Placement Algorithms for Hierarchical Cooperative Caching and Other Location Problems

by

Madhukar Korupolu, B.Tech., M.S.

(Supervisor: C. Greg Plaxton)

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1999

To my parents

Abstract

In a large-scale information system, such as a digital library or the world wide web, a set of distributed caches can improve their effectiveness by cooperating with one another, both in serving each other's requests as well as in deciding what to store. This dissertation explores the potential of such cooperative caching, and provides basic *placement algorithms* using which the caches can coordinate their storage decisions.

The first part of the dissertation focuses on variants of the placement problem involving a single object. The most well-known of these variants are the facility location problems, which have received considerable attention in the operations research literature due to their widespread applicability. We prove that a simple *local search* heuristic, proposed about 25 years ago, yields polynomial-time constant-factor approximations for several metric facility location problems.

The second part of the dissertation addresses the simultaneous placement of a collection of objects in *hierarchical* networks. We provide both exact and approximate polynomial-time algorithms for this hierarchical placement problem. Our exact algorithm is based on a reduction to min-cost flow, and does not appear to be practical for large problem sizes. Hence we are motivated to look for simpler approximate algorithms. Our main result is a simple constant-factor approximation algorithm that admits an efficient distributed implementation.

The third and final part of the dissertation describes our simulation experiments which were performed to ascertain the practical potential of cooperative caching, and to investigate the practical performance of various placement and replacement algorithms. Driven by both synthetic and web-trace workloads, these experiments compare several traditional as well as novel caching algorithms. They demonstrate that cooperation improves performance significantly, especially when the cache sizes are *small*. More surprisingly, the experiments show that our amortizing algorithm yields placements that are extremely close, within 5%, to the optimal when the access patterns are stable. We also examine *hybrid* strategies to cope with changing access patterns.

Acknowledgments

First and foremost, I am deeply indebted to my dissertation supervisor Greg Plaxton. His guidance and encouragement have been invaluable in shaping this dissertation and taking it to completion. Besides supervising this dissertation, Greg has been an amiable mentor and a great friend all along. His infectious enthusiasm together with his 'work-hard, play-hard' philosophy will continue to inspire me for years to come.

I have benefitted greatly from discussions with several other faculty members as well. In particular, David Zuckerman and Martin Wong were very helpful during the initial stages of my graduate studies. David's classes and puzzles were always challenging and reinforced my zeal for research in algorithms. Mike Dahlin and Harrick Vin helped me in my initial forays into experimental work and provided valuable insights. Mike also contributed directly to this dissertation by collaborating with me on the experimental work presented in Chapter 4. I am also grateful to Eva Tardos of Cornell University, as well as Mike Dahlin, Harrick Vin, and David Zuckerman for graciously serving on my dissertation committee and providing encouraging and insightful comments.

In addition to Greg and Mike, another researcher who contributed significantly to this dissertation is Rajmohan Rajaraman. Raj is a close friend and has been involved with this dissertation right from the beginning; he co-authored the results on the facility location problems in Chapter 2 and the placement problem in Chapter 3.

Several colleagues in the computer science department were instrumental in making my graduate life all the more interesting and enjoyable. In particular, the numerous discussions, technical or otherwise, with Ashis Tarafdar, John Havlicek, Kedar Namjoshi, Sriram Rao, Vasilis Samoladas, and Renu Tewari have produced several memorable moments. On the outdoors front, Ramgopal Mettu introduced me to the sport of mountain-biking, and Yannis Smaragdakis helped me extensively in my hunt for a used motorcycle.

Throughout my stay in Austin, I have been very fortunate to have some great roommates and friends who made my stay a very pleasant and memorable one. I will always remember Navin and his delicious chicken dishes and his ever-surprising take-it-easy policy, Harsha and his infectious cheerfulness (except before the exams!), Vicky and our regular workout sessions, and of course, the affable Randy and his in-depth analysis of the NBA and NFL games. During the early years, the sixth street was an in-thing and Gooty, Santanu, Mouli, and Dan were cool company on several trips. Of late, hanging out with the entire Rio group – Nandu, MP, Gounder, Shail, Satish and Soumya Pillais, Freaky, Gokul, VMS, and several others – has been great fun. Though I missed out on several of their activities due to time constraints, the memories of the card games, the cricket matches, and the birthday parties in J192 will remain for a long while.

Aparna has been a wonderful friend ever since she moved into the office next door to mine in UA9. If not for her coffee and lively conversations, I wouldnt have spent as much time working in the office and this dissertation would have taken that much longer.

Of course, I would not have been able to do any of the above without the financial support provided by the National Science Foundation, The University of Texas at Austin, and the summer internships at Fujitsu Labs of America and Bell Labs. For providing me these internship opportunities and for making them a worthwhile experience, technically or otherwise, I am grateful to Adnan Aziz, Sreeranga P. Rajan, S. Muthukrishnan, and Ganesh Sundaram. Also, I would like to thank Gloria Ramirez and Doug Rockstead, our department staff, for helping me with the various administrative formalities right from day one.

Lastly, and most importantly, my parents and my sister Rajitha have been an endless source of love and support throughout my graduate study, and all through my life. In spite of the long years of geographic separation, they – along with my cousin brother Srikanth and the rest of my folks at Hyderabad – never missed a beat in keeping my morale high irrespective of the circumstances. My parents have always given me the

freedom to pursue my interests, and have always backed up my choice with tremendous confidence and affection; if not for that, I wouldnt be writing this today. This dissertation is dedicated to them.

Madhukar Korupolu

The University of Texas at Austin
August 1999

Contents

Chapter 1

Introduction

1.1 Cooperative caching

Consider a large-scale information system, such as a digital library or the world wide web, that provides access to shared objects. A fundamental technique for improving the performance of such a system is to cache popular objects close to the clients. This enables requests to be satisfied by a nearby copy and hence reduces not only the access latency but also the burden on the network as well as the server.

In the simplest caching scheme, caches never consult one another, and when a cache miss occurs, a cache contacts the server directly. To improve the effectiveness of caching, a simple yet powerful paradigm is *cooperation*, where caches cooperate both in serving each other's requests as well as in making storage decisions. Such cooperation is particularly attractive in environments where machines trust one another such as within an Internet service provider, a cache service provider, or a corporate intranet.

There are two orthogonal issues in cooperative caching: finding nearby copies of objects (*object search*) and coordinating the caches while making storage decisions (*object placement*). Several recent studies focused on the object search problem (e.g., Summary Cache [24], CRISP [26], Cache Digest [51], Hint Cache [55], and Adaptive Web Caching [63]). However, none of these studies address the cache coordination issue. For local-area networks, the cache coordination issues were studied by Dahlin *et al.* [21], Feeley *et al.* [25], and Leff *et al.* [40]. However, their techniques do not apply to the scenario of wide-area networks, primarily because the *distances*, or communication costs, in these networks can vary significantly from one pair of nodes to another.

In this work, we focus on the cache coordination issue for wide-area networks, and provide basic placement algorithms that allow caches to coordinate their storage decisions. Loosely speaking, the placement algorithms attempt to solve the following problem: given a set of cooperating nodes, the amount of cache space at each node, the network distances between the nodes, and the predictions of access rates from each node to a set of objects, determine where to place the copies of each object in order to minimize the average access cost. On the other hand, the replacement algorithms also attempt to minimize the access cost, but rather than explicitly computing a placement based on access frequencies, they proceed by evicting objects when a cache miss occurs.

We make several simplifying assumptions in order to focus on the basic problem. One assumption is that all objects are unit-sized and read-only. Second, we assume that the network distances, or communication costs, between node pairs are fixed and do not change over time. An interesting area for future work is to relax these assumptions.

The contributions of this dissertation are three-fold, and involve both theoretical as well as empirical results. In the next three sections, we give a brief overview of our results and then in Section 1.5, we give an outline of the rest of this dissertation.

1.2 Single object placement: Facility location problems

The first part of this dissertation focuses on facility location problems. These problems address a scenario in which we are given a set of client locations that need to be provided a certain service (or commodity), and the objective is to determine locations at which to open facilities so as to minimize the cost of providing service. Typical questions of interest are the following: (1) If we want to open exactly k facilities, where should we locate them in order to minimize the cost of providing service? (2) If instead we are willing to open any number of facilities, and the cost of opening a facility at each potential location is known, where should we locate them in order to minimize the sum of the total construction cost and the cost of providing service? For a precise definition of these problems, we refer the reader to Chapter 2.

Facility location problems model applications in diverse fields including public policy (e.g., locating fire stations in a city), telecommunications (e.g., locating base stations in wireless networks), and information retrieval (e.g., clustering of documents), and have been studied extensively over the past three decades. Our interest in these problems stems from their close connection to basic network caching problems such as the following: If we are willing to place k copies of a file in the network, where should we place them in order to minimize the average access cost? This is essentially the problem of placing copies of a single object in a network.

In Chapter 2, we analyze a simple local search heuristic for metric facility location problems. (In a metric facility location problem, it is assumed that the distances are nonnegative, symmetric, and satisfy the triangle inequality.) We prove that this simple heuristic yields polynomial-time constant-factor approximations for several metric facility location problems.

The local search heuristic that we analyze is extremely straightforward. The idea is to start with any feasible solution (set of locations) and then to iteratively improve the solution by repeatedly moving to the best "neighboring" feasible solution, where one solution is a neighbor of another if it can be obtained by either adding a facility, deleting a facility, or changing the location of a facility. This heuristic was first proposed by Kuehn and Hamburger [39], and was subsequently shown to exhibit good practical performance in empirical studies (see, e.g., [22, 54]). However, our work is the first to provide a nontrivial upper bound on the performance of this heuristic.

All the previously known approximation algorithms for the facility location problems (e.g., [41, 53]), are based on a careful rounding of fractional solutions to linear programming relaxations. Thus, these algorithms are substantially more sophisticated than the local search heuristic that we analyze. (Note that known polynomial-time algorithms for linear programming [34, 35] are quite complicated.)

1.3 Multiple object placement: Hierarchical placement problem

The second part of the dissertation focuses on the simultaneous placement of copies of multiple objects in a network. Motivated by the hierarchical structure of existing wide-area networks, we restrict our attention to the *hierarchical network* model, wherein the distance between two nodes is given by the diameter of the smallest hierarchy containing both nodes. (For a precise definition, see Chapter 3.) Equivalently, the distance function is given by an *ultrametric* [33]. Similar hierarchical models of wide-area networks have been implicitly used in several previous studies (e.g., [13, 55, 56]).

In Chapter 3, we provide both exact and approximate algorithms for this *hierarchical placement problem*. By viewing it as a combinatorial optimization problem, we first design an optimal centralized algorithm via a reduction to the minimum-cost flow problem. This is a generalization of the approach of Leff *et al.* [40], who solved the problem for a single level hierarchy. While this algorithm may be practical for small local-area networks, its centralized nature as well as the high computational and bandwidth requirements make it impractical for larger wide-area networks.

The remainder of our results are aimed towards the design of a fast and simple algorithm that is amenable to a distributed implementation. To this end, we relax our objective of true optimality to that of achieving a constant-factor approximation. We first consider a greedy algorithm that obtains a placement by a simple bottom-up pass through the hierarchical decomposition tree. This algorithm is based on a natural greedy improvement heuristic. Somewhat surprisingly, we show that the approximation ratio of the greedy algorithm is $\Omega(n)$, where n is the number of nodes in the network. The lower bound proof leads us to a natural refinement of the greedy algorithm, the *amortizing algorithm*. Like the greedy algorithm, the amortizing algorithm starts with a placement in which each machine places in its cache the locally optimal set of objects. The algorithm then proceeds by iteratively improving the placement in a bottom-up manner as the machines cooperate and share information about the access frequencies across larger regions of the network.

Our main result is that the amortizing algorithm achieves a constant-factor approximation for the hierarchical placement problem. (The constant factor is less than 14.) Moreover we show that this algorithm can be implemented efficiently even in a distributed setting. The theoretical guarantee coupled with the simplicity of the algorithm led us to conjecture that the algorithm would work well in practice.

1.4 Experimental study of cooperative caching

The third and final part of the dissertation describes our simulation experiments which were performed to ascertain the practical potential of cooperative caching, and to investigate the practical performance of various placement and replacement algorithms. The experiments were driven by both synthetic as well as web-trace [20] workloads. Aside from the three coordinated placement algorithms – namely, the optimal, the greedy, and the amortizing algorithms – mentioned in Section 1.3, our study also includes traditional *non-coordinated* algorithms such as MFU placement, LRU replacement, and greedy-dual replacement [12, 61]. In the non-coordinated algorithms, each cache makes its storage decisions independently, that is, without consulting any other cache. For details regarding the algorithms, the particular workloads used, and the experimental methodology, we refer the reader to Chapter 4.

Our experiments demonstrate that cooperative caching improves performance significantly, especially when the cache sizes are small compared to the universe of objects. More surprisingly, our experiments show that both the amortizing as well as the greedy placement algorithms yield placements that are very close, within 5%, to the optimal when the access patterns are stable. This is encouraging for two reasons. First, in systems with stable or predictable access patterns, these are practical algorithms that can be expected to provide near-optimal performance. (Digital libraries and subscription-based systems are good examples of such systems.) Second, for large-scale studies of cache coordination, these can provide a practical "best case" baseline which can be used to evaluate other algorithms.

For systems with changing access patterns, replacement algorithms may be a good alternative to placement algorithms. Hence our experimental study focuses on replacement algorithms as well. Our preliminary experiments revealed that the greedy-dual replacement algorithm, in spite of being non-coordinated, facilitates some implicit coordination among the caches and thereby performs surprisingly better than the other non-coordinated algorithms. With the motivation of further improving performance via coordination, we experimented with several hierarchical generalizations of the greedy-dual algorithm. Interestingly only one of these generalizations – namely our *hierarchical-*

greedy-dual – outperforms greedy-dual, and exhibits good performance improvements, particularly when there are idle caches in the system.

Another natural way to cope with changing access frequencies is to run the placement algorithms at regular intervals of time. In addition, within each interval, we could use a replacement strategy to refine the placement after each access request. We evaluate several such hybrid strategies in our experimental study. For details, we refer the reader to Chapter 4.

1.5 Organization

As indicated in the previous three sections, the rest of this dissertation is organized in three parts as follows. Chapters 2 and 3 present our results for the facility location problems and the hierarchical placement problem, respectively. Both of these results are based on theoretical analysis. Chapter 4 then presents our experimental results evaluating the performance of various cooperative caching algorithms. Finally, Chapter 5 concludes with a few directions for future research.

Each of the following chapters is self-contained. A reader interested in the experimental results alone can safely skip Chapters 2 and 3 and proceed directly to Chapter 4.

Chapter 2

Facility Location Problems

2.1 Introduction

Suppose that you plan to build a new chain of widget stores in a given city, and you have identified potential store sites in a number of different neighborhoods. Further assume that the demand for widgets in each neighborhood of the city is known. If you want to build exactly k stores, where should you locate them in order to minimize the average traveling distance of your customers? If instead you are willing to build any number of stores, and the cost of building a store at each potential site is known, where should you build stores in order to minimize the sum of the total construction cost and the average traveling distance of your customers?

The preceding questions are examples of *facility location problems.* Facility location problems model applications in diverse fields including public policy (e.g., locating fire stations in a city), telecommunications (e.g., locating base stations in wireless networks), and information retrieval (e.g., clustering of documents), and have been studied extensively over the past three decades. For more information on the applications of facility location problems, we refer the reader to [43, 46].

This dissertation is concerned with metric versions of the following specific facility location problems: (i) the uncapacitated k-median problem (UKM), (ii) the uncapacitated facility location problem (UFL), (iii) the capacitated k-median problem with

---

This is joint work with Greg Plaxton and Rajmohan Rajaraman. A preliminary version of this work appeared in [37].

unsplittable demands (CKMU), (iv) the capacitated facility location problem with unsplittable demands (CFLU), (v) the capacitated k-median problem with splittable demands (CKMS), and (vi) the capacitated facility location problem with splittable demands (CFLS). Formal definitions of these problems, all of which are known to be NP-hard, are given in Section 2.2.

Informally, UKM (resp., UFL) corresponds to the first (resp., second) of the two questions stated above, where we assume that there is no upper bound on the demand that can be satisfied by a given store, and hence the traveling distance of a customer is just the distance from that customer's neighborhood to the nearest store. In the capacitated versions of these problems, there is an upper bound on the demand that can be satisfied by a given store, and we need to exhibit an assignment of customers to stores that minimizes the average traveling distance while respecting the capacity bound of each store. In assigning the demand of customers to stores, there are two natural variations to consider: (i) the demand of a customer must be met by a single store (unsplittable demands), and (ii) the demand of a customer may be divided across any number of stores (splittable demands). The assumption of unsplittable demands yields the problems CKMU and CFLU, while the assumption of splittable demands yields the problems CKMS and CFLS.

The input to a facility location problem includes a distance matrix specifying the traveling distance from node i to node j, for all i and j. (The nodes correspond to the neighborhoods in our widget chain example.) In the *metric version* of a facility location problem, it is assumed that these distances are nonnegative, symmetric, and satisfy the triangle inequality. All of our results are for metric facility location problems.

The local search heuristic for facility location problems, described more formally in Section 2.3, is extremely straightforward. The idea is to start with any feasible solution (set of stores) and then to iteratively improve the solution by repeatedly moving to the best "neighboring" feasible solution, where one solution is a neighbor of another if it can be obtained by either adding a facility (store), deleting a facility, or changing the location of a facility. This heuristic was proposed by Kuehn and Hamburger [39], and was subsequently shown to exhibit good practical performance in empirical studies (see, e.g., [22, 54]).

For the k-median problems we consider, namely, UKM, CKMU, and CKMS, we define an (a, b)-*approximation algorithm* as a polynomial-time algorithm that computes a solution using at most bk facilities and with cost at most a times the cost of an optimal solution using at most k facilities. For facility location variants UFL, CFLU, and CFLS, we define an a-*approximation algorithm* as a polynomial-time algorithm that computes a solution with cost at most a times optimal. Using this terminology, and letting n denote the number of nodes in the input instance, we now summarize the previous approximation results known for the facility location problems considered in this dissertation: (i) Hochbaum [30] showed that a simple greedy algorithm is an $O(\log n)$-approximation algorithm for UFL with general distances (i.e., not restricted to metric distances); (ii) Lin and Vitter [42] gave a $(1+\varepsilon, (1+1/\varepsilon)(\ln n+1))$-approximation algorithm for UKM with general distances (their algorithm can also be adapted to match Hochbaum's result); (iii) Lin and Vitter [41] gave a $(2(1+\varepsilon), 1+(1/\varepsilon))$-approximation algorithm for metric UKM; (iv) Shmoys et al. [53] gave a 3.16-approximation algorithm for metric UFL, a 7-approximation algorithm for metric CFLS that requires a blowup of 7/2 in the capacity of each facility, and a 9-approximation algorithm for metric CFLU that requires a blowup of 4 in the capacity of each facility. More recently, the result of [53] for metric UFL has been improved by Guha and Khuller [27] who give a 2.41-approximation algorithm. It is also shown in [27] that there is no polynomial-time 7/6-approximation algorithm for metric UFL unless P = NP. (It is also shown in [27] that there is no polynomial-time 1.46-approximation algorithm unless $\mathrm{NP} \subseteq \mathrm{DTIME}[n^{O(\log\log n)}]$.)

Our results. The approximation algorithms of Lin and Vitter [41, 42] and Shmoys et al. [53] are based on careful rounding of fractional solutions to linear programming relaxations. Thus, these algorithms are substantially more sophisticated than the local search heuristic analyzed in this dissertation. (Note that known polynomial-time algorithms for linear programming [34, 35] are quite complicated.) Nevertheless, we prove that the local search heuristic yields simple algorithms for UKM and UFL with approximation bounds matching (to within constant factors) the corresponding results in (iii) and (iv) above. In particular, for metric UKM, we prove that for any constant

Problem	*Bound*	*Reference*	Approach
metric UKM	$(1+\varepsilon, (1+1/\varepsilon)(\ln n+1))$	[42]	LP-rounding
	$(2(1+\varepsilon), 1+(1/\varepsilon))$	[41]	LP-rounding
	$(1+\varepsilon, 3+5/\varepsilon)$	here	local search
	$(1+5/\varepsilon, 3+\varepsilon)$	here	local search
	$(20/3, 1)$	[15]*	LP-rounding
	$(6, 1)$	[32]*	primal-dual
Euclidean UKM	$(1+\varepsilon, 1)$	[3]*	dynamic prog.
general UFL	$O(\log n)$	[30]	greedy
metric UFL	≈ 3.16	[53]	LP-rounding
	≈ 2.41	[27]	LP-rounding
	$5+\varepsilon$	here	local search
	$1+2/e \approx 1.74$	[16]*	LP-rounding
	3	[32]*	primal-dual

Table 2.1: Approximation bounds for uncapacitated facility location problems. The notation "$\approx$" used in the entries for metric UFL means "approximately equal to". All of the results marked with an asterisk have been obtained subsequent to our results.

$\varepsilon > 0$, local search yields (a, b)-approximation bounds with $(a, b) = (1+\varepsilon, 3+5/\varepsilon)$ and $(a, b) = (1+5/\varepsilon, 3+\varepsilon)$. For metric UFL, we prove that for any constant $\varepsilon > 0$, local search yields a $(5+\varepsilon)$-approximation algorithm.

Our results for the capacitated facility location problems are similar. For metric CKMS, we prove that for any constant $\varepsilon > 0$, local search yields (a, b)-approximation bounds with $(a, b) = (1+\varepsilon, 5+5/\varepsilon)$ and $(a, b) = (1+5/\varepsilon, 5+\varepsilon)$ with no blowup in the capacity of each facility. For metric CFLS, we prove that for any constant $\varepsilon > 0$, local search yields an $(8+\varepsilon)$-approximation algorithm with no blowup in the capacity of each facility. Using the reduction of [53] that is based on solving a special case of the generalized assignment problem [52], the above results for the capacitated problems with splittable demands imply constant-factor approximation bounds for the capacitated problems with unsplittable demands (CKMU and CFLU) assuming a factor of 2 blowup in the capacity of each facility.

Recent related work. Subsequent to the publication of our results, several new approximation algorithms have been obtained for the k-median and the facility location

Problem	*Bound*	*Capacity Blowup*	*Reference*	*Approach*
metric CKMS	$(1+\varepsilon, 5+5/\varepsilon)$	none	here	local search
	$(1+5/\varepsilon, 5+\varepsilon)$	none	here	local search
	$(16, 1)$	3	[15]	LP-rounding
metric CFLS	7	7/2	[53]	LP-rounding
	$8+\varepsilon$	none	here	local search
	$6+\varepsilon$	none	[18]*	local search
metric CKMU	$(1+\varepsilon, 5+5/\varepsilon)$	2	here	local search
	$(1+5/\varepsilon, 5+\varepsilon)$	2	here	local search
	$(16, 1)$	4	[15]*	LP-rounding
metric CFLU	9	4	[53]	LP-rounding
	$16+\varepsilon$	2	here	local search

Table 2.2: Approximation bounds for capacitated facility location problems. All of the results marked with an asterisk have been obtained subsequent to our results.

problems. For the k-median problem, one shortcoming of the approximation bounds that we have discussed thus far is that while the optimal solution uses at most k facilities, the number of facilities used in the approximate solutions may exceed k by a constant factor. In recent work, Charikar et al. [15] have obtained the first constant-factor approximation algorithm for metric UKM without incurring a blowup in the number of facilities. Their algorithm, which relies on rounding the fractional solution of an appropriate linear programming relaxation, yields a $(20/3, 1)$-approximation. Arora et al. [3] give a polynomial-time approximation scheme for the special case of metric UKM in which the distances are Euclidean. The best known approximation ratio for UFL is achieved by the recent 1.74-approximation algorithm due to Chudak [16]. More recently, Jain and Vazirani [32] have obtained fast $(6, 1)$- and 3-approximation algorithms for UKM and UFL, respectively, using an elegant primal-dual approach.

Improved approximation algorithms have been recently obtained for the capacitated versions as well. Charikar et al. [15] give $(16, 1)$-approximation algorithms to both CKMS and CKMU while incurring a blowup in capacity by factors of 3 and 4, respectively.

All of the approximation bounds discussed above are summarized in Tables 2.1 and 2.2. We remark that the analysis presented in this dissertation is somewhat sharper than that provided in the preliminary version [37]; as a result, we have obtained constant-factor improvements in all of our bounds. In a recent paper, Chudak and Williamson [18] have simplified and improved our analysis of the local search heuristic for CFLS to show that the same heuristic yields a $(6+\varepsilon)$-approximation for CFLS, for any constant $\varepsilon > 0$.

The facility location problems that we consider in this dissertation are closely related to a number of other optimization problems (e.g., the k-center problem) that have been studied in the computer science and operations research literature. We refer the reader to [19, 30] for more information about other problems related to facility location.

The rest of the chapter is organized as follows. Section 2.2 formally defines the six facility location problems considered in this dissertation. Section 2.3 defines the local search heuristic that we analyze. Section 2.3.1 explains how to find an initial feasible solution. Sections 2.5 and 2.6 analyze the uncapacitated k-median and facility location problems, respectively. Sections 2.8 and 2.9 analyze the capacitated k-median problem and the capacitated facility location problem with splittable demands, respectively. Section 2.10 extends these results to capacitated problems with unsplittable demands. Finally, Appendix 2.11 presents some proofs that are deferred in Section 2.9.

2.2 Facility location problems

Let $N = \{1, \ldots, n\}$ be a set of locations and $F \subseteq N$ be a set of locations at which we may open a facility. Each location j in N has a demand d_j that must be shipped to j. For any two locations i and j, let c_{ij} denote the cost of shipping a unit of demand from i to j. We assume that the costs are nonnegative, symmetric, and satisfy the triangle inequality. In each problem studied in this dissertation, we wish to determine a set of open facilities and an assignment of locations to open facilities such that a given objective function is minimized.

The uncapacitated k-median problem (UKM). In UKM, we seek a set of at most k open facilities and an assignment of locations to open facilities such that the shipping cost of the solution is minimized, where the shipping cost associated with a set S of open facilities and an assignment $\sigma : N \mapsto S$ is given by $\sum_{j \in N} d_j c_{j\sigma(j)}$. Given a set S

of open facilities, an assignment that minimizes the shipping cost may be obtained by assigning each location j in N to the closest open facility in S. Thus, any solution to UKM is completely characterized by the set of open facilities. For any set S of open facilities, let $C_s(S)$ denote the shipping cost obtained by assigning each location to its closest open facility.

The uncapacitated facility location problem (UFL). In UFL, for each location i in F, we are given a nonnegative cost f_i of opening a facility at i. The problem is to determine a set of open facilities and an assignment of locations to facilities such that the sum of the cost of opening the facilities and the shipping cost is minimized. (The shipping cost is defined as in UKM.) As in UKM, given a set S of open facilities, an assignment that minimizes the total cost is to assign each location j in N to the closest open facility in S. Therefore, any solution to this problem is characterized by the set of open facilities. Given any solution S of open facilities, we let $C_s(S)$, $C_f(S)$, and $C(S)$ denote the shipping cost, facility cost, and the total cost, respectively, obtained by assigning each location to its closest open facility. (Formally, $C_f(S) = \sum_{i \in S} f_i$ and $C(S) = C_s(S) + C_f(S)$.)

The capacitated versions. We now define *capacitated* variants of the above two problems in which there is a bound M on the total demand that can be shipped from any facility. The capacitated variants are defined in the same manner as the uncapacitated variants except that any solution must satisfy the additional constraint imposed by the capacity M. That is, a solution given by a set S of open facilities and an assignment σ is feasible only if the capacity constraints are respected at all facilities in S. We note that we allow at most one facility to be opened at any location. For recent work that considers capacitated facility location problems in which multiple facilities may be opened at any location, see [17, 18].

There are two natural variants of capacitated location problems: (i) *splittable demands*, where the demand of each location can be split across more than one facility, and (ii) *unsplittable demands*, where the demand of each location has to be shipped from a single facility. For the capacitated problems with splittable demands (that is, CKMS and CFLS), given a set S of open facilities, an assignment is given by a function

$\sigma : N \times S \mapsto \mathbf{R}$, where $\sigma(j, i)$ denotes the amount of demand shipped from facility i to location j. For the capacitated problems with unsplittable demands (that is, CKMU and CFLU), an assignment is given by a function $\sigma : N \mapsto S$, as in the uncapacitated problems.

In the uncapacitated problems, given a set of open facilities, an optimal assignment is obtained by simply assigning each location to its closest open facility. In a capacitated problem, however, such an assignment may violate the capacity constraint. Fortunately, for the capacitated problems with splittable demands, given a set S of open facilities, an optimal assignment can be computed in polynomial time by solving an appropriately defined instance of the transportation problem [53]. Thus, for the capacitated variants with splittable demands, any solution is completely characterized by the set S of open facilities. For the capacitated problems with unsplittable demands, however, it is NP-hard to compute an optimal assignment for a given set S of open facilities. Therefore, we require a solution to one of the capacitated problems with unsplittable demands to specify a feasible assignment together with the set of open facilities.

In the k-median problems, note that the total cost for a solution S is same as the shipping cost for S. Hence we use $C(S)$ and $C_s(S)$ interchangeably for the k-median problems.

2.3 The local search paradigm

All of the algorithms considered in this chapter are based on the following framework. We start with an arbitrary feasible solution and repeatedly improve the solution by performing a local search step. We show that within a polynomial number of local search steps, we arrive at a solution achieving the desired approximation factor. The problem of finding an initial feasible solution is discussed in Section 2.3.1. We now describe the local search step.

Let us consider the uncapacitated problems UKM and UFL and the capacitated problems with splittable demands CKMS and CFLS. Recall that any solution to these problems is completely characterized by the associated set S of open facilities. For the k-median problem, we define the *neighborhood* of S as $\{T \subseteq F : |S - T| = |T - S| = 1\}$.

For the facility location problem, we define the neighborhood of S as $\{T \subseteq F : |S - T| \leq 1 \text{ and } |T - S| \leq 1\}$. Given a current solution corresponding to a set S of open facilities, the local search step sets the new solution to be a minimum-cost set T in the neighborhood of S. Since the neighborhood contains $O(n^2)$ solutions, a local search step can be performed in polynomial time.

The main motivation behind using the local search step is the following key property of the neighborhood of any solution. Let S^* be an optimal solution. Then, for a sufficiently large constant δ, given any solution S such that $C(S) > (1+\delta)C(S^*)$ (for the k-median problems, we also require that $|S| = (1+\alpha)k$ for some sufficiently large constant α), there exists a solution T in the neighborhood of S such that $C(T) \leq C(S)(1 - 1/p(n))$, where $p(n)$ is a fixed polynomial in n. (This property is proved in Theorems 2.5.1, 2.6.1, 2.8.1, and 2.9.1 for UKM, UFL, CKMS, and CFLS, respectively.)

Thus, if we start with a solution S_0 and perform $p(n)\log(C(S_0)/C(S^*))$ local search steps, we will arrive at a solution that has cost at most $(1+\delta)C(S^*)$. Since $\log(C(S_0))$ is polynomial in the input size, the number of local search steps involved is also polynomial in the input size. Moreover, each local search step takes polynomial time. We thus obtain a polynomial-time algorithm achieving an approximation factor of $(1+\delta)$, for a sufficiently large constant δ.

2.3.1 Finding an initial feasible solution

In this subsection we show how to obtain an initial feasible solution for the uncapacitated problems and the capacitated problems with splittable demands. The task of selecting a valid set of open facilities is trivial. For the facility location problems, we select an arbitrary subset S of F. For the k-median problems, we require that $|S| = (1+\alpha)k$ for some sufficiently large constant α. In the extreme case where the total number of available facilities, $|F|$, is less than $(1+\alpha)k$, our algorithm trivially returns F as the solution. Hence throughout this chapter, we assume without loss of generality that $|F|$ is at least $(1+\alpha)k$ for all k-median problems. Given this assumption, we obtain an initial feasible solution for the k-median problems by selecting an arbitrary set S of

$(1+\alpha)k$ facilities.

As mentioned in Section 2.2, the set S of open facilities chosen above completely specifies the initial feasible solution. This is because we can obtain an optimal assignment for the initial solution by simply assigning each location to its closest open facility in the case of the uncapacitated problems, and by solving an appropriately defined instance of the transportation problem in the case of the capacitated problems with splittable demands.

2.4 Preliminaries for uncapacitated problems

In this section we introduce some notations and definitions that are useful for our analysis of UKM in Section 2.5 and UFL in Section 2.6.

Given a set S of open facilities and an assignment σ for S, let $N_i(S,\sigma)$ denote the set of locations assigned to a facility i in S. A location j is called a *client* of facility i, under σ, if σ assigns j to i. Let $D_i(S,\sigma)$ denote the total demand shipped from a facility i in S under assignment σ. For notational convenience, the parameters S and/or σ may be dropped from the expressions $N_i(S,\sigma)$ and $D_i(S,\sigma)$ when there is no ambiguity.

Recall that an optimal assignment for a set of open facilities S assigns every client to its closest open facility. Let S^* (resp., S) denote an optimal (resp., current) set of open facilities for the problem instance under consideration, and let σ^* (resp., σ) denote an optimal assignment for S^* (resp., S). Given S, S^*, σ, and σ^*, we define g_i and g_i^* (for each facility i in S) as follows: g_i denotes the shipping cost paid by the clients of i, that is, $g_i = \sum_{j \in N_i(S,\sigma)} d_j c_{ji}$. Correspondingly, g_i^* denotes the shipping cost paid by the same set of clients (i.e., $N_i(S,\sigma)$) in the optimal solution, that is, $g_i^* = \sum_{j \in N_i(S,\sigma)} d_j c_{j\sigma^*(j)}$. Note that $C_s(S) = \sum_{i \in S} g_i$ and $C_s(S^*) = \sum_{i \in S} g_i^*$.

2.5 Uncapacitated k-median

In this section, we analyze the local search heuristic for UKM. Throughout this section, we assume that $p(n)$ is a polynomial in n and that α and β are positive constants satisfying the following condition:

$$(1+\beta)\left(1-\frac{2}{\alpha}-\frac{k}{p(n)}\right)\geq 1+\frac{2}{\alpha}. \tag{2.1}$$

One possible choice, for example, is $\alpha = 4$, $\beta = 3$, and $p(n) = 8k$. The main result of this section is that, given a solution S of $(1+\alpha)k$ facilities with cost greater than $(1+\beta)C(S^*)$, we can perform a swapping of facilities to get another solution with the same size but significantly reduced cost.

Theorem 2.5.1 (Swapping facilities) *Let S be any subset of F such that $|S| = (1+\alpha)k$ and $C(S) > (1+\beta)C(S^*)$. Then there exist u in S and v in F such that $C(S) - C(S+v-u) \geq \frac{C(S)}{p(n)}$.*

We prove the theorem in two stages. First, in Lemma 2.5.1 we show that we can add a facility to S and get a significant reduction in the cost. Then, in Lemma 2.5.2, we show that we can drop a facility from S without increasing the cost too much. Combining these two lemmas then yields the above theorem.

Lemma 2.5.1 (Adding a facility) *Let S be any subset of F such that $C(S) > C(S^*)$. Then there exists v in F such that $C(S) - C(S+v) \geq \frac{C(S)-C(S^*)}{k}$.*

Proof: Recall that σ and σ^* denote the optimal assignments for S and S^*, respectively. We have

$$C(S) - C(S^*) \;=\; \sum_{j\in N} d_j(c_{j\sigma(j)} - c_{j\sigma^*(j)}) = \sum_{i\in S^*}\sum_{j\in N_i(S^*)} d_j(c_{j\sigma(j)} - c_{ji}).$$

Since $|S^*| = k$, it follows from a simple averaging argument that there exists v in S^* for which

$$\sum_{j\in N_v(S^*)} d_j(c_{j\sigma(j)} - c_{jv}) \geq \frac{C(S)-C(S^*)}{k}.$$

Now consider the set $S+v$ along with the assignment σ' defined as follows: Set $\sigma'(j)$ to v if j belongs to $N_v(S^*)$, and to $\sigma(j)$ otherwise. Using the fact that $C(S+v,\sigma') \geq C(S+v)$, we obtain

$$C(S) - C(S+v) \;\geq\; C(S) - C(S+v,\sigma') = \sum_{j\in N_v(S^*)} d_j(c_{j\sigma(j)} - c_{jv}) \geq \frac{C(S)-C(S^*)}{k}.$$

■

Lemma 2.5.2 (Dropping a facility) *Let S be any subset of F such that $|S| = (1 + \alpha)k$. Then there exists u in S such that $C(S - u) - C(S) \leq 2 \cdot \frac{C(S)+C(S^*)}{\alpha k}$.*

Before proceeding to prove this lemma, we first classify the facilities in S as "primary" or "secondary" as follows. For each facility i^* in S^*, consider those facilities of S that are assigned to i^* by σ^* and pick the closest one among these, breaking ties arbitrarily. The closest facility thus chosen is said to be the *primary* facility of i^*. Let S_P denote the set of all primary facilities. Facilities which are not primary are said to be *secondary*. For each secondary facility i in S, we define the associated primary facility as the primary facility of $\sigma^*(i)$. Note that the number of secondary facilities is at least $|S| - |S^*|$, which is αk. We will show that one of the secondary facilities can be dropped from S without incurring a "large" increase in the shipping cost. As a first step in this direction, the following claim gives an upper bound on the increase in the shipping cost if a client of a secondary facility i is reassigned to the primary facility associated with i.

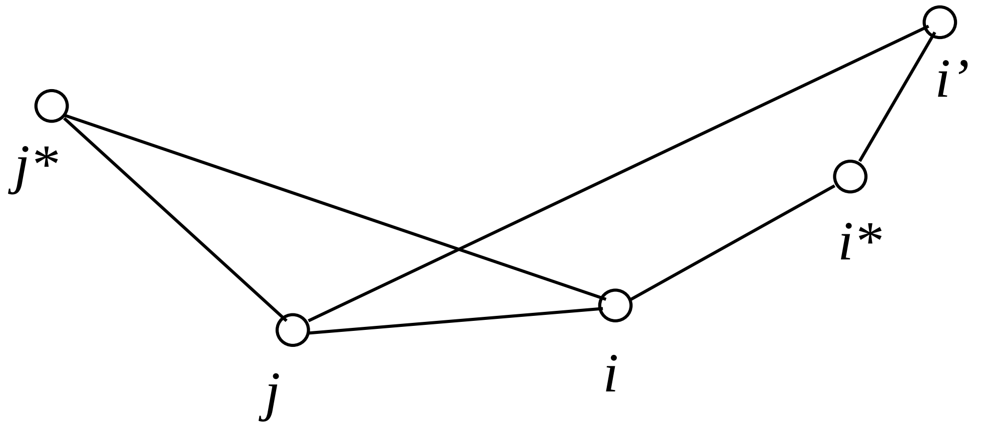


Figure 2.1: Illustration for Claim 2.5.1. Here, $i^* = \sigma^*(i)$, $j^* = \sigma^*(j)$, and i' is the primary facility of i^*.

Claim 2.5.1 *Consider a secondary facility i in S, and let i' be the primary facility associated with i. Then for all j in N, we have $c_{ji'} - c_{ji} \leq 2(c_{ji} + c_{j\sigma^*(j)})$.*

Proof: Let i^* denote $\sigma^*(i)$, the facility in S^* that is closest to i. (See Figure 2.1.) Using the triangle inequality, we upper bound $c_{ji'}$ as follows:

$$c_{ji'} \leq c_{ji} + c_{ii^*} + c_{i^*i'} \leq c_{ji} + 2c_{ii^*} \leq c_{ji} + 2c_{i\sigma^*(j)} \leq c_{ji} + 2c_{ij} + 2c_{j\sigma^*(j)}.$$

(The second step uses the fact that i' is the primary associated with i^*, i.e., $c_{i^*i'} \leq c_{ii^*}$. The third step holds since i^* is the closest to i among all facilities in S^*. The fourth step uses the triangle inequality.) The claim now follows. ■

Claim 2.5.2 *Consider a secondary facility i in S and let i' be the primary facility associated with i. If we drop the facility i from S and reassign all clients of i to i', then the increase in the shipping cost is at most $2(g_i + g_i^*)$.*

Proof: It is easy to verify that the increase in shipping cost due to this reassignment is $\sum_{j \in N_i(S)} d_j(c_{ji'} - c_{ji})$. Using the upper bound on $c_{ji'} - c_{ji}$ given by Claim 2.5.1, and then plugging in the definitions of g_i and g_i^*, completes the proof. ■

The following claim is needed only for the proof of Theorem 2.6.1 in Section 2.6, but is in the same spirit as the previous claim.

Claim 2.5.3 *Consider any facility i in S and let i^* denote $\sigma^*(i)$. If we drop facility i from S and reassign all clients of i to i^*, then the increase in the shipping cost is at most $g_i + g_i^*$.*

Proof: For every client j of i, we can upper bound $c_{ji^*} - c_{ji}$ by $c_{ji} + c_{j\sigma^*(j)}$ using the same approach as in Claim 2.5.1. The remainder of the proof is similar to that of Claim 2.5.2. ■

We are now ready to prove Lemma 2.5.2.

Proof of Lemma 2.5.2: Recall that S_P is the set of primary facilities of S. Using the definitions of g_i and g_i^*, we have

$$\sum_{i \in S - S_P} (g_i + g_i^*) \leq \sum_{i \in S} (g_i + g_i^*) = C_s(S) + C_s(S^*).$$

By an averaging argument, it follows that there exists a secondary facility v for which $g_v + g_v^*$ is at most $\frac{C_s(S)+C_s(S^*)}{\alpha k}$ (since $|S - S_P| \geq \alpha k$). Applying Claim 2.5.2 for this facility v, it follows that v satisfies the requirements of this dropping lemma. ■

Proof of Theorem 2.5.1: We first apply Lemma 2.5.2 to drop a facility u from S. We then apply Lemma 2.5.1 to add a facility v to $S - u$. Combining the two operations, we have

$$C(S) - C(S-u+v) \;\; \geq \;\; \frac{C(S-u) - C(S^*)}{k} - 2 \cdot \frac{C(S) + C(S^*)}{\alpha k}.$$

It now suffices to show that the term on the right side is at least $\frac{C(S)}{p(n)}$. Since $C(S-u) \geq C(S)$, it suffices to show that

$$C(S)\left(1 - \frac{2}{\alpha} - \frac{k}{p(n)}\right) \geq C(S^*)\left(1 + \frac{2}{\alpha}\right).$$

Since $C(S) \geq (1+\beta)C(S^*)$ and α, β, and $p(n)$ satisfy Equation (2.1), this inequality always holds. This completes the proof of the theorem. ■

The corollary below is related to the question of how small the constants α and β can be. Note that any choice of α and β subject to the constraint of Equation (2.1) is valid. It is shown below that β can be made arbitrarily small, while on the other hand, α can be made arbitrarily close to 2.

Corollary 2.5.1.1 *For any constant $\varepsilon > 0$, there is a constant δ such that the following can be computed in polynomial time using the local search heuristic: (i) a solution S with $C(S) \leq (1+\varepsilon)C(S^*)$ and $|S| \leq (1+\delta)k$, and (ii) a solution S with $|S| \leq (3+\varepsilon)k$ and $C(S) \leq (1+\delta)C(S^*)$.*

Proof: It can be verified that the following choices of the constants satisfy the constraint of Equation (2.1): (i) $\beta = \varepsilon$, $\delta = \alpha = \frac{5}{\varepsilon} + 2$, and $p(n) = \alpha k(1+\varepsilon)$, and (ii) $\alpha = 2 + \varepsilon$, $\delta = \beta = 1 + \frac{5}{\varepsilon}$, and $p(n) = \frac{5\alpha k}{\varepsilon}$.

The first part of the corollary follows by applying Theorem 2.5.1 with the first set of constants and using the local search algorithm described in Section 2.3. For the second part of the corollary, we use the second set of constants. ■

The following corollary addresses the issue of making α arbitrarily small. The local improvement method requires α to satisfy the constraint of Equation (2.1) and hence it has to be above 2. However if we postprocess the solution from the local improvement phase by dropping facilities greedily, we can achieve arbitrarily small α while keeping the cost within a constant factor of optimal.

Corollary 2.5.1.2 *For any constant $\varepsilon > 0$, there is a constant δ such that a solution S with $|S| \le (1+\varepsilon)k$ and $C(S) \le (1+\delta)C(S^*)$ can be computed in polynomial time using the local search heuristic.*

Proof: Suppose we have a solution S with $|S| = (1+\gamma)k$ and $C(S) \le (1+\mu)C(S^*)$. For example, by part (ii) of Corollary 2.5.1.1, we can choose $\gamma = 2+\varepsilon'$ and $\mu = 1+\frac{5}{\varepsilon'}$ for any $\varepsilon' > 0$.

Lemma 2.5.2 implies that there exists a facility u in S that can be dropped without increasing the cost by more than $\frac{2C(S)+2C(S^*)}{\gamma k}$, which is at most $\frac{4C(S)}{\gamma k}$ assuming $C(S^*) \le C(S)$. Therefore, the cost of $S - u$ is at most $C(S)(1+\frac{4}{\gamma k})$. We repeat this dropping process until the number of facilities reduces to $(1+\varepsilon)k$. Let $\hat{S}$ be the set of facilities remaining at the end of this process. Since the $(t+1)$th facility dropped increases the cost by at most a multiplicative factor of $(1+\frac{4}{\gamma k - t})$, it follows that

$$\begin{aligned} C(\hat{S}) &\le C(S)\left[\left(1+\frac{4}{\gamma k}\right)\left(1+\frac{4}{\gamma k - 1}\right)\cdots\left(1+\frac{4}{\varepsilon k+1}\right)\right] \\ &= C(S)\frac{(\gamma k+4)(\gamma k+3)(\gamma k+2)(\gamma k+1)}{(\varepsilon k+4)(\varepsilon k+3)(\varepsilon k+2)(\varepsilon k+1)} \\ &\le C(S)\left(\frac{\gamma}{\varepsilon}\right)^4. \end{aligned}$$

(The second step follows by expanding and canceling out the common terms in the numerator and the denominator. The third step uses the fact that $\frac{\gamma k+i}{\varepsilon k+i} \le \frac{\gamma}{\varepsilon}$, whenever $i \ge 0$ and $\gamma \ge \varepsilon$.) Thus $\hat{S}$ is a solution with $|\hat{S}| = (1+\varepsilon)k$, and if γ is chosen to be $2+\varepsilon'$ (for any $\varepsilon' > 0$) we have

$$C(\hat{S}) \le C(S^*)\left(2+\frac{5}{\varepsilon'}\right)\left(\frac{2+\varepsilon'}{\varepsilon}\right)^4.$$

The corollary follows if we set the constant δ to $\left(2+\frac{5}{\varepsilon'}\right)\left(\frac{2+\varepsilon'}{\varepsilon}\right)^4 - 1$, for any $\varepsilon' > 0$.

■

2.6 Uncapacitated facility location

The main result of this section is Theorem 2.6.1, which shows that a local search step for UFL gives a significant improvement in the cost of the current solution if the cost of the current solution is sufficiently larger than the optimal cost. Throughout this section, we assume that $p(n)$ is a sufficiently large polynomial in n.

Theorem 2.6.1 *Let S be any subset of F such that $C(S) > (5+\varepsilon)C(S^*)$, for some constant $\varepsilon > 0$. Then there exists $T \subseteq F$ such that $|S - T| \leq 1$, $|T - S| \leq 1$, and $C(S) - C(T) \geq \frac{C(S)}{p(n)}$.*

Corollary 2.6.1.1 *For any constant $\varepsilon > 0$, the local search heuristic yields a $(5+\varepsilon)$-approximate solution in polynomial time.* ■

We prove the theorem in two stages. First, in Lemma 2.6.1, we show that if the shipping cost of S is sufficiently large then we can add a facility to S and obtain a significant improvement in the cost. Second, in Lemma 2.6.2, we show that if the facility cost of S is sufficiently large then we can either drop a facility from S or substitute a facility in S to obtain a significant improvement in the cost. If none of adding, dropping, or substituting gives a good improvement in the cost, then we combine these two lemmas to obtain a contradictory upper bound on the total cost of S.

Lemma 2.6.1 (Adding a facility) *Let S be any subset of F such that $C_s(S) \geq C(S^*) + \frac{nC(S)}{p(n)}$. Then there exists v in F such that $C(S) - C(S+v) \geq \frac{C(S)}{p(n)}$.*

Proof: It suffices to show that for any set $S \subseteq F$, there exists v in F, such that $C(S) - C(S+v) \geq \frac{C_s(S) - C(S^*)}{n}$. The quantity $C_s(S) - C(S^*)$ can be expanded as follows:

$$\begin{aligned}
C_s(S) - C_s(S^*) - C_f(S^*) &= -C_f(S^*) + \sum_{j \in N} d_j(c_{j\sigma(j)} - c_{j\sigma^*(j)}) \\
&= \sum_{i \in S^*} -f_i + \sum_{i \in S^*} \sum_{j \in N_i(S^*)} d_j(c_{j\sigma(j)} - c_{ji}) \\
&= \sum_{i \in S^*} \left(-f_i + \sum_{j \in N_i(S^*)} d_j(c_{j\sigma(j)} - c_{ji}) \right).
\end{aligned}$$

By a simple averaging argument, it follows that there exists a facility v in S^* such that

$$-f_v + \sum_{j \in N_v(S^*)} d_j(c_{j\sigma(j)} - c_{jv}) \geq \frac{C_s(S) - C(S^*)}{n}. \tag{2.2}$$

Now consider the solution $S+v$ along with the assignment σ' defined as follows: If j is in $N_v(S^*)$, then set $\sigma'(j)$ to v; otherwise set $\sigma'(j)$ to $\sigma(j)$. Since $C(S+v,\sigma')$ is at least $C(S+v)$, Equation (2.2) implies that

$$\begin{aligned} C(S) - C(S+v) &\geq C(S) - C(S+v,\sigma') = -f_v + \sum_{j \in N_v(S^*)} d_j(c_{j\sigma(j)} - c_{jv}) \\ &\geq \frac{C_s(S) - C(S^*)}{n}, \end{aligned}$$

completing the proof of the lemma. ■

Lemma 2.6.2 (Dropping or swapping facilities) *Let S be any subset of F. If no facility in S can be dropped or substituted (by some facility not in S) to improve the cost by at least $\frac{C(S)}{p(n)}$, then the following upper bound on the facility cost of S holds:*

$$C_f(S)\left(1 - \frac{n^2}{p(n)}\right) < 2C(S^*) + 2C_s(S) + \frac{C(S)}{n}.$$

We start by developing some useful tools for proving this lemma. A facility i in S is called *cheap* if $f_i \leq \frac{C(S)}{n^2}$, and *expensive* otherwise. Let S_C denote the set of cheap facilities in $S - S^*$. Each facility in S is also classified as *primary* or *secondary* in the same fashion as in Section 2.5. We let S_{EP} (resp., S_{ES}) denote the set of expensive facilities in $S - S^* - S_C$ that are classified as primary (resp., secondary). Thus $S \cap S^*$, S_C, S_{EP}, and S_{ES} partition the set S. Note that for each i in S_{EP}, the facility $i^* = \sigma^*(i)$ cannot be in S; if i^* belonged to S, then i^*, not i, would have been designated as the primary facility for i^*.

We start by constructing a set of *candidate operations*, one for each facility in S, as follows:

1. For i in $S \cap S^*$, do nothing.
2. For i in S_C, do nothing.
3. For i in S_{ES}, drop i and reassign the clients of i to the primary facility i' associated with i. Note that i' is already in S.
4. For i in S_{EP}, substitute i with $i^* = \sigma^*(i)$, and reassign all clients of i to i^*. Recall that i^* does not belong to S.

A candidate operation is called *good* if it reduces the cost of the solution by at least $\frac{C(S)}{p(n)}$, and *bad* otherwise. By the hypothesis of Lemma 2.6.2, we can assume that all of the candidate operations are *bad*. Our main tools for proving Lemma 2.6.2 are Claims 2.6.1 and 2.6.2, which establish the following: If the candidate operation for a facility i is bad, then the facility cost of i is "small".

We first consider S_C, the set of cheap facilities. Since the total number of facilities is at most n and since the cost of each cheap facility is at most $C(S)/n^2$, it trivially follows that

$$C_f(S_C) \leq \frac{C(S)}{n}. \tag{2.3}$$

Claim 2.6.1 *If the candidate operation for every facility in S_{ES} is bad, then*

$$C_f(S_{ES})\left(1-\frac{n^2}{p(n)}\right) \leq \sum_{i\in S_{ES}} 2(g_i+g_i^*).$$

Proof: Consider an expensive secondary facility i in S_{ES}. The decrease in the facility cost caused by the associated candidate operation is f_i and the increase in the shipping cost, by Claim 2.5.2, is at most $2(g_i+g_i^*)$. So the net decrease in the cost is at least $f_i - 2(g_i+g_i^*)$. If this quantity is greater than $\frac{n^2 f_i}{p(n)}$, then since i is expensive, this candidate operation will be good. Therefore, if this candidate operation is not good, the following upper bound on the facility cost of i holds: $f_i(1-\frac{n^2}{p(n)}) < 2(g_i+g_i^*)$. Summing over all of the expensive secondary facilities yields the required upper bound on the facility cost of S_{ES}. ■

Claim 2.6.2 *If the candidate operation for every facility in S_{EP} is bad, then*

$$C_f(S_{EP})\left(1-\frac{n^2}{p(n)}\right) \leq C_f(S^*-S) + \sum_{i\in S_{EP}} (g_i+g_i^*).$$

Proof: Consider an expensive primary facility i in S_{EP}. The decrease in the facility cost caused by its candidate operation is $f_i - f_{i^*}$ and the increase in the shipping cost, by Claim 2.5.3, is at most $g_i+g_i^*$. So the net decrease in the cost is at least $f_i-(g_i+g_i^*+f_{i^*})$. If this quantity is greater than $\frac{n^2 f_i}{p(n)}$, then since i is expensive, this candidate operation is good. Therefore, if this candidate operation is not good, the following upper bound on the facility cost of i holds: $f_i(1 - \frac{n^2}{p(n)}) < g_i + g_i^* + f_{i^*}$. Summing over all of the expensive primary facilities yields

$$C_f(S_{EP})\left(1 - \frac{n^2}{p(n)}\right) \quad < \quad \sum_{i \in S_{EP}} (g_i + g_i^* + f_{i^*}) < C_f(S^* - S) + \sum_{i \in S_{EP}} (g_i + g_i^*).$$

(The second step holds since $\sigma^*(i)$ belongs to $S^* - S$ for all i in S_{EP} and $\sigma^*(i) \neq \sigma^*(j)$ for all i and j in S_{EP} such that $i \neq j$. Hence, each facility of $S^* - S$ occurs at most once in the summation.) ■

Proof of Lemma 2.6.2: Using the upper bounds from Equation (2.3), Claim 2.6.1, and Claim 2.6.2, and adding $C_f(S \cap S^*)$ to both sides, we obtain

$$\begin{aligned} C_f(S)\left(1 - \frac{n^2}{p(n)}\right) \quad &< \quad \frac{C(S)}{n} + C_f(S^*) + 2 \cdot \sum_{i \in S_{EP} \cup S_{ES}} (g_i + g_i^*) \\ &< \quad \frac{C(S)}{n} + C_f(S^*) + 2C_s(S) + 2C_s(S^*) \\ &< \quad \frac{C(S)}{n} + 2C_s(S) + 2C(S^*). \end{aligned}$$

(The second step follows from the definition of g_i and g_i^*, and the fact that the sets S_{EP} and S_{ES} are disjoint.) ■

Proof of Theorem 2.6.1: If no facility can be added to S to get a cost improvement of at least $\frac{C(S)}{p(n)}$, then by Lemma 2.6.1, the shipping cost of S is upper-bounded as follows: $C_s(S) < C(S^*) + \frac{nC(S)}{p(n)}$. Similarly, if no facility of S can be dropped or substituted to get a cost improvement of at least $\frac{C(S)}{p(n)}$, then by Lemma 2.6.2 we have the following upper bound on the facility cost of S:

$$C_f(S)\left(1 - \frac{n^2}{p(n)}\right) \quad < \quad 2C(S^*) + 2C_s(S) + \frac{C(S)}{n} < 4C(S^*) + nC(S)\left(\frac{2}{p(n)} + \frac{1}{n^2}\right).$$

(The second step follows from the upper bound on $C_s(S)$.) Adding the upper bounds on the shipping cost and the facility cost, we have

$$C(S)\left(1 - \frac{n^2}{p(n)}\right) < 5C(S^*) + nC(S)\left(\frac{3}{p(n)} + \frac{1}{n^2}\right).$$

Rearranging, we obtain

$$C(S)\left(1 - \frac{n^2}{p(n)} - \frac{3n}{p(n)} - \frac{1}{n}\right) < 5C(S^*).$$

If $p(n)$ is chosen to be sufficiently large, then this contradicts the hypothesis (of Theorem 2.6.1) that $C(S) > (5+\varepsilon)C(S^*)$. Specifically, we need

$$(5+\varepsilon)\left(1 - \frac{n^2}{p(n)} - \frac{3n}{p(n)} - \frac{1}{n}\right) > 5.$$

This completes the proof of the theorem. ■

2.7 Preliminaries for capacitated problems with splittable demands

In this section we introduce some notations and definitions that are useful in our analysis of the capacitated problems with splittable demands, CKMS (see Section 2.8) and CFLS (see Section 2.9).

Recall that for the capacitated problems with splittable demands, given a set S of open facilities, an assignment is given by a function $\sigma : N \times S \mapsto \mathbf{R}$, where $\sigma(j,i)$ denotes the amount of demand shipped from facility i to location j. Given S and an assignment σ, let $D_v(S,\sigma)$ denote the amount of demand shipped from a facility v in S under the assignment σ, that is, $D_v(S,\sigma) = \sum_{j\in N} \sigma(j,v)$. For notational convenience, we sometimes drop the parameters S and/or σ when there is no ambiguity.

Also recall that given a set S of open facilities, the corresponding optimal splittable assignment σ can be computed in polynomial time by a reduction to the transportation problem [53]. Hence, any solution to CKMS or CFLS is completely characterized by the set S of open facilities.

We let S^* (resp., S) denote an optimal (resp., current) solution for a given instance of CKMS or CFLS, and let σ^* (resp., σ) denote an optimal assignment for S^* (resp., S). Recall that M denotes the upper bound on the capacity of any facility.

2.8 Capacitated k-median with splittable demands

In this section we consider the capacitated k-median problem with splittable demands (CKMS) and show in Theorem 2.8.1 that, as in the uncapacitated case, each step of the local search heuristic yields a significant improvement in cost whenever the current cost is sufficiently higher than the optimal cost. Throughout this section, we assume that $p(n)$ is a polynomial in n and that α and β are positive constants satisfying the following constraint,

$$(1+\beta)\left(1-\frac{2}{\alpha}-\frac{k}{p(n)}\right) \geq 1+\frac{2}{\alpha}. \tag{2.4}$$

Note that this constraint is the same as that of Equation (2.1). Hence, as before, one possible choice for the parameters is $\alpha = 4$, $\beta = 3$, and $p(n) = 8k$.

Theorem 2.8.1 *Let S be any subset of F such that $|S| = (3+\alpha)k$ and $C(S) > (1+\beta)C(S^*)$. Then there exist u in S and v in F such that $C(S) - C(S+v-u) \geq \frac{C(S)}{p(n)}$.*

As in the uncapacitated case, we prove this theorem in two stages. First, in Lemma 2.8.1, we show that we can add a facility to S and get a significant reduction in the cost. Due to the capacity constraints, the proof of this lemma is more involved than in the uncapacitated case (Lemma 2.5.1). Section 2.8.1 gives the proof of this lemma after developing the required technical machinery. Then, in Lemma 2.8.2, we show that we can drop a facility from S without increasing the cost too much. The proof of this lemma is given in Section 2.8.2.

Lemma 2.8.1 (Adding a facility) *Let S be any subset of F such that $C(S) > C(S^*)$. Then there exists v in F such that $C(S) - C(S+v) \geq \frac{C(S)-C(S^*)}{k}$.*

Lemma 2.8.2 (Dropping a facility) *Let S be any subset of F such that $|S| = (3+\alpha)k$. Then there exists u in S such that $C(S-u) - C(S) \leq 2 \cdot \frac{C(S)+C(S^*)}{\alpha k}$.*

Theorem 2.8.1 follows directly from Lemmas 2.8.1 and 2.8.2; the proof is along the same lines as the uncapacitated case (Theorem 2.5.1). We now turn to the questions of how small the constants α and β can be. Recall that the constraint of Equation (2.4) to be satisfied by these constants is the same as in the uncapacitated case. Hence, we obtain the following corollaries to Theorem 2.8.1, which are analogous to Corollaries 2.5.1.1 and 2.5.1.2 in the uncapacitated case. The only difference is an additive term of $2k$ in the number of facilities used.

Corollary 2.8.1.1 *For any constant $\varepsilon > 0$, there is a constant δ such that the following can be computed in polynomial time using the local search heuristic: (i) a solution S with $C(S) \leq (1+\varepsilon)C(S^*)$ and $|S| \leq (1+\delta)k$, and (ii) a solution S with $|S| \leq (5+\varepsilon)k$ and $C(S) \leq (1+\delta)C(S^*)$.* ■

Corollary 2.8.1.2 *For any constant $\varepsilon > 0$, there is a constant δ such that a solution S with $|S| \leq (3+\varepsilon)k$ and $C(S) \leq (1+\delta)C(S^*)$ can be computed in polynomial time.* ■

2.8.1 Adding a facility

In this section, we prove Lemma 2.8.1. We begin by developing some technical machinery that is useful in analyzing the differences between the two solutions S and S^*. Let σ and σ^* be any optimal (splittable) assignments for S and S^*, respectively. In any solution to the uncapacitated problem UKM, each client is assigned to a unique facility, namely, the closest facility in the given solution. The preceding property is useful in providing a simple expression for the difference in the shipping costs of the two solutions S and S^* as a summation over all facilities in S^* (see the proof of Lemma 2.5.1). The same is not true, however, for CKMS since a splittable assignment may split the demand of a client among multiple facilities. To overcome this hurdle, we partition the demand of each client into smaller entities, which we refer to as *chunks*, such that each chunk is shipped from a unique facility in S (resp., S^*) under assignment σ (resp., σ^*). We now describe this partitioning in Lemma 2.8.3.

Lemma 2.8.3 (Splitting into chunks) *Let σ and σ^* be any assignments for S and S^* respectively. Then for each client j, there exists a partition of the demand d_j into a finite number of chunks, each of which has an associated size, such that the following four properties hold:*

A1. *Each chunk is shipped from a unique facility in S under assignment σ.*

A2. *For each facility i in S, the total size of all chunks of d_j shipped from i under assignment σ is $\sigma(j,i)$.*

A3. *Each chunk is shipped from a unique facility in S^* under assignment σ^*.*

A4. *For each facility i^* in S^*, the total size of all chunks of d_j shipped from i^* under assignment σ^* is $\sigma^*(j, i^*)$.*

Proof: Fix a client j. The assignment σ splits the demand d_j among the facilities in S. Thus, for each facility i in S that satisfies $\sigma(j, i) > 0$, σ defines a chunk of size $\sigma(j, i)$ that is shipped from i. Let X denote the set of these chunks. While the set X satisfies properties A1 and A2 X may not satisfy properties A3 and A4. We now further partition each chunk in X into smaller chunks and obtain the desired set Y of chunks satisfying all of the four properties.

Consider any chunk α of client j in X. Let i be the facility that ships α in σ. For each i^* in S^* such that $\sigma^*(j, i^*) > 0$, we include a chunk α_{i^*} of size $s\sigma^*(j, i^*)/d_j$ in Y, where s is the size of α. We stipulate that chunk α_{i^*} is shipped from i in σ and from i^* in σ^*. Since Y is obtained by separately dividing each chunk of X into smaller chunks, Y still satisfies properties A1 and A2 above. Moreover, by construction, Y satisfies properties A3 and A4. ■

We note that while Lemma 2.8.3 states that each chunk is shipped from a unique facility under σ (resp., σ^*), it does not preclude two distinct chunks associated with the same client from being shipped from the same facility under σ (resp., σ^*).

Using the above notion of chunks, one can attempt to extend the proof of Lemma 2.5.1 for the uncapacitated problem UKM to apply for the capacitated problem CKMS. We now illustrate why such a direct argument fails. For each chunk α, let $size(\alpha)$ denote the size of the chunk, $cl(\alpha)$ denote the client associated with the chunk, and $f(\alpha)$ (resp., $f^*(\alpha)$) denote the facility that ships α in assignment σ (resp., σ^*). We can now write the difference between $C_s(S)$ and $C_s(S^*)$ as follows:

$$
\begin{aligned}
C_s(S) - S_s(S^*) &= \sum_{j \in N} \sum_{\alpha : cl(\alpha) = j} size(\alpha)(c_{jf(\alpha)} - c_{jf^*(\alpha)}) \\
&= \sum_{i \in S^*} \sum_{\alpha : f^*(\alpha) = i} size(\alpha)(c_{cl(\alpha)f(\alpha)} - c_{cl(\alpha)i}).
\end{aligned}
$$

Furthermore, by a simple averaging argument, we obtain that there exists a v in S^* for which

$$\sum_{\alpha : f^*(\alpha) = v} size(\alpha)(c_{cl(\alpha)f(\alpha)} - c_{cl(\alpha)v}) \geq \frac{C_s(S) - C_s(S^*)}{k}.$$

If v is not in S, then v can be added to S and the demands reassigned to obtain a new feasible assignment, as in Lemma 2.5.1, without violating any facility capacities. In this case, we have the desired claim: $C_s(S) - C_s(S+v) \geq (C_s(S) - C_s(S^*))/k$. If v is in S, however, then assigning more demand to v may violate its capacity constraint, and hence the argument of Lemma 2.5.1 no longer applies. In order to address this problem, we introduce the notion of a "difference graph" that captures the differences in the assignments σ and σ^*. We then use this difference graph to show that there exists a facility in $S^* - S$ that satisfies the requirement of the lemma.

Given the set τ of all chunks, taken over all of the clients, we construct a directed (multi)graph $G(\sigma, \sigma^*)$ (or simply G, when the context is clear) in which we associate a label, a cost, and a size with each arc. The node set is $S \cup S^*$. The arc set is in one-to-one correspondence with the set τ, and is constructed as follows: For each chunk α that is shipped from facilities i and i^* in σ and σ^*, respectively, we add a directed arc $\hat{a}$ from i to i^*. This arc $\hat{a}$ is labeled by the corresponding chunk α. The size of $\hat{a}$, denoted $size(\hat{a})$, is set to the $size(\alpha)$. The cost of $\hat{a}$, denoted $cost(\hat{a})$, is set to $c_{ji} - c_{ji^*}$, where j is $cl(\alpha)$. The directed graph G thus constructed is called the *difference graph*.

Note that the difference graph G may contain cycles, self-loops, as well as multiple arcs between the same pair of nodes. Furthermore, there are no outgoing arcs for nodes in $S^* - S$, and no incoming arcs for nodes in $S - S^*$.

For notational convenience, throughout this section we interpret $D_v(S, \sigma)$ as zero whenever v does not belong to S. (Recall that for v in S, $D_v(S, \sigma)$ denotes the total demand that is shipped from facility v in solution S.) The following lemma is immediate from Lemma 2.8.3 and the definition of the difference graph.

Lemma 2.8.4 *Let σ and σ^* be any assignments for S and S^*, respectively. Let $G(\sigma, \sigma^*)$ be the difference graph for assignments σ and σ^*. Then, for each v in $S \cup S^*$, the total size of all arcs coming into v is $D_v(S^*, \sigma^*)$, and the total size of all arcs going out of v is $D_v(S, \sigma)$.* ■

We can view the difference graph as a flow network in which the capacity of every arc is simply equal to the size of the arc. Now consider a flow that saturates every arc in the network; that is, let the flow along any arc equal to the size (hence, capacity) of the arc. Using network flow terminology, we say that a node v in the difference graph is a *surplus* node if the total size of all arcs coming into v is greater than the total size of all arcs going out of v. For a surplus node v, the quantity $D_v(S^*, \sigma^*) - D_v(S, \sigma)$ is called the *excess* at v. We now invoke the well-known flow decomposition theorem (see, for example, Theorem 3.5 of [1]) to decompose the difference graph into a set $\mathcal{C}$ of directed cycles and a set $\mathcal{P}$ of directed paths that satisfy some useful properties.

Lemma 2.8.5 (Decomposition of G) *There exists a decomposition of the difference graph G into a set of cycles $\mathcal{C}$ and a set of paths $\mathcal{P}$, with an associated positive size for each path and each cycle, such that:*

B1. *For each arc $\hat{a}$ in G, the sum of the sizes of all paths and cycles that contain the arc $\hat{a}$ is equal to the size of arc $\hat{a}$ in G.*

B2. *All paths terminate at a surplus node.*

B3. *For each surplus node v, the sum of the sizes of all paths terminating at v is equal to the excess at v.*

Properties B1 and B2 follow immediately from the flow decomposition theorem, while Property B3 follows from the definition of a surplus node.

We define the cost of a path or cycle P to be $size(P) \cdot \sum_{\hat{a} \in P} cost(\hat{a})$. (We note that since the cost of an arc may be negative, the cost of a path or cycle may also be negative.) We define the cost of a set of paths (resp., cycles) to be the sum of the costs of the paths (resp., cycles) in the set.

Lemma 2.8.6 *Let $\mathcal{C}$ and $\mathcal{P}$ be the set of cycles and paths, respectively, obtained in the decomposition of G. Then $cost(\mathcal{C}) + cost(\mathcal{P}) = C_s(S) - C_s(S^*)$.*

Proof: Consider any arc $\hat{a}$ with label α in G. From property B1 of Lemma 2.8.5, it follows that the total contribution from arc $\hat{a}$ to the cost of $\mathcal{C}$ and $\mathcal{P}$ equals $size(\hat{a})cost(\hat{a})$, which is the difference between the shipping cost of the chunk α in assignment σ and the shipping cost of the same chunk in assignment σ^*. Thus, by summing $size(\hat{a})cost(\hat{a})$ over all arcs, and thus accounting for all chunks, we obtain that $cost(\mathcal{C}) + cost(\mathcal{P}) = C_s(S) - C_s(S^*)$. ■

By the hypothesis of Lemma 2.8.1, it follows that $C_s(S)$ is greater than $C_s(S^*)$, and hence that the set $S^* - S$ is nonempty. We partition the set of paths $\mathcal{P}$ on the basis of the terminating nodes of the paths, all of which are in S^*. Let ℓ denote $|S^* - S|$ and let $u_1, \ldots, u_\ell$ be the facilities in $S^* - S$. Let $\mathcal{P}_i$, for $1 \leq i \leq \ell$, denote the set of paths in $\mathcal{P}$ with terminating node u_i. Let $\mathcal{P}_0$ denote the set of paths in $\mathcal{P}$ that terminate at a node in $S \cap S^*$. We will prove Lemma 2.8.1 by showing that there exists an i, $1 \leq i \leq \ell$, such that the cost of the solution $S + u_i$ satisfies the upper bound stated in the lemma. We determine u_i, along with an associated assignment, by modifying the original assignment σ through a sequence of "augmentation" operations.

Given a difference graph $G(\sigma', \sigma^*)$, an *augmentation* along a path or cycle P in $\mathcal{C} \cup \mathcal{P}$ yields a new assignment σ'' and a new difference graph $G(\sigma'', \sigma^*)$ that are defined as follows. We initially set σ'' to σ'. Then, for each arc (u, v) on P with label α, we decrease $\sigma''(u, j)$ by $\mathit{size}(P)$ and increase $\sigma''(v, j)$ by $\mathit{size}(P)$, where j is $\mathit{cl}(\alpha)$, the client associated with chunk α. That is, the chunk α which is shipped from u in σ' is now shipped from v in σ''. Note that in σ^*, the chunk α is shipped from v. Therefore, the new difference graph $G(\sigma'', \sigma^*)$ is obtained by making the following simple modification to $G(\sigma', \sigma^*)$: Replace every arc (u, v) in P with label α by an arc (v, v) with label α. The cost of this new arc is 0; thus, the shipping cost of the assignment σ'' is $\mathit{cost}(P)$ less than the shipping cost of assignment σ'.

The validity of σ'' follows from the fact that any augmentation does not increase the total size of the arcs coming into any node. Therefore, by Lemma 2.8.4, for each node v, the total demand shipped from v under assignment σ'' is at most the total demand shipped from v under assignment σ', which is at most M. Hence, σ'' is a valid assignment with shipping cost equal to $C_s(S) - \mathit{cost}(P)$. If we now consider a sequence of augmentations, the following lemma is immediate.

Lemma 2.8.7 (Augmentations) *Let σ' be an assignment obtained from σ by a sequence of augmentations along a subset $\mathcal{Q}$ of distinct paths and cycles drawn from $\mathcal{C} \cup \mathcal{P}$. Then, σ' is a valid assignment and has shipping cost $C_s(S) - \mathit{cost}(\mathcal{Q})$.*

We now apply Lemma 2.8.7 to the paths in $\mathcal{P}_i$ and the paths and cycles in $\mathcal{C} \cup \mathcal{P}_0$ to obtain Lemmas 2.8.8 and 2.8.9, respectively.

Lemma 2.8.8 *For each u_i in $S^* - S$, there exists an assignment for $S + u_i$ with a shipping cost of $C_s(S) - cost(\mathcal{P}_i)$.*

Proof: The validity and the shipping cost of the new assignment follows directly from Lemma 2.8.7. Moreover, since σ uses only the facilities in S, and since all paths of $\mathcal{P}_i$ terminate at u_i, the final assignment uses only the facilities in $S + u_i$. This completes the proof. ■

Lemma 2.8.9 *Each path or cycle in $\mathcal{C} \cup \mathcal{P}_0$ has a nonpositive cost.*

Proof: First, note that each node of any path or cycle in $\mathcal{C} \cup \mathcal{P}_0$ belongs to S. Now let P be a path or a cycle in $\mathcal{C} \cup \mathcal{P}_0$ with positive cost, if one exists. By starting with σ and augmenting along P, we obtain a new assignment for S with a shipping cost that is strictly less than the shipping cost of the current assignment σ. This contradicts the optimality of σ. ■

We are now ready to prove Lemma 2.8.1.

Proof of Lemma 2.8.1: From Lemma 2.8.6 and Lemma 2.8.9, it follows that $\sum_{i=1}^{\ell} cost(\mathcal{P}_i) \geq C_s(S) - C_s(S^*)$. By averaging, there exists an i such that $cost(\mathcal{P}_i) \geq \frac{C_s(S)-C_s(S^*)}{\ell}$. Applying Lemma 2.8.8 with this value of i, we obtain $C_s(S + u_i) = C_s(S) - cost(\mathcal{P}_i)$. Combining these bounds, we have

$$\begin{aligned} C(S) - C(S + u_i) &= C_s(S) - C_s(S + u_i) \\ &= cost(\mathcal{P}_i) \\ &\geq \frac{C_s(S) - C_s(S^*)}{\ell} \\ &\geq \frac{C_s(S) - C_s(S^*)}{k}, \end{aligned}$$

since $\ell \leq k$. This completes the proof of the lemma. ■

2.8.2 Dropping a facility

The simple approach used in the uncapacitated case (Lemma 2.5.2) does not suffice for the proof of Lemma 2.8.2. The main reason is that the operation of dropping a facility

i and assigning all of its clients to another facility i' may violate the capacity constraint at i'. To get around this hurdle, we use the notion of "heavy" and "light" facilities. A facility i in S is called *heavy* if $D_i(S,\sigma)$ is greater than $M/2$, and *light* otherwise. The motivation behind this definition is that dropping a light facility i and reassigning the demand $D_i(S,\sigma)$ to another light facility does not violate the capacity constraints.

Let S_L denote the set of these light facilities in S. Our strategy will be to focus on S_L and proceed essentially as in the uncapacitated case. (Note that the number of light facilities is at least $(1+\alpha)k$. This is because the total demand of all clients cannot exceed kM and hence the number of heavy facilities can be at most $2k$.) The basic idea remains the same as before: classify the facilities of S_L as "primary" and "secondary" and then show that a secondary facility of S_L can be dropped without incurring a "large" increase in the shipping cost. However, some differences do arise in the analysis due to the fact that a client's demand may be split across several facilities.

Let $\hat{\sigma} : N \mapsto S^*$ be a function that maps each location i to the closest facility in S^*, breaking ties arbitrarily. (If there were no capacity constraints, then σ^* would have been exactly the same as this function.) Now using $\hat{\sigma}$, we classify the facilities in S_L as "primary" or "secondary", as follows: For each facility i^* in S^*, consider those facilities of S_L that are assigned to i^* by $\hat{\sigma}$ and pick the closest one among these, breaking ties arbitrarily. The closest facility thus chosen is said to be the *primary* facility of i^*. Let S_{LP} denote the set of these primary facilities. (Note that all primary facilities are light.) Facilities of S_L that are not primary are said to be *secondary.* For each secondary facility i in S_L, we define the associated primary facility as the primary facility of $\hat{\sigma}(i)$. Note that the number of secondary facilities is given by $|S_L| - |S_{LP}|$, and is at least αk. We will show that one of these secondary facilities can be dropped without incurring a "large" increase in the shipping cost.

Claim 2.8.1 *Consider a secondary facility i in S_L, and let i' be its associated primary facility. Then for all j in N, we have $c_{ji'} - c_{ji} \leq 2(c_{ji} + c_{j\hat{\sigma}(j)})$.*

Proof: Let i^* denote $\hat{\sigma}(i)$, the facility in S^* that is closest to i. (Note that $\hat{\sigma}(i')$ is also i^*.) Now using the same reasoning as in the proof of Claim 2.5.1, we obtain:

$$c_{ji'} \leq c_{ji} + c_{ii^*} + c_{i^*i'} \leq c_{ji} + 2c_{ii^*} \leq c_{ji} + 2c_{i\hat{\sigma}(j)} \leq c_{ji} + 2c_{ij} + 2c_{j\hat{\sigma}(j)}.$$

■

Let Δ_i denote the increase incurred in the shipping cost when a secondary facility i is dropped and the demand $D_i(S,\sigma)$ is reassigned to the primary facility associated with i. (Note that this reassignment does not violate any capacity constraints since both these facilities are light.)

Claim 2.8.2 *For any secondary facility i in S_L, we have $\Delta_i \leq \sum_{j\in N} 2\sigma(j,i)(c_{ji}+c_{j\hat{\sigma}(j)})$.*

Proof: Let i' denote the primary facility associated with i. The reassignment of demand $D_i(S,\sigma)$ to i' causes $\sigma(j,i)$ units of demand that was originally shipped from i to a client j to now be shipped from i' to j. Hence the increase Δ_i in shipping cost is equal to $\sum_{j\in N}\sigma(j,i)(c_{ji'}-c_{ji})$. Applying the upper bound of Claim 2.8.1, the result follows. ■

Proof of Lemma 2.8.2: It suffices to show that there exists a secondary facility i for which $\Delta_i \leq 2\cdot\frac{C_s(S)+C_s(S^*)}{\alpha k}$. Recall that $S_L - S_{LP}$ is the set of secondary facilities, and that $|S_L - S_{LP}| \geq \alpha k$. We have

$$\begin{aligned}
\sum_{i\in S_L-S_{LP}} \Delta_i &\leq 2\cdot\sum_{i\in S_L-S_{LP}}\sum_{j\in N}\sigma(j,i)(c_{ji}+c_{j\hat{\sigma}(j)}) \\
&\leq 2\cdot\sum_{i\in S}\sum_{j\in N}\sigma(j,i)(c_{ji}+c_{j\hat{\sigma}(j)}) \\
&= 2\cdot C_s(S) + 2\cdot\sum_{j\in N}\sum_{i\in S}\sigma(j,i)c_{j\hat{\sigma}(j)} \\
&= 2C_s(S) + 2\cdot\sum_{j\in N} d_j c_{j\hat{\sigma}(j)} \\
&\leq 2C_s(S) + 2C_s(S^*).
\end{aligned}$$

(The first step uses Claim 2.8.2. The fourth step uses the fact that $\sum_{i\in S}\sigma(j,i) = d_j$, the demand at j. The last step uses the fact that $\hat{\sigma}(j)$ is the closest facility to j among all facilities in S^*.) By a simple averaging argument, it follows that the required secondary facility exists. ■

2.9 Capacitated facility location with splittable demands

In this section we consider the capacitated facility location problem with splittable demands (CFLS) and show in Theorem 2.9.1 that, as in the uncapacitated case, each step of the local search heuristic yields a significant improvement in cost whenever the current cost is sufficiently higher than the optimal cost. Throughout this section, we assume that $p(n)$ is a sufficiently large polynomial in n.

Theorem 2.9.1 *Let S be any subset of F such that $C(S) > (8+\varepsilon)C(S^*)$. Then there exists $T \subseteq F$ such that $|S-T| \leq 1$, $|T-S| \leq 1$, and $C(S) - C(T) \geq \frac{C(S)}{p(n)}$.*

Corollary 2.9.1.1 *For any constant $\varepsilon > 0$, the local search heuristic yields a $(8+\varepsilon)$-approximate solution in polynomial time.* ■

We prove the theorem in two stages, as in the uncapacitated case. First, we show in Lemma 2.9.1 that if the shipping cost of S is sufficiently large, then we can add a facility to S and obtain a significant improvement in the cost. Due to the capacity constraints, the proof of this lemma is more involved than in the uncapacitated case (Lemma 2.6.1). Section 2.9.1 gives the proof of this lemma using much of the technical machinery developed in Section 2.8.1, where the operation of adding a facility for CKMS is analyzed. Second, we show in Lemma 2.9.2 that if the facility cost of S is sufficiently large then we can either drop a facility from S or substitute a facility in S to obtain a significant improvement in the cost. The proof of this lemma is given in Section 2.9.2.

Lemma 2.9.1 (Adding a facility) *Let S be any subset of F such that $C_s(S) \geq C(S^*) + \frac{nC(S)}{p(n)}$. Then there exists a v in F such that $C(S) - C(S+v) \geq \frac{C(S)}{p(n)}$.*

Lemma 2.9.2 (Dropping or swapping facilities) *Let S be any subset of F. If no facility of S can be dropped or substituted (by a facility not in S) to improve the cost by at least $\frac{C(S)}{p(n)}$, then the following upper bound on the facility cost of S holds:*

$$C_f(S)\left(1 - \frac{n^2}{p(n)}\right) < 5C(S^*) + 2C_s(S) + \frac{C(S)}{n}.$$

Theorem 2.9.1 follows directly from Lemmas 2.9.1 and 2.9.2.

Proof of Theorem 2.9.1: This is similar to the proof of Theorem 2.6.1, except that we use Lemmas 2.9.1 and 2.9.2 instead of Lemmas 2.6.1 and 2.6.2. In this case, we obtain

$$C(S)\left(1-\frac{n^2}{p(n)}-\frac{3n}{p(n)}-\frac{1}{n}\right) < 8C(S^*),$$

which is a contradiction if $p(n)$ is chosen sufficiently large. ■

2.9.1 Adding a facility

In this section, we prove Lemma 2.8.1. By the hypothesis of the lemma it follows that $C_s(S)$ is greater than $C_s(S^*)$, and hence that the set $S^* - S$ is nonempty. Let ℓ denote the size of $S^* - S$ and let $u_1, \ldots, u_\ell$ be the facilities in $S^* - S$. We will now show that there exists a facility u_i in $S^* - S$ that satisfies the requirement of the lemma.

We use the technical machinery developed in Section 2.8.1 for CKMS. As in that case, we construct a difference graph based on the sets S and S^* and the assignments σ and σ^*. Since Lemmas 2.8.3 through 2.8.9 only concern the difference graph and the shipping costs associated with the solutions S and S^*, all of these claims also hold for CFLS. Lemma 2.9.1 now follows from Lemmas 2.8.6, 2.8.8, and 2.8.9.

Proof of Lemma 2.9.1: We are given that $C_s(S) - C(S^*) \geq nC(S)/p(n)$. From Lemmas 2.8.6 and 2.8.9, it follows that $\sum_{i=1}^{\ell} cost(\mathcal{P}_i) \geq C_s(S) - C_s(S^*)$. Subtracting $C_f(S^*)$ from both sides, we get

$$\sum_{i=1}^{\ell}(cost(\mathcal{P}_i) - f_{u_i}) \geq C_s(S) - C_s(S^*) - C_f(S^*).$$

By averaging, there exists an i such that $cost(\mathcal{P}_i) - f_{u_i} \geq \frac{C_s(S)-C_s(S^*)-C_f(S^*)}{\ell}$. Applying Lemma 2.8.8 with this value of i, we obtain $C_s(S+u_i) = C_s(S) - cost(\mathcal{P}_i)$. Combining these bounds, we have

$$\begin{aligned}
C(S) - C(S+u_i) &= C_s(S) - C_s(S+u_i) - f_{u_i} \\
&= cost(\mathcal{P}_i) - f_{u_i} \\
&\geq \frac{C_s(S) - C_s(S^*) - C_f(S^*)}{\ell} \\
&\geq \frac{C_s(S) - C(S^*)}{n} \\
&\geq \frac{C(S)}{p(n)},
\end{aligned}$$

since $C_s(S) - C(S^*) \geq nC(S)/p(n)$ by assumption. This completes the proof of the lemma. ■

2.9.2 Dropping or swapping facilities

It remains to prove Lemma 2.9.2. Due to the capacity constraint, the simple approach used in the uncapacitated case (Lemma 2.6.2) does not suffice here. In particular, some of the candidate operations may be invalid because reassigning the demand associated with a facility i to another facility j may violate the capacity constraint at j. We overcome this difficulty by studying a problem that we call the auxiliary allocation problem, the solution of which leads to a construction of suitable candidate operations. More specifically, the solution of the auxiliary allocation problem yields a valid reassignment of the demand associated with a facility that is being dropped or substituted. The auxiliary allocation problem is defined in Section 2.9.2.1. Section 2.9.2.2 contains the design and the analysis of the candidate operations. Finally, Section 2.9.2.3 completes the proof of Lemma 2.9.2.

2.9.2.1 The auxiliary allocation problem

Given any two solutions S and S^*, an assignment σ for S, and the original distance metric c, we define the *auxiliary allocation problem* R in order to capture the task of reassigning demand when a facility from S is dropped or swapped. The problem R is specified as follows:

- The nodes of $S - S^*$ are the *requestors.*
- The nodes of S^* are the *servers.*
- Each requestor node v in $S - S^*$ has a demand of $D_v(S, \sigma)$.
- Each server node v in $S \cap S^*$ has a capacity of $M - D_v(S, \sigma)$.
- Each server node v in $S^* - S$ has a capacity of M.
- The distance function between the nodes is given by c.

(For ease of notation, we drop S and σ and use D_v for $D_v(S, \sigma)$ throughout this section.) We seek an *allocation* $\beta : (S - S^*) \times S^* \mapsto \mathbf{R}$, that satisfies the demands at all of the requestor nodes without violating the capacity constraints at the server nodes. This allocation β will be used to guide the reassignment of the demand D_v when a facility v in $S - S^*$ is being considered for dropping or substituting. For a requestor node v in $S - S^*$, let $q_\beta(v)$ denote the cost of satisfying the demand at v; that is, $q_\beta(v)$ equals $\sum_{j \in S^*} \beta(v, j) c_{vj}$. Let q_β denote the overall cost of the allocation β, defined as $\sum_{v \in S - S^*} q_\beta(v)$. Figure 2.2 illustrates an allocation.

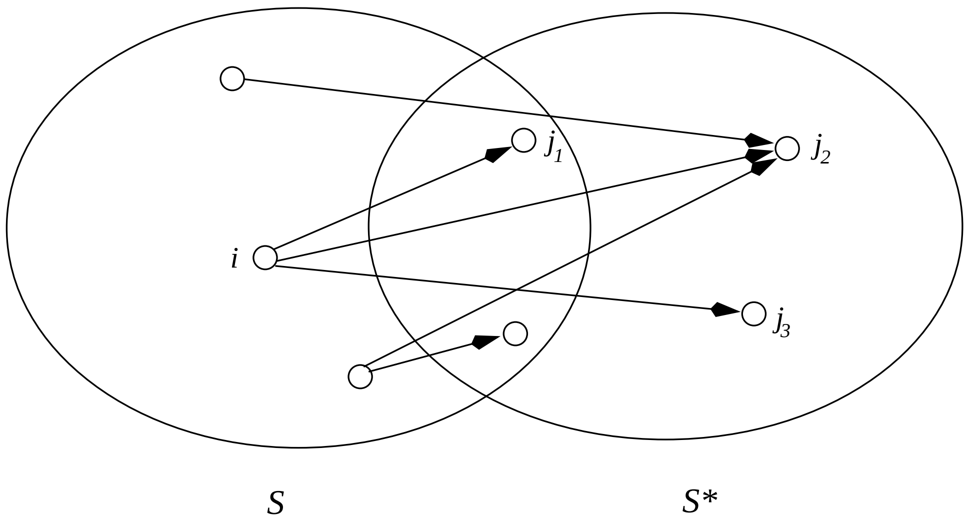


Figure 2.2: An allocation β. Only those edges (i, j) are shown for which $\beta(i, j) > 0$. Thus, for example, β reassigns the demand of requestor i to servers j_1, j_2, and j_3. Note that every edge is incident from a node in $S - S^*$ to a node in S^*.

We note that an allocation is conceptually the same as a splittable assignment. An allocation and an assignment differ only in their domains. (While the domain of an assignment is the set of all clients, the domain of an allocation is a subset of the facilities.) Indeed, our analysis of the auxiliary allocation problem uses the concept of a difference graph and much of the other technical machinery developed in Section 2.8 for the analysis of splittable assignments. We use the techniques of Section 2.8 to prove the following lemma, which states that there is an allocation that is not too expensive. The proof is deferred to Appendix 2.11.

Lemma 2.9.3 *There exists an allocation β for the auxiliary allocation problem R with cost q_β at most $C_s(S) + C_s(S^*)$.*

For any requestor node v, we define the servers j in S^* for which $\beta(v, j)$ is nonzero as the *β-suppliers* of v. If every requestor node v in $S - S^*$ has at most one *β-supplier* in $S^* - S$, then β is said to be *refined*. Note that a refined allocation β allows requestor nodes to have multiple β-suppliers in $S \cap S^*$. Figure 2.3 illustrates a refined allocation. Note that unlike Figure 2.2, there is only one incoming arc into any node in $S^* - S$.

For the purpose of constructing the candidate operations, a refined allocation is more useful than an unrefined allocation. To see this, recall that when a facility v in $S - S^*$ is being dropped or substituted, our plan is to reassign the demand D_v among the β-suppliers of v. Since we are allowed to add only one new facility to $S - \{v\}$, it is essential that at most one β-supplier of v be outside S. This condition is satisfied only if β is refined.

The lemma below guarantees the existence of an "inexpensive" refined allocation $\hat{\beta}$ provided that we relax the auxiliary allocation problem by increasing the capacity of each server node i^* in $S^* - S$ from M to $2M$. We use $\hat{R}$ to denote this relaxed auxiliary allocation problem. (We note that this blowup in the capacities is for the purposes of the analysis only.)

Lemma 2.9.4 *There exists a refined auxiliary allocation $\hat{\beta}$ for the relaxed auxiliary allocation problem, $\hat{R}$, with cost $q_{\hat{\beta}}$ at most $C_s(S) + C_s(S^*)$.*

Proof: The desired claim follows from Lemma 2.9.3 and a result of Shmoys and Tardos [52] on the generalized assignment problem. The detailed proof is deferred to Appendix 2.11. ■

Let D'_v (resp., D''_v) denote the amount of demand shipped to v from servers in $S \cap S^*$ (resp., $S^* - S$) under the refined allocation $\hat{\beta}$. If D''_v is nonzero, then there is a unique $\hat{\beta}$-supplier in $S^* - S$ that we refer to as the *$\hat{\beta}$-foreign-supplier* of v. For the sake of uniformity, we extend this notion even to those requestor nodes v for which D''_v is zero. Though such requestor nodes do not have any $\hat{\beta}$-suppliers in $S^* - S$, we pick an arbitrary facility from $S^* - S$ and call it the $\hat{\beta}$-foreign-supplier of v.

Table 2.3 summarizes the notation developed in this section for the auxiliary allocation problem.

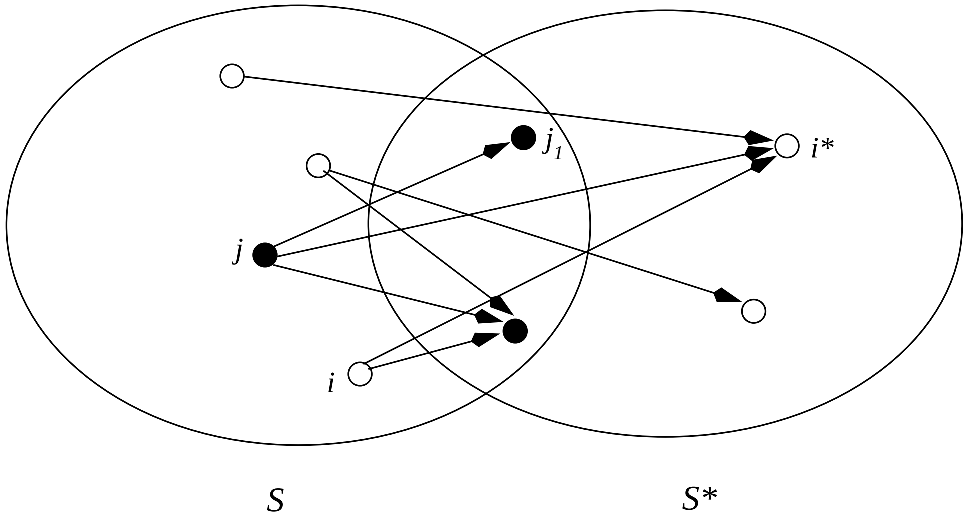


Figure 2.3: A refined allocation $\hat{\beta}$. Note that for each node u in $S - S^*$, there is at most one node in $S^* - S$ to which the demand of u is reassigned. The figure also illustrates a candidate operation for a facility i in S_{ELS}. Facility i^* is the $\hat{\beta}$-foreign-supplier of i, and j is the primary facility of i^*. The candidate operation for i, which is defined in Section 2.9.2.2, reassigns the demand of i among j and the $\hat{\beta}$-suppliers of j in $S \cap S^*$; that is, the demand of i is reassigned among the three darkly shaded nodes.

2.9.2.2 Candidate operations

Let S^* denote the optimal solution and S denote a current solution that satisfies the hypothesis of Lemma 2.9.2. In this section, we define and analyze a set of candidate operations for the facilities in S; our analysis leads to the proof of Lemma 2.9.2 in Section 2.9.2.3.

We partition S into five classes and define a candidate operation for the facilities of each class. As in the uncapacitated case, a facility i in S is called *cheap* if $f_i \leq C(S)/n^2$, and *expensive* otherwise. Moreover, a facility i in $S - S^*$ is called *light* if $D''_i \leq M/2$, and *heavy* otherwise. Let S_C denote the set of cheap facilities in $S - S^*$. Let S_{EL} (resp., S_{EH}) denote the set of expensive facilities in $S - S^* - S_C$ that are light (resp., heavy).

We now partition the set S_{EL} of expensive light facilities into *primary* and *secondary* facilities. Let i^* be any facility in $S^* - S$. If there is at least one facility in S_{EL} for which i^* is the $\hat{\beta}$-foreign-supplier, then we designate one such facility in S_{EL}, appropriately chosen, as the primary facility for i^*. The precise criterion for selecting the primary facility is deferred to later in this section. The facilities in S_{EL} that are not the primary

Notation	*What it denotes*
β	allocation: a function from $S - S^* \times S^*$ to $\mathbf{R}$ that indicates the demand that is satisfied at each requestor through each server
β-supplier of i	any facility j such that $\beta(i,j) > 0$
q_β	cost of allocation β
$\hat{\beta}$	refined allocation: an allocation in which for each $i \in S - S^*$ there is at most one facility j in $S^* - S$ such that $\hat{\beta}(i,j) > 0$
D_v	demand of requestor v in the allocation problem
D'_v	demand shipped to v from servers in $S \cap S^*$ under $\hat{\beta}$
D''_v	demand shipped to v from servers in $S^* - S$ under $\hat{\beta}$
$\hat{\beta}$-foreign-supplier of i	if $D''_i > 0$, then the unique facility $j \in S^* - S$ that satisfies $\hat{\beta}(i,j) > 0$; otherwise, an arbitrary facility in $S^* - S$

Table 2.3: Notation used for the auxiliary allocation problem

facility of any facility in $S^* - S$ are referred to as secondary facilities. We let S_{ELP} (resp., S_{ELS}) denote the set of expensive light facilities that are classified as primary (resp., secondary). Thus, $S \cap S^*$, S_C, S_{EH}, S_{ELP}, and S_{ELS} form a partition of S.

We now define *candidate operations* for facilities in each of these subsets, as follows:

1. For i in $S \cap S^*$, do nothing.

2. For i in S_C, do nothing.

3. For i in S_{EH}, substitute i with i^*, the $\hat{\beta}$-foreign-supplier of i. The demand D_i is reassigned among the $\hat{\beta}$-suppliers of i.

4. For i in S_{ELP}, substitute i with i^*, the $\hat{\beta}$-foreign-supplier of i. The demand D_i is reassigned among the $\hat{\beta}$-suppliers of i.

5. For i in S_{ELS}, drop i and let j be the primary facility associated with the $\hat{\beta}$-foreign-supplier of i. The demand D_i is reassigned as follows. First, D'_i is reassigned among the $\hat{\beta}$-suppliers of i in $S \cap S^*$. Second, D''_i is reassigned among j and the $\hat{\beta}$-suppliers of j in $S \cap S^*$; more specifically, a demand of up to $M - D_j$ is reassigned to j, and the remaining demand $(D''_i - M + D_j)$, if any, is reassigned among the

$\hat{\beta}$-suppliers of j in $S \cap S^*$. The set of nodes involved in the reassignments of demand D_i is illustrated in Figure 2.3.

We first need to ensure that each candidate operation is valid, i.e., the capacity constraints are not violated when an individual candidate operation is performed. (We note that each candidate operation is performed on the solution (S, σ).) The candidate operations for facilities in $S \cap S^*$ and S_C are trivially valid. The validity of a candidate operation for a facility i in $S_{EH} \cup S_{ELP}$ follows from the following two observations: (i) the reassignment respects the capacity constraint at each facility j in $S \cap S^*$ since the capacity of j in the relaxed auxiliary allocation problem is equal to the unused capacity of j in σ, and (ii) the demand reassigned to the $\hat{\beta}$-foreign-supplier of i is at most D_i, which is at most M. Finally, we consider a candidate operation for a facility i in S_{ELS}. The reassignment of D_i' for i in S_{ELS} respects the capacity constraints since it is performed only among facilities in $S \cap S^*$, and for each facility in $S \cap S^*$, its capacity in the auxiliary allocation problem equals its unused capacity in σ. But for the reassignment of D_i'', we need to ensure that there is sufficient remaining capacity at j and the $\hat{\beta}$-suppliers of j in $S \cap S^*$. This is ensured by the particular criterion by which we partition S_{EL} into primary and secondary facilities, which we now describe.

Let $Q(i^*)$ denote the set of requestor nodes for which i^* is the $\hat{\beta}$-foreign-supplier. Note that the sets $Q(i^*)$, for i^* in $S^* - S$, induce a partition of $S - S^*$. For each i^* in $S^* - S$ such that $Q(i^*) \cap S_{EL}$ is nonempty, the primary facility associated with i^* is defined as the facility j in $Q(i^*) \cap S_{EL}$ that minimizes $c_{ji^*} + \theta_j$, breaking ties arbitrarily. The parameter θ_j is defined as the average cost per unit demand for servicing an additional demand of $M/2$ units at j, using the remaining capacity (in the current solution (S, σ)) at facility j and at the $\hat{\beta}$-suppliers of j in $S \cap S^*$. (Remark: If there were no capacity constraints, then θ_j would be zero and the criterion for picking the primary facility would degenerate to that used in the uncapacitated case.) As mentioned earlier, a facility i in S_{EL} is referred to as a secondary facility if i is not the primary facility for any i^* in $S^* - S$.

For the parameter θ_j to be well-defined, we need to ensure that the total remaining capacity at facility j and at the $\hat{\beta}$-suppliers of j in $S \cap S^*$ is at least $M/2$. (Note that ensuring that the total remaining capacity at facility j and at the $\hat{\beta}$-suppliers of j in $S \cap S^*$ is at least $M/2$ also guarantees the validity of the candidate operation for any facility in S_{ELS}.) To see that this condition is satisfied, we first observe that the remaining capacity at j is $M - D_j$. Then, we observe that the remaining capacity at the $\hat{\beta}$-suppliers of j in $S \cap S^*$ is at least D'_j, since D'_j is the amount of demand shipped from $\hat{\beta}$-suppliers of j in $S \cap S^*$ to the requestor node j in the auxiliary allocation problem. Adding these two bounds, and using the fact that $D_j - D'_j = D''_j \leq M/2$ (since j is light), we find that the remaining capacity is at least $M/2$. Hence θ_j is well-defined.

We now analyze the three non-trivial candidate operations, namely, the ones associated with sets S_{EH}, S_{ELP}, and S_{ELS}. We call a candidate operation *good* if it reduces the cost of the solution by at least $\frac{C(S)}{p(n)}$, and *bad* otherwise. By the hypothesis of the Lemma 2.9.2, it follows that all of the candidate operations are bad. Claims 2.9.1, 2.9.2, and 2.9.3 show that if the candidate operation for a facility is bad, then its facility cost is "small". The proof of Lemma 2.9.2 then follows by combining these claims.

Claim 2.9.1 *If the candidate operation for every facility in S_{EH} is bad, then*

$$C_f(S_{EH})\left(1 - \frac{n^2}{p(n)}\right) \leq 4C_f(S^* - S) + \sum_{i \in S_{EH}} q_{\hat{\beta}}(i).$$

Proof: Consider an expensive heavy facility i in S_{EH}. The decrease in the facility cost caused by its candidate operation is $f_i - f_{i^*}$, and by the triangle inequality, the increase in the shipping cost is at most $q_{\hat{\beta}}(i)$. So the net decrease in cost is at least $f_i - (q_{\hat{\beta}}(i) + f_{i^*})$. If this quantity is at least $\frac{n^2 f_i}{p(n)}$, then since i is expensive, this candidate operation is good. Therefore, if this candidate operation is bad, the following upper bound on the facility cost of i holds: $f_i(1 - \frac{n^2}{p(n)}) < f_{i^*} + q_{\hat{\beta}}(i)$.

Since any facility i^* in $S^* - S$ can supply a demand of at most $2M$ under the refined auxiliary allocation $\hat{\beta}$ and each heavy facility receives at least $M/2$ units from its unique supplier, it follows that i^* can be the candidate substitute for at most four heavy facilities. Hence, summing over all of the expensive heavy facilities yields the required upper bound on $C_f(S_{EH})$. ■

Claim 2.9.2 *If the candidate operation for every facility in S_{ELP} is bad, then*

$$C_f(S_{ELP})\left(1-\frac{n^2}{p(n)}\right) \leq C_f(S^*-S) + \sum_{i\in S_{ELP}} q_{\hat{\beta}}(i).$$

Proof: The net cost decrease caused by the substitution of an expensive light primary facility i in S_{ELP} is $f_i - (q_{\hat{\beta}}(i) + f_{i^*})$. If this substitution is bad then, by proceeding as in the previous claim, we obtain the following upper bound on the facility cost of i: $f_i(1-\frac{n^2}{p(n)}) < f_{i^*} + q_{\hat{\beta}}(i)$.

Summing over all of the facilities of S_{ELP}, and using the fact that each i^* in $S^* - S$ has at most one associated primary facility, we obtain the required upper bound on $C_f(S_{ELP})$. ■

Claim 2.9.3 *If the candidate operation for every facility in S_{ELS} is bad, then*

$$C_f(S_{ELS})\left(1-\frac{n^2}{p(n)}\right) \leq \sum_{i\in S_{ELS}} 2q_{\hat{\beta}}(i).$$

Proof: The heart of our proof is to establish the following claim: The net cost decrease caused by the candidate operation for an expensive light secondary facility i in S_{ELS} is at least $f_i - 2q_{\hat{\beta}}(i)$. We note that the preceding claim implies that if the candidate operation for a facility i in S_{ELS} is bad, then $f_i(1-\frac{n^2}{p(n)}) \leq 2q_{\hat{\beta}}(i)$. Summing over all expensive light secondary facilities then yields the desired upper bound on $C_f(S_{ELS})$.

We now prove the desired lower bound on the decrease in cost caused by the candidate operation for an expensive light secondary facility i in S_{ELS}. Let i^* be the $\hat{\beta}$-foreign-supplier of i and let j be the primary facility associated with i^*. Let $q'_{\hat{\beta}}(i)$ denote the cost of shipping the demand D'_i from the $\hat{\beta}$-suppliers of i in $S \cap S^*$, and let $q''_{\hat{\beta}}(i)$ denote the cost of shipping the demand D''_i from i^*. Thus $q_{\hat{\beta}}(i) = q'_{\hat{\beta}}(i) + q''_{\hat{\beta}}(i)$.

When i is dropped, the demand D'_i is reassigned to the $\hat{\beta}$-suppliers of i in $S \cap S^*$. This increases the shipping cost by at most $q'_{\hat{\beta}}(i)$. It now suffices to bound the shipping cost increase due to the reassignment of D''_i by $q'_{\hat{\beta}}(i) + 2q''_{\hat{\beta}}(i)$.

We know that the candidate operation of i reassigns the demand D''_i to j and the $\hat{\beta}$-suppliers of j in $S \cap S^*$. By the definition of θ_j, the cost of servicing an additional demand of D''_i originating at j is at most $D''_i\theta_j$ (Recall that $D''_i \leq M/2$ since i is light).

Therefore, the increase in the shipping cost due to this reassignment of D''_i is at most

$$\begin{aligned} D''_i(c_{ij} + \theta_j) &\leq D''_i(c_{ii^*} + c_{ji^*} + \theta_j) \\ &\leq D''_i(c_{ii^*} + c_{ii^*} + \theta_i) \\ &\leq 2q''_{\hat{\beta}}(i) + \frac{M}{2} \cdot \theta_i \\ &\leq 2q''_{\hat{\beta}}(i) + q'_{\hat{\beta}}(i). \end{aligned}$$

(The second step uses the fact that j, and not i, was chosen as the primary for i^*; and hence $c_{ji^*} + \theta_j \leq c_{ii^*} + \theta_i$. The third step follows since $q''_{\hat{\beta}}(i) = D''_i c_{ii^*}$ and $D''_i \leq M/2$. The fourth step needs some elaboration. The goal here is to prove that $q'_{\hat{\beta}}(i)$ is an upper bound on $\frac{M}{2}\theta_i$, which is the cost of servicing an additional demand of $M/2$ units at i using the remaining capacities at i and the $\hat{\beta}$-suppliers of i. Note that the remaining capacity at i is $M - (D'_i + D''_i)$, which is at least $\frac{M}{2} - D'_i$ since $D''_i \leq M/2$. Therefore, $\frac{M}{2}\theta_i$ is at most the cost of satisfying D'_i units of demand at the $\hat{\beta}$-suppliers of i, which is at most $q'_{\hat{\beta}}(i)$.) This completes the proof of the claim. ■

2.9.2.3 Proof of Lemma 2.9.2

Since $C_f(S) = C_f(S \cap S^*) + C_f(S_C) + C_f(S_{EH}) + C_f(S_{ELP}) + C_f(S_{ELS})$ and $C_f(S_C) \leq C(S)/n$, by Claims 2.9.1, 2.9.2, and 2.9.3, we obtain

$$\begin{aligned} C_f(S)\left(1 - \frac{n^2}{p(n)}\right) &\leq C_f(S \cap S^*) + \frac{C(S)}{n} + 5C_f(S^* - S) + \sum_{i \in S - S^*} 2q_{\hat{\beta}}(i) \\ &\leq \frac{C(S)}{n} + 2q_{\hat{\beta}} + 5C_f(S^*) \\ &\leq \frac{C(S)}{n} + 2C_s(S) + 2C_s(S^*) + 5C_f(S^*) \\ &\leq \frac{C(S)}{n} + 2C_s(S) + 5C(S^*). \end{aligned}$$

(The second step follows from the definition of $q_{\hat{\beta}}$; the third follows from Lemma 2.9.4.) This completes the proof of the lemma.

2.10 Capacitated problems with unsplittable demands

In this section we consider the capacitated problems with unsplittable demands, namely, CKMU and CFLU. We obtain constant-factor approximation algorithms for these problems by combining our results for CKMS and CFLS, respectively, with a result of Shmoys and Tardos on the generalized assignment problem. In [52], Shmoys and Tardos show how to obtain an unsplittable assignment from a splittable assignment. Their result is presented in the context of job scheduling and is based on a careful rounding of the given fractional solution (i.e., splittable assignment) to an integer solution (i.e., unsplittable assignment). Here we give an adaptation of their result to our context.

Theorem 2.10.1 (Generalized assignment problem [52]) *Let S be a set of open facilities and N be a set of clients, and let d_i denote the demand at client i. Suppose there is a splittable assignment from N to S with shipping cost C that assigns at most M units of demand to any facility. Then there exists a polynomial-time computable unsplittable assignment from N to S with shipping cost at most C that assigns at most $M + \max_{i \in N} d_i$ units of demand to any facility.*

We now show how to obtain constant-factor approximation algorithms for CKMU and CFLU. We first relax the unsplittability requirement, and use the local search heuristic to solve the relaxed problem. By Corollary 2.8.1.1 (resp., Corollary 2.9.1.1), it follows that the splittable solution (S, σ) thus obtained is within a constant factor of the optimal for the CKMS (resp., CFLS). We then use Theorem 2.10.1 to obtain an unsplittable assignment σ' for S with shipping cost at most that of the splittable assignment σ for S. However, Theorem 2.10.1 requires the capacities of all facilities to be increased by a factor of at most two. If we assume, as in [53], that a blowup in the capacity incurs a proportionate increase in the facility cost, then the facility cost for S in the unsplittable case is at most twice that in the splittable case. Therefore, the total cost of the unsplittable solution (S, σ') is at most twice the total cost of the splittable solution (S, σ). Thus, the cost of (S, σ') is within a constant factor of the optimal unsplittable solution. Theorems 2.10.2 and 2.10.3 below summarize our results for CKMU and CFLU, respectively.

Theorem 2.10.2 (CKMU) *For any constant $\varepsilon > 0$, there is a constant δ such that the following can be computed in polynomial time, assuming a factor two blowup in the capacity of each facility: (i) a solution S with $C(S) \leq (1+\varepsilon)C(S^*)$ and $|S| \leq (1+\delta)k$, and (ii) a solution S with $|S| \leq (5+\varepsilon)k$ and $C(S) \leq (1+\delta)C(S^*)$.* ■

Theorem 2.10.3 (CFLU) *For any constant $\varepsilon > 0$, the local search heuristic yields a $(16+\varepsilon)$-approximate solution in polynomial time, assuming a factor two blowup in the capacity of each facility.* ■

2.11 Appendix: Proof of Lemmas 2.9.3 and 2.9.4

In this section we prove Lemmas 2.9.3 and 2.9.4, which are related to the auxiliary allocation problem defined in Section 2.9.2.1. We first consider Lemma 2.9.3, which states that the allocation problem R defined for solutions S and S^* with arbitrary assignments σ and σ^*, respectively, has an allocation β with cost at most $C_s(S^*)+C_s(S)$. The allocation β is constructed from the assignments σ and σ^*. The proof uses some of the techniques that were developed in Section 2.8.1.

We start by constructing another difference graph $G'(\sigma, \sigma^*)$ (or, simply G', when the context is clear) that is identical to the difference graph $G(\sigma, \sigma^*)$ constructed in Section 2.8.1, except that the costs on the arcs are different: While the cost of an arc $\hat{a}$ is set to $c_{ji} - c_{ji^*}$ in $G(\sigma, \sigma^*)$, we set it to $c_{ji} + c_{ji^*}$ in $G'(\sigma, \sigma^*)$. The cost of the new difference graph G' is defined as before, namely, $\sum_{\hat{a}\in A} size(\hat{a}) cost(\hat{a})$.

Since G and G' are identical except for the arc costs, Lemmas 2.8.4 and 2.8.5 proved for G also hold for G'.

Lemma 2.11.1 *Same as Lemma 2.8.4, but with G replaced by G'.* ■

Lemma 2.11.2 (Decomposition of G') *Same as Lemma 2.8.5, but with G replaced by G'.* ■

Lemma 2.11.3 *If $\mathcal{P}$ is the set of paths obtained by the decomposition of G', as described in Lemma 2.11.2, then $cost(\mathcal{P})$ is at most $C_s(S) + C_s(S^*)$.*

Proof: Let $\mathcal{C}$ denote the set of cycles obtained by the decomposition of G' in Lemma 2.11.2. Then, $cost(\mathcal{P}) + cost(\mathcal{C})$ is the sum of the costs of all the arcs in G', which equals $C_s(S) + C_s(S^*)$. Since all of the arc costs in G' are positive, it follows that $cost(\mathcal{C}) \geq 0$ and hence the desired claim holds. ■

Proof of Lemma 2.9.3: Using $\mathcal{P}$, we construct the allocation β as follows: For each i in $S - S^*$ and j in $S \cup S^*$, let $\mathcal{P}_{ij}$ denote the set of paths in $\mathcal{P}$ that start at i and terminate at j. Then we set $\beta(i,j) = \sum_{P \in \mathcal{P}_{ij}} size(P)$. We need to ensure that β is a function from $(S - S^*) \times S^*$ to $\mathbf{R}$. For this it suffices to show that the terminating nodes of all paths P in $\mathcal{P}$ belong to S^*. By Lemma 2.11.1, all nodes with an incoming arc in G' belong to S^*, since all other nodes v have $D_v(S^*, \sigma^*) = 0$. Hence the nodes of S^* are the only nodes at which a path P in $\mathcal{P}$ can terminate.

Claim 2.11.1 below establishes that β is a valid allocation. Then Claim 2.11.2 establishes the required upper bound on the cost of β. ■

Claim 2.11.1 *The allocation β, constructed as above, is a valid solution to the allocation problem R.*

Proof: We first show that the demands of all requestor nodes are satisfied. Consider an arbitrary requestor node v in $S - S^*$. By Lemma 2.11.1, the total size of all arcs going out of v in G' is $D_v(S, \sigma)$. Moreover, since $D_v(S^*, \sigma^*)$ is zero, there are no arcs in G' coming into v. Hence the node v can only occur in the decomposition of G' as the starting node of a path in $\mathcal{P}$. It follows from property B1 of Lemma 2.11.2 that the sum of the sizes of all the paths in $\mathcal{P}$ starting at v is $D_v(S, \sigma)$. Therefore, by the construction of β, the total demand shipped to v is $D_v(S, \sigma)$.

We now show that β observes the capacity constraint at each server node. Consider an arbitrary server node v in S^*. By the construction of β, the total demand shipped from v under β (i.e., $\sum_{i \in S - S^*} \beta(i, v)$) is at most the sum of the sizes of all paths in $\mathcal{P}$ terminating at v. By properties B2 and B3 of Lemma 2.11.2, the sum of the sizes of all paths terminating at v is equal to the excess of v, which is $D_v(S^*, \sigma^*) - D_v(S, \sigma)$. Since $D_v(S^*, \sigma^*) \leq M$, it follows that the total demand shipped from v under β is at most the capacity of v in the allocation problem. Thus, all of the capacity constraints are observed. This establishes that β is a valid allocation. ■

Claim 2.11.2 *The cost of the allocation β is at most $C_s(S) + C_s(S^*)$.*

Proof: Consider any arc $\hat{a}$ in G'. Let the endpoints of $\hat{a}$ be u and u^*, and let the chunk corresponding to $\hat{a}$ be α, and let y be $cl(\alpha)$, that is, the client associated with chunk α. Recall that $cost(\hat{a})$ is $c_{uy} + c_{yu^*}$. By the triangle inequality, this is at least c_{uu^*}. Now consider any path P in $\mathcal{P}$ with starting and terminating nodes i and j. Then $\sum_{\hat{a} \in P} cost(\hat{a}) \geq c_{ij}$ by repeated applications of the triangle inequality. Thus, $cost(P)$ is at least $size(P)c_{ij}$. Summing over all paths P starting at i and terminating at j, we obtain $\beta(i,j)c_{ij} \leq \sum_{P \in \mathcal{P}_{ij}} cost(P)$. Then, summing over all i and j, it follows that the cost of allocation β is at most $cost(\mathcal{P})$. The claim then follows from Lemma 2.11.3. ■

We now turn to the proof of Lemma 2.9.4, which establishes the existence of a refined allocation for the relaxed allocation problem.

Proof of Lemma 2.9.4: Let β be the (unrefined) allocation provided by Lemma 2.9.3 for the allocation problem R. Using R and β, we first construct a restricted allocation problem R', which involves the servers in $S^* - S$ only. We then apply a result of Shmoys and Tardos (Theorem 2.10.1) to R' to obtain an unsplittable allocation for R', which is then combined with β to construct the required refined allocation $\hat{\beta}$ for R. (An allocation α is said to be unsplittable if each requestor has at most one supplier.)

Before proceeding, we need some notation to split the demands and shipping costs into two parts, one for servers in $S \cap S^*$, and the other for servers in $S^* - S$. Throughout this proof, for notational convenience, we use D_v to denote $D_v(S, \sigma)$. Let D'_v (resp., D''_v) denote the amount of demand shipped to a requestor node v from servers in $S \cap S^*$ (resp., $S^* - S$) under the allocation β. That is, D'_v equals $\sum_{j \in S \cap S^*} \beta(v,j)$ and D''_v equals $\sum_{j \in S^* - S} \beta(v,j)$. Note that $D'_v + D''_v = D_v$. The precise formulation of R' can now be given as follows:

- The nodes of $S - S^*$ are the requestors.
- The nodes of $S^* - S$ are the servers.
- Each requestor node v in $S - S^*$ has a demand of D''_v .
- Each server node v in $S^* - S$ has a capacity of M.
- The distance function between the nodes is given by c.

Let q'_β (resp., q''_β) denote the cost of shipping the demand from servers in $S \cap S^*$ (resp., $S^* - S$) under allocation β. More precisely, q'_β equals $\sum_{i \in S - S^*} \sum_{j \in S \cap S^*} \beta(i,j)c_{ij}$

and q''_β equals $\sum_{i \in S-S^*} \sum_{j \in S^*-S} \beta(i,j) c_{ij}$. Clearly, $q'_\beta + q''_\beta = q_\beta$. Consider the allocation β' obtained by restricting β to servers in $S^* - S$. Thus, for each i in $S - S^*$ and j in $S^* - S$, set $\beta'(i,j) = \beta(i,j)$. The allocation β' is clearly valid for R' and has cost q''_β. Theorem 2.10.1 then implies that there exists an unsplittable allocation β'' of cost at most q''_β that is feasible for R' provided that the capacity constraints on the servers are relaxed to $M + \max_{i \in S-S^*} D''_i$, which is at most $2M$. Since β'' is an unsplittable allocation, it ensures that each requestor node has at most one supplier in $S^* - S$. Using β'' and β, we construct a refined allocation $\hat{\beta}$ for $\hat{R}$ as follows:

- For each i in $S - S^*$ and j in $S^* - S$, set $\hat{\beta}(i,j) = \beta''(i,j)$.
- For each i in $S - S^*$ and j in $S^* \cap S$, set $\hat{\beta}(i,j) = \beta(i,j)$.

It is straightforward to verify that the above allocation $\hat{\beta}$ is a valid refined allocation for $\hat{R}$. Moreover, the cost of $\hat{\beta}$ is equal to the cost of β'' plus q'_β. Hence it is upper-bounded by q_β, which in turn is at most $C_s(S)+C_s(S^*)$, by Lemma 2.9.3. This completes the proof. ■

Chapter 3

The Hierarchical Placement Problem

3.1 Introduction

Cooperative caching [21] is an emerging paradigm in the design of scalable high performance distributed systems. In traditional caching schemes, the primary function of a cache is to act as a fast intermediate storage between a client or a collection of clients and the servers. In such schemes, each request is satisfied by the cache associated with the requesting node or by the server of the requested object. Moreover, the storage decisions made by one cache are independent of those made by other caches in the system. The defining characteristic of cooperative caching schemes, on the other hand, is that the caches cooperate in serving one another's requests as well as in storage decisions.

A number of recent studies have discussed the benefits of cooperative caching for distributed file systems and large-scale information systems such as digital libraries and the World Wide Web. These studies include analytical results (e.g., [5, 40]), simulation experiments (e.g., [10, 24, 28, 55]) and prototypes and products (e.g., Harvest [11, 13], xFS [2]). The widely deployed and studied Harvest cache system [11], employs a hierarchical arrangement of object caches to improve access performance. In the Squid cache system [57], which is a successor to Harvest, the caches cooperate via the Internet

This is joint work with Greg Plaxton and Rajmohan Rajaraman. A preliminary version of this work appeared in [38].

Cache Protocol [58] to serve one another's misses and thus reduce overall traffic. Recent experimental work of [55] also indicates the potential for significant performance gains by cooperative caching on the Internet. In the context of local-area networks, the xFS system [2] utilizes cooperative caches to obtain a serverless file system. While the appropriate level of cooperation depends on the kind and scale of the application, it is evident from these studies that cooperative caching will play a significant role in future information systems.

This dissertation studies an important component of cooperative caching schemes, which we refer to as placement. A cooperative caching scheme can be loosely divided into three components: placement, search, and consistency. The *placement* component determines where to place the copies of the objects. The *search* component directs each request to an appropriate copy of the requested object. Finally, the *consistency* component maintains the desired level of consistency among the various copies of an object.

In [5], Awerbuch, Bartal, and Fiat study a general on-line cooperative caching problem on arbitrary networks and present a polylog(n)-competitive algorithm, where n is the number of nodes in the network. Their result is impressive in that it addresses the search, placement, and consistency problems in a general adversarial setting. However, the time and space bounds established may exceed optimal by a polylogarithmic factor. In this dissertation, we study a special case of cooperative caching with the aim of developing simple algorithms that obtain near-optimal (e.g., constant-factor approximation) solutions. We focus our attention on the placement component of cooperative caching and develop placement algorithms for a class of networks that we refer to as *hierarchical networks*. Our definition of hierarchical networks, which is based on the ultrametric cost model used in [33], is motivated by the fact that modern wide-area networks tend to admit natural hierarchical decompositions. In fact, the existence of a hierarchical decomposition is implicit in several previous studies (e.g., see [13, 56, 57]).

3.1.1 The problem

We address the following placement problem for hierarchical networks. Let Ψ denote a set of unit-size read-only objects and let $size(u)$ denote the cache size at node u. We are given for each node u and each object ψ the access frequency for ψ at u. Further assume that for any node u and object ψ, the cost of satisfying an access request for ψ originating at u is given by the communication cost between u and v, where v is the closest node (with respect to the communication cost function) that holds a copy of ψ. The objective of a placement algorithm is to determine a placement of object copies in the node caches subject to space constraints such that the average access cost over all nodes and all objects is minimized. See Section 3.2 for a formal statement of the problem.

Our problem formulation is most suitable for applications in which writes are infrequent and changes in the access pattern over short time intervals are moderate. Infrequent writes imply a low overhead in maintaining consistency among the copies of an object, allowing us to separate the concerns of consistency and placement. Moderate changes in the access pattern can be addressed by invoking the placement algorithm at regular intervals. Finally, we consider our assumption that each request is satisfied by the nearest copy of the requested object. This assumption is justified by the existence of algorithms for the search component that direct each request to a nearby, if not the nearest, copy of the requested object [6, 24, 48, 56]. Another useful idea in this regard is the summary cache protocol of [24] in which each cache maintains a synopsis of the contents of nearby caches so that it can redirect a request to a nearby copy (if one exists) in the event of a cache miss.

3.1.2 Our results

We first present a polynomial-time exact algorithm for the hierarchical placement problem based on a reduction to minimum-cost flow. Our reduction, described in Section 3.3, generalizes the approach of Leff, Wolf, and Yu [40], who solved the problem for the special case of a single-level hierarchy. While the algorithm of Section 3.3 runs in polynomial time, the degree of the polynomial is sufficiently high to make the algorithm largely impractical.

Thus, we focus on the goal of developing a fast, simple constant-factor approximation algorithm for the hierarchical placement problem that admits an efficient distributed implementation. In Section 3.4.1 we consider a natural bottom-up greedy algorithm, but find that this algorithm has approximation ratio $\Omega(n)$. The lower bound proof leads us to a natural refinement of the greedy algorithm, the amortizing algorithm, described in Section 3.4.2. Like the greedy algorithm, the amortizing algorithm starts with a placement in which each the cache of each node holds the locally optimal set of objects. The algorithm then iteratively improves the placement in a bottom-up manner as the nodes cooperate and share information about the access frequencies across larger regions of the network.

Our main result, shown in Section 3.5, is that the amortizing algorithm achieves a constant-factor approximation for the hierarchical placement problem. (The constant factor is less than 14.) The proof of our main technical lemma appears in Section 3.6. In Section 3.7, we describe an efficient distributed implementation of the amortizing algorithm.

It is worth noting that the recent results on the approximation of general metrics by tree metrics [7, 8, 14] imply that any hierarchical placement algorithm can be used to obtain a placement algorithm for general metrics giving up an extra $O(\log n \log\log n)$ factor in the approximation.

3.1.3 Related work

Dowdy and Foster [23] initiated the study of cooperative caching in the context of allocating files in a distributed network [23]. A sequence of results [4, 9, 44] obtained improved algorithms for centralized as well as distributed file allocation. These results, however, did not consider cache capacities at the individual nodes. As mentioned earlier in this section, Awerbuch, Bartal, and Fiat [5] provide a polylog(n)-competitive on-line algorithm for the general placement problem under the assumption that the size of each cache in the on-line algorithm is polylog(n) times more than the size in the optimal algorithm. In contrast, we obtain an optimal centralized and a constant-factor approximate distributed algorithm for the off-line version of the problem on hierarchical networks without any blowup in the cache sizes.

In [40], Leff, Wolf, and Yu study the placement problem for a network of workstations, which they model as a single-level hierarchy. In addition to providing an optimal centralized algorithm for this case, they give heuristics for a distributed solution. These heuristics, however, make use of particular properties of a single-level hierarchy that are not applicable in an arbitrary hierarchical setting.

By adopting a communication model based on a fixed cost function, we endeavor to separate the concerns of caching (a higher-level operation) from routing (a lower-level operation). In contrast, some recent papers have incorporated routing issues into caching by either combining the two problems or making use of available routing information. For example, the algorithms developed in [29, 49, 60] tend to cache copies of an object in nodes that either lie on or are close to the path along which the object is being transferred. Routing information is also used in the placement algorithms developed in [45], where the primary aim is to minimize the network congestion that may occur when requests and their responses are routed within the network. We also remark that cost models have been adopted in uniprocessor caching systems to model scenarios in which the costs incurred in the retrieval of objects on cache misses may vary from one object to another [12, 31, 61].

With regard to uniprocessor caching schemes, recent research has addressed the challenge of designing cache replacement policies that take into account the differing costs incurred in the retrieval of objects on cache misses. This has led to studies formulating generalizations of the traditional uniprocessor caching problems that account for the differing costs [12, 31, 61].

In a recent experimental study [36], Korupolu and Dahlin evaluate the practical performance of several placement and replacement algorithms for cooperative caching. Their simulation experiments demonstrate that, in practice, both our greedy placement algorithm as well as our amortizing placement algorithm are in fact very close to the optimal.

We remark that the placement problem can be viewed as an instance of facility location with multiple types of facilities and constraints on the number of facilities that can located at a point. To the best of our knowledge, this multiple facilities location problem has not been studied previously. For a survey of results related to facility location, see [19, 53].

3.2 Preliminaries

In this section, we formally define the hierarchical placement problem. For the purpose of simplifying the exposition, we define this problem with respect to a fixed tuple $(\Psi, \mathcal{V}, \mathit{distance}, \mathit{frequency}, \mathit{size}, \mathit{penalty})$, where Ψ is a set of unit-size read-only *objects*, $\mathcal{V}$ is a set of *nodes*, $\mathit{distance} : \mathcal{V} \times \mathcal{V} \to \mathbf{R}$, $\mathit{frequency} : \mathcal{V} \times \Psi \to \mathbf{R}$, $\mathit{size} : \mathcal{V} \to \mathbf{N}$, and *penalty* is a real number. We assume that the set of nodes $\mathcal{V}$ forms a hierarchy as defined in Section 3.2.1. We assume that *penalty* is at least as large as $\mathit{diameter}(\mathcal{V})$, where for any set of nodes U, $\mathit{diameter}(U)$ is defined as the maximum value of $\mathit{distance}(u, v)$ over all nodes u and v in U. The hierarchical placement problem is defined in Section 3.2.2.

3.2.1 Hierarchies

We now inductively define the notion of a *hierarchy*. For any node u in $\mathcal{V}$, the singleton set $\{u\}$ is a hierarchy iff $\mathit{distance}(u, u) = 0$. A set of nodes U such that $|U| > 1$ is a hierarchy iff there is a partition of U into $k > 1$ disjoint hierarchies α_i, $0 \le i < k$, such that $\mathit{distance}(u, v) = \mathit{diameter}(U)$ (resp., $\mathit{distance}(u, v) < \mathit{diameter}(U)$) for all nodes u in α_i and v in α_j for which $i \neq j$ (resp., $i = j$). Note that for any non-singleton hierarchy α, this partition is unique. For any non-singleton hierarchy α with associated partition $\{\alpha_i : 0 \le i < k\}$, we define each hierarchy α_i as a *child* of the hierarchy α, and we define the *parent* of each α_i, denoted $\mathit{parent}(\alpha_i)$, as α. We inductively define the notion of a *descendant* of a hierarchy α as follows: A hierarchy β is a descendant of a hierarchy α iff $\alpha = \beta$ or β is a descendant of some child of α. A descendant β of α is *proper* iff $\alpha \neq \beta$.

Hierarchies can be used to model a large class of distributed networks. For example, a homogeneous, k-node local-area network may be modeled as a single-level hierarchy. In fact, this is precisely the model used in [2, 40] in the study of caching schemes for networks of workstations. Furthermore, it seems plausible that some nontrivial hierarchy should provide a reasonable first-order model for a complex, heterogeneous wide-area network such as the Internet.

Hierarchies can also be used to model multi-level storage; we can introduce a hierarchy for each level of storage and incorporate the disparate speeds of the different levels in the distance function. As a simple example, consider a machine with two levels of storage (e.g., memory and disk) having local access latencies a and b, $a < b$. This machine may be modeled as a hierarchy with diameter b and two children: (i) a singleton hierarchy with zero access frequencies and storage capacity equal to that of the slow level of storage, and (ii) a hierarchy with diameter a and two singleton children, one with zero access frequencies and storage capacity equal to that of the fast level of storage, and another with zero storage and access frequencies equal to those of the original machine. This approach can be generalized to capture both network and cache latencies in a heterogeneous distributed network with caches of varying speeds.

3.2.2 The hierarchical placement problem

Having fixed the tuple $(\Psi, \mathcal{V}, \mathit{distance}, \mathit{frequency}, \mathit{size}, \mathit{penalty})$ as specified at the beginning of Section 3.2, any descendant of the hierarchy $\mathcal{V}$ determines an instance of the hierarchical placement problem. We now present a sequence of definitions leading up to the definition of the hierarchical placement problem.

It is convenient to extend the definitions of the functions *frequency* and *size* to act on hierarchies. For any hierarchy α and any object ψ, we define $\mathit{frequency}(\alpha, \psi)$ as the sum of $\mathit{frequency}(u, \psi)$ over all nodes u in α. For any hierarchy α, we define $\mathit{size}(\alpha)$ as the sum of $\mathit{size}(u)$ over all nodes in α. The following definitions involving hierarchies will also prove to be useful. For any proper descendant α of $\mathcal{V}$, we define $\mathit{miss}(\alpha)$ as $\mathit{diameter}(\mathit{parent}(\alpha))$. For the hierarchy $\mathcal{V}$ itself, we define $\mathit{miss}(\mathcal{V})$ as $\mathit{penalty}$. For any hierarchy α and any object ψ, we define $\mathit{value}(\alpha, \psi)$ as $\mathit{frequency}(\alpha, \psi) \cdot (\mathit{miss}(\alpha) - \mathit{diameter}(\alpha))$.

A *copy* is a pair (α, ψ) where α is a hierarchy and ψ is an object. (It is worth remarking that the placements that we will define are more general than the ordinary (*concrete*) placements, in the sense that they allow objects to be placed at non-singleton hierarchies as well.) A copy (α, ψ) is *concrete* iff α is singleton. A set of copies is *concrete* iff it is a set of concrete copies. A *refinement* of a set of copies P is a set of copies Q for which $|P| = |Q|$ and there exists a bijection $\phi : P \rightarrow Q$ such that for all $p = (\alpha, \psi)$ in P, the copy $\phi(p) = (\beta, \psi)$ for some descendant β of α. A concrete set of copies P is *feasible* iff $|\{(\alpha, \psi) \in P : \psi \in \Psi\}| \leq size(\alpha)$ for each singleton hierarchy α in $\mathcal{V}$. A non-concrete set of copies is *feasible* iff it admits a feasible concrete refinement.

A *placement* is a pair (α, P) where α is a hierarchy, P is a set of copies, and each copy in P is of the form (β, ψ) for some descendant β of α. For any placement $A = (\alpha, P)$, we define $Hierarchy(A)$ and $Copies(A)$ as α and P, respectively. For any hierarchy α, a placement A is an *α-placement* iff $Hierarchy(A) = \alpha$. A placement A is *concrete* (resp. *feasible*) iff $Copies(A)$ is concrete (resp., feasible). A placement (α, P) is a *refinement* of a placement (β, Q) iff $\alpha = \beta$ and P is a refinement of Q. For any placement A and any descendant α of $Hierarchy(A)$, we define $restrict(A, \alpha)$ as the placement (α, P) where P is the set of all copies (β, ψ) in $Copies(A)$ such that β is a descendant of α. The following lemma is straightforward to prove.

Lemma 3.2.1 *For any hierarchy α, an α-placement A is feasible iff for all descendants β of α, $|Copies(restrict(A, \beta))| \leq size(\beta)$.* ■

For any placement A and object ψ, a copy p is an *(A, ψ)-copy* iff p belongs to $Copies(A)$ and has associated object ψ. For any placement A and object ψ, we define $count(A, \psi)$ as the number of (A, ψ)-copies. For any placement A, we define $Missing(A)$ as the set of all objects ψ such that $count(A, \psi) = 0$. For any placement A, node u in $Hierarchy(A)$, and object ψ, we define $latency(A, u, \psi)$ as follows: If ψ belongs to $Missing(A)$, then $latency(A, u, \psi) = miss(Hierarchy(A))$; otherwise, $latency(A, u, \psi)$ is the minimum value of $diameter(\alpha)$ over all descendants α of $Hierarchy(A)$ such that u belongs to α and $count(restrict(A, \alpha), \psi) > 0$. Note that if A is a refinement of B, then $latency(A, u, \psi) \leq latency(B, u, \psi)$. For any placement A, we define the *cost* of A, denoted $cost(A)$, as the sum of $frequency(u, \psi) \cdot latency(A, u, \psi)$ over all nodes u in $Hierarchy(A)$ and all objects ψ in Ψ. Given a hierarchy α, the *hierarchical placement problem* is to find a feasible concrete α-placement of minimum cost. We remark that if A is a refinement of B, then $cost(A) \leq cost(B)$; it follows that some minimum-cost feasible α-placement is concrete.

A set of placements is defined to be *comparable* iff each placement in the set has the same associated hierarchy. Two placements A and B are defined to be *coupled* iff they are comparable and $count(A, \psi) = count(B, \psi)$ for all objects ψ. A triple of placements (A, B, C) is defined to be *nice* iff A, B, and C are comparable and A and B are coupled.

3.3 A reduction to minimum-cost flow

In this section we reduce a given instance α of the hierarchical placement problem to a corresponding instance G_α of the minimum-cost flow problem. If the hierarchy α is singleton then the placement problem is trivial; our presentation assumes that α is non-singleton.

The minimum-cost flow instance G_α is constructed as follows. The vertex set consists of the following: a vertex β for every singleton descendant β of α; a vertex $\langle \psi, \beta \rangle$ for every object ψ and descendant β of α; a source s and sink t. The edge set consists of four types of edges: for each singleton descendant β of α and each object ψ, there is a unit-capacity edge $(\langle \psi, parent(\beta) \rangle, \beta)$ with cost $-value(\beta, \psi)$; for each non-singleton proper descendant β of α and each object ψ, there are two parallel edges $(\langle \psi, parent(\beta) \rangle, \langle \psi, \beta \rangle)$ with capacities 1 and ∞, and costs $-value(\beta, \psi)$ and 0, respectively; for each object ψ, there are two parallel edges $(s, \langle \psi, \alpha \rangle)$ with capacities 1 and ∞, and costs $-value(\alpha, \psi)$ and 0, respectively; for each singleton descendant β of α, there is an edge (β, t) with capacity $size(\beta)$ and cost 0.

Let c denote the sum, over all nodes u in α and all objects ψ in Ψ, of $frequency(u, \psi) \cdot miss(\alpha)$. The next two lemmas imply that an integral minimum-cost flow in G_α corresponds to a feasible concrete α-placement of minimum cost.

Lemma 3.3.1 *For every feasible concrete α-placement A, there is an integral flow in G_α with cost equal to $cost(A) - c$.*

Proof: For any hierarchy β, let A_β denote the placement $restrict(A, \beta)$. We construct the desired flow as follows. For each edge $(\langle \psi, parent(\beta) \rangle, \beta)$, we set the flow to 1 if

$count(A_\beta, \psi) > 0$, and to 0 otherwise. For each unit-capacity edge $(\langle\psi, parent(\beta)\rangle, \langle\psi, \beta\rangle)$, we set the flow to 1 if $count(A_\beta, \psi) > 0$, and to 0 otherwise. For each infinite-capacity edge $(\langle\psi, parent(\beta)\rangle, \langle\psi, \beta\rangle)$, we set the flow to $\max\{0, count(A_\beta, \psi) - 1\}$. For each edge $(s, \langle\psi, \alpha\rangle)$, we set the flow to 1 if $count(A_\beta, \psi) > 0$, and to 0 otherwise. For each edge (β, t), we set the flow to the sum over all objects ψ of $count(A_\beta, \psi)$. It is straightforward to prove that the above flow is feasible and has cost equal to $cost(A) - c$. ■

Lemma 3.3.2 *For every integral minimum-cost flow with cost c' in G_α, there is a feasible concrete α-placement A such that $cost(A) = c + c'$.*

Proof: Given an integral minimum-cost flow in G_α, we define a corresponding concrete α-placement A as follows: For any singleton descendant β of α and any object ψ, $Copies(A)$ includes the concrete copy (β, ψ) iff the flow along edge $(\langle\psi, parent(\beta)\rangle, t)$ is 1. (Note that A is concrete and feasible.) It remains to establish that $cost(A) = c + c'$. To establish this, we make the following key observation regarding the flows on adjacent parallel arcs. Let e_0 and e_1 denote the parallel edges with capacities 1 and ∞, respectively, between a pair of vertices. If the flow along e_1 is positive, then the flow along e_0 is 1. Otherwise, transferring a unit of flow from e_1 to e_0 would yield another feasible flow with smaller cost, a contradiction. It follows that if $count(restrict(A, \beta), \psi) > 0$ for some proper descendant β of α, then the flow along edge $(\langle\psi, parent(\beta)\rangle, \langle\psi, \beta\rangle)$ is 1, and hence there is an appropriate negative contribution to the cost of the flow. ■

3.4 An approximation algorithm

While the algorithm of Section 3.3 computes an optimal solution to the hierarchical placement problem, its run-time complexity is prohibitively high, at least quadratic in the product of the number of nodes n and the number of objects m. Moreover, since the algorithm is centralized, it requires all the frequency information to be transferred to a single central node, thus incurring a high bandwidth complexity as well. This motivates us to seek a simpler and faster approximation algorithm. Two candidate algorithms are presented in this section. The first algorithm, which we refer to as the greedy algorithm, uses a natural local improvement heuristic. We show, however, that the greedy algorithm has an approximation ratio of $\Omega(n)$. The lower bound proof leads us to a variant of the greedy algorithm that we refer to as the amortizing algorithm. Section 3.5 establishes that the amortizing algorithm is a constant-factor approximation algorithm. Section 3.7 outlines an efficient distributed implementation of the amortizing algorithm.

Given any feasible non-concrete placement A, the following simple procedure can be used to obtain a feasible concrete refinement of A. Note that $Copies(A)$ contains a copy p of the form (α, ψ) for some non-singleton hierarchy α and object ψ. Using Lemma 3.2.1, it follows that $|Copies(restrict(A, \alpha))$ is less than or equal to $size(\alpha)$. This along with the existence of copy p implies that there exists a child β of α for which $|Copies(restrict(A, \beta))| < size(\beta)$. Hence we can obtain a feasible refinement of A by removing p from $Copies(A)$ and replacing it with the copy (β, ψ). Repeated application of this argument yields a concrete refinement of A. (Remark: For any feasible placement A, a minimum-cost feasible concrete refinement of A can be obtained by solving a suitably defined weighted matching problem.)

The greedy and amortizing placement algorithms described in Sections 3.4.1 and 3.4.2, respectively, each compute a placement A that is feasible but not necessarily concrete. The refinement procedure of the preceding paragraph may then be applied to obtain a feasible concrete placement B such that $cost(B) \leq cost(A)$.

The greedy and amortizing algorithms, though well-defined for arbitrary hierarchies, are only intended to be directly applied to hierarchies that are "λ-separated" for some constant $\lambda > 1$. A hierarchy U is *λ-separated* iff $miss(V) \geq \lambda \cdot diameter(V)$ for every descendant V of U. It is straightforward to show that any c-approximation algorithm for the λ-separated hierarchical placement problem implies a $c\lambda$-approximation algorithm for the hierarchical placement problem. (The main idea is to transform the given hierarchy into a λ-separated hierarchy by rounding up all distances to the nearest integral power of λ.) Thus, for the purposes of obtaining a constant-factor approximation algorithm, we may assume without loss of generality that the input hierarchy is λ-separated for an arbitrary constant $\lambda > 1$.

3.4.1 The greedy placement algorithm

We now present a sequence of definitions leading up to the formal description of the greedy algorithm. (For a less formal description, we refer the reader to Chapter 4.)

For any placement A and any copy p in $\mathit{Copies}(A)$, we define $\mathit{min\text{-}benefit}(A, p)$ as the amount by which $\mathit{cost}(A)$ would increase if p were removed from $\mathit{Copies}(A)$. Given a placement A for which $\mathit{Copies}(A)$ is nonempty, the *greedy elimination rule* removes from $\mathit{Copies}(A)$ the copy p minimizing $\mathit{min\text{-}benefit}(A, p)$; this copy is denoted $\mathit{victim}(A)$. (Ties may be broken in an arbitrary consistent manner. For example, we could assign a unique integer ID to each copy, and use these IDs to break ties. Such tie-breaking conventions will be assumed throughout the remainder of the paper without further comment.)

In the following definitions, let A denote a placement, let ψ denote an object, let k denote $|\mathit{Copies}(A)|$, let $A_0 = A$, let A_{i+1} denote the placement $(\mathit{Hierarchy}(A), \mathit{Copies}(A_i) - \{\mathit{victim}(A_i)\})$, $0 \leq i < k$, let p denote a (A, ψ)-copy, and let j denote the maximum value of i such that p belongs to $\mathit{Copies}(A_i)$. We define $\mathit{benefit}(A, p)$ as $\mathit{min\text{-}benefit}(A_j, p)$. The copy p belongs to the set $\mathit{Primary}(A)$ iff there is no other (A_j, ψ)-copy. We define $\mathit{Secondary}(A)$ as $\mathit{Copies}(A) - \mathit{Primary}(A)$. For any placement A, we define $\mathit{secondary\text{-}victim}(A)$ as the copy p in $\mathit{Secondary}(A)$ minimizing $\mathit{benefit}(A, p)$. In case $\mathit{Secondary}(A)$ is empty, then $\mathit{secondary\text{-}victim}(A)$ is undefined and it is convenient to assume that $\mathit{benefit}(A, \mathit{secondary\text{-}victim}(A)) = \infty$.

In order to facilitate the next definition, we assume that $\mathit{Missing}(A)$ is nonempty for any placement A. This assumption is made without loss of generality since the set of objects Ψ can be augmented with arbitrarily many dummy objects for which the associated access frequencies are all zero. For any placement A, we define $\mathit{candidate}(A)$ as the object ψ in $\mathit{Missing}(A)$ maximizing $\mathit{value}(\mathit{Hierarchy}(A), \psi)$.

The swapping procedure of Figure 3.1 is used in all of our approximation algorithms for the hierarchical placement problem. Lemma 3.2.1 implies that if the placement passed to the swapping procedure is feasible, then the updated placement is also feasible.

The proofs of the following six lemmas are straightforward and hence are omitted. The greedy placement algorithm is described after these lemmas.

Swapping Procedure: *Swap*(A, p)
• Set *Copies*(A) to $(Copies(A) - \{p\}) \cup \{(Hierarchy(A), candidate(A))\}$.

Figure 3.1: The swapping procedure. The input is a placement A and a copy p in *Copies*(A).

Lemma 3.4.1 *For any placement A such that $Copies(A)$ is nonempty, and any p in $Copies(A)$, we have $benefit(A, victim(A)) \leq benefit(A, p)$.* ∎

Lemma 3.4.2 *For any placement A and any object ψ such that there is at least one (A, ψ)-copy, a unique (A, ψ)-copy p belongs to $Primary(A)$ and for all (A, ψ)-copies q, $benefit(A, p) \geq benefit(A, q)$.* ∎

Lemma 3.4.3 *Let α denote a λ-separated hierarchy for some $\lambda > 1$, let A denote an α-placement, and let p denote a (A, ψ)-copy in $Copies(A)$. Then $benefit(A, p) \leq \frac{\lambda}{\lambda - 1} \cdot value(\alpha, \psi)$.* ∎

Lemma 3.4.4 *(Delete) Let p denote $victim(A)$ (resp., $secondary\text{-}victim(A)$) for some placement A, and let A' denote the placement $(Hierarchy(A), Copies(A) - \{p\})$. Then the following conditions hold for all q in $Copies(A')$: $benefit(A, q) = benefit(A', q)$; q belongs to $Primary(A)$ (resp., $Secondary(A)$) iff q belongs to $Primary(A')$ (resp., $Secondary(A')$).* ∎

Lemma 3.4.5 *(Insert) Let α denote a hierarchy, let A denote an α-placement, let ψ denote an object in $Missing(A)$, let p denote the copy (α, ψ) and let A' denote the placement $(\alpha, Copies(A) \cup \{p\})$. Then p belongs to $Primary(A')$, $benefit(A', p) = value(\alpha, p)$, and the following claims hold for all q in $Copies(A)$: $benefit(A, q) = benefit(A', q)$; q belongs to $Primary(A)$ (resp., $Secondary(A)$) iff q belongs to $Primary(A')$ (resp., $Secondary(A')$).* ∎

Lemma 3.4.6 *(Combine) Let α denote a non-singleton hierarchy with k children α_i, $0 \leq i < k$, let A_i denote an α_i-placement, $0 \leq i < k$, and let A denote the placement $(\alpha, \cup_{0 \leq i < k} Copies(A_i))$. For any object ψ such that there is at least one (A, ψ)-copy,*

Greedy Algorithm

- **Combining.** Initialize *Hierarchy*(A) to α. If α is singleton, initialize *Copies*(A) to $\{(\alpha,\psi) : \psi \in S\}$, where S is an arbitrarily chosen subset of Ψ of size *size*(α). Otherwise, initialize *Copies*(A) to $\cup_{0\le i<k}$ *Copies*(A_i), where the A_i's are the placements previously computed at the k children of α.
- **Swapping.** While *benefit*$(A, \mathit{victim}(A)) <$ *value*$(\alpha, \mathit{candidate}(A))$, call *Swap*$(A, \mathit{victim}(A))$.

Figure 3.2: The greedy algorithm. We assume that the children, if any, of a given hierarchy α have already been processed, and describe the computation at α.

let p denote the unique (A,ψ)-copy in Primary(A), and let x denote benefit(A,p) − value(α,ψ). Then for any i, $0 \le i < k$, and any (A_i,ψ)-copy q, the following claim holds: If $p = q$ then benefit$(A_i,q) = x$; otherwise, benefit$(A,q) =$ benefit$(A_i,q) \le x$. ∎

The greedy algorithm is presented in Figure 3.2. Using Lemmas 3.4.4 and 3.4.5, it is straightforward to establish termination of the swapping loop; it follows that the greedy algorithm terminates. It is straightforward to prove that the placement A computed by the greedy algorithm is feasible.

One might conjecture the greedy algorithm to be a constant-factor approximation algorithm for the hierarchical placement problem. Unfortunately, this conjecture is false; below we construct an n-node hierarchy α for which the placement computed by the greedy algorithm has cost exceeding the optimal by a $\Theta(n)$ factor.

We label n nodes from 0 to $n-1$ and construct the hierarchy $\alpha = \alpha_{n-1}$ as follows. First, we create a hierarchy α_1 with two singleton children corresponding to nodes 0 and 1. Then, for i running from 2 to $n-1$, we create a hierarchy α_i with two children: the hierarchy α_{i-1} constructed previously and a singleton hierarchy containing node i. The cache size of each node is 1. We define the distance function between the nodes in such a way that *diameter*(α_i) equals n^{i-1}, $1 \le i < n$. The *penalty*, which is required to be at least as large as *diameter*(α), is set to n^{n-1}. We assign nonzero frequencies to n objects ψ_i, $0 \le i < n$. For node 0, *frequency*$(0,\psi_i) = 1/n^i$, $0 \le i < n$. For node 1, *frequency*$(1, \psi_i)$ is 1 if i is 0, and 0 otherwise. For node j, $2 \leq j < n$, *frequency*(j, ψ_i) is 0 for all i.

For this example, the set of copies associated with an optimal feasible concrete α-placement A consists of $(\{0\}, \psi_0)$, $(\{1\}, \psi_1)$, and $(\{i\}, \psi_i)$, $2 \leq i < n$. The cost of A is $2 - 1/n$ since *latency*$(A, 0, \psi_i)$ is n^{i-1}, $1 \leq i < n$, and *latency*$(A, 1, \psi_0)$ is 1. In contrast, we claim that the cost of the greedy α-placement is $n - 1$. (Furthermore, any refinement of this α-placement also has cost $n - 1$.) It can be shown by induction that, for each hierarchy α_i, $1 \leq i < n$, the set of copies associated with the greedy α_i-placement consists of $(\{0\}, \psi_0)$, $(\{1\}, \psi_0)$, and (α_j, ψ_{j-1}), $2 \leq j \leq i$. The cost of the greedy α-placement B is $n - 1$ since *latency*$(B, 0, \psi_i)$ is n^i for $1 \leq i < n$. We conclude that the approximation ratio of the greedy algorithm is $\Omega(n)$.

3.4.2 The amortizing placement algorithm

The construction given at the end of Section 3.4.1 leads us to consider a natural variant of the greedy algorithm that we call the amortizing algorithm. The amortizing algorithm is presented in Figure 3.3. Using Lemmas 3.4.4 and 3.4.5, it is straightforward to establish termination of the amortized swapping loop; it follows that the amortizing algorithm terminates. It is straightforward to prove that the placement A computed by the amortized algorithm is feasible.

An efficient distributed implementation of the amortizing algorithm is given in Section 3.7. It is particularly noteworthy that the amortized swapping loop executed at each non-singleton hierarchy is highly parallelizable.

3.5 Analysis of the amortizing algorithm

In this section we prove our main result, namely, that the cost of the placement constructed by the amortizing algorithm is within a constant factor of optimal. To facilitate our analysis, we introduce another placement algorithm that we call the bridging algorithm. The bridging algorithm computes three comparable feasible placements that we refer to as the amortizing, arbitrary, and bridging placements, respectively. The amortizing placement is identical to that computed by the amortizing algorithm. The

Amortizing Algorithm

- **Combining.** This step is the same as the combining step of the greedy algorithm, except that we also initialize an auxiliary potential variable Φ. If α is singleton, then Φ is set to 0. Otherwise, Φ is set to the sum of the potentials Φ_i, $0 \leq i < k$, computed at the k children of α.
- **Local initialization.** Initialize Δ to the sum over all objects ψ in $\mathit{Missing}(A)$ of $\mathit{value}(\alpha, \psi)$.
- **Amortized swapping.** This step is similar to the swapping step of the greedy algorithm, except that the potential Φ is used to reduce the benefits of certain secondary copies.
  1. Let $x = \mathit{benefit}(A, \mathit{secondary\text{-}victim}(A))$, let $y = \mathit{benefit}(A, \mathit{victim}(A))$, and let $z = \mathit{value}(\alpha, \mathit{candidate}(A))$.
  2. If $x - \Phi \leq \min(y, z)$ then call $\mathit{Swap}(A, \mathit{secondary\text{-}victim}(A))$, subtract z from Δ, set Φ to $\max(0, \Phi - x)$, and goto line 1.
  3. If $y < z$, then call $\mathit{Swap}(A, \mathit{victim}(A))$, add $y - z$ to Δ, and goto line 1.
- **Potential update.** Add Δ to Φ.

Figure 3.3: The amortizing algorithm. We assume that the children, if any, of a given hierarchy α have already been processed, and describe the computation at α.

arbitrary placement is simply an arbitrary concrete placement. The construction of the bridging placement depends on the choice of the arbitrary placement, and is designed to ensure that the cost of the bridging placement can be relatively easily compared to that of the amortizing and arbitrary placements. In particular, we establish our main theorem via the following two main steps. First, we prove that the cost of the amortizing placement is at most that of the bridging placement. Second, we prove that the cost of the arbitrary placement is at least some constant fraction of that of the bridging placement.

3.5.1 The bridging algorithm

For any placement A and copy $p = (\alpha, \psi)$ in $Copies(A)$, we define $Region(A, p)$ as the set of nodes u such that $latency(A, u, \psi)$ would increase if p were removed from $Copies(A)$; it is straightforward to prove that $Region(A, p)$ is a hierarchy. For any comparable placements A and B, we define $Min\text{-}matched(A, B)$ as the set of all $p = (\alpha, \psi)$ in $Copies(A)$ for which there is a copy $q = (\beta, \psi)$ in $Copies(B)$ such that β is a descendant of $Region(A, p)$. We define $Min\text{-}unmatched(A, B)$ as $Copies(A) - Min\text{-}matched(A, B)$.

In the following definitions, let A and B denote two comparable placements, let ψ denote an arbitrary object, and let P (resp., Q) denote the set of all (A, ψ)-copies in $Min\text{-}matched(A, B)$ (resp., $Min\text{-}unmatched(A, B)$). If there are one or more (A, ψ)-copies, then we define $quasivictim(A, B, \psi)$ as follows: If Q is empty (resp., nonempty), then $quasivictim(A, B, \psi)$ is the copy p in P (resp., Q) minimizing $min\text{-}benefit(A, p)$. For the remaining definitions in this paragraph, let k denote the number of (A, ψ)-copies, let $A_0 = A$, let A_{i+1} denote the placement $= (Hierarchy(A), Copies(A_i) - \{quasivictim(A_i, B, \psi)\})$, $0 \leq i < k$, let p denote a (A, ψ)-copy, and let j denote the maximum value of i such that p belongs to $Copies(A_i)$. We define $quasibenefit(A, B, p)$ as $min\text{-}benefit(A_j, p)$. The copy p belongs to the set $Quasiprimary(A, B)$ iff $j = k - 1$. We define $Quasisecondary(A, B)$ as $Copies(A) - Quasiprimary(A, B)$. The copy p belongs to the set $Matched(A, B)$ iff p belongs to $Min\text{-}matched(A_j, B)$. We define $Unmatched(A, B)$ as $Copies(A) - Matched(A, B)$.

The next six lemmas are analogous to Lemmas 3.4.1, 3.4.2, 3.4.3, 3.4.4, 3.4.5, and 3.4.6, respectively. The proofs are all straightforward and hence are omitted.

Lemma 3.5.1 *Let A and B denote two comparable placements, let ψ denote an object such that $count(A, \psi) > 0$, and let P (resp., Q) denote the set of all (A, ψ)-copies in $Matched(A, B)$ (resp., $Unmatched(A, B)$). Then if Q is empty (resp., nonempty), $quasivictim(A, B, \psi)$ belongs to P (resp., Q), and moreover for all p in P (resp., Q), $quasibenefit(A, B, quasivictim(A, B, \psi)) \leq quasibenefit(A, B, p)$.* ■

Lemma 3.5.2 *Let A and B denote two comparable placements, let ψ denote an object such that $count(A, \psi) > 0$, and let P (resp., Q) denote the set of all (A, ψ)-copies in $Matched(A, B)$ (resp., $Unmatched(A, B)$). Then a unique (A, ψ)-copy p belongs to $Quasiprimary(A, B)$, p belongs to P unless P is empty, and $quasibenefit(A, B, p) \geq quasibenefit(A, B, q)$ for all q in P (resp., Q).* ■

Lemma 3.5.3 *Let α denote a λ-separated hierarchy for some $\lambda > 1$, let A and B denote two α-placements, and let p denote an (A, ψ)-copy in $Copies(A)$. Then we have $quasibenefit(A, B, p) \leq \frac{\lambda}{\lambda - 1} \cdot value(\alpha, \psi)$.* ■

Lemma 3.5.4 *(Delete) Let A and B denote two comparable placements, let ψ denote an object that satisfies $count(A, \psi) > 0$, and let A' denote the new placement $(Hierarchy(A), Copies(A) - \{quasivictim(A, B, \psi)\})$. Then the following claims hold for all p in $Copies(A')$: $quasibenefit(A, B, p) = quasibenefit(A', B, p)$; p belongs to $Quasiprimary(A, B)$ (resp., $Quasisecondary(A, B)$, $Matched(A, B)$, $Unmatched(A, B)$) iff p belongs to $Quasiprimary(A', B)$ (resp., $Quasisecondary(A', B)$, $Matched(A', B)$, $Unmatched(A', B)$).* ■

Lemma 3.5.5 *(Insert) Let α denote a hierarchy, let A and B denote two α-placements, let ψ denote an object in $Missing(A)$, let p denote the copy (α, ψ) and let A' denote the placement $(\alpha, Copies(A) \cup \{p\})$. Then p belongs to $Quasiprimary(A', B)$, $quasibenefit(A', B, p) = value(\alpha, p)$, and the following claims hold for all q in $Copies(A)$: $quasibenefit(A, B, q) = quasibenefit(A', B, q)$; q belongs to $Quasiprimary(A, B)$ (resp., $Quasisecondary(A, B)$, $Matched(A, B)$, $Unmatched(A, B)$) if and only if q belongs to $Quasiprimary(A', B)$ (resp., $Quasisecondary(A', B)$, $Matched(A', B)$, $Unmatched(A', B)$).*
■

Lemma 3.5.6 *(Combine) Let α denote a non-singleton hierarchy with k children α_i, $0 \leq i < k$, let A_i and B_i denote two α_i-placements, $0 \leq i < k$, let A denote the placement $(\alpha, \cup_{0 \leq i < k} Copies(A_i))$, let B denote the placement $(\alpha, \cup_{0 \leq i < k} Copies(B_i))$, let P denote $\cup_{0 \leq i < k} Matched(A_i, B_i)$, let Q denote $\cup_{0 \leq i < k} Unmatched(A_i, B_i)$, let ψ denote an object such that $count(A, \psi) > 0$, let p denote the unique (A, ψ)-copy in $Quasiprimary(A, B)$, and let x denote $quasibenefit(A, B, p) - value(\alpha, \psi)$. Then the following claims hold: if P is nonempty then p belongs to P; $quasibenefit(A_i, B, q) = x$; p belongs to $Matched(A, B)$ (resp., $Unmatched(A, B)$) iff $count(B, \psi) > 0$ (resp., $count(B, \psi) = 0$). Furthermore, for any i, $0 \leq i < k$, and any (A_i, ψ)-copy q different from p, the following claims hold: q belongs to $Matched(A, B)$ (resp., $Unmatched(A, B)$) iff q belongs to P (resp., Q); $quasibenefit(A, B, q) = quasibenefit(A_i, B_i, q)$; if q belongs to $Quasisecondary(A_i, B_i)$ then q belongs to $Quasisecondary(A, B)$; if Q is empty or q belongs to Q then $quasibenefit(A, B, q) \leq x$.* ■

In the following definition, let (A, B, C) denote a nice triple of α-placements, let k denote $|Copies(A)|$, let $A_0 = A$ and $B_0 = B$, let A_{i+1} denote the placement $(\alpha, Copies(A_i) - \{victim(A_i)\})$, let B_{i+1} denote the placement $(\alpha, Copies(B_i) - \{quasivictim(B_i, C, \psi)\})$ where ψ denotes the object associated with $victim(A_i)$, $0 \leq i < k$, let p denote a copy in $Copies(B)$, and let j denote the maximum value of i such that p belongs to $Copies(B_i)$. (Note that each triple (A_i, B_i, C), $0 \leq i < k$, is nice.) We define $mate(A, B, C, p)$ as $victim(A_j)$.

The coupled swapping procedure of Figure 3.4 executes a pair of swaps, one involving each of the coupled placements in a given nice triple. Lemmas 3.5.7 and 3.5.8 below are useful for analyzing the effect of a call to the coupled swapping procedure for which the parameter p is either $victim(A)$ or $secondary\text{-}victim(A)$.

Lemma 3.5.7 *(Delete) Let p be equal to $victim(A)$ (resp., $secondary\text{-}victim(A)$) for some placement A belonging to a nice triple (A, B, C), let ψ be such that p is an (A, ψ)-copy, let A' denote the placement $(Hierarchy(A), Copies(A) - \{victim(A)\})$, and let B' denote the placement $(Hierarchy(B), Copies(B) - \{quasivictim(A, B, \psi)\})$. Then (A', B', C) is nice.*

Proof: Straightforward from Lemmas 3.4.4 and 3.5.4. ■

Coupled Swapping Procedure: $Swaps(A, B, C, p)$

- Let ψ be such that p is a (A, ψ)-copy.
- Call $Swap(A, p)$ and $Swap(B, quasivictim(B, C, \psi))$.

Figure 3.4: The coupled swapping procedure. The input is a nice triple of placements (A, B, C) and a copy p in $Copies(A)$. Because A and B are coupled, $candidate(A) = candidate(B)$ and hence the same copy is inserted in both calls to the swapping procedure. It follows that the output triple (A, B, C) is nice.

Lemma 3.5.8 *(Insert) Let (A, B, C) denote a nice triple of placements, let α denote their common hierarchy, let ψ denote an object in $Missing(A)$, let A' denote the placement $(\alpha, Copies(A) \cup \{(\alpha, \psi)\})$, and let B' denote the placement $(\alpha, Copies(B) \cup \{(\alpha, \psi)\})$. Then (A', B', C) is nice.*

Proof: Straightforward from Lemmas 3.4.5 and 3.5.5. ■

Lemma 3.5.9 *(Combine) Let α denote a non-singleton hierarchy with k children α_i, $0 \le i < k$, let A_i, B_i, and C_i denote α_i-placements such that the triple (A_i, B_i, C_i) is nice, $0 \le i < k$, and let A, B, and C denote the placements $(\alpha, \cup_{0 \le i < k} Copies(A_i))$, $(\alpha, \cup_{0 \le i < k} Copies(B_i))$, and $(\alpha, \cup_{0 \le i < k} Copies(C_i))$, respectively. Then (A, B, C) is nice.*

Proof: Straightforward from Lemmas 3.4.6 and 3.5.6. ■

The bridging algorithm is presented in Figure 3.5. The program variables A, B, and C correspond to the amortizing, bridging, and arbitrary placements, respectively.

The following lemma implies that the swaps associated with the bridging placement are well-defined. Given this lemma, it is straightforward to prove that all of the placements associated with the bridging algorithm are feasible. Furthermore, it is straightforward to prove that the amortizing placement A computed by the bridging algorithm is the same as the placement computed by the amortizing algorithm.

Lemma 3.5.10 *After the combining step of the bridging algorithm, and after each iteration of the amortized swapping loop of the bridging algorithm, (A, B, C) is nice.*

Bridging Algorithm

- **Combining.** Initialize A and Φ as in the amortizing algorithm. If α is singleton, initialize B and C to A. Otherwise, initialize $\mathit{Hierarchy}(B)$ (resp., $\mathit{Hierarchy}(C)$) to α, and initialize $\mathit{Copies}(B)$ (resp., $\mathit{Copies}(C)$) to $\cup_{0 \le i < k} \mathit{Copies}(B_i)$, where the B_i's (resp., C_i's) are the bridging (resp., arbitrary) placements previously computed at the k children of α.
- **Local initialization.** Set Δ to 0 and Υ to $\mathit{Missing}(A)$.
- **Amortized swapping.** This step applies the same sequence of swaps to the amortizing placement A as in the amortizing algorithm. Each of these swaps is accompanied by a corresponding swap involving the bridging placement B. (For a proof that the latter swaps are well-defined, see Lemma 3.5.10 below.)

  1. Let $x = \mathit{benefit}(A, \mathit{secondary\text{-}victim}(A))$, let $y = \mathit{benefit}(A, \mathit{victim}(A))$, and let $z = \mathit{value}(\alpha, \mathit{candidate}(A))$.
  2. If $x - \Phi \le \min(y, z)$ then call $\mathit{Swaps}(A, B, C, \mathit{secondary\text{-}victim}(A))$, set Φ to $\max(0, \Phi - x)$, remove $\mathit{candidate}(A)$ from Υ, and goto line 1.
  3. If $y < z$, then call $\mathit{Swaps}(A, B, C, \mathit{victim}(A))$, add y to Δ, remove $\mathit{candidate}(A)$ from Υ, and goto line 1.

- **Accounting.** For each object ψ in Υ, add $\mathit{value}(\alpha, \psi)$ to Δ and remove ψ from Υ.
- **Potential update.** Add Δ to Φ.

Figure 3.5: The bridging algorithm. We introduce this algorithm for the sole purpose of analyzing the amortizing algorithm. We assume that the children, if any, of a given hierarchy α have already been processed, and describe the computation at α.

Proof: Lemmas 3.5.7 and 3.5.8 imply that the claim cannot fail for the first time after an iteration of the amortized swapping loop. We now argue that the claim cannot fail for the first time after the combining step associated with some hierarchy α. If α is singleton, then A, B, and C are all equal and the claim follows. If α is non-singleton then (A_i, B_i, C_i) are nice, $0 \leq i < k$, then Lemma 3.5.9 implies that (A, B, C) is nice. The lemma follows since no other part of the code modifies A, B, or C. ■

3.5.2 Cost comparison: amortizing versus bridging

In this section we compare the cost of the amortizing and bridging placements computed by the bridging algorithm for a given hierarchy.

For any multiset of reals X, we define $sum(X)$ as the sum of the elements of X. For any multiset of reals X, and any integer i such that $0 \leq i \leq |X|$, we define $Big(X, i)$ (resp., $Little(X, i)$) as the multiset consisting of the i largest (resp., smallest) reals in X. For any two multisets of reals X and Y such that $|X| = |Y|$, we write $X \leq Y$ to mean that the ith largest element of X is less than or equal to the ith largest element of Y, $1 \leq i \leq |X|$. A triple (X, Y, Z) is defined to be *good* iff the following conditions hold: (i) X, Y, and Z are finite multisets of reals, (ii) $|X| + |Y| = |Z|$, (iii) $X \geq Little(Z, |X|)$, (iv) $Y \leq Big(Z, |Y|)$, and (v) $sum(X) + sum(Y) \leq sum(Z)$.

Lemma 3.5.11 *For any real x, the triples $(\{x\}, \{\}, \{x\})$ and $(\{\}, \{x\}, \{x\})$ are good.* ■

Lemma 3.5.12 *Let (X, Y, Z) be good and assume that $Z' = Big(Z, |Z| - 1)$. If X is empty, then let $X' = X$ and $Y' = Big(Y, |Y| - 1)$. Otherwise, let $X' = Big(X, |X| - 1)$ and $Y' = Y$. In either case, (X', Y', Z') is good.* ■

Lemma 3.5.13 *Let (X_i, Y_i, Z_i) be good, $1 \leq i \leq k$, and let $X = \cup_{1 \leq i \leq k} X_i$, $Y = \cup_{1 \leq i \leq k} Y_i$, and $Z = \cup_{1 \leq i \leq k} Z_i$. Then (X, Y, Z) is good. Furthermore, if Y is empty and x is the maximum element of X, then $(X - \{x\}, \{x\}, Z)$ is good.* ■

Lemma 3.5.14 *Let (X, Y, Z) be good and let w be a nonnegative real. Assume that z is a maximum element of Z and let $Z' = (Z - \{z\}) \cup \{w + z\}$. If Y is nonempty,* *then let* $X' = X$ *and* $Y' = (Y - \{y\}) \cup \{w + y\}$ *where* y *is a maximum element of* Y*. Otherwise, let* $Y' = Y$ *and* $X' = (X - \{x\}) \cup \{w + x\}$ *where* x *is a maximum element of* X*. In either case,* (X', Y', Z') *is good.* ∎

Lemma 3.5.15 *Let* A*,* B*, and* C *denote the corresponding program variables after the combining step of the bridging algorithm, or after some iteration of the amortized swapping loop of the bridging algorithm. Let* X*,* Y*, and* Z *denote the multisets* $\{\mathit{quasibenefit}(B, C, p) : p \in \mathit{Unmatched}(B, C)\}$*,* $\{\mathit{quasibenefit}(B, C, p) : p \in \mathit{Matched}(B, C)\}$*, and* $\{\mathit{benefit}(A, p) : p \in \mathit{Copies}(A)\}$*, respectively. Then* (X, Y, Z) *is good.*

Proof: After the combining step, there are two cases to consider. If α is singleton, then $A = B = C$, $\mathit{Unmatched}(B, C)$ is empty, $\mathit{Matched}(B, C) = \mathit{Copies}(A)$, and $\mathit{benefit}(A, p) = \mathit{quasibenefit}(B, C, p)$ for all p in $\mathit{Copies}(A)$; the claim follows. If α is non-singleton, the claim follows by Lemmas 3.4.6, 3.5.6, 3.5.13, and 3.5.14. (Remark: Lemma 3.5.14 takes care of the increase in benefit/quasibenefit associated with the primary/quasiprimary copy of each object.)

It remains to consider the effect of the pair of swapping operations (one applied to A, the other to B) occurring in some iteration of the amortized swapping loop of the bridging algorithm. This pair of swaps can be viewed as a pair of deletions followed by a pair of insertions. Lemmas 3.4.4, 3.5.4, and 3.5.12 imply that the claim holds after the pair of deletions. Lemmas 3.4.5, 3.5.5, and 3.5.11 imply that the claim holds after the pair of insertions. ∎

Lemma 3.5.16 *For any hierarchy* α*, let* A *(resp.,* B*) denote the amortizing (resp., bridging) placement computed by the bridging algorithm. Then* $\mathit{cost}(A) \leq \mathit{cost}(B)$*.*

Proof: Let c denote the cost of the empty α-placement $(\alpha, \{\})$. We then have,

$$\begin{aligned} \mathit{cost}(A) &= c - \sum_{p \in \mathit{Copies}(A)} \mathit{benefit}(A, p), \\ \text{and } \mathit{cost}(B) &= c - \sum_{p \in \mathit{Copies}(B)} \mathit{quasibenefit}(B, C, p). \end{aligned}$$

The claim now follows from Lemma 3.5.15 and condition (v) in the definition of a good triple. ∎

3.5.3 Cost comparison: bridging versus arbitrary

In this section we compare the cost of the bridging and arbitrary placements computed by the bridging algorithm for a given hierarchy. The following lemma is useful for our analysis.

Lemma 3.5.17 *Let A, B, and C denote the corresponding program variables after the combining step of the bridging algorithm, or after some iteration of the amortized swapping loop of the bridging algorithm. Let q belong to $Copies(B)$, let $p = mate(A, B, C, q)$, let $x = benefit(A, p)$, and let $y = quasibenefit(B, C, q)$. If q belongs to $Matched(B, C)$ (resp., $Unmatched(B, C)$) then $x \geq y$ (resp., $x \leq y$).*

Proof: Follows from Lemma 3.5.15 and conditions (iii) and (iv) in the definition of a good triple. ■

We now state our main technical lemma. The proof of this lemma is given in Section 3.6.

Lemma 3.5.18 *Let B (resp., C) denote the bridging (resp., arbitrary) placement computed by the bridging algorithm for a given λ-separated hierarchy. Then $cost(B) \leq (1 + \frac{3\lambda}{\lambda-1}) \cdot cost(C)$.* ■

Given that the arbitrary placement is potentially an optimal placement, we conclude that the cost of the bridging placement is within a constant factor of optimal.

3.5.4 The main theorem

Using Lemmas 3.5.16 and 3.5.18, we obtain the following result.

Lemma 3.5.19 *For any λ-separated hierarchy, the cost of the placement computed by the amortizing algorithm is at most $(1 + \frac{3\lambda}{\lambda-1})$ times optimal.* ■

Recall that while the placement A computed by the amortizing algorithm is not necessarily concrete, A can easily be refined to a concrete placement B such that $cost(B) \leq cost(A)$, as discussed at the beginning of Section 3.4. Lemma 3.5.19 assumes that the given hierarchy is λ-separated as defined in Section 3.4. If this is not the case, we first transform the given hierarchy into a λ-separated hierarchy as indicated in Section 3.4, introducing an extra factor of λ into the approximation bound.

Theorem 3.5.1 *For any hierarchy α and for any constant $\lambda > 1$, the cost of the placement computed by the amortizing algorithm is at most $\lambda \cdot (1 + \frac{3\lambda}{\lambda - 1})$ times optimal.* ■

The above approximation ratio is less than 13.93 for the optimal choice of $\lambda = 1 + \sqrt{3}/2 \approx 1.866$.

3.6 Proof of the main technical lemma

In this section we prove Lemma 3.5.18 which compares the cost of the bridging and the arbitrary placements computed by the bridging algorithm. Throughout this section, we let A, B, and C denote the program variables of the bridging algorithm corresponding to the amortizing, bridging, and arbitrary placements. Note that by Lemma 3.5.10, the triple (A, B, C) is nice and therefore $\mathit{Hierarchy}(A) = \mathit{Hierarchy}(B) = \mathit{Hierarchy}(C)$. We let α denote this common hierarchy, and furthermore if α is not a singleton we let α_i, $0 \le i < k$, denote the children of α. For conciseness, the above notational conventions are assumed throughout this section without further repetition.

To facilitate the comparison of the bridging placement B with the arbitrary placement C, we introduce and maintain another comparable placement D that is closely related to C.

The rest of this section is organized as follows. First, in Section 3.6.1 we introduce the notions of "emulation" and "domination" to describe the relationship that we maintain between the placements B, C, and D. In Section 3.6.2 we list the variables used to specify the state of the computation. In Section 3.6.3 we list a number of invariants that are claimed to hold at particular points in the execution of the bridging algorithm, and prove that Lemma 3.5.18 follows from these invariants. Finally, in Sections 3.6.4 through 3.6.9 we examine how the state is affected by each step of the algorithm, and prove that the claimed invariants are indeed maintained.

3.6.1 Emulation and domination

A placement D is said to be a (B, C)-*emulator* iff the following two conditions hold for all objects ψ and all descendants β of α: if $\mathit{count}(\mathit{restrict}(C, \beta), \psi) = 0$ then $\mathit{count}(\mathit{restrict}(D, \beta), \psi) = 0$; and moreover, if both $\mathit{count}(\mathit{restrict}(C, \beta), \psi) > 0$ and $\mathit{count}(\mathit{restrict}(B, \beta), \psi) > 0$ then $\mathit{count}(\mathit{restrict}(D, \beta), \psi) > 0$. Note that C itself is a (B, C)-emulator. Furthermore, if D is a (B, C)-emulator, $\mathit{count}(\mathit{restrict}(C, \beta), \psi) > 0$, and $\mathit{count}(\mathit{restrict}(B, \beta), \psi) = 0$, then there is no requirement on $\mathit{count}(\mathit{restrict}(D, \beta), \psi)$. The next two lemmas are straightforward and are stated without proof.

Lemma 3.6.1 *If D is a (B, C)-emulator, then $\mathit{Matched}(B, C) = \mathit{Matched}(B, D)$ and $\mathit{Unmatched}(B, C) = \mathit{Unmatched}(B, D)$.* ∎

Lemma 3.6.2 *If D is a (B, C)-emulator, then $|\mathit{Unmatched}(B, C)| \geq |\mathit{Copies}(B)| - |\mathit{Copies}(D)|$.* ∎

The placement B is said to *dominate* a comparable placement D iff the following condition holds for all objects ψ and for all descendants β of α: If $\mathit{count}(\mathit{restrict}(B, \beta), \psi) = 0$ then $\mathit{count}(\mathit{restrict}(D, \beta), \psi) = 0$. The following lemma is immediate.

Lemma 3.6.3 *If B dominates D, then $\mathit{cost}(B) \leq \mathit{cost}(D)$.* ∎

We now introduce a slightly weakened version of domination. Given a set S of objects, the placement B is said to "S-dominate" a comparable placement D iff the following two conditions hold for all objects ψ and for all descendants β of α: if ψ does not belong to S and $\mathit{count}(\mathit{restrict}(B, \beta), \psi) = 0$, then $\mathit{count}(\mathit{restrict}(D, \beta), \psi) = 0$; if ψ belongs to S then $\mathit{count}(B, \psi) = 0$, $\mathit{count}(D, \psi) = 1$, and the unique copy of ψ in $\mathit{Copies}(D)$ is (α, ψ). Note that the placement B dominates D iff B $\emptyset$-dominates D. The following three lemmas will be used to prove that certain emulation and domination properties are preserved during the execution of the bridging algorithm; these lemmas are straightforward and are stated without proof.

Lemma 3.6.4 *(Insert) Let S be an arbitrary set of objects such that B S-dominates D and D is a (B, C)-emulator, let ψ be any object in $\mathit{Missing}(B)$ such that either*

Pruning Procedure: $Prune(D, B, p)$

- Let ψ be such that p is a (B, ψ)-*copy*.
- Update $Copies(D)$ by removing all copies of the form (β, ψ), where β is a descendant of $Region(B, p)$.

Figure 3.6: The pruning procedure. The input is two comparable placements D and B, and a copy p in $Copies(B)$ such that $p = quasivictim(B, D, \psi)$ belongs to $Matched(B, D)$. This procedure is used to modify the placement D whenever the placement B is modified by deleting the copy p from $Copies(B)$. (Remark: By Lemma 3.6.1, if D is a (B, C)-emulator, then $Matched(B, C) = Matched(B, C)$ and $quasivictim(B, C, \psi) = quasivictim(B, D, \psi)$.)

$\psi \in S$ or $count(C, \psi) = 0$, let p denote the copy (α, ψ), and let B' denote the placement $(\alpha, Copies(B) \cup \{p\})$. Then D is a (B', C)-emulator, and if $\psi \notin S$ (resp., $\psi \in S$) then B' S-dominates (resp., $(S - \{\psi\})$-dominates) D. ■

Lemma 3.6.5 *(Delete) Let S be an arbitrary set of objects such that B S-dominates D and D is a (B, C)-emulator. For any object ψ such that there is at least one (B, ψ)-copy, let p denote $quasivictim(B, C, \psi)$ and let B' denote $(\alpha, Copies(B) - \{p\})$. If p is in $Unmatched(B, C)$, then let $D' = D$; otherwise, let D' denote the new value of D after a call to $Prune(D, B, p)$. Then D' is a (B', C)-emulator and B' S-dominates D'. Furthermore, if p belongs to $Matched(B, C)$, then $|Copies(D')| \leq |Copies(D)| - 1$ and $cost(D') \leq cost(D) + quasibenefit(B, C, p)$.* ■

Lemma 3.6.6 *(Combine) Assume that α is non-singleton and let B_i, C_i, and D_i denote three α_i-placements, $0 \leq i < k$, such that D_i is a (B_i, C_i)-emulator and B_i dominates D_i. Assume that B, C, and D are equal to the placements $(\alpha, \cup_{0 \leq i < k} Copies(B_i))$, $(\alpha, \cup_{0 \leq i < k} Copies(C_i))$, and $(\alpha, \cup_{0 \leq i < k} Copies(D_i))$, respectively. Further, let D' denote the placement $(\alpha, Copies(D) \cup P)$ where P is the set of copies $\{(\alpha, \psi) : count(D, \psi) = 0 \ \wedge \ count(C, \psi) > 0\}$. Then, D' is a (B, C)-emulator and B S-dominates D', where $S = \{\psi : count(B, \psi) = 0 \ \wedge \ count(D', \psi) > 0\}$.* ■

3.6.2 State variables

In this section we specify the variables that are used to capture the state of the computation. In addition to the program variables that appear in the pseudocode of Section 3.5.1, we also define a number of auxiliary variables. Each auxiliary variable is classified as either independent or dependent. We modify the values of the independent variables explicitly, in effect augmenting the pseudocode. The value of each dependent auxiliary variable is determined by the values of the program variables and the independent auxiliary variables. Below is a list of all state variables.

1. **Program variables**: Placements A, B, and C; potential Φ; change in potential Δ; the set of objects Υ.

2. **Independent auxiliary variables**: (i) the placement D; (ii) the nonnegative reals *deficit*, *surplus*, *newdeficit*, and *newsurplus*; (iii) the nonnegative integers *numdead* and *numlift*; (iv) a color, either red or blue, for each copy in $\mathit{Copies}(B)$.

3. **Dependent auxiliary variables**: (i) *numred*, the number of copies that are colored red; (ii) *threshold*, defined as $\min(\max(0, x - \Phi), y)$ where x denotes $\mathit{benefit}(A, \mathit{secondary\text{-}victim}(A))$ and y denotes $\mathit{benefit}(A, \mathit{victim}(A))$; (iii) the two sets of objects $\Upsilon_U = \Upsilon \cap \{\psi : \mathit{count}(C, \psi) = 0\}$ and $\Upsilon_M = \Upsilon - \Upsilon_U$; (iv) the six sets of copies $\hat{P}$, $\hat{Q}$, $\hat{R}$, $\check{P}$, $\check{Q}$, and $\check{R}$.

   The last six sets partition the set $\mathit{Copies}(B)$ as follows: $\hat{P}$ (resp., $\check{P}$) is the set of all blue copies in $\mathit{Matched}(B, C)$ that belong to $\mathit{Quasiprimary}(B, C)$ (resp., $\mathit{Quasisecondary}(B, C)$); $\hat{Q}$ (resp., $\check{Q}$) is the set of all blue copies in $\mathit{Unmatched}(B, C)$ that belong to $\mathit{Quasiprimary}(B, C)$ (resp., $\mathit{Quasisecondary}(B, C)$); $\hat{R}$ (resp., $\check{R}$) is the set of all red copies in $\mathit{Unmatched}(B, C)$ that belong to $\mathit{Quasiprimary}(B, C)$ (resp., $\mathit{Quasisecondary}(B, C)$). It is worth remarking that our coloring mechanism guarantees that none of the copies in $\mathit{Matched}(B, C)$ are colored red, and hence the above six sets alone partition the set $\mathit{Copies}(B)$.

3.6.3 The invariant properties

In this section, we list certain properties that are claimed to hold at various points in the execution of the bridging algorithm.

Invariant 3.6.1 *The following properties hold after the combining step of the bridging algorithm: (i) B S-dominates D, where $S = \{\psi : \mathit{count}(B,\psi) = 0 \ \wedge \ \mathit{count}(C,\psi) > 0\}$; (ii) D is a (B,C)-emulator; (iii) $|\mathit{Copies}(D)| \leq |\mathit{Copies}(B)| - \mathit{numdead}$; (iv) $\mathit{numred} \leq \mathit{numdead}$; (v) $(1 + \frac{3\lambda}{\lambda-1})\mathit{cost}(C) \geq \mathit{cost}(D) - \mathit{deficit} + \mathit{surplus} + \sum_{r \in \check{Q} \cup \hat{Q}} 3 \cdot \mathit{quasibenefit}(B,C,r) + \sum_{r \in \hat{R}} 2 \cdot \mathit{quasibenefit}(B,C,r) + \sum_{\psi \in T} 3 \cdot \mathit{value}(\alpha,\psi)$, where $T = \{\psi : \mathit{count}(B,\psi) = 0 \wedge \mathit{count}(C,\psi) = 0\}$; (vi) $\mathit{deficit} \leq \min(\Phi, \sum_{r \in \check{Q}} \mathit{quasibenefit}(B,C,r))$; (vii) $\mathit{surplus} \geq \Phi - \mathit{deficit}$.*

Invariant 3.6.2 *The following properties hold after the local initialization step, after each iteration of the amortized swapping loop, and after each iteration of the accounting loop of the bridging algorithm: (i) B Υ_M-dominates D; (ii) D is a (B,C)-emulator; (iii) $|\mathit{Copies}(D)| \leq |\mathit{Copies}(B)| - (\mathit{numdead} + \mathit{numlift})$; (iv) $\mathit{numred} \leq \mathit{numdead}$; (v) $(1 + \frac{3\lambda}{\lambda-1})\mathit{cost}(C) \geq \mathit{cost}(D) - \mathit{deficit} + \mathit{surplus} - \mathit{newdeficit} + \mathit{newsurplus} + \sum_{r \in \check{Q} \cup \hat{Q}} 3 \cdot \mathit{quasibenefit}(B,C,r) + \sum_{r \in \hat{R}} 2 \cdot \mathit{quasibenefit}(B,C,r) + \sum_{\psi \in \Upsilon_U} 3 \cdot \mathit{value}(\alpha,\psi)$; (vi) $\mathit{deficit} \leq \min(\Phi, \sum_{r \in \check{Q}} \mathit{quasibenefit}(B,C,r))$; (vii) $\mathit{surplus} \geq \Phi - \mathit{deficit}$; (viii) $\mathit{numlift} = |\{\psi : \mathit{count}(B,\psi) = 0 \ \wedge \ \mathit{count}(D,\psi) = 0 \ \wedge \ \mathit{count}(C,\psi) > 0\}|$; (ix) $\mathit{newdeficit} \leq \Delta$; (x) $\mathit{newdeficit} \leq \mathit{numlift} \cdot \mathit{threshold}$; (xi) $\mathit{newsurplus} \geq \Delta - \mathit{newdeficit}$.*

Invariant 3.6.3 *The following properties hold after the potential update step of the bridging algorithm: (i) B dominates D; (ii) D is a (B,C)-emulator; (iii) $|\mathit{Copies}(D)| \leq |\mathit{Copies}(B)| - (\mathit{numdead} + \mathit{numlift})$; (iv) $\mathit{numred} \leq \mathit{numdead}$; (v) $(1 + \frac{3\lambda}{\lambda-1})\mathit{cost}(C) \geq \mathit{cost}(D) - \mathit{deficit} + \mathit{surplus} - \mathit{newdeficit} + \mathit{newsurplus} + \sum_{r \in \check{Q} \cup \hat{Q}} 3 \cdot \mathit{quasibenefit}(B,C,r)$; (vi) $\mathit{deficit} \leq \min(\Phi - \Delta, \sum_{r \in \check{Q}} \mathit{quasibenefit}(B,C,r))$; (vii) $\mathit{surplus} \geq \Phi - \Delta - \mathit{deficit}$; (viii) $\mathit{numlift} = |\{\psi : \mathit{count}(D,\psi) = 0 \ \wedge \ \mathit{count}(C,\psi) > 0\}|$; (ix) $\mathit{newdeficit} \leq \Delta$; (x) $\mathit{newdeficit} \leq \mathit{numlift} \cdot w$, where w is such that each $r \in \hat{Q}$ satisfies $\mathit{quasibenefit}(B,C,r) \geq w$ and each $r \in \check{Q}$ satisfies $\mathit{quasibenefit}(B,C,r) - \mathit{deficit} \geq w$; (xi) $\mathit{newsurplus} \geq \Delta - \mathit{newdeficit}$.*

The above invariants are established in Sections 3.6.4 through 3.6.9. Our main technical lemma follows from Invariant 3.6.3.

Proof of Lemma 3.5.18: Property (i) of Invariant 3.6.3 along with Lemma 3.6.3 implies that $cost(B) \leq cost(D)$. Moreover the fact that $cost(D) \leq (1 + \frac{3\lambda}{\lambda-1})cost(C)$ follows from property (v) of Invariant 3.6.3 along with the following two inequalities: (a) $deficit \leq \sum_{r \in \check{Q}} quasibenefit(B, C, r)$ and (b) $newdeficit \leq \sum_{r \in \check{Q} \cup \hat{Q}} quasibenefit(B, C, r)$.

Inequality (a) above follows directly from property (vi). For inequality (b), we use Lemma 3.6.2 to deduce that $|Unmatched(B, C)|$ is at least $|Copies(B)| - |Copies(D)|$, which in turn is at least $numred + numlift$ by properties (iii) and (iv). Therefore, $|\check{Q} \cup \hat{Q}|$, the number of blue copies in $Unmatched(B, C)$ is at least $numlift$. Moreover property (x) implies that for each $r \in \check{Q} \cup \hat{Q}$, $quasibenefit(B, C, r) \geq w$, and hence $\sum_{r \in \check{Q} \cup \hat{Q}} quasibenefit(B, C, r) \geq numlift \cdot w \geq newdeficit$. ■

3.6.4 Local initialization step

In this section, we assume that Invariant 3.6.1 holds before the local initialization step, and prove that Invariant 3.6.2 holds after the local initialization step.

- *State change.* Initialize Υ to $Missing(B)$, and set $\Delta = newdeficit = newsurplus = numlift = 0$.
- *Dependent variables.* $\Upsilon_U = \Upsilon \cap \{\psi : count(C, \psi) = 0\}$ and $\Upsilon_M = \Upsilon \cap \{\psi : count(C, \psi) > 0\}$.
- *Analysis.* All properties of Invariant 3.6.2 either follow from the corresponding properties of Invariant 3.6.1 or are trivially satisfied. Note that Υ_M is equal to the set S in property (i) of Invariant 3.6.1 and Υ_U is equal to the set T in property (v) of Invariant 3.6.1.

Thus Invariant 3.6.2 holds after the local initialization step.

3.6.5 Amortized swapping loop

In this section, we assume that Invariant 3.6.2 holds before an iteration of the amortized swapping loop, and prove that it holds after the iteration. We treat each swap as an insertion followed by a deletion. To facilitate this decomposition, we introduce a slightly stronger version of property (iii) which we call property (iii)$'$: $|\mathit{Copies}(D)| \leq |\mathit{Copies}(B)| - 1 - (\mathit{numdead} + \mathit{numlift})$. We first show in Section 3.6.5.1 that Invariant 3.6.2 and property (iii)$'$ hold after the insertion. Then in Section 3.6.5.2 we show that Invariant 3.6.2 holds after the deletion.

3.6.5.1 Insertion

In this section, we assume that Invariant 3.6.2 holds before an insertion, and prove that Invariant 3.6.2 and property (iii)$'$ hold after the insertion. Let ψ denote the incoming object $\mathit{candidate}(B)$. For notational convenience, we use unprimed (resp., primed) symbols to denote the values of variables before (resp., after) the insertion. If the value of a variable does not change then we use the unprimed symbol throughout.

- *State change.* Let p denote the copy (α, ψ). Set $\mathit{Copies}(A') = \mathit{Copies}(A) \cup \{p\}$, $\mathit{Copies}(B') = \mathit{Copies}(B) \cup \{p\}$, and $\Upsilon' = \Upsilon - \{\psi\}$. Set the color of p to blue.
- *Dependent variables.* If $\mathit{count}(C, \psi) > 0$ then $\Upsilon'_M = \Upsilon_M - \{\psi\}$ and $\hat{P}' = \hat{P} \cup \{p\}$. Otherwise $\Upsilon'_U = \Upsilon_U - \{\psi\}$ and $\hat{Q}' = \hat{Q} \cup \{p\}$.
- *Inequalities.* By Lemma 3.5.5, $\mathit{quasibenefit}(B', C, p) = \mathit{value}(\alpha, \psi)$ and for all copies q in $\mathit{Copies}(B)$, $\mathit{quasibenefit}(B', C, q) = \mathit{quasibenefit}(B, C, q)$.
- *Analysis.* Properties (i) and (ii) follow from Lemma 3.6.4. Property (iii)$'$ follows since $|\mathit{Copies}(B)| = |\mathit{Copies}(B')| - 1$. Properties (iv) and (vii) through (xi) are not affected. Properties (v) and (vi) follow from the above inequalities.

Thus Invariant 3.6.2 and property (iii)$'$ hold after the insertion.

3.6.5.2 Deletion

In this section, we assume that Invariant 3.6.2 and property (iii)$'$ hold before a deletion, and prove that Invariant 3.6.2 holds after the deletion. Throughout this section, let q denote the copy deleted from A, let ψ denote the object associated with q, and let p denote $\mathit{quasivictim}(B, C, \psi)$. Note that q is either $\mathit{secondary\text{-}victim}(A)$ or

$victim(A)$ and $q = mate(A, B, C, p)$. The copy p belongs to either $Quasiprimary(B, C)$ or $Quasisecondary(B, C)$, and accordingly q belongs to $Primary(A)$ or $Secondary(A)$. The deletion is handled differently in each case. Each case is further split into three different subcases depending on the color of p and whether p is in $Matched(B, C)$ or not.

For notational convenience, we use unprimed (resp., primed) symbols to denote the values of variables before (resp., after) the deletion. If the value of a variable does not change then we use the unprimed symbol throughout. In all cases below we use the result of Lemma 3.5.4 that if $B' = (\alpha, Copies(B) - \{p\})$ then for every copy r in $Copies(B')$, $quasibenefit(B', C, r) = quasibenefit(B, C, r)$.

Case 1: $p \in Quasisecondary(B, C) = \check{R} \cup \check{Q} \cup \check{P}$

- *Remark.* In this case Δ, $newdeficit$, $newsurplus$, and $numlift$ do not change. Hence properties (ix), (x), and (xi) are not affected. However the placements B and D, along with the variables Φ, $deficit$, $surplus$, and $numdead$ may change. This affects properties (i) through (viii) and these have to be restored. Of these, properties (i) and (ii) follow from Lemma 3.6.5 while properties (iii), (iv), and (viii) are easy to verify. The crucial properties to establish are (iv), (v), (vi), and (vii).
- *State change.* Set $\Phi' = \max(0, \Phi - benefit(A, q))$.
- *Analysis.* Note that the RHS of (vii), as well as the first term in the RHS of (vi), is reduced by $\min(\Phi, benefit(A, q))$.
- **Case 1.1:** $p \in \check{R}$
  - *State change.* Set $Copies(A') = Copies(A) - \{q\}$ and also $Copies(B') = Copies(B) - \{p\}$.
  - *Dependent variables.* $\check{R}' = \check{R} - \{p\}$, and $numred' = numred - 1$.
  - *Analysis.* Properties (i), (ii), (iii), and (viii) hold and are not affected further.
  - **Case 1.1.1:** $deficit = 0$
    - *State change.* None.

  * *Analysis.* Properties (iv) and (v) are unaffected, while properties (vi) and (vii) are directly satisfied. Hence all properties hold.

- **Case 1.1.2:** $\mathit{deficit} > 0$
  * *Remark.* By property (vi) of Invariant 3.6.2, $\check{Q} \neq \emptyset$.
  * *State change.* Pick an arbitrary copy r from $\check{Q}$ and color it red. Set $\mathit{deficit}' = \max(0, \mathit{deficit} - \mathit{quasibenefit}(B, C, r))$ and $\mathit{surplus}' = \mathit{surplus} + \mathit{quasibenefit}(B, C, r)$.
  * *Dependent variables.* $\check{Q}' = \check{Q} - \{r\}$, $\check{R}' = \check{R} \cup \{r\}$, and $\mathit{numred}' = \mathit{numred}$.
  * *Inequalities.* By Lemma 3.5.17, it follows that $\mathit{quasibenefit}(B, C, r) \geq \mathit{benefit}(A, \mathit{mate}(A, B, C, r)) \geq \mathit{benefit}(A, q)$.
  * *Analysis.* For (iv), note that the LHS and the RHS are both unchanged. For (v), note that the net change in the RHS is $-\mathit{deficit}' + \mathit{surplus}' + \mathit{deficit} - \mathit{surplus} - 3 \cdot \mathit{quasibenefit}(B, C, r)$, which is nonpositive. For (vi), note that the RHS is always nonnegative, and furthermore, if $\mathit{deficit}'$ is greater than zero then the decrease in the LHS is $\mathit{quasibenefit}(B, C, r))$ while the decrease in the RHS is no more than this. For (vii), note that the LHS increases by $\mathit{quasibenefit}(B, C, r)$, which is an upper bound on the cumulative increase in the RHS.

• **Case 1.2:** $p \in \check{Q}$

- *State change.* Set $\mathit{Copies}(A') = \mathit{Copies}(A) - \{q\}$ and also $\mathit{Copies}(B') = \mathit{Copies}(B) - \{p\}$.
- *Dependent variables.* $\check{Q}' = \check{Q} - \{p\}$.
- *Analysis.* Properties (i), (ii), (iii), and (viii) hold and are not affected further. Property (iv) also holds since the LHS and the RHS are both unchanged.
- **Case 1.2.1:** $\mathit{deficit} = 0$
  * *State change.* None.

    * *Analysis.* For (v) and (vii), note that the LHS is unchanged and the RHS only decreases. For (vi), note that the LHS is zero and the RHS remains nonnegative.

  - **Case 1.2.2:** $\mathit{deficit} > 0$
    * *State change.* Set $\mathit{deficit}' = \max(0, \mathit{deficit} - \mathit{quasibenefit}(B, C, p))$ and $\mathit{surplus}' = \mathit{surplus} + \mathit{quasibenefit}(B, C, p)$.
    * *Inequalities.* By Lemma 3.5.17, $\mathit{quasibenefit}(B, C, p) \geq \mathit{benefit}(A, q)$.
    * *Analysis.* For (v), note that the net change in the RHS is $-\mathit{deficit}' + \mathit{surplus}' + \mathit{deficit} - \mathit{surplus} - 3 \cdot \mathit{quasibenefit}(B, C, p)$, which is nonpositive. For (vi), note that the RHS is always nonnegative, and furthermore, if $\mathit{deficit}'$ is greater than zero then the decrease in the LHS is $\mathit{quasibenefit}(B, C, p))$ while the decrease in the RHS is no more than this. For (vii), note that the LHS increases by $\mathit{quasibenefit}(B, C, p)$, which is an upper bound on the cumulative increase in the RHS.

- **Case 1.3:** $p \in \check{P}$
  - *State change.* Set $\mathit{Copies}(A') = \mathit{Copies}(A) - \{q\}$ and also $\mathit{Copies}(B') = \mathit{Copies}(B) - \{p\}$. Set D' to the new value of D after a call to $\mathit{Prune}(D, B, p)$ and set $\mathit{numdead}' = \mathit{numdead} + 1$.
  - *Dependent variables.* $\check{P}' = \check{P} - \{p\}$.
  - *Analysis.* Properties (i), (ii), and (iii) follow from Lemma 3.6.5 and are not affected further. Note that the RHS of (iv) increases by one while the RHS of (v) increases by $\mathit{cost}(D') - \mathit{cost}(D)$, which is at most $\mathit{quasibenefit}(B, C, p)$.
  - **Case 1.3.1:** $\mathit{deficit} = 0$
    * **Case 1.3.1.1:** $\mathit{surplus} \geq \mathit{quasibenefit}(B, C, p)$
      · *State change.* Set $\mathit{surplus}' = \mathit{surplus} - \mathit{quasibenefit}(B, C, p)$.
      · *Inequalities.* By Lemma 3.5.17, $\mathit{quasibenefit}(B, C, p) \leq \mathit{benefit}(A, q)$.
      · *Analysis.* For (iv), note that the LHS does not change. For (v), note that the RHS now reduces by $\mathit{quasibenefit}(B, C, p)$ and hence the

cumulative change is nonpositive. For (vi), note that the LHS is zero. For (vii), note that the LHS reduces by $\mathit{quasibenefit}(B, C, p)$ while the RHS either becomes zero or reduces by at least $\mathit{benefit}(A, q)$.

* **Case 1.3.1.2:** $\mathit{surplus} < \mathit{quasibenefit}(B, C, p)$
    - *Remark.* By Lemma 3.6.2 and property (ii), $|\mathit{Unmatched}(B', C)|$ is at least $|\mathit{Copies}(B')| - |\mathit{Copies}(D')|$, which is at least $\mathit{numdead}' \geq \mathit{numred} + 1$, using properties (iii) and (iv). Hence there exists at least one blue copy in $|\mathit{Unmatched}(B, C)|$ and therefore $|\hat{Q} \cup \check{Q}| \geq 1$.
    - *State change.* Pick an arbitrary copy $r \in \hat{Q} \cup \check{Q}$ and color it red. Set $\mathit{surplus}' = 0$.
    - *Dependent variables.* If r is in $\mathit{Quasiprimary}(B, C)$, then $\hat{Q}' = \hat{Q} - \{r\}$ and $\hat{R}' = \hat{R} \cup \{r\}$. Otherwise $\check{Q}' = \check{Q} - \{r\}$ and $\check{R}' = \check{R} \cup \{r\}$. Moreover $\mathit{numred}' = \mathit{numred} + 1$.
    - *Inequalities.* By Lemma 3.5.17, we have $\mathit{quasibenefit}(B, C, r) \geq \mathit{benefit}(A, \mathit{mate}(A, B, C, r)) \geq \mathit{benefit}(A, q) - \Phi$. By Lemma 3.5.17, this is at least $\mathit{quasibenefit}(B, C, p) - \Phi$. Thus $\mathit{quasibenefit}(B, C, r) + \Phi \geq \mathit{quasibenefit}(B, C, p)$, and hence by property (vii), we obtain $\mathit{quasibenefit}(B, C, r) + \mathit{surplus} \geq \mathit{quasibenefit}(B, C, p)$. Moreover, $\Phi' = 0$, because by property (vii) and the preconditions for case 1.3.1.2, $\Phi \leq \mathit{surplus} < \mathit{quasibenefit}(B, C, p) \leq \mathit{benefit}(A, q)$.
    - *Analysis.* For (iv), note that the LHS and the RHS both increase by one. For (v), note that the RHS further reduces by at least $\mathit{quasibenefit}(B, C, r) + \mathit{surplus}$ which is at least $\mathit{quasibenefit}(B, C, p)$ by the above inequality; it follows that the net change in the RHS of (v) is nonpositive. For (vi), note that the LHS is zero. For (vii), note that the LHS and the RHS are both zero.

- **Case 1.3.2:** $\mathit{deficit} > 0$
    * *Remark.* By property (vi), $\check{Q} \neq \emptyset$.
    * *State change.* Pick an arbitrary copy r from $\check{Q}$ and color it red. Set $\mathit{deficit}' = \max(0, \mathit{deficit} - \mathit{quasibenefit}(B, C, r))$ and $\mathit{surplus}' = \mathit{surplus} +$

$quasibenefit(B, C, r)$.

- *Dependent variables.* $\check{Q}' = \check{Q} - \{r\}$, $\check{R}' = \check{R} \cup \{r\}$, and $numred' = numred + 1$.
- *Inequalities.* By Lemma 3.5.17, it follows that $quasibenefit(B, C, r) \geq benefit(A, mate(A, B, C, r)) \geq benefit(A, q) \geq quasibenefit(B, C, p)$.
- *Analysis.* For (iv), note that the LHS and the RHS both increase by one. For (v), note that the new change in the RHS is $-deficit' + surplus' + deficit - surplus - 3 \cdot quasibenefit(B, C, r)$, which is at most $-quasibenefit(B, C, p)$; it follows that the net change in the RHS is nonpositive. For (vi), note that the RHS is always nonnegative, and furthermore, if $deficit'$ is greater than zero then the decrease in the LHS is $quasibenefit(B, C, r))$ while the decrease in the RHS is no more than this. For (vii), note that the LHS increases by $quasibenefit(B, C, r)$, which is an upper bound on the cumulative increase in the RHS.

Case 2: $p \in Quasiprimary(B, C) = \hat{R} \cup \hat{Q} \cup \hat{P}$

- *Remark.* In this case, Φ, *deficit*, *surplus*, and *numdead* do not change and *numred* can only decrease. Hence properties (iv), (vi), and (vii) are not affected at all. However the placements B and D, along with the variables Δ, *newdeficit*, *newsurplus*, and *numlift* may change. Hence the remaining properties may change and have to be restored. Of these, (i) and (ii) will again follow from Lemma 3.6.5 while (iii) and (viii) are easy to verify. The crucial properties to establish are (v), (ix), (x), and (xi).
- *State change.* Set $\Delta' = \Delta + benefit(A, q)$.
- *Analysis.* Note that the RHS of (ix) and (xi) increases by $benefit(A, q)$.
- **Case 2.1:** $p \in \hat{R}$
  - *State change.* Set $Copies(A') = Copies(A) - \{q\}$ and also $Copies(B') = Copies(B) - \{p\}$. Set $newsurplus' = newsurplus + quasibenefit(B, C, p)$.

- *Dependent variables.* $\hat{R}' = \hat{R} - \{p\}$ and $numred' = numred - 1$.
- *Inequalities.* By Lemma 3.5.17, $quasibenefit(B, C, p) \geq benefit(A, q)$.
- *Analysis.* Properties (i), (ii), (iii), (viii), and (ix) are all easy to verify. For property (v), note that the change in the RHS is $newsurplus' - newsurplus - 2 \cdot quasibenefit(B, C, p)$, which is nonpositive. For (x), note that there is no change in the LHS while the RHS can only increase. For (xi), note that the LHS increases by $quasibenefit(B, C, p)$ while the RHS increases by $benefit(A, q)$.

- **Case 2.2:** $p \in \hat{Q}$
  - *State change.* Set $Copies(A') = Copies(A) - \{q\}$ and also $Copies(B') = Copies(B) - \{p\}$. Set $newsurplus' = newsurplus + quasibenefit(B, C, p)$.
  - *Dependent variables.* $\hat{Q}' = \hat{Q} - \{p\}$.
  - *Inequalities.* By Lemma 3.5.17, $quasibenefit(B, C, p) \geq benefit(A, q)$.
  - *Analysis.* Properties (i), (ii), (iii), (viii), and (ix) are all easy to verify. For property (v), note that the change in the RHS is $newsurplus' - newsurplus - 3 \cdot quasibenefit(B, C, p)$, which is nonpositive. For (x), note that there is no change in LHS while the RHS can only increase. For (xi), note that the LHS increases by $quasibenefit(B, C, p)$ while the RHS increases by $benefit(A, q)$.

- **Case 2.3:** $p \in \hat{P}$
  - *State change.* Set $Copies(A') = Copies(A) - \{q\}$ and also $Copies(B') = Copies(B) - \{p\}$. Furthermore set D' to the new value of D after a call to $Prune(D, B, p)$, set $newdeficit' = newdeficit + benefit(A, q)$, and $numlift' = numlift + 1$.
  - *Dependent variables.* $\hat{P}' = \hat{P} - \{p\}$.
  - *Inequalities.* By Lemma 3.5.17, $quasibenefit(B, C, p) \leq benefit(A, q)$ and $benefit(A, q) = threshold$. By Lemma 3.6.5, we have $cost(D') - cost(D) \leq quasibenefit(B, C, p)$.

- *Analysis.* Properties (i), (ii), (iii), and (viii) hold. For (v), note that the change in RHS is $cost(D') - cost(D) - newdeficit' + newdeficit$, which is at most $quasibenefit(B, C, p) - benefit(A, q)$ and hence is nonpositive. For (ix), note that the LHS and the RHS both change by the same amount. For (x), note that the LHS increases by $benefit(A, q)$ while the RHS increases by *threshold*. For (xi), note that the net change in the RHS is zero while the LHS is unchanged.

Thus Invariant 3.6.2 holds after an iteration of the amortized swapping loop.

3.6.6 Accounting loop

In this section, we assume that Invariant 3.6.2 holds before an iteration of the accounting loop, and prove that it holds after the iteration. Let ψ denote the object that is removed from the set Υ in this iteration. For notational convenience, we use unprimed (resp., primed) symbols to denote the values of variables before (resp., after) the iteration. If the value of a variable does not change then we use the unprimed symbol throughout.

- *Remark.* Since $\Upsilon = \Upsilon_U \cup \Upsilon_M$, ψ belongs to either Υ_U or Υ_M. The placement B and the variables Φ, *deficit*, *surplus*, *numdead*, and *numred* do not change here. Hence properties (iv), (vi), and (vii) are unaffected. However the placement D and the variables Δ, *newdeficit*, *newsurplus*, and *numlift* may change. This affects the other properties and they have to be restored.

- *State change.* Set $\Upsilon' = \Upsilon - \{\psi\}$ and $\Delta' = \Delta + value(\alpha, \psi)$.

- *Analysis.* Note that the RHS of (ix) and (xi) increase by $value(\alpha, \psi)$.

- **Case 1:** $\psi \in \Upsilon_U$

  - *Dependent variables.* $\Upsilon'_U = \Upsilon_U - \{\psi\}$.
  - *State change.* Set $newsurplus' = newsurplus + value(\alpha, \psi)$.
  - *Analysis.* Properties (i), (ii), (iii), (viii), (ix), (x), and (xii) are easy to verify. For (v), note that the change in the RHS is $newsurplus' - newsurplus - 3 \cdot$

$value(\alpha, \psi)$, which is nonpositive. For (xi), note that the LHS and the RHS each increase by the same amount.

- **Case 2:** $\psi \in \Upsilon_M$
  - *Remark.* We have $count(B, \psi) = 0$ and, by property (i), $count(D, \psi) = 1$ where the corresponding copy is (α, ψ). Furthermore, the definition of Υ_M implies that $count(C, \psi) > 0$.
  - *Dependent variables.* $\Upsilon'_M = \Upsilon_M - \psi$.
  - *State change.* Set $Copies(D') = Copies(D) - \{p\}$, where $p = (\alpha, \psi)$. Set $newdeficit' = newdeficit + value(\alpha, \psi)$ and $numlift' = numlift + 1$.
  - *Inequalities.* We have $cost(D') - cost(D) = value(\alpha, \psi)$, and moreover since the object ψ was not swapped in, we have $value(\alpha, \psi) \leq threshold$.
  - *Analysis.* The modification of D preserves property (i) with respect to Υ'_M, while property (ii) is unaffected. Properties (iii) and (viii) hold due to the increase in $numlift$. For (v), note that the change in the RHS is $cost(D') - cost(D) - newdeficit' + newdeficit$, which is zero. For (ix), note that the LHS and the RHS both increase by the same amount. For (x), note that the LHS increases by $value(\alpha, \psi)$ while the RHS increases by $threshold$. For (xi), note that the net change in the RHS is zero while the LHS is unchanged.

Thus Invariant 3.6.2 holds after an iteration of the accounting loop.

3.6.7 Potential update step

In this section, we assume that Invariant 3.6.2 holds before the potential update step, and prove that Invariant 3.6.3 holds after the step. We use unprimed (resp., primed) symbols to denote the values of variables before (resp., after) the potential update step. If the value of a variable does not change then we use the unprimed symbol throughout.

- *Remark.* Note that $\Upsilon = \Upsilon_U = \Upsilon_M = \emptyset$.

- *Analysis.* Property (i) of Invariant 3.6.3 follows from property (i) of Invariant 3.6.2 since $\Upsilon_M = \emptyset$. Properties (ii), (iii), (iv), (viii), (ix), and (xi) are same as the corresponding properties of Invariant 3.6.2 and hold without change. Property (v) of Invariant 3.6.3 follows by dropping the last two summation terms of property (v) in Invariant 3.6.2.

- *State change.* Set $w = \mathit{threshold}$.

- *Analysis.* For property (x) we use the following observations. By Lemma 3.5.17, for each r in $\hat{Q} \cup \check{Q}$ we have $\mathit{quasibenefit}(B, C, r) \geq \mathit{benefit}(A, r')$, where $r' = \mathit{mate}(A, B, C, r)$. Since r' was not swapped out, it follows that $\mathit{benefit}(A, r') \geq \mathit{threshold} = w$. Furthermore, if $r \in \check{Q}$, then $\mathit{benefit}(A, r') - \Phi \geq \mathit{threshold} = w$, and by property (vi) of Invariant 3.6.2, $\mathit{benefit}(A, r') - \mathit{deficit} \geq w$. This along with property (x) of Invariant 3.6.2 implies property (x) of Invariant 3.6.3.

- *State change.* Set $\Phi' = \Phi + \Delta$.

- *Analysis.* Properties (vi) and (vii) follow from the corresponding properties of Invariant 3.6.2.

Thus Invariant 3.6.3 holds after the potential update step.

3.6.8 Combining step at a singleton hierarchy

In this section, we establish the base case of our proof by showing that Invariant 3.6.1 holds after the combining step at a singleton hierarchy α.

- *State change.* Initialize A, B, C, and D to the same arbitrary α-placement and set $\Phi = \mathit{deficit} = \mathit{surplus} = \mathit{numdead} = 0$. Set the color of all the copies in B to blue.

- *Dependent variables.* Initialize the six dependent sets $\hat{R}$, $\hat{Q}$, $\hat{P}$, $\check{R}$, $\check{Q}$, $\check{P}$ using B, C, and the colors of the copies in B. Set $\mathit{numred} = 0$.

- *Inequalities.* We have $\mathit{cost}(C) = \mathit{cost}(D) \geq \sum_{\psi \in T} \mathit{value}(\alpha, \psi)$, where $T = \{\psi : \mathit{count}(B, \psi) = \mathit{count}(C, \psi) = 0\}$. Moreover, $\check{Q} = \hat{Q} = \hat{R} = \emptyset$.

- *Analysis.* Properties (i) and (ii) follow directly from the definitions of domination and emulation, while property (iii) follows directly. Properties (iv), (vi), and (vii) hold trivially. Property (v) follows from the above inequalities.

Thus Invariant 3.6.1 holds after the combining step at a singleton hierarchy.

3.6.9 Combining step at a non-singleton hierarchy

In this section, we assume that α is a non-singleton hierarchy and that Invariant 3.6.3 holds after the potential update step at each child α_i, and prove that Invariant 3.6.1 holds after the combining step at α. (Recall that α_i, $0 \leq i < k$, denote the children of α.) We use the subscript i on the state variables to denote the variable computed at the child α_i.

1. *State change.* Initialize $A = (\alpha, \cup_{0 \leq i < k} \mathit{Copies}(A_i))$, $B = (\alpha, \cup_{0 \leq i < k} \mathit{Copies}(B_i))$, and $C = (\alpha, \cup_{0 \leq i < k} \mathit{Copies}(C_i))$. The color of the copies in $\mathit{Copies}(B)$ does not change during this union (i.e., each copy p in $\mathit{Copies}(B_i)$ retains the same color when it enters B.) Initialize $\mathit{numdead} = \sum_{0 \leq i < k} \mathit{numdead}_i$.

2. *Dependent variables.* Variable $\mathit{numred} = \sum_{0 \leq i < k} \mathit{numred}_i$.

3. *State change.* Initialize $D = (\alpha, \cup_{0 \leq i < k} \mathit{Copies}(D_i))$ and then for each object ψ such that $\mathit{count}(C, \psi) > 0$ and $\mathit{count}(D, \psi) = 0$, add a copy (ψ, u) to $\mathit{Copies}(D)$.

4. *Analysis.* Lemma 3.6.6 guarantees that properties (i) and (ii) of Invariant 3.6.1 hold after this modification. Property (iii) is established in Lemma 3.6.7 below. Property (iv) follows by using property (iv) of Invariant 3.6.3, and summing over all children of α. It remains to establish properties (v), (vi), and (vii) by choosing appropriate values for *deficit* and *surplus*. Before choosing these values, we first write down expressions to relate the $\mathit{cost}(C)$ to $\sum_{0 \leq i < k} \mathit{cost}(C_i)$, and the $\mathit{cost}(D)$ to $\sum_{0 \leq i < k} \mathit{cost}(D_i)$. Let $\hat{T}$ denote the set $\{\psi : \mathit{count}(C, \psi) = 0\}$, or equivalently the set $\{\psi : \mathit{count}(D, \psi) = 0\}$. Clearly,

$$\mathit{cost}(C) = \sum_{0 \leq i < k} \mathit{cost}(C_i) + \sum_{\psi \in \hat{T}} \mathit{value}(\alpha, \psi), \tag{3.1}$$

$$cost(D) = \sum_{0 \le i < k} cost(D_i) + \sum_{\psi \in \hat{T}} value(\alpha, \psi). \tag{3.2}$$

Moreover note that this set $\hat{T}$ is precisely the union of Υ_U, and the set of objects corresponding to copies in $\hat{R}$ and $\hat{Q}$. This in conjunction with Lemma 3.5.3 yields

$$\sum_{\psi \in \hat{T}} value(\alpha, \psi) \ge \sum_{\psi \in \Upsilon_U} value(\alpha, \psi) + \sum_{r \in \hat{Q} \cup \hat{R}} \frac{\lambda - 1}{\lambda} \cdot quasibenefit(B, C, r). \tag{3.3}$$

5. *Dependent variables.* Initialize the six dependent sets $\hat{R}$, $\hat{Q}$, $\hat{P}$, $\check{R}$, $\check{Q}$, $\check{P}$ using B, C, and the colors on the copies in B.

6. *Remark.* By Lemma 3.5.6, $\check{Q} = \cup_{0 \le i < k} \check{Q}_i + \cup_{0 \le i < k} \bar{Q}_i$ where $\bar{Q}_i = \hat{Q}_i - \hat{Q}$.

7. *State change.* For each i, $0 \le i < k$, set $deficit'_i = deficit_i + newdeficit_i - (|\hat{Q}_i - \bar{Q}_i|) \cdot w_i$ and $surplus'_i = surplus_i + newsurplus_i + (|\hat{Q}_i - \bar{Q}_i|) \cdot w_i$.

8. *Analysis.* Using the above values of $deficit'_i$ and $surplus'_i$, in Lemma 3.6.8, we establish that

$$\begin{aligned} \left(1 + \frac{3\lambda}{\lambda - 1}\right) cost(C_i) &\ge cost(D_i) - deficit'_i + surplus'_i + \\ &\quad \sum_{r \in \check{Q}_i \cup \bar{Q}_i} 3 \cdot quasibenefit(B_i, C_i, r), && (3.4) \\ deficit'_i &\le \min(\Phi_i, \sum_{r \in \check{Q}_i \cup \bar{Q}_i} quasibenefit(B_i, C_i, r)), && (3.5) \\ surplus'_i &\ge \Phi_i - deficit'_i. && (3.6) \end{aligned}$$

9. *State change.* Initialize Φ, *deficit*, and *surplus* to $\sum_{0 \le i < k} \Phi_i$, $\sum_{0 \le i < k} deficit'_i$, and $\sum_{0 \le i < k} surplus'_i$, respectively.

10. *Analysis.* Summing Equations (3.4), (3.5), and (3.6) over all children α_i, $0 \le i < k$, we obtain

$$\begin{aligned} \sum_{0 \le i < k} \left(1 + \frac{3\lambda}{\lambda - 1}\right) cost(C_i) &\ge \sum_{0 \le i < k} cost(D_i) - deficit + surplus + \\ &\quad \sum_{r \in \check{Q}} 3 \cdot quasibenefit(B, C, r), && (3.7) \\ deficit &\le \min(\Phi, \sum_{r \in \check{Q}} quasibenefit(B, C, r)), && (3.8) \\ surplus &\ge \Phi - deficit. && (3.9) \end{aligned}$$

(Here we used the result of Lemma 3.5.6 that $\check{Q} = \cup_{0 \leq i < k} \check{Q}_i + \cup_{0 \leq i < k} \bar{Q}_i$, and that for each $r \in \check{Q}_i \cup \bar{Q}_i$, $\mathit{quasibenefit}(B_i, C_i, r) = \mathit{quasibenefit}(B, C, r)$.) Properties (vi) and (vii) are essentially the latter two equations. Finally, we establish property (v) using Equations (3.1), (3.2), (3.3), and (3.7), and bounding the quantity $\left(1 + \frac{3\lambda}{\lambda - 1}\right) \mathit{cost}(C)$ as follows:

$$\begin{aligned}
&= \left(1 + \frac{3\lambda}{\lambda - 1}\right) \sum_{0 \leq i < k} \mathit{cost}(C_i) + \sum_{\psi \in \hat{T}} \left(1 + \frac{3\lambda}{\lambda - 1}\right) \mathit{value}(\alpha, \psi) \\
&\geq \sum_{0 \leq i < k} \mathit{cost}(D_i) - \mathit{deficit} + \mathit{surplus} + \sum_{r \in \check{Q}} 3 \cdot \mathit{quasibenefit}(B, C, r) + \\
&\quad \sum_{\psi \in \hat{T}} \left(1 + \frac{3\lambda}{\lambda - 1}\right) \mathit{value}(\alpha, \psi) \\
&\geq \mathit{cost}(D) - \mathit{deficit} + \mathit{surplus} + \sum_{r \in \check{Q}} 3 \cdot \mathit{quasibenefit}(B, C, r) + \\
&\quad \sum_{\psi \in \hat{T}} \left(\frac{3\lambda}{\lambda - 1}\right) \mathit{value}(\alpha, \psi) \\
&\geq \mathit{cost}(D) - \mathit{deficit} + \mathit{surplus} + \sum_{r \in \check{Q} \cup \hat{Q} \cup \hat{R}} 3 \cdot \mathit{quasibenefit}(B, C, r) + \\
&\quad \sum_{\psi \in \Upsilon_U} 3 \cdot \mathit{value}(\alpha, \psi).
\end{aligned}$$

(The above four equations follow by using Equations (3.1), (3.7), (3.2), and (3.3), respectively.)

11. *Conclusion.* Thus Invariant 3.6.1 holds after the combining step.

Lemma 3.6.7 *Suppose that properties (iii) and (viii) of Invariant 3.6.3 hold for each child α_i of α, and that the placements B, C, and D and the variable numdead are initialized as indicated in the preceding state change descriptions. Then property (iii) of Invariant 3.6.1 holds after the initialization.*

Proof: Note that $|\mathit{Copies}(B)| = \sum_{0 \leq i < k} |\mathit{Copies}(B_i)|$ and also that $|\mathit{Copies}(D)| = \sum_{0 \leq i < k} |\mathit{Copies}(D_i)| + |\hat{T}|$ where $\hat{T} = \{\psi : \mathit{count}(D, \psi) = 0 \ \wedge \ \mathit{count}(C, \psi) > 0\}$. Using property (iii) of Invariant 3.6.3 to upper bound $|\mathit{Copies}(D_i)|$, it follows that $|\mathit{Copies}(D)| \leq \sum_{0 \leq i < k}(|\mathit{Copies}(B_i)| - \mathit{numdead}_i - \mathit{numlift}_i) + |\hat{T}| = |\mathit{Copies}(B)| - \mathit{numdead} + (|\hat{T}| - \sum_{0 \leq i < k} \mathit{numlift}_i)$. Moreover, $|\hat{T}| \leq \sum_{0 \leq i < k} \mathit{numlift}_i$, since $\hat{T} \subseteq$

$\cup_{0 \le i < k} \hat{T}_i$, where $\hat{T}_i = \{\psi : \mathit{count}(D_i, \psi) = 0 \wedge \mathit{count}(C_i, \psi) > 0\}$, and by property (viii) of Invariant 3.6.3, $|\hat{T}_i| = \mathit{numlift}_i$. Thus property (iii) holds. ∎

Lemma 3.6.8 *Suppose that Invariant 3.6.3 holds and let $\bar{Q}$ be an arbitrary subset of $\hat{Q}$. Let x and y be two new variables that are set to $\mathit{deficit} + \mathit{newdeficit} - |\hat{Q} - \bar{Q}| \cdot w$ and $\mathit{surplus} + \mathit{newsurplus} + |\hat{Q} - \bar{Q}| \cdot w$, respectively. Then the following three equations hold:*

$$\left(1 + \frac{3\lambda}{\lambda - 1}\right) \mathit{cost}(C) \ \ge \ \mathit{cost}(D) - x + y + \sum_{r \in \check{Q} \cup \bar{Q}} 3 \cdot \mathit{quasibenefit}(B, C, r), \quad (3.10)$$

$$x \ \le \ \min(\Phi, \sum_{r \in \check{Q} \cup \bar{Q}} \mathit{quasibenefit}(B, C, r)), \quad (3.11)$$

$$y \ \ge \ \Phi - x. \quad (3.12)$$

Proof: Let j denote $|\hat{Q}| - |\bar{Q}|$. Property (x) of Invariant 3.6.3 implies that each $r \in \hat{Q}$ satisfies $\mathit{quasibenefit}(B, C, r) \ge w$. Therefore

$$-x + y \le -\mathit{deficit} + \mathit{surplus} - \mathit{newdeficit} + \mathit{newsurplus} + \sum_{r \in \hat{Q} - \bar{Q}} 3 \cdot \mathit{quasibenefit}(B, C, r).$$

Equation (3.10) now follows from property (v) of Invariant 3.6.3. Equation (3.12) follows directly from properties (vii) and (x) of Invariant 3.6.3, since $y = \mathit{surplus} + \mathit{newsurplus} + j \cdot w \ge (\Phi - \Delta - \mathit{deficit}) + (\Delta - \mathit{newdeficit}) + j \cdot w = \Phi - x$.

It remains to establish Equation (3.11). The first part (i.e., $x \le \Phi$), follows directly from properties (vi) and (viii), which imply that $\mathit{deficit} \le \Phi - \Delta$ and $\mathit{newdeficit} \le \Delta$, respectively. For the second part, we first establish that $|\check{Q} + \bar{Q}| \ge (\mathit{numlift} - j)$. By Lemma 3.6.2, $|\mathit{Unmatched}(B, C)|$ is at least $|\mathit{Copies}(B)| - |\mathit{Copies}(D)|$ which, by properties (iii) and (v) of Invariant 3.6.3, is at least $\mathit{numred} + \mathit{numlift}$. Hence the number of blue copies in $\mathit{Unmatched}(B, C)$ (i.e., $|\hat{Q}| + |\check{Q}|$) is at least $\mathit{numlift}$. It follows that $|\check{Q}| + |\bar{Q}| \ge \mathit{numlift} - j$. Now by property (x) of Invariant 3.6.3, each $r \in \check{Q}$ has $\mathit{quasibenefit}(B, C, r) - \mathit{deficit} \ge w$. Moreover, each $r \in \hat{Q}$ satisfies $\mathit{quasibenefit}(B, C, r) \ge w$. Therefore,

$$\sum_{r \in \check{Q} \cup \bar{Q}} \mathit{quasibenefit}(B, C, r) \ge \mathit{deficit} + (\mathit{numlift} - j) \cdot w \ge \mathit{deficit} + \mathit{newdeficit} - j \cdot w = x. \quad (3.13)$$

(For the second inequality, we use property (ix) of Invariant 3.6.3.) This establishes Equation (3.11), thus proving the lemma. ■

3.7 An efficient distributed implementation

The main strength of the amortizing algorithm is that, in contrast with the flow-based algorithm of Section 3.3, it admits a fast distributed implementation. In this section, we briefly sketch such an implementation. The techniques employed are not particularly novel. The main point we wish to emphasize is that while the pseudocode of Section 3.4.2 may appear to be inherently sequential, in fact the algorithm is highly parallelizable.

The amortizing algorithm determines a placement for the given hierarchy α in a bottom-up manner. During the computation, the current placement, along with associated control information, is distributed across the nodes of the network. We begin by describing how this information is organized.

For each object ψ and each descendant β of α, we designate a node in β as the *manager* for ψ in β, denoted $manager(\beta, \psi)$. For load balancing purposes, this manager is chosen at random from β, where the probability of choosing a particular node u is $size(u)/size(\beta)$. For each descendant β of α, we choose a node uniformly at random from β and designate it as the *leader* of β, denoted $leader(\beta)$.

The current α-placement A is distributed across the nodes in α in the following natural manner. For every object ψ, the $manager(\alpha, \psi)$ stores $frequency(\alpha, \psi)$ along with a bit indicating whether ψ belongs to $Missing(A)$. If ψ belongs to $Missing(A)$, then $manager(\alpha, \psi)$ also stores $value(\alpha, \psi)$. Otherwise, for each copy $p = (\beta, \psi)$ in $Copies(A)$, the $manager(\beta, \psi)$ maintains the $benefit(A, p)$ along with a bit indicating whether p belongs to $Primary(A)$. The variables Φ and Δ are maintained by the leader of α. We note that a suitably random selection of managers and leaders ensures that the α-placement A is distributed across the nodes in α in a balanced manner with high probability.

As mentioned earlier, the amortizing algorithm pseudocode of Section 3.4.2 may appear to be inherently sequential. In particular, in the amortized swapping loop, the placement is modified one swap at a time, and the number of swaps could be large. Moreover, the desired swaps, which satisfy some "global" objective, are determined from a distributed placement. Fortunately, as we discuss below, all of the steps in the algorithm can be expressed in terms of instances of a simple prefix sum operation for any given node ordering. (We stress that for the prefix sum operations used in our implementation, the particular ordering of the nodes is not important. In fact, this ordering is allowed to change from one invocation of the operation to another. Consequently, the prefix sum operation can be efficiently implemented on any spanning tree.)

Let us now consider the process of computing a α-placement A at a non-singleton hierarchy α with k children α_i, $0 \le i < k$. Let A_i denote the α_i-placement computed by the amortizing algorithm. The amortizing algorithm proceeds in four steps: the combining step, the local initialization step, the amortized swapping loop, and the potential update step.

In the combining step, A is set to $(\alpha, \cup_{0 \le i<k} A_i)$. This requires no movement of copies in our distributed storage; however there are three computations that are implicit in the description of the algorithm in Section 3.4.2 that need to be performed. These are the calculation of $\mathit{benefit}(A,p)$ for each copy p in $\mathit{Primary}(A)$, the calculation of $\mathit{value}(\alpha,\psi)$ for each object ψ in $\mathit{Missing}(A)$, and the adjustment of the bit for each copy p in $\mathit{Primary}(A_i) - \mathit{Primary}(A)$. Lemma 3.4.6 describes how these benefits and values change during the combining step. The computations for each object ψ are performed by $\mathit{manager}(\alpha,\psi)$, $\mathit{manager}(\alpha_i,\psi)$ for $0 \le i < k$, and by $\mathit{manager}(\beta,\psi)$ for each copy (β,ψ) in $\cup_{0\le i<k}\mathit{Primary}(A_i)$. The computation of the potential Φ is a simple summation involving the leader of α and the leaders of the α_i's.

The local initialization step of the amortizing algorithm involves summation over a subset of nodes in the hierarchy α. This can be implemented efficiently using prefix sums in a straightforward manner.

We now turn to the amortized swapping loop. We implement this loop as an amortization phase followed by a swapping phase. (For the rest of this discussion, we say that a copy p is a *secondary* iff it belongs to $\mathit{Secondary}(A)$.) Let X denote the list of secondaries in A, sorted according to their benefits in nonincreasing order. Let Φ

denote the value of the potential after the combining step. In the amortization phase, we determine the largest prefix X' of X such that the sum of the benefits of copies in X' is at most Φ. (The copies in X' are guaranteed to be swapped out in the amortized swapping loop.) Thus, the amortization phase corresponds to a selection problem. Similarly, the swapping phase corresponds to the following abstract selection problem. Let X and Y denote two lists of numbers sorted in nondecreasing and nonincreasing orders, respectively. (The lists X and Y correspond to the benefits of the copies in the placement, and the values of the objects missing from the placement, respectively.) Our goal is to determine a largest prefix X' of X and a prefix Y' of Y such that $|X'|$ equals $|Y'|$ and no element of X' is greater than any element of Y'.

While the amortization and swapping phases are straightforward to perform sequentially, in the distributed setting the lists are partitioned across the nodes and thus are not available in sorted order. We would like to avoid explicitly sorting these lists, since sorting would require costly large-scale movement of list elements across the network. Moreover, we would like to perform the amortization and swapping steps "in place", that is, without moving the list elements. We now briefly describe an efficient distributed recursive implementation of the amortization phase; a similar approach can be used for the swapping phase. If $|X| \leq 1$ the problem is trivial. Otherwise, we first select a splitter among the secondaries in X. Second, we determine the set Z of those secondaries in X with benefit at most that of the splitter. Third, we sum up the benefits of the secondaries in Z. Finally, in a manner that depends on whether the sum exceeds the given potential Φ, we define a new instance of the problem with fewer elements and recurse. Standard probabilistic arguments can be used to establish a logarithmic bound on the depth of recursion. Moreover, each level of the recursion can be implemented using simple distributed operations such as broadcast and sum. We remark that a more efficient implementation can be obtained by using a large number of splitters to partition the list of secondaries.

The potential update step is straightforward to implement efficiently. In the computation of the placement for a hierarchy α, the combining step requires a constant number of messages per primary stored in the hierarchy and for each leader. The amortized swapping loop and the local initialization step each require $O(\log(\mathit{size}(\alpha)))$ prefix sum operations. The top-down pass that is used to convert the placement to a concrete placement can be easily implemented within the same complexity bounds as for the bottom-up amortizing algorithm.

Chapter 4

Experimental Study of Cooperative Caching

4.1 Introduction

Consider a large-scale distributed information system, such as a digital library or the world wide web. Caching popular objects close to clients is a fundamental technique for improving the performance and scalability of such a system. Caching enables requests to be satisfied by a nearby copy and hence reduces not only the access latency but also the burden on the network as well as the server.

A powerful paradigm to improve cache effectiveness is *cooperation*, where caches cooperate both in serving each other's requests and in making storage decisions. Such cooperation is particularly attractive in environments where machines trust one another such as within an Internet service provider, cache service provider, or corporate intranet. In addition, cooperation across such entities could be based on peering arrangements such as are now common for Internet routing.

There are two orthogonal issues to cooperative caching: finding nearby copies of objects (*object location*) and coordinating the caches while making storage decisions (*object placement*). The object location problem has been widely studied [11, 13, 56]. Many recent studies (e.g., Summary Cache [24], Cache Digest [51], Hint Cache [55], CRISP [26] and Adaptive Web Caching [63]) also focus on the location problem, but

This is joint work with Michael Dahlin. A preliminary version of this work appeared in [36].

none of these address the placement issue which is the focus of this article.

Efficient coordinated object placement algorithms would greatly improve the effectiveness of a given amount of cache space and are hence crucial to the performance of a cooperative caching system. We believe that the importance of such algorithms will increase in the future as the number of shared objects continues to grow enormously and as the Internet becomes the home of more large multimedia objects.

Here we focus on the cache coordination issue and provide placement and replacement algorithms that allow caches to coordinate storage decisions. The placement algorithms attempt to solve the following problem: given a set of cooperating caches, the network distances between caches, and predictions of the access rates from each cache to each object, determine where to place each object in order to minimize the average access cost. Compared to placement algorithms, replacement algorithms also attempt to minimize the access cost, but rather than explicitly computing a placement based on access frequencies, they proceed by evicting objects when cache misses occur.

Coordinated caching helps for two reasons. First, coordination allows a busy cache to utilize a nearby idle cache [21, 25]. Second, coordination balances the improved hit time achieved by increasing the replication of popular objects against the improved hit rate achieved by reducing replication and storing more unique objects.

In this work, we examine an optimal placement algorithm and three practical placement algorithms and compare them to several uncoordinated replacement algorithms (such as LFU, LRU, greedy-dual [12, 61]) and a novel coordinated replacement algorithm. We drive this comparison with simulation based on both synthetic and trace workloads. The synthetic workloads allow us to examine system behavior in a wide range of situations, and the trace allows us to examine performance under a workload of widespread interest: web browsing.

We draw five conclusions from these experiments.

- Cooperative placement can significantly improve performance compared to local replacement particularly when the size of individual caches is limited compared to the universe of objects.

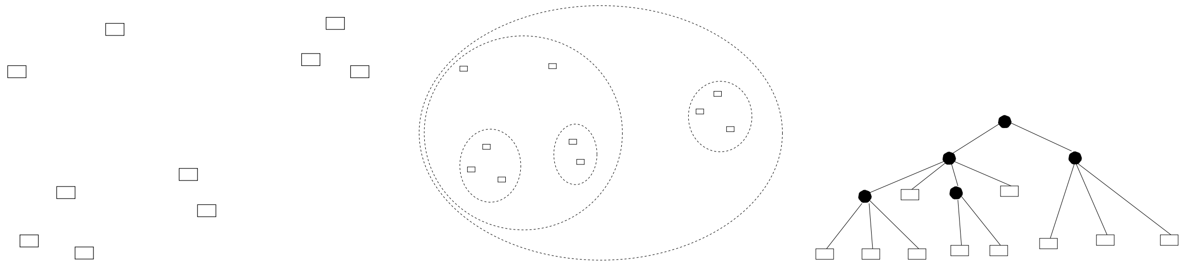
- It was established in Chapter 3 that, under a hierarchical model for distances, the *amortizing* placement algorithm is always within a constant factor of the optimal. Although the proof only guarantees that the amortizing placement algorithm is within a factor of 14 of the optimal, in practice we find that it is within 5% for a wide range of workloads. This is an important result for two reasons. First, in systems that can generate good estimates of access frequencies, amortizing placement is a practical algorithm that can be expected to provide near-optimal performance. Second, for large-scale studies of cache coordination, amortizing placement can provide a practical "best case" baseline that can be used to evaluate other algorithms. In addition, we find that the *greedy* placement algorithm, which is a simplified version of the amortizing algorithm, also provides an excellent approximation of the optimal even though in theory its performance can be arbitrarily worse than the optimal.

- Previous work [12] has shown that the greedy-dual algorithm works well for standalone caches. Our contribution is to examine the performance of this algorithm in cooperative caching scenarios. We find that, for cooperative caching, it significantly outperforms other local replacement algorithms because it includes miss costs in its replacement decisions, thereby creating an implicit channel for coordinating the caches.

- Our *hierarchical-greedy-dual* replacement algorithm yields further improvements over the greedy-dual replacement algorithm especially when there are idle caches in the system.

- A key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses.

The rest of this chapter is organized as follows: First, Section 4.2 describes the algorithms we study. Sections 4.3 and 4.4 detail our experimental results under synthetic and trace workloads, respectively. Finally, Section 4.5 gives a survey of the related work.



(a) A set of cooperating caches (b) A clustering (c) The cluster tree

Figure 4.1: *Model for network distances.*

4.2 Algorithms

In this section, we present several placement and replacement algorithms for coordinated caching. We make several simplifying assumptions in order to focus on the coordination problem. One assumption is that all the objects have the same size and are read-only. Second, we assume that the network distances (or communication costs) between node pairs are fixed and do not change over time. An interesting area for future work is to relax these assumptions.

In order to capture the varying degrees of locality between the nodes, we use a clustering-based network model. This is illustrated in Figure 4.1 which shows a set of cooperating nodes and a possible network-locality based clustering of these. This clustering is a natural consequence of how network topologies reflect organizational and geographic realities. For example, in a collection of universities, each node typically belongs to the department cluster which in turn belongs to the university cluster and so on. This cluster structure can be captured using a *cluster-tree* (or, a network-locality tree) as shown in the figure. The individual caches form the leaves of this tree, and the internal nodes correspond to the clusters. A cluster $\mathcal{C}$ is a child of cluster $\mathcal{C}'$ if $\mathcal{C}$ is immediately contained within $\mathcal{C}'$.

Because communication between two clusters is likely to traverse the same bottleneck link regardless of which particular nodes are conversing, we use a simple model of network distances: each cluster has an associated *diameter*, and the distance between any pair of nodes is given by the *diameter* of the smallest cluster that contains both of these nodes. This model is same as the ultrametric model used by Karger et al. in [33].

4.2.1 Non-cooperative algorithms

In this subsection, we outline four baseline algorithms that make all their placement or replacement decisions locally without consulting any other cache.

MFU placement. The cache looks at the local access frequencies to the various objects, and if the size of the cache is k, it stores the k most frequently used objects.

LRU replacement. When a cache miss occurs, this algorithm evicts the least recently used object.

LFU replacement. When a cache miss occurs, this algorithm evicts the object with the least (local) frequency of access.

Greedy-dual replacement. This is a generalization of the LRU algorithm to the case where each object has a different fetch cost [12, 61]. The motivation behind the greedy-dual algorithm is that the objects with larger fetch costs should stay in the cache for a longer time.

The algorithm maintains a *value* for each object that is currently in the cache. When an object is fetched into the cache, its value is set to its fetch cost. When a cache miss occurs, the object with the minimum value is evicted from the cache, and the values of all the other objects in the cache are reduced by this minimum value. And if an object in the cache is accessed (or 'touched'), then its value is restored to its fetch cost.

From an implementation point of view, it would be expensive to modify the value of each cache object, upon each cache miss. However, this expense can be avoided by noting that it is only the relative values, not the absolute ones, that matter [12]. In an efficient implementation, we use an additional variable – called *threshold* – to track the value of the object that was last removed from the cache. (The variable threshold is initially set to zero.) Upon a cache miss the minimum valued object is evicted from the cache; the variable threshold is set to the value of this object; and no other values are modified. However, when an object is touched or added, its value is set to its fetch cost plus the threshold.

Input:.

- Set of caches and the cache sizes.
- Set of objects.
- Access frequencies from each cache to each object.
  - Let $f(u, \psi)$ denote the frequency from node u to object ψ.
  - Let $f(\mathcal{C}, \psi) = \sum_{u \in \mathcal{C}} f(u, \psi)$, denote the aggregate frequency from cluster $\mathcal{C}$ to object ψ.
- Cluster tree T with diameters for each cluster.
  - Let $diam(\mathcal{C})$ denote the diameter for cluster $\mathcal{C}$.
  - Let $p(\mathcal{C})$ denote the parent of cluster $\mathcal{C}$ in T.
- A maximum cost called *penalty* which must be paid if no cache has the object.
  - Define $diam(p(root))$ to be *penalty*.

Output:.

- A placement of objects among the caches.
  - Represented as a set of items, where each item is a triple of the form $(objectId, cacheId, benefit)$.

Figure 4.2: Notation for the placement algorithms.

4.2.2 Cooperative placement algorithms

A *placement* assigns objects to caches without violating the cache size constraints. The cost of a placement is defined in the natural manner: the sum over all nodes u and all objects ψ of the access frequency for object ψ at node u times the distance from node u to the closest copy of that object. The goal of a cooperative placement algorithm is to compute a placement with minimum cost. Even though we do not explicitly minimize the network load and the server load these would typically be low when the access cost is minimized. This is because the latter objective would encourage objects to be stored closer to the clients, thereby reducing the load on both the network and the server.

We study three cooperative placement algorithms. One of them is provably optimal, but unfortunately it is impractical for scenarios with large numbers of nodes and objects. The other two algorithms are not provably optimal, but they are much simpler and can be implemented efficiently even in a distributed setting. Figure 4.2 presents the notation used for describing the placement algorithms.

4.2.2.1 An optimal placement algorithm

A centralized optimal algorithm for the placement problem was developed in Chapter 3 using a reduction to the minimum cost flow problem. The algorithm and its proof of optimality appear there, hence we do not reproduce it here. The instance of the minimum-cost flow problem constructed by this reduction has $\Theta(nm)$ vertices, where n is the number of nodes and m is the number of objects in the system.

Since the minimum cost flow problem is computationally intensive, this optimal algorithm incurs a high running time complexity. In particular, even the fastest known algorithm for minimum cost flow takes at least quadratic time and hence the running time of this optimal algorithm is at least quadratic in the product of n and m. Moreover, since the algorithm is centralized, it requires all the frequency information to be transferred to a single node, thereby imposing a high bandwidth requirement. These factors make this algorithm impractical for use with large inputs, and hence our sole use for this algorithm is as a benchmark for evaluating other algorithms.

4.2.2.2 The greedy placement algorithm

This algorithm follows a natural greedy improvement paradigm, and involves a bottom-up pass along the cluster-tree. It starts with a tentative placement in which each cache (i.e., a leaf in the cluster-tree) picks the locally most valuable set of objects. The algorithm then proceeds up the cluster-tree improving the placement iteratively. Figure 4.3 summarizes this algorithm by giving separate pseudocodes for the leaf and internal node computations.

In a general step, suppose we have computed the tentative placements for clusters $\mathcal{C}_1, \ldots \mathcal{C}_k$ which constitute a larger cluster $\mathcal{C}$. While computing the placement for cluster $\mathcal{C}_i$, the algorithm uses the access frequency information from within that cluster only.

At a leaf u

- If the size of cache at u is k, pick the k most frequently used objects (locally) at u. For each such object ψ, set the benefit b to $f(u, \psi) \cdot (\mathit{diam}(p(u)) - \mathit{diam}(u))$, and add the item (ψ, u, b) to the local placement Q.

At a cluster $\mathcal{C}$

1. **Merge.** Let Q_i be the placement computed for the ith child cluster of $\mathcal{C}$. (We assume that computations at the children of $\mathcal{C}$, if any, were already completed.) Initialize the placement Q to be the union of these Q_i's.
2. **Adjust benefits.** For each object ψ that has a copy in Q, pick its highest-benefit copy (breaking ties arbitrarily), and designate it as the *primary* copy for ψ. All other copies of ψ are *secondary*. Increase the benefit of the primary copy of ψ by $f(\mathcal{C}, \psi) \cdot (\mathit{diam}(p(\mathcal{C})) - \mathit{diam}(\mathcal{C}))$. The benefits of the secondary copies are not changed.
3. **Value missing objects.** For each Q-missing object ψ', set $\mathit{value}(\psi')$ to $f(\mathcal{C}, \psi') \cdot (\mathit{diam}(p(\mathcal{C})) - \mathit{diam}(\mathcal{C}))$. Let X be the set of all Q-missing objects.
4. **Greedy swap.** Pick the highest valued object ψ' from X, and the least benefit item y from Q. If $\mathit{value}(\psi')$ is greater than the $\mathit{benefit}(y)$ then swap ψ' for y and repeat step 4. The swap step involves removing the item y from Q, adding the item $(\psi', \mathit{cacheId}(y), \mathit{value}(\psi'))$ to Q, and then removing ψ' from X.
5. **Wind up.** If $\mathcal{C}$ is the root cluster, then return the placement Q. Otherwise proceed to the parent cluster of $\mathcal{C}$.

Figure 4.3: The greedy placement algorithm. (*Remark:* The algorithm here is slightly different from the corresponding greedy placement algorithm in Chapter 3. The algorithm there, like the one here, also makes a single bottom-up pass through the tree, but instead of directly computing a concrete placement with objects assigned to the leaves, it computes a more general (non-concrete) placement where objects might be assigned to cluster nodes as well. Then a separate top down pass is used to convert this non-concrete placement to a concrete placement. However, the notion of a non-concrete placement was needed only for the purpose of analysis and not for correctness. Hence here we give an equivalent one-pass description of those two passes combined, avoiding the notion of a non-concrete placement and highlighting the ease of implementation.)

Now at cluster $\mathcal{C}$, we first merge the tentative placements computed for subclusters $\mathcal{C}_1$ through $\mathcal{C}_k$. The placement Q obtained by this merging is clearly a starting placement for cluster $\mathcal{C}$, but it may be improved using the information about the aggregate frequencies across different subclusters in $\mathcal{C}$.

For example, there may be an object ψ that is not chosen in any of the clusters $\mathcal{C}_1$ through $\mathcal{C}_k$ since its access frequency within each cluster is small. But its aggregate frequency in the larger cluster $\mathcal{C}$ may be large enough that the placement Q can be improved by taking a copy of object ψ and dropping a less beneficial item – for example, an item which is replicated many times or whose aggregate access frequency within the cluster C is not as high. To determine such useful swaps, we calculate a *benefit* for each item in Q and a *value* for each object not in Q. (Such objects are said to be Q-*missing*.)

The value of a Q-*missing* object is $\mathcal{C}$'s aggregate access frequency for that object times the cost of leaving the cluster $\mathcal{C}$ to fetch that object. The latter quantity is the difference between the diameter of the parent cluster of $\mathcal{C}$ and that of the cluster $\mathcal{C}$ itself.

The benefit of an item x in placement Q, on the other hand, corresponds to the increase in the cost of the placement when the item x is dropped. Benefits are computed in a bottom-up manner. Each subcluster $\mathcal{C}_i$ calculates a local benefit for each item in its placement. After the merge step, the parent cluster $\mathcal{C}$ updates the benefits as follows. For each object that has one or more copies in Q, we pick the copy with the highest local benefit as the *primary* copy, and call all other copies as *secondary* copies. The benefits of the secondary copies are not changed, but the benefit of the primary copy is increased by $\mathcal{C}$'s aggregate access frequency to the object times the cost of leaving the cluster $\mathcal{C}$. The intuition is that among all the copies of an object, the primary copy will be the last one to be removed from Q, and its removal will increase the cost of the placement by the above amount.

Once the benefits and values have been computed, we use a simple greedy swapping phase to improve the placement Q. While there is a Q-missing object ψ whose value is more than the least beneficial item x in Q, we remove x from Q and substitute a copy of ψ. This phase terminates when the benefit of each item in Q is higher than the value of each Q-missing object.

This swapping phase concludes the computation for cluster $\mathcal{C}$, and the algorithm proceeds to the parent cluster of $\mathcal{C}$ iteratively.

Although this greedy algorithm looks simple and promising, it is shown in Chapter 3 that its worst-case performance can be arbitrarily far from the optimal. However, we conjecture that such worst-case examples occur rarely and that the algorithm would perform well in practice.

4.2.2.3 The amortizing placement algorithm

The worst-case analysis indicates that a drawback of the greedy algorithm is the following: A single secondary copy of some object may prevent the swapping in of several missing objects, one at each level of the cluster tree. Though the benefit of the secondary copy is larger than the value of each such missing object, on the whole it might be much less than the sum of all these values put together.

To circumvent this problem, the greedy algorithm is augmented with an amortization step using a potential function. The potential function accumulates the values of all the missing objects, and the accumulated potential is then used to reduce the benefits of certain secondary items thereby accelerating their removal from the placement. Figure 4.4 summarizes this improved algorithm.

It was proved in Chapter 3 that the above amortizing placement algorithm is always within a constant factor of the optimal. The constant factor is about 13.93. However, this factor is still large for practical purposes. We conjecture that, in practice, this algorithm will be much closer to the optimal.

Note that the above description of the greedy and the amortizing placement algorithms is still centralized. For example, all the computation at the last level of recursion is performed at the single node corresponding to the root of the network-locality tree. This induces a centralized bottleneck at this node. However, such bottlenecks can be avoided by mapping the computation of each internal node to the leaves in its subtree in a load-balanced manner. For details about such a scalable distributed implementation, please see Chapter 3.

At a leaf u

- Same as in the greedy algorithm, except that we also set the potential Φ to zero.

At a cluster $\mathcal{C}$

1. **Merge.** Same as in the greedy algorithm, except that we also initialize the potential Φ to the sum of the potentials $\Phi_1, \ldots \Phi_k$, computed by the children of u.

2. **Adjust benefits.** Same as in the greedy algorithm.

3. **Value missing objects.** Same as in the greedy algorithm, except that we also initialize Δ to the sum of the values of all the Q-missing objects.

4. **Amortized swap.** Similar to the greedy swap, except that the potential Φ is used to reduce the benefits of certain secondary items.

   (a) Pick the highest valued (Q-missing) object ψ' from X, and the least benefit primary and secondary items (y_p and y_s respectively) from Q.

   (b) If $\mathit{value}(\psi') > \mathit{min}(\mathit{benefit}(y_p), \mathit{benefit}(y_s) - \Phi)$, then depending on which of the latter two terms is smaller perform one of the following two swap operations and then repeat step (4).

      - If $\mathit{benefit}(y_p) < \mathit{benefit}(y_s) - \Phi$, then swap ψ' for y_p and adjust Δ. The swap step involves removing the item y_p from Q, adding the item $(\psi', \mathit{cacheId}(y_p), \mathit{value}(\psi'))$ to Q, and then removing ψ' from X. The variable Δ is set to $\Delta - \mathit{value}(\psi') + \mathit{benefit}(y_p)$.
      - Otherwise, swap ψ' for y_s, reduce the potential Φ, and adjust variable Δ. As before, the swap step involves removing the item y_s from Q, adding the item $(\psi', \mathit{cacheId}(y_s), \mathit{value}(\psi'))$ to Q, and then removing ψ' from X. The potential Φ is set to $\max(0, \Phi - \mathit{benefit}(y_s))$ while the variable Δ is set to $\Delta - \mathit{value}(\psi')$.

5. **Update potential.** Add Δ to Φ.

6. **Wind up.** If $\mathcal{C}$ is the root cluster, then return the placement Q. Otherwise proceed to the parent cluster of $\mathcal{C}$.

Figure 4.4: The amortizing placement algorithm. (*Remark:* This algorithm too is slightly different from the corresponding amortizing placement algorithm defined in Chapter 3. However, the difference is exactly the same as in the case of the greedy placement algorithm. We refer the reader to Figure 4.3 for a note on this.)

4.2.3 A cooperative replacement algorithm

Our experiments in Section 4.3 show that, in a cooperative scenario, the greedy-dual algorithm performs much better than the other local replacement algorithms. This is because even though the greedy-dual algorithm makes entirely local decisions, its value structure enables some implicit coordination with other caches. In particular, an object that is fetched from a nearby cache has a smaller value than an object that is fetched from afar. Hence the former object would be evicted from the cache first, thus reducing unnecessary replication among nearby caches. However, this limited degree of coordination does not exploit all the benefits of cooperation. For example, the idle caches are not exploited by nearby busy caches.

Hence we devise *hierarchical-greedy-dual*, a cooperative replacement algorithm that not only preserves the implicit coordination offered by greedy-dual but also enables busy caches to utilize nearby idle caches.

In this algorithm, each individual cache runs the local greedy-dual algorithm using the efficient implementation described in Section 4.2.1. Recall that the local greedy-dual algorithm maintains a *value* for each object in the cache, and upon a cache miss, it evicts the object with the minimum value. In our hierarchical generalization, the evicted object is then "passed up" to the parent cluster for possible inclusion in one of its caches. When a cluster $\mathcal{C}$ receives an evicted object ψ from one of its child clusters, it first checks to see if there is any other copy of ψ among its caches. If not, it picks the minimum valued object ψ' among all the objects cached in $\mathcal{C}$. Then the following simple admission control test is used to determine if ψ should replace ψ'. If the copy of ψ was used more recently than the copy of ψ', then ψ replaces ψ' and the new evicted object ψ' is recursively passed on to the parent cluster of $\mathcal{C}$. Otherwise, the object ψ itself is recursively passed on to the parent cluster of $\mathcal{C}$.

From our experiments, we learned that the particular admission control test described above is crucial for obtaining good performance. This is because an important purpose of the admission control test is to prevent rarely-accessed objects from jumping from cache to cache without ever leaving the system. Such objects typically have a high fetch cost since no other (nearby) cache stores them, and hence any fetch-cost based admission control test would repeatedly reinsert such objects even after they are evicted by individual caches. This can result in worse performance than even the local greedy-dual algorithm. We avoid this problem by maintaining a *last-use* timestamp on every object in the cache. (The timestamp is updated whenever the copy is accessed, either by the local client or by a remote client.) With this timestamp based admission control strategy, rarely used objects are eventually released from the system.

For a practical implementation, algorithm would use data-location directories [24, 26, 51, 55] to determine if other copies exist in the subtree, and would use randomized [21] or deterministic [25] strategies to approximate the selection of ψ'.

4.3 Performance evaluation on synthetic workloads

This section explores the performance of the above algorithms under a range of synthetic workloads. These workloads allow us to explore a broader range of system behavior than trace workloads. In addition, because the synthetic workloads are small enough be tractable under the optimal algorithm, we can compare our algorithms to the optimal placement.

This section first describes our methodology in detail and then shows the results of our experiments. These results support the first four conclusions listed in Section 4.1. For the sake of brevity and for ease of reference, throughout the rest of this chapter, we use the phrase *GreedyPlace* (resp., *AmortPlace*, *MFUPlace*, *GreedyDual*, *HrcGreedy-Dual*) as a shorthand for the greedy placement (resp., amortizing placement, MFU placement, greedy-dual, hierarchical-greedy-dual) algorithm.

4.3.1 Methodology

We simulate a collection of caches that include a directory system, such as the ones provided by Hint Cache [55], Summary Cache [24], Cache Digests [51] or CRISP [26], so that caches can send each local miss directly to the nearest cache or server that has the data. For the placement algorithms MFUPlace, GreedyPlace, AmortPlace, and Optimal, we compute the initial data placement according to the algorithm under simulation, and the data remain in their initial caches throughout the run. For the replacement algorithms LFU, LRU, GreedyDual, and HrcGreedyDual, we begin with empty caches, and for each request we modify the cache contents as dictated by the replacement algorithm. In that case, we use an initial warm-up phase to prime the caches before gathering statistics.

We parameterize the network architecture and workload along a number of axes. The parameters are defined in detail in the following two subsections. Table 4.1 summarizes the default values for these parameters.

Parameter	Meaning	Default Value
L	Number of levels	3
D	Degree of each internal node	3
λ	Diameter growth factor	4
C	Cache size percentage	20% (synthetic only)
m	No. of local objects per node	25 (synthetic only)
r	Sharing parameter	0.75 (synthetic only)
PAT	Access pattern	Uniform (synthetic only)
I	Idle cache factor	1

Table 4.1: Default system parameters.

4.3.1.1 Network architecture

Recall from Section 4.2 that the distances between the cache nodes are completely specified once the network-locality tree and the cluster diameters are given. We create an L-level network-locality tree with the degree of each internal node being D. The root is considered to be at level L and the leaves are at level zero. The cluster diameters are captured by λ, the diameter growth factor. The diameter for a cluster at level i is λ^i, and the cost of leaving the root cluster is λ^{L+1}.

Because all objects have the same size, it suffices to express the size of a cache in terms of the number of objects it can hold. We set all cache sizes to be the same, using a single parameter C which is called the cache size percentage. Specifically, the cache size at a node is set to $CM^*/100$, where M^* is the average number of objects accessed by the node.

4.3.1.2 Workload

As observed in Section 4.1, an important parameter for the performance of cooperative strategies is the degree of similarity of interests among nearby nodes. At one extreme, there is total similarity (all nodes access the same set of shared objects with the same frequencies) while at the other extreme there is absolutely no similarity (each node accesses its own set of local objects).

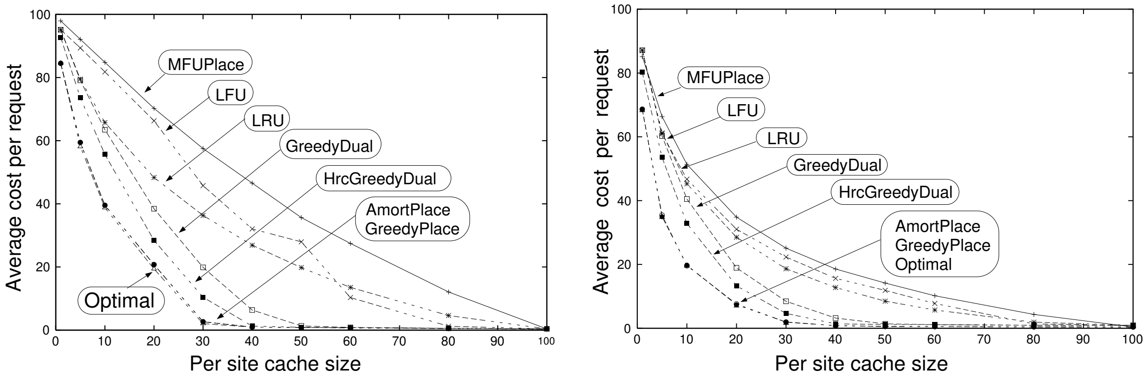
Our synthetic workload models such sharing by creating m objects for each cluster in the network. This pattern could represent a hierarchical organization where some objects are local to an individual, some to a group, some to a department, and some of organization-wide interest. The sharing parameter, r, determines the mix of requests to the "private", "group," "department," and "organization" collections. The fraction of requests that a client sends to level-i objects is proportional to r^i. Note that as r varies from 0 to infinity, the degree of sharing increases: when $r < 1$, clients are more likely to access "local" objects, when $r = 1$ they are equally likely to access objects from all levels, and when $r > 1$ they focus much of their attention on "global" objects.

Within each cluster, we select objects according to a pattern PAT that is either "Zipf-like" or "Uniform." Thus, for a particular cache v and a particular object ψ that is local to cluster $\mathcal{C}$ and that is the kth-ranked object of the m objects local to $\mathcal{C}$, the fraction of node v's requests that go to object ψ, is computed as follows:

$$F_v(\psi) = \begin{cases} 0 & \text{if } \mathcal{C} \text{ does not contain } v \\ ar^i & \text{if } \mathcal{C} \text{ contains } v \text{ and } PAT \text{ is ``Uniform''} \\ \frac{ar^i}{k} & \text{if } \mathcal{C} \text{ contains } v \text{ and } PAT \text{ is ``Zipf-like''} \end{cases}$$

for an appropriate normalization constant a.

The above workloads ensure that all clients are almost equally active. However in reality, there may be several caches that are idle for periods of time. We model this effect by using another parameter I (called the idle cache factor) and by adding a special cache called the idle cache for each level-one cluster. The idle cache makes no access requests at all, but it has a cache of size I times that of any other cache. As I is increased from 0 upwards, the amount of idle cache space in the system increases.



(a) *With PAT = "Uniform"* (b) *With PAT = "Zipf-like"*

Figure 4.5: *Varying cache size*

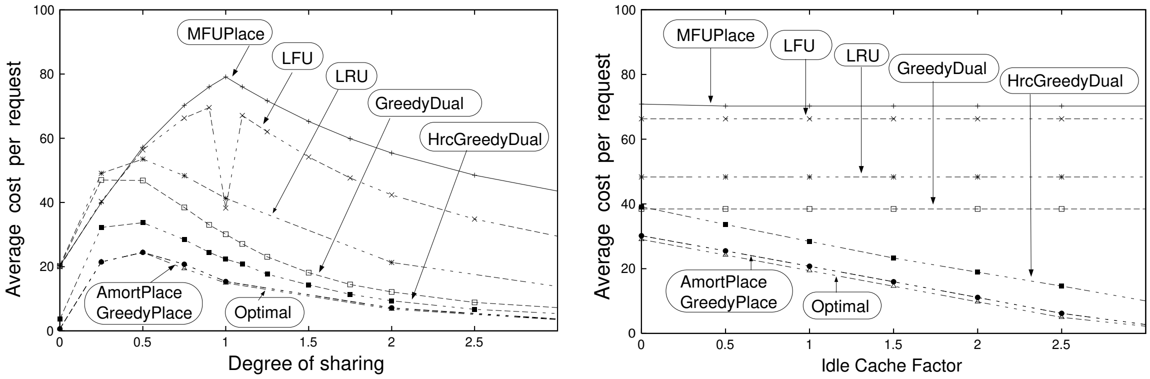
4.3.2 Results

Figure 4.5 plots the performance of the algorithms as the cache size percentage C is varied from 1 to 100, with other parameters set to their default values.

The y-axis corresponds to the average cost per request, as a percentage of the base cost. The latter is the cost that is paid if there are no copies of the object in the hierarchy, and is given by the expression λ^{L+1}. The results for the case where the pattern within each category is Zipf-like are similar and are presented in Figure 4.5-b.

The primary conclusion from this data is that increasing coordination can improve performance, particularly with small caches. When comparing the three categories of algorithms—local (MFUPlace, LFU, LRU, GreedyDual), cooperative replacement (HrcGreedyDual), and cooperative placement (AmortPlace, GreedyPlace, Optimal)—cooperative replacement generally outperforms local, and cooperative placement generally outperforms cooperative replacement.

Within each category, the effect of increasing coordination can also be seen. Although MFUPlace, LFU, LRU, and GreedyDual are all "local" algorithms, their performance differs markedly. MFUPlace performs poorly because caches tend to contain exactly the same objects from the cluster, which wastes cache space with inefficient replication. LFU and LRU do somewhat better because randomization has an effect similar to coordination—reducing the replication of the most frequently accessed objects while increasing the replication of less frequently accessed ones. Finally, the miss-cost



(a) Varying degree of sharing *(b) Varying idle cache per level-one cluster*

Figure 4.6: *Varying degree of sharing and idle cache factor*

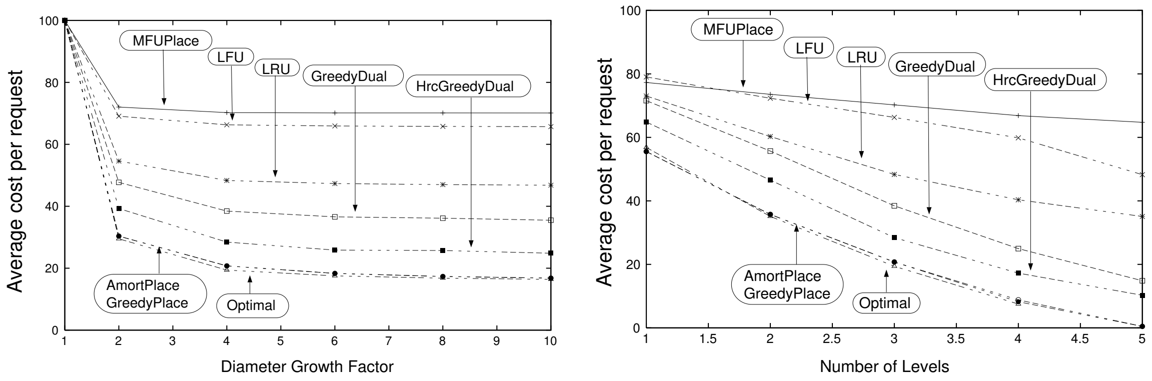
consideration in GreedyDual makes it expensive to throw away objects that are not cached by nearby neighbors, which induces significant coordination across caches. In fact, for the "no idle cache" case, GreedyDual matches the performance of HrcGreedyDual (Figure 4.6-b). However, when there is idle cache space to exploit, HrcGreedyDual outperforms GreedyDual as Figure 4.5-b shows.

As we increase cache size, the performance of all these algorithms improves. None of the algorithms perform well when caches are tiny, but for small to medium sized caches, the coordinated algorithms significantly outperform traditional replacement algorithms.

We also note that the amortizing and the greedy placement algorithms effectively match optimal across a wide range of workloads.

4.3.2.1 Sensitivity to other parameters

Figure 4.6-a shows performance as we vary the sharing parameter, r. When r is between 0 and 1, smaller values have better performance for all of the algorithms because smaller values result in clients sending more requests to their "local" collection of objects, of which a large fraction will fit in their local caches under any of the algorithms studied. When $r > 1$, increasing r actually helps performance because sharing among caches becomes more effective. The performance spike for LFU at $r = 1$ occurs because a client is spreading its requests across all levels of objects evenly and it considers all objects equally likely to be referenced; all replacement decisions are ties and are broken



(a) Diameter growth factor (b) Number of levels

Figure 4.7: *Varying diameter growth factor and number of levels*

randomly, which results in most objects being widely cached.

We also note that the same general patterns emerge as for the earlier experiment: across a wide range of sharing factors, algorithms with more coordination have better performance and GreedyPlace and AmortPlace closely track the performance of Optimal.

Figure 4.6-b shows what happens as the amount of idle cache in each level-one cluster is increased. The "implicit" coordination of LRU and GreedyDual is not able to take advantage of the increasing idle cache space. On the other hand, the performance of the explicitly coordinated algorithms—HrcGreedyDual, AmortPlace, GreedyPlace, and Optimal—improves.

Figure 4.7-a shows the impact of varying the diameter growth factor, λ. For small λ, all algorithms have approximately the same performance since all cache hits and misses have more or less the same cost. For large values of λ, performance is dominated by the number of misses and hence the coordinated algorithms perform well. The lines are flat for large λ because neither the number of objects nor the total cache space changes with λ, and so so the fraction of objects that are not stored in the cache system is constant. The average access cost, which is largely dominated by the accesses to these objects, increases at the same rate as the base cost. Therefore the average access cost as a percentage of the base cost is almost constant.

Figure 4.7-b shows the performance as the number of levels in the hierarchy, L, is

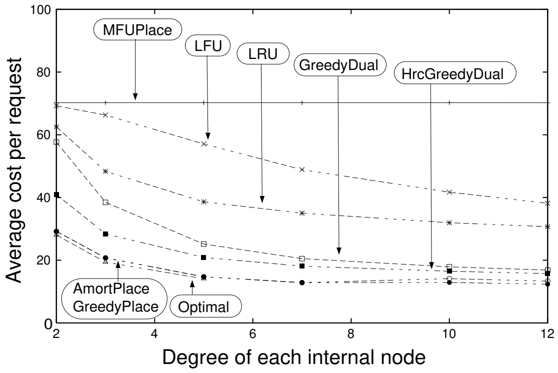


Figure 4.8: *Varying the degree of each node*

varied. For our workloads, increasing L increases the total cache space in the system more quickly than it increases the total number of objects. (To be precise, note that (1) the number of leaves in the tree, call it n, is D^L; (2) the total number of nodes in the tree is at most $2n$; and (3) each leaf accesses $(L+1)m$ objects and hence has a cache of size $C(L+1)m/100$. Therefore the total cache space in the tree is $nC(L+1)m/100$ while the total number of objects is at most $2nm$.) Hence more objects can be cached in the system, thereby reducing the number of accesses that need to pay the base cost. Hence the average access cost as a percentage of the base cost also reduces.

Finally, Figure 4.8 shows the performance as the degree of each internal node, D, is increased. Again, for our synthetic workloads, increasing D increases the total number of nodes in the system. However, the ratio of the total number of objects to the total available cache space (which is $\frac{2nm}{nC(L+1)m/100}$) does not change even as D changes. Therefore the fraction of accesses that have to pay the base cost is almost fixed, and hence there is not much variation in the average access cost when it is expressed as a percentage of the base cost.

On the whole, the basic trends in all of these experiments are similar to those described earlier in Section 4.3.2: Algorithms with more coordination have a better performance and, moreover, the greedy and the amortizing placement algorithms closely match the optimal in their performance.

4.4 Performance evaluation on web-trace workloads

In this section our goal is to evaluate the performance of the various placement and replacement algorithms on trace workloads. The main conclusions for the synthetic workload are also supported here. In addition, we find that a key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses. As a result, it appears that hybrid placement-replacement algorithms may offer the best option.

4.4.1 Methodology

Our simulations use the Digital web proxy trace [20], which was collected at a proxy serving about $16,000$ clients over a period of 25 days from August 29th, 1996 to September 22nd, 1996. About 24 million events were logged using about 4.15 million distinct URLs. For our simulations we use only the first one week of the trace and focus only on cacheable read accesses (i.e., events with *GET* method, without involving CGI scripts, et cetera).

Because the trace does not provide any information regarding the architecture of the network connecting the clients, we use our standard synthetic architecture which was described in Section 4.3.1. This synthetic network has 27 nodes (i.e., the leaves of the network-locality tree), and we map each of the $16,000$ trace clients randomly on to one of these 27 nodes.

We believe that such a random mapping will generally inhibit the performance of cooperative algorithms for two reasons. Firstly, random mapping of clients to nodes will eliminate any similarity of interests among nearby clients. Secondly, with about 600 random nodes multiplexed to each node, cache load is evenly balanced and hence there would be few chances to exploit idle cache memory.

The large number of objects in the trace makes it infeasible to run the optimal placement algorithm. Nevertheless, as seen in the previous section, we can obtain almost optimal placements by using the simpler greedy placement algorithm. Hence we focus on the greedy placement algorithm alone.

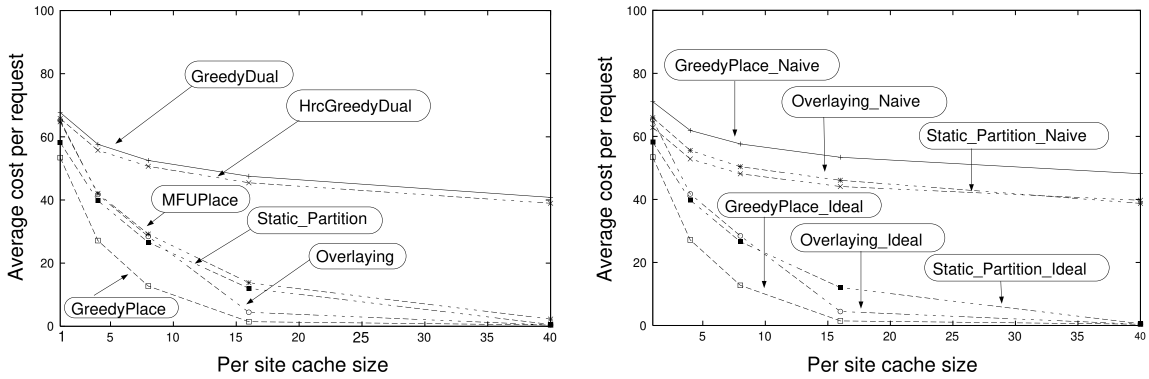
4.4.2 Design issues

We have seen that the placement algorithms yield significant performance gains when the access patterns are stable and known, as was the case for the synthetic workloads. However, in reality access patterns change over time, and an effective placement strategy must be able to cope with these changes. A natural way of coping with dynamically-changing access patterns is to run the placement algorithms at regular intervals to reorganize the data more effectively. However, there are two crucial factors that affect the performance of such a strategy: (1) How frequently should the placement algorithms be run? and (2) How do we predict the access frequencies for use by the placement algorithms?

The dynamic versions of our placement algorithms break the time into *epochs* and run the placement algorithm at the beginning of every epoch. If the epoch size were too large, then the placement would get outdated and hence yield bad performance. On the other hand, if the epoch size were too small then the bandwidth cost of reorganizing the data would be prohibitive. For our experiments, we set the default epoch size to 6 hours.

A key challenge for placement algorithms is to predict the future access frequencies based on past accesses. Ideally, a sophisticated prediction technique would exploit the temporal, spatial, and geographical localities among requests to predict future requests. (Spatial locality refers to the fact that related objects such as objects from the same server or that are hyperlinked to each other tend to be accessed together. Geographical locality refers to the fact that clients that are close to each other may have similar interests.) However a study of these techniques is orthogonal to our current focus, and we do not delve into this question.

For the purposes of evaluating the placement algorithms, we consider two extreme prediction strategies. The first one is an idealized predictor, based on future knowledge, that looks ahead into the next epoch to determine the access frequencies for each $\langle$client, object$\rangle$ pair. This unrealizable algorithm serves as a benchmark for the best any prediction technique can achieve. The second one is a naive predictor that computes the predicted access counts for the next epoch using the access counts from the ear-



(a) With ideal prediction *(b) Effect of prediction*

Figure 4.9: *DEC proxy trace: Varying cache size*

lier epochs, along with a damping factor r: if the access count for a particular $\langle$client, object$\rangle$ pair was c_i during the ith preceeding epoch, then that epoch contributes $c_i \cdot r^{i-1}$ to the predicted access count for the coming epoch.

Finally, to cope with the dynamic access patterns in these traces, we examine the performance of hybrid placement-replacement algorithms. These hybrid algorithms run a placement algorithm at epoch boundaries and also run a replacement algorithm during the epoch. We examine two hybridization techniques. *Static partition* divides the cache space into two portions and runs the placement algorithm on one portion and the dynamic replacement algorithm on the other. In our experiments, we use half of the cache for each partition. The second technique, *overlaying*, reorganizes the entire cache using the placement algorithm at the start of each epoch and then gives the replacement algorithm control of the entire cache during the epoch.

4.4.3 Results

Figure 4.9-a shows the performance of the algorithms under the ideal prediction strategy described above. The x-axis shows the per-node cache size, as a percentage of the number of objects accessed by the node, and is varied from 1 to 40. All the other

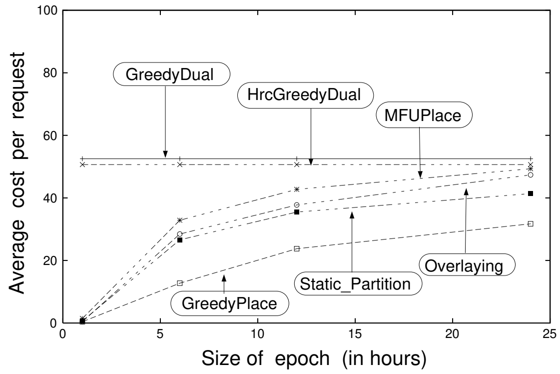


Figure 4.10: *DEC proxy trace: Varying epoch size.*

parameters are set to their default values.

The experiments show that when the predictions are ideal, the placement algorithms perform significantly better than the replacement algorithms. This is particularly encouraging for systems that can provide good predictions of access patterns (e.g., subscription based systems). The effect of increasing coordination can also be seen: the hierarchical-greedy-dual outperforms the greedy-dual algorithm while the greedy placement algorithm performs better than the local-MFU placement algorithm. However hybridization hurts the performance of the placement algorithms when the predictions are good.

Figure 4.9-b shows the effect of prediction on the performance of the placement algorithms by comparing these algorithms with naive prediction at one extreme and ideal prediction at the other. With naive predictions, the most effective algorithms are the hybrid combinations of placement and replacement. Even with these combinations, the performance gains are modest for this set of parameters under the naive predictor.

This suggests that developing more accurate frequency predictors could be a fertile area for future work. The reason for this optimism is that our current naive predictor uses only one type of locality, namely the temporal locality, out of the three localities expected in web accesses.

Finally, Figure 4.10 shows the effect of varying epoch size on the performance of the algorithms. The x-axis shows the epoch size in hours while the y-axis shows the average access cost as before. (The cache size at each node is set to 8%.) As expected, performance is excellent when the epochs are short. As epochs become longer the placement becomes coarse grained and the performance gains are more modest. However, even with only daily reorganization significant performance gains appear possible.

4.5 Related work

A number of recent studies have examined the question of what to store in caches. There are several studies and prototypes (e.g., [13, 12, 50, 59]) that employ purely local replacement strategies such as LRU or greedy-dual at each cache. The greedy-dual local replacement algorithm was evaluated in [12, 31, 61], but for single stand-alone caches only. Here we study their performance in a cooperative caching scenario.

The placement and replacement algorithms for local-area networks were studied by Leff et al. [40], Dahlin et al. [21], and Feeley et al. [25]. However the scenario of wide-area networks is vastly different and relatively unexplored. The contributing factors are that the number of users as well as the number of objects are larger by orders of magnitude, and that the network distances between one pair of nodes can be much different from that between another pair.

Recently, Yu and MacNair [62] studied the question of wide-area cache coordination, but under a simplistic model where all the network distances are assumed to be the same. In such a scenario, clearly the best strategy is simply to avoid duplication altogether. Here, we study a more general problem with non-uniform network distances.

The issue of server-initiated on-line replication has received a good deal of attention [29, 33, 47, 49, 60] recently. Two of these [33, 47] give theoretical results while the remaining three [29, 49, 60] present heuristics with empirical evaluation. In all of these studies, the concern is more with the issue of reducing server load when hot-spots occur (i.e., when the load on server increases) and less with the issue of reducing the latency when there are no hot spots.

Another useful technique for reducing the client latency is that of push caching [28, 55]. Here the server keeps track of client access patterns and pushes data towards the clients before they ask for it, thus avoiding the compulsory misses. Such schemes involve two orthogonal components: predicting future access patterns and distributing the data according to these predictions. The studies in [28, 55] aim at evaluating the potential benefits of push caching by focusing on the first component and assuming that the cache sizes are infinite. Our study of placement problem address the second component of push caching, under the more realistic assumption that the cache sizes are bounded. The placement algorithms proposed here are ideal for systems, such as subscription-based services, where good predictions of access patterns are available.

Chapter 5

Conclusions and Future Directions

The grand vision of the current information revolution is to enable a multitude of users at any place to access information from anywhere at any time. To realise such a vision, it is essential to have automatic techniques for organizing and retrieving data (text, video, audio) correctly and efficiently. Such techniques have been studied extensively for traditional small-scale systems, but the wide-area and wireless networks are vastly different, posing new challenges and demanding new tradeoffs. The key distinguishing features are that the number of users is larger by orders of magnitude, network bandwidth is limited, failures are common, and nodes have varying computing power and varying degrees of trust. In such a setting, it is an exciting challenge to identify the key problems and to design efficient and practical algorithms for them.

This dissertation takes a small step in this direction, by exploring the technique of cooperative caching in wide-area networks. It provides basic placement and replacement algorithms – using which the cooperating caches can coordinate their storage decisions – and analyzes them both theoretically as well as experimentally.

A unifying theme throughout this dissertation is the emphasis on obtaining simple algorithms that achieve optimal or near-optimal aymptotic performance, since such algorithms are the ones that are more likely to be useful in practice. For instance, for the facility location problems, we study and analyze a simple local search heuristic, and for the hierarchical placement problem, we focus on algorithms that process the network in a simple bottom-up fashion.

We made several simplifying assumptions in order to focus on the basic problem.

One assumption was that all objects are unit-sized and read-only. Second, we assumed that the network distances, or communication costs, between node pairs are fixed and do not change over time. An interesting area for future work is to relax these assumptions.

In addition, there are several key data management problems, arising from the Internet and wireless network domains, that are worth exploring. We conclude by outlining a few specific examples of such problems.

- *Caching with bandwidth constraints.* The caching problems studied in this dissertation imposed only a space constraint on each cache. It would be interesting to enforce a bandwidth constraint on each cache in addition to the space constraint. Such bandwidth constraints are especially important for applications involving continuous media objects such as audio and video streams.

- *Facility location.* In Chapter 2, we proved that a simple local search heuristic yields near-optimal solutions for several facility location problems. Though this local search heuristic is simple, it is not the simplest heuristic one could imagine. It would be interesting to analyze simpler heuristics, for example, fixing one facility at a time according to a *suitable* greedy criterion. Such a greedy heuristic would not only be simpler than local search, but would also be much faster.

- *Web server selection.* Consider a scenario where a set of geographically distributed clients are issuing requests for web objects. Typically there would be multiple web servers that are capable of delivering each object. In such a scenario, to which server should a particular request be directed? Directing each client to its closest server might cause adverse load imbalances. On the other hand, directing to a lightly loaded but distant server, may increase the latency and the network load. A further challenge in practice is that the load information, if maintained by the clients, might become stale over time. In addition, the requests may have real-time requirements. A good example is the scenario of a set of geographically distributed video-on-demand servers serving a set of geographically distributed clients.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Rosselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, 1995.

[3] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for Euclidean k-medians and related problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 106–113, May 1998.

[4] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 164–173, May 1993.

[5] B. Awerbuch, Y. Bartal, and A. Fiat. Heat & Dump: Competitive distributed paging. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31, November 1993.

[6] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 37:1021–1058, 1995.

[7] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, October 1996.

[8] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 161–168, May 1998.

[9] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 39–50, May 1992.

[10] M. A. Blaze. Caching in large-scale distributed file systems. Technical Report TR–397–92, Department of Computer Science, Princeton University, January 1993. PhD Thesis.

[11] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The Harvest information discovery and access system. In *Proceedings of the 2nd Intl. World Wide Web Conference*, pages 763–771, October 1994.

[12] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[13] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the USENIX Technical Conference*, pages 22–26, January 1996.

[14] M. Charikar, C. Chekuri, A. Goel, and S. Guha. Rounding via trees: Deterministic approximation algorithms for group Steiner trees and k-median. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 106–113, May 1998.

[15] M. Charikar, S. Guha, D. Shmoys, and É. Tardos. A constant-factor approximation algorithm for the k-median problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 1–10, May 1999.

[16] F. A. Chudak. Improved approximation algorithms for the uncapacitated facility location problem. In *Proceedings of the 6th Conference on Integer Programming and Combinatorial Optimization*, pages 180–194, June 1998.

[17] F. A. Chudak and D. Shmoys. Improved approximation algorithms for a capac-

itated facility location problem. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages S875–S876, January 1999.

[18] F. A. Chudak and D. Williamson. Improved approximation algorithms for capacitated facility location problems. In *Proceedings of the 7th Conference on Integer Programming and Combinatorial Optimization*, June 1999. To appear.

[19] G. Cornuéjols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. In P. Mirchandani and R. Francis, editors, *Discrete Location Theory*, pages 119–171. Wiley, New York, NY, 1990.

[20] Digital Equipment Corporation. Web proxy traces, september, 1996. Available from ftp://ftp.digital.com/pub/DEC/traces/proxy.webtraces.html, September 1996.

[21] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[22] G. Diehr. An algorithm for the p-median problem. Technical Report No. 191, Western Management Science Institute, UCLA, June 1972.

[23] D. Dowdy and D. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14:287–313, 1982.

[24] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the 1998 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 254–265, August 1998.

[25] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[26] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1997. http://www.cs.duke.edu/ari/cisi/crisp-recycle.ps.

[27] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 649–657, January 1998.

[28] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 51–57, May 1995.

[29] A. Heddaya and S. Mirdad. WebWave: Globally load balanced fully distributed caching of hot published documents. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 160–168, May 1997.

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

[31] S. Irani. Page replacement with multi-size pages and applications to Web caching. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 701–710, May 1997.

[32] K. Jain and V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. Manuscript, April 1999.

[33] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[34] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[35] L. G. Khachiyan. A polynomial algorithm for linear programming. *Soviet Math. Doklady*, 20:191–194, 1979.

[36] M. R. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. In *Proceedings of the IEEE Workshop on Internet Applications*, pages 62–71, July 1999. Full version appears as Technical Report TR–98–30, Department of Computer Science, University of Texas at Austin, December, 1998.

[37] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, January 1998. Full version appears as Technical Report TR–98–30, DIMACS, Rutgers University, June 1998. Invited to appear in the special issue of *Journal of Algorithms* devoted to selected papers from SODA 1998.

[38] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 586–595, January 1999. Full version appears as Technical Report TR–99–16, Department of Computer Science, University of Texas at Austin, June 1999. Invited to appear in the special issue of *Journal of Algorithms* devoted to selected papers from SODA 1999.

[39] A. A. Kuehn and M. J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9:643–666, 1963.

[40] A. Leff, J. L. Wolf, and P. S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204, 1993.

[41] J.-H. Lin and J. S. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44:245–249, 1992.

[42] J.-H. Lin and J. S. Vitter. ε-approximations with minimum packing constraint violation. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 771–782, May 1992.

[43] R. F. Love, J. G. Morris, and G. O. Wesolowsky. *Facilities Location: Models and Methods.* North-Holland, New York, NY, 1988.

[44] C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-line distributed data management. In J. van Leeuwen, editor, *Proceedings of the 2nd Annual European Symposium on Algorithms,* Lecture Notes in Computer Science, volume 855, pages 202–214. Springer-Verlag, September 1994.

[45] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 284–293, October 1997.

[46] P. Mirchandani and R. Francis. *Discrete Location Theory.* Wiley, New York, NY, 1990.

[47] C. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 570–579, October 1996.

[48] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[49] M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic replication on the Internet. Technical Report HA6177000–980305–01-TM, AT&T Labs – Research, April 1998.

[50] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. Technical Report RN–98–13, Department of Computer Science, University College, London, UK, 1998.

[51] A. Rousskov and D. Wessels. Cache digests. In *Proceedings of the 3rd International Web Caching Workshop*, June 1998. http://wwwcache.ja.net/events/workshop/.

[52] D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

[53] D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 265–274, May 1997.

[54] M. B. Teitz and P. Bart. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research*, 16:955–961, 1968.

[55] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, June 1999. To appear.

[56] M. van Steen, F. J. Hauck, and A. S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proceedings of the 1996 Conference on Telecommunications Information Networking Architecture (TINA 96)*, pages 203–212, September 1996.

[57] D. Wessels. Squid internet object cache. http://squid.nlanr.net/Squid, January 1998.

[58] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2, request for comments rfc–2187. http://ds.internic.net/rfc/rfc2186.txt, September 1997.

[59] S. Williams, M. Abrams, C. R. Standbridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for world wide web documents. In *Proceedings of the ACM-SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1996.

[60] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(4):255–314, 1997.

[61] N. E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, January 1998.

[62] P. S. Yu and E. A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. In *Proceedings of 7th International World Wide Web Conference*, Brisbane, Australia, April 1998.

[63] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the NLANR Web Cache Workshop*, June 1997.

Vita

Madhukar R. Korupolu was born in Hyderabad, India on December 27, 1972, the son of Bharathi Korupolu and Ananth Reddy Korupolu. He completed his schooling from Hyderabad, India, in June 1990. Thereafter, in August 1994, he received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology, Madras, India. After that, he joined the graduate program in Computer Sciences at the University of Texas at Austin, and received the Master of Sciences degree in December 1996. He was employed as a research scientist at Fujitsu Laboratories, California, and at Bell Laboratories, New Jersey, during the summers of 1997 and 1998 respectively. He was awarded the MCD graduate fellowship, in August 1994, by the University of Texas at Austin. Earlier, during his high school studies in April 1990, he secured the fourth rank in the Indian National Mathematics Olympiad.

Permanent Address: c/o Ananth Reddy Korupolu
17-1-391/T/40, Saraswathi Nagar
Saidabad, Hyderabad - 500659
India

This dissertation was typeset with LaTeX 2ε[1] by the author.

[1] LaTeX 2ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.