COMPETITIVE ANALYSIS OF ONLINE DISTRIBUTED PAGING ALGORITHMS

A thesis submitted in partial fulfillment of the requirements
For the degree of Master of Science
in Computer Science

By

Roman Christoph Zulauf

December 2012
The thesis of Roman Christoph Zulauf is approved:

______________________________  _______________________
Richard J. Lorentz, Ph.D.        Date

______________________________  _______________________
Richard Covington, Ph.D.         Date

______________________________  _______________________
John Noga, Ph.D., Chair          Date

California State University, Northridge
Dedication

This thesis is dedicated to my wife Marissa who put up with long hours of work and re-search, and to my parents Béatrice and Hans-Christoph.
Acknowledgements

I would like to thank all the CSUN Professors I have had in my graduate studies, who have all contributed to me undergoing this culminating experience. And a special thank you to Doctor John Noga for his support and technical guidance that he extended to me during the course of this thesis.
Preface

What is distributed paging? Consider today’s massively distributed networks of computers, including servers, home desktops, tablets and mobile phones. Files may be distributed and replicated on various nodes in the network in an effort to minimize file access costs.

There are several different ways of measuring an algorithm’s cost and effectiveness. For example, the asymptotic runtime complexity of an algorithm is one such measure. However, for an optimization algorithm which computes responses for partial input (this is called an online algorithm) asymptotic runtime complexity is only half the story. In practice, a good measure of an online algorithm’s effectiveness is how well it optimizes cost. In other words, we want to know about the cost of the algorithm’s online solution in comparison to the optimal solution on the full input (which is not known in advance by the online algorithm).

In competitive analysis, an algorithm’s performance against an optimal, offline algorithm is measured. In competitive distributed paging, online paging algorithms are analyzed and put into competition against the offline optimal algorithm.

This thesis studies and summarizes previous results in competitive, distributed paging and analyzes variants of previously studied problems.
Table of Contents

Copyright ii
Signature page iii
Dedication iv
Acknowledgements v
Preface vi
Abstract ix

1 Introduction 1
1.1 Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Distributed Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.3 Multi-Level Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.4 Weighted Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.5 Competitive Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Techniques for Competitive Analysis 4
2.1 The Potential Function Method . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.1 Amortized Costs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.2 Delta-Potential Bounds . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.3 Interleaving Moves . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Lower and Upper Bounds . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.1 Adversaries and Cruel Adversaries . . . . . . . . . . . . . . . . . . . 6
2.2.2 Two-Person Zero-Sum Games . . . . . . . . . . . . . . . . . . . . . . 6
2.2.3 Averaging over Adversaries . . . . . . . . . . . . . . . . . . . . . . . 7

3 Previous Work 8
3.1 List Updating . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.3 File Migration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.4 File Replication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.5 File Allocation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.6 Distributed Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4 Variations of Distributed Paging 18
4.1 Simplified Constrained File Replication with Canonical Copy . . . . . . . . . 18
4.1.1 Uniform Network Topologies . . . . . . . . . . . . . . . . . . . . . . . 18
4.1.2 Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.1.3 Arbitrary Network Topologies . . . . . . . . . . . . . . . . . . . . . . 20
4.2 Constrained File Replication with Canonical Copy . . . . . . . . . . . . . . 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1 Uniform Network Topologies</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2 Line with 3 nodes and $k = 1$</td>
<td>22</td>
</tr>
<tr>
<td>4.2.3 Line with 3 nodes and $k = 2$</td>
<td>24</td>
</tr>
<tr>
<td>4.2.4 Arbitrary Network Topologies</td>
<td>26</td>
</tr>
<tr>
<td>4.2.5 Analysis of Some Candidate Algorithms</td>
<td>27</td>
</tr>
<tr>
<td>4.2.6 NP-Completeness and Constrained File Replication With Canonical Copy</td>
<td>28</td>
</tr>
<tr>
<td>4.3 Conclusion</td>
<td>30</td>
</tr>
</tbody>
</table>

References 33

A List of Theorems 34

B Lower bound search code 35
ABSTRACT

COMPETITIVE ANALYSIS OF ONLINE DISTRIBUTED PAGING ALGORITHMS

By

Roman Christoph Zulauf

Master of Science in Computer Science

In this thesis, problems are considered that arise in distributed computing environments, such as shared memory multiprocessor systems where memory is physically distributed amongst the processing nodes. Classic results in paging, file migration, file replication, file allocation and distributed paging are presented.

In the constrained file allocation with canonical copy problem, the issue that is addressed is which individual nodes should contain copies of pages (or files) of data under the constraints of node memory capacities. In the simplified version of the problem, the additional constraint of restricted activity of an algorithm between requests is imposed.

For simplified constrained file allocation with canonical copy, tight bounds for the competitive ratio of online algorithms are presented on uniform networks, trees and arbitrary network topologies. For these topologies, optimal algorithms with a competitive ratio of $k$, $k(d - 1)$ and $kd$ respectively are given.

For constrained file allocation with canonical copy, this thesis presents efficient online algorithms for restricted cases of uniform networks and bus network topologies. On uniform networks, the presented algorithm achieves a competitive ratio of $k$, whereas on lines with 3 nodes and restricted $k = 1$ and $k = 2$ the given algorithms achieve a competitive ratio of 1.5 and 3 respectively.
1.1 Paging

On a personal computer there are multiple memory levels. Two such levels are the hard disk and RAM. The disk can store a large amount of information, but is very slow to access. On the other hand RAM is much smaller in its memory storage capabilities, but can be utilized much more quickly. Typically, the data on the disk is divided into equal sized items or pages and when a particular item is needed it is brought into RAM (if it is not already present). Since RAM is much faster and smaller than the disk, this often means that there will be a delay in processing while the item is retrieved and that another item will have to be removed from RAM. In an effort to minimize delay, there is an attempt to remove or evict items in such a way as to minimize the number of times the disk needs to be accessed. A paging algorithm decides what items to evict. Typically this needs to be done without knowledge of which items will be needed in the future.

1.2 Distributed Paging

In a multiprocessor system with a globally shared address-space as well as in a distributed network of processors where files are stored, a distributed version of the paging problem arises. In fact, for many modern applications data needs to be shared amongst networked computers. Envision for example, a web search engine serving results on every continent on Earth. Other examples would include web-based platforms and services such as Facebook, Twitter or Gmail. Data can be written to and read from these services from virtually any computer as long as a connection to the Internet exists. The data of these services must be distributed in some fashion simply because of the sheer quantity of the data.

Locally stored data can be accessed much more quickly than data that needs to be accessed via the network. In an effort to minimize these delays, files are often stored at multiple locations. The objective for each processor is to keep the information that it will need stored locally. When a processor requires access to a file, it sends a request to a processor holding a copy of that file. The file is then sent back to the processor requesting it, incurring a communication cost which is proportional to the distance between the corresponding processors.

There are differences between this situation and ordinary paging, but the similarities have led to this sort of problem being called distributed paging.

Because of the continuing trend of digitization of the world’s knowledge (e.g. look at books, news, local weather information, real-time traffic updates and travel itineraries, to name a few), the amount of information to be managed and made accessible and useful to end-users is ever increasing. By sheer volume of data it is impossible to store all of it on a single computer. Storage systems consisting of distributed, interconnected processors have
become a necessity to satisfy the demand for increasing, vast storage capacity.

Adding to this the complex network of government, commerce and consumer computing systems, called the Internet, it becomes clear that it is a technological challenge to make the world’s information accessible to anybody who wishes to use it. Network topology has no small role in this. For example, network links connecting mainland and remote islands including even Australia typically have a very large communication cost. This makes file accesses into or out of such geographical areas very expensive. How can one efficiently distribute and cache data to minimize cost and yet give access to all the needed information? This problem is addressed by distributed paging.

1.3 Multi-Level Paging

One problem in regular paging is that typically larger caches have a better hit rate but longer latency. In other words, there is an implicit tradeoff between cache latency and hit rate. One method of addressing this tradeoff is to use multiple levels of cache. Small caches with very low latency are backed by increasingly larger and slower caches. If a page is requested from memory, the fastest (and smallest) cache is queried for the page first. On a cache miss, the second cache (slower and bigger) is checked for the presence of the page. This is continued until either there is a cache hit, or finally the page has to be loaded from the slowest memory. Modern processors typically utilize two or three levels of on-die cache.

The use of multi-level caches have also been studied in distributed file systems, where the file system is distributed across a network of interconnected processors. Some papers on the subject include [CZZ+05] and [MH92].

1.4 Weighted Paging

The problem of regular paging can be extended by page weights (not all pages are created equal!). The weight of a page is typically its size or cost to bring it into cache. Weighted paging finds practical applications in NUMA computer architectures where memory access costs are non-uniform, and in distributed applications where communication costs between nodes is not uniform.

1.5 Competitive Analysis

In practice, many problems require decisions to be made by an algorithm without full knowledge of the impact of that decision for the future. A series of inputs are processed, one after the other, without knowledge of future inputs. Consider, for example, a network router processing incoming data packets. A decision must be made on which outbound link the packet be sent to arrive at its destination with the minimum cost without knowledge of future traffic. Algorithms solving problems of this kind are called online algorithms, as opposed to offline algorithms which are provided the complete input data for the problem at the onset of the computation.

Competitive analysis is a rather young method of providing a measure of algorithmic
efficiency that has developed over the past three decades (see for example, [ST85]). Rather than analyzing worst-case or average-case asymptotic time and space complexity of an algorithm, competitive analysis takes a different approach, which has the appeal of providing a more robust measure of efficiency than previous conventional methods for many problems.

Typically, online algorithms solve an optimization problem, which may be one of cost minimization or profit maximization. Because most optimization problems in computer science are minimization problems, the following text will primarily deal in cost minimization problems.

A good summary of online computation and competitive analysis is given by Borodin and El-Yaniv [BEY05]. Early papers employing techniques of competitive analysis include [ST85], [DKM+86], [MMS88], [BLS92] and [BDBK+90].

An optimization problem \( \mathcal{P} \) consists of an input sequence set \( \Sigma \) and a cost function \( C \). For every input sequence \( \sigma \) in \( \Sigma \), there is a set of possible (valid) outputs \( O \) given by a function \( F(\sigma) \). The cost function \( C(\sigma, O) \) maps each valid output for any input onto the positive reals.

An optimization algorithm \( \text{Alg} \) (in the set \( \mathcal{A} \) of possible, valid algorithms) solves an optimization problem \( \mathcal{P} \) by computing a valid output \( \text{Alg}[\sigma] \in F(\sigma) \). The cost associated with the valid output is indicated by \( \text{Alg}(\sigma) = C(\sigma, \text{Alg}[\sigma]) \). An optimal algorithm \( \text{Opt} \) for an optimization problem \( \mathcal{P} \) is defined to be an algorithm such that for all inputs in \( \Sigma \),

\[
\text{Opt}(\sigma) = \min_{O \in F(\sigma)} C(\sigma, O). \tag{1.1}
\]

In competitive analysis, a candidate algorithm \( \text{Alg} \) is compared against (or set in competition against) an optimal algorithm \( \text{Opt} \). An online algorithm \( \text{Alg} \) is \( k \)-competitive if there is a constant \( \alpha \) such that for all finite input sequences \( \sigma \),

\[
\text{Alg}(\sigma) \leq k \cdot \text{Opt}(\sigma) + \alpha. \tag{1.2}
\]

An algorithm that is \( k \)-competitive is said to attain a competitive ratio of \( k \). If \( \alpha = 0 \), the algorithm is said to be strictly \( k \)-competitive. For many problems, this distinction is irrelevant. The competitive ratio of an algorithm \( \text{Alg} \) is the infimum over the set of all values of \( k \) such that \( \text{Alg} \) is \( k \)-competitive. This is denoted by \( R(\text{Alg}) \).

Note that the actual optimal algorithm does not necessarily need be known to prove the competitive ratio of \( \text{Alg} \).
Chapter 2
Techniques for Competitive Analysis

2.1 The Potential Function Method

An essential method of analyzing competitive ratios of algorithms is the use of a potential function.

The potential function method essentially represents accrued cost advantages over the adversary algorithm as “potential energy” or “potential” which can be used later to cover costs of expensive operations. As such, it is a method of accounting and of amortized analysis.

In analyzing an algorithm using the potential function method, one considers an event sequence \( e_1, e_2, \ldots, e_n \) in response to a request sequence \( \sigma \). An event sequence consists of the specific operations executed by both the candidate and the adversary algorithm for servicing some requests. Depending on the proof style used, a single event \( e_i \) in a sequence of events may consist of a set of operations, each executed by either the candidate or the adversary algorithm, or may consist of a set of operations executed exclusively by the candidate or exclusively by the adversary algorithm.

The configuration of an algorithm is defined as the state of the algorithm. More precisely, it is the state of all variables which may be read by the algorithm in executing an operation or which are used in determining the cost of an operation.

A potential function \( \Phi \) is defined as a mapping from the configurations of \( \text{Alg} \) and \( \text{Opt} \) into the reals. More formally, denote by \( S_{\text{Alg}} \) and \( S_{\text{Opt}} \) the sets of possible configurations for \( \text{Alg} \) and \( \text{Opt} \), respectively. A potential function \( \Phi \) is any mapping, \( \Phi: S_{\text{Alg}} \times S_{\text{Opt}} \rightarrow \mathbb{R} \).

In simple terms, the potential \( \Phi_i \) is a lower bound on how much less the candidate algorithm has paid in processing the first \( i \) requests than it would be allowed to have paid to be \( k \)-competitive. For every event in which the adversary is active, the potential increases by some amount. For every event in which the candidate is active, the potential decreases.

Several proof styles using the potential function method have been successfully used. A few of them are introduced here.

2.1.1 Amortized Costs

This style was used by Sleator and Tarjan in proving the competitive ratio of the move-to-front algorithm for the list update problem [ST85].

Let \( \text{Alg}_i \) (\( \text{Opt}_i \)) denote the cost incurred for processing the \( i \)th event. The amortized cost \( a_i \) of the algorithm for processing the \( i \)th event with respect to the potential function \( \Phi \) is defined by

\[
a_i = \text{Alg}_i + \Phi_i - \Phi_{i-1}.
\]  

(2.1)
For a sequence $\sigma$ of $n$ requests, this yields

$$C_{\text{Alg}}(\sigma) = \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i + (\Phi_0 - \Phi_n).$$

(2.2)

The following conditions are necessary and sufficient to prove that an algorithm is $k$-competitive:

1. For each event $e_i$, $a_i \leq k \cdot \text{Opt}_i$. In other words, the amortized cost of any single event is within the competitive ratio $k$.

2. There exists a constant $\alpha$ such that for any request sequence $\sigma$ in $\Sigma$ and for all $i$, $\Phi_i \geq \alpha$. This requirement ensures that the potential function is bounded from below by a constant (remember that a positive potential denotes a credit in terms of cost that can be used for a later, expensive operation) and the algorithm is not allowed to “debit” from the potential in a way that would violate it being $k$-competitive.

2.1.2 Delta-Potential Bounds

In this proof style, events are partitioned into three types: only $\text{Alg}$ is active during the event, only $\text{Opt}$ is active during the event, or both $\text{Alg}$ and $\text{Opt}$ are active during the event.

Let $\text{Alg}_i$ ($\text{Opt}_i$) denote the cost incurred for processing the $i$th event. Define the following inequality:

$$k \cdot \text{Alg}_i - \text{Opt}_i \geq \Phi_i - \Phi_{i-1}$$

(2.3)

It is sufficient to find a potential function $\Phi$ which satisfies the following conditions to prove that algorithm $\text{Alg}$ is $k$-competitive:

1. For each event $e_i$, inequality (2.3) holds.

2. There exists a constant $\alpha$ such that for any request sequence $\sigma$ in $\Sigma$ and for all $i$, $\Phi_i \geq \alpha$.

The following follows from inequality (2.3). If only $\text{Alg}$ is active during an event, $\Phi$ must decrease by at least $\text{Alg}_i$. If only $\text{Opt}$ is active during an event, $\Phi$ may increase by at most $k \cdot \text{Opt}_i$.

A full proof requires separate analysis of all three types of events, proving the inequality for each.

This proof style was used by Black and Sleator in proving the competitive ratio of a file migration algorithm [DBS89].
2.1.3 Interleaving Moves

This proof style is very similar to the delta-potential method described above. It is described in [BEY05].

Only the following type of events are valid using this method: only \( A \) is active during the event and only \( O \) is active during the event.

To prove that algorithm \( A \) is \( k \)-competitive, it is sufficient to show the following conditions hold with respect to any event sequence:

1. If only \( A \) is active during event \( e_i \) and pays \( A_i \), \( \Phi \) decreases by at least \( A_i \).
2. If only \( O \) is active during event \( e_i \) and pays \( O_i \), \( \Phi \) increases by at most \( k \cdot O_i \).
3. There exists a constant \( \alpha \) such that for any request sequence \( \sigma \) in \( \Sigma \) and for all \( i \), \( \Phi_i \geq \alpha \).

2.2 Lower and Upper Bounds

Several techniques exist for finding lower and upper bounds on the competitive ratio of a given problem. A few are given below.

2.2.1 Adversaries and Cruel Adversaries

An adversarial proof describes a sequence \( \sigma \) of requests, calculates the cost of \( A \) on \( \sigma \), and a way of serving those requests which necessarily has no less cost than \( O \). A cruel adversary makes use of a sequence \( \sigma \) of requests where each request has non-zero cost for \( A \) and zero cost for \( O \) unless no such request exists.

These techniques can be employed in establishing a lower bound for an algorithm. It is sufficient to find a single request sequence forcing \( A \) to perform \( k \) times worse than an optimal algorithm to prove that \( A \) can be at best \( k \)-competitive. Furthermore, if the ratio of costs \( A(\sigma)/O(\sigma) \) for an arbitrarily expensive request sequence \( \sigma \) is not bounded from above by a constant factor \( k \) then \( A \) is not competitive.

For some problems, the above properties can be shown to hold for any online algorithm \( A \), proving that no online algorithm having a lower competitive ratio can exists.

2.2.2 Two-Person Zero-Sum Games

Finding a simple cruel request sequence forcing any online algorithm to perform badly is not always feasible. The following technique can be applied to any problem; if other techniques can be applied they are preferred due to the combinatorial complexity of this technique.

Competitive analysis can be viewed as the analysis of a two-person zero-sum game between an online player and an adversary. Consider a request-answer game of a cost
minimization problem. The online player seeks to minimize cost, whereas the adversary seeks to feed the online player requests that maximize the online players cost versus optimal offline cost. This ratio is the competitive ratio and is the value of the zero-sum game.

The value of the game can be determined by constructing a game tree (for most non-trivial games the tree will be exponential in size) and applying the minimax theorem [VNM44]. The theorem is given below:

**Theorem 2.1.** Every finite two-person zero-sum game has a value. That is,

$$\max_x \min_y H(x, y) = \min_y \max_x H(x, y)$$  (2.4)

where $H : S^1 \times S^2$ is a payoff function and $S^i$ is a strategy for player $i \in \{1, 2\}$.

### 2.2.3 Averaging over Adversaries

Sometimes it is hard to get tight bounds on the optimal offline cost for a given problem and request sequence $\sigma$. In that case an averaging technique can help, which works as follows: if we know that a set of algorithms have a total cost of at least $c$, then there must exist an optimal algorithm $Opt$ with cost of at most the average of costs of all considered algorithms.

More formally, determine a set $M$ of algorithms, each of which serve the same request sequence $\sigma$. An upper bound for the cost of $Opt(\sigma)$ is given by

$$Opt(\sigma) \leq \frac{\sum_M M(\sigma)}{|M|}$$  (2.5)
Chapter 3
Previous Work

In this chapter, previous work on problems related to distributed paging are presented. Specifically, these include:

- **List Updating**: maintain a set of items in a list under an intermixed sequence of accesses, inserts and deletes.
- **Paging**: maintain contents of fast memory under a sequence of page accesses.
- **File Migration** [DBS89]: optimize file access costs under a sequence of reads, migrating files to different processors in the network as necessary.
- **File Replication** [DBS89]: optimize file access costs under a sequence of reads, replicating files to different processors in the network as necessary.
- **File Allocation** [Chu69], [DF82], [ML77]: optimize file access costs under a sequence of reads and writes, replicating and deleting files amongst processors as necessary, keeping copies consistent upon write.
- **Distributed Paging** [BFR95]: optimize file access costs under a sequence of reads and writes, replicating and deleting files amongst processors as necessary, keeping copies consistent upon write, under the additional constraints of local storage capacity of processors.

3.1 List Updating

In their seminal paper of 1985, Sleator and Tarjan analyzed the efficiency of online list update and paging algorithms [ST85].

The **list update problem** (also called the **list accessing problem**) is formally defined as follows.

Maintain a set of items in a list under an intermixed sequence of the following three request types:

- **access**(*x*): Locate item *x* in the list.
- **insert**(*x*): Insert item *x* in the list.
- **delete**(*x*): Delete item *x* from the list.
Using an unsorted list, these request types can be serviced in the following way. To access an item, the list is scanned from the front until the item is located. To insert an item, the entire list is scanned to verify the item is not already present; if it is not, it is inserted at the end of the list. To delete an item, the list is scanned from the front until the item is located and is then removed.

The cost of servicing a request is defined as follows. Accessing or deleting the $i$th item in the list has a cost of $i$. Inserting an item costs $n + 1$, where $n$ is the size of the list before the insertion.

The list may be reorganized at any time by performing a sequence of transpositions (or exchanges) of consecutive items. Immediately following an access or insertion of an item $x$, $x$ may be moved to any position closer to the beginning of the list at no cost. This constitutes free transpositions. Any other transposition is a paid transposition and has a cost of 1.

Note that the list update problem is a special case of the dictionary problem, which does not impose the restriction of using a simple unsorted list as the data structure used for storing the set of items.

Well-known algorithms that have been extensively studied are:

- **MOVE-TO-FRONT** (MTF): After accessing or inserting an item, move it to the front of the list, without changing the relative order of the other items.

- **TRANSPOSE** (TRANS): After accessing or inserting an item, transpose it with the immediately preceding item.

- **FREQUENCY-COUNT** (FC): Maintain a frequency counter for each item, initially zero. Whenever an item is accessed or inserted, increment its frequency count by 1. When an item is deleted, set its frequency count to zero. Maintain the list so that immediately after each frequency counter update, the items are organized in nonincreasing order of their frequencies.

For any algorithm $\text{ALG}$ and any sequence of requests $\sigma$, let $C_{\text{ALG}}(\sigma)$ be the total cost of servicing all requests, not counting paid transpositions, let $P_{\text{ALG}}(\sigma)$ be the number of paid transpositions and let $F_{\text{ALG}}(\sigma)$ be the number of free transpositions. Note that $P_{\text{MTF}}(\sigma) = P_{\text{TRANS}}(\sigma) = P_{\text{FC}}(\sigma) = 0$.

**Theorem 3.1.** For any request sequence $\sigma$ consisting of $n$ requests, starting with an empty list,

$$C_{\text{MTF}}(\sigma) \leq 2 \cdot C_{\text{Opt}}(\sigma) + P_{\text{Opt}}(\sigma) - F_{\text{Opt}}(\sigma) - n.$$  

**Proof.** Consider both MTF and Opt processing a sequence of requests $\sigma$ concurrently. A potential function maps the state of the lists of MTF and Opt to a real number $\Phi$. If
servicing request $i$ has cost $c_i$, and changes the state of the lists from one having potential $\Phi_{i-1}$ to $\Phi_i$, then the amortized cost $a_i$ of servicing request $i$ is defined to be $c_i + \Phi_i - \Phi_{i-1}$.

Processing this sequence of requests, with $\Phi_0$ being the initial potential and $\Phi_n$ being the final potential, it is clear to see that:

$$C_{MTF}(\sigma) = \sum_{i=1}^{n} c_i = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i. \tag{3.2}$$

The number of inversions in MTF’s list with respect to Orr’s list is used as the potential function. An inversion is defined to be an ordered pair of items $(j, k)$ where $j$ precedes $k$ in MTF’s list, but $k$ precedes $j$ in Orr’s list. In other words, $\Phi_i$ represents the number of inversions in MTF’s list with respect to Orr’s list after processing request $i$. Note that $\Phi_i$ is always non-negative. Further note that $\Phi_0$ is 0 by definition, because the initial state of both algorithms is an empty list.

To obtain the theorem, it will be shown that the amortized cost of MTF to service an access request for item $j$ is at most $2j - 1$, the amortized cost of mtf to service an insert request for a list of size $j$ is at most $2(j+1) - 1$, and the amortized cost of mtf to service a delete request for item $j$ is at most $j$, with $j$ being the index of the item in Orr’s list. Additionally, the amortized cost charged to mtf when Orr does a free transposition is $-1$, and at most 1 for a paid transposition. Once these are established, the theorem is proven, since the $-1$’s, one per operation, sum up to $-m$.

It remains to prove the bounds on the amortized costs of servicing individual requests. Firstly, consider an access request to an item $j$. Let $k$ be the position of $j$ in MTF’s list and let $v$ denote the number of items that precede $j$ in MTF’s list but succeed $j$ in Orr’s list. It follows that $k - 1 - v$ items precede $j$ in both lists. When MTF moves item $j$ to the front of the list, $v$ inversions are destroyed, while $k - 1 - v$ new inversions are created with respect to the list of Orr. Thus, the amortized cost of servicing an access request is $k + (k - 1 - v) - v = 2(k - v) - 1$. But $(k - v) \leq j$ because of the $k - 1$ items preceding $j$ in MTF’s list, only $j - 1$ items precede $j$ in Orr’s list. Therefore, we get

$$k + (k - 1 - v) - v = 2(k - v) - 1 \leq 2j - 1 \tag{3.3}$$

which proves the upper bound of amortized cost for servicing the access request.

Next, consider an insert request of an item $j$. Since it is not known where in the list Orr will place the new item, the same argument of inversions applies with virtually no change. Inserting a new item $j$ costs $j + 1$ and we have at most $j - 1$ new inversions, we get

$$(j + 1) + (j - 1) = 2j \leq 2(j + 1) - 1 \tag{3.4}$$

as the amortized cost.
In the case of servicing a delete request, no new inversions are created. Using the same notion of destroyed inversions as above, we get the amortized cost of $k - v$ which clearly satisfies the inequality $k - v \leq j$ because again, of the $k - 1$ items preceding $j$ in MTR’s list, only $j - 1$ items precede $j$ in Opt’s list. Thus we see that servicing a delete request has an amortized cost of no more than $j$.

Because paid and free transposition conducted by Opt have zero cost to MTR, the amortized cost is simply the increase in the number of inversions caused by the transposition. If a paid transposition is executed, there can be at most 1 additional inversion. If a free transposition is executed, there are exactly $-1$ additional inversions, because MTF has the previously accessed item in the front of the list and the free transposition will destroy exactly one inversion moving the item forward in Opt’s list.

3.2 Paging

On a personal computer there are multiple memory levels of differing speed. Two such levels are random access memory which is very fast but a very constrained resource (low capacity), and the computer’s hard disk which is slower by several orders of magnitude but has a much greater capacity. Typically, the data on the disk is divided into equal sized items or pages and when a particular item is needed it is brought into memory (if it is not already there). Since random access memory is much faster and smaller than the disk, this often means that there will be a delay in processing while the item is retrieved and that another item will have to be removed from RAM. In an effort to minimize delay, there is an attempt to remove or evict items in such a way as to minimize the number of times the disk needs to be accessed. A paging algorithm decides what items to evict. Typically this needs to be done without knowledge of which items will be needed in the future.

A formal definition of the paging problem follows. Consider a computer with two levels of memory of varying speed. The memory is divided into pages of fixed uniform size. Denote by $n$ the size of the computer’s fast memory.

Maintain the contents of the fast memory under a sequence of requests of type:

- **access(x)**: Access page $x$.

If the request accesses a page that is already in the computer’s fast memory, the access cost is zero. If the request accesses a page that is not already in fast memory, it must be read from slow memory and substituted for a page in fast memory. This case represents what is called a page fault and has access cost of one. For any sequence of access requests, the goal is to minimize the number of page faults (and thus the total cost).

The following are paging rules that have been extensively studied. The first four are online paging strategies. Only the last algorithm (Mls) makes use of future knowledge and is thus an offline algorithm.
• **LEAST-RECENTLY-USED** (LRU): upon a page fault, evict the page whose most recent access is the earliest

• **FIRST-IN-FIRST-OUT** (FIFO): evict the oldest page in fast memory

• **LAST-IN-FIRST-OUT** (LIFO): evict the page that was added to fast memory most recently

• **LEAST-FREQUENTLY-USED** (LFU): maintain a frequency counter for each page in fast memory, initially zero. Whenever a page is accessed, increment its frequency count by one. Upon a page fault, evict a page with minimal frequency count.

• **LONGEST-FORWARD-DISTANCE** (MIN): evict the page whose next access is the latest.

The strategy longest forward distance exactly minimizes page faults for the paging problem and is thus known as (MIN) and is optimal [Bel66].

The following theorem states the general lower bound for paging.

**Theorem 3.2.** Let \( \text{Alg} \) be any on-line paging algorithm. Denote by \( n_{\text{Alg}} \) the size of \( \text{Alg} \)'s fast memory and by \( n_{\text{Min}} \) the size of Min’s fast memory. There is an infinite sequence of requests \( \sigma \) such that

\[
\text{Alg}(\sigma(n)) \geq (n_{\text{Alg}}/(n_{\text{Alg}} - n_{\text{Min}} + 1)) \cdot \text{Min}(\sigma(n)).
\]  

for infinitely many values of \( n \), where \( \sigma(n) \) denotes the first \( n \) requests of \( \sigma \).

**Proof.** The following request sequence \( \sigma \) forces \( \text{Alg} \) to perform poorly.

The first \( n_{\text{Alg}} - n_{\text{Min}} + 1 \) accesses are to pages which are neither in \( \text{Alg} \)'s or Min’s fast memory. The next \( n_{\text{Alg}} - 1 \) accesses are to pages not currently in \( \text{Alg} \)'s fast memory. In this sequence, the last \( n_{\text{Alg}} \) accesses all cause \( \text{Alg} \) to page fault, whereas Min only faults \( (n_{\text{Alg}} - n_{\text{Min}} + 1) \) times. This follows from the fact that Min retains all pages needed for the last \( n_{\text{Min}} - 1 \) accesses. To obtain the theorem, the sequence can be continued in the same fashion as many times as desired. \( \square \)

**Theorem 3.3.** For any sequence \( \sigma \),

\[
\text{LRU}(\sigma(n)) \leq (n_{\text{LRU}}/(n_{\text{LRU}} - n_{\text{Min}} + 1)) \cdot \text{Min}(\sigma(n)) + n_{\text{Min}}.
\]  

**Proof.** Given any sequence of requests \( \sigma \) divide it into phases \( \sigma_1, \sigma_2, \ldots \) such that \( \sigma_i \) is the longest sequence of requests immediately after \( \sigma_{i-1} \) that contains \( n_{\text{LRU}} \) pages distinct from the last request in \( \sigma_{i-1} \). For each phase LRU will have cost at most \( n_{\text{LRU}} \) because it will fault at most once on any page in a single phase. The algorithm min will have cost at least \( n_{\text{LRU}} - n_{\text{Min}} + 1 \). \( \square \)
3.3 File Migration

A general survey of file allocation and migration is given in [GLS90]. Black and Sleator analyzed online algorithms for file replication and migration in a distributed setting [DBS89].

The file migration problem is formally defined as follows.

An undirected, weighted graph \( G = (V, E) \) is given, representing a network of \( n \) interconnected processors. The edge weight \( w(i, j) \) with \( i, j \in V \) specifies the distance between nodes \( i \) and \( j \). In the model considered by Black and Sleator, only distance metrics which are symmetric and satisfy the triangle inequality are allowed. The symmetric nature is given by the fact that the input graph is undirected. To satisfy the triangle inequality, every edge weight \( w(i, j) \) must be equal to the shortest-path distance \( \delta(i, j) \) between nodes \( i \) and \( j \).

Maintain exactly one copy of a file in the network under a sequence of requests originating from any processor \( p \in V \). Only one request type is allowed:

- \textbf{read}(x, p): Read file \( x \) at processor \( p \).

The cost of servicing a read request is zero if the file is already resident on the processor requesting the file. If the requesting processor does not already have the file, the cost is the distance between the requesting processor and the nearest processor with the file in the network, given by \( \delta(i, j) \).

In addition to servicing read requests, a processor may migrate a file resident on another node to itself at any time. Let \( m \) be a constant defining the ratio of file replication cost over read cost. The cost for a file migration is \( m \cdot \delta(i, j) \).

\textbf{Theorem 3.4}. Let \( A_{lg} \) be any on-line page migration algorithm for a topology with at least two nodes. Then there is an infinite sequence of requests \( \sigma \) such that \( C_{A_{lg}}(\sigma(n)) \geq n \), and

\[ A_{lg}(\sigma(n)) \geq 3 \cdot \text{Opt}(\sigma(n)) \]  

for infinitely many values of \( n \), where \( \sigma(n) \) denotes the first \( n \) requests of \( \sigma \).

Black and Sleator [DBS89] showed that the “on-line block retention problem in a model allowing \textit{Supplythrough} and \textit{Updatethrough}” provided there are at “at least two caches” can be generalized to page migration. For this problem, a lower bound of 3 has already been established [DKM+86], hence proving the above lower bound for page migration. For a full proof, the interested reader is referred to the papers cited.

A description of algorithm M follows. For each node in \( V \), maintain an integral counter in the range \([0, 2m]\), initially set to zero; \( c_p \) denotes the count on node(processor) \( p \). Service a read request to processor \( p \) as follows: if processor \( p \) already has the file, the request is free and a no-op. If processor \( p \) does not have the file and \( c_p < 2m \), increment \( c_p \) and
decrement some other non-zero counter if there is one. If processor \( p \) does not have the file and \( c_p = 2m \) then migrate the page to processor \( p \) and set \( c_p \) to zero.

Consider a fully connected graph with the uniform distance metric, i.e. on a graph where \( \forall i, j \in V \delta(i, j) = 1 \) and thus the replication cost between any pair of processors is an integer \( m \). On this graph, the algorithm pays a cost of zero if the requested file is already resident on the processor servicing the request, a cost of one if the file is not resident and \( c_p < 2m \), and a cost of \( m \) if the file is not resident, \( c_p = 2m \) and the file is migrated.

Consider the following simple example. Let \( G = (V, E) \) with \( V = \{1, 2\} \) and \( E = \{(1, 2)\} \). Let \( R_p \) denote the set of files resident on processor \( p \). Let \( R_1 = \emptyset \) and \( R_2 = \{1\} \). Let \( \sigma_i = (f, p) \) be a request reading file \( f \) at processor \( p \).

The following request sequence \( \sigma \) (which can be repeated infinitely) will force algorithm \( M \) to perform three times worse than the optimal algorithm \( \text{Opt} \).

\[
\sigma_i = \begin{cases} 
(1, 1) & \text{if } 1 \leq i \leq 2m + 1 \\
(1, 2) & \text{if } 2m + 1 < i \leq 2(2m + 1)
\end{cases}
\]

In this request sequence, \( M \) is forced to pay the cost of migrating a file with no benefit, and migrating it back to its original location. \( \text{Opt} \) does not migrate the file and simply pays the access cost for all requests where the file is not resident on the processor handling the request.

While intuitively this scenario has the appearance of holding for a fully-connected graph of any size, a more rigorous analysis is required for a complete proof.

**Theorem 3.5.** Algorithm \( M \) is strictly 3-competitive for the file migration problem on a uniform network. In particular, for any sequence \( \sigma \) of requests and an optimal online or offline algorithm \( \text{Opt} \)

\[
C_M(\sigma) \leq 3 \cdot C_{\text{Opt}}(\sigma)
\]

under the assumption that \( M \) and \( \text{Opt} \) start in the same state.

In 1997, Chrobak proved the lower bound for deterministic algorithms to be no lower than \( 85/27 \approx 3.148 \) [CLRW93]. For this proof, a very specific ring with 4 vertices was used. In 2007, this lower bound was improved to 3.164 by Matsubayashi [Mat07], using a specially weighted ring with 5 vertices.

On general networks, the algorithm Move-To-Min due to [ABF93] achieves a competitive ratio of 7. An improvement of this algorithm, Move-To-Local-Min achieves the currently best competitive ratio achieved by a deterministic algorithm on general networks, 4.086 [BCI97].
3.4 File Replication

The file replication problem, defined and analyzed by Black and Sleator [DBS89], attempts to optimize access to read-only files replicated amongst a set of processors in a network.

The same model of network and distance metrics is used as for the file migration problem. A single file may be replicated to any processor, but never deleted. The cost of replication is \( r \) times the distance to the nearest processor with the file.

In this problem, also only one request type is allowed:

- **read**(\( x, p \)): Read file \( x \) at processor \( p \).

It is worth noting that in this problem, only strictly competitive algorithms are considered. Without this restriction, a 0-competitive algorithm would exist: simply replicate all files to all nodes at the beginning before processing the request sequence. Because the cost of replication is independent of the request sequence, it can be hidden in the additive constant of \( \alpha \) [ACN96], [Bie12].

**Theorem 3.6.** For any request sequence \( \sigma \) consisting of \( n \) requests,

\[
\text{Alg}(\sigma(n)) \geq 2 \cdot \text{Opt}(\sigma(n)).
\]  

**Proof.** Let \( s \) and \( t \) be the two nodes in a two-node network. Consider a sequence \( \sigma \) of events where a file \( f \) is read repeatedly at processor \( t \) until the online algorithm replicates file \( f \) to \( t \). Let \( n \) be the number of requests in \( \sigma \). The online algorithm incurs the cost of reading the file \( n \) times plus the cost of replication, which equates to \( (n + r)\delta_{st} \). An optimal, offline algorithm pays at most half of this cost. In the case where \( n < r \), the offline algorithm never replicates the file and incurs a cost of \( n\delta_{st} \). In the case where \( n \geq r \), the offline algorithm replicates the file before the first read and incurs a cost of \( r\delta_{st} \).

**Theorem 3.7.** There exists a deterministic 2-competitive algorithm for file replication on a fully connected graph with uniform distance metrics.

**Proof.** It is easy to derive this algorithm from the lower bound proven above. Let algorithm \( \mathbf{r} \) maintain a counter on each node of the number of reads issued at that node. When the counter at a node \( i \) reaches \( r \), replicate the file to \( i \).

Let \( k \) be the number of requests issued at node \( i \). The cost incurred by algorithm \( \mathbf{r} \) at \( i \) is \( k\delta_{ij} \) if \( k < r \) (which is optimum) or \( 2r\delta_{ij} \) otherwise; the minimal cost in this case is \( r\delta_{ij} \), thus \( C_{\mathbf{r}}(\sigma) \leq 2 \cdot C_{\text{Opt}}(\sigma) \).

Similar 2-competitive algorithms exist for file replication on trees and uniform trees, the details of which are available in [DBS89].
Albers and Koga [AK98] introduced a technique for transforming any $k$-competitive algorithm on trees into a $2k$-competitive algorithm on rings. This technique automatically yields a 4-competitive algorithm on rings.

Subsequently, Fleisher, Glazek and Seiden studied file replication on rings in particular, improving lower and upper bounds [Gła99], [FS00], [Gła01], [FGS04]. Specifically, for deterministic algorithms they narrowed the gap of bounds on a continuous ring to [2.31023, 2.54150], conjecturing that the lower bounds can be raised to match the upper bound.

### 3.5 File Allocation

While in file migration a single copy of a file is migrated between processors, and in file replication a file is replicated to multiple processors under a sequence of reads (no writes), the file allocation problem considers both reads and writes in replicating files for minimizing access costs. File allocation collapses to file migration if only write requests are processed, and collapses to file replication if only read requests are processed.

Formally, the file allocation problem can be defined as follows [BFR95]. An undirected, weighted graph $G = (V, E)$ is given, representing a network of $n$ interconnected processors. The edge weight $w(i, j)$ with $i, j \in V$ specifies the distance between nodes $i$ and $j$.

Initially, each file is stored on a non-empty subset $Q \subseteq V$ of processors. The algorithm processes two types of requests:

- **read($x, r$):** Read file $x$ at processor $r$. The request is served by a processor $p$ closest to processor $r$.
- **write($x, w$):** Write file $x$ at processor $w$. The algorithm must update the file on all processors holding a copy of the file.

The cost of a read request is the distance between processors $p$ and $r$. The cost incurred for a write request is the minimal Steiner tree spanning $Q \cup \{w\}$ ($Q$ being the current sub-set of processors holding a copy of the file).

In addition to servicing the above two types of requests, the algorithm may at any time:

- Delete a copy of a file at a processor, unless it is the last copy in the network. This operation has a cost of 0.
- Replicate a file from a processor $p$ holding a copy of the file, to a subset $Q' \subset V$. This operation has a cost of $D$ times the minimal Steiner tree spanning $Q' \cup \{p\}$, where $D$ is the ratio between the size of the entire file and the size of the minimal data unit being read or written.

In [BFR95], Bartal et. al. give a a deterministic 3-competitive algorithm for uniform networks, as well as a $(3 + O(1/D))$-competitive, deterministic algorithm for trees.
[ABF93] gives an optimal deterministic $O(\log n)$-competitive algorithm for the file allocation problem on arbitrary network topologies.

### 3.6 Distributed Paging

The **distributed paging problem**, also referred to as constrained file allocation, is the solution of the file allocation problem under the additional constraints of local storage capacity of the processors [BFR95].

Define by $m = \sum_p k_p$, the total number of files that can be stored in the network. Let $k = \max_p k_p$ be the maximum number of files that can be stored on any single processor.

As in file allocation, the algorithm processes two types of requests:

- **read**($x$, $r$): Read file $x$ at processor $r$. The request is served by a processor $p$ closest to processor $r$.
- **write**($x$, $w$): Write file $x$ at processor $w$. The algorithm must update the file on all processors holding a copy of the file.

with the same cost functions as file allocation.

**Theorem 3.8.** The competitive ratio of any distributed paging algorithm is at least $2m - 1$ in any topology when the storage capacity of all processors is equal.

The proof of this lower bound is contained in [BFR95]. It is achieved using the multiple adversary lower bound technique introduced by [MMS88].

On uniform topologies, the deterministic **DISTRIBUTED-FLUSH-WHEN-FULL** algorithm attains a competitive ratio of $3m$ [BFR95]. This algorithm is a distributed version of **FLUSH-WHEN-FULL** given in [DKM*86].

In the paper [ABF96], Awerbuch, Bartal and Fiat analyzed distributed paging for general networks. Attaining a competitive deterministic algorithm at all involves reducing read/write distributed paging to read-only distributed paging and applying several other results, such as the **vertex inclusion-exclusion theorem**. They show that a deterministic, $\text{polylog}(m, \delta)$-competitive algorithm exists for general networks, called **HIERARCHICAL PAGING**.
4.1 Simplified Constrained File Replication with Canonical Copy

A formal definition of the simplified constrained file replication with canonical copy problem follows. A set of processors $P$ are interconnected in a network with a symmetrical distance metric, i.e. $\delta(i, j) = \delta(j, i)$. Within this network, data is organized into equal-sized, indivisible blocks (files or pages). Each file has a canonical copy stored on one of the data storage nodes. These canonical copies cannot be moved, deleted, or modified in any way. Each data storage node has the ability to hold copies of $k$ additional items.

The algorithm processes requests of the following type:

- **read**($x, p$): Read file $x$ at processor $p$.

The cost of servicing a read request is zero if the file is already resident on the processor requesting the file. If the requesting processor does not already have the file, the cost is the distance between the requesting processor and the closest processor in the network holding a copy of the requested file, given by $\delta(i, j)$.

Upon servicing a read request, a processor may store a copy of the served page in its local storage, evicting a different page as needed.

This problem differs from the previously studied file replication problem in that copies of files may be deleted (but never their canonical copy). It differs from constrained file allocation in that because there is always a canonical copy of a file somewhere in the network, replicas in the additional storage may always be deleted at any time, without restriction that it must be replicated to another processor if it is the only copy in the network.

### 4.1.1 Uniform Network Topologies

In this section, a lower bound for simplified constrained file replication with canonical copy on uniform network topologies is presented, along with a deterministic algorithm realizing that lower bound.

The following theorem states the lower bound for simplified constrained file replication with canonical copy.

**Theorem 4.1.** Denote by $k$ the additional storage capacity of a processor. There is an infinite sequence of requests $\sigma$ such that

$$\text{ALG}(\sigma(n)) \geq k \cdot \text{OPT}(\sigma(n)).$$

(4.1)

for infinitely many values of $n$, where $\sigma(n)$ denotes the first $n$ requests of $\sigma$. 

18
Proof. The following cruel request sequence $\sigma$ forces bad behavior of any online algorithm $\text{Alg}$. The first request reads a file which is in neither of $\text{Alg}$’s or $\text{Opt}$’s local storage. The next $k - 1$ requests read files not currently in $\text{Alg}$’s local storage. In this sequence, the last $k$ requests all require $\text{Alg}$ to retrieve the file from a remote processor, whereas $\text{Opt}$ only has to transmit a locally missing file one time. This follows from the fact that $\text{Opt}$ retains all files needed for the last $k - 1$ accesses. To obtain the theorem, the sequence can be continued in the same fashion as many times as desired.

This theorem, along with its proof, follow directly from a reduction of the problem to regular paging. Simplified constrained file replication with canonical copy on uniform network topologies can be viewed as $|P|$ instances of the paging problem, with $k$ being the size of the fast memory of both $\text{Alg}$ and $\text{Opt}$. Conversely, longest forward distance is the optimal offline algorithm for simplified constrained file replication with canonical copy on uniform network topologies.

The algorithm least-recently-used (LRU) adapted to this problem, realizes the above described lower bound. For this problem, LRU works as follows: if the read request to a processor requests a file either stored in the processors canonical copy storage or in its additional storage, the file is served and no further action is taken (incurring a cost of zero). If the request is for a file not in the local storage of the processor, the processor reads the file from any processor in the network holding a copy of the file, incurring a cost of one. A copy of the file is stored locally in the additional storage of the processor. If this storage is currently full, the file for which the last access has been the earliest is removed, making room for the new file.

**Theorem 4.2.** For any sequence $\sigma$, \[
\text{LRU}(\sigma(n)) \leq k \cdot \text{opt}(\sigma(n)).
\] (4.2)

Proof. Because of the uniformity of the network, the problem may be viewed as $|P|$ instances of the paging problem. For each instance $i$, a single processor $p_i$ constitutes a fast memory of size $k$. The set of processors $P - \{p_i\}$ make up the slow memory. For read requests of files of which the request processing processor has a canonical copy, both LRU and Opt incur a cost of zero. For read requests of files for which the request processing processor does not have a canonical copy, the proof of regular paging applies to show that LRU may incur a cost of at most $k$ times the cost incurred by Opt.

4.1.2 Trees

Simplified constrained file replication with canonical copy on trees yields a distinct lower bound.

**Claim 4.3.** In a tree network topology with discrete distance metrics, there is an infinite sequence of requests $\sigma$ such that
\[ \text{ALG}(\sigma(n)) \geq k(d - 1) \cdot \text{Opt}(\sigma(n)) \quad (4.3) \]

for infinitely many values of \( n \), where \( \sigma(n) \) denotes the first \( n \) requests of \( \sigma \).

This claim will be proven in the analysis of the lower bound on arbitrary network topologies.

### 4.1.3 Arbitrary Network Topologies

For arbitrary networks, the diameter of the network affects the lower bound of the competitive ratio achievable by an online algorithm.

**Theorem 4.4.** Denote by \( k \) the additional storage capacity of a processor. Denote by \( d \) the diameter of the network. There is an infinite sequence of requests \( \sigma \) such that

\[ \text{ALG}(\sigma(n)) \geq kd \cdot \text{Opt}(\sigma(n)) \quad (4.4) \]

for infinitely many values of \( n \) and \( k > 1 \), where \( \sigma(n) \) denotes the first \( n \) requests of \( \sigma \).

**Proof.** The above lower bound can be achieved by the following request sequence \( \sigma \) of an adversary: the sequence consists of two distinct parts, a warm-up period used to get the algorithm in a state which can be exploited by the adversary, and the following requests which force the algorithm to perform badly.

Consider a network with processors \( \{p_s, p_t, p_u, p_v\} \subseteq P \). Define the distance metrics of

\[ \delta(s, t) = d, \quad \delta(t, \{u, v\}) = d \quad \text{and} \quad \delta(s, \{u, v\}) = 1. \]

**Warm-up period.** 2\( k \) read request to processor \( p_u \) of files existing exclusively on processor \( p_t \). The adversary selects a different set of \( k \) files to be in \( p_u \)'s additional storage such that the candidate holds no file that the adversary holds. Repeat read requests and selection of files stored in additional storage for \( p_v \).

Next make \( k \) read requests to processor \( p_s \) of files existing exclusively on processor \( p_t \) for the candidate, but existing either on \( p_u \) or \( p_v \) for the adversary. Both the candidate and the optimum algorithm may have the same contents in additional storage on processor \( p_s \).

**Exploitation.** The first read request for processor \( p_s \) is a file in neither \( \text{ALG} \)'s nor \( \text{Opt} \)'s local storage, but in \( \text{Opt} \)'s local storage of processor \( p_u \) or \( p_v \) and \( \text{ALG} \)'s local storage of processor \( p_t \). \( \text{ALG} \) incurs a cost of \( d \) and \( \text{Opt} \) pays 1 for transmitting the file from their respective sources.

The next \( k - 1 \) read requests for processor \( p_s \) are to files not in \( \text{ALG} \)'s local storage of \( p_s \) but existing exclusively on \( p_t \) (for \( \text{ALG} \)). This causes \( \text{ALG} \) to transmit the requested file from \( p_t \) for every request, incurring a cost of \( d(k - 1) \), totaling a cost of \( dk \) for \( k \) requests.

Because \( \text{Opt} \) retains all files needed for the last \( k - 1 \) accesses (by longest forward distance), it only needs to transmit a remote file one time for the entire series of \( k \) requests,
and the file can be transmitted from one of processors \( p_u \) or \( p_v \), incurring a cost of one.

This sequence can be repeated as many times as desired, giving the theorem. □

The same proof technique can be applied to prove claim 4.3, substituting the distance metrics above with a tree metrics, \( \delta(s, t) = d - 1 \), \( \delta(t, \{u, v\}) = d \) and \( \delta(s, \{u, v\}) = 1 \), the tree being of height 1 with \( s \) as the root.

**Theorem 4.5.** The least-recently-used algorithm (LRU) is \( kd \)-competitive for simplified constrained file replication with canonical copy on arbitrary networks.

**Proof.** In proving the above theorem, it suffices to show that for every processor \( p_i \in P \), the cost accrued at processor \( p_i \) for LRU is within a factor of \( kd \) of the cost accrued by \( \text{Opt} \) at processor \( p_i \).

In analyzing the cost at a single processor, the same analysis can be applied as with uniform networks, in that for every pair of processors \( p_i, p_j \in P \), \( \delta(i, j) \leq d \). Thus, no matter from which processor a file must be transmitted to the processor servicing the read request, a maximum cost of \( d \) accrues. Because for any sequence of \( k \) requests serviced by a single processor, \( \text{Opt} \) must transmit a file not in local storage at least once (incurring a cost of at least one) in order to make LRU have to transmit a file \( k \) times, LRU can perform no more than \( kd \) times worse than \( \text{Opt} \) on any single processor in the network. □

### 4.2 Constrained File Replication with Canonical Copy

In **constrained file replication with canonical copy**, the same network model is used as for simplified constrained file replication with canonical copy. Each file has a canonical copy stored on one of the data storage nodes. These canonical copies cannot be moved, deleted, or modified in any way. Each data storage node has the ability to hold copies of \( k \) additional items. The algorithm processes requests of the following type:

- **read** \( (x, p) \): Read file \( x \) at processor \( p \).

In addition to processing read requests, the algorithm may at any time

- **copy** a file from any source processor \( p_i \) to target processor \( p_j \) (respecting the capacity of additional storage on the target processor \( p_j \)), incurring a cost of \( \delta(i, j) \).

- **delete** any file from the additional storage of any processor, incurring a cost of zero.

Note that in this model, we explicitly allow a look-ahead of one. This means the algorithm may defer servicing a request until one or more replication or deletion actions have been completed. This is in contrast to look-ahead zero, where an algorithm may not look at the next request when making replication or deletion decisions.
4.2.1 Uniform Network Topologies

Theorem 4.6. Denote by \( k \) the additional storage capacity of a processor. There is an infinite sequence of requests \( \sigma \) such that

\[
\text{ALG}(\sigma(n)) \geq k \cdot \text{OPT}(\sigma(n))
\]  \( (4.5) \)

for infinitely many values of \( n \), where \( \sigma(n) \) denotes the first \( n \) requests of \( \sigma \).

Proof. On a uniform network, the algorithm’s additional capabilities over simplified constrained file replication with canonical copy, namely copying or deleting a file at any time during request processing, constitute premature paging. Because premature paging cannot possibly reduce the number of page faults and thus cannot improve the algorithm’s performance, these capabilities can be ignored in analysis. Thus the proof is identical to theorem 4.1. \( \square \)

Theorem 4.7. \( \text{LRU} \) is \( k \)-competitive on uniform network topologies.

Proof. This theorem follows implicitly from the previous proof (see proof of theorem 4.2). \( \square \)

4.2.2 Line with 3 nodes and \( k = 1 \)

This section analyzes the restricted case of the special topology of a line with 3 nodes, a distance metric equal to the number of hops between nodes and \( k = 1 \). At first glance, this problem appears too trivial to even consider for analysis, since a cache size of 1 with forced loading offers no choice of a file to evict at the node on which a file is being loaded.

But considering the distributed nature of paging, the middle node in the line introduces an interesting question when a file is loaded at a boundary node: should that file be cached on the middle node? This precise question is what makes analysis of problems with \( k > 1 \) difficult.

Theorem 4.8. On a line with three nodes and a distance metric equal to the number of hops between nodes,

\[
\text{ALG}(\sigma(n)) \geq 1.5 \cdot \text{OPT}(\sigma(n))
\]  \( (4.6) \)

for infinitely many values of \( n \) and \( k = 1 \), where \( \sigma(n) \) denotes the first \( n \) requests of \( \sigma \).

Proof. Consider the following adversary strategy which forces any algorithm to perform 1.5 times worse than \( \text{OPT} \).

Denote by \( p_1, p_2, p_3 \) the three connected processors in a line topology. Let \( f_i \in F \) be a file having a canonical copy on processor \( p_3 \).

Let \( S_i = (F \times F) \times (F \times F) \) be a tuple of tuples indicating the current resident sets of files stored at nodes \( p_1, p_2 \) in \( \text{ALG} \)'s configuration and the current resident sets of files stored at
nodes \( p_1, p_2 \) in \( \text{Opt} \)'s configuration respectively, after processing request \( \sigma_i \). The resident sets of files stored at node \( p_3 \) are irrelevant in this analysis.

Let \( S_0 = ([f_1, f_2], [f_1, f_2]) \). Request \( f_3 \) at \( p_1 \). Both \( \text{Opt} \) and \( \text{Alg} \) pay 2. A candidate algorithm has the choice of either ejecting \( f_2 \) at \( p_2 \) and caching \( f_3 \), or retaining \( f_2 \). The following case analysis covers both cases:

Case 1. \( \text{Alg} \) ejects \( f_2 \) at \( p_2 \) and caches \( f_3 \). The adversary retains \( f_2 \) at \( p_2 \). \( S_1 = ([f_3, f_3], [f_3, f_3]) \). The adversary requests \( f_2 \) at \( p_2 \). \( \text{Opt} \) pays zero and \( \text{Alg} \) pays 1. \( S_2 = ([f_3, f_2], [f_3, f_2]) \) with a repetition of state and a cost ratio of \( 3/2 = 1.5 \).

Case 2. \( \text{Alg} \) retains \( f_2 \) at \( p_2 \). The adversary ejects \( f_2 \) at \( p_2 \) and caches \( f_3 \). \( S_1 = ([f_3, f_3], [f_3, f_3]) \). The adversary requests \( f_3 \) at \( p_2 \). \( \text{Opt} \) pays zero and \( \text{Alg} \) pays 1. \( S_2 = ([f_3, f_3], [f_3, f_3]) \). The adversary requests \( f_1 \) at \( p_2 \). Again, a candidate algorithm may either retain or eject \( f_3 \) at node \( p_2 \). The adversary does the opposite, leading to \( S_3 \in \{([f_1, f_1], [f_1, f_3]), ([f_1, f_3], [f_1, f_1])\} \), both being a state repetition with cost ratio of \( 3/2 = 1.5 \).

**Theorem 4.9.** Any algorithm \( \text{Alg} \) in the set \( \mathcal{A} \) of possible, valid algorithms which is inactive between requests is 1.5 competitive.

**Proof.** When putting an algorithm \( \text{Alg} \) in competition against \( \text{Opt} \), the set of possible configurations can be categorized into states. Let \( S_I \) denote the state where \( \text{Alg} \) and \( \text{Opt} \) share an identical configuration, i.e. store exactly the same file at the same node. Let \( S_D \) denote the state where \( \text{Alg} \) and \( \text{Opt} \) have a different configuration and have a different file resident in cache on node \( p_2 \). Because of forced loading, both \( \text{Alg} \) and \( \text{Opt} \) must have an identical resident set on both end nodes \( p_1 \) and \( p_3 \). These two are the only possible different states, assuming a start configuration which is reachable from empty caches.

Consider state \( S_I \) where \( \text{Alg} \) and \( \text{Opt} \) share an identical configuration. By definition, both algorithms must pay the same amount for any request. To transition to \( S_D \), the adversary must request a file \( f_i \) at \( p_j \in \{p_1, p_3\} \) such that \( f_i \) is not resident on \( p_j \) and \( p_2 \), and take the opposite action of the candidate algorithm (retain or eject cached file at \( p_2 \)). For this transition, both algorithms pay 2.

Consider state \( S_D \) where \( \text{Alg} \) and \( \text{Opt} \) have a different configuration and have a different file resident in cache on node \( p_2 \). The only way \( \text{Opt} \) pays less than \( \text{Alg} \) in this state is by requesting its cached file on the middle node, in which case the algorithms transition to state \( S_I \), \( \text{Opt} \) pays zero and \( \text{Alg} \) incurs a cost of 1.

This leads to the following observation: cycling in \( S_I \) or in \( S_D \) cannot increase the competitive ratio as both algorithms incur the same cost. Creating a cycle by transitioning from \( S_I \) to \( S_D \) and back (or from \( S_D \) to \( S_I \) and back) incurs a minimum cost of 2 for \( \text{Opt} \) and 3 for \( \text{Alg} \), giving the ratio and theorem. □
4.2.3 Line with 3 nodes and $k = 2$

This section analyzes the restricted case of the special topology of a line with 3 nodes, a distance metric equal to the number of hops between nodes and $k = 2$.

Claim 4.10. On a line with 3 nodes and $k = 2$, there is an infinite sequence of requests $\sigma$ such that

$$\text{ALG}(\sigma(n)) \geq 3 \cdot \text{OPT}(\sigma(n))$$

(4.7)

for infinitely many values of $n$, where $\sigma(n)$ denotes the first $n$ requests of $\sigma$.

A computer program computing the game tree for the problem using brute-force was employed to determine this lower bound. The full source code listing for the program can be found in the appendix.

Algorithm M: When reading a file at a boundary node ($p_1$ or $p_3$) with a cost of 2, copy the file to the middle node $p_2$. This incurs no additional cost. When copying a file to the middle node, apply a first-in-first-out (FIFO) policy to evict a file if needed. When reading a file, apply a least-recently-used (LRU) policy for file eviction.

Theorem 4.11. For any request sequence $\sigma$ consisting of $n$ requests,

$$C_M(\sigma) \leq 3 \cdot C_{\text{OPT}}(\sigma) + \alpha.$$  

(4.8)

Proof. Consider both M and OPT processing a sequence of requests $\sigma$ concurrently. A potential function maps the state of the lists of M and OPT onto a real number $\Phi$. If servicing request $i$ has cost $c_i$, and changes the state of the algorithms from one having potential $\Phi_{i-1}$ to $\Phi_i$, then the amortized cost $a_i$ of servicing request $i$ is defined to be $c_i + \Phi_i - \Phi_{i-1}$.

The following potential function satisfies the following inequality for every configuration:

$$3 \cdot \Delta C_{\text{OPT}} - \Delta C_M \geq \Delta \Phi$$

(4.9)

The $\Delta$ indicates the change in the value of the parameter as a result of servicing one or more requests in $\sigma$. It remains to verify the above inequality and prove a constant lower bound $b$, such that independent of the request sequence for all $i$, $\Phi_i \geq b$.

Denote by $R_{i,\text{OPT}}, R_{i,M}$ the resident set of files currently stored at node $p_i$ in OPT’s and M’s configuration respectively. Let $S_i = R_{i,\text{OPT}} - R_{i,M}$, denoting the files in OPT’s but not in...
M’s resident set on any single node \( p_i \). Denote by \( \delta(f_j, i) \) the distance of the closest copy of file \( f_j \) to node \( p_i \) in M’s configuration.

Define the potential function:

\[
\Phi = \sum_{i=1}^{3} \sum_{s \in S_i} \delta(s, i) \tag{4.10}
\]

Consider an event where a file \( f_j \) is read from a boundary node \( i \) (\( p_1 \) or \( p_3 \)) and Opt has the file in local storage of \( p_i \) (\( f_j \in R_{i,\text{Opt}} \)). There are three cases, and in each inequality (4.9) holds:

- \( f_j \in R_{i,M} \): Both algorithms pay 0. There is no change in resident sets and \( \Delta \Phi = 0 \).
- \( f_j \notin R_{i,M} \cup R_{2,M} \): Opt pays 0. M pays 2. \( -2 \leq \Delta \Phi \leq -1 \).
- \( f_j \notin R_{i,M}, f_j \in R_{2,M} \): Opt pays 0. M pays 1. \( \Delta \Phi = -1 \).

Consider an event where a file \( f_j \) is read from a boundary node \( i \) (\( p_1 \) or \( p_3 \)) and Opt’s closest copy is in the middle node \( p_2 \) (\( f_j \notin R_{i,\text{Opt}}, f_j \in R_{2,\text{Opt}} \)). Again, there are three cases, and in each inequality (4.9) holds:

- \( f_j \notin R_{i,M} \cup R_{2,M} \): Opt pays 1. M pays 2. At the boundary node the resident set may change by one file, contributing a maximum of +2 to the potential delta. Because of M’s eviction policy at the middle node, there can only ever be a difference of maximum one file (unless Opt copies a file to the middle node in between requests, increasing its cost by 1). Because M stores the requested file on the middle node, \( R_{2,M} = R_{2,\text{Opt}} \) after this operation, contributing \(-1\) to the potential delta. Thus \( \Delta \Phi = 1 \).
- \( f_j \notin R_{i,M}, f_j \in R_{2,M} \): Opt pays 1. M pays 1. At the boundary node the resident set may change by one file each, so \( 0 \leq \Delta \Phi \leq 2 \).
- \( f_j \in R_{i,M} \): Opt pays \( \leq 1 \). M pays 0. \( 0 \leq \Delta \Phi \leq 3 \).

Consider an event where a file \( f_j \) is read from a boundary node \( i \) (\( p_1 \) or \( p_3 \)) and Opt’s closest copy is in the opposite boundary node (\( f_j \notin R_{i,\text{Opt}} \cup R_{2,\text{Opt}} \)). There are three cases, and in each inequality (4.9) holds:

- \( f_j \notin R_{i,M} \cup R_{2,M} \): Both algorithms pay 2. At the boundary node and the middle node the resident set may change by one file each, so \( 0 \leq \Delta \Phi \leq 3 \).
- \( f_j \notin R_{i,M}, f_j \in R_{2,M} \): Opt pays 2. M pays \( \leq 1 \). At the boundary node and the middle node the resident set may change by one file each, so \( 0 \leq \Delta \Phi \leq 3 \).
• $f_j \in R_{2,M}$: \texttt{Opt} pays 2. M pays 0. At the boundary node and the middle node the resident set may change by one file each, so $0 \leq \Delta \Phi \leq 3$.

Next, consider an event where file $f_j$ is read from the middle node $p_2$. There are four cases, and in each inequality (4.9) holds:

• $f_j \notin R_{2,\text{Opt}}, f_j \notin R_{2,M}$: Both algorithms pay 1. There may be a change in resident set by one file, so $0 \leq \Delta \Phi \leq 1$.

• $f_j \notin R_{2,\text{Opt}}, f_j \in R_{2,M}$: \texttt{Opt} pays 1. M pays $\leq 0$. There may be a change in resident set by one file, so $0 \leq \Delta \Phi \leq 1$.

• $f_j \in R_{2,\text{Opt}}, f_j \notin R_{2,M}$: \texttt{Opt} pays 0. M pays $\leq 1$.

• $f_j \in R_{2,\text{Opt}}, f_j \in R_{2,M}$: Both algorithms pay 0. There is no change in resident sets and $\Delta \Phi = 0$.

Consider an event where \texttt{Opt} spontaneously copies a file from one node to another. \texttt{Opt} pays 1 or 2, whereas M incurs no cost. $-2 \leq \Delta \Phi \leq 2$.

This completes the case analysis.

It remains to show that for some $b$, independent of the request sequence for all $i$, $\Phi_i \geq b$. This is trivial, since the potential function reaches its lowest value of 0 when \texttt{Alg} and \texttt{Opt} have the same configuration (cache contents) and cannot go lower.

This gives the theorem. \hfill $\Box$

4.2.4 Arbitrary Network Topologies

The question arises if the lower bound of the competitive ratio on constrained file replication with canonical copy on general networks can be improved from the bound on uniform networks ($k$) to $dk$ which is the lower bound of simplified constrained file replication with canonical copy on general networks.

Because the candidate algorithm may replicate files before and after any request in the request sequence $\sigma$, the proof of the lower bound on simplified constrained file replication with canonical copy for arbitrary network topologies does not apply here.

**Lemma 4.12.** The lower bound of the competitive ratio on constrained file replication with canonical copy on general networks is at least $k$.

This follows from the fact that the lower bound on uniform networks is $k$.

**Conjecture 4.13.** The lower bound of the competitive ratio on constrained file replication with canonical copy on general networks is at most $kd$. 

26
4.2.5 Analysis of Some Candidate Algorithms

**Algorithm Zealous**: The algorithm copies a requested file from the closest processor holding a copy of the file to every processor on the shortest path \((p_i, p_{i+1}, \ldots, p_{j-1}, p_j)\) between the requesting processor \(p_i\) and processor holding a copy of the file \(p_j\). In other words, the algorithm copies the file from processor \(p_j\) to processor \(p_{j-1}\), from processor \(p_{j-1}\) to processor \(p_{j-2}\) and so on until the file finally reaches its destination at processor \(p_i\). Because the path traversed by the file is the shortest path between \(p_i\) and \(p_j\), the algorithm pays \(\delta(i, j)\). In deciding which file to evict from additional storage, the algorithm uses a least-recently-used policy.

Intuition might lead one to believe this algorithm can achieve a good competitive ratio. Contrary to this, however, Zealous is not competitive.

**Theorem 4.14.** Zealous is not competitive on a tree of height \(\geq 3\).

**Proof.** Consider a tree of height 3 with 4 nodes (i.e. a straight line) consisting of processors \(\{p_1, p_2, p_3, p_4\}\). Let both Zealous and Opt be in the same state, with additional storage empty on all processors. The following cruel request sequence \(\sigma\) breaks Zealous’ competitiveness.

**Warm-up period.** Request at processor \(p_3\) \(k\) files that are stored at \(p_1\) but not \(p_2\). Zealous stores the files on both processors \(p_2\) and \(p_3\). Request at processor \(p_2\) \(k\) files that are stored at \(p_4\) but not \(p_3\). Zealous stores the files on both processors \(p_2\) and \(p_3\), evicting any files that were originally stored at processor \(p_1\). Opt stores the requested files only on the target processors and not on processors on the shortest path. Thus, Opt has its additional storage on processor \(p_2\) filled with files copied from processor \(p_4\), and its additional storage on processor \(p_3\) filled with files copied from processor \(p_1\).

**Exploitation.** Request at processor \(p_3\) a file resident on both processor \(p_1\) (in the configuration of both algorithms) and processor \(p_3\) in Opt’s configuration, but not Zealous’ configuration. Opt pays zero, as the file is already collocated with the read request. Zealous must read the file from processor \(p_1\) and evict a file on both processors \(p_2\) and \(p_3\). Zealous pays \(\delta(1, 3)\).

But now, Opt has at least one file from processor \(p_4\) replicated to processor \(p_2\) that Zealous does not have replicated (the file that was just evicted in the previous step). Request this file at \(p_2\). Again, Opt pays zero, having the file resident at request location. Zealous must read the file from processor \(p_4\) and evict a file on both processors \(p_2\) and \(p_3\), paying \(\delta(2, 4)\). The file evicted at processor \(p_3\) is still resident on that processor in Opt’s configuration, thus this sequence can be repeated indefinitely, forcing Zealous to continue paying for operations that are free for Opt. Hence, Zealous is not competitive.

**Algorithm Closest Copy**: After servicing a read request, the algorithm decides whether or not to store the read file in the processors additional storage,
and if so, what file to evict from additional storage. To decide which files get stored or evicted, the file which has the closest copy on another processor in the network gets evicted (or if it is the file previously read, does not get stored in additional storage).

This eviction policy takes into account the global state of the system, in that it works towards minimizing the average access cost of any file for any single processor. This however, reduces the algorithm’s competitiveness.

**Theorem 4.15.** Closest Copy is not competitive on non-uniform networks with at least 4 nodes.

*Proof.* Choose processors \( \{p_1, p_2, p_3, p_4\} \subseteq P \) such that \( \delta(p_1, p_2) \leq \delta(p_1, p_3) < \delta(p_1, p_4) \). Let \( c_1 \) be a file with canonical copy at processor \( p_1 \). Define \( k = 2 \).

A cruel request sequence \( \sigma \) can be constructed in the following way. Read \( c_4 \) at \( p_1 \). Closest Copy will copy this file to additional storage. Read \( c_2 \) at \( p_1 \), followed by \( c_3 \) at \( p_1 \). At this point the algorithm must decide whether to store \( c_3 \) and evict \( c_2 \), or to keep \( c_2 \). The adversary now repeatedly reads at \( p_1 \) the file in \( \{c_2, c_3\} \) which is not in additional storage at \( p_1 \).

In this request sequence, \( c_4 \) can never be evicted because it is the furthest canonical copy. Thus, the candidate algorithm is required to pay for every single read request. Opt on the other hand never puts \( c_4 \) in additional storage but instead keeps the files \( \{c_2, c_3\} \) replicated at \( p_1 \). Thus Opt pays a constant cost, in contrast to Closest Copy, proving that the candidate algorithm is not competitive.

\( \square \)

### 4.2.6 NP-Completeness and Constrained File Replication With Canonical Copy

The problem of finding an optimum solution to serving a request sequence \( \sigma \) for constrained file replication with canonical copy can be shown to be NP-complete.

In the following proof, the following formal decision-problem definitions will be used.

- **Problem:** Steiner Tree
  
  *Input:* Weighted graph \( G = (V, E) \), a subset of vertices \( T \subseteq V \), positive integer \( c \).
  
  *Output:* Does there exist a tree connecting all vertices in \( T \) such that the total weight of the tree \( \leq c \)?

- **Problem:** Constrained File Replication With Canonical Copy
  
  *Input:* Weighted graph \( G = (V, E) \), set of files \( F \), set of canonical copies \( I \) (\( I_i \subseteq F \)), request sequence \( \sigma \) (\( \sigma_i = (f_j, v_l) \), \( f_j \in F, v_l \in V \)), positive integers \( c, k \).
  
  *Output:* Does there exist a sequence of responses such that the cost of servicing request sequence \( \sigma \) on graph \( G \) with additional storage \( k \) at each vertex is \( \leq c \)?
The Steiner Tree problem is NP-Complete, in fact it is one of Karp’s original 21 NP-Complete problems in his seminal 1972 paper titled “Reducibility Among Combinatorial Problems” [Kar10].

The Steiner Tree problem can be reduced to constrained file replication with canonical copy as follows.

**Lemma 4.16.** Constrained file replication with canonical copy ∈ NP.

*Proof.* There exists an efficient certifier for constrained file replication with canonical copy. Take the response sequence $\lambda$ (of read, copy and delete actions) and find a sequence $\lambda'$ such that for all $\lambda'_i \in \lambda'$ and for all $\sigma_i \in \sigma$, $\lambda'_i$ is a read action servicing request $\sigma_i$. This mapping can be done in polynomial time using dynamic programming. Having verified that every read request is serviced by $\lambda$, verify that $\sum_{i \in [|\lambda|]} \text{cost}(\lambda_i) \leq c$. This obviously can be done in polynomial time. □

Next, it will be shown that constrained file replication with canonical copy is NP-hard. This is done by reducing the Steiner Tree problem to constrained file replication with canonical copy.

**Lemma 4.17.** An instance of the Steiner Tree problem can be converted to an instance of the constrained file replication with canonical copy problem.

*Proof.* Given an instance of the Steiner Tree problem, the following conversion can be applied to construct an instance of the constrained file replication with canonical copy problem. Given input $G = (V, E), T, c$ to the Steiner Tree instance, let $G$ and $c$ be identical for the constrained file replication with canonical copy problem input. Let $k \geq 2$, $F = \{1\}$, $I_1 = F_1$ and $I_j = \emptyset$ for all $j > 1$. Further, define request sequence $\sigma = \{(T_1, F_1), (T_2, F_1), \ldots, (T_n, F_1) : n = |T|\}$. This conversion can be done in polynomial time. □

**Lemma 4.18.** Using the above conversion, there exists a solution to the Steiner Tree problem if and only if there exists a solution to the constrained file replication with canonical copy problem.

*Proof.* *(if)* This follows from the following observation. Since only one file is being read ($F_1$) and $k \geq 2$, a file may be copied to any node in $G$ and never evicted. Thus the minimal cost of reading a file at every node in $T$ is equal to its minimal Steiner Tree by definition, because the file must be copied from one node to all other nodes in $T$ following a path that is at least the distance of the minimal Steiner Tree. It follows that if there exists a solution to the constrained file replication with canonical copy instance, a solution to the Steiner Tree instance must exist.

*(only if)* Conversely, if there is no way to replicate the file to all nodes in $T$ with cost $c$ meaning there is no solution to the constrained file replication with canonical copy problem,
this implicitly means there is no minimal Steiner Tree with cost $c$, so there is no solution to the minimal Steiner Tree instance.

\[ \square \]

**Theorem 4.19.** Constrained File Replication With Canonical Copy is NP-complete.

**Proof.** This follows from the above lemmas. \[ \square \]

### 4.3 Conclusion

This thesis analyzed simplified constrained file replication with canonical copy and presented optimal online algorithms matching the lower bounds of $k$ for uniform networks, $k(d - 1)$ for trees, and $kd$ arbitrary network topologies. Additionally, restricted cases of constrained file replication with canonical copy were analyzed. For uniform networks, optimal online algorithms with a competitive ratio of $k$ were given. On lines of three nodes with $k = 1$ and $k = 2$, optimal online algorithms with a competitive ratio of 1.5 and 3 were presented.

Constrained file replication with canonical copy has been quite intractable in analysis and has given the author more problems than anticipated. For the simplified version, where a candidate algorithm may only read a page at the node processing the request and not store the page at any other node, tight bounds were not hard to find. However, given the freedom of action of an algorithm in the non-simplified variation, the problem very quickly takes on an intractable complexity.

Consider the restricted case of a bus network topology with only three nodes and a cache size of 2. Even in this simple case, finding a good, non-trivial lower bound is quite hard. A computer program computing the game tree for the problem using brute-force was employed. This program ran to depth 13 to find a lower bound of 3, which is a tight bound. That even a very small problem instance requires computation to such depth is quite surprising - yet given that the problem is NP-complete maybe should be expected.

It remains an open question whether the lower bound of $kd$ for simplified constrained file allocation with canonical copy drops for the non-simplified version even in the non-restricted case on arbitrary networks. Intuition suggests that this might be the case (since a candidate algorithm may cache pages intelligently to evenly distribute pages towards a global minimum of access cost of any page at any node).

While network topologies of uniform graphs and lines with three nodes and restricted $k$ have been analyzed, future work may include the analyzing the problem on general lines, rings, stars and trees.

Given that no good ($< kd$) competitive algorithms are known for constrained file allocation with canonical copy, the use of heuristics may yield better results and the development of algorithms using heuristics may be worth pursuing.
References


Appendix B

Lower bound search code

```java
package edu.csun.comp698;

import java.util.logging.Logger;

/**
 * This class implements a brute force search for the competitive ratio
 * of constrained file replication with canonical copy on a line of three
 * nodes. Alpha and beta parameters are hard-coded for a single run.
 */
public class FindLowerBoundLine3_forThesis {

    private final static Logger logger = Logger.getLogger(FindLowerBoundLine3_forThesis.class.getName());

    private enum File {
        F1, F2, F3, F4, F5
    }

    private enum Node {
        P1, P2
    }

    static class Request {
        private final File file;
        private final Node node;

        Request(File file, Node node) {
            this.file = file;
            this.node = node;
        }

        @Override
        public String toString() {
            return file + "@" + node;
        }
    }

    private static class ComputeOptimum {

        private final Request[] requests;
        private final int numberOfRequests;

        ComputeOptimum(Request[] requests, int numberOfRequests) {
            this.requests = requests;
            this.numberOfRequests = numberOfRequests;
        }

        int compute() {
            return compute(File.F1, File.F2, File.F3, File.F4, 0);
        }
    }
```
private int compute(File p1_1, File p1_2, File p2_1, File p2_2, int request) {
    if (request >= numberOfRequests) {
        // terminate here
        int cost = 0;
        if (p1_1 != File.F1 && p2_1 != File.F1)
            cost += 2;
        if (p1_1 != File.F2 && p2_1 != File.F2)
            cost += 2;
        if (p2_1 != File.F3 && p2_2 != File.F3)
            cost += 1;
        if (p2_1 != File.F4 && p2_2 != File.F4)
            cost += 1;
        return cost;
    }

    Request r = requests[request];
    if (r.node == Node.P1) {
        if (p1_1 == r.file || p1_2 == r.file) {
            return compute(p1_1, p1_2, p2_1, p2_2, request + 1);
        }

        boolean cachedAtNode2 = (p2_1 == r.file || p2_2 == r.file);

        // eject 1
        int cost = compute(r.file, p1_2, p2_1, p2_2, request + 1);
        if (!cachedAtNode2) {
            // eject 1, 3
            cost = Math.min(cost,
                            compute(r.file, p1_2, r.file, p2_2, request + 1));
            // eject 1, 4
            cost = Math.min(cost,
                            compute(r.file, p1_2, p2_1, r.file, request + 1));
        }

        // eject 2
        cost = Math.min(cost,
                        compute(p1_1, r.file, p2_1, p2_2, request + 1));
        if (!cachedAtNode2) {
            // eject 2, 3
            cost = Math.min(cost,
                            compute(p1_1, r.file, r.file, p2_2, request + 1));
            // eject 2, 4
            cost = Math.min(cost,
                            compute(p1_1, r.file, p2_1, r.file, request + 1));
        }

        int d = cachedAtNode2 ? 1 : 2;
        return d + cost;
    } else if (r.node == Node.P2) {
        if (p2_1 == r.file || p2_2 == r.file) {
            return compute(p1_1, p1_2, p2_1, p2_2, request + 1);
        }
    }
}
/ eject 3
int cost = compute(p1_1, p1_2, r.file, p2_2, request + 1);
// eject 4
cost = Math.min(cost,
        compute(p1_1, p1_2, p2_1, r.file, request + 1));
return 1 + cost;
} else
    throw new AssertionError();
}

static class Competition {
    private final float alpha = 2f;
    private final float beta = 3f;
    private final Request[] r;

    Competition(int depth) {
        r = new Request[depth];
    }

    int computeOptimum(int depth) {
        return new ComputeOptimum(r, r.length - depth).compute();
    }

    float minMax(int depth) {
        logger.info("Matching candidate against adversary...");
        return maxAdversaryMoves(depth, 0, File.F1, File.F2, File.F3, File.F4);
    }

    // candidate picks the server to move to
    float minCandidateMoves(int depth, int cost, Request r, File p1_1,
                            File p1_2, File p2_1, File p2_2) {
        float ratio = Float.POSITIVE_INFINITY;
        int newCost = cost + (cachedAtNode2 ? 1 : 2);

        if (r.node == Node.P1) {
            if (p1_1 == r.file || p1_2 == r.file)
                throw new AssertionError();

            boolean cachedAtNode2 = (p2_1 == r.file || p2_2 == r.file);
            int newCost = cost + (cachedAtNode2 ? 1 : 2);

            // eject 1
            ratio = Math.min(
                ratio,
                maxAdversaryMoves(depth, newCost, r.file, p1_2, p2_1,
                                   p2_2));

            if (ratio <= alpha)
                return ratio;
        }

    }
```java
if (!cachedAtNode2) {
    // eject 1, 3
    ratio = Math.min(
        ratio,
        maxAdversaryMoves(depth, newCost, r.file, p1_2,
        r.file, p2_2));
    if (ratio <= alpha)
        return ratio;

    // eject 1, 4
    ratio = Math.min(
        ratio,
        maxAdversaryMoves(depth, newCost, r.file, p1_2,
        p2_1, r.file));
    if (ratio <= alpha)
        return ratio;
}

if (r.node == Node.P2) {
    int newCost = cost + 1;

    // eject 3
    ratio = Math.min(
```
ratio,
maxAdversaryMoves(depth, newCost, p1_1, p1_2, r.file, p2_2));
if (ratio <= alpha)
    return ratio;

// eject 4
ratio = Math.min(
    ratio,
    maxAdversaryMoves(depth, newCost, p1_1, p1_2, p2_1, r.file));
if (ratio <= alpha)
    return ratio;
if (ratio > 3) {
} else {
    throw new AssertionError();
}

// adversary picks a request
float maxAdversaryMoves(int depth, int cost, File p1_1, File p1_2,
    File p2_1, File p2_2) {
    float r = Float.NEGATIVE_INFINITY;

    // handle leaf nodes
    int totalCost = cost;
    if (p1_1 != File.F1 && p1_2 != File.F1)
        totalCost += 2;
    if (p1_1 != File.F2 && p1_2 != File.F2)
        totalCost += 2;
    if (p2_1 != File.F3 && p2_2 != File.F3)
        totalCost += 1;
    if (p2_1 != File.F4 && p2_2 != File.F4)
        totalCost += 1;
    int optimalCost = computeOptimum(depth);
    float ratio;
    if (optimalCost == 0) {
        if (totalCost == 0) {
            ratio = 1;
        } else {
            ratio = Float.POSITIVE_INFINITY;
        }
    } else {
        ratio = ((float) totalCost) / optimalCost;
    }
    r = Math.max(r, ratio);

    if (depth > 0) {
        boolean at2 = false;
        boolean notAt2 = false;
        for (File f : File.values()) {
if (p1_1 != f && p1_2 != f) {
    if (p2_1 != f && p2_2 != f) {
        if (!notAt2) {
            notAt2 = true;
            r = Math.max(
                r,
                minCandidateMoves(depth - 1, cost,
                                 new Request(f, Node.P1), p1_1,
                                             p1_2, p2_1, p2_2));
            if (r >= beta)
                return r;
        }
    }
    else {
        if (!at2) {
            at2 = true;
            r = Math.max(
                r,
                minCandidateMoves(depth - 1, cost,
                                 new Request(f, Node.P1), p1_1,
                                             p1_2, p2_1, p2_2));
            if (r >= beta)
                return r;
        }
    }
}

for (File f : File.values()) {
    if (p2_1 != f && p2_2 != f) {
        boolean at1 = false;
        boolean notAt1 = false;
        if (p1_1 != f && p1_2 != f) {
            if (!notAt1) {
                notAt1 = true;
                r = Math.max(
                    r,
                    minCandidateMoves(depth - 1, cost,
                                     new Request(f, Node.P2), p1_1,
                                             p1_2, p2_1, p2_2));
                if (r >= beta)
                    return r;
            }
        }
    }
    else {
        if (!at1) {
            at1 = true;
            r = Math.max(
                r,
                minCandidateMoves(depth - 1, cost,
                                 new Request(f, Node.P2), p1_1,
                                             p1_2, p2_1, p2_2));
            if (r >= beta)
                return r;
        }
    }
}
return r;