This is to certify that the
thesis entitled

TYPED CACHING AND SERVER PROBLEMS

presented by

Martin Douglas Porcelli

has been accepted towards fulfillment
of the requirements for the

M.S. degree in Computer Science

Major Professor's Signature

July 18, 2007

Date

# TYPED CACHING AND SERVER PROBLEMS

By

Martin Douglas Porcelli

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

2007

ABSTRACT

TYPED CACHING AND SERVER PROBLEMS

By

Martin Douglas Porcelli

The $k$-server problem is a resource allocation problem in which $k$ servers living on some metric space must move in order to satisfy a sequence of requests. Each request specifies some point in the metric space and is satisfied by moving one of the servers to that point. An algorithm controls the movement of the servers, and its goal is to miminize the total distance they travel. The caching problem is a special case of the $k$-server problem.

In the typed $k$-server problem a request also specifies a set of servers called a type. This set contains the servers that are allowed to satisfy the request. Until recently no one has studied the typed $k$-server problem. In this paper we study both the typed $k$-server problem and a typed version of the caching problem. We introduce a set of algorithms for the typed caching problem, called EP algorithms, that utilize reorganization. We give a lower bound on the competitive ratio of these algorithms and an upper bound on the competitive ratio of marking EP algorithms. We also give a lower bound on the competitive ratio of any deterministic algorithm for the typed $k$-server problem, and discuss a work function approach for solving it. Finally we show that parts of the proof that the work function algorithm is $(2k-1)$-competitive [KP95] do not apply in the typed $k$-server problem.

## TABLE OF CONTENTS

## LIST OF FIGURES

# Chapter 1

# Typed Caching

## 1.1 Introduction

Caches are an important part of modern computers. A cache is a small block of fast memory that sits between the processor and main memory. When a memory page is requested by the processor, the system first checks to see if the page is currently in the cache. If the page is not present, a copy is put into the cache. Since the processor is much faster than main memory, it must normally wait a (relatively) long time for a requested page. With a cache, this wait is reduced for many requests.

## 1.2 Fully-Associative Caching

Most studies of caching have focused on fully-associative caches. In a fully-associative cache every page may be placed into every slot. This simplifies management of the cache. Page location is unimportant; all that matters is whether or not a page is in the cache.

Set-associative caches are the simplest generalization of fully-associative caches. In an $m$-way set associative cache, the memory pages and slots are divided into $m$ equal sized sets, call them $P_1, P_2, \ldots, P_m$ and $S_1, S_2, \ldots, S_m$ respectively. The memory page from any $P_i$ can only be placed into a slot from $S_i$. In this way the cache is actually a collection of $m$ independent fully-associative caches. A $k$-way set-associative cache where $k$ is the number of slots is called a direct mapped cache.

## 1.3 Typed Caching

We focus on more complex caches called typed or restricted caches. A typed cache is any cache that cannot be reduced to a collection of independent fully-associative

caches.

Previous work on the typed caching problem has focused on so called skew-associative caches and companion caches [Sez93, BETW03]. In an $m$-way skew associative cache there are overlapping sets of size $k/m$. Every memory page is associated with exactly one of these sets, and a memory page can only be placed into a slot in its associated set. An $(m, n)$-companion cache pairs a direct mapped main cache with $m$ slots and a fully-associative companion cache with $n$ slots. Victim caches are examples of a companion cache.

### 1.4 Definitions

We now give a formal definition for a cache that applies to both fully-associative and typed caches.

**Definition 1** (Cache). *A cache is a tuple $(S, P, M)$ where $S$ is a set of slots, $P$ is a set of memory pages, and $M$ is an association matrix. Here $S$ has cardinality $k$ and is equal to $\{s_1, s_2, \ldots, s_k\}$, $P$ has cardinality $l$ and is equal to $\{p_1, p_2, \ldots, p_l\}$, and $M$ is a $k \times l$ matrix on $\{0, 1\}$ with $M_{ij} = 1$ whenever $p_j$ can be placed into $s_i$.*

We will use $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ to reference a sequence of requests to memory pages. Here $\sigma_i$ is the $i$-th memory page requested.

We can now formally define the typed caching problem. Given a cache $(S, P, M)$ and a request sequence $\sigma$, an algorithm $A$ must satisfy the requests while incurring the minimum possible cost. To satisfy the $i$-th request, $A$ must ensure that $\sigma_i$ is in the cache. Since the cache is finite, this may involve evicting some memory page from the cache. It may also involve moving memory pages between slots, a process dubbed reorganization.

We define the cost incurred by an algorithm $A$ when servicing a request to a memory page $p$ as

$$\text{cost}(A, p) = \begin{cases} 1 + b\alpha & p \text{ is not in the cache} \\ 0 & \text{otherwise} \end{cases}$$

Here $0 \leq b \leq 1$ is the cost of moving a single memory page from one slot to another and $\alpha$ is the number of such moves made. We can define the total cost incurred by $A$ over a request sequence $\sigma$ as

$$\text{cost}(A, \sigma) = \sum_{i=1}^{n} \text{cost}(A, \sigma_i)$$

The algorithms we study are termed online algorithms. These algorithms possess no knowledge of future requests. The performance of these algorithms are often measured using the competitive ratio [ST85]. An algorithm $A$ is said to be $c$-competitive if

$$\text{cost}(A, \sigma) \leq c \cdot \text{cost}(A', \sigma) + c_0$$

for every algorithm $A'$ and sequence $\sigma$. Here $c_0$ is a constant that depends on the initial conditions of the problem. For randomized algorithms the competitive ratio is defined differently. A randomized algorithm $A$ is said to be $c$-competitive if

$$E[\text{cost}(A, \sigma)] \leq c \cdot \text{cost}(A', \sigma) + c_0$$

for every algorithm $A'$ and sequence $\sigma$. Here $E[\text{cost}(A, \sigma)]$ is the expected cost of $A$ on $\sigma$.

### 1.5 Our Work

We first define a set of algorithms called EP algorithms for the typed caching problem and provide a lower bound on the competitive ratio of these algorithms. We also

define a subset of these algorithms called marking EP algorithms and give an upper bound on the competitive ratio of these algorithms. Both bounds are dependent on 2 parameters of the cache, $k$ and $d$, which we define in section 1.10.

## 1.6 Past Work

The case of a fully-associative cache with $k$ slots has been well studied. Sleator and Tarjan [ST85] showed that every deterministic algorithm is at best $k$-competitive, and they prove that the algorithms LRU and FIFO achieve this bound. Fiat et al. [FKL+91] showed that every randominized algorithm is at least $H_k$-competitive. They also introduced the algorithm MARK which is $2H_k$-competitive in general and $H_k$-competitive when the number of memory pages is $k+1$. The partitioning algorithm of Manasse et al. [MMS90] achieves the $H_k$ bound in general.

Brehob et al. [BETW03, BETW01] were the first to study $(m, n)$ companion caches. They considered cases with or without bypassing (when a cache is not forced to add the last requested memory page) and with or without free reorganization. They showed that LRU is not competitive in any case and that FIFO is $(m + 3n)$-competitive when bypassing is allowed. They introduced the algorithm $MCF$ and proved that it is $(2n+3)$-competitive when bypassing is allowed. They also introduced the algorithm $MCF_B$ and proved that is $(2n - 2)$-competitive when bypassing is not allowed. They showed that every deterministic algorithm is at least $(2n + 2)$-competitive when $m = n+1$ and bypassing is allowed and at least $(n+1)$-competitive when $m = n + 1$ and bypassing is not allowed. Finally they gave a reorganizing variant of LRU, called RLRU, that is $(2n + 3)$-competitive when bypassing and free reorganization are allowed.

Fiat et al. [FMS02, MS04] further studied $(m, n)$ companion caches with free reorganization allowed. They showed that any deterministic algorithm is at least $(2n+1)$-competitive and that every randomized algorithm is at least $H_{2n+1}$-competitive.

They proved that any deterministic marking algorithm [BRIS91] is optimal. They also defined a subset of marking algorithms called type preference algorithms and gave two randomized type preference algorithms that are $O(\log(n))$-competitive.

Peserico [Pes03] independently studied the case of an arbitrary typed cache with parameters $k$ and $d$ in 2002. He proved a different lower bound of $(k+1)/2$ on the competitive ratio of any EP algorithm. He also showed that EP variants of many traditional caching algorithms are $k(1+bd)$-competitive, and that an EP variant of the algorithm MARK is $2(H_k+1)/(1+bd)$-competitive. Finally he gave a non-reorganizing algorithm called TOUR which is $2k$-competitive.

### 1.7 States

Our definitions and proofs will require some notion of state.

**Definition 2** (state). *Let $C = (S, P, M)$ be a cache. Define* $\text{state}(C,t) : P \to S \cup \{\phi\}$ *such that*

$$\text{state}(C,t)(p) = \begin{cases} s & p \text{ occupies } s \text{ at time } t \\ \phi & p \text{ is not in the cache} \end{cases}$$

### 1.8 Cache Graph

Our algorithms will require a second notion of state, the cache graph.

**Definition 3** (Cache Graph). *Let $C = (S, P, M)$ be a cache. Define $G(C,t)$ as the digraph with $P$ as the nodes and $(p,q)$ a directed edge if* $\text{state}(C,t)(q) \neq \phi$, $p \neq q$, *and $p$ can be placed into* $\text{state}(C,t)(q)$.

Certain paths in a cache graph define a series of actions that can be taken to add a memory page to the cache. For example, let $v_1, v_2, \ldots, v_n$ be a path in $G(C,t)$

and let $\text{state}(C,t)(v_1) = \phi$. Then $v_1$ could be added to the cache using the following procedure.

Evict $v_n$

**for** $i \leftarrow n-1, n-2, \ldots, 1$ **do**

Move $v_i$ to $\text{state}(C,t)(v_{i+1})$

**end for**

We call this procedure evicting along a path from $v_1$ to $v_n$.

**Definition 4** (Eviction Tree). *Let $C$ be a cache. Define $D(C,v,t)$ to be the nodes in the directed tree of $G(C,t)$ rooted at $v$ and excluding $v$.*

### 1.9 EP Algorithms

We now define a class of algorithms for the typed caching problem called eviction path or EP because they evict along paths.

**Definition 5** (EP Algorithm). *Let $C$ be a cache. On a request to a page $v$ at a time $t$ that is not in the cache, an EP algorithm chooses a page $w \in D(C,v,t)$ and evicts it along some path from $v$.*

Note that this class includes such algorithms as LRU and FIFO. These algorithms do not take advantage of reorganization; instead they always evict along paths of length 1. As a result, there are caches for which LRU is not competitive and the performance of FIFO is poor [Pes03, BETW03]. There exist variants of LRU and FIFO respectively that take full advantage of reorganization. We will show that these variants perform quite well in the general case.

**Definition 6** (EPLRU). *Let $C$ be a cache. On a request to a page $v$ at a time $t$ that is not in the cache, EPLRU chooses the page $w \in D(C,v,t)$ least recently requested and evicts it along the shortest path from $v$.*

**Definition 7** (EPFIFO). *Let $C$ be a cache. On a request to a page $v$ at a time $t$ that is not in the cache, EPFIFO chooses the page $w \in D(C, v, t)$ put into the cache earliest and evicts it along the shortest path from $v$.*

### 1.10 A Lower Bound for EP Algorithms

We can prove a lower bound of $k/(1+bd)$ on the competitive ratio of any EP algorithm. Here the maximum size of an eviction tree in a cache graph of $C$ and the maximum length of an eviction path in a cache graph of $C$ are $k+1$ and $d+2$, respectively.

**Lemma 1.** *Let $C$ be a cache and $A$ an EP algorithm. If $r$ is the memory page requested at time $t$ and $A$ evicts a memory page $e$ to satisfy the request, then $r + D(C, r, t) = e + D(C, e, t+1)$.*

*Proof.* The proof is to show that any page $v$ reachable from $r$ in $G(C, t)$ is also reachable from $e$ in $G(C, t+1)$. The opposite direction will hold by symmetry. Let $e_1, e_2, \ldots, e_n$ be the path that $e$ was evicted along. It follows that $e_n, e_{n-1}, \ldots, e_1$ is a path in $G(C, t+1)$. Now let $v_1, v_2, \ldots, v_m$ be a path from $r$ to $v$ in $G(C, t)$. There exists some $i$ and $j$ such that $e_1, e_2, \ldots, e_i, v_j, v_{j+1}, \ldots, v_m$ is also a path in $G(C, t)$ and $i$ is maximal. It follows that $e_i, v_j, v_{j+1}, \ldots, v_m$ is still a path in $G(C, t+1)$. Thus $e_n, e_{n-1}, \ldots, e_i, v_j, v_{j+1}, \ldots, v_m$ is a walk in $G(C, t+1)$. □

**Definition 8** (same). *Let $C$ be a cache and $A_1, A_2, \ldots, A_k, A_{k+1}$ algorithms. The property* same$(t, A_1, A_2, \ldots, A_k, A_{k+1})$ *is true for a time $t$ if there exists a sequence of memory pages $v_1, v_2, \ldots, v_k, v_{k+1}$ where* state$(C, t)(v_i) = \phi$ *and $v_i + D(C, v_i, t) = \{v_1, v_2, \ldots, v_k, v_{k+1}\}$ for each $A_i$.*

**Theorem 1.** *Let $C$ be a cache and $A$ an algorithm. Then $A$ is at least $k/(1+bd)$-competitive.*

*Proof.* The proof is by induction. We show that for a certain request sequence and adversaries $B_1, B_2, \ldots, B_k$ that same$(t, A, B_1, B_2, \ldots, B_k)$ is true at every time $t$. The

inductive step will detail every action taken by the algorithms to satisfy the requests. We will use this to count the costs they incur.

The request sequence $\sigma$ is chosen such that state$(C, 1)(\sigma_1) = \phi$ for $A$ and $\sigma_t$, $t > 1$, is the memory page evicted by $A$ at time $t - 1$. We assume WLOG that $\sigma_1 + D(C, \sigma_1, 1) = \{\sigma_1, v_1, v_2, \ldots, v_k\}$ for $A$.

The initial state of $C$ for each adversary is such that same$(1, A, B_1, B_2, \ldots, B_k)$ holds for the sequence $\sigma_1, v_1, v_2, \ldots, v_k$. We initialize $C$ for each $B_i$ exactly as we did for $A$, except that we also evict $v_i$ along the shortest path from $\sigma_1$. It follows from lemma 1 that $v_i + D(C, v_i, 1) = \{\sigma_1, v_1, v_2, \ldots, v_k\}$ for each $B_i$.

The actions of the algorithms assure that same$(t + 1, A, B_1, B_2, \ldots, B_k)$ holds given that same$(t, A, B_1, B_2, \ldots, B_k)$ holds for some sequence $\sigma_t, w_1, w_2, \ldots, w_k$. At time $t$, $A$ evicts some $w_i$ and $B_i$ evicts $\sigma_t$ along the shortest path from $w_i$. The remaining adversaries do nothing. It follows from lemma 1 that $w_i + D(C, w_i, t+1) = \{\sigma_t, w_1, w_2, \ldots, w_k\}$ for $A$ and $\sigma_t + D(C, \sigma_t, t + 1) = \{\sigma_t, w_1, w_2, \ldots, w_k\}$ for $B_i$.

We now count the costs incurred by the algorithms. Since $A$ adds a memory page to the cache for every request, it incurs a total cost of at least $n$. The adversaries as a whole do the same, so they incur a total cost of at most $n(1 + bd)$. Pigeonhole Principle implies that some $B_i$ incurs a total cost at most $n(1 + bd)/k$. □

### 1.11 Marking EP Algorithms

Marking EP algorithms are a subset of EP algorithms that base their decisions on marks they apply to pages. Marking EP algorithms are a generalization of traditional marking algorithms [BRIS91, Tor95].

**Definition 9** (Marking EP Algorithm). *Let $C$ be a cache. On a request to a page $v$ at a time $t$ that is not in the cache, a marking EP algorithm chooses a page $w \in D(C, v, t)$ that is unmarked and evicts it along some path from $v$. If there are no unmarked pages in $D(C, v, t)$ then the algorithm unmarks all of them and tries again. After satisfying*

*any request the algorithm marks the requested page.*

Many algorithms are marking EP algorithms. In the same way that many traditional algorithms are EP algorithms, so all traditional marking algorithms are marking EP algorithms. EPLRU is also a marking EP algorithm.

## 1.12 An Upper Bound for Marking EP Algorithms

The rigid behavior of marking EP algorithms allows us to prove an upper bound of $k(1 + bd)$ on the competitive ratio of any one of them. Here $k$ and $d$ are the same constants as in the lower bound proof.

**Theorem 2.** *Let $C$ be a cache and $A$ a marking EP algorithm. Then $A$ is at most $k(1 + bd)$-competitive.*

*Proof.* We begin by dividing the request sequence into phases. We then bound the cost incurred by the algorithm in each phase. Finally we bound the cost incurred by the optimal algorithm in terms of the number of evictions it performs.

We will use the following definitions in the proof. We call the marking EP algorithm $A$ and the optimal algorithm $OPT$.

We divide the request sequence into phases as follows. Let $t_1, t_2, \ldots, t_m$ be the times when $A$ unmarks pages. For each $i$, $1 \leq i \leq m$, we define a phase $\rho_i$ in

$$\rho_i = \{u \mid A \text{ marks an unmarked } \sigma_u \text{ at } u \text{ and unmarks it at } t_i\}$$

The whole of the request sequence is not covered by the phases. Instead the requests for which $A$ incurs a cost are covered. Therefore any analysis using only the phases will give an upper bound on the competitive ratio.

We can bound the cost incurred by $A$ in any phase by $k(1 + bd)$. Consider some phase $\rho_i$. Every request during this phase caused $A$ to mark some page and thus

incur a possible cost of at most $(1 + bd)$. Since the size of $\rho_i$ is equal to the number of pages $A$ unmarked at $t_i$, the number of requests during $\rho_i$ is bounded above by $k$.

$OPT$ must perform at least one eviction by the end of every phase. Consider some phase $\rho_i$. Since the memory pages of $\sigma_{t_i} + D(C, \sigma_{t_i}, t_i)$ can not all fit in $C$ at once, $OPT$ must evict one of $D(C, \sigma_{t_i}, t_i)$ by $t_i$ to make room.

$OPT$ must perform at least one eviction per phase. Let $\rho_i$ and $\rho_j$ be phases and assume WLOG that $t_i < t_j$. Furthermore let $v$ be the memory page $OPT$ evicts by $t_i$. If $v \notin D(C, \sigma_{t_j}, t_j)$ then its eviction can not count for $\rho_j$. If $v \in D(C, \sigma_{t_j}, t_j)$ then there exists some $t \in \rho_j$ at which time $A$ marks an unmarked $v$. Since $v$ is marked until $t_i$ and is not marked again at $t_i$, it follows that $t > t_i$. Thus $OPT$ must evict again by $t$ in order to bring $v$ back into the cache.

We can now bound the total cost incurred by the algorithms. Since there are $m$ phases $A$ incurs a total cost at most $mk(1 + bd)$. Since the number of evictions is equal to the number of times a memory page is added to the cache, $OPT$ incurs a total cost of at least the number of evictions. We have shown this to be $m$. □

## 1.13 Conclusion

We have defined the class of EP algorithms for the typed caching problem and given a lower bound for the competitive ratio of these algorithms. We have also defined the marking subset of EP algorithms and given an upper bound for the competitive ratio of such algorithms.

Future work may involve bridging the gap between the upper and lower bounds for marking EP algorithms. It may also involve analyzing randomized EP algorithms and finding algorithms whose performance is independent of cache structure. Some of this work has already been done in [Pes03].

# Chapter 2

# Server Problems

## 2.1 Introduction

The $k$-server problem is one of the most well studied online problems. In the $k$-server problem there are $k$ servers that exist in some metric space. These servers are moved around in order to service a sequence of requests. Each request specifies some point in the metric space and is serviced by moving one of the servers to that point. An algorithm controls the servers and strives to minimize the total distance covered by the servers.

The fully-associative caching problem is a special case of the $k$-server problem. A fully-associative cache with $k$ slots and $m$ memory pages can be modeled using an instance of the $k$-server problem. Here the metric space is the complete graph with $m$ nodes (one for each memory page) and all distances equal to 1.

The $k$-server conjecture posed in [MMS90] postulates the existence of an algorithm that is $k$-competitive on any metric space. The conjecture has been proven for metric spaces such as the real line and any space with $k+2$ points [BK04] using the Work Function Algorithm [CL92]. This algorithm is also $(2k-1)$-competitive on arbitrary metric spaces [KP95]. It is the best performing algorithm known.

## 2.2 Work Functions and the $k$-Server Problem

A work function is any Lipschitz continuous real valued function on a metric space.

**Definition 10** (Work Function). *Let $(\mathbb{M}, d)$ be a metric space. A function $w : \mathbb{M} \to \mathbb{R}^+$ is a work function if*

$$w(x) \leq w(y) + d(x, y) \quad \forall x, y \in \mathbb{M}$$

Work functions are useful tools for the $k$-server problem. They can be used to describe the minimum cost of servicing a request sequence [CL92]. They are also the key concept used in the Work Function Algorithm.

## 2.3 Our Work

We first introduce a new variant of the $k$-server problem and give a deterministic lower bound for it. We then rederive the properties of work functions from [CL92] for the variant. Finally we give a new version of the minimizer concept from [KP95] and show that a property of minimizers does not hold using these definitions in this variant.

## 2.4 The Typed $k$-Server Problem

The typed $k$-server problem is a generalization of the traditional $k$-server problem. In the typed $k$-server problem requests specify both a point on a metric space and a set of servers called the *type*. A request can only be satisfied by servers in its type.

This approach to generalizing the $k$-server problem differs greatly from the *generalized* $k$-server problem [SS06]. In the generalized $k$-server problem, the servers lie in different metric spaces and each server can still be used to satisfy every request. We know of no way to relate these variants.

## 2.5 Definitions

In the traditional $k$-server problem the servers are indistiguishable; therefore previous work has used multisets to track server locations. Multisets are not appropriate when servers are distinguishable, as in the typed $k$-server problem. We propose the use of tuples as a replacement. Their use leads to a simple definition of a type as a subset of $\{1, 2, \ldots, k\}$ and affords us a simple method for measuring the cost of moving servers.

**Definition 11** (Location). *Let* $(\mathbb{M}, d)$ *be a metric space and* $(s_k)$ *some ordering of the servers. We say that the servers are located at some tuple* $(p_k) \in \mathbb{M}^k$ *if* $s_i$ *is located at* $p_i$ *for all* $1 \leq i \leq k$.

**Definition 12** (Cost). *Let* $(\mathbb{M}, d)$ *be a metric space and* $(p_k) \in \mathbb{M}^k$ *the current location of the servers. The cost of moving the servers to some* $(q_k) \in \mathbb{M}^k$ *is*

$$\operatorname{manh}\big((p_k), (q_k)\big) = \sum_{i=1}^{k} d(p_i, q_i)$$

The set $\mathbb{M}^k$ along with the distance function manh is a metric space whenever $(\mathbb{M}, d)$ is one. This allows us to discuss work functions on this space.

## 2.6 A Deterministic Lower Bound for the Typed $k$-Server Problem

The typed $k$-server problem can be considered harder than the traditional $k$-server problem. For example, given a metric space with $k+1$ points we can prove a general lower bound of $2k-1$ for the typed problem. In the traditional problem the highest possible bound for an arbitrary metric space with $k+1$ points is $k$.

**Theorem 3.** *Let* $(\mathbb{M}, d)$ *be the complete graph of size* $k+1$ *where the distance between any two vertices is 1. Every deterministic algorithm is at least* $(2k-1)$*-competitive on this space.*

*Proof.* Let $A$ be the algorithm and assume that $\mathbb{M} = \{1, 2, \ldots, k, k+1\}$. $A$ competes against an adversary $B$. Initially both $A$ and $B$'s servers are located at $(1, 2, \ldots, k)$. $B$ is responsible for issuing requests, and its strategy is to force $A$ into moving all of its servers to a single point and then back. It acts in the following manner.

1. Issue a request $\big(k+1, \{1, 2, \ldots, k\}\big)$. Assume WLOG that $A$ satisfies the request with server 1. $B$ then satisfies the request with server $k$.

2. For each $2 \leq i \leq k$ issue a request $\big(k+1, \{i, i+1, \ldots, k\}\big)$. Assume WLOG that $A$ satisfies the request with server $i$. $B$ need not do anything to satisfy the request.

3. For each $1 \leq i \leq k-1$ issue a request $\big(i, \{i\}\big)$. $B$ need not do anything to satisfy the request.

4. $A$ and $B$'s servers are now both at $(1, 2, \ldots, k-1, k+1)$. Relabel the points in $\mathbb{M}$ so that the servers are at $(1, 2, \ldots, k)$ and goto step 1.

$A$ incurs a cost of $2k-1$ in steps 1 through 3, while $B$ incurs a cost of only 1. □

## 2.7 Work Functions and the Typed $k$-Server Problem

There exists work functions on the metric space $(\mathbb{M}^k, \text{manh})$ that define the minimum cost of servicing a sequence of requests. We show this by generating these work functions from an initial work function and using the requests themselves.

We start by defining the *update operator*. This operator is used to generate a new work function from a given one and a request.

**Definition 13** (Update Operator). *Let $(\mathbb{M}, d)$ be a metric space, $w$ a work function on $(\mathbb{M}^k, \text{manh})$, and $(r, \mu)$ a request. The update operator generates a work function $w \wedge (r, \mu)$ on $(\mathbb{M}^k, \text{manh})$ such that*

$$w \wedge (r, \mu)(A) = \min_{i \in \mu} w(A') + d(r, a_i)$$

*for $A = (a_k)$ and $A' = (a_1, a_2, \ldots, a_{i-1}, r, a_{i+1}, \ldots, a_k)$.*

**Lemma 2.** *The update operator is well-defined.*

*Proof.* Let $A = (a_k)$ and $B = (b_k)$. There exists $i \in \mu$ such that

$$w \wedge (r,\mu)(A) \leq w(A') + d(r,a_i)$$

$$w \wedge (r,\mu)(B) = w(B') + d(r,b_i)$$

for $A' = (a_1, a_2, \ldots, a_{i-1}, r, a_{i+1}, \ldots, a_k)$ and $B' = (b_1, b_2, \ldots, b_{i-1}, r, b_{i+1}, \ldots, b_k)$. Combining these equations yields

$$w \wedge (r,\mu)(A) - w \wedge (r,\mu)(B) \leq w(A') - w(B') + d(r,a_i) - d(r,b_i)$$

This simplifies to

$$w \wedge (r,\mu)(A) - w \wedge (r,\mu)(B) \leq \text{manh}(A',B') + d(a_i,b_i)$$

This is equivalent to

$$w \wedge (r,\mu)(A) - w \wedge (r,\mu)(B) \leq \text{manh}(A,B)$$

□

There is an equivalent definition of the update operator that will be useful in future proofs. Lemma 3 gives this definition.

**Lemma 3.** *Let* $(\mathbb{M}, d)$ *be a metric space,* $w$ *a work function on* $(\mathbb{M}^k, \text{manh})$, *and* $(r,\mu)$ *a request. Then*

$$w \wedge (r,\mu)(A) = \inf_{B \in \text{serv}(r,\mu)} w(B) + \text{manh}(A,B)$$

*where* $\text{serv}(r,\mu) = \{(c_k) \mid \exists i \text{ such that } c_i = r\}$.

*Proof.* Let $A' = (a_1, a_2, \ldots, a_{i-1}, r, a_{i+1}, \ldots, a_k)$ and $B = (b_1, b_2, \ldots, b_{i-1}, r, b_{i+1}, \ldots, b_k)$. It is true that

$$\text{manh}(A, B) = \text{manh}(A, A') + \text{manh}(A', B)$$

This implies that

$$\text{manh}(A, B) - \text{manh}(A, A') \geq w(A') - w(B)$$

This simplifies to

$$w(A') + d(r, a_i) \leq w(B) + \text{manh}(A, B)$$

□

We now show how to generate work functions that define the minimum cost of servicing a sequence of requests.

**Lemma 4.** *Let* $(\mathbb{M}, d)$ *be a metric space,* $X$ *the initial location of the servers, and* $((r_n, \mu_n))$ *a sequence of requests. Then there exists a sequence* $(w_n)$ *of work functions on* $(\mathbb{M}^k, \text{manh})$ *such that* $w_t(A)$ *is the minimum cost of servicing the first* $t$ *requests and ending with the servers at* $A$.

*Proof.* The proof is by induction on $t$. The case of $t = 1$ can be proven with the inductive step by defining $w_0(C) = \text{manh}(X, C)$ for all $C$.

Assume there exists $w_t$ as in the lemma statement and let $(C_m)$ be a sequence of tuples that represents some way of servicing the first $t + 1$ requests and ending with the servers at $A$. There must exist some $1 \leq i \leq m$ such that $C_i \in \text{serv}(r_{t+1}, \mu_{t+1})$. It is true by assumption that

$$w_t(C_i) + \sum_{j=i+1}^{m} \text{manh}(C_{j-1}, C_j) \leq \sum_{j=0}^{m} \text{manh}(C_{j-1}, C_j)$$

This simplifies to

$$w_t(C_i) + \text{manh}(C_i, A) \leq \sum_{j=0}^{m} \text{manh}(C_{j-1}, C_j)$$

Lemma 3 implies that

$$w \wedge (r_{t+1}, \mu_{t+1})(A) \leq \sum_{j=0}^{m} \text{manh}(C_{j-1}, C_j)$$

The inductive step shows that defining $w_{t+1} = w_t \wedge (r_{t+1}, \mu_{t+1})$ for every $t$ gives the required sequence. We can apply the original definition of the update operator to get the actual steps to achieve the minimum cost. □

### 2.8 Minimizers

The $(2k-1)$-competitive proof of [KP95] relies on certain properties of multisets known as minimizers. Reproducing this proof for the typed $k$-server problem would require an analogue to the minimizer concept. We give two possibilities.

**Definition 14** (Minimizer). *Let $(\mathbb{M}, d)$ be a metric space and $w$ a work function on $(\mathbb{M}^k, \text{manh})$. A minimizer of $w$ with respect to a request $(r, \mu)$ is a tuple $A = (a_k)$ such that $w(A) - \sum_{i=1}^{k} d(r, a_i)$ is minimal.*

**Definition 15** (Minimizer). *Let $(\mathbb{M}, d)$ be a metric space and $w$ a work function on $(\mathbb{M}^k, \text{manh})$. A minimizer of $w$ with respect to a request $(r, \mu)$ is a tuple $A = (a_k)$ such that $w(A) - \sum_{i \in \mu} d(r, a_i)$ is minimal.*

One of the properties given is that a minimizer of a work function $w$ with respect to request $r$ is also a minimizer of $w \wedge r$ with respect to $r$. The analogue of this property for the typed $k$-server problem does not hold even for simple cases. Consider the case where $k = 2$, $\mathbb{M}$ is the real line, $(0, 0)$ is the initial configuration of the servers, and $\big(1, \{1, 2\}\big), \big(2, \{1\}\big)$ are the requests. Figure 2.1 shows the minimizer values for $w_1$ and $w_2$ with respect to $\big(2, \{1\}\big)$ using definition 14. Figure 2.2 is the same but uses

definition 15. Note that in each figure, configuration $(0, 1)$ is a minimum value of $w_1$ but not a minimum value of $w_2$.

## 2.9 Conclusion

We have introduced a new variant of the $k$-server problem and provided some analysis of work functions in this new variant. More analysis remains, which includes redefining the work function algorithm and finding bounds for it. It is also worth trying to repair the $(2k - 1)$-competitive proof of [KP95] when looking for these bounds.

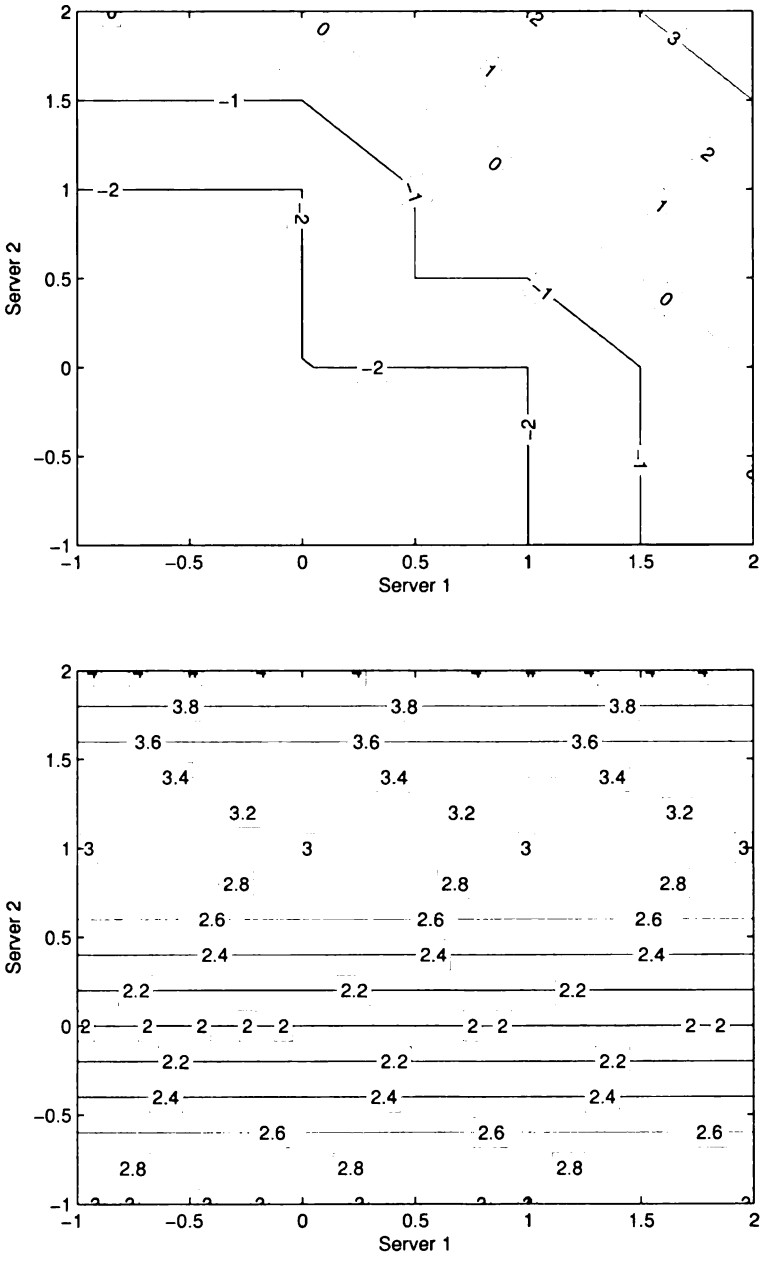


Figure 2.1: Minimizer values with respect to $(2, \{1\})$

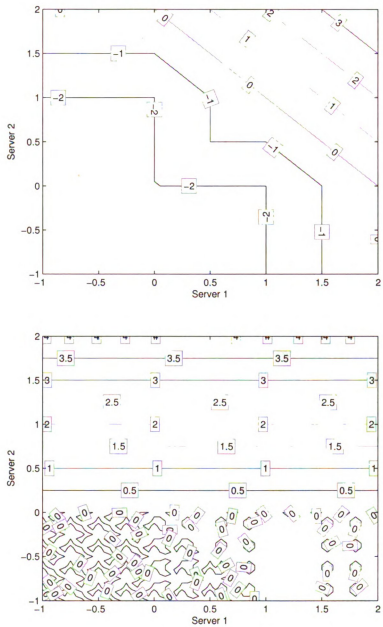


Figure 2.2: Minimizer values with respect to $(2, \{1\})$

# BIBLIOGRAPHY


[BETW01] Mark Brehob, Richard Enbody, Eric Torng, and Stephen Wagner. On-line restricted caching. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 374–383, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[BETW03] Mark Brehob, Richard Enbody, Eric Torng, and Stephen Wagner. On-line restricted caching. *J. of Scheduling*, 6(2):149–166, 2003.

[BK04] Yair Bartal and Elias Koutsoupias. On the competitive ratio of the work function algorithm for the k-server problem. *Theor. Comput. Sci.*, 324(2-3):337–345, 2004.

[BRIS91] Allan Borodin, Prabhakar Raghavan, Sandy Irani, and Baruch Schieber. Competitive paging with locality of reference. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 249–259, New York, NY, USA, 1991. ACM Press.

[CL92] Marek Chrobak and Lawrence L. Larmore. The server problem and on-line games. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 7, pages 11–64. 1992.

[FKL+91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, 1991.

[FMS02] Amos Fiat, Manor Mendel, and Steven S. Seiden. Online companion caching. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 499–511, London, UK, 2002. Springer-Verlag.

[KP95] Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *J. ACM*, 42(5):971–983, 1995.

[MMS90] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, 1990.

[MS04] M. Mendel and Steven S. Seiden. Online companion caching. *Theor. Comput. Sci.*, 324(2-3):183–200, 2004.

[Pes03] Enoch Peserico. Online paging with arbitrary associativity. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 555–564, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[Sez93] André Seznec. A case for two-way skewed-associative caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993. ACM Press.

[SS06] René A. Sitters and Leen Stougie. The generalized two-server problem. *J. ACM*, 53(3):437–458, 2006.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[Tor95] Eric Torng. A unified analysis of paging and caching. In *IEEE Symposium on Foundations of Computer Science*, pages 194–203, 1995.