corresponds to its assignment in the solution for $\widehat{\mathcal{I}}$. The service cost component of our solution for $\mathcal{I}$ is the same as for $\widehat{\mathcal{I}}$. If a facility is contained in more than one

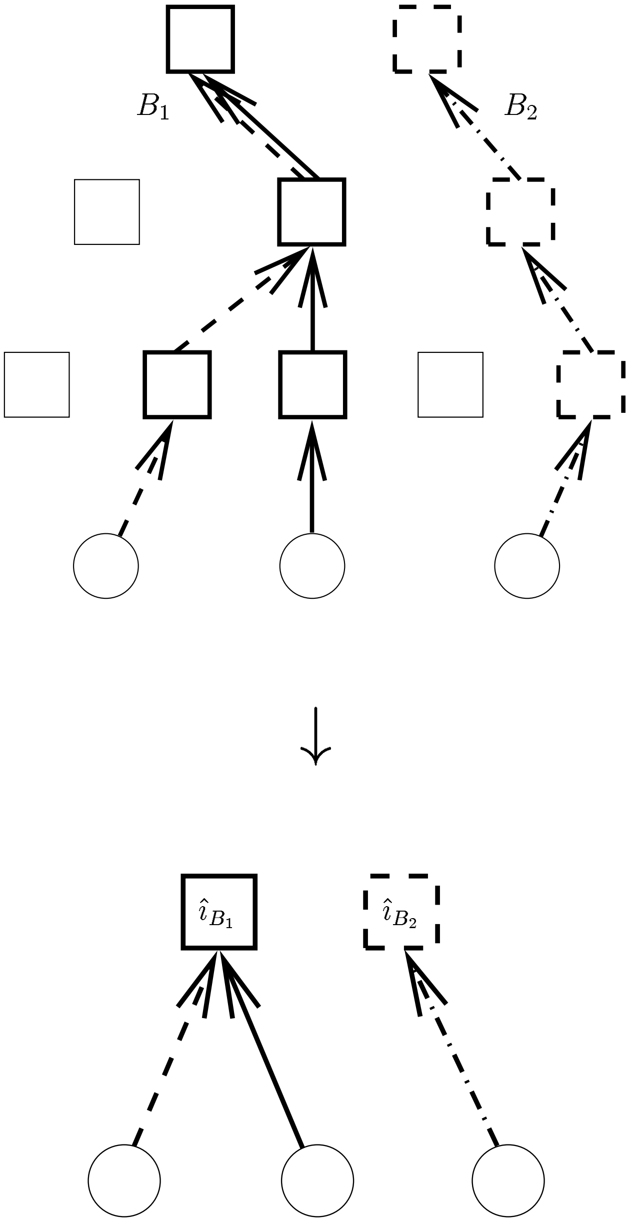

Figure 2.2: Steiner in-branching reduction.

virtual super-facility used in the solution for $\widehat{\mathcal{I}}$, then the solution for $\widehat{\mathcal{I}}$ pays for it more than once. Therefore, the facility cost of our solution for $\mathcal{I}$ is no more than the facility cost of the solution for $\widehat{\mathcal{I}}$. $\square$

Theorem 5. *Given an instance $\mathcal{I}$ of the multi-level facility location problem and an instance $\widehat{\mathcal{I}}$ of the facility location problem resulting from an application of the Steiner in-branching reduction, the optimal value of $\mathcal{I}$ equals the optimal value of $\widehat{\mathcal{I}}$.*

Proof. Let z and $\widehat{z}$ be the optimal values of $\mathcal{I}$ and $\widehat{\mathcal{I}}$ respectively.

Corollary 3 establishes the existence of an optimal solution for $\mathcal{I}$ that consists of disjoint Steiner in-branchings. We can construct a feasible solution for $\widehat{\mathcal{I}}$ consisting of a virtual super-facility for each of the Steiner in-branchings in the optimal solution for $\mathcal{I}$. This feasible solution for $\widehat{\mathcal{I}}$ costs the same as our optimal solution for $\mathcal{I}$, so we must have $\widehat{z} \leq z$. An application of Theorem 4 shows that $\widehat{z} \geq z$. Therefore $z = \widehat{z}$. $\square$

Corollary 6. *The Steiner in-branching reduction of the multi-level facility location problem to a facility location problem is approximation-preserving.*

For this approximation-preserving reduction to be useful, we need to demonstrate that the resulting instances are appropriate for the facility location problem algorithms of Section 1.2. We must show that the Steiner in-branching analogue of Lemma 1 holds.

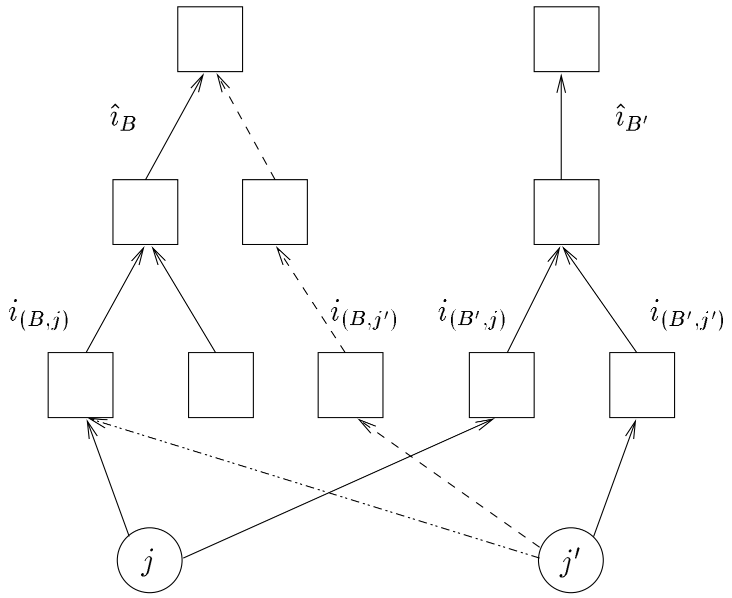

Figure 2.3: Picture proof of Lemma 7

Lemma 7. *Let $\widehat{\mathcal{I}} = (\widehat{F}, \widehat{D}, \widehat{f}, \widehat{c})$ be the Steiner in-branching reduction of the metric instance $\mathcal{I} = (F_1, \ldots, F_k, D, f, c)$ of the multi-level facility location problem. Then*

$$\widehat{c}_{\hat{\imath}\hat{\jmath}'} \leq \widehat{c}_{\hat{\imath}\hat{\jmath}} + \widehat{c}_{\hat{\imath}'\hat{\jmath}} + \widehat{c}_{\hat{\imath}'\hat{\jmath}'}$$

for all $\hat{\imath}, \hat{\imath}' \in \widehat{F}$ and $\hat{\jmath}, \hat{\jmath}' \in \widehat{D}$.

Proof. Given the virtual super-facilities $\hat{\imath} = \hat{\imath}_B \in \widehat{F}$ and $\hat{\imath}' = \hat{\imath}_{B'} \in \widehat{F}$ we decompose the service costs.

$$\widehat{c}_{\hat{\imath}\hat{\jmath}'} = c_{i_{(B,j')}j'} + c_{p_{(B,j')}} \tag{2.4}$$

$$\leq c_{i_{(B,j)}j'} + c_{p_{(B,j)}} \tag{2.5}$$

where (2.5) follows from the definition of service costs (2.3).

Expanding the right hand side:

$$\widehat{c}_{\hat{i}\hat{j}} + \widehat{c}_{\hat{i}'\hat{j}} + \widehat{c}_{\hat{i}'\hat{j}'} = c_{i_{(B,j)}j} + c_{p_{(B,j)}} + c_{i_{(B',j)}j} + c_{p_{(B',j)}} + c_{i_{(B',j')}j'} + c_{p_{(B',j')}} \tag{2.6}$$

Notice that the second term in inequality (2.5) and equation (2.6) are the same. Therefore, we need to show

$$c_{i_{(B,j)}j'} \leq c_{i_{(B,j)}j} + c_{i_{(B',j)}j} + c_{p_{(B',j)}} + c_{p_{(B',j')}} + c_{i_{(B',j')}j'} \tag{2.7}$$

A careful reading of these quantities and Figure 2.3, make it clear that the quantities on the right hand side of (2.7) trace out a path beginning at j' and ending at $i_{(B,j)}$. Therefore, since we have a metric multi-level facility location problem instance, it follows that (2.7) holds. □

We have established the following.

Theorem 8. *The Steiner in-branching reduction is a* ≈ 1.728*-approximation technique for the* k*-level facility location problem.*

Proof. We achieve this approximation guarantee by running the approximation algorithm of Charikar and Guha [CG99] on the Steiner in-branching reduction single-level problem. Corollary 6 and Lemma 7 guarantee this approximation constant is valid for the resulting solution to the k-level facility location problem. □

We emphasize here that to achieve this approximation guarantee, we must run each of the algorithms of Section 1.2 on an exponentially-sized problem. How big does the facility location problem reduction get? Let $n_l = |F_l|$, for $l = 1, \ldots, k$. The number of Steiner in-branchings is bounded above by $n_1 2^{n_2 \dots n_k}$. This gets big very, very quickly!

From an implementation standpoint, for all but very small problems, the explicit enumeration approach is completely impractical. For such small problems, it is likely we would be better off blindly applying an integer programming solution package such as CPLEX to ($\mathrm{IP}_{\mathrm{MLFL}}$) to get an optimal solution, than applying Theorem 8. Further, each of the algorithms of Section 1.2 relies on being able to iterate through, count, or operate on the facilities of an instance in polynomial time, something that we clearly cannot do. However, this reduction does leave open the possibility of the discovery of an adaptation of the algorithms of Section 1.2 to handle instances resulting from this reduction in polynomial time.

2.2 Paths

The exponential size of the Steiner in-branching reductions and the resulting intractability of the resultant single-level instances leads us to look for alternatives. Motivated by the work of Aardal, Chudak and Shmoys, see Section 1.3 and [ACS99], we consider solutions to the multi-level facility location problem that consist of paths. Let $n_l = |F_l|$ for $l = 1, \dots, k$ and $n = |F|$. If we consider the parameter k of the k-level facility location problem to be a constant, the number of paths is bounded by $n_1 \dots n_k \leq n^k$, a polynomial in the number of facilities n. Therefore, for constant k, this reduction results in polynomially-sized single-level instances. The

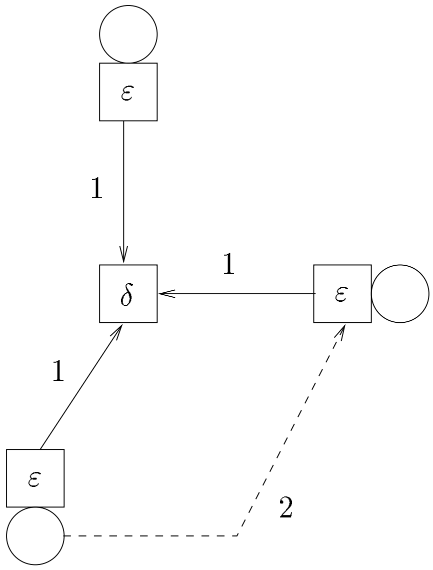

Figure 2.4: Disjoint-path solution lower bound instance.

following theorem shows that we do not loose too much by restricting our attention to these solutions.

Theorem 9. *The cost of the optimal disjoint-path solution of the multi-level facility location problem is at most 3 times the cost of the optimal solution.*

Proof. The 3-approximation algorithm of Aardal, Chudak and Shmoys produces solutions consisting of disjoint paths. The theorem follows. □

In fact, this bound is asymptotically tight in the worst case for disjoint-path solutions. Consider the metric multi-level facility location problem instance in Figure 2.4. This instance is constructed in $\mathbb{R}^d$, Figure 2.4 shows the case where $d = 3$. The instance consists of a level 1 facility of cost δ at the origin, and level 2 facilities of cost ε at the head of each of the standard basis vectors of $\mathbb{R}^d$. Co-located with each level 2 facility is a client. We use the l_1 norm to determine the service cost between two points. The optimal solution to the problem selects all the facility locations, while the disjoint-path solution can only install a single path. The optimal solution to this problem has cost $\delta + d\varepsilon + d$, while the optimal disjoint-path solution has cost $\delta + \varepsilon + 1 + (d-1)3$. As $d \longrightarrow \infty$ the ratio of these solution values goes to 3, for small ε and δ. We point out, however, that this instance says little about the best we can do in the worst case if we may select overlapping paths, paying for the overlapped facilities multiple times, as the path reduction, to follow, allows.

We define the path reduction as with the Steiner in-branching reduction. Given an instance $\mathcal{I} = (F_1, \ldots, F_k, D, f, c)$ of the multi-level facility location problem we construct an instance $\widehat{\mathcal{I}} = (\widehat{F}, \widehat{D}, \widehat{f}, \widehat{c})$ of the facility location problem. For every path p of G from a node of F_k to a node of F_1, we create a virtual facility $\hat{\imath}_p \in \widehat{F}$, and for each client $j \in D$, we create a client $\hat{\jmath} \in \widehat{D}$. For each $\hat{\imath} = \hat{\imath}_p \in \widehat{F}$ we define $\widehat{f}_{\hat{\imath}} = \sum_{i \in p} f_i$. Lastly, we define $\widehat{c}_{\hat{\imath}\hat{\jmath}}$ for each $\hat{\imath} = \hat{\imath}_p \in \widehat{F}$ and $\hat{\jmath} \in \widehat{D}$. Assigning $\hat{\jmath}$ to $\hat{\imath}$ incurs the same service cost as assigning j to the path p in G. Denote the facility of F_k used by p as i_p. Then $\widehat{c}_{\hat{\imath}\hat{\jmath}} = c_{i_p j} + c_p$.

In order to use this path reduction, we need to show that feasible solutions for the path reduction facility location problem cost no more when converted into solutions for the multi-level facility location problem. The service costs do not change when a solution is converted back, and the facility costs can only decrease as a particular facility might have been paid for more than once by the path reduction facility location problem. We also need to show that the algorithms of Section 1.2 can be applied to these reductions. In fact, Lemma 7 applies here too, since the paths of G are certainly members of the set of all Steiner in-branchings of G.

Theorem 10. *For each* $k = 1, 2, \ldots$, *there exists a* ≈ 5.184*-approximation algorithm for the* k*-level facility location problem.*

Proof. We apply the path reduction to obtain a single-level facility location problem. The ≈ 1.728-approximation algorithm of Charikar and Guha [CG99] finds a solution of cost at most ≈ 1.728 times the optimal path solution cost. Theorem 9 shows that this optimal path solution cost, bounded above by the optimal disjoint-path solution cost, is at most 3 times the optimal cost for the k-level facility location problem. Therefore, the result of the Charikar-Guha algorithm applied to the single-level path reduction is a ≈ 5.184-approximation algorithm for the k-level facility location problem. $\square$

Note that while this approximation constant is not very impressive, and it certainly does not compare well with the approximation constant of 3 of Aardal, Chudak and Shmoys [ACS99], in practice, this approach produces excellent solutions that are typically much closer to optimal than the solutions of the algorithm of Aardal, Chudak and Shmoys. Chapter 4 demonstrates this in some detail.

As with the Steiner in-branching reduction, the path reduction facility location problem can be formed by explicitly enumerating all paths of G. However, imposing a path structure on the reduction makes it possible to run the rounding algorithm of Chudak and Shmoys [CS] and the randomized local improvement algorithm of Charikar and Guha [CG99] without explicitly enumerating all the paths of G. Instead, in each case, we can add paths as they are encountered or required by the algorithm. This significantly increases the practicality of this path reduction approach. Chapter 4 demonstrates the effectiveness of this technique.

2.3 Promising Theoretical Directions

These reductions for the multi-level facility location problem and the associated observations about the structure of its solutions suggest that adaptations of the single-level algorithms of Section 1.2 might be possible. We briefly discuss each of the techniques applied in Section 1.2 to the single-level problem, and sketch the obstacles that need to be overcome in order to adapt these algorithms to the multi-level case.

Randomized Rounding

Our initial efforts with respect to the multi-level facility location problem revolved around the optimal fractional solution to ($\mathrm{LP_{MLFL}}$). Given a decomposition of the optimal fractional solution into a convex combination of Steiner in-branchings, we could then run the linear program solution rounding rounding procedure of Chudak and Shmoys [CS], with its ≈ 1.736 approximation constant. A variety of approaches were tried to achieve this decomposition, from the assumption of a completeness property to the imposition of additional variables and constraints on ($\mathrm{LP_{MLFL}}$). To date, none of these techniques have proven successful.

Local Improvement

The local improvement algorithm of Charikar and Guha [CG99] seems ripe for adaption to the multi-level case. Unfortunately, while there are plenty of plausible ways in which heuristics for incrementally improving a multi-level solution might be constructed, we could not construct an improvement heuristic to which we could apply the averaging argument used by Charikar and Guha in the proof of their guarantee for the single-level case. The other issue to resolve with respect to this algorithmic technique is how to pick one of the possible local improvement steps uniformly at random from the exponentially many possibilities. This can be done for paths, but it is not clear how to achieve this for Steiner in-branchings. Even if sampling from such a large distribution is possible in polynomial time via a technique like Monte Carlo Markov chains, with the uniform sampling used by Charikar and Guha, the size of the set being sampled from appears in the running time bound of the algorithm.

Primal-Dual Schema

As with the other techniques applied to the single-level facility location problem, it seems like it should be possible to adapt the intuition of the primal-dual algorithm design technique to the multi-level case. With respect to the Steiner in-branchings of the graph G, we notice that the service costs of a clients' assignment to all the Steiner in-branchings that contain a particular path from that client are simultaneously paid for. Unfortunately, it does not seem possible to collapse all of the accounting required for each Steiner in-branching to construct a polynomial-time algorithm, despite this. As demonstrated in Chapter 4, the first phase of the primal-dual algorithm can be adapted relatively straightforwardly for paths. An analogue of the path clusters introduced by Aardal, Chudak and Shmoys [ACS99] seemed like a promising avenue. Unfortunately, the penalty in service cost for those clients that were reassigned could not be suitably bounded.

Chapter 3

Multi-Level p-Median

The p-median problem, like the facility location problem, concerns the optimal location of the facilities needed to provide some service to a set of clients. In the facility location case, our solution must factor in the cost of the facilities used. The p-median problem instead imposes a hard constraint on the number of facilities that may be used. The p-median problem requires that no more than p facilities are used.

Usually, the p-median problem uses the language of clustering algorithms, discussing points rather than clients, centers rather than facilities, distance rather than service cost. To maintain consistency with the rest of this thesis, we will continue to use the language of facility location to talk about the p-median problem, even when this is a little awkward.

Given its similarity to the facility location problem, it is natural to ask what can be said about a multi-level version of the problem. We will show that we can apply an idea similar to that used by Aardal, Chudak and Shmoys [ACS99], discussed in Section 1.3, to achieve a constant approximation algorithm for a particular multi-level generalization of the p-median problem.

We will first set the stage by talking about the single-level version of the problem, and then discuss the multi-level case.

3.1 p-Median

The objective of the p-median problem is to assign all clients to some nearby facility while only using at most p facilities. Owing to its origin as a clustering problem, the p-median problem considers each client to be a potential facility location. We will use the terminology non-bipartite to describe this style of input, as necessary.

As with the facility location problem, we shall be concerned with the uncapacitated metric version of the p-median problem, unless otherwise indicated.

Problem Statement

The uncapacitated metric p-median problem input consists of a set of clients D and metric service costs c_{ij} for each $i, j \in D$. Feasible solutions assign each client to a facility-client, using at most p clients as facilities. The set of clients used as facilities will be denoted I throughout. A feasible solution is charged the total of the service costs of the clients' assignments. An optimal solution to the p-median problem is a feasible solution of minimum cost.

$$
\begin{aligned}
(\mathrm{IP}^1_{p\mathrm{M}}) \qquad \min \quad & \sum_{i \in D} \sum_{j \in D} c_{ij} x_{ij} && && (3.1)\\
\text{s.t.} \quad & \sum_{i \in D} x_{ij} = 1, && \text{for each } j \in D, && (3.2)\\
& x_{ij} \le y_i, && \text{for each } i \in D, j \in D, && (3.3)\\
& \sum_{i \in D} y_i \le p, && && (3.4)\\
& x_{ij} \in \{0,1\}, && \text{for each } i \in D, j \in D, && (3.5)\\
& y_i \in \{0,1\}, && \text{for each } i \in D. && (3.6)
\end{aligned}
$$

As with the facility location problem, the set of clients used as facilities completely describe a feasible solution, since each client may be assigned to the closest facility of I.

Integer Programming Formulation

The uncapacitated p-median problem is modeled by the integer programming formulation $(\mathrm{IP}^1_{p\mathrm{M}})$. The 0-1 variables y_i represent whether or not a client i is used as a facility whereas the 0-1 variables x_{ij} represent whether or not a client j has been assigned to facility i. This formulation has the same constraints as $(\mathrm{IP}_{\mathrm{FL}})$ plus the constraint (3.4), which requires that at most p clients are used as facilities. Note too that there is no term in the objective involving facility costs.

Previous Work

As with the metric uncapacitated facility location problem, much progress has been made on the metric uncapacitated p-median problem recently. The first constant approximation algorithm for the problem, due to Charikar, Guha, Tardos and Shmoys [CGTS99] appeared less than two years ago. The algorithm uses the fractional optimal solution to the linear programming relaxation of $(\mathrm{IP}^1_{p\mathrm{M}})$ to construct a good solution for the p-median problem. This result is the basis for our multi-level p-median result and as such, will be described in detail shortly.

The other recent results for the metric p-median problem all make use of the following observation. Suppose we consider the p-median problem as an instance of the facility location problem in which clients and potential facility locations are co-located and all facilities are given identical facility costs f. Then there exists an f^* such that an optimal solution to the facility location problem instance is also an optimal solution to the p-median problem instance. Notice that as f increases, the optimal solution to the facility location problem instance will open fewer and fewer facilities.

Other than the result of Charikar, Guha, Tardos and Shmoys, all of the constant approximation algorithms for the p-median problem rely on this connection with the facility location problem. Typically, these algorithms use binary search to look for a uniform facility cost just above and just below f^*, and then combine the two solutions corresponding to these estimates of f^* by taking advantage of additional structure in the two solutions.

The primal-dual algorithm of Jain and Vazirani [JV99] works in this way. This algorithm uses the 3-approximation algorithm for the facility location problem algorithm outlined in Section 1.2 as a subroutine to get a 6-approximation for the p-median problem. Charikar and Guha [CG99] use the primal-dual algorithm of Jain and Vazirani and a tighter analysis to construct a 4-approximation for the p-median problem.

CGTS Algorithm

We will be showing how to extend the algorithm of Charikar, Guha, Tardos and Shmoys [CGTS99] from the p-median problem to a multi-level generalization of the p-median problem. As a basis for this extension, we first provide a detailed outline of the CGTS algorithm.

The CGTS algorithm actually solves a generalization of the p-median problem, in which each client may specify a demand d_j. The typical p-median problem is equivalent to setting $d_j = 1$ for all $j \in D$. The demand d_j can be thought of as the number of clients at a given client location. The p-median problem with demands is modeled by the integer program ($\mathrm{IP}_{p\mathrm{M}}$).

The linear programming relaxation ($\mathrm{LP}_{p\mathrm{M}}$) of ($\mathrm{IP}_{p\mathrm{M}}$) is formed by replacing the equations (3.11) and (3.12) with non-negativity constraints.

The algorithm produces a sequence of solutions to a series of problems, achieving its constant approximation by only losing a small additive or multiplicative factor at each stage.

$$
\begin{aligned}
(\mathrm{IP}_{p\mathrm{M}}) \qquad & \min \quad \sum_{i \in D} \sum_{j \in D} d_j c_{ij} x_{ij} && && (3.7)\\
& \text{s.t.} \quad \sum_{i \in D} x_{ij} = 1, && \text{for each } j \in D, && (3.8)\\
& \qquad x_{ij} \le y_i, && \text{for each } i \in D, j \in D, && (3.9)\\
& \qquad \sum_{i \in D} y_i \le p, && && (3.10)\\
& \qquad x_{ij} \in \{0,1\}, && \text{for each } i \in D, j \in D, && (3.11)\\
& \qquad y_i \in \{0,1\}, && \text{for each } i \in D. && (3.12)
\end{aligned}
$$

$$
\begin{aligned}
(\mathrm{LP}_{p\mathrm{M}}) \qquad & \min \quad \sum_{i \in D} \sum_{j \in D} d_j c_{ij} x_{ij} && && (3.13)\\
& \text{s.t.} \quad \sum_{i \in D} x_{ij} = 1, && \text{for each } j \in D, && (3.14)\\
& \qquad x_{ij} \le y_i, && \text{for each } i \in D, j \in D, && (3.15)\\
& \qquad \sum_{i \in D} y_i \le p, && && (3.16)\\
& \qquad x_{ij} \ge 0, && \text{for each } i \in D, j \in D, && (3.17)\\
& \qquad y_i \ge 0, && \text{for each } i \in D. && (3.18)
\end{aligned}
$$

1. **Solve** $(\mathrm{LP}_{p\mathrm{M}})$

   We solve the linear program $(\mathrm{LP}_{p\mathrm{M}})$ to get an optimal (fractional) solution.

2. **Concentrate demand**

   We use the linear programming solution to construct a new demand profile that chooses super-clients to assume the demand of and act on behalf of nearby clients.

3. **Relocate facilities**

   We construct a new solution with the property that only super-clients are used as facilities.

4. **Find good solution for special subproblem**

   We now have a special p-median subproblem. We show it is possible to obtain a good integer solution for the special subproblem.

5. **Redistribute demand**

   Finally, each client in the neighborhood of a super-client is assigned to the super-client's facility.

We now present each of these steps in more detail.

1. Solve $(\mathrm{LP}_{p\mathrm{M}})$

We solve the linear program $(\mathrm{LP}_{p\mathrm{M}})$ to get an optimal (fractional) solution (x, y) with value z. Since the optimal value of the linear program $(\mathrm{LP}_{p\mathrm{M}})$ is a lower bound

Algorithm 3.1 CONCENTRATEDEMAND

1: Label clients s.t. $C_1 \leq C_2 \leq \ldots \leq C_{|D|}$

2: $d'_j \leftarrow d_j$ for each $j \in D$. {Initialize d'.}

3: **for** $j = 1$ to $|D|$ **do**

4: **if** there exists $i < j$ s.t. $d'_i > 0$ and $c_{ij} \leq 4C_j$ **then** {Too close?}

5: $d'_i \leftarrow d'_i + d'_j$; $d'_j \leftarrow 0$. {Move j's demand to i.}

on the optimal integer solution, we will bound the value of subsequent solutions in terms of this lower bound.

2. Concentrate Demand

In order to simplify the structure of the input, we choose super-clients to represent all the clients in some neighborhood. To achieve this simplification, we require the selected super-clients to be sufficiently far from one another. On the other hand, the algorithm must ensure that the clients represented by a super-client are not too far from it. We will use the cost of the fractional assignment of a unit of demand at client j, defined by

$$C_j = \sum_{i \in D} c_{ij} x_{ij}, \qquad \text{for each } j \in D, \tag{3.19}$$

in order to choose the super-clients.

We will construct a new demand profile d' that satisfies

$$d'_i > 0 \text{ and } d'_j > 0 \quad \Rightarrow \quad c_{ij} > 4 \max\{C_i, C_j\}, \qquad \text{for each } i, j \in D. \tag{3.20}$$

The new demand profile is constructed in one pass through the clients by Algorithm 3.1: ConcentrateDemand. ConcentrateDemand examines each client j in turn, in order of increasing C_j, either leaving the client's demand in place, effectively designating the client a super-client; or, if the client is too close to some previously designated super-client, moving its demand to that super-client. We see immediately that the new demand satisfies the condition imposed by (3.20). Consider the linear program $(\mathrm{LP}_{p\mathrm{M}})$ with the new input data d', denoted $(\mathrm{LP}'_{p\mathrm{M}})$. Notice that our existing fractional solution to the original linear program $(\mathrm{LP}_{p\mathrm{M}})$ is still feasible for $(\mathrm{LP}'_{p\mathrm{M}})$. Denote the value of our existing solution (x, y) with respect to $(\mathrm{LP}'_{p\mathrm{M}})$ by z'. Lemma 11 shows that the cost of our existing solution with respect to the new linear program is no more than it was with respect to the original.

Lemma 11. *Let z be the cost of the optimal solution (x, y) of $(\mathrm{LP}_{p\mathrm{M}})$, and let z' be the cost of (x, y) with respect to $(\mathrm{LP}'_{p\mathrm{M}})$. Then $z' \le z$.*

Proof.

$$z' = \sum_{i \in D} \sum_{j \in D} d'_j c_{ij} x_{ij} = \sum_{j \in D} d'_j C_j \le \sum_{j \in D} d_j C_j = \sum_{i \in D} \sum_{j \in D} d_j c_{ij} x_{ij} = z \tag{3.21}$$

where the inequality follows from the fact that in the algorithm ConcentrateDemand, demand is reassigned to clients with smaller C_j. □

The clients with $d'_j > 0$ will be our super-clients. Each client whose demand was concentrated at super-client j is considered to be in the neighborhood of j.

Algorithm 3.2 RelocateFacilities

1: $\tilde{y}_i \leftarrow 0$ for each $i \in D$. {Initialize $\tilde{y}$.}

2: $\tilde{x}_{ij} \leftarrow 0$ for each $i, j \in D$. {Initialize $\tilde{x}$.}

3: **for each** $j \in D$ **do**

4: **if** $y_j > 0$ **then** {Open facility?}

5: Let i be a super-client of minimum distance from j.

6: $\tilde{y}_i \leftarrow \min\{\tilde{y}_i + y_j, 1\}$. {Move open facility from j to i.}

7: $\tilde{x}_{ij'} \leftarrow \tilde{x}_{ij'} + x_{jj'}$ for each $j' \in D$.

3. Relocate Facilities

In this step, we construct a new solution that opens facility locations only at super-clients. Algorithm 3.2: RelocateFacilities makes a single pass through the facility locations relocating open facilities at super-clients. RelocateFacilities examines each facility in turn. If the facility is (fractionally) open we relocate it to its closest super-client. For open facilities located at super-clients, their closest super-client is at their current location, so these open facilities are "relocated" to their current location.

We note first that the solution $(\tilde{x}, \tilde{y})$ computed by RelocateFacilities is feasible for $(\mathrm{LP}'_{p\mathrm{M}})$. In addition, we have achieved our goal of ensuring that the only open facility locations are super-clients. Denote the value of the solution $(\tilde{x}, \tilde{y})$ with respect to $(\mathrm{LP}'_{p\mathrm{M}})$ by $\tilde{z}$. We must now evaluate the cost of this new solution.

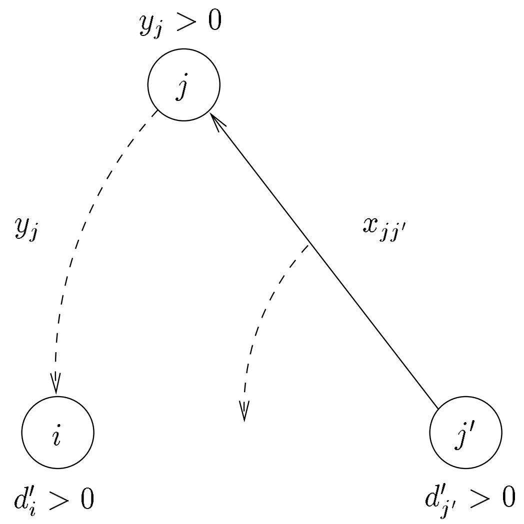

Figure 3.1: Facility relocation operation.

Lemma 12. *Let z' be the cost, with respect to* $(\mathrm{LP}'_{p\mathrm{M}})$*, of the optimal solution* (x, y) *of* $(\mathrm{LP}_{p\mathrm{M}})$*; and let $\tilde{z}$ be the cost of $(\tilde{x}, \tilde{y})$ with respect to* $(\mathrm{LP}'_{p\mathrm{M}})$*. Then $\tilde{z} \leq 2z'$.*

Proof. Consider client j, and let i be a super-client closest to j. We observe that in order to evaluate $\tilde{z}$, we must only bound the cost $c_{ij'}$ in terms of $c_{jj'}$ for those j' with $d'_{j'} > 0$. The first service cost is the new cost of a fractional reassignment of the demand at j', the second service cost is the old cost. Since we consider only metric instances, we can bound $c_{ij'}$ using the triangle inequality, $c_{ij'} \leq c_{ij} + c_{jj'}$.

Now, since j' has $d'_{j'} > 0$ it is a super-client, but we know that the super-client i is, by definition, a super-client closest to j. Therefore, $c_{ij} \leq c_{jj'}$ and $c_{ij'} \leq 2c_{jj'}$. □

Corollary 13. *Let z be the cost of the optimal solution* (x, y) *of* $(\mathrm{LP}_{p\mathrm{M}})$*, and $\tilde{z}$ be the cost of $(\tilde{x}, \tilde{y})$ with respect to* $(\mathrm{LP}'_{p\mathrm{M}})$*. Then $\tilde{z} \leq 2z$.*

We have shown that this new solution $(\tilde{x}, \tilde{y})$ does not cost too much more than (x, y) with respect to $(\mathrm{LP}'_{p\mathrm{M}})$ or $(\mathrm{LP}_{p\mathrm{M}})$.

Furthermore, this new solution has very special additional structure. To show this, we will use the filtering technique of Lin and Vitter [LV92a], which relies on the following fact about weighted averages.

Lemma 14. *Given $s_1, \ldots, s_n \in \mathbb{R}^+$ and $\alpha_1, \ldots, \alpha_n \in \mathbb{R}^+$ such that $\sum_{l=1}^{n} \alpha_l = 1$, we define $\bar{s} = \sum_{l=1}^{n} s_l \alpha_l$, the weighted average of $\{s_1, \ldots, s_n\}$ with respect to $\{\alpha_1, \ldots, \alpha_n\}$. Then*

$$\sum_{l: s_l > \gamma \bar{s}} \alpha_l < \frac{1}{\gamma}.$$

Notice that C_j is really a weighted average of the $\{c_{ij}, i \in D\}$ with respect to $\{x_{ij}, i \in D\}$. This observation leads to the following result about the structure of $(\tilde{x}, \tilde{y})$.

Lemma 15. *$\tilde{y}_i \geq \frac{1}{2}$ for all super-clients i.*

Proof. Using Lemma 14 we can write

$$\sum_{j: c_{ij} > 2C_i} x_{ij} < \frac{1}{2}. \tag{3.22}$$

Therefore,

$$\sum_{j: c_{ij} \leq 2C_i} y_j \geq \sum_{j: c_{ij} \leq 2C_i} x_{ij} \geq \frac{1}{2} \tag{3.23}$$

where the first inequality follows from inequality (3.15) of $(\mathrm{LP}_{p\mathrm{M}})$ and the second from inequality (3.22).

Finally, we notice that due to Stage 2 in which we concentrated demand and ensured that all other super-clients were more than $4C_i$ from super-client i, all open facility locations j such that $c_{ij} \leq 2C_i$ are closer to i than any other super-client. Therefore, we can bound $\tilde{y}_i$ by the expression on the left hand side of (3.23), and we have the required bound. $\square$

4. Find good solution from special subproblem

At its core, the CGTS algorithm actually finds a good integer solution to a special p-median subproblem. The special subproblem formulation $(\mathrm{IP_{SS}})$ allows only the super-clients S, the clients $j \in D$ for which $d'_j > 0$, to act as facilities.

Charikar, Guha, Tardos and Shmoys find good solutions for this problem in a roundabout manner. Instead of considering solutions to the linear programming relaxation of $(\mathrm{IP_{SS}})$, they consider solutions to the strengthened linear program $(\mathrm{LP'_{SS}})$.

The following theorem is at the heart of the CGTS algorithm.

Theorem 16. **[CGTS99]** *Given a* $\{0, \frac{1}{2}, 1\}$ *solution to* $(\mathrm{LP'_{SS}})$ *of cost* z^**, the polynomial-time algorithm due to Charikar, Guha, Tardos and Shmoys finds an integer solution to* $(\mathrm{IP_{SS}})$ *of cost at most* $\frac{4}{3}z^*$.

Furthermore, Charikar, Guha, Tardos and Shmoys show that there is an optimal $\{0, \frac{1}{2}, 1\}$ solution to $(\mathrm{LP'_{SS}})$ that can be found using a straightforward greedy algorithm. Therefore, we can conclude the following.

Corollary 17. *Given any feasible solution to* $(\mathrm{LP'_{SS}})$ *of cost* $\tilde{z}$*, there exists an integer solution to* $(\mathrm{IP_{SS}})$*, computable in polynomial time, of cost at most* $\frac{4}{3}\tilde{z}$.

$$
\begin{aligned}
(\mathrm{IP}_{\mathrm{SS}}) \qquad & \min \quad \sum_{i \in D} \sum_{j \in D} d'_j c_{ij} x_{ij} && && (3.24)\\
& \text{s.t.} \quad \sum_{i \in D} x_{ij} = 1, && \text{for each } j \in D, && (3.25)\\
& \qquad x_{ij} \leq y_i, && \text{for each } i \in D, j \in D, && (3.26)\\
& \qquad \sum_{i \in D} y_i \leq p, && && (3.27)\\
& \qquad y_i = 0 && \text{for each } i \in D \setminus S, && (3.28)\\
& \qquad x_{ij} \in \{0,1\}, && \text{for each } i \in D, j \in D, && (3.29)\\
& \qquad y_i \in \{0,1\}, && \text{for each } i \in D. && (3.30)
\end{aligned}
$$

$$
\begin{aligned}
(\mathrm{LP}'_{\mathrm{SS}}) \qquad & \min \quad \sum_{i \in D} \sum_{j \in D} d'_j c_{ij} x_{ij} && && (3.31)\\
& \text{s.t.} \quad \sum_{i \in D} x_{ij} = 1, && \text{for each } j \in D, && (3.32)\\
& \qquad x_{ij} \leq y_i, && \text{for each } i \in D, j \in D, && (3.33)\\
& \qquad \sum_{i \in D} y_i \leq p, && && (3.34)\\
& \qquad y_i \geq \frac{1}{2} && \text{for each } i \in S, && (3.35)\\
& \qquad y_i = 0 && \text{for each } i \in D \setminus S, && (3.36)\\
& \qquad x_{ij} \in \{0,1\}, && \text{for each } i \in D, j \in D, && (3.37)\\
& \qquad y_i \in \{0,1\}, && \text{for each } i \in D. && (3.38)
\end{aligned}
$$

We omit the details of the proofs of these results. They form the heart of this single-level result, but are used as a black box by the result for the multi-level case to follow.

A moment's thought about the solution $(\tilde{x}, \tilde{y})$ to $(\mathrm{LP}'_{p\mathrm{M}})$ and its structure, reveals that it is, in fact, a feasible solution to $(\mathrm{LP}'_{\mathrm{SS}})$. Therefore, we can apply Corollary 17 to get the integer solution $(\hat{x}, \hat{y})$. We note that this is a valid integer solution not only for $(\mathrm{IP}_{\mathrm{SS}})$, but also for $(\mathrm{LP}'_{p\mathrm{M}})$. We denote the value of the integer solution $(\hat{x}, \hat{y})$ with respect to $(\mathrm{LP}'_{p\mathrm{M}})$ by $\hat{z}$.

Corollary 18. *Let z be the cost of the optimal solution (x, y) of $(\mathrm{LP}_{p\mathrm{M}})$, and let $\hat{z}$ be the cost of the integer solution $(\hat{x}, \hat{y})$ with respect to $(\mathrm{LP}'_{p\mathrm{M}})$. Then $\hat{z} \leq \frac{8}{3} z$.*

Proof. Let $\tilde{z}$ be the cost of $(\tilde{x}, \tilde{y})$ with respect to $(\mathrm{LP}'_{p\mathrm{M}})$. Then

$$\hat{z} \;\leq\; \frac{4}{3}\tilde{z} \;\leq\; \frac{8}{3} z. \tag{3.39}$$

□

5. Redistribute demand

We form our final integer solution $(\bar{x}, \bar{y})$ for $(\mathrm{LP}_{p\mathrm{M}})$ from $(\hat{x}, \hat{y})$ by considering each super-client to have acted on behalf of the clients in its neighborhood. Each client in the neighborhood of a super-client is assigned to the same facility location as that super-client. We assume that the assignment that is appropriate for a super-client is appropriate for all of the clients in its neighborhood. We must evaluate the cost of the solution $(\bar{x}, \bar{y})$ with respect to $(\mathrm{LP}_{p\mathrm{M}})$, denoted $\bar{z}$, to ensure that this is the case.

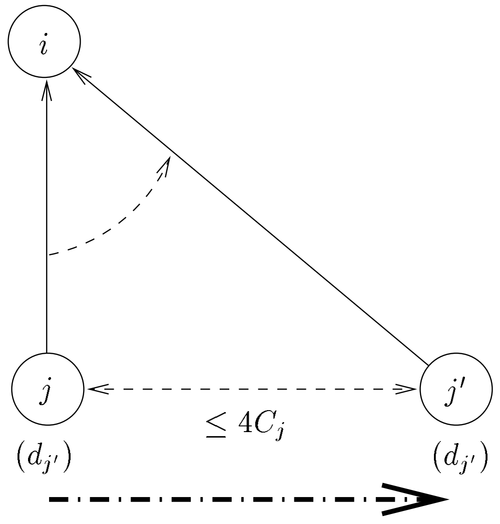

Figure 3.2: Demand redistribution operation.

Lemma 19. *Let z be the cost of the optimal solution (x, y) of* $(\mathrm{LP}_{p\mathrm{M}})$*, let $\hat{z}$ be the cost of the integer solution $(\hat{x}, \hat{y})$ with respect to* $(\mathrm{LP}'_{p\mathrm{M}})$*, and let $\bar{z}$ be the cost of the integer solution $(\bar{x}, \bar{y})$ with respect to* $(\mathrm{LP}_{p\mathrm{M}})$*. Then $\bar{z} \leq \hat{z} + 4z$.*

Proof. Let j be some super-client, assigned to the facility location i in the solution $(\bar{x}, \bar{y})$ and let j' be some client in the neighborhood of j. With respect to $(\mathrm{LP}'_{p\mathrm{M}})$, the demand originally at j' contributes $d_{j'}c_{ij}$ to $\hat{z}$. With respect to $(\mathrm{LP}_{p\mathrm{M}})$, however, the demand at j' contributes $d_{j'}c_{ij'}$ to $\bar{z}$. Using the triangle inequality once more, we have $c_{ij'} \leq c_{ij} + c_{jj'}$.

Our demand concentration step allows us to bound $c_{jj'}$ by $4C_{j'}$. Therefore

$$d_{j'}c_{ij'} \leq d_{j'}c_{ij} + d_{j'}c_{jj'} \leq d_{j'}c_{ij} + 4d_{j'}C_{j'}. \tag{3.40}$$

Summing over all $j' \in D$, we obtain the desired bound. $\square$

Assembling all the pieces, we have shown the following.

Theorem 20. *The algorithm of Charikar, Guha, Tardos and Shmoys for the p-median problem is a* $6\frac{2}{3}$*-approximation.*

Proof. We apply Corollary 18 to Lemma 19 to obtain the required bound. □

3.2 Multi-Level p-Median

The multi-level p-median problem presented here is just one possible way to generalize the p-median problem to the multi-level case. Nevertheless, considering this generalization demonstrates another application of the algorithm design technique for multi-level problems used by Aardal, Chudak and Shmoys [ACS99] to apply the results for single-level facility location to the multi-level case. We consider a multi-level setting in which we may use at most p top level facilities and we must consider facility costs. We will show that an extension of the CGTS p-median algorithm is sufficient to achieve a constant approximation for this problem. We note that this particular generalization of the p-median problem to the multi-level case is the same as the multi-level facility location problem with the addition side constraint that at most p top level facilities may be selected.

Problem Statement

The metric uncapacitated k-level p-median problem input consists a set of facilities F partitioned into k levels, denoted $F_1, \ldots, F_k$, a set of clients D, facility costs f_i

for each $i \in F$ and metric service costs c_{ij} for each $i, j \in F \cup D$. Feasible solutions must assign each client to a sequence of k facilities, one from each level $F_k, \ldots, F_1$, in order. The set of facilities used by the clients, denoted I throughout, may contain at most p facilities of F_1. A feasible solution is charged the sum of the facility costs of the facilities used, plus the total of the service costs of the clients' assignments. Each client's service cost is the sum of the service cost from itself to the first facility of its sequence, plus the service cost between successive facilities of its sequence. An optimal solution to the metric uncapacitated k-level p-median problem is a feasible solution of minimum cost.

As with the multi-level facility location problem, it will be convenient to formally define the graph G implicit in this problem statement. G is a directed graph defined on the nodes $F \cup D$. The edges E of G are of the form $i_l i_{l-1}$ for $i_l \in F_l, i_{l-1} \in F_{l-1}, i = 2, \ldots, k$ or $j i_k$ for $j \in D, i_k \in F_k$. For $ij \in E$, we define the service cost of the edge ij to be c_{ij}. The assignment of a client $j \in D$ to a valid sequence of facilities can now be represented by the assignment of j to a path p in G from j to one of the top level facilities of F_1. We will switch between these two characterizations of valid sequences of facilities at will.

Again, we can fully describe a feasible solution to the multi-level p-median problem by a set of used facilities I. Given a set I of facilities containing at least one facility from each level, each client j is assigned to the shortest path, with respect to service cost, from j to one of the top level facilities of F_1 in G_I.

$$
\begin{aligned}
(\mathrm{IP}_{\mathrm{ML}p\mathrm{M}}) \qquad \min \quad & \sum_{i \in F} f_i y_i + \sum_{p \in P} \sum_{j \in D} d_j c_{pj} x_{pj} && && (3.41)\\
\text{s.t.} \quad & \sum_{p \in P} x_{pj} = 1, && \text{for each } j \in D, && (3.42)\\
& \sum_{p \ni i} x_{pj} \le y_i, && \text{for each } i \in F, j \in D, && (3.43)\\
& \sum_{i \in F_1} y_i \le p, && && (3.44)\\
& x_{pj} \in \{0,1\}, && \text{for each } p \in P, j \in D, && (3.45)\\
& y_i \in \{0,1\}, && \text{for each } i \in F. && (3.46)
\end{aligned}
$$

Integer Programming Formulation

The uncapacitated k-level p-median problem is modeled by the integer programming formulation $(\mathrm{IP}_{\mathrm{ML}p\mathrm{M}})$. We define the set of all valid sequences of facilities by $P = F_k \times \cdots \times F_1$, and the service cost of client j's assignment to sequence $p = (i_k, \ldots, i_1)$ by $c_{pj} = c_{ji_k} + c_{i_k i_{k-1}} + \cdots + c_{i_2 i_1}$. The 0-1 variable y_i represents whether or not facility i is used while the 0-1 variable x_{pj} represents whether or not client j is assigned to path p. As with the p-median problem we permit each client j to specify a demand d_j. The linear programming relaxation $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$ of $(\mathrm{IP}_{\mathrm{ML}p\mathrm{M}})$ is formed by replacing (3.45) and (3.46) with non-negativity constraints.

$$
\begin{aligned}
(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}}) \qquad \min \quad & \sum_{i \in F} f_i y_i + \sum_{p \in P} \sum_{j \in D} d_j c_{pj} x_{pj} && && (3.47) \\
\text{s.t.} \quad & \sum_{p \in P} x_{pj} = 1, && \text{for each } j \in D, && (3.48) \\
& \sum_{p \ni i} x_{pj} \leq y_i, && \text{for each } i \in F, j \in D, && (3.49) \\
& \sum_{i \in F_1} y_i \leq p, && && (3.50) \\
& x_{pj} \geq 0, && \text{for each } p \in P, j \in D, && (3.51) \\
& y_i \geq 0, && \text{for each } i \in F. && (3.52)
\end{aligned}
$$

A Constant Approximation Algorithm

We create a constant approximation algorithm for the multi-level p-median problem by extending the CGTS algorithm in three ways. First, we must account for the bipartite nature of the problem input; second, we must bound the facility costs; and third, we must deal with facility paths rather than facility locations. Much the same algorithm structure works.

The algorithm consists of the following steps:

1. Solve $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$

We solve the linear program $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$ to get an optimal (fractional) solution.

2. Concentrate demand

We use the linear programming solution to construct a new demand profile that chooses super-clients to assume the demand of and act on behalf of nearby clients.

3. Relocate facilities

We construct a new solution for the special subproblem with the property that only super-clients are used as facilities.

4. Find good solution for special subproblem

We now have a special p-median subproblem. We obtain a good integer solution for the special subproblem.

5. Select facilities

Each super-client chooses a path facility from its neighborhood.

6. Redistribute demand

Each client in the neighborhood of a super-client is assigned to the path facility of that super-client.

We now present each of these stages in more detail.

1. Solve $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$

We solve the linear program $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$ to get an optimal (fractional) solution (x, y) of value z. Since the optimal value of the linear program $(\mathrm{LP}_{p\mathrm{M}})$ is a lower bound on the optimal integer solution, we will bound the value of subsequent solutions in terms of z. As necessary, we will denote the facility cost and service cost portions of

z by z_F and z_C respectively. We will use these subscripts consistently in the following discussion to denote the facility and service cost portions of various solutions.

Note that solving the linear program ($\text{LP}_{\text{ML}p\text{M}}$) is considerably more difficult than solving ($\text{LP}_{p\text{M}}$), given the exponential number of variables of ($\text{LP}_{\text{ML}p\text{M}}$). Nevertheless, since there are only polynomially-many constraints and there is a polynomial-time separation oracle for the exponentially-many constraints of the dual, we can solve the dual linear program in polynomial time using the ellipsoid method, see [GLS88] and [Kar84]. We omit the details here, but Section 1.3 and Chapter 4 discuss this at some length.

2. Concentrate demand

As in the p-median algorithm, in order to simplify the structure of the input, we choose super-clients to represent all of the clients in some neighborhood. We define

$$C_j = \sum_{p \in P} c_{pj} x_{pj}, \qquad \text{for each } j \in D, \tag{3.53}$$

which represents the cost of the fractional assignment of one client at location j.

As before, we will construct a new demand profile d' that satisfies

$$d'_j > 0 \text{ and } d'_{j'} > 0 \quad \Rightarrow \quad c_{jj'} > 4 \max\{C_j, C_{j'}\}, \qquad \text{for each } j, j' \in D. \tag{3.54}$$

The new demand profile is constructed in a single pass though the clients by Algorithm 3.3: ConcentrateDemand. ConcentrateDemand examines each client j in turn, in order of increasing C_j, either leaving the client's demand in place, effectively designating the client a super-client; or, if the client is too close to some previously designated super-client, moving its demand to that super-client.

Algorithm 3.3 CONCENTRATEDEMAND

1: Label clients s.t. $C_1 \leq C_2 \leq \ldots \leq C_{|D|}$

2: $d'_j \leftarrow d_j$ for each $j \in D$. {Initialize d'.}

3: **for** $j = 1$ to $|D|$ **do**

4: **if** there exists $j_0 < j$ s.t. $d'_{j_0} > 0$ and $c_{jj_0} \leq 4C_j$ **then** {Too close?}

5: $d'_{j_0} \leftarrow d'_{j_0} + d'_j$; $d'_j \leftarrow 0$. {Move j's demand to j_0.}

We see immediately that the new demand satisfies the condition imposed by (3.54). Consider the linear program ($\mathrm{LP}_{\mathrm{ML}p\mathrm{M}}$) with the new input data d'. Notice that our existing fractional solution to the original linear program ($\mathrm{LP}_{\mathrm{ML}p\mathrm{M}}$) is still feasible for the new linear program, which we will denote ($\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}}$). Denote the value of our solution (x, y) with respect to this new linear program by z'. Lemma 21 shows that the service cost of our existing solution with respect to the new linear program is no more than it was with respect to the original.

Lemma 21. *Let z_C be the service cost of the optimal solution (x, y) of* ($\mathrm{LP}_{\mathrm{ML}p\mathrm{M}}$), *and let z'_C be the service cost of (x, y) with respect to* ($\mathrm{LP}'_{p\mathrm{M}}$). *Then $z'_C \leq z_C$.*

Proof.

$$\sum_{p \in P} \sum_{j \in D} d'_j c_{pj} x_{pj} = \sum_{j \in D} d'_j C_j \tag{3.55}$$

$$\leq \sum_{j \in D} d_j C_j \tag{3.56}$$

$$= \sum_{p \in P} \sum_{j \in D} d_j c_{pj} x_{pj} \tag{3.57}$$

Algorithm 3.4 RELOCATEFACILITIES

1: $\tilde{y}_i \leftarrow 0$ for each $i \in D$. {Initialize $\tilde{y}$.}

2: $\tilde{x}_{ij} \leftarrow 0$ for each $i, j \in D$. {Initialize $\tilde{x}$.}

3: **for each** $i \in F_1$ **do**

4: **if** $y_i > 0$ **then** {Top-level facility open?}

5: Define $P(i)$ to be the set of paths ending at i.

6: Let j_0 be a super-client which minimizes $c_{p_0 j_0}$, for $p_0 \in P(i)$.

7: $\tilde{y}_{j_0} \leftarrow \min\{\tilde{y}_{j_0} + y_i, 1\}$. {Collapse open path facilities ending at i to j_0.}

8: $\tilde{x}_{j_0 j} \leftarrow \tilde{x}_{j_0 j} + x_{pj}$ for each $p \in P(i), j \in D$.

---

where the inequality follows from the fact that in Algorithm 3.3: CONCENTRATE-DEMAND, demand is reassigned to clients with smaller C_j. □

We note that the facility cost is not affected by the change in demand profile.

The clients with $d'_j > 0$ will be our super-clients S. Each client whose demand was concentrated at super-client j is considered to be in the neighborhood of j.

3. Relocate Facilities

In this step, we construct a new solution for the special p-median subproblem linear relaxation of (IP_{SS}). Algorithm 3.4: RELOCATEFACILITIES makes a single pass through the facility locations F_1 to construct the solution $(\tilde{x}, \tilde{y})$ for the linear relaxation of (IP_{SS}). Each open facility $i \in F_1$ represents the set of used paths that terminate at i. All of these paths are collapsed to a super-client closest to i.

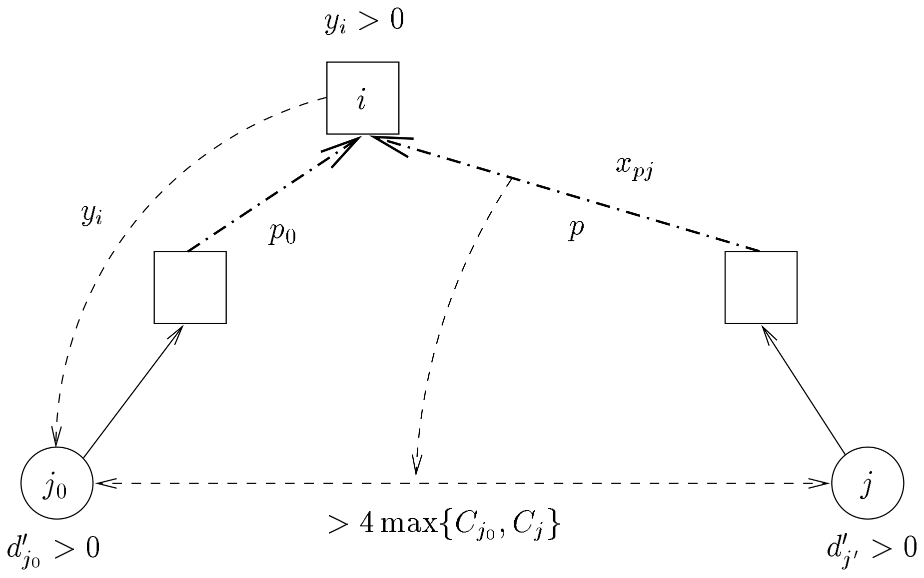

Figure 3.3: Facility relocation operation.

As before, we claim that $(\tilde{x}, \tilde{y})$ is feasible not only for the linear relaxation of $(\mathrm{IP_{SS}})$ but also for $(\mathrm{LP'_{SS}})$.

Lemma 22. *$\tilde{y}_j \geq \frac{1}{2}$ for all super-clients j.*

Proof. Let j_0 be a super-client, and define $P(j) = \{p : c_{pj} \leq 2C_j\}$. For $p \in P(j)$, define $i_1(p)$ to be the last node of p, a facility location of F_1. We claim that all $i_1(p), p \in P(j_0)$ are relocated to j_0. Suppose not, and some $i_1(p), p \in P(j_0)$ is relocated to some other super-client j via minimum cost path p. The condition in Algorithm 3.4: RelocateFacilities that decides relocation guarantees that $c_{pj} \leq c_{p_0 j_0}$, but from the triangle inequality we have that $c_{jj_0} \leq c_{pj} + c_{p_0 j_0}$, since both p_0 and p end at $i_1(p_0)$. Therefore, $c_{jj_0} \leq 2c_{p_0 j_0} \leq 4C_{j_0}$, which contradicts our definition of super-clients from Stage 2. Therefore, all $i_1(p), p \in P(j_0)$ are relocated to j_0. Define $I_1(j) = \{i_1(p) : p \in P(j)\}$.

We can now use Lemma 14 to show the required inequality.

$$\tilde{y}_{j_0} \geq \sum_{i \in I_1(j_0)} y_i \geq \sum_{p \in P(j_0)} x_{pj_0} \geq \frac{1}{2} \tag{3.58}$$

where the second inequality follows from inequality (3.49) of ($\mathrm{LP}_{\mathrm{ML}p\mathrm{M}}$). □

We now bound the cost of $(\tilde{x}, \tilde{y})$ with respect to ($\mathrm{LP}_{\mathrm{SS}}$), denoted $\tilde{z}$, in terms of the service cost paid by (x, y) in ($\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}}$).

Lemma 23. *Let z'_C be the service cost of the optimal solution (x, y) of ($\mathrm{LP}_{\mathrm{ML}p\mathrm{M}}$) with respect to ($\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}}$), and let $\tilde{z}$ be the cost of the solution $(\tilde{x}, \tilde{y})$ with respect to ($\mathrm{LP}_{\mathrm{SS}}$). Then $\tilde{z} \leq 2z'_C$.*

Proof. Consider client j with $d'_j > 0$. For p such that $x_{pj} > 0$, let j_0 be the super-client to which $i = i_1(p)$ is relocated, and let p_0 be the path that minimizes $\min_{\{p' : i_1(p') = i\}} c_{p'j_0}$. To show the required bound, we need only show $c_{j_0 j} \leq 2c_{pj}$.

As in the proof of Lemma 22, we use the triangle inequality. Since $i_1(p) = i_1(p_0) = i$, we can apply the triangle inequality to $c_{j_0 j}$:

$$c_{j_0 j} \leq c_{p_0 j_0} + c_{pj} \leq 2c_{pj} \tag{3.59}$$

where the final inequality follows from the fact that i was relocated to j_0, which implies that $c_{p_0 j_0} \leq c_{pj}$. □

4. Find good solution for special subproblem

We have shown that the solution $(\tilde{x}, \tilde{y})$ is feasible for ($\mathrm{LP}'_{\mathrm{SS}}$), with $\tilde{z} \leq 2z'_C$. We now use Theorem 16 and Corollary 17 as a black box in order to construct an integer solution $(\hat{x}, \hat{y})$ for ($\mathrm{IP}_{\mathrm{SS}}$) with cost $\hat{z} \leq \frac{4}{3}\tilde{z}$.

Algorithm 3.5 SELECTFACILITIES

$\check{y}_i \leftarrow 0$ for each $i \in F$. {Initialize $\check{y}$.}
$\check{x}_{pj} \leftarrow 0$ for each $p \in P, j \in D$. {Initialize $\check{x}$.}
for each super-client j such that $\hat{y}_j = 1$ **do**
  Let $p_j \in P_\alpha(j)$ have minimum total facility cost. {Pick the cheapest.}
  $\check{y}_i \leftarrow \hat{y}_j$ for each facility i of p_j. {Open the facilities of p_j.}
  $\check{x}_{p_j j'} \leftarrow \hat{x}_{jj'}$ for each $j' \in D$. {Assign clients of j to p_j.}

Lemma 24. *Let z'_C be the service cost of the optimal solution (x, y) of* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$ *with respect to* $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$*, and let $\hat{z}$ be the cost of the integer solution $(\hat{x}, \hat{y})$ with respect to* $(\mathrm{IP}_{\mathrm{SS}})$*. Then $\hat{z} \leq \frac{8}{3} z'_C$.*

Proof. The result follows from Theorem 16 and Corollary 17 and the application of Lemma 23. □

5. Facility selection

We must convert the solution produced in Stage 4 into a solution appropriate for the multi-level problem and the linear program $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$. For each super-client j with $\hat{y}_j = 1$ we choose the path $p \in P$ of minimum facility cost in some neighborhood of j. For some $1 < \alpha \leq 2$ define $P_\alpha(j) = \{p \in P : c_{pj} \leq \alpha C_j\}$ and let p_j be the facility from $P_\alpha(j)$ of minimum total facility cost. The converted solution $(\check{x}, \check{y})$ is constructed from $(\hat{x}, \hat{y})$ by Algorithm 3.5: SELECTFACILITIES.

We now evaluate the cost of this solution with respect to $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$, which we denote $\check{z}$. First, the service costs.

Lemma 25. *Let z_C be the service cost of the optimal solution (x, y) of* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, *let $\hat{z}$ be the cost of the integer solution $(\hat{x}, \hat{y})$ with respect to* $(\mathrm{IP}_{\mathrm{SS}})$, *and let $\check{z}_C$ be the service cost of the integer solution $(\check{x}, \check{y})$ with respect to* $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$. *Then $\check{z}_C \leq \hat{z} + \alpha z_C$.*

Proof. Consider super-client j' assigned to super-client j by $(\hat{x}, \hat{y})$, we have $\hat{x}_{jj'} = 1$. Let p_j be the path selected by super-client j in Algorithm 3.5: SELECTFACILITIES. Then in the solution $(\check{x}, \check{y})$ we must have $\check{x}_{p_j j'} = 1$, which contributes $c_{p_j j'}$ to $\check{z}_C$. Denote the facilities of path p_j as $(i_k, \ldots, i_1)$.

$$
\begin{aligned}
c_{p_j j'} &= c_{j' i_k} + c_{i_k i_{k-1}} + \cdots + c_{i_2 i_1} && (3.60) \\
&\leq c_{j' j} + c_{j i_k} + c_{i_k i_{k-1}} + \cdots + c_{i_2 i_1} && (3.61) \\
&= c_{j' j} + c_{p_j j} && (3.62) \\
&\leq c_{j' j} + \alpha C_j. && (3.63)
\end{aligned}
$$

where (3.61) follows by virtue of the triangle inequality, and (3.63) follows from the definition of $P_\alpha(j)$, of which p_j is a member. Multiplying (3.63) by $d_{j'}$ and summing over all j' we obtain the desired bound. □

Corollary 26. *Let z_C be the service cost of the optimal solution (x, y) of* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, *and let $\check{z}_C$ be the service cost of the integer solution $(\check{x}, \check{y})$ with respect to* $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$. *Then $\check{z}_C \leq \left(\frac{8}{3} + \alpha\right) z_C$*

Proof. Application of Lemma 24 and Lemma 21 to Lemma 25 produces the required bound. □

Evaluating the facility costs incurred by $(\check{x}, \check{y})$ is a little more involved. First, we must guarantee that no facility is shared between paths considered for selection by two super-clients.

Lemma 27. *Let j_1 and j_2 be super-clients. Then the paths of $P_\alpha(j_1)$ and $P_\alpha(j_2)$ do not share any facility locations.*

Proof. Suppose j_1 and j_2 are super-clients such that some facility location i is shared by $p_1 \in P_\alpha(j_1)$ and $p_2 \in P_\alpha(j_2)$. Then by the triangle inequality,

$$c_{j_1 j_2} \leq c_{p_1 j_1} + c_{p_2 j_2} \leq \alpha C_{j_1} + \alpha C_{j_2} \leq 2\alpha \max\{C_{j_1}, C_{j_2}\}. \tag{3.64}$$

Since j_1 and j_2 are both super-clients, $c_{j_1 j_2} > 4 \max\{C_{j_1}, C_{j_2}\}$, so $\alpha > 2$, which contradicts the definition of α. □

To bound the facility cost incurred by $(\check{x}, \check{y})$, we again use the filtering technique of Lemma 14.

Lemma 28. *Let z_F be the facility cost of the optimal solution (x, y) of $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, and let $\check{z}_F$ be the facility cost of the integer solution $(\check{x}, \check{y})$ with respect to $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$. Then $\check{z}_F \leq \left(\frac{\alpha}{\alpha-1}\right) z_F$.*

Proof. For super-client j, by Lemma 14, we have

$$\sum_{p \in P_\alpha(j)} x_{pj} \geq \left(1 - \frac{1}{\alpha}\right). \tag{3.65}$$

Therefore, since p_j is the facility of $P_\alpha(j)$ of minimum total facility cost,

$$\begin{aligned}
\left(1-\frac{1}{\alpha}\right)\sum_{i\in p_j} f_i &\leq \left(\sum_{p\in P_\alpha(j)} x_{pj}\right)\sum_{i\in p_j} f_i && (3.66)\\
&\leq \sum_{p\in P_\alpha(j)}\sum_{i\in p} f_i x_{pj} && (3.67)\\
&= \sum_{i\in p: p\in P_\alpha(j)} f_i \left(\sum_{p\ni i: p\in P_\alpha(j)} x_{pj}\right) && (3.68)\\
&\leq \sum_{i\in p: p\in P_\alpha(j)} f_i y_i && (3.69)
\end{aligned}$$

Therefore, using Lemma 27, and summing over all super-clients j, we have the required expression. $\square$

6. Redistribute demand

We form our final integer solution $(\bar{x},\bar{y})$ for $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$ from $(\check{x},\check{y})$ by considering each super-client to have acted on behalf of the clients in its neighborhood. Each client in the neighborhood of a super-client is assigned to the same path facility as its super-client. We assume that the assignment that is appropriate for a super-client is appropriate for all of the clients in its neighborhood. The facility costs of $(\bar{x},\bar{y})$ are the same as $(\check{x},\check{y})$, since $\bar{y}=\check{y}$. We must, however, evaluate the service cost of the solution $(\bar{x},\bar{y})$ with respect to $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, denoted $\bar{z}_C$.

Lemma 29. *Let z_C be the service cost of the optimal solution (x,y) of $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, let $\check{z}_C$ be the service cost of the integer solution $(\check{x},\check{y})$ with respect to $(\mathrm{LP}'_{\mathrm{ML}p\mathrm{M}})$, and let $\bar{z}_C$ be the service cost of the integer solution $(\bar{x},\bar{y})$ with respect to $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$. Then $\bar{z}_C \leq \check{z}_C + 4z_C$.*

Proof. Let j' be some client in the neighborhood of super-client j. The demand $d_{j'}$ contributes $c_{p_j j'}$ to the left hand side of the required expression. We denote the facilities of the path p_j by $(i_k, \ldots, i_1)$. Then

$$\begin{aligned} c_{p_j j'} &= c_{j' i_k} + c_{i_k i_{k-1}} + \cdots + c_{i_2 i_1} && (3.70) \\ &\leq c_{j' j} + c_{j i_k} + c_{i_k i_{k-1}} + \cdots + c_{i_2 i_1} && (3.71) \\ &\leq 4C_{j'} + c_{p_j j} && (3.72) \end{aligned}$$

Multiplying (3.72) by $d_{j'}$ and summing over all $j' \in D$, we obtain the desired bound. □

Corollary 30. *Let z_C be the service cost of the optimal solution (x, y) of* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, *and let $\bar{z}_C$ be the service cost of the integer solution $(\bar{x}, \bar{y})$ with respect to* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$. *Let z_F be the facility cost of the optimal solution (x, y) of* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$, *and let $\bar{z}_F$ be the facility cost of the integer solution $(\bar{x}, \bar{y})$ with respect to* $(\mathrm{LP}_{\mathrm{ML}p\mathrm{M}})$. *Then* $\bar{z}_C \leq \left(\frac{8}{3} + \alpha + 4\right) z_C$ *and* $\bar{z}_F \leq \left(\frac{\alpha}{\alpha - 1}\right) z_F$.

Proof. We apply Corollary 26 to Lemma 29 to get the desired bound. □

Theorem 31. *The extension of the CGTS algorithm to the multi-level p-median problem is a* ≈ 7.814*-approximation.*

Proof. We set $\alpha \approx 1.148$ to balance facility and service costs and apply Lemma 28 and Corollary 30. □

Chapter 4

Computational Experiments

In order to evaluate the practicality of these approximation algorithms and techniques, a variety of approaches have been implemented. We have implemented the linear program solution rounding 3-approximation algorithm of Aardal, Chudak and Shmoys [ACS99] for the k-level facility location problem (**MLRR**); the linear program solution rounding ≈ 1.736-approximation algorithm of Chudak and Shmoys[CS] for the path reduction of the k-level facility location problem to a single-level problem (**PR-RR**); and the local improvement 3-approximation algorithm of Charikar and Guha [CG99] for the path reduction of the k-level facility location problem to a single-level problem (**PR-LI**). In addition, we have implemented a facility cost oblivious shortest path algorithm (**SP**) for comparison. Both algorithms for the path reduction single-level problem generate paths only as required, avoiding the enumeration of all paths. The computation of both linear program solution rounding algorithms is dominated by the solution of their linear programs, for which we describe a variety of column and constraint generation solution techniques.

There are, to our knowledge, no libraries of public instances for the k-level facility location problem, so we have constructed our own testbed of reasonable instances from other sources. We have created multi-level instances from a library of public instances for the single-level case, generated random instances, and constructed instances from population and location data for major US cities.

We will show in particular that generating single-level facility location problems via the path reduction works well in practice and that the direct application of linear program solution rounding to the multi-level facility location problem, while sometimes producing solutions inferior to the other approaches, can be run on larger problems in a reasonable amount of time. We will also demonstrate that, not surprisingly, ignoring facility costs and forming a shortest path tree solution works well on some instances but performs quite poorly on others.

4.1 Implementation

Environment

In order to evaluate each of these algorithms in a time efficient manner, we have chosen to use a programming environment appropriate for rapid prototyping of algorithms. Our goal is to consider the performance of these algorithms relative to the optimal solution in order the compare them against one another, speed of execution is relevant only as an indication of the contexts in which each algorithm might be considered practical.

We have used Matlab 5.3 as our implementation environment with this in mind. Matlab provides a variety of data structures and a comprehensive set of functions for manipulating those data structures, invisible memory management with garbage collection, and a straightforward interpreted scripting language. There are, however, a number of areas where Matlab is somewhat lacking. Matlab's support for linear and integer programming is weak, and its heritage as a platform for numerical computing means that its support for combinatorial objects is almost non-existent.

In order to use Matlab as a platform for this work, we have had to fill these gaps. The MatCPX package provides an interface between Matlab and the linear and integer programming software CPLEX 6.6, while the MatLEDA package provides an interface between Matlab and the graph class from the C++ class library LEDA. MatCPX provides high level functions for linear programming, linear programming with column generation, and mixed integer programming, plus access to many of the functions in the CPLEX callable library. In addition, MatCPX frees the user from the CPLEX imposed drudgery of handling sparse matrix data structures and dynamic memory management. The interested reader is referred to Appendix A for more details. MatLEDA provides a way to use the Matlab programming environment for the implementation of graph algorithms in a clear, concise and natural way. In addition, the user unfamiliar with C++ can use the basic functionality of LEDA without needing to learn C++. The interested reader is referred to Appendix B for more details. Note however, that this convenience does not come for free; an overhead in computation and memory is paid for this convenience.

This combination of Matlab, CPLEX via MatCPX, and LEDA via MatLEDA has been an effective environment for the rapid prototyping of approximation algorithms for facility location. In particular, this environment makes it possible to place the responsibility for implementing critical functionality on those software packages that do a particular task well. Under this model, it is possible to concentrate on domain specific algorithmic and programming questions without being distracted by side implementation issues.

Linear Program Solution Rounding

As an algorithmic technique for combinatorial optimization in practice, rounding the fractional optimal solution to a linear programming relaxation of an integer programming formulation works because we often get lucky. That is, often the optimal solution to the linear programming relaxation is integral and we have an optimal solution to our problem for the price of solving one linear program. Further, even when the optimal solution to the linear programming relaxation is not integral, the fractional solution often offers excellent solutions that are nearly integral. Typically, then, algorithms that round the solutions to linear programs are quite straightforward, in some sense the linear programming solution package has already done all the hard work. This is certainly demonstrated by the two rounding algorithms implemented here for the facility location problem.

Many linear program solution rounding algorithms have the nice property that integer values in the linear programming relaxation's solution are retained in the algorithm's output. This means that when the linear programming relaxation solves to an integer solution, the rounding algorithm will output the optimal integer solution. Unfortunately, the rounding algorithms of Chudak and Shmoys, and Aardal, Chudak and Shmoys do not have this property. These algorithms are, in some sense, too conservative in the facilities they choose to pay for, as a facility that was fully paid for by the linear program might not end up in the final solution. This effect is clearest in the algorithm of Aardal, Chudak and Shmoys and is indirectly responsible for its inferior relative performance.

Each of the linear program solution rounding algorithms for facility location considers only the variables that are non-zero in the optimal solution. These non-zero variables are used to select cluster centers and hence clusters that are then the basis for each algorithm's rounding procedure. Considering only the variables that are non-zero in the optimal linear programming solution reduces the rounding step from a major computation to a short calculation, even though the set of all variables of these linear programs is exponentially large. This makes it possible to run the rounding step repeatedly for a given fractional linear programming relaxation solution, increasing the likelihood that a good solution will be found.

All of this points to the fact that the difficulty in implementing these algorithms is not, in fact, the rounding procedure due to Chudak and Shmoys, or Aardal, Chudak and Shmoys, but instead the solution of these large linear programs. We describe how these linear programs are solved in the next two sections.

Solution of $(\mathrm{LP}_{\mathrm{MLFL}})$

As noted in Section 1.3, $(\mathrm{LP}_{\mathrm{MLFL}})$ has exponentially many variables for arbitrary k. We can solve this linear program in polynomial time using the ellipsoid algorithm, in theory, but in practice we turn to column generation. We have polynomially many constraints, few enough that we can generate all the rows of the constraint matrix of $(\mathrm{LP}_{\mathrm{MLFL}})$, but generating all the columns of the constraint matrix would force us to explicitly enumerate all the paths of P, which we cannot afford to do. Note that we can afford to include all the y_i variables; it is only the x_{pj} variables that must be generated only as needed.

To implement our column generation scheme, we need to specify how we seed our initial linear program, and how, given a solution to a partial linear program, we decide whether or not the solution is optimal, and if not, what path-client variables need to be added. We have already outlined our separation procedure in Section 1.3. Given a solution to a partial instance of $(\mathrm{LP}_{\mathrm{MLFL}})$, we must run a shortest path computation for each client and check whether or not the shortest path value is less than the corresponding dual variable. If the shortest path has value at least the dual variable then no further path variables are required to show optimality for this client, in this iteration. If the shortest path value for this client is less than the corresponding dual variable, then we add up to N path-client variables, using the cheapest paths that cost less than the dual variable. The parameter N is provided as input to the column generation procedure.

Algorithm 4.1: AcyclicShortestPaths is used as a subroutine for this computation. AcyclicShortestPaths finds shortest paths from origin nodes, which

Algorithm 4.1 ACYCLICSHORTESTPATHS(B, N)

Determine a lexicographic ordering of V.
for all $i \in V$ with outdegree$(i) > 0$ in reverse lexicographic order **do**
  $B(i) = \max\{B(i') - c_{ii'} : ii' \in E\}$.
Add distance 0 to queue $\mathcal{D}(i)$ for all i with indegree$(i) = 0$.
for all $i \in V$ in lexicographic order **do**
  for all $ii' \in E$ **and** $d \in \mathcal{D}(i)$ **do**
    if $d + c_{ii'} < B(i')$ **then**
      Add distance $d + c_{ii'}$ to $\mathcal{D}(i')$.
      if $|\mathcal{D}(i')| > N$ **then**
        Remove maximum distance from $\mathcal{D}(i')$.

have indegree zero, to all other (lexicographically greater) nodes of G. Two methods are provided for controlling the number of paths generated: a limit on the number of paths per node, N; and a path length upper bound on each terminal, which each have outdegree zero. The path length upper bound $B(i)$ for terminal i ensures that only the paths from an origin node to i with length at most $B(i)$ are returned. In fact, we can infer conservative intermediate path length upper bounds $B(i)$ for non-terminal nodes i by working back from the terminals. When we do not require either a bound on path lengths or on the number of paths per node, these can be easily eliminated from the algorithm in the obvious way. Note that in the statement of ACYCLICSHORTESTPATHS, the queue of valid path distances from the root to a node i is denoted $\mathcal{D}(i)$. For the multi-level facility location problem, we run this algorithm on the graph $\widetilde{G}$ of the alternate proof of Corollary 3 using service costs as edge costs. The dummy source facility i_0 is the sole origin node of $\widetilde{G}$; each client is a terminal.

The remaining issue to resolve is how to seed the initial linear program. If we initialize with too few path-client variables, then we will need many column additions to attain optimality. If we initialize with too many variables, our linear program becomes needlessly large. Three initialization procedures were implemented: N shortest paths (**N-Shortest**), primal-dual bounded paths (**P-D Bound**), and all possibly optimal paths (**All Poss.**).

Initializing our set of path-client variables with the N shortest paths for each client tends to lead to the smallest initial linear programs that require the most column generation iterations to attain optimality.

The third initialization procedure finds the length of the shortest path for each client *including* facility costs, as if each client was solely responsible to pay for the facility costs on the paths it uses. Notice that if a client in an optimal linear programming solution uses a path with service cost greater than this value, we could switch to this path and be better off. Therefore, if we initialize our linear program with all paths from a client whose service cost is less than this value, we will have added to the linear program all paths that could possibly be non-zero in an optimal solution. Therefore, initializing the linear program with all possibly optimal paths means that no column generation is necessary. Typically, this approach leads to far and away the largest initial linear programs.

Finally, the second procedure mentioned, primal-dual bounded paths, constructs a feasible dual solution to bound the length of the paths used to seed the initial linear program. We construct a multi-level analogue of the first phase of the primal-dual facility location algorithm, Algorithm 1.5: PrimalDualExpansion. Each client uniformly increases the amount they are prepared to pay to be connected to a top level facility, promising to contribute to the cost of facilities encountered on the way. This provides us with a feasible dual solution. We use the dual variable for each client to bound the length of the paths for which we will add a path-client variable to the initial linear program. This approach does not add all the variables required in order to attain optimality; we typically must still add columns to reach an optimal solution, but fewer column generation iterations are required compared to initializing with the N-shortest paths for each client. This technique seeds the initial linear program with many path-client variables for clients with many cheap paths, and few path-client variables for clients with few cheap paths. Algorithm 4.2: MultiLevelPrimalDualExpansion computes these path length upper bounds. The notation for MultiLevelPrimalDualExpansion is the same as that used to state Algorithm 1.5: PrimalDualExpansion in Section 1.2.

Solution of path reduction ($\mathrm{LP}_{\mathrm{FL}}$)

The facility location problem for the path reduction of a k-level problem has not only an exponential number of variables, but also an exponential number of constraints. To solve these linear programs without explicitly generating all possible paths, we must generate columns and rows only as necessary. While, in theory, for constant k

Algorithm 4.2 MultiLevelPrimalDualExpansion

$t_0 \leftarrow 0$; $F(j) \leftarrow \emptyset$ for each $j \in D$; $D(i) \leftarrow \emptyset, n_i \leftarrow 0, r_i \leftarrow f_i$ for each $i \in F$.
Add Contribute(i, j) to $\mathcal{E}$ at time c_{ij}, for each $i \in F_k, j \in D$.
while some client is not yet assigned **do**
  Pop next event and time t_1 from $\mathcal{E}$; $r_i \leftarrow r_i - (t_1 - t_0) * n_i$ for each $i \in F$.
  if next event is Contribute(i, j) and j is not yet assigned **then**
    if $i \notin I$ **and** $j \notin D(i)$ **then**
      $F(j) \leftarrow F(j) \cup \{i\}$; $D(i) \leftarrow D(i) \cup \{j\}$; $n_i \leftarrow n_i + 1$.
      Replace or add PaidFor(i) in $\mathcal{E}$ at time $t_1 + r_i/n_i$.
    else if $i \in I$ **then**
      if $i \in F_1$ **then**
        Client j is assigned to facility location i, $v_j = t_1$.
        $n_{i'} \leftarrow n_{i'} - 1$ for each $i' \in F(j) \setminus I$.
        Reset PaidFor(i') in $\mathcal{E}$ to time $t_1 + r_{i'}/n_{i'}$ for each $i' \in F(j) \setminus I$.
      else $\{ i \in F_l, l \geq 2 \}$
        Add Contribute(i, j) to $\mathcal{E}$ at time $t_1 + c_{ij}$, for each $i \in F_{l-1}, j \in D$.
  else if next event is PaidFor(i) **then**
    $I \leftarrow I \cup \{i\}$.
    for all unassigned $j \in D(i)$ **do**
      if $i \in F_1$ **then**
        Client j is assigned to facility location i, $v_j = t_1$.
        $n_{i'} \leftarrow n_{i'} - 1$ for each $i' \in F(j) \setminus I$.
        Reset PaidFor(i') in $\mathcal{E}$ to time $t_1 + r_{i'}/n_{i'}$ for each $i' \in F(j) \setminus I$.
      else $\{ i \in F_l, l \geq 2 \}$
        Add Contribute(i, j) to $\mathcal{E}$ at time $t_1 + c_{ij}$, for each $i \in F_{l-1}, j \in D$.
  $t_0 \leftarrow t_1$.

this is a polynomial time algorithm, in reality we must generate path facilities only as needed or the linear program rapidly will become too big. As with the solution of ($\mathrm{LP}_{\mathrm{MLFL}}$), we initialize the linear program with a small number of paths, each of which is a virtual super-facility, and then add additional paths to the formulation until we prove optimality.

Unlike testing a solution to ($\mathrm{LP}_{\mathrm{MLFL}}$) for optimality, we do not run a separate shortest path computation for each client to evaluate whether or not a solution to a path reduction ($\mathrm{LP}_{\mathrm{FL}}$) is optimal; instead we must check the next shortest path not yet considered, for that client. To achieve this, we maintain a M-shortest path cache consisting of the M shortest paths from each client to a facility of the top level. We set $M = 100$ initially, and resize by doubling M and recomputing the M-shortest paths from scratch whenever we need to add a path-client variable and the cache is exhausted for that client.

We apply the same three initialization procedures to the path reduction ($\mathrm{LP}_{\mathrm{FL}}$) as used for ($\mathrm{LP}_{\mathrm{MLFL}}$). The only modification we make is that instead of adding all paths bounded in length by the primal-dual bound, we add only as many paths for a client as is available in the path-cache.

Local Improvement

In order to run the local improvement algorithm of Charikar and Guha [CG99] on a path-reduction single-level instance, we require only that we are able to generate a virtual super-facility uniformly at random. In general, it might not be possible to generate a random observation from an exponentially-sized set in polynomial time; in this case however, we are lucky. Since each virtual super-facility is a path containing exactly one facility location from each of $F_1, \ldots, F_k$, we can uniformly generate paths by selecting, uniformly at random, one node from each of $F_1, \ldots, F_k$. This can clearly be done in polynomial time. Despite being able to sample quickly from this exponentially-sized set, the exponential number of virtual super-facilities is still the dominant term in the running time bound. We cannot afford to run this procedure for the number of iterations suggested by the analysis, instead we end the algorithm after some number of steps with no improvement.

Charikar and Guha's result indicates that we can expect to have met our approximation guarantee after $O(|\widehat{F}| \log |\widehat{F}|)$ iterations, but implicitly from their analysis, we know we can expect an improvement after $O(|\widehat{F}|)$ iterations. We terminate the algorithm after $2|\widehat{F}|$ iterations with no improvement.

The final detail that we must consider is that we need to record each of the paths in the current set of virtual super-facilities, so that we know whether or not to compare the potential improvement against its facility cost or zero. We do this by using a hash table to store all paths generated by the algorithm. This also gives us a cache of computed values for each generated path, saving us the cost of recomputing the service and facility costs of a path when it is generated again later.

Table 4.1: Imperial College OR Library uncapacitated facility location instances.

Instances	$\lvert F\rvert$	$\lvert D\rvert$	f_i
`cap71`, `cap72`, `cap73`, `cap74`	16	50	7500, 12500, 17500, 25000
`cap101`, `cap102`, `cap103`, `cap104`	25	50	7500, 12500, 17500, 25000
`cap131`, `cap132`, `cap133`, `cap134`	50	50	7500, 12500, 17500, 25000

4.2 Instances

OR Library

The Imperial College OR library [Bea90] contains instances for a variety of operations research problems, including the uncapacitated facility location problem. The library contains 12 instances of the uncapacitated facility location problem that we will use: `cap71`, `cap72`, `cap73`, `cap74`, `cap101`, `cap102`, `cap103`, `cap104`, `cap131`, `cap132`, `cap133`, and `cap134`. Instances differing only in their last digit have the same service cost matrix, but different facility costs. The default facility cost of each instance applies to all facilities except for one facility with zero facility cost. The default facility cost increases with the last digit of the instance. Table 4.1 summarizes these instances.

We must modify these instances in a number of ways to construct multi-level instances. We must first decide the level hierarchy to impose. Our intuition about the structure of the solutions suggests that the number of facilities in each level should increase with the level number. Given the number of nodes in the top level,

n_{F_1}, we construct the level structure by requiring that each level has twice as many facilities as the level above it, and that each level has at least as many facilities as the level above it. For instance `cap71`, setting $n_{F_1} = 2$ results in $k = 3$ levels of $2, 4$ and 10 facilities. Given a level heirachy, the assignment of facilities to levels is done at random. We will specify a random instance of `cap71`, with $k = 3$, levels of $2, 4$ and 10 facilities, and 50 clients as `cap71.2-4-10.50`.

The OR Library instances come with incomplete distance data, as only the distance between clients and facilities is provided. Our multi-level instances require inter-facility distances. We construct inter-facility distances by taking the minimum two hop distance between facilities via a client. Note that this particular distance construction ensures that these new distances do not violate the metric assumption.

Random Instances

We generate random instances for the multi-level facility location problem by placing facility and client locations uniformly inside a bounded two-dimensional box. Given n_{F_1} and k, we again impose the exponential level structure used for the OR Library instances. Facility costs for facilities at level l are drawn from a normal distribution with mean $\mu(l)$ and standard deviation $\sigma(l)$, defined in terms of $\mu(k)$ and $\sigma(k)$. The facility cost mean and standard deviation double for each level above k. Formally, $\mu(l) = 2\mu(l+1)$ and $\sigma(l) = 2\sigma(l+1)$ for $l = (k-1), \ldots, 1$. We construct five random instance classes: `random0`, `random1`, `random2`, `random3`, `random4`. Table 4.2 summarizes the facility costs for these instance classes. We choose the l_2 norm to compute the service cost between two points. We specify an

Table 4.2: Facility cost mean and standard deviation for random instance classes for the multi-level facility location problem.

Instance Class	$\mu(k)$	$\sigma(k)$
`random0`	100	10
`random1`	500	50
`random2`	1000	100
`random3`	5000	500
`random4`	10000	1000

instance of instance class `random1` with $n_{F_1} = 2$, $k = 3$ (resulting in levels of 2, 4 and 8 facilities), and $|D| = 100$ as `random1.2-4-8.100`. This instance will have level 1 facility costs drawn from a Normal$(2000, 200)$ distribution, level 2 facility costs drawn from a Normal$(1000, 100)$ distribution, and level 3 facility costs drawn from a Normal$(500, 50)$ distribution.

US Cities

The data file `US.lat-long`, cited in instances from TSPLIB, contains location and population data for all registered place names in the United States. We construct instances in which the levels are defined by population. The cities of population 1 000 000 or more make up level 1; cities with population between 500 000 and 1 000 000, level 2; 250 000 and 500 000, level 3; and cities with population between 50 000 and 250 000 are the clients. We construct instances with the same hierarchical

Table 4.3: Facility cost mean and standard deviation for US city instances for the multi-level facility location problem.

Instance	$\mu(k)$	$\sigma(k)$
US0	100	0
US1	500	0
US2	1000	0
US3	5000	0
US4	10000	0

doubling facility cost structure as our random instances, except that we assume deterministic facility costs equal to the mean. The population levels above lead to instances such as `US2.6-16-30.348` with 6 level 1 facilities, 16 level 2 facilities, 30 level 3 facilities, and 348 clients. Table 4.3 summarizes the facility costs of these instances.

4.3 Results

General Discussion

Tables 4.4 through 4.17, and Figures 4.1 through 4.10 document the results of running our set of multi-level facility location algorithms (**MLRR**, **PR-RR**, **PR-LI**, and **SP**) on the described set of instances. Each table entry concerning the OR Library or the random instances represents 100 trials; mean and standard deviation summary statistics are presented for each entry. The US cities instances have no random component; these tables represent the result of a single trial only.

The linear program column generation parameter N is set to 10 throughout. Setting N larger than this did not seem to improve convergence, either in total elapsed time or in the number of column generation iterations. We use the same value, $N = 10$, when seeding our linear programs with the N shortest paths.

The shortest path cache used in the solution of the path reduction $(\mathrm{LP}_{\mathrm{FL}})$ initially stores the 100 shortest paths for each client, and is resized by doubling its size and recomputing from scratch.

Our randomized local improvement algorithm is set to terminate after $2|\widehat{F}|$ iterations without improvement.

These parameters could be hand tuned for each instance or instance class, but this is not necessary in order to achieve a qualitative comparison of the algorithms. These values for the parameters are used throughout.

To ensure that we get the fairest possible comparison of these algorithms, the output of each algorithm is cleaned up by reassigning each client to the shortest service cost path through the installed facilities. The service cost is then computed by summing these shortest path distances, and any facility that was installed but not used is removed, possibly improving the facility cost.

Solution of $(\mathrm{LP}_{\mathrm{MLFL}})$

Tables 4.4 and 4.5 summarize the results of running each of the linear programming column generation schemes for $(\mathrm{LP}_{\mathrm{MLFL}})$ on a medium sized problem. We make some general observations. First, it is clear that the magnitude of the facility costs is very significant. When facilities are very expensive, many paths must be considered to prove optimality. The N shortest path seeding scheme is most immune to this effect, while the primal-dual bounded and all possible optimal paths schemes very quickly start to produce huge initial linear programs that take a long time to solve. We note that the primal-dual bounded seeding scheme starts out, for cheap facilities, behaving like the N-shortest path scheme and ends up, for expensive facilities, behaving like the all possible optimal paths scheme.

As a result of these experiments, we use the N shortest path seeding scheme with column generation to solve ($\mathrm{LP}_{\mathrm{MLFL}}$) in the following experiments.

Table 4.4: Mean (standard deviation) initial and final non-zeros required to solve ($\mathrm{LP}_{\mathrm{MLFL}}$) by column generation.

Instance	**N-Shortest**		**P-D Bound**		**All Poss.**	
`cap101.4-8-13.50`	3250	(0)	3968	(780)	53461	(4253)
	3814	(133)	5260	(872)		
`cap102.4-8-13.50`	3250	(0)	6456	(1234)	63182	(3021)
	4284	(221)	8190	(1395)		
`cap103.4-8-13.50`	3250	(0)	8568	(1167)	68313	(2876)
	4701	(320)	10505	(1265)		
`cap104.4-8-13.50`	3250	(0)	11624	(1145)	72787	(2212)
	5240	(438)	13621	(1219)		
`random0.4-8-16.50`	3600	(0)	1997	(85)	25019	(5889)
	3628	(40)	2514	(204)		
`random1.4-8-16.50`	3600	(0)	2992	(427)	103999	(0)
	4161	(227)	4121	(521)		
`random2.4-8-16.50`	3600	(0)	4492	(854)	104000	(0)
	5027	(500)	5774	(929)		
`random3.4-8-16.50`	3600	(0)	36327	(8178)	104000	(0)
	8608	(1312)	36363	(8162)		
`random4.4-8-16.50`	3600	(0)	87845	(5377)	104000	(0)
	11573	(2319)	87845	(5376)		

Table 4.5: Mean (standard deviation) time in seconds to solve (LP_{MLFL}) by column generation.

Instance	***N*-Shortest**		**P-D Bound**		**All Poss.**	
`cap101.4-8-13.50`	4.74	(0.89)	17.50	(3.22)	41.60	(6.21)
`cap102.4-8-13.50`	6.81	(1.40)	19.36	(2.73)	57.07	(5.22)
`cap103.4-8-13.50`	8.72	(1.88)	20.87	(2.50)	66.14	(5.56)
`cap104.4-8-13.50`	11.73	(2.67)	23.05	(2.70)	75.40	(4.16)
`random0.4-8-16.50`	2.43	(1.33)	22.42	(8.15)	11.87	(4.49)
`random1.4-8-16.50`	6.00	(1.48)	32.81	(8.14)	173.27	(7.53)
`random2.4-8-16.50`	9.38	(2.24)	40.36	(11.86)	171.23	(6.59)
`random3.4-8-16.50`	22.95	(4.40)	53.63	(11.47)	172.73	(7.67)
`random4.4-8-16.50`	34.54	(8.16)	143.64	(15.45)	173.06	(7.23)

Solution of path reduction $(\mathrm{LP}_{\mathrm{FL}})$

Tables 4.6 and 4.7 summarize the results of running each of the linear programming row and column generation schemes for the path reduction $(\mathrm{LP}_{\mathrm{FL}})$ on a medium sized problem. We make some general observations. First, we notice the same effect as for the solution of $(\mathrm{LP}_{\mathrm{MLFL}})$, the linear programs get much more difficult to solve as the facilities become more expensive. Unlike the $(\mathrm{LP}_{\mathrm{MLFL}})$ case, however, the N-shortest path seeding scheme adds almost as many variables as the scheme that adds paths for all possibly optimal paths, only it has to do it N paths at a time. Adding all of these paths is expensive because we must retrieve them from the shortest path cache, and computing the cache of the M shortest paths for $M \geq 800$ starts to become prohibitively expensive, computationally. It is not unusual for this size problem to have one client that forces the path cache to be built for sizes of 3200 or more. This is most apparent in Table 4.7 in which the time to solve the linear program by enumerating all possibly optimal paths stays approximately constant, while the time to solve the linear program by seeding with the N shortest paths or primal-dual bounded paths explodes. The behavior is very instance dependent, however, as enumerating all possibly optimal paths becomes impossible for large instances or instances with expensive facilities.

We use a mixture of the N shortest paths and all possibly optimal paths initial linear program seeding strategies in order to solve the path reduction $(\mathrm{LP}_{\mathrm{FL}})$ in the experiments to follow.

Table 4.6: Mean (standard deviation) initial and final non-zeros required to solve the path reduction ($\mathrm{LP_{FL}}$) by row and column generation.

Instance	**N-Shortest**		**P-D Bound**		**All Poss.**	
`cap101.4-8-13.50`	1500	(0)	1663	(239)	38765	(2684)
	4198	(431)	5331	(498)		
`cap102.4-8-13.50`	1500	(0)	2274	(220)	46261	(2175)
	5064	(552)	6565	(595)		
`cap103.4-8-13.50`	1500	(0)	2769	(233)	50070	(2012)
	5604	(622)	7399	(626)		
`cap104.4-8-13.50`	1500	(0)	3525	(304)	53635	(1590)
	6669	(808)	8762	(805)		
`random0.4-8-16.50`	1500	(0)	292	(57)	18014	(4806)
	1561	(61)	1206	(167)		
`random1.4-8-16.50`	1500	(0)	1081	(312)	76800	(0)
	2530	(440)	2515	(499)		
`random2.4-8-16.50`	1500	(0)	2233	(796)	76800	(0)
	3727	(1002)	4135	(1290)		
`random3.4-8-16.50`	1500	(0)	14694	(497)	76800	(0)
	26951	(6399)	27151	(5291)		
`random4.4-8-16.50`	1500	(0)	15000	(0)	76800	(0)
	65246	(4429)	52220	(15251)		

Table 4.7: Mean (standard deviation) time in seconds to solve the path reduction (LP_{FL}) by row and column generation.

Instance	N-Shortest		P-D Bound		All Poss.	
`cap101.4-8-13.50`	4.49	(1.07)	13.42	(2.15)	23.34	(4.41)
`cap102.4-8-13.50`	5.28	(0.62)	13.63	(1.57)	29.13	(5.92)
`cap103.4-8-13.50`	6.01	(0.71)	14.41	(1.43)	31.52	(6.14)
`cap104.4-8-13.50`	7.44	(0.99)	16.04	(1.83)	36.81	(8.18)
`random0.4-8-16.50`	1.71	(0.59)	20.86	(9.90)	7.97	(2.63)
`random1.4-8-16.50`	2.82	(0.68)	28.92	(9.28)	50.53	(6.81)
`random2.4-8-16.50`	4.10	(1.17)	33.91	(10.71)	52.57	(6.06)
`random3.4-8-16.50`	92.30	(39.40)	96.33	(33.28)	49.53	(6.38)
`random4.4-8-16.50`	456.23	(60.28)	316.95	(152.05)	50.23	(6.51)

Performance of multi-level facility location algorithms

Tables 4.8 through 4.17 and Figures 4.1 through 4.10 present the results of each algorithm run on each of the instances: the multi-level linear program solution rounding algorithm of Aardal, Chudak and Shmoys [ACS99] (**MLRR**), the path reduction linear program solution rounding algorithm (**PR-RR**), the path reduction local improvement algorithm (**PR-LI**), and the facility cost oblivious shortest path algorithm (**SP**).

We note first that the path reduction algorithms perform very well compared to the optimal solution of the linear programming relaxation on the instances tested. Typically the solution output by **PR-RR** and **PR-LI** is within a few percent of the lower bound on the cost of the optimal solution. These algorithms take longer to run than the others and often require significantly more memory, but for small to moderately large sized instances, this would seem to be the method of choice.

The multi-level rounding algorithm **MLRR** can typically handle larger problems than the path reduction approaches. Usually, **MLRR** will require significantly less memory and a good deal less time to produce a solution. On the other hand, the solution quality is usually inferior to the solution produced by the path reduction approaches, except when the facility costs start to dominate the total cost of the solution. **MLRR** tends to open fewer facilities than the other algorithms due to its clustering strategy; this is exactly the right thing to do when facilities are expensive. When facilities are cheap, however, this approach leads to the clients paying higher than necessary service costs.

The facility cost oblivious shortest path algorithm, somewhat predictably, does exceptionally well when the facilities are cheap, and quite poorly when the facilities are expensive. It takes almost no time to run, produces an excellent solution if facilities are cheap, and provides a test for the significance of facility costs in the optimal solution. If, by running **SP**, we establish that facilities are cheap, we need not bother running the more expensive multi-level facility location algorithms at all. If running **SP** establishes that facilities are expensive, then we know that running a multi-level facility location algorithm is necessary, and is likely to result in a better solution.

However, as facilities become expensive, the two linear program solution rounding approaches consume significantly more memory and running time. This is because each linear program requires more paths to prove optimality, as the facility costs increase. This effect is most significant for the path reduction ($\mathrm{LP}_{\mathrm{FL}}$), but ($\mathrm{LP}_{\mathrm{MLFL}}$) suffers from this problem too. On the other hand, the local improvement algorithm terminates earlier as facility costs increase. This is because the stopping condition is based on the number of iterations since the last improvement. As facilities get more expensive, it gets harder to find an improving move and the algorithm is likely to stop earlier. We note that this does not seem to affect the quality of the solution for the instances tested here, though it seems that stopping earlier should result in inferior solutions. The theoretical running time bound of Charikar and Guha is independent of facility costs and might be more appropriate as a stopping criterion for instances in which facility costs dominate. Nevertheless, the local improvement algorithm provides a nice complement to the linear programming approaches, that take longer to run as the facility costs increase.

Finally, we point out that as the number of clients increases, **MLRR**'s running times skyrocket. This is demonstrated by the running times for the US cites instances, which have 348 clients. The separation algorithm that our column generation procedure uses for (LP_{MLFL}) requires the solution of a shortest path problem for every client; as the number of clients increases this computation starts to dominate. Note that this effect is most evident for the N-shortest paths seeding strategy since it typically requires the most column generation iterations.

Table 4.8: Mean (standard deviation) performance of multi-level facility location algorithms relative to (LP_{MLFL}) lower bound.

Instance	MLRR		PR-RR		PR-LI		SP	
cap71.2-4-10.50	1.4664	(0.5538)	1.0179	(0.0079)	1.0167	(0.0074)	1.0063	(0.0030)
cap72.2-4-10.50	1.5317	(0.5860)	1.0128	(0.0075)	1.0123	(0.0075)	1.0173	(0.0054)
cap73.2-4-10.50	1.3378	(0.3825)	1.0081	(0.0054)	1.0083	(0.0055)	1.0388	(0.0092)
cap74.2-4-10.50	1.3743	(0.4513)	1.0150	(0.0169)	1.0169	(0.0165)	1.0713	(0.0116)
cap71.4-12.50	1.6478	(0.6707)	1.0075	(0.0062)	1.0079	(0.0061)	1.0107	(0.0043)
cap72.4-12.50	1.4305	(0.5334)	1.0107	(0.0077)	1.0111	(0.0074)	1.0310	(0.0087)
cap73.4-12.50	1.2835	(0.2872)	1.0096	(0.0085)	1.0102	(0.0080)	1.0570	(0.0101)
cap74.4-12.50	1.2805	(0.3047)	1.0054	(0.0080)	1.0066	(0.0086)	1.1093	(0.0136)
cap71.8-8.50	1.4424	(0.4753)	1.0056	(0.0054)	1.0056	(0.0054)	1.0068	(0.0046)
cap72.8-8.50	1.5724	(0.5926)	1.0061	(0.0078)	1.0065	(0.0076)	1.0188	(0.0080)
cap73.8-8.50	1.3723	(0.4362)	1.0053	(0.0080)	1.0056	(0.0080)	1.0335	(0.0110)
cap74.8-8.50	1.4186	(0.4761)	1.0057	(0.0093)	1.0069	(0.0095)	1.0615	(0.0161)

Table 4.9: Mean (standard deviation) time in seconds required by multi-level facility location algorithms.

Instance	**MLRR**	**PR-RR**	**PR-LI**	**SP**
`cap71.2-4-10.50`	2.7308 (0.7239)	1.7720 (0.1361)	2.2715 (0.4304)	0.0103 (0.0017)
`cap72.2-4-10.50`	2.9878 (0.5080)	1.9404 (0.1779)	2.2268 (0.3991)	0.0098 (0.0020)
`cap73.2-4-10.50`	3.6286 (0.7129)	2.0263 (0.1791)	2.1974 (0.4064)	0.0097 (0.0017)
`cap74.2-4-10.50`	4.1283 (0.9607)	2.2140 (0.2791)	2.1795 (0.4690)	0.0100 (0.0000)
`cap71.4-12.50`	1.8817 (0.1876)	1.4639 (0.1084)	1.5350 (0.2754)	0.0097 (0.0022)
`cap72.4-12.50`	2.2607 (0.3633)	1.4876 (0.0953)	1.5363 (0.3077)	0.0087 (0.0034)
`cap73.4-12.50`	2.6084 (0.3364)	1.5555 (0.1076)	1.4255 (0.2792)	0.0092 (0.0027)
`cap74.4-12.50`	2.9709 (0.4479)	1.6080 (0.1290)	1.4204 (0.2487)	0.0092 (0.0027)
`cap71.8-8.50`	2.2283 (0.3763)	1.5325 (0.1022)	1.8430 (0.4712)	0.0084 (0.0037)
`cap72.8-8.50`	2.6297 (0.5916)	1.5349 (0.1262)	1.8792 (0.3689)	0.0085 (0.0036)
`cap73.8-8.50`	3.0190 (0.6622)	1.6676 (0.1369)	1.7453 (0.3385)	0.0088 (0.0033)
`cap74.8-8.50`	3.5462 (0.6952)	1.7689 (0.1561)	1.7478 (0.3649)	0.0090 (0.0030)

Table 4.10: Mean (standard deviation) performance of multi-level facility location algorithms relative to (LP_{MLFL}) lower bound.

Instance	**MLRR**	**PR-RR**	**PR-LI**	**SP**
`cap101.4-8-13.50`	1.6402 (0.4837)	1.0232 (0.0140)	1.0228 (0.0143)	1.0105 (0.0050)
`cap102.4-8-13.50`	1.4370 (0.2802)	1.0210 (0.0119)	1.0207 (0.0120)	1.0311 (0.0093)
`cap103.4-8-13.50`	1.4563 (0.3643)	1.0141 (0.0100)	1.0142 (0.0102)	1.0611 (0.0111)
`cap104.4-8-13.50`	1.3561 (0.2496)	1.0149 (0.0137)	1.0143 (0.0131)	1.1132 (0.0209)
`cap101.8-17.50`	1.7006 (0.4995)	1.0158 (0.0114)	1.0155 (0.0113)	1.0226 (0.0058)
`cap102.8-17.50`	1.4777 (0.3010)	1.0169 (0.0155)	1.0176 (0.0154)	1.0604 (0.0100)
`cap103.8-17.50`	1.3706 (0.1781)	1.0175 (0.0137)	1.0170 (0.0132)	1.1110 (0.0123)
`cap104.8-17.50`	1.3618 (0.2168)	1.0112 (0.0143)	1.0113 (0.0138)	1.1947 (0.0173)
`cap101.12-13.50`	1.7698 (0.5935)	1.0101 (0.0088)	1.0097 (0.0083)	1.0140 (0.0051)
`cap102.12-13.50`	1.6228 (0.4703)	1.0108 (0.0140)	1.0108 (0.0128)	1.0384 (0.0071)
`cap103.12-13.50`	1.4707 (0.4088)	1.0128 (0.0129)	1.0132 (0.0123)	1.0758 (0.0111)
`cap104.12-13.50`	1.4108 (0.3104)	1.0103 (0.0134)	1.0108 (0.0128)	1.1338 (0.0146)

Table 4.11: Mean (standard deviation) time in seconds required by multi-level facility location algorithms.

Instance	**MLRR**		**PR-RR**		**PR-LI**		**SP**	
`cap101.4-8-13.50`	4.8423	(0.8549)	4.2757	(0.3702)	10.9062	(2.3215)	0.0105	(0.0022)
`cap102.4-8-13.50`	6.7878	(1.5830)	5.0671	(0.6202)	9.8895	(1.9981)	0.0102	(0.0014)
`cap103.4-8-13.50`	8.8167	(1.9613)	5.9484	(0.6730)	10.0648	(2.3602)	0.0103	(0.0017)
`cap104.4-8-13.50`	11.5991	(2.4986)	7.0082	(0.8306)	10.0914	(2.1839)	0.0105	(0.0022)
`cap101.8-17.50`	3.0347	(0.5710)	1.7601	(0.1318)	4.2549	(0.9458)	0.0106	(0.0024)
`cap102.8-17.50`	4.4312	(1.0123)	2.0100	(0.2198)	3.8785	(1.0653)	0.0101	(0.0010)
`cap103.8-17.50`	6.2608	(1.8612)	2.1796	(0.2388)	3.3801	(0.7296)	0.0104	(0.0020)
`cap104.8-17.50`	7.6459	(1.7907)	2.3195	(0.2133)	3.3819	(0.7553)	0.0101	(0.0010)
`cap101.12-13.50`	3.8250	(0.4793)	1.9663	(0.1317)	4.7102	(1.1500)	0.0097	(0.0017)
`cap102.12-13.50`	5.2474	(0.7356)	2.1930	(0.2027)	4.5084	(1.1534)	0.0097	(0.0017)
`cap103.12-13.50`	7.1861	(1.3122)	2.3821	(0.2417)	3.8899	(0.8992)	0.0096	(0.0020)
`cap104.12-13.50`	9.2109	(1.6566)	2.6290	(0.2833)	3.8528	(0.8539)	0.0095	(0.0022)

Table 4.12: Mean (standard deviation) performance of multi-level facility location algorithms relative to (LP_{MLFL}) lower bound.

Instance	**MLRR**		**PR-RR**		**PR-LI**		**SP**	
cap131.8-16-26.50	1.9852	(0.6155)	1.0299	(0.0178)	1.0301	(0.0180)	1.0798	(0.0105)
cap132.8-16-26.50	1.5413	(0.3493)	1.0288	(0.0151)	1.0308	(0.0144)	1.1556	(0.0154)
cap133.8-16-26.50	1.4336	(0.2133)	1.0266	(0.0142)	1.0268	(0.0144)	1.2321	(0.0223)
cap134.8-16-26.50	1.3565	(0.2147)	1.0230	(0.0152)	1.0235	(0.0152)	1.3621	(0.0336)
cap131.12-38.50	1.7319	(0.5360)	1.0240	(0.0133)	1.0242	(0.0129)	1.1479	(0.0104)
cap132.12-38.50	1.4326	(0.1620)	1.0302	(0.0159)	1.0307	(0.0163)	1.2725	(0.0180)
cap133.12-38.50	1.3667	(0.0974)	1.0248	(0.0167)	1.0253	(0.0159)	1.4122	(0.0213)
cap134.12-38.50	1.3115	(0.0571)	1.0220	(0.0158)	1.0225	(0.0146)	1.6175	(0.0313)
cap131.16-34.50	1.7335	(0.5340)	1.0219	(0.0123)	1.0220	(0.0123)	1.1256	(0.0120)
cap132.16-34.50	1.4821	(0.2711)	1.0265	(0.0186)	1.0265	(0.0176)	1.2379	(0.0178)
cap133.16-34.50	1.3865	(0.1756)	1.0244	(0.0186)	1.0245	(0.0182)	1.3611	(0.0252)
cap134.16-34.50	1.3260	(0.1166)	1.0236	(0.0156)	1.0236	(0.0153)	1.5475	(0.0326)

Table 4.13: Mean (standard deviation) time in seconds required by multi-level facility location algorithms.

Instance	MLRR	PR-RR	PR-LI	SP
`cap131.8-16-26.50`	14.4068 (2.6837)	79.6092 (10.1252)	105.4819 (25.1068)	0.0165 (0.0048)
`cap132.8-16-26.50`	21.7567 (4.6841)	105.0441 (22.1061)	83.3658 (16.2891)	0.0147 (0.0050)
`cap133.8-16-26.50`	29.5086 (5.5658)	133.2602 (27.9786)	79.1091 (17.3509)	0.0162 (0.0049)
`cap134.8-16-26.50`	42.9266 (9.4201)	183.8802 (43.8830)	75.0918 (17.5789)	0.0161 (0.0049)
`cap131.12-38.50`	11.9970 (1.6120)	4.7360 (1.5285)	16.4970 (4.1799)	0.0192 (0.0027)
`cap132.12-38.50`	16.9463 (2.5916)	6.1477 (0.7402)	11.5427 (3.2143)	0.0170 (0.0046)
`cap133.12-38.50`	22.1845 (2.9791)	6.7031 (0.8217)	10.3977 (2.2014)	0.0180 (0.0040)
`cap134.12-38.50`	29.2424 (3.5499)	7.6561 (0.8656)	10.0775 (2.3451)	0.0184 (0.0037)
`cap131.16-34.50`	12.8150 (3.2186)	5.8230 (1.4754)	18.9799 (5.0332)	0.0189 (0.0031)
`cap132.16-34.50`	16.9274 (2.5633)	7.3936 (1.3256)	14.1575 (4.1136)	0.0160 (0.0049)
`cap133.16-34.50`	23.4183 (3.3878)	8.2285 (1.2467)	13.3315 (3.3673)	0.0173 (0.0045)
`cap134.16-34.50`	31.9039 (5.7351)	9.6404 (1.4332)	12.1346 (3.1513)	0.0181 (0.0039)

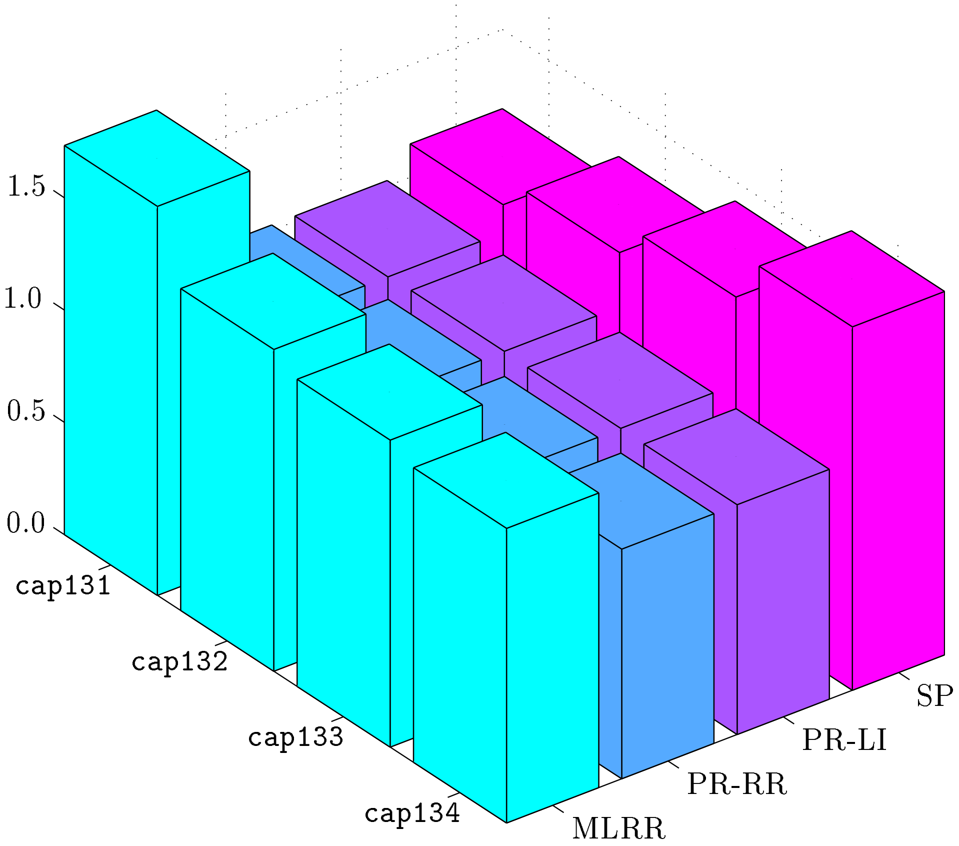

Figure 4.1: Mean relative performance of multi-level facility location algorithms, `cap13x.12-38.50` instances.

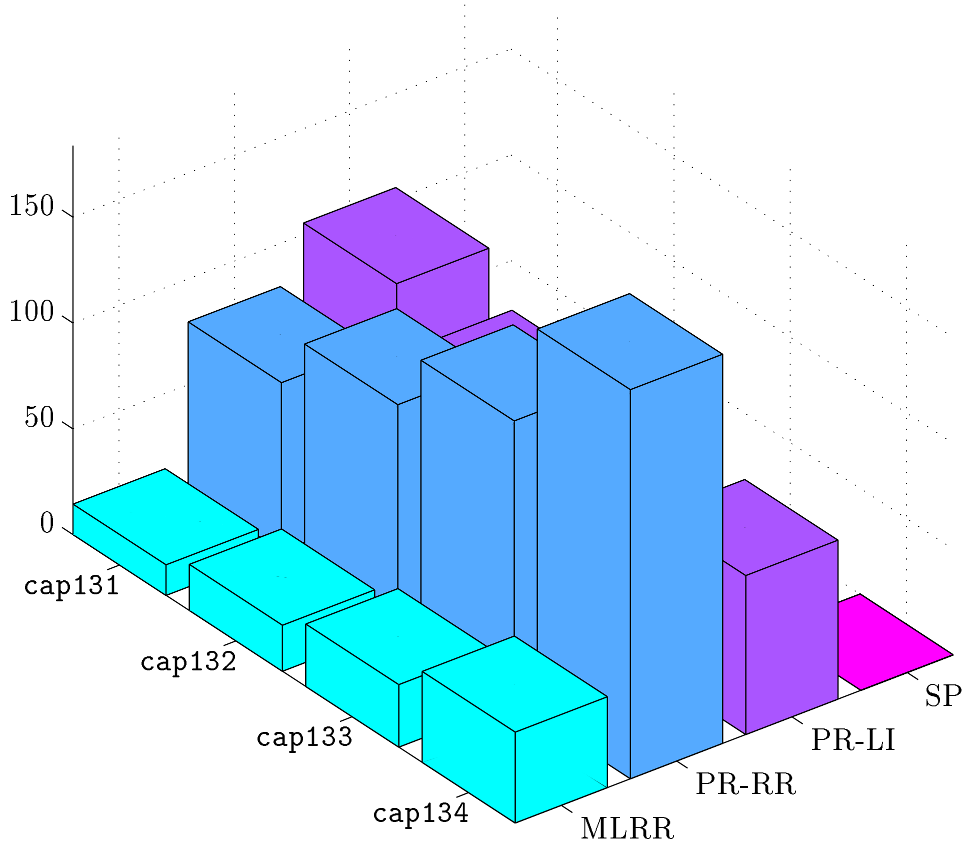

Figure 4.2: Mean time in seconds required by multi-level facility location algorithms, `cap13x.8-16-26.50` instances.

Table 4.14: Mean (standard deviation) performance of multi-level facility location algorithms relative to (LP_{MLFL}) lower bound.

Instance	**MLRR**		**PR-RR**		**PR-LI**		**SP**	
`random0.2-4-8-16.50`	1.1052	(0.0730)	1.0265	(0.0163)	1.0268	(0.0171)	1.0221	(0.0139)
`random1.2-4-8-16.50`	1.0561	(0.0552)	1.0389	(0.0320)	1.0431	(0.0385)	1.1699	(0.0683)
`random2.2-4-8-16.50`	1.0257	(0.0355)	1.0157	(0.0195)	1.0209	(0.0263)	1.3429	(0.1147)
`random3.2-4-8-16.50`	1.0004	(0.0023)	1.0002	(0.0015)	1.0019	(0.0073)	1.9981	(0.3214)
`random4.2-4-8-16.50`	1.0000	(0.0000)	1.0000	(0.0000)	1.0019	(0.0062)	2.3508	(0.3983)
`random0.4-8-16.50`	1.0936	(0.0705)	1.0188	(0.0123)	1.0167	(0.0110)	1.0254	(0.0130)
`random1.4-8-16.50`	1.0543	(0.0635)	1.0288	(0.0312)	1.0346	(0.0354)	1.2182	(0.0724)
`random2.4-8-16.50`	1.0399	(0.0489)	1.0243	(0.0298)	1.0268	(0.0329)	1.4656	(0.1270)
`random3.4-8-16.50`	1.0002	(0.0017)	1.0002	(0.0017)	1.0019	(0.0084)	2.7631	(0.3544)
`random4.4-8-16.50`	1.0000	(0.0000)	1.0000	(0.0000)	1.0023	(0.0096)	3.4239	(0.4990)
`random0.16-32.50`	1.0277	(0.0251)	1.0098	(0.0071)	1.0128	(0.0089)	1.0899	(0.0248)
`random1.16-32.50`	1.0061	(0.0166)	1.0024	(0.0060)	1.0072	(0.0112)	1.6399	(0.1171)
`random2.16-32.50`	1.0023	(0.0079)	1.0012	(0.0043)	1.0088	(0.0145)	2.2564	(0.1838)
`random3.16-32.50`	1.0000	(0.0000)	1.0000	(0.0000)	1.0036	(0.0194)	5.8381	(0.5983)
`random4.16-32.50`	1.0000	(0.0000)	1.0000	(0.0000)	1.0030	(0.0138)	8.3261	(0.7849)

Table 4.15: Mean (standard deviation) time in seconds required by multi-level facility location algorithms.

Instance	**MLRR**		**PR-RR**		**PR-LI**		**SP**	
random0.2-4-8-16.50	3.1327	(1.3079)	92.6423	(28.5342)	23.3704	(5.7417)	0.0117	(0.0038)
random1.2-4-8-16.50	7.3342	(2.8235)	154.2694	(25.2282)	18.5447	(3.9542)	0.0119	(0.0039)
random2.2-4-8-16.50	11.3663	(3.7633)	155.9923	(21.0568)	18.7691	(3.7916)	0.0122	(0.0042)
random3.2-4-8-16.50	31.5145	(9.0398)	157.0255	(19.0562)	19.9649	(4.8448)	0.0120	(0.0040)
random4.2-4-8-16.50	45.8278	(13.8938)	156.8794	(17.1692)	18.6431	(3.4388)	0.0129	(0.0046)
random0.4-8-16.50	2.2294	(0.4617)	7.6348	(2.4222)	11.9577	(2.8342)	0.0105	(0.0022)
random1.4-8-16.50	6.4361	(3.5660)	56.3082	(3.8736)	9.5326	(2.1407)	0.0107	(0.0026)
random2.4-8-16.50	10.3104	(4.0772)	60.3888	(4.9384)	8.7875	(2.1498)	0.0112	(0.0033)
random3.4-8-16.50	23.1795	(4.5208)	61.7212	(5.5599)	9.0509	(1.9139)	0.0114	(0.0035)
random4.4-8-16.50	33.7978	(8.6704)	56.3169	(3.0971)	8.7020	(1.7592)	0.0108	(0.0027)
random0.16-32.50	2.8999	(2.1814)	2.0507	(0.2885)	13.6763	(2.8839)	0.0153	(0.0050)
random1.16-32.50	8.1978	(3.2354)	64.4653	(13.8592)	11.8361	(2.9603)	0.0154	(0.0050)
random2.16-32.50	14.1620	(3.3827)	68.4252	(4.4770)	10.1975	(2.8084)	0.0153	(0.0050)
random3.16-32.50	42.5430	(6.8675)	86.2557	(5.7337)	8.1436	(1.8155)	0.0154	(0.0050)
random4.16-32.50	61.7692	(10.8758)	76.6527	(8.2481)	8.3457	(1.4966)	0.0157	(0.0050)

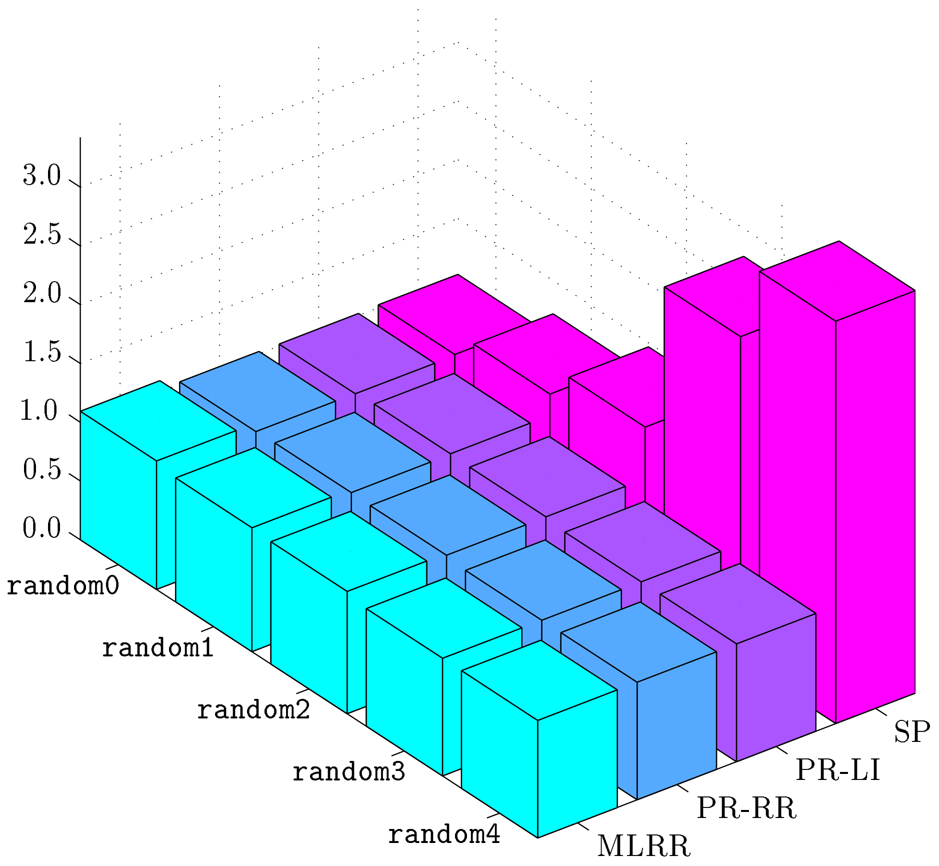

Figure 4.3: Mean relative performance of multi-level facility location algorithms, `randomx.4-8-16.50` instances.

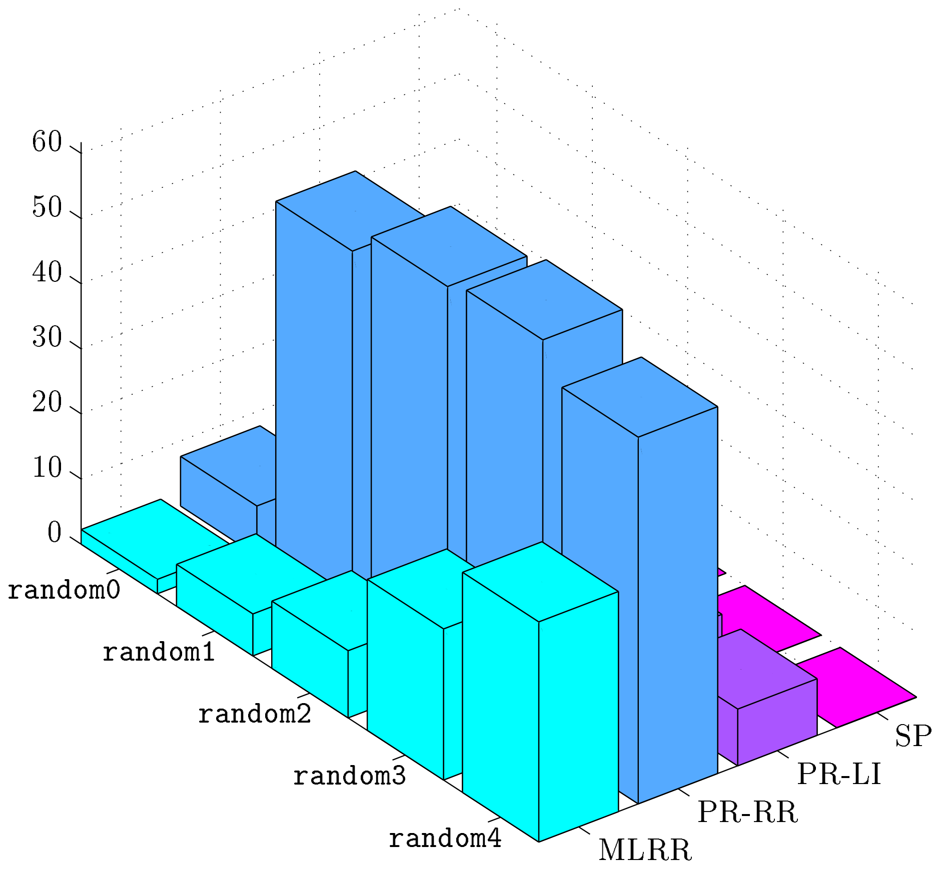

Figure 4.4: Mean time in seconds required by multi-level facility location algorithms, `randomx.4-8-16.50` instances.

Table 4.16: Performance of multi-level facility location algorithms relative to (LP_{MLFL}) lower bound.

Instance	**MLRR**	**PR-RR**	**PR-LI**	**SP**
`US0.6-16-30.348`	1.0876	1.0038	1.0038	1.0046
`US1.6-16-30.348`	1.0577	1.0065	1.0035	1.0302
`US2.6-16-30.348`	1.0449	1.0015	1.0015	1.0783
`US3.6-16-30.348`	1.0460	1.0333	1.0333	1.4609
`US4.6-16-30.348`	1.0455	1.0289	1.1079	1.9191

Table 4.17: Time in seconds required by multi-level facility location algorithms.

Instance	**MLRR**	**PR-RR**	**PR-LI**	**SP**
`US0.6-16-30.348`	31.0800	16.0500	126.1100	0.0600
`US1.6-16-30.348`	64.9500	18.2100	105.5200	0.0600
`US2.6-16-30.348`	143.5000	23.2600	158.5900	0.0600
`US3.6-16-30.348`	420.9200	49.9100	105.2000	0.0700
`US4.6-16-30.348`	813.4700	160.5100	89.6500	0.0700

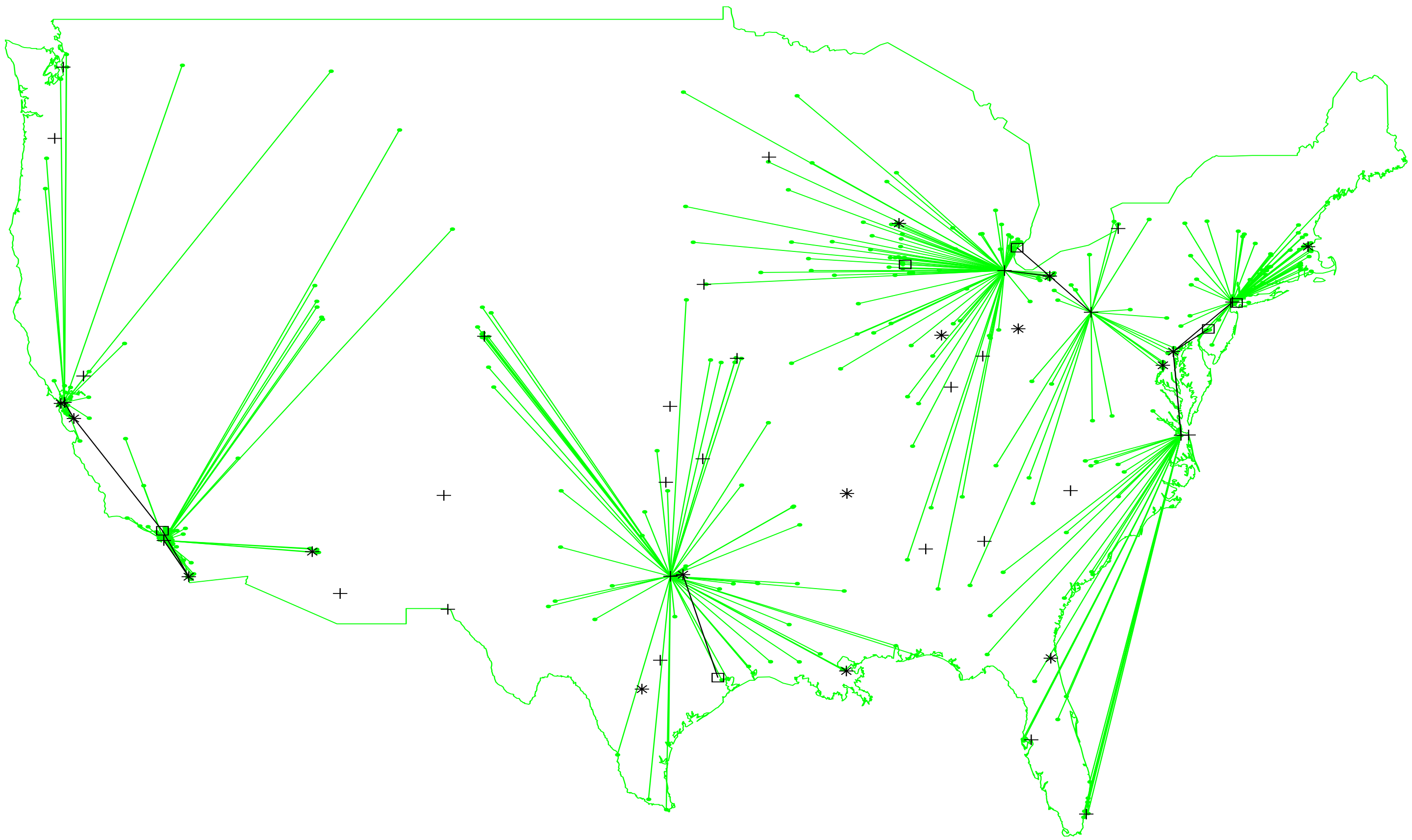

Figure 4.5: Optimal solution for `US2.6-16-30.348` instance. Cost: 294820.

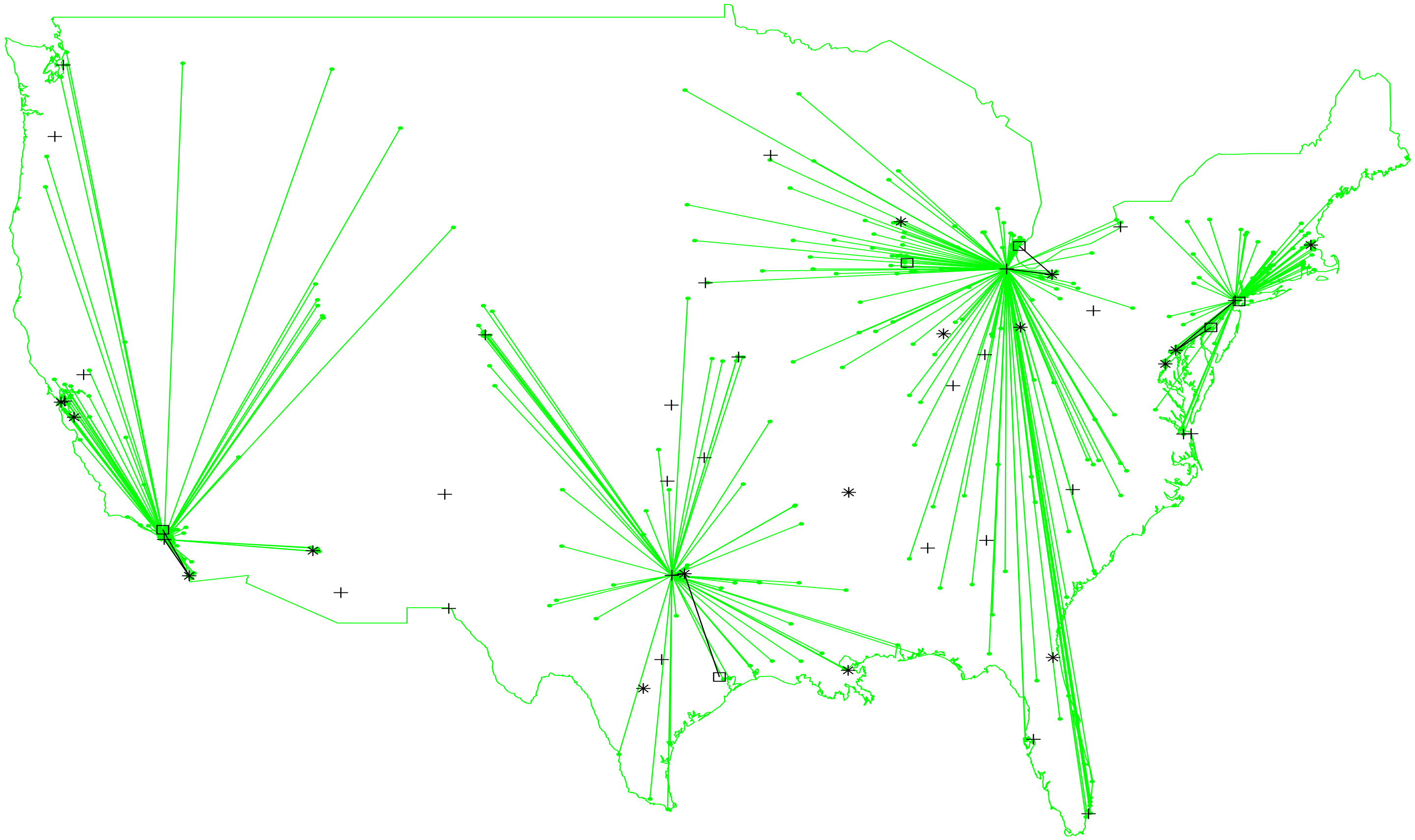

Figure 4.6: **MLRR** solution for `US2.6-16-30.348` instance. Cost: 308060.

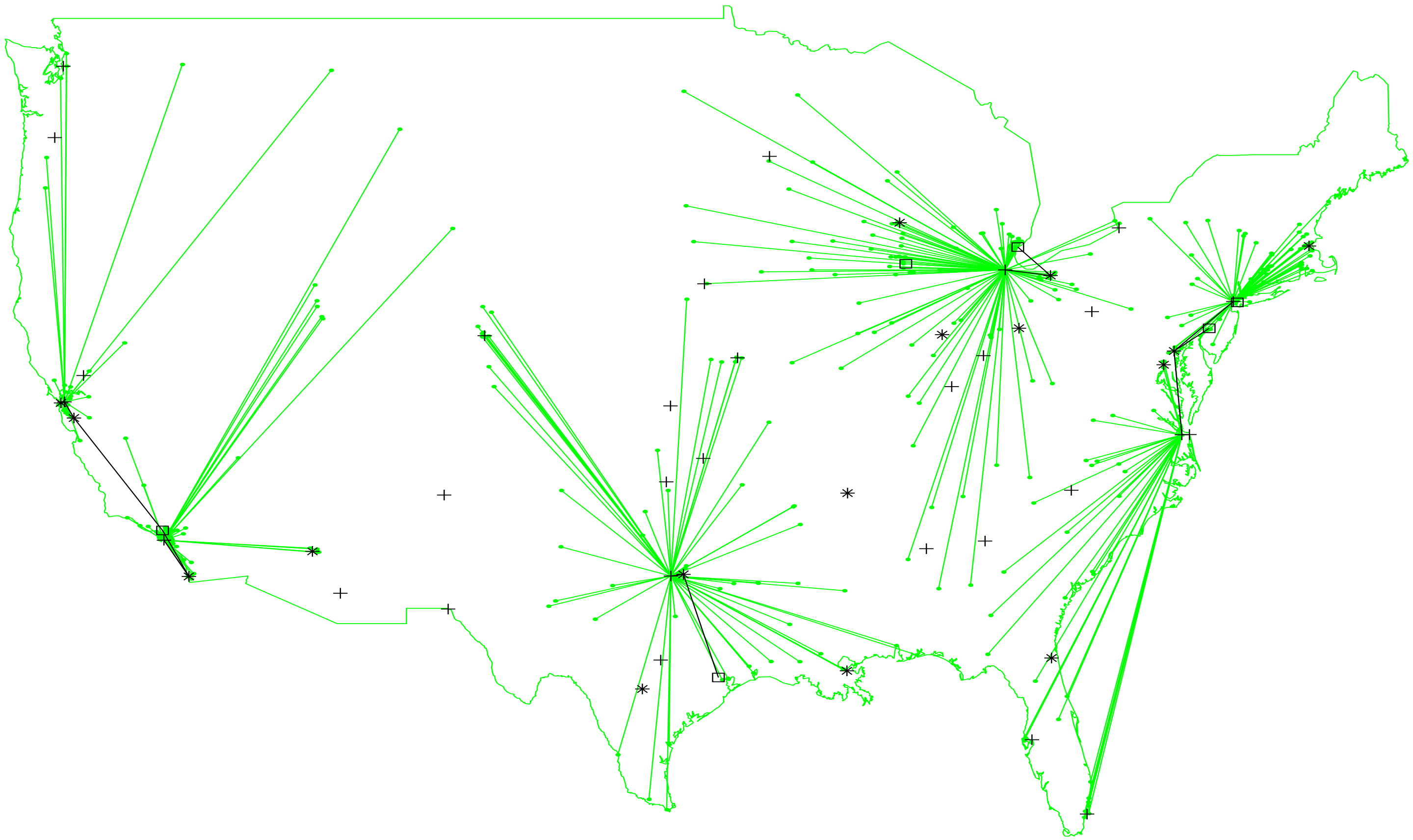

Figure 4.7: **PR-RR/PR-LI** solution for `US2.6-16-30.348` instance. Cost: 295250.

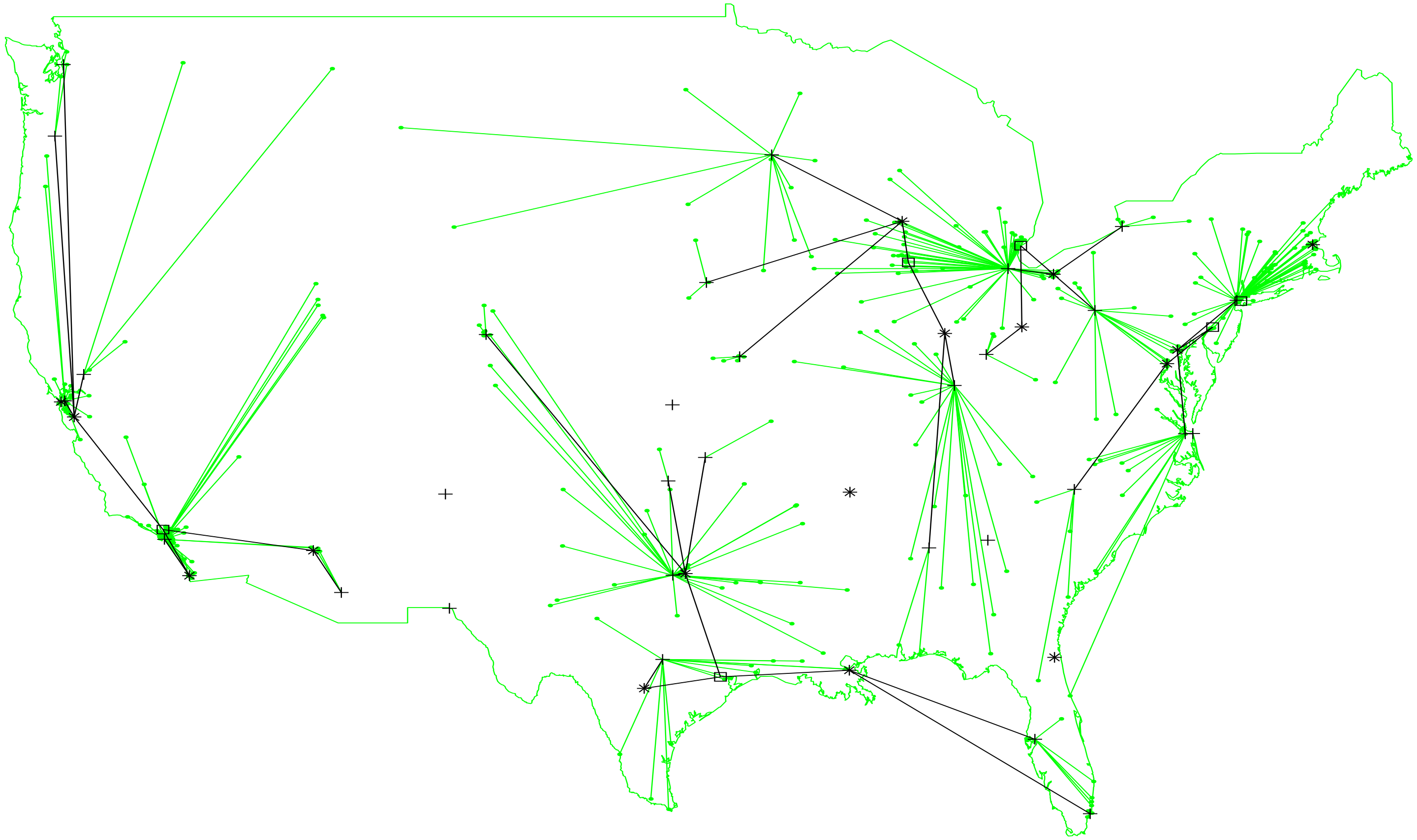

Figure 4.8: **SP** solution for `US2.6-16-30.348` instance. Cost: 317900.

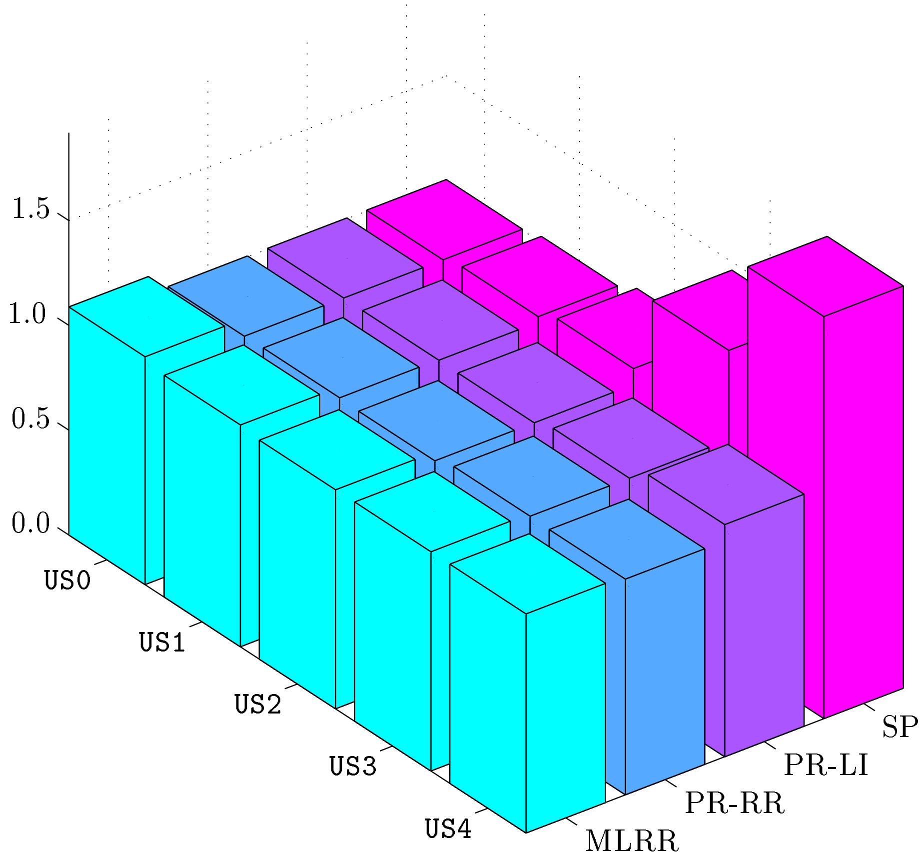

Figure 4.9: Relative performance of multi-level facility location algorithms, `USx.6-16-30.348` instances.

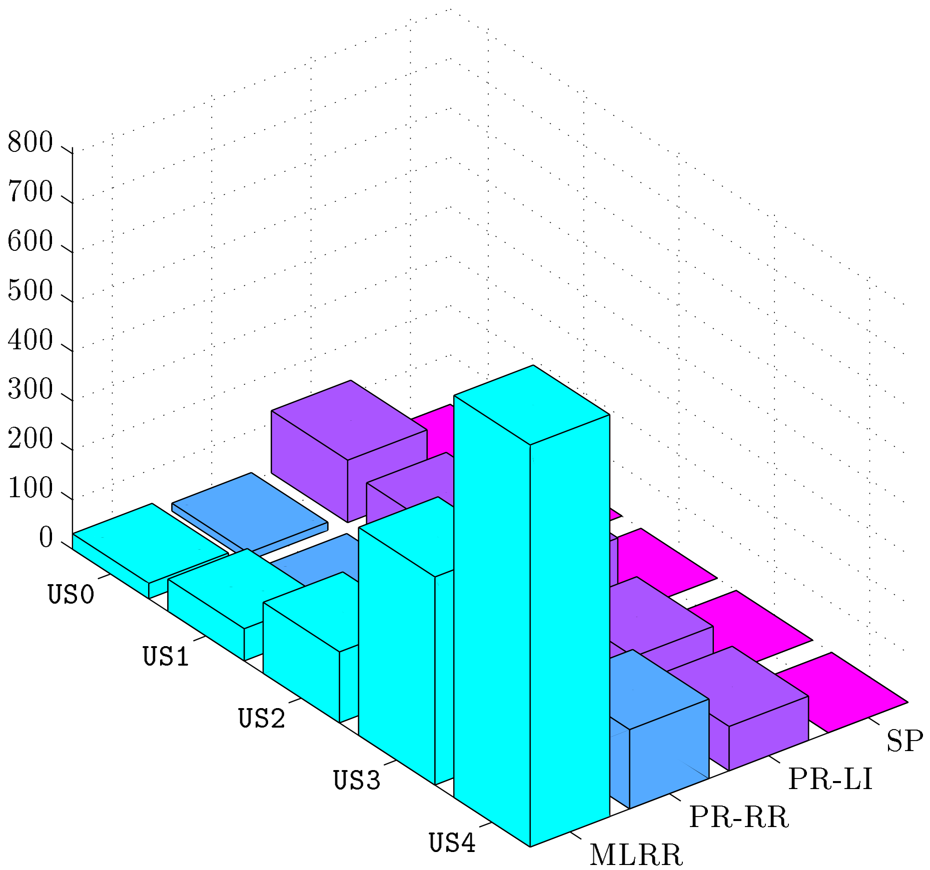

Figure 4.10: Time in seconds required by multi-level facility location algorithms, `USx.6-16-30.348` instances.

4.4 Conclusions

These computation experiments, though qualitative in nature, have clearly demonstrated the efficacy of all of the multi-level algorithms implemented. Each algorithm has contexts in which it is most effective. The multi-level linear program solution rounding algorithm of Aardal, Chudak and Shmoys [ACS99] is most appropriate when the problems are large and the facilities expensive. The path reduction approximation algorithms both perform well across the board for small to medium sized instances. Finally, the facility cost oblivious shortest path algorithm, for which we have no guarantee, performs well as long as facilities are cheap. The shortest path algorithm can also be run on instances too large for the other algorithms to handle, due to its polynomial-time running-time bound.

There is a nice complementarity between the two path reduction algorithms. As observed previously, the path reduction linear program solution rounding algorithm takes longer to run as the facility costs increase. Conversely, the path reduction local improvement algorithm takes less time to run as the facilities become more expensive.

We also notice that when there are many clients, the multi-level linear program solution rounding algorithm becomes a less viable option. However, we point out that as the number of clients increases, the service cost portion of the objective function starts to dominate, which implies that the instance might be more appropriately handled by the facility cost oblivious shortest path algorithm.

The complementary strengths of these algorithms suggests that in practice a combined approach would be very effective. Given an instance of the multi-level facility location problem, the first step should be to run the facility cost oblivious shortest path algorithm. This not only provides a feasible solution, but also a crude upper and lower bound on the value of the optimal solution. The value of the solution found by the oblivious shortest path algorithm forms the crude upper bound, and by considering just the service cost portion of the shortest path solution we have a crude lower bound. If the gap between the upper and lower bounds is acceptable, we are done. If not, we should solve ($\mathrm{LP}_{\mathrm{MLFL}}$) next. This provides a typically strong lower bound on the optimal solution, and if we are lucky, the optimal solution of ($\mathrm{LP}_{\mathrm{MLFL}}$) will actually be integral. If the optimal solution of ($\mathrm{LP}_{\mathrm{MLFL}}$) is fractional, we can run the multi-level linear program solution rounding algorithm, producing a good solution for when facility costs dominate the cost of the optimal solution. Assuming we have not struck paydirt and found the optimal solution by solving ($\mathrm{LP}_{\mathrm{MLFL}}$), we now have two solutions that are typically good in different instance domains, and a strong lower bound with which to evaluate them. At this point, if necessary, we can use the path reduction algorithms. This may be computationally expensive, but we have already verified that the neither the service or facility costs dominate, and have some idea about which of the two path reduction algorithms is likely to most appropriate. We should run the path reduction linear program solution rounding algorithm if the facilities are relatively cheap, and the path reduction local improvement algorithm if the facilities are relatively expensive. The computational experiments of the preceding section indicate that the solution resulting from this combined procedure is likely to be within a few percent of the optimal solution.

Chapter 5

Concluding Remarks

There are two schools of thought concerning the multi-level facility location generalization of the facility location problem. The first believes that the two problems are as hard, or as easy, as each other. The second school of thought believes that the multi-level version of the problem is intrinsicly more difficult than the single-level case.

Initially in the first camp, the author is a convert to the second school of thought. To prove the first camp correct, we must show an polynomial-time approximation-preserving reduction from the multi-level case to a facility location problem. To prove the second camp correct, we must demonstrate that the multi-level facility location problem is MAX-SNP-hard with respect to a constant greater than ≈ 1.728, the current best approximation constant for the facility location problem. The question of which of these two positions is provably true is an open question.

Circumstancial evidence for the first case includes the multi-level linear program solution rounding algorithm of Aardal, Chudak, and Shmoys [ACS99] which matches the approximation constant of the corresponding version of the Chudak and Shmoys [CS] algorithm for the single-level case. Circumstantial evidence for the second case includes the Steiner in-branching structure of multi-level facility location solutions, since optimizing a linear function over the convex hull Steiner in-branchings is *NP*-hard.

In practice, however, these theoretical questions matter little. More important is whether these algorithms can be used to solve real problems. While the path reduction does nothing to answer these theoretical questions, it results in algorithms that perform excellently in practice. The multi-level linear program solution rounding algorithm too, performs considerably better than its approximation guarantee of 3, even on instances for which it performs poorly in comparison to the other algorithms. This thesis has demonstrated the effectiveness of these algorithms applied to real problems.

There are many open questions that remain unanswered. It would be satisfying to know which of the two schools of thought are correct. From the effectiveness of the path reduction algorithms in the computational experiments, it seems likely that a better approximation constant could be shown for the path reduction approach. The observation that the lower bound instance for disjoint path solutions does not apply to general path solutions further suggests an improved constant for the path reduction.

We have not touched on the myriad of reasonable extensions for this problem. Foremost amongst these is the addition of capacity constraints on the facilities. Certainly, problems occurring in real life are very rarely uncapacitated, and the level hierarchy immediately suggests capacity decreasing with level. Another interesting variation moves the facility costs to the edges of the graph, suggesting the fixed charge network design problem. A further generalization removes the hierarchical structure of the graph, permitting short-cutting, or even imposing no structure at all. We would be interested to see what, if anything, could be said about these problems, using the techniques of this thesis.

Appendix A

MatCPX: A Matlab-CPLEX Interface

The MatCPX Matlab-CPLEX interface provides a way to access the power of CPLEX from Matlab. At its core, MatCPX consists of Matlab MEX functions that interface with the CPLEX callable library, but most users find the high level linear and integer program solution functions, written using these MEX files, suitable for most applications.

A.1 Linear and Integer Program Solvers

A.1.1 lp: Solve linear programs in standard form using CPLEX

Synopsis

```
[optval,optx,optpi,optslack,optdj,status] = lp(c,A,b[,CPXparam]);
```

Summary

`lp` solves the linear programing problem

$$\begin{aligned} \min\ & c'x \\ \text{s.t. } & Ax \leq b \\ & x \geq 0 \end{aligned}$$

using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c`	Matlab vector of doubles, size $n \times 1$, representing the linear program's cost vector c.
`A`	Matlab matrix of doubles, size $m \times n$, representing the linear program's constraint matrix A.
`b`	Matlab vector of doubles, size $m \times 1$, representing the linear program's right hand side vector b.

`CPXparam`	Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval`	Matlab double containing the linear program's optimal value.
`optx`	Matlab vector of doubles, size $n \times 1$, containing the linear program's optimal primal solution.
`optpi`	Matlab vector of doubles, size $m \times 1$, containing the linear program's optimal dual solution.
`optslack`	Matlab vector of doubles, size $m \times 1$, containing the slack or surplus of each constraint at the linear program's optimal solution.
`optdj`	Matlab vector of doubles, size $n \times 1$, containing the reduced cost of each variable at the linear program's optimal solution.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `lp` to solve the linear program

$$\begin{array}{rl} \min & 50x_1 + 25x_2 + 30x_3 + 80x_4 \\ \text{s.t.} & 400x_1 + 200x_2 + 150x_3 + 500x_4 \geq 500 \\ & 3x_1 + 2x_2 \geq 6 \\ & 2x_1 + 2x_2 + 4x_3 + 4x_4 \geq 15 \\ & 2x_1 + 4x_2 + x_3 + 5x_4 \geq 8 \\ & x_1, \; x_2, \; x_3, \; x_4 \geq 0 \end{array}$$

```
>> c = [ 50 25 30 80 ]';
>> A = -[
400 200 150 500
  3   2   0   0
  2   2   4   4
  2   4   1   5
];
>> b = -[ 500 6 15 8 ]';
>> [optval,optx,optpi,optslack,optdj,status] = lp(c,A,b)
No LP presolve or aggregator reductions.
Presolve time =    0.01 sec.

Iteration Log . . .
Iteration:     1    Infeasibility =             5.999998
Switched to Devex.
Iteration:     3    Objective     =           198.125000

optval =
  142.5000

optx =
         0
```

```
    3.0000
    2.2500
         0

optpi =
         0
   -5.0000
   -7.5000
         0

optslack =
  437.5000
         0
         0
    6.2500

optdj =
    20
     0
     0
    50

status =
     1

>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.OFF;
>> optval = lp(c,A,b,CPXparam)

optval =
  142.5000
```

A.1.2 mip: Solve mixed integer programs in standard form using CPLEX

Synopsis

```
[optval,optx,status] = mip(c,A,b,I[,CPXparam]);
```

Summary

`mip` solves the mixed integer programing problem

$$
\begin{aligned}
\min\ & c'x \\
\text{s.t. } Ax &\le b \\
x &\ge 0 \\
x_i &\text{ integer, } i \in I.
\end{aligned}
$$

using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c` Matlab vector of doubles, size $n \times 1$, representing the mixed integer program's cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the mixed integer program's constraint matrix A.

`b` Matlab vector of doubles, size $m \times 1$, representing the mixed integer program's right hand side vector b.

`I` Matlab array of doubles containing the indices of the integer variables I.

`CPXparam` Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval` Matlab double containing the mixed integer program's optimal value.

`optx` Matlab vector of doubles, size $n \times 1$, containing the mixed integer program's optimal solution.

`status` Matlab double containing the termination status returned by CPLEX.

Discussion

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `mip` to solve the mixed integer program

$$
\begin{array}{rrrrrl}
\min & 50x_1+ & 25x_2+ & 30x_3+ & 80x_4 & \\
\text{s.t.} & 400x_1+ & 200x_2+ & 150x_3+ & 500x_4 & \geq 500 \\
 & 3x_1+ & 2x_2 & & & \geq 6 \\
 & 2x_1+ & 2x_2+ & 4x_3+ & 4x_4 & \geq 15 \\
 & 2x_1+ & 4x_2+ & x_3+ & 5x_4 & \geq 8 \\
 & x_1, & x_2, & x_3, & x_4 & \geq 0 \\
 & & x_3 & \text{integer.} & &
\end{array}
$$

```
>> c = [ 50 25 30 80 ]';
>> A = -[
400 200 150 500
  3   2   0   0
  2   2   4   4
  2   4   1   5
];
>> b = -[ 500 6 15 8 ]';
>> I = [ 3 ];
>> [optval,optx,status] = mip(c,A,b,I)
No MIP presolve or aggregator reductions.
Presolve time =    0.00 sec.

        Nodes                                             Cuts/
   Node  Left   Objective  IInf  Best Integer     Best Node  ItCnt

      0     0    142.5000     1                    142.5000      5
*     1     1    165.0000     0      165.0000      147.5000      6
*     2     0    147.5000     0      147.5000                    7
```

```
optval =
  147.5000

optx =
         0
    3.5000
    2.0000
         0

status =
   101

>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.OFF;
>> optval = mip(c,A,b,I,CPXparam)

optval =
  147.5000
```

A.1.3 ip: Solve integer programs in standard form using CPLEX

Synopsis

```
[optval,optx,status] = ip(c,A,b[,CPXparam]);
```

Summary

`ip` solves the integer programing problem

$$\begin{aligned} \min\ & c'x \\ \text{s.t. } & Ax \leq b \\ & x \geq 0 \\ & x \text{ integer.} \end{aligned}$$

using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c` Matlab vector of doubles, size $n \times 1$, representing the integer program's cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the integer program's constraint matrix A.

`b` Matlab vector of doubles, size $m \times 1$, representing the integer program's right hand side vector b.

`CPXparam`	Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval`	Matlab double containing the integer program's optimal value.
`optx`	Matlab vector of doubles, size $n \times 1$, containing the integer program's optimal solution.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use ip to solve the integer program

$$\begin{array}{rrrrrl}
\min & 50x_1+ & 25x_2+ & 30x_3+ & 80x_4 & \\
\text{s.t.} & 400x_1+ & 200x_2+ & 150x_3+ & 500x_4 & \geq 500 \\
& 3x_1+ & 2x_2 & & & \geq 6 \\
& 2x_1+ & 2x_2+ & 4x_3+ & 4x_4 & \geq 15 \\
& 2x_1+ & 4x_2+ & x_3+ & 5x_4 & \geq 8 \\
& x_1, & x_2, & x_3, & x_4 & \geq 0 \\
& & x & \text{integer.} & &
\end{array}$$

```
>> c = [ 50 25 30 80 ]';
>> A = -[
400 200 150 500
  3   2   0   0
  2   2   4   4
  2   4   1   5
];
>> b = -[ 500 6 15 8 ]';
>> [optval,optx,status] = ip(c,A,b)
No MIP presolve or aggregator reductions.
Presolve time =    0.00 sec.

        Nodes                                                Cuts/
   Node  Left    Objective  IInf  Best Integer     Best Node  ItCnt

      0     0     142.5000     1                    142.5000      5
*     1     1     165.0000     0      165.0000      147.5000      6
*     3     0     160.0000     0      160.0000                    8

optval =
```

```
   160

optx =
     0
     4
     2
     0

status =
   101

>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.OFF;
>> [optval,optx,status] = ip(c,A,b,CPXparam)

optval =
   160
```

A.1.4 lin_prog: Solve linear programs using CPLEX

Synopsis

```
[optval,optx,optpi,optslack,optdj,status] = ...
                    lin_prog(c,A,sense,b,lb,ub[,CPXparam]);
```

Summary

`lin_prog` solves the linear programing problem

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad Ax & \begin{bmatrix} \leq \\ = \\ \geq \end{bmatrix} b \\ & l \leq x \leq u \end{aligned}$$

using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c` Matlab vector of doubles, size $n \times 1$, representing the linear program's cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the linear program's constraint matrix A.

`sense` Matlab char vector, size $m \times 1$, representing the sense of each of the linear program's constraints. `sense(i)` is one of `'L'`, `'E'`, or `'G'`, representing less than or equal to, equal to, or greater than or equal to, respectively. Note that case is significant!

`b` Matlab vector of doubles, size $m \times 1$, representing the linear program's right hand side vector b.

`lb` Matlab vector of doubles, size $n \times 1$, representing the linear program's lower bound l on the variables. Can be the empty matrix `[]`, indicating no lower bounds. Individual lower bounds can be set to `-inf` to represent no lower bound.

`ub` Matlab vector of doubles, size $n \times 1$, representing the linear program's upper bound u on the variables. Can be the empty matrix `[]`, indicating no upper bounds. Individual upper bounds can be set to `inf` to represent no upper bound.

`CPXparam` Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval` Matlab double containing the linear program's optimal value.

`optx` Matlab vector of doubles, size $n \times 1$, containing the linear program's optimal primal solution.

`optpi` Matlab vector of doubles, size $m \times 1$, containing the linear program's optimal dual solution.

`optslack`	Matlab vector of doubles, size $m \times 1$, containing the slack or surplus of each constraint at the linear program's optimal solution.
`optdj`	Matlab vector of doubles, size $n \times 1$, containing the reduced cost of each variable at the linear program's optimal solution.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `lin_prog` to solve the linear program

$$\begin{aligned} \min \ & 3x_1 + 4x_2 - x_3 \\ \text{s.t.} \ & x_1 + x_2 + x_3 \geq 5.4 \\ & 5x_1 - x_2 + 3x_3 = 3.1 \\ & x_1, \quad x_2, \quad x_3 \geq 0 \end{aligned}$$

```
>> c = [ 3 4 -1 ]';
>> A = [
1  1 1
5 -1 3
];
>> sense = [ 'G' 'E' ]';
>> b = [ 5.4 3.1 ]';
>> lb = zeros(3,1);
>> [optval,optx,optpi,optslack,optdj,status] = ...
lin_prog(c,A,sense,b,lb,[])
No LP presolve or aggregator reductions.
Presolve time =    0.00 sec.

Iteration Log . . .
Iteration:     1    Infeasibility =             0.000000
Switched to Devex.

optval =
   10.9750

optx =
         0
    3.2750
    2.1250

optpi =
    2.7500
   -1.2500

optslack =
     0
     0

optdj =
    6.5000
         0
         0

status =
     1
```

```
>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.OFF;
>> optval = lin_prog(c,A,sense,b,lb,[],CPXparam)

optval =
   10.9750
```

A.1.5 mixed_int_prog: Solve mixed integer programs using CPLEX

Synopsis

```
[optval,optx,status] = ...
                    mixed_int_prog(c,A,sense,b,lb,ub,type[,CPXparam]);
```

Summary

`mixed_int_prog` solves the mixed integer programing problem

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad & Ax \begin{bmatrix} \leq \\ = \\ \geq \end{bmatrix} b \\ & l \leq x \leq u \\ & x_i \text{ integer, } i \in I \\ & x_i \text{ binary, } i \in B. \end{aligned}$$

using the CPLEX callable library via the MatCPX interface.

Input Arguments

c	Matlab vector of doubles, size $n \times 1$, representing the linear program's cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the linear program's constraint matrix A.

`sense` Matlab char vector, size $m \times 1$, representing the sense of each of the linear program's constraints. `sense(i)` is one of `'L'`, `'E'`, or `'G'`, representing less than or equal to, equal to, or greater than or equal to, respectively. Note that case is significant!

`b` Matlab vector of doubles, size $m \times 1$, representing the linear program's right hand side vector b.

`lb` Matlab vector of doubles, size $n \times 1$, representing the linear program's lower bound l on the variables. Can be the empty matrix `[]`, indicating no lower bounds. Individual lower bounds can be set to `-inf` to represent no lower bound.

`ub` Matlab vector of doubles, size $n \times 1$, representing the linear program's upper bound u on the variables. Can be the empty matrix `[]`, indicating no upper bounds. Individual upper bounds can be set to `inf` to represent no upper bound.

`type` Matlab char vector, size $n \times 1$, representing the type of each variable. `type(i)` is one of `'C'`, `'I'`, or `'B'`, representing continuous, integer, or binary, respectively. Note that CPLEX requires binary variables to be enforced by constraints. Note too that case is significant!

`CPXparam` Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval`	Matlab double containing the mixed integer program's optimal value.
`optx`	Matlab vector of doubles, size $n \times 1$, containing the mixed integer program's optimal solution.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `mixed_int_prog` to solve the mixed integer program

$$\begin{aligned}
\min\ & 3x_1+4x_2-\ x_3 \\
\text{s.t.}\ & x_1+\ x_2+\ x_3 \geq 5.4 \\
& 5x_1-\ x_2+3x_3 = 3.1 \\
& x_1,\quad x_2,\quad x_3 \geq 0 \\
& x_1 \text{ integer} \\
& x_3 \in \{0,1\}
\end{aligned}$$

```
>> c = [ 3 4 -1 ]';
>> A = [
1  1 1
5 -1 3
];
>> sense = [ 'G' 'E' ]';
>> b = [ 5.4 3.1 ]';
>> lb = zeros(3,1);
>> ub = [ inf inf 1]';
>> type = [ 'I' 'C' 'B' ]';
>> [optval,optx,status] = mixed_int_prog(c,A,sense,b,lb,ub,type)
No MIP presolve or aggregator reductions.
Presolve time =    0.00 sec.

        Nodes                                               Cuts/
   Node  Left   Objective  IInf  Best Integer     Best Node  ItCnt

      0     0     15.8500     1                     15.8500      2
*     2     1     21.6000     0       21.6000       20.1833      4

optval =
   21.6000
```

```
optx =
    1.0000
    4.9000
    1.0000

status =
   101

>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.OFF;
>> optval = mixed_int_prog(c,A,sense,b,lb,ub,type,CPXparam)

optval =
   21.6000
```

A.1.6 lin_prog_cg: Solve linear programs by column generation using CPLEX

Synopsis

```
[optval,optx,optpi,optslack,optdj,AUXparam,status] = ...
   lin_prog_cg(c,A,sense,b,lb,ub,CGfunc,CGparam,AUXparam[,CPXparam]);
```

Summary

`lin_prog_cg` solves the linear programing problem

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad & Ax \begin{bmatrix} \leq \\ = \\ \geq \end{bmatrix} b \\ & l \leq x \leq u \end{aligned}$$

by a column generation procedure using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c` Matlab vector of doubles, size $n \times 1$, representing the linear program's initial cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the linear program's initial constraint matrix A.

`sense` Matlab char vector, size $m \times 1$, representing the sense of each of the linear program's constraints. `sense(i)` is one of `'L'`, `'E'`, or `'G'`, representing less than or equal to, equal to, or greater than or equal to, respectively. Note that case is significant!

`b` Matlab vector of doubles, size $m \times 1$, representing the linear program's right hand side vector b.

`lb` Matlab vector of doubles, size $n \times 1$, representing the linear program's initial lower bound vector l. Can be the empty matrix `[]`, indicating no lower bounds. Individual lower bounds can be set to `-inf` to represent no lower bound.

`ub` Matlab vector of doubles, size $n \times 1$, representing the linear program's initial upper bound vector u. Can be the empty matrix `[]`, indicating no upper bounds. Individual upper bounds can be set to `inf` to represent no upper bound.

`CGfunc` Matlab string specifying the Matlab m-file that determines if the current solution is optimal for the linear program. See below.

`CGparam` Matlab struct containing parameters for the column generation procedure. See below.

`AUXparam` Matlab variable (usually a struct) for passing user data into `CGfunc`, and for preserving state information between calls of `CGfunc`. See below.

`CPXparam` Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval`	Matlab double containing the linear program's optimal value.
`optx`	Matlab vector of doubles, size $n \times 1$, containing the linear program's optimal primal solution.
`optpi`	Matlab vector of doubles, size $m \times 1$, containing the linear program's optimal dual solution.
`optslack`	Matlab vector of doubles, size $m \times 1$, containing the slack or surplus of each constraint at the linear program's optimal solution.
`optdj`	Matlab vector of doubles, size $n \times 1$, containing the reduced cost of each variable at the linear program's optimal solution.
`AUXparam`	Matlab variable (usually a struct) for passing user data out of `CGfunc`, and for preserving state information between calls of `CGfunc`. See below.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

The Matlab m-file `CGfunc` determines if the current solution, passed in as input, is optimal for the linear program. If not, it must return the data for additional variables to add to the linear program. `CGfunc` must have the signature:

```
function [isopt,c1,A1,lb1,ub1,AUXparam] = ...
                          CGfunc(optval,optx,optpi,optslack,optdj,AUXparam)
```

The input arguments `optval`, `optx`, `optpi`, `optslack`, and `optdj` are the optimal solution to the current linear program, which may or may not represent an optimal solution for the linear program we really want to solve. `AUXparam` contains user data passed into `lin_prog_cg` and (possibly) changed in subsequent calls to `CGfunc`.

The output argument `isopt` should be 1 if the input values represent an optimal solution to the linear program we really want to solve and 0 otherwise; the remaining arguments supply the additional data to add to the linear program if the solution is not optimal. `c1` is a Matlab vector of doubles containing the objective function coefficients of the new variables to be added, `A1` is a Matlab matrix of doubles containing the additional columns to add to the constraint matrix, and `lb1` and `ub1` are Matlab vectors of doubles containing the lower and upper bounds on the new variables. To avoid a warning from Matlab, all output arguments should be set to some value, even if the current solution is optimal. The empty matrix `[]` is sufficient for this purpose. Changes made to `AUXparam` will be preserved across calls to `CGfunc` and returned at the end of the optimization.

`CGparam` is a struct array consisting of parameters for the column generation algorithm itself. There are two required fields for `CGparam`, `nmax` and `nzmax`.

`nmax` Maximum number of variables this linear program will require to solve to optimality.

`nzmax` Maximum number of non-zeros in the constraint matrix this linear program will require to solve to optimality.

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `lin_prog_cg` to solve the following linear program by column generation.

$$\begin{array}{rrrrrl}
\min & 50x_1+ & 25x_2+ & 30x_3+ & 80x_4 & \\
\text{s.t.} & 400x_1+ & 200x_2+ & 150x_3+ & 500x_4 & \geq 500 \\
 & 3x_1+ & 2x_2 & & & \geq 6 \\
 & 2x_1+ & 2x_2+ & 4x_3+ & 4x_4 & \geq 15 \\
 & 2x_1+ & 4x_2+ & x_3+ & 5x_4 & \geq 8 \\
 & x_1, & x_2, & x_3, & x_4 & \geq 0
\end{array}$$

We first create our column generation m-file, `cgen.m`. For this simple example, our initial linear program contains just the variable x_1 and `cgen` adds the other variables one at a time.

```
function [isopt,c1,A1,lb1,ub1,AUXparam] = ...
                    cgen(optval,optx,optpi,optslack,optdj,AUXparam)
optval
switch AUXparam.iteration
```

```
case 0,
   isopt = 0;
   c1 = 25; lb1 = 0; ub1 = [];
   A1 = [ 200 2 2 4 ]';
case 1,
   isopt = 0;
   c1 = 30; lb1 = 0; ub1 = [];
   A1 = [ 150 0 4 1 ]';
case 2,
   isopt = 0;
   c1 = 80; lb1 = 0; ub1 = [];
   A1 = [ 500 0 4 5 ]';
otherwise,
   isopt = 1;
   c1 = []; lb1 = []; ub1 = [];
   A1 = [];
end
AUXparam.iteration = AUXparam.iteration + 1;
```

We can now use `lin_prog_cg` to solve our example linear program.

```
>> c = 50; lb = 0;
>> A = [ 400 3 2 2 ]';
>> sense = [ 'G' 'G' 'G' 'G']';
>> b = [ 500 6 15 8 ]';
>> CGparam.nmax = 4;
>> CGparam.nzmax = 14;
>> AUXparam.iteration = 0;
>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.ON;
>> [optval,optx,optpi,optslack,optdj,AUXparam,status] = ...
lin_prog_cg(c,A,sense,b,lb,[],'cgen',CGparam,AUXparam,CPXparam)
Tried aggregator 1 time.
LP Presolve eliminated 4 rows and 1 columns.
All rows and columns eliminated.
Presolve time =    0.00 sec.

optval =
```

```
   375

Using devex.

Iteration log . . .
Iteration:     1    Objective     =           187.500000

optval =
  187.5000

Using devex.

Iteration log . . .
Iteration:     1    Objective     =           142.500000

optval =
  142.5000

Using devex.

optval =
  142.5000

optval =
  142.5000

optx =
         0
    3.0000
    2.2500
         0

optpi =
         0
    5.0000
    7.5000
         0

optslack =
 -437.5000
```

```
         0
         0
   -6.2500

optdj =
    20
     0
     0
    50

AUXparam =
    iteration: 4

status =
     1
```

A.1.7 lin_prog_rg: Solve linear programs by row generation using CPLEX

Synopsis

```
[optval,optx,optpi,optslack,optdj,AUXparam,status] = ...
   lin_prog_rg(c,A,sense,b,lb,ub,RGfunc,RGparam,AUXparam[,CPXparam]);
```

Summary

`lin_prog_rg` solves the linear programing problem

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad & Ax \begin{bmatrix} \le \\ = \\ \ge \end{bmatrix} b \\ & l \le x \le u \end{aligned}$$

by a row generation procedure using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c` Matlab vector of doubles, size $n \times 1$, representing the linear program's cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the linear program's initial constraint matrix A.

`sense` Matlab char vector, size $m \times 1$, representing the sense of each of the linear program's initial constraints. `sense(i)` is one of `'L'`, `'E'`, or `'G'`, representing less than or equal to, equal to, or greater than or equal to, respectively. Note that case is significant!

`b` Matlab vector of doubles, size $m \times 1$, representing the linear program's initial right hand side vector b.

`lb` Matlab vector of doubles, size $n \times 1$, representing the linear program's lower bound vector l. Can be the empty matrix `[]`, indicating no lower bounds. Individual lower bounds can be set to `-inf` to represent no lower bound.

`ub` Matlab vector of doubles, size $n \times 1$, representing the linear program's upper bound vector u. Can be the empty matrix `[]`, indicating no upper bounds. Individual upper bounds can be set to `inf` to represent no upper bound.

`RGfunc` Matlab string specifying the Matlab m-file that determines if the current solution is feasible for the linear program. See below.

`RGparam` Matlab struct containing parameters for the row generation procedure. See below.

`AUXparam` Matlab variable (usually a struct) for passing user data into `RGfunc`, and for preserving state information between calls of `RGfunc`. See below.

`CPXparam` Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval`	Matlab double containing the linear program's optimal value.
`optx`	Matlab vector of doubles, size $n \times 1$, containing the linear program's optimal primal solution.
`optpi`	Matlab vector of doubles, size $m \times 1$, containing the linear program's optimal dual solution.
`optslack`	Matlab vector of doubles, size $m \times 1$, containing the slack or surplus of each constraint at the linear program's optimal solution.
`optdj`	Matlab vector of doubles, size $n \times 1$, containing the reduced cost of each variable at the linear program's optimal solution.
`AUXparam`	Matlab variable (usually a struct) for passing user data out of `RGfunc`, and for preserving state information between calls of `RGfunc`. See below.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

The Matlab m-file `RGfunc` determines if the current solution, passed in as input, is feasible for the linear program. If not, it must return the data for additional constraints to add to the linear program. `RGfunc` must have the signature:

```
function [isfeas,A1,s1,b1,AUXparam] = ...
                        RGfunc(optval,optx,optpi,optslack,optdj,AUXparam)
```

The input arguments `optval`, `optx`, `optpi`, `optslack`, and `optdj` are the optimal solution to the current linear program, which may or may not represent a feasible solution for the linear program we really want to solve. `AUXparam` contains user data passed into `lin_prog_rg` and (possibly) changed in subsequent calls to `RGfunc`.

The output argument `isfeas` should be 1 if the input values represent a feasible solution to the linear program we really want to solve and 0 otherwise; the remaining arguments supply the additional data to add to the linear program if the solution is not feasible. `A1` is a Matlab matrix of doubles containing the additional rows to add to the constraint matrix, `s1` is a Matlab char vector containing the sense of the additional rows, and `b1` is a Matlab vector of doubles containing the right hand sides of the additional rows. To avoid a warning from Matlab, all output arguments should be set to some value, even if the current solution is feasible. The empty matrix `[]` is sufficient for this purpose. Changes made to `AUXparam` will be preserved across calls to `RGfunc` and returned at the end of the optimization.

`RGparam` is a struct array consisting of parameters for the row generation algorithm itself. There are two required fields for `RGparam`, `mmax` and `nzmax`.

`mmax` Maximum number of constraints this linear program will require to achieve feasibility.

`nzmax` Maximum number of non-zeros in the constraint matrix this linear program will require to achieve feasibility.

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of 'integer' or 'double'; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `lin_prog_rg` to solve the following linear program by row generation.

$$
\begin{array}{rrrrrl}
\min & 50x_1+ & 25x_2+ & 30x_3+ & 80x_4 & \\
\text{s.t.} & 400x_1+ & 200x_2+ & 150x_3+ & 500x_4 & \geq 500 \\
 & 3x_1+ & 2x_2 & & & \geq 6 \\
 & 2x_1+ & 2x_2+ & 4x_3+ & 4x_4 & \geq 15 \\
 & 2x_1+ & 4x_2+ & x_3+ & 5x_4 & \geq 8 \\
 & x_1, & x_2, & x_3, & x_4 & \geq 0
\end{array}
$$

We first create our row generation m-file, `rgen.m`. For this simple example, our initial linear program contains just the first constraint and `rgen` adds the other constraints one at a time.

```
function [isfeas,A1,s1,b1,AUXparam] = ...
                     rgen(optval,optx,optpi,optslack,optdj,AUXparam)
optval
switch AUXparam.iteration
case 0,
   isfeas = 0;
   s1 = 'G'; b1 = 6;
   A1 = [ 3 2 0 0 ];
```

```
case 1,
   isfeas = 0;
   s1 = 'G'; b1 = 15;
   A1 = [ 2 2 4 4 ];
case 2,
   isfeas = 0;
   s1 = 'G'; b1 = 8;
   A1 = [ 2 4 1 5 ];
otherwise,
   isfeas = 1;
   s1 =[]; b1 = [];
   A1 = [];
end
AUXparam.iteration = AUXparam.iteration + 1;
```

We can now use `lin_prog_rg` to solve our example linear program.

```
>> c = [ 50 25 30 80 ]'; lb = 0;
>> A = [ 400 200 150 500 ];
>> sense = [ 'G' ];
>> b = [ 500 ];
>> RGparam.mmax = 4;
>> RGparam.nzmax = 14;
>> AUXparam.iteration = 0;
>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.ON;
>> [optval,optx,optpi,optslack,optdj,AUXparam,status] = ...
lin_prog_rg(c,A,sense,b,lb,[],'rgen',RGparam,AUXparam,CPXparam)
Tried aggregator 1 time.
LP Presolve eliminated 1 rows and 4 columns.
All rows and columns eliminated.
Presolve time =    0.00 sec.

optval =
   62.5000

Iteration log . . .
Iteration:     1    Infeasibility =             0.000000
```

```
Switched to devex.

optval =
    75

Iteration log . . .
Iteration:     1    Infeasibility =             0.000000
Switched to devex.

optval =
  142.5000

Using devex.

optval =
  142.5000

optval =
  142.5000

optx =
         0
    3.0000
    2.2500
         0

optpi =
         0
    5.0000
    7.5000
         0

optslack =
 -437.5000
         0
         0
   -6.2500

optdj =
    20
```

```
     0
     0
   -30

AUXparam =
    iteration: 4

status =
     1
```

A.1.8 lin_prog_rcg: Solve linear programs by row and column generation using CPLEX

Synopsis

```
[optval,optx,optpi,optslack,optdj,AUXparam,status] = ...
 lin_prog_rcg(c,A,sense,b,lb,ub,RCGfunc,RCGparam,AUXparam[,CPXparam]);
```

Summary

`lin_prog_rcg` solves the linear programing problem

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad & Ax \begin{bmatrix} \leq \\ = \\ \geq \end{bmatrix} b \\ & l \leq x \leq u \end{aligned}$$

by a row and column generation procedure using the CPLEX callable library via the MatCPX interface.

Input Arguments

`c` Matlab vector of doubles, size $n \times 1$, representing the linear program's initial cost vector c.

`A` Matlab matrix of doubles, size $m \times n$, representing the linear program's initial constraint matrix A.

`sense` Matlab char vector, size $m \times 1$, representing the sense of each of the linear program's initial constraints. `sense(i)` is one of `'L'`, `'E'`, or `'G'`, representing less than or equal to, equal to, or greater than or equal to, respectively. Note that case is significant!

`b` Matlab vector of doubles, size $m \times 1$, representing the linear program's initial right hand side vector b.

`lb` Matlab vector of doubles, size $n \times 1$, representing the linear program's initial lower bound vector l. Can be the empty matrix `[]`, indicating no lower bounds. Individual lower bounds can be set to `-inf` to represent no lower bound.

`ub` Matlab vector of doubles, size $n \times 1$, representing the linear program's initial upper bound vector u. Can be the empty matrix `[]`, indicating no upper bounds. Individual upper bounds can be set to `inf` to represent no upper bound.

`RCGfunc` Matlab string specifying the Matlab m-file that determines if the current solution is optimal and feasible for the linear program. See below.

`RCGparam` Matlab struct containing parameters for the row and column generation procedure. See below.

`AUXparam` Matlab variable (usually a struct) for passing user data into `RCGfunc`, and for preserving state information between calls of `RCGfunc`. See below.

`CPXparam` Matlab struct array of CPLEX runtime parameters (optional). See below.

Output Arguments

`optval`	Matlab double containing the linear program's optimal value.
`optx`	Matlab vector of doubles, size $n \times 1$, containing the linear program's optimal primal solution.
`optpi`	Matlab vector of doubles, size $m \times 1$, containing the linear program's optimal dual solution.
`optslack`	Matlab vector of doubles, size $m \times 1$, containing the slack or surplus of each constraint at the linear program's optimal solution.
`optdj`	Matlab vector of doubles, size $n \times 1$, containing the reduced cost of each variable at the linear program's optimal solution.
`AUXparam`	Matlab variable (usually a struct) for passing user data out of `RCGfunc`, and for preserving state information between calls of `RCGfunc`. See below.
`status`	Matlab double containing the termination status returned by CPLEX.

Discussion

The Matlab m-file `RCGfunc` determines if the current solution, passed in as input, is optimal and feasible for the linear program. If not, it must return the data for additional variables and constraints to add to the linear program. `RCGfunc` must have the signature:

```
function [isopt,c1,rA1,cA1,s1,b1,lb1,ub1,AUXparam] = ...
                        RCGfunc(optval,optx,optpi,optslack,optdj,AUXparam)
```

The input arguments `optval`, `optx`, `optpi`, `optslack`, and `optdj` are the optimal solution to the current linear program, which may or may not represent an optimal or feasible solution for the linear program we really want to solve. `AUXparam` contains user data passed into `lin_prog_rcg` and (possibly) changed in subsequent calls to `RCGfunc`.

The output argument `isopt` should be 1 if the input values represent a feasible optimal solution to the linear program we really want to solve and 0 otherwise; the remaining arguments supply the additional data to add to the linear program if the solution is not optimal or not feasible. `c1` is a Matlab vector of doubles containing the objective function coefficients of the new variables to be added, `rA1` is a Matlab matrix of doubles containing the additional rows to add to the constraint matrix, `cA1` is a Matlab matrix of doubles containing the additional columns to add to the constraint matrix, `s1` is a Matlab char vector containing the sense of the additional rows, `b1` is a Matlab vector of doubles containing the right hand sides of the additional rows, and `lb1` and `ub1` are Matlab vectors of doubles containing the lower and upper bounds on the new variables. To avoid a warning from Matlab, all output arguments should be set to some value, even if the current solution is optimal and feasible. The empty matrix `[]` is sufficient for this purpose. Changes made to `AUXparam` will be preserved across calls to `RCGfunc` and returned at the end of the optimization.

`RCGparam` is a struct array consisting of parameters for the column generation algorithm itself. There are three required fields for `RCGparam`, `nmax`, `mmax` and `nzmax`.

`nmax` Maximum number of variables this linear program will require to solve to optimality.

`mmax` Maximum number of constraints this linear program will require to solve to optimality.

`nzmax` Maximum number of non-zeros in the constraint matrix this linear program will require to solve to optimality.

`CPXparam` is an optional Matlab struct array of CPLEX runtime parameter settings. For each CPLEX struct array element `i`, `CPXparam(i).type` is one of `'integer'` or `'double'`; `CPXparam(i).param` is the CPLEX parameter to be set, either as a number or a CPLEX symbolic constant; and `CPXparam(i).value` is the value to assign to the CPLEX runtime parameter. Note that before using a CPLEX symbolic constant, the MatCPX function `MatCPXconst` must be called. The CPLEX documentation contains a list of available runtime parameters.

Example

We use `lin_prog_rcg` to solve the following linear program by row and column generation

$$
\begin{array}{rrrrrl}
\min & 50x_1+ & 25x_2+ & 30x_3+ & 80x_4 & \\
\text{s.t.} & 400x_1+ & 200x_2+ & 150x_3+ & 500x_4 & \geq 500 \\
 & 3x_1+ & 2x_2 & & & \geq 6 \\
 & 2x_1+ & 2x_2+ & 4x_3+ & 4x_4 & \geq 15 \\
 & 2x_1+ & 4x_2+ & x_3+ & 5x_4 & \geq 8 \\
 & x_1, & x_2, & x_3, & x_4 & \geq 0
\end{array}
$$

We first create our row/column generation m-file, `rcgen.m`. For this simple example, our initial linear program contains just the variable x_1 and the first constraint and `rcgen` adds one variable and one constraint each time it is called.

```
function [isopt,c1,rA1,cA1,s1,b1,lb1,ub1,AUXparam] = ...
                    rcgen(optval,optx,optpi,optslack,optdj,AUXparam)
optval
switch AUXparam.iteration
case 0,
   isopt = 0;
   c1 = 25; lb1 = 0; ub1 = [];
   cA1 = [ 200 ];
   s1 = 'G'; b1 = 6;
   rA1 = [ 3 2 ];
case 1,
   isopt = 0;
   c1 = 30; lb1 = 0; ub1 = [];
   cA1 = [ 150 0 ]';
   s1 = 'G'; b1 = 15;
   rA1 = [ 2 2 4 ];
```

```
case 2,
   isopt = 0;
   c1 = 80; lb1 = 0; ub1 = [];
   cA1 = [ 500 0 4 ]';
   s1 = 'G'; b1 = 8;
   rA1 = [ 2 4 1 5 ];
otherwise,
   isopt = 1;
   c1 = []; lb1 = []; ub1 = [];
   cA1 = [];
   s1 =[]; b1 = [];
   rA1 = [];
end
AUXparam.iteration = AUXparam.iteration + 1;
```

We can now use lin_prog_rcg to solve our example linear program.

```
>> c = 50; lb = 0;
>> A = [ 400 ];
>> sense = [ 'G' ];
>> b = [ 500 ];
>> RCGparam.nmax = 4;
>> RCGparam.mmax = 4;
>> RCGparam.nzmax = 14;
>> AUXparam.iteration = 0;
>> MatCPXconst;
>> CPXparam(1).param = CPX.PARAM_SCRIND;
>> CPXparam(1).type  = 'integer';
>> CPXparam(1).value = CPX.ON;
>> [optval,optx,optpi,optslack,optdj,AUXparam,status] = ...
lin_prog_rcg(c,A,sense,b,lb,[],'rcgen',RCGparam,AUXparam,CPXparam)
Tried aggregator 1 time.
LP Presolve eliminated 1 rows and 1 columns.
All rows and columns eliminated.
Presolve time =    0.00 sec.

optval =
   62.5000

Iteration log . . .
```

```
Iteration:     1    Infeasibility =             1.000000
Switched to devex.

optval =
    75

Iteration log . . .
Iteration:     1    Infeasibility =             0.000000
Switched to devex.

optval =
  142.5000

Using devex.

optval =
  142.5000

optval =
  142.5000

optx =
         0
    3.0000
    2.2500
         0

optpi =
         0
    5.0000
    7.5000
         0

optslack =
 -437.5000
         0
         0
   -6.2500

optdj =
```

```
    20
     0
     0
    50

AUXparam =
    iteration: 4

status =
     1
```

A.2 Miscellaneous MatCPX Functions

A.2.1 MatCPXconst: Load CPLEX symbolic constants

Synopsis

`MatCPXconst;`

Summary

`MatCPXconst` loads CPLEX symbolic constants into the current Matlab context.

Discussion

CPLEX callable library symbolic constants come in a number of forms. Typically, CPLEX constants have the form `XXX_YYY`, where `XXX` is usually `CPX`, but may also be `CPXALG`, `CPXERR`, `CPXIIS`, `CPXMIP`, `CPXNET`, or `CPXPROB`. After calling `MatCPXconst`, these are available in the calling Matlab context as `XXX.YYY`. The few CPLEX constants of the form `YYY` are made available as `CPX.YYY`.

Example

We demonstrate the use of CPLEX callable library symbolic constants `CPX_ON`, `CPX_OFF`, `CPX_PARAM_SCRIND`, and `INFBOUND`.

```
>> MatCPXconst;
>> CPX.ON

ans =
     1

>> CPX.OFF
```

```
ans =
     0

>> CPX.PARAM_SCRIND

ans =
        1035

>> CPX.INFBOUND

ans =
   1.0000e+20
```

A.2.2 MatCPXcleanup: Free CPLEX & MatCPX resources

Synopsis

`MatCPXcleanup;`

Summary

`MatCPXcleanup` frees CPLEX license tokens and dynamic memory held by MatCPX.

Discussion

`MatCPXcleanup` is called automatically if Matlab or the operating system raises an exception, the casual user should never need to explicitly use it. An advanced user might require `MatCPXcleanup` to make sure that CPLEX license tokens are released when they are no longer required.

A.3 MatCPX Access Functions for the CPLEX Callable Library

The MatCPX access functions are useful when the high level linear and integer programming solvers of Section A.1 do not offer sufficient power or flexibility for the problem at hand. It should be noted that if a good deal of power or flexibility is required, then there is no substitute for getting down and dirty and programming directly in C. Nevertheless, it is possible to achieve most of the functionality of coding directly in C using the CPLEX callable library by using the MatCPX access functions, albeit with some time and memory penalty.

MatCPX provides the following Matlab functions which interface to the CPLEX callable library access functions of the same name.

`CPXaddcols`	`CPXaddrows`	`CPXbaropt`
`CPXchecklp`	`CPXchgbds`	`CPXchgcoef`
`CPXchgctype`	`CPXchgfreecoef`	`CPXchgfromfreerow`
`CPXchgname`	`CPXchgobj`	`CPXchgobjsen`
`CPXchgprobtype`	`CPXchgqpcoef`	`CPXchgrhs`
`CPXchgsense`	`CPXchgtofreerow`	`CPXcloseCPLEX`
`CPXcopybase`	`CPXcopyorder`	`CPXcopystart`
`CPXdelcols`	`CPXdelrows`	`CPXdelsetcols`
`CPXdelsetrows`	`CPXdualopt`	`CPXdualwrite`
`CPXfreeprob`	`CPXgetbase`	`CPXgetdj`

`CPXgeterrorstring`	`CPXgetitc`	`CPXgetitcnt`
`CPXgetmobjval`	`CPXgetmx`	`CPXgetobjval`
`CPXgetphase1cnt`	`CPXgetpi`	`CPXgetslack`
`CPXgetstat`	`CPXgetx`	`CPXhybbaropt`
`CPXhybnetopt`	`CPXloadbase`	`CPXloadctype`
`CPXloadlp`	`CPXloadlpwnames`	`CPXloadorder`
`CPXloadstart`	`CPXlpmread`	`CPXlpread`
`CPXlprewrite`	`CPXlpwrite`	`CPXmipoptimize`
`CPXmpsmread`	`CPXmpsread`	`CPXmpsrewrite`
`CPXmpswrite`	`CPXnetopt`	`CPXopenCPLEX`
`CPXopenCPLEXdevelop`	`CPXoptimize`	`CPXprimopt`
`CPXsavmread`	`CPXsavread`	`CPXsavwrite`
`CPXsetdblparam`	`CPXsetintparam`	`CPXsolution`
`CPXunloadprob`		

The CPLEX documentation is not reproduced here, but it should be consulted for information about each of these functions and how they are used together. Each of these Matlab functions has a corresponding help file, accessed by the Matlab `help` command. The parameter names used in the Matlab help file correspond to the parameters of the same name in the CPLEX documentation, with the following exceptions:

- The CPLEX documentation appends the postfix `_p` to parameter names to indicate pointers to data types. The `_p` postfix has been dropped in the MatCPX version of the parameter name.

- Wherever the CPLEX Callable Library access functions call for (constraint) matrix information in a column (or row) oriented sparse data structure consisting of three integer arrays (for example: matbeg, matcnt, matind) and one double array (matval), these parameters are replaced with a single sparse Matlab array.

There are a few other differences between the access functions as described by CPLEX versus MatCPX, mostly due to differences in the data types available in C versus Matlab.

- Since C can only return a single value from a function, many CPLEX callable library access functions return values through function parameters. With MatCPX, output values are returned from the function and input parameters are passed to the function.

- Whenever the CPLEX callable library access function permits a null pointer to be passed, the empty matrix `[]` may be passed as the corresponding parameter to the MatCPX function.

- Whenever CPLEX can use a C preprocessor constant, MatCPX uses the symbolic constants defined by calling `MatCPXconst`. Note that `MatCPXconst` is automatically called in the context of any use of the function `CPXopenCPLEX`.

- Whenever the CPLEX callable library access function calls for an array of strings, use a Matlab cell array of strings.
- Whenever the CPLEX callable library access functions call for names of rows and columns (e.g. `CPXloadlpwnames`), requiring three parameters, `XXXname`, `XXXnamestore` and `XXXnamespace`, these parameters are replaced with a single Matlab cell array of strings.

When using these MatCPX access functions directly, care must be taken to ensure that CPLEX licensing tokens are not held for longer than intended. Since the Matlab process does not terminate when an error is encountered, the CPLEX licensing tokens are not released, even though Matlab has returned to its prompt. The function `MatCPXcleanup` is provided to free up all the resources in use by MatCPX for this sort of exceptional circumstance. See the Matlab source files for the linear and integer program solvers for ways to have Matlab call `MatCPXcleanup` for you automatically when it encounters an error.

The source of each of the high level linear and integer program solvers provides some excellent examples of how to use the MatCPX access functions. In addition, the MatCPX home directory (most likely stored in the environment variable `$MATCPX_HOME`) contains the subdirectory `examples`, holding MatCPX versions of CPLEX example programs `lpex1.c`, `lpex3.c`, and `lpex6.c`.

A.4 Site Specific Usage Information

In order to use MatCPX on the Sun workstations in the Computer Science Department at Cornell University, you must add the following to your `~/.cshrc` file:

```
if (! $?MATCPX_HOME) then
    setenv MATCPX_HOME /usr/local/ccop/matlab/matlab_cplex/cplex_6.6
    if ($?MATLABPATH) then
        setenv MATLABPATH ${MATCPX_HOME}:${MATLABPATH}
    else
        setenv MATLABPATH ${MATCPX_HOME}
    endif
endif
setenv SHORTHOST `hostname | sed -e 's/\..*//'`
setenv CPLEXLICDIR ~ccopstaff/cplex_6.6/cpxdir.$SHORTHOST
unsetenv SHORTHOST
```

Once these environment variables are set (use the command `source ~/.cshrc` to make sure), MatCPX can be used on any machine that has a CPLEX license. You can find out which machines have licenses by checking the contents of the directory `~ccopstaff/cplex_6.6`

Note: This information is correct as of September 11, 2000.

Appendix B

MatLEDA: A Matlab-LEDA Interface

The MatLEDA Matlab-LEDA interface provides a way to use the graph data structures of the C++ class library LEDA from Matlab.

Only a small subset of the functionality of the entire LEDA class library has been implemented. Most of the graph class and a small amount of the GraphWin class is available via MatLEDA. In combination with the data structures provided by Matlab and linear programming solvers from MatCPX, this subset of the LEDA class library was sufficient to implement all of the algorithms described in this thesis.

MatLEDA provides two Matlab object types, `graph` and `GraphWin`. Instances of these object types are created using the respective constructors. The methods of these object types are invoked using the dot syntax of C++. The nodes and edges of a graph are represented in MatLEDA by a unique index with Matlab type `int32` and `uint32` respectively. The index numbers the nodes and edges consecutively from 1, meaning that nodes and edges can be used to index Matlab arrays. The node and edge types are hidden from the user for the most part, but occasionally Matlab's error messages will refer to them.

Each MatLEDA method implements the LEDA method of the same name. Whenever a LEDA method takes a node or edge as an argument the corresponding MatLEDA method can accept an array of nodes or edges. Calling Matlab MEX functions is sufficiently expensive that using the methods in this vectorized way is necessary to achieve satisfactory performance. The extended example provided demonstrates how this is done.

B.1 `graph` Class

Creation and Deletion

`graph`

Constructs a new graph object.

Example: `>> G = graph;`

`delete(graph G)`

Destroys a graph object `G`, freeing all resources associated with it.

Example: `>> delete(G);`

Access Operations

`boolean graph::empty`

Returns `1` if the graph has no nodes or edges, `0` otherwise.

Example: `>> if G.empty return end;`

`boolean graph::is_directed`

Returns `1` if the graph is directed, `0` otherwise.

Example: `>> if G.is_directed return end;`

`boolean graph::is_undirected`

Returns `1` if the graph is undirected, `0` otherwise.

Example: `>> if G.is_undirected return end;`

`nodes graph::all_nodes`

Returns all nodes of the graph.

Example: `>> V = G.all_nodes;`

`edges graph::all_edges`

Returns all edges of the graph.

Example: `>> E = G.all_edges;`

`int graph::number_of_nodes`

Returns the number of nodes of the graph.

Example: `>> n = G.number_of_nodes;`

`int graph::number_of_edges`

Returns the number of edges of the graph.

Example: `>> m = G.number_of_edges;`

`nodes graph::source(edges EL)`

Returns the source node of each edge of `EL`.

Example: `>> NL = G.source(E);`

`nodes graph::target(edges EL)`

Returns the target node of each edge of `EL`.

Example: `>> NL = G.target(E);`

`nodes graph::opposite(nodes NL, edges EL)`

For each node edge pair `(v,e)` at corresponding positions of `NL` and `EL`, returns the target of `e` if `v` is the source, and the source of `e` if `v` is the target.

Example: `>> NL = G.opposite(G.source(E),E);`

`edges graph::adj_edges(nodes NL)`

Returns the edges with source `v` for each node `v` of `NL`.

Example: `>> EL = G.adj_edges(V);`

`nodes graph::adj_nodes(nodes NL)`

Returns the target nodes of the edges with source `v` for each node `v` of `NL`.

Example: `>> NL = G.adj_nodes(V);`

`edges graph::out_edges(nodes NL)`

Returns the edges with source `v` for each node `v` of `NL`.

Example: `>> EL = G.out_edges(V);`

`edges graph::in_edges(nodes NL)`

If the graph is directed, returns the edges with target `v` for each node `v` of `NL`. If the graph is undirected, returns the empty list.

Example: `>> EL = G.in_edges(V);`

`ints graph::outdeg(nodes NL)`

Returns the number of edges with source `v` for each node `v` of `NL`.

Example: `>> d = G.outdeg(V);`

`ints graph::indeg(nodes NL)`

If the graph is directed, returns the number of edges with target `v` for each node `v` of `NL`. If the graph is undirected, returns zero for each node of `NL`.

Example: `>> d = G.indeg(V);`

`ints graph::degree(nodes NL)`

Returns `outdeg(NL) + indeg(NL)`.

Example: `>> d = G.degree(V);`

`nodes graph::choose_node(int n)`

Returns `n` nodes selected uniformly at random (with replacement).

Example: `>> NL = G.choose_node(10);`

`edges graph::choose_edge(int m)`

Returns `m` edges selected uniformly at random (with replacement).

Example: `>> EL = G.choose_edge(10);`

`graph::print`

Prints the graph.

Example: `>> G.print;`

Update Operations

`graph::make_directed`

Makes the graph directed.

Example: `>> G.make_directed;`

`graph::make_undirected`

Makes the graph undirected.

Example: `>> G.make_undirected;`

`graph::make_bidirected`

Makes the graph bidirected.

Example: `>> G.make_bidirected;`

`nodes graph::new_node(int n)`

Adds `n` nodes to the graph, and returns them.

Example: `>> V = G.add_node(10);`

`edges graph::new_edge(nodes NL1, nodes NL2)`

Adds an edge `(v1,v2)` for each `v1` of `NL1` and `v2` of `NL2`, and returns the added edges.

Example: `>> E = G.add_edge(V,V);`

`edges graph::new_edge(node-pairs NPL)`

For the $2 \times m$ array of nodes `NPL`, adds an edge `(v1,v2)` for each node pair `(v1,v2)` of `NLP`, and returns the added edges.

Example: `>> E = G.add_edge([V;V]);`

`graph::del_all_nodes`

Deletes all nodes from the graph.

Example: `>> G.del_all_nodes;`

`graph::del_all_edges`

Deletes all edges from the graph.

Example: `>> G.del_all_edges;`

`graph::clear`

Deletes all nodes and edges from the graph.

Example: `>> G.clear;`

`graph::del_node(nodes NL)`

Delete each node of `NL` from the graph.

Example: `>> G.del_node(V);`

`graph::del_edge(edges EL)`

Delete each edge of `EL` from the graph.

Example: `>> G.del_edge(E);`

`graph::rev_all_edges`

Reverses all edges of the graph.

Example: `>> G.rev_all_edges;`

`edges graph::insert_reverse_edges`

Adds the reverse of each edge to the graph and returns the added edges.

Example: `>> E = G.insert_reverse_edges;`

Hide Operations

`graph::hide_node(nodes NL)`

Hide each node of `NL`.

Example: `>> G.hide_node(V);`

`graph::hide_edge(edges EL)`

Hide each edge of `EL`.

Example: `>> G.hide_edge(E);`

`graph::restore_node(nodes NL)`

Restore each node of `NL`.

Example: `>> G.restore_node(V);`

`graph::restore_edge(edges EL)`

Restore each edge of `EL`.

Example: `>> G.restore_edge(E);`

`booleans graph::is_hidden(nodes NL)`

Returns a `1` if `v` is hidden and `0` otherwise, for each node `v` of `NL`.

Example: `>> if any(G.is_hidden(V)) return end;`

`booleans graph::is_hidden(edges EL)`

Returns a `1` if `e` is hidden and `0` otherwise, for each edge `e` of `EL`.

Example: `>> if any(G.is_hidden(E)) return end;`

`nodes graph::hidden_nodes`

Returns all hidden nodes.

Example: `>> V = G.hidden_nodes;`

`edges graph::hidden_edges`

Returns all hidden edges.

Example: `>> E = G.hidden_edges;`

`graph::restore_all_nodes`

Restores all nodes.

Example: `>> G.restore_all_nodes;`

`graph::restore_all_edges`

Restores all edges.

Example: `>> G.restore_all_edges;`

B.2 GraphWin Class

Creation and Deletion

`GraphWin(graph G)`

Constructs a new GraphWin object associated with the graph `G`.

Example: `>> GW = GraphWin(G);`

`delete(GraphWin GW)`

Destroys a GraphWin object `GW`, freeing all resources associated with it.

Example: `>> delete(GW);`

Operations

`GraphWin::display`

Displays the graph associated with the Graph Window.

Example: `>> GW.display;`

`boolean GraphWin::edit`

Place the Graph Window in edit mode. In edit mode, the user can modify the graph associated with the Graph Window interactively. Edit mode is terminated by the user either pressing the done button, in which case `1` is returned, or by selecting exit from the file menu, in which case `0` is returned.

Example: `>> GW.edit;`

`boolean GraphWin::open`

Displays the graph associated with the Graph Window and places the Graph Window into edit mode. Equivalent to a `display` followed by an `edit`. Edit mode is terminated by the user either pressing the done button, in which case `1` is returned, or by selecting exit from the file menu, in which case `0` is returned.

Example: `>> GW.open;`

`GraphWin::close`

Closes the Graph Window.

Example: `>> GW.close;`

`GraphWin::update_graph`

Notify the Graph Window that the graph it is associated with has changed.

Example: `>> GW.update_graph;`

`GraphWin::redraw`

Force a redraw of the Graph Window.

Example: `>> GW.redraw;`

B.3 Miscellaneous MatLEDA Functions

Shortest Path

```
[nodes NL, doubles D, edges P]
::ACYCLIC_SHORTEST_PATH(graph G, node s, edges EL, doubles costs)
```

Solves the single source shortest path problem with respect to source `s` for the acyclic graph `G`. Each edge double pair `(e,c)` at corresponding positions of `EL` and `costs` represents the cost `c` of edge `e`. Returns the nodes for which shortest paths have been computed `NL`, shortest path distances `D` for the nodes at corresponding positions of `NL`, and the predecessor edges `P` on the shortest path for the nodes at corresponding positions of `NL`.

Example: `>> [NL, D, P] = ACYCLIC_SHORTEST_PATH(G, s, E, c);`

Type conversion

```
nodes node(doubles);
```

Converts a Matlab double to a node datatype.

Example: `>> nil = node(0);`

```
edges edge(doubles);
```

Converts a Matlab double to an edge datatype.

Example: `>> EL = edge([]);`

B.4 Extended Example

To demonstrate how to use MatLEDA, we provide an example implementation of two functions from LEDA: A platform for combinatorial and geometric computing, by Mehlhorn and Näher [MN99]. We implement two algorithms, the topological ordering algorithm of pages 244-245 and the acyclic shortest path algorithm of pages 332-333, which uses the topological ordering algorithm as a subroutine. In each case, we first provide a literal transcription of the C++ code for LEDA into a form suitable for Matlab and MatLEDA, and then vectorize the code in order to achieve better performance.

```
% Filename: TOPSORT.m
function [ is_acyclic, node_ord ] = TOPSORT(G)

% Initialization
ZEROINDEG = node([]);
for v = G.all_nodes
  INDEG(v) = G.indeg(v);
  if INDEG(v) == 0
    ZEROINDEG(end+1) = v;
  end
end

% Removing nodes of indegree zero
count = 0;
while ~ isempty(ZEROINDEG)
  v = ZEROINDEG(1);
  ZEROINDEG = ZEROINDEG(2:end);
  count = count + 1;
  node_ord(v) = count;
  for e = G.out_edges(v)
    w = G.target(e);
    INDEG(w) = INDEG(w) - 1;
    if INDEG(w) == 0
      ZEROINDEG(end+1) = w;
    end
  end
end
is_acyclic = (count == G.number_of_nodes);
```

Figure B.1: Literal implementation of topological ordering with MatLEDA.

```
% Filename: TOPSORT.m
function [ is_acyclic, node_ord ] = TOPSORT(G)

% Initialization
V = G.all_nodes;
INDEG = G.indeg(V);
ZEROINDEG = V(find(INDEG == 0));

% Removing nodes of indegree zero
count = 0;
while ~ isempty(ZEROINDEG)
  v = ZEROINDEG(1);
  ZEROINDEG = ZEROINDEG(2:end);
  count = count + 1;
  node_ord(v) = count;
  ANL = G.target(G.out_edges(v));
  INDEG(ANL) = INDEG(ANL) - 1;
  ZEROINDEG = [ ZEROINDEG ANL(find(INDEG(ANL) == 0)) ];
end
is_acyclic = (count == G.number_of_nodes);
```

Figure B.2: Vectorized implementation of topological ordering with MatLEDA.

```
% Filename: ACYCLIC_SHORTEST_PATHS.m
function [ dist, pred ] = ACYCLIC_SHORTEST_PATH(G, s, c)

% establish topological order
[is_acyclic, top_ord] = TOPSORT(G);
n = G.number_of_nodes;
for w = G.all_nodes
  v(top_ord(w)) = w;
end

% acyclic_shortest_path main function
for w = G.all_nodes
  pred(w) = edge(0);
end
dist(s) = 0;
for i = 1:n
  u = v(i);
  if (pred(u) == 0) & (u ~= s)
    % do nothing, Matlab has no continue statement...
  else
    du = dist(u);
    for e = G.adj_edges(u)
      w = G.target(e);
      if (pred(w) == 0) | (du + c(e) < dist(w))
        pred(w) = e;
        dist(w) = du + c(e);
      end
    end
  end
end
```

Figure B.3: Literal implementation of acyclic shortest paths with MatLEDA.

```
% Filename: ACYCLIC_SHORTEST_PATHS.m
function [ dist, pred ] = ACYCLIC_SHORTEST_PATH(G, s, c)

% establish topological order
V = G.all_nodes;
[is_acyclic, top_ord] = TOPSORT(G);
U(top_ord) = V;

% acyclic_shortest_path main function
pred(V) = edge(0);
dist(V) = inf;
dist(s) = 0;
for u = U
  if (pred(u) == 0) & (u ~= s)
    % do nothing, Matlab has no continue statement...
  else
    AEL = G.adj_edges(u);
    ANL = G.target(AEL);
    ind1 = find(pred(ANL) == 0);
    ind2 = find(dist(u) + c(AEL) < dist(ANL));
    NL = [ ANL(ind1) ANL(ind2) ];
    EL = [ AEL(ind1) AEL(ind2) ];
    pred(NL) = EL;
    dist(NL) = dist(u) + c(EL);
  end
end
```

Figure B.4: Vectorized implementation of acyclic shortest path with MatLEDA.

```
>> G = graph;                          % Construct empty graph G.
>> V = G.new_node(8);                  % Add the nodes
>> from = V([1 1 2 2 3 3 4 4 4 5 5 7 7 8]);
>> to =   V([4 2 4 3 5 6 3 7 5 8 6 5 8 6]);
>> E = G.new_edge([from;to]);          % and the edges.
>> G                                   % Print the graph.
G =
[0] : [0]---->[3][0]---->[1]
[1] : [1]---->[3][1]---->[2]
[2] : [2]---->[4][2]---->[5]
[3] : [3]---->[2][3]---->[6][3]---->[4]
[4] : [4]---->[7][4]---->[5]
[5] :
[6] : [6]---->[4][6]---->[7]
[7] : [7]---->[5]

>> c(E) = rand(1,G.number_of_edges); % Create random edge costs.
>> [D,P] = ACYCLIC_SHORTEST_PATH(G,V(1),c); % Find shortest paths.
>> G.hide_edge(E);                     % Hide edges that are not used
>> G.restore_edge(P(2:end));           % by restoring the used ones.
>> G                                   % Print the graph.
G =
[0] : [0]---->[1][0]---->[3]
[1] : [1]---->[2]
[2] :
[3] : [3]---->[4][3]---->[6]
[4] : [4]---->[5]
[5] :
[6] : [6]---->[7]
[7] :

>> delete(G);                          % Delete the graph G.
```

Figure B.5: Acyclic shortest paths demonstration.

B.5 Site Specific Usage Information

In order to use MatCPX on the Sun workstations in the Computer Science Department at Cornell University, you must add the following to your `~/.cshrc` file:

```
if (! $?MATLEDA_HOME) then
    setenv MATLEDA_HOME /usr/local/ccop/matlab/matlab_leda
    if ($?MATLABPATH) then
        setenv MATLABPATH ${MATLEDA_HOME}:${MATLABPATH}
    else
        setenv MATLABPATH ${MATLEDA_HOME}
    endif
    if ($?LD_LIBRARY_PATH) then
        setenv LD_LIBRARY_PATH ${MATLEDA_HOME}:${LD_LIBRARY_PATH}
    else
        setenv LD_LIBRARY_PATH ${MATLEDA_HOME}
    endif
endif
```

Once these environment variables are set at Matlab startup time (use the command `source ~/.cshrc` to make sure), MatLEDA is ready to use.

Note: This information is correct as of September 11, 2000.

Bibliography

[ACS99] K. Aardal, F. Chudak, and D. Shmoys. A 3-approximation algorithm for the k-level uncapacitated facility location problem. *IPL: Information Processing Letters*, 72, 1999.

[AL97] S. Arora and C. Lund. Hardness of approximations. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 10. PWS Publishing Company, 1997.

[ALLQ96] K. Aardal, M. Labbé, J. Leung, and M. Queyranne. On the two-level uncapacitated facility location problem. *INFORMS Journal of Computing*, 8:289–301, 1996.

[AZ98] M. Andrews and L. Zhang. The access network design problem. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 40–49, Los Alamitos, CA, November 8–11 1998. IEEE Computer Society.

[Bea90] J. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990. See `http://www.ms.ic.ac.uk/info.html`.

[CG99] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[CGTS99] M. Charikar, S. Guha, É. Tardos, and D. Shmoys. A constant-factor approximation algorithm for the k-median problem (extended abstract). In *Proc. 31st Annu. ACM Symposium on Theory of Comput.*, pages 1–10. ACM Press, New York, NY, 1999.

[Cha99] E. Chan. *Perfect Sampling Using Bounding Chains*. PhD thesis, Cornell University, 1999.

[Chu98a] F. Chudak. *Improved Approximation Algorithms for the Uncapacitated Facility Location Problem.* PhD thesis, Cornell University, 1998.

[Chu98b] F. Chudak. Improved approximation algorithms for uncapacitated facility location. In *IPCO: 6th Integer Programming and Combinatorial Optimization Conference*, 1998.

[CNW90] G. Cornuéjols, G. Nemhauser, and L. Wolsey. The uncapacitated facility location problem. In R. Maichandani and R. Francis, editors, *Discrete Location Theory.* John Wiley and Sons, Inc., New York, 1990.

[CS] F. Chudak and D. Shmoys. Improved approximation algorithms for uncapacitated facility location. Unpublished manuscript.

[dB95] A. de Barros. *Discrete and fractional programming techniques for location problems.* PhD thesis, Erasmus Universiteit Rotterdam, 1995.

[Edm72] J. Edmonds. Edge-disjoint branchings. In R. Rustin, editor, *Combinatorial Algorithms*, pages 91–96. Algorithmics Press, New York, 1972.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[GK98] S. Guha and S. Khuller. Greedy strikes back: improved facility location algorithms. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 649–657, San Francisco, California, 25–27 January 1998.

[GLS88] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer, Berlin, 1988.

[GM95] H. Gabow and K. Manu. Packing algorithms for arborescences (and spanning trees) in capacitated graphs. In *IPCO: 4th Integer Programming and Combinatorial Optimization Conference*, 1995.

[GW95] M. Goemans and D. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, April 1995.

[Hoc82] D. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22:148–162, 1982.

[Hoc97] D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1997.

[Hub99] M. Huber. *Perfect Sampling Using Bounding Chains.* PhD thesis, Cornell University, 1999.

[JV99] K. Jain and V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[Kar84] N. Karmarkar. A new polynomial–time algorithm for linear programming. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

[KH63] A. Kuehn and M. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9(9):643–666, July 1963.

[KPR98] M. Korupolu, C. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, San Francisco, California, 25–27 January 1998.

[Kru98] S. Kruglyak. *DNA Sequencing and Modeling Repeat Sequence Evolution.* PhD thesis, Cornell University, 1998.

[KvdEH77] L. Kaufman, M. van den Eede, and P. Hansen. A plant and warehouse location problem. *Operational Research Quarterly*, 28:547–557, 1977.

[LV92a] J.-H. Lin and J. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44(5):245–249, December 1992.

[LV92b] J.-H. Lin and J. Vitter. ε-approximation with minimum packing constraint violation. In *Proceedings of the 24th Annual ACM Symosium on Theory of Computing*, pages 771–782, 1992.

[Man64] A. Manne. Plant location under economies-of-scale-decentralization and computation. *Management Science*, 11:213–235, 1964.

[MF90] R. Maichandani and R. Francis, editors. *Discrete Location Theory.* John Wiley and Sons, Inc., New York, 1990.

[MN99] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, January 1999.

[NW88] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[Pan00] G. Pangborn. Personal communication, 2000.

[STA97] D. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 265–274, El Paso, Texas, 4–6 May 1997.

[Sto61] J. Stollsteimer. *The effect of technical change and output expansion on the optimum number, size and location of pear marketing facilities in a California pear producing region.* PhD thesis, University of California at Berkeley, 1961.

[Sto63] J. Stollsteimer. A working model for plant numbers and locations. *Journal of Farm Economics*, 45:631–645, 1963.

[TL84] D. Tcha and B. Lee. A branch-and-bound algorithm for the multi-level uncapacitated location problem. *European Journal of Operations Research*, 18:25–43, 1984.

[VE82] T. Van Roy and D. Erlenkotter. A dual based procedure for dynamic facility location. *Management Sciences*, 28:1091–1105, 1982.

[Zan98] P. Zanjacomo. *On Weighted Paths for Nonlinear Semidefinite Complementarity Problems and Newton Methods for Semidefinite Programming.* PhD thesis, Georgia Institute of Technology, 1998.