Does LRU always outperform FIFO?

A mathematical and simulation-based approach to identify better performing caching algorithms

by

V.N.W. Terlouw

to obtain the degree of Bachelor of Science
to be defended publicly on Tuesday November 25, 2025 at 16:00.

Student number: 5615097
Project duration: September, 2025 – November, 2025
Thesis committee: Dr. C. E. Groenland, TU Delft, supervisor
Dr. T. W. C. Vroegrijk, TU Delft, assessment committee member

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Caching plays a crucial role in keeping the internet accessible. Many algorithms exist to determine which page should be evicted from the cache, but most analyses of these algorithms only focus on the competitive ratio. The competitive ratio, however, cannot identify which algorithm performs well in practice and which only performs well on paper.

The goal of this research is to study how the performance of deterministic and randomized caching algorithms can be compared, and how the algorithms behave on differently structured request sequences. A purely mathematical analysis is combined with simulations of uniformly random, recency-based, and favorite element request sequences.

Sleator and Tarjan [6] claim that the FIFO and LRU algorithms are competitive and provide a proof for the LRU algorithm. This research provides a proof for the competitiveness of the FIFO algorithm.

In addition, Panagiotou and Souza [4] use the characteristic vector to determine the number of cache misses for LRU. This research shows that for the FIFO algorithm, both an upper and lower bound can be obtained by using the characteristic vector, with p the number of distinct pages and ℓ the number of distinct pages between two requests σ_i and σ_j:

$$p + \sum_{\ell \geq 2k-1} C_\ell(\sigma) \leq FIFO(\sigma) \leq p + \sum_{\ell > 0} C_\ell(\sigma).$$

Finally, simulations show that different request structures lead to different well-performing algorithms. The LFU algorithm, although not competitive, performs best on the favorite element request sequences, while LRU performs best on random and recency-based request sequences.

More broadly, the results show that a higher cache size yields a higher ratio of the number of cache misses for the algorithms and optimal solution, while a larger universe leads to a lower ratio.

Lay summary

Caching keeps the internet and all of its content accessible within seconds. Instead of loading every page when it is requested, some pages are kept in a small, fast memory, the cache. However, whenever the cache is full, a page should be removed from the cache to make space for a new page. This raises an important question: which page should be removed first?

There are many algorithms that answer this question, such as the First In, First Out (FIFO) algorithm, Least Recently Used (LRU) algorithm, or simply removing a random page (UNI). With so many options, it is natural to ask which algorithm performs best. Mathematicians and theoretical computer scientists try to answer this question by analyzing request sequences and the number of cache misses, the number of times a page was requested but was not already in the cache.

In this research, the competitive ratio, a worst-case analysis of algorithms, is used to classify and compare caching algorithms. In addition, the characteristic vector, which represents the structure of a request sequence, is used to compute upper and lower bounds for the number of cache misses for the FIFO and LRU algorithms.

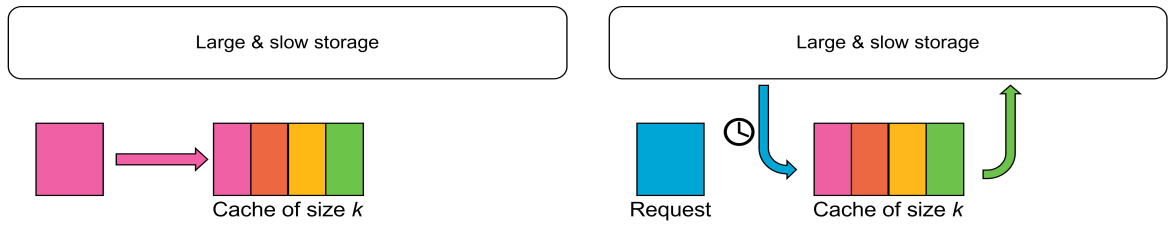
Besides a mathematical analysis, this research also uses a simulation-based approach. All sequences with 5 distinct elements of length 10 are generated, and the performance of each algorithm is evaluated on this complete set of sequences. These results show that the LRU algorithm performs very well.

However, real request patterns are not completely random, so this research introduces two types of structured request patterns: a recency-based request pattern and a favorite element request pattern. The performance of the algorithms is compared for these request patterns with different numbers of distinct pages and cache sizes. As expected, different algorithms perform better for different request patterns. The Least Frequently Used (LFU) algorithm performs well for the favorite element request pattern, while the LRU algorithm performs well for the recency-based request pattern.

Contents

1 Introduction

The rapid and endless growth of the content that can be found on the internet highlights the importance of caching. Caching keeps the internet and all of its content accessible within seconds. Instead of retrieving every single requested page directly from the internet, a storage system with a limited space of k pages is introduced, known as a cache. Whenever a user requests a page, it is checked if the page is already in the cache. If the page is in the cache, the page can be accessed immediately without any delays, as shown in Figure 1.1a. If the page is not in the cache, known as a cache miss, the page must be loaded into the cache, which takes additional time. Due to the limited capacity of the cache, whenever a new page is loaded into the cache, another page must be removed from the cache, as shown in Figure 1.1b.



(a) Behavior of the cache when the requested page is in the cache

(b) Behavior of the cache when the requested page is not in the cache

Figure 1.1: Behavior of the cache with requested pages

To decide which page should be removed from the cache whenever a cache miss occurs, an eviction strategy or algorithm is used. An eviction strategy consists of a set of rules or instructions that determine which page should be removed from the cache. There are several types of eviction strategies; it is possible to select the page completely at random or use a deterministic algorithm. Deterministic algorithms are characterized by the fact that they produce the same output for the same input every time. Finally, there exist algorithms that use predictors to determine which page is the least likely to be requested again and evict that page.

Whenever a cache miss occurs, extra time is required to load the new request into the cache, and thus, the goal is to minimize the number of cache misses. If the entire sequence of requests is known in advance, an optimal solution that minimizes the number of cache misses exists. However, when surfing the internet, it is not possible to know future requests, which makes it impossible to determine the optimal solution while requests are coming in. The goal, therefore, is to find an eviction strategy that performs well, despite this uncertainty. In order to determine how well algorithms perform, the competitive ratio was introduced. The competitive ratio is the maximum, for all possible input sequences, of the number of cache misses of the eviction strategy, divided by the minimum number of cache misses as determined by the optimal algorithm. For randomized algorithms, the number of cache misses of the eviction strategy equals the expectation of the cost. While it is nice to have something to compare the algorithm with, all deterministic algorithms have a competitive ratio of at least k, the size of the

cache, under the assumption that the requested pages are completely random. Thus, the competitive ratio cannot really be used to compare deterministic algorithms. Furthermore, the fact that an eviction strategy has a low competitive ratio is no guarantee for how well an eviction strategy performs in practice. Finally, in reality, surfing behavior is not completely random. Users may have preferred pages that they will visit more frequently than others.

As a result, the objective of this research is (1) to compare deterministic and randomized algorithms using a different measure than the competitive ratio and (2) to investigate the impact of differently structured request sequences. These objectives lead to the following research question:

"How do deterministic and randomized algorithms perform with differently structured request sequences in the caching problem, and how can these algorithms be compared to one another?"

In order to answer the research question, four sub-questions are needed. These sub-questions are:

1. Which metric can be used to compare different algorithms?
2. Which characteristics does this metric have for various algorithms?
3. How do algorithms compare on a complete set of sequences?
4. How do algorithms compare for various types of sequences?

The remainder of this research is organized in the following manner; Chapter 2 lays the theoretical foundation to answer all of these questions. The caching problem is formalized, and the concept of the competitive ratio is further explored. Additionally, different types of eviction strategies are introduced.

Chapter 3 builds on the theoretical foundation of Chapter 2 and shows that the upper bound of the competitive ratio for the LRU and FIFO algorithms is equal to the cache size, k. This chapter also explores the number of cache misses for both the FIFO and LRU algorithms using the characteristic vector, introduced by Panagiotou and Souza [4].

In Chapter 4, all algorithms are compared for a complete set of sequences. Specifically, the ratio between the algorithm and optimal solution is computed for all possible sequences with length 10 with 5 distinct pages.

Chapter 5 elaborates on this analysis. The ratio between the algorithms and the optimal solution is analyzed for longer sequences with more distinct pages. Furthermore, in this chapter, not only are randomized sequences analyzed, but also sequences that include preferred elements and sequences based on the recency of elements.

Finally, Chapter 6 combines all results and discusses both the implications and limitations of this research.

2

Theoretical framework

In this chapter, a theoretical framework is established to be able to compare the different eviction strategies. First, the caching problem is formalized. The competitive ratio and optimal eviction strategy will also be introduced. Secondly, a classification for eviction strategies is introduced. Furthermore, for the deterministic eviction strategies, it is shown that the competitive ratio is at least equal to k. Finally, a few different eviction strategies, based on the previous classification, are introduced for further analysis within this research.

2.1. Formalization of the caching problem

The caching problem arises within memory systems whenever a user requests a page σ_i. When the request σ_i has been satisfied, the user can request a new page, σ_{i+1}. All pages that the user requests form a sequence, $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_t)$. All possible pages that the user can request are stored in a large but slow storage U, containing n pages. It is also possible for pages to be stored in a smaller but faster storage, also known as the cache, C, with size k. The elements within the cache can be denoted as $c_1, c_2, \dots, c_k$. Whenever a user requests a page, it must be retrieved from the cache. If the page is already in the cache C, there is no cost. When a requested page is not in the cache, it must be loaded into the cache from the slower storage, known as the universe, U, this is also known as a cache miss. In most cases, the cache already contains k pages and thus in order to load a new element into the cache, another element needs to be removed from the cache. An eviction strategy decides which element should be removed from the cache in this case.

When browsing the internet, a request σ_i must be satisfied before another request σ_{i+1} can be satisfied. In a special case, known as the offline situation, the requests are known beforehand. It is then possible to determine a perfect eviction strategy in which the number of cache misses is minimized, also known as the optimal eviction strategy. This optimal eviction strategy can be used to compare algorithms. The competitive ratio is defined differently for deterministic and random algorithms.

Definition 1 (Optimal offline algorithm)**.** *The optimal algorithm minimizes the number of cache misses for any request sequence σ. The optimal offline algorithm evicts the element for which the next request is the furthest in the future.*

Definition 2 (Competitive ratio deterministic algorithms)**.** *The competitive ratio of a deterministic eviction strategy equals*

$$\max_{\sigma} \frac{ALG(\sigma)}{OPT(\sigma)} \tag{2.1}$$

where $ALG(\sigma)$ is the number of cache misses caused by the imposed eviction strategy, and $OPT(\sigma)$ is the number of cache misses with the optimal eviction strategy.

Definition 3 (Competitive ratio randomized algorithms)**.** *The competitive ratio of a randomized eviction strategy equals*

$$\max_{\sigma} \frac{\mathbb{E}[ALG(\sigma)]}{OPT(\sigma)} \tag{2.2}$$

Where $\mathbb{E}[ALG(\sigma)]$ is the expected cost of the algorithm for a sequence σ and $OPT(\sigma)$ the cost of the optimal eviction strategy.

2.2. Classification of eviction strategies

Eviction strategies can mainly be categorized as deterministic and randomized strategies. A deterministic strategy will always produce the same output for a certain input. Randomized strategies can produce varying outputs for the same input. Most strategies are key-based, meaning that items are compared using a key, and the item that scores the worst on this key is evicted from the cache. Strategies can also be categorized based on their key, or combined keys.

2.3. Minimal competitive ratio of deterministic algorithms

The following theorem and proof are based on lecture notes from Gupta [3].

Theorem 4. *The competitive ratio of any deterministic algorithm for the caching problem is at least k.*

Proof. Consider a large storage U containing $k+1$ elements and a cache C with size k. It is possible to create a sequence, $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_t)$, with t requests, using the chosen eviction strategy, for which the next requested page will not be in the cache. Hence, for this specific σ, the number of cache misses, $ALG(\sigma)$, equals the number of requests t. If the sequence σ is known beforehand, the optimal eviction strategy can be applied, always evicting the item the furthest in the future. Because the large storage U only consists of $k+1$ pages, removing the request furthest in the future ensures that at least the next $k-1$ requests can be satisfied by the pages already in the cache. Thus, a cache miss occurs at most every $1/k$ requests. So $OPT(\sigma) \leq t/k$. This yields the competitive ratio of at least k. □

2.4. Introduction of algorithms

Using the classification introduced by Podlipnig and Böszörmenyi [5] and the algorithms that Belady [2] and Gupta [3] talk about, several algorithms have been chosen for further analysis. These algorithms are (1) First In First Out, (2) Least Recently Used, (3) Least Frequently Used, (4) Segmented Least Recently Used, and (5) Uniform Randomized. The final algorithm, (6) Recency-Based Randomized, was designed specifically for this research.

2.4.1. FIFO - First In First Out

The First In First Out eviction strategy (FIFO) is the most basic algorithm that can be applied to the caching problem. It does not consider any information about the previous requests, except for the order in which they entered the cache. Whenever a request is already in the cache, nothing happens. If a request is not in the cache, the element that entered the cache first will be removed.
The FIFO algorithm is deterministic, but it is also a competitive algorithm, for which a proof is provided in Chapter 3. For a competitive algorithm, the lower bound of the competitive ratio equals the upper bound. So the competitive ratio for FIFO equals k.

2.4.2. LRU - Least Recently Used

The Least Recently Used strategy (LRU) is based on the idea that pages recently requested are more likely to be requested. It is a competitive, which will be proven in Chapter 3, and deterministic algorithm. The LRU eviction strategy works with a sorted list, also known as a queue. Whenever a page is requested, it is put at the end of the queue, even if the page was already in the cache. If the cache is full, the page at the front of the queue is removed.

2.4.3. LFU - Least Frequently Used

The Least Frequently Used algorithm (LFU) is based on popularity. The assumption is that pages that are requested more frequently are more likely to be requested again. It is possible to distinguish between Perfect LFU and In-Cache LFU. Perfect LFU keeps track of the frequency of all items the entire time, while In-Cache LFU resets an item's frequency counter whenever it is evicted from the cache. In this research, In-Cache LFU is used because it requires a lot less memory usage. If a page needs to be evicted from the cache, the page with the lowest frequency counter is chosen. The element that enters the cache gets a frequency of one. The frequency counter of an evicted element is reset to zero. Figure

2.1 illustrates behavior the LFU algorithm for the request sequence $1-1-2-3-2-3-2-3$. It can be seen that the LFU algorithm does not evict element 1, because the element 1 has the highest in-cache popularity. For this specific request sequence, the LFU algorithm causes seven cache misses while the optimal algorithm only causes three cache misses. This shows that there is a request sequence such that $\frac{LFU(\sigma)}{OPT(\sigma)} \geq 2$.

LFU algorithm

Request	1	1	2	3	2	3	2	3
Evicted				2	3	2	3	2
Cache – 1	1	1	2	3	2	3	2	3
Cache – 2			1	1	1	1	1	1

Optimal algorithm

Request	1	1	2	3	2	3	2	3
Evicted				1				
Cache – 1	1	1	1	3	3	3	3	3
Cache – 2			2	2	2	2	2	2

Figure 2.1: Behavior of the LFU and optimal algorithm for the request sequence $1-1-2-3-2-3-2-3$

2.4.4. SLRU - Segmented Least Recently Used

The Segmented Least Recently Used (SLRU) eviction strategy combines the assumption that both recency and frequency are indicators of which elements are requested again. The SLRU algorithm divides the cache into two segment, the protected and unprotected cache.
This protected and unprotected cache divide all elements in the cache into two groups: a group of pages that can be evicted and those that cannot. Whenever an element first enters the cache, it is placed in the unprotected cache. If an element is requested again and is still in the unprotected cache, it is moved to the protected cache. If the protected cache is full, the LRU algorithm is used to move an element to the unprotected cache. If the unprotected cache is full, the LRU algorithm is used to determine which element is removed. The flowchart for the behavior of the SLRU algorithm is provided in Figure 2.2. The SLRU algorithm has two parameters, the cache size and protected cache size.

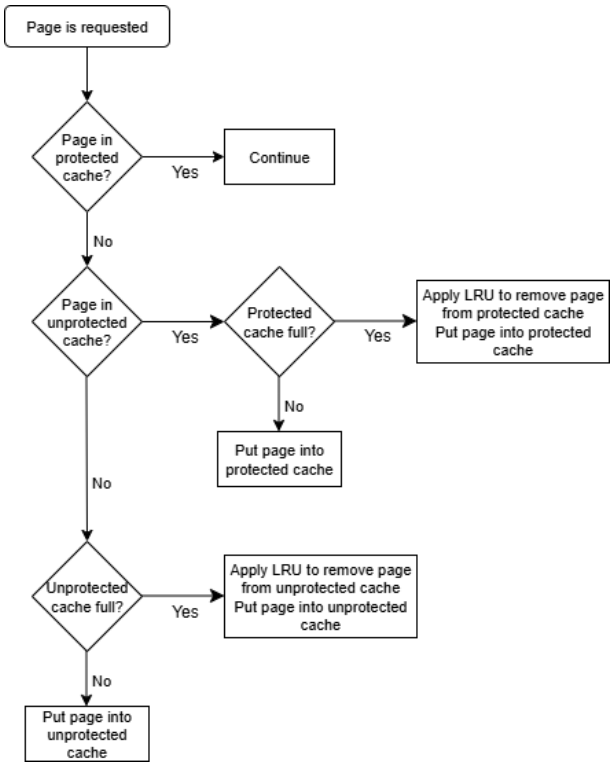


Figure 2.2: Flowchart for the SLRU algorithm

2.4.5. UNI - Uniform randomized

All previous eviction strategies were deterministic. It is also possible to use randomized eviction strategies. Their competitive ratio is typically better because it is based on the expected number of cache misses. The uniform randomized eviction strategy chooses a random element to evict whenever the cache is full. All pages have an equal opportunity to be evicted.

2.4.6. RRE - Randomized recency based

The recency-based randomized eviction strategy also evicts random items, but the probability of eviction depends on the recency of the item. Pages that have been requested more recently have a smaller probability of being evicted than pages that have not been requested for a while. In this case, the probability that the i^{th} page is evicted in a cache with k pages, when the cache is sorted on recency, is equal to $\frac{i}{k(k+1)/2}$.

3

Further mathematical analysis of the FIFO and LRU algorithm

The competitive ratio and the minimal competitive ratio for deterministic algorithms have been introduced in Chapter 2. For both the FIFO and LRU algorithms, the claim was that these algorithms are competitive. The first section of this chapter shows that the LRU algorithm is indeed competitive, using a proof based on the research of Sleator and Tarjan [6]. The second section contains a proof showing that the FIFO algorithm is competitive. Finally, the characteristic vector, as defined by Panagiotou and Souza [4], is used to prove some (in)equalities about the number of cache misses for the FIFO and LRU algorithms.

3.1. The competitive ratio upperbound for LRU

Sleator and Tarjan [6] provided a proof for the upper bound of the competitive ratio of the LRU algorithm. This section contains an extended version of their proof.

Theorem 5. *The competitive ratio of the LRU algorithm is at most* k.

Proof. Let σ be a request sequence. Then it is possible to partition σ into n subsequences $\sigma_0, \sigma_1, \ldots, \sigma_n$ such that σ_0 contains the first accessed element and at most k cache misses, while $\sigma_1, \ldots, \sigma_n$ have exactly k cache misses with the LRU algorithm.

Let p be the element accessed before σ_i. Then it is possible to conclude three things about the cache and the cache misses caused by the LRU algorithm:

1. At the start of the subsequence σ_i, the LRU cache and OPT cache share at least one element, namely p.

2. For a subsequence σ_i, the LRU algorithm will never fault twice on the same page a. To fault twice on the same page, all elements in the cache must have been removed. This causes k cache faults, and requesting a again would cause an additional cache miss, resulting in $k + 1$ cache misses, which is not possible by design of the subsequences.

3. For a subsequence σ_i it is not possible to fault on p during σ_i. At the beginning of σ_i, p must be in the cache. To fault on p again, the entire cache must be cleared, causing k cache misses, and requesting p again, which is needed to fault on p, causes another cache miss. This is not possible because σ_i has at most k cache misses.

The assumption is that the cache is empty at the beginning of the sequence. Then, for the first subsequence, σ_0, the number of cache misses for the LRU algorithm equals the number of cache misses for the optimal algorithm.

Now assume that the number of cache misses during σ_i using the LRU algorithm equals f_i. The optimal algorithm can only prevent cache misses by using elements already in the cache at the start of a new request sequence. The number of elements in the cache equals k, but during σ_i, LRU will never fault on p, so there are at most $k-1$ elements for which OPT can prevent a cache miss. Thus $OPT(\sigma_i)$ is at least $f_i - (k-1) = f_i - k + 1$.

Then, for any i

$$\frac{LRU(\sigma_i)}{OPT(\sigma_i)} \leq \frac{f_i}{f_i - k + 1} \leq \frac{k}{k - k + 1} = k.$$

And thus the conclusion is that

$$\frac{LRU(\sigma)}{OPT(\sigma)} \leq k.$$

□

Because both the lower and upper bounds of the competitive ratio for the LRU algorithm equal k, the competitive ratio for the LRU algorithm equals k.

3.2. The competitive ratio upper bound for FIFO

As shown in the previous section, Sleator and Tarjan [6] provided a proof for the competitive ratio of the upper bound for the LRU algorithm. Furthermore, they claim that the proof for the FIFO algorithm is similar, but did not provide it. In this section, a proof for the upper bound of the FIFO algorithm is presented.

The main difference between the proof of the FIFO and LRU algorithms is the way in which the partition of the request sequence σ is defined. A redefinition of the partition of the request sequence is needed because the FIFO algorithm evicts elements based on when they entered the queue. The definition of the subsequences, as in Theorem 5, makes it possible for an algorithm to cause no faults on a subsequence on which the FIFO algorithm does cause k faults. An example of this is given in Figure 3.1. It is also possible that the optimal offline algorithm does not fault on a subsequence for which the FIFO algorithm has k cache misses.

An algorithm

Request	1	2	3	2	1	2
Evicted			2	3		
Cache – 1	1	1	1	1	1	1
Cache – 2		2	3	2	2	2

FIFO algorithm

Request	1	2	3	2	1	2
Evicted			1		2	3
Cache – 1	1	1	2	2	3	1
Cache – 2		2	3	3	1	2

Figure 3.1: Behavior of an algorithm and the FIFO algorithm for the request sequence $1-2-3-2-1-2$ with subsequences

Theorem 6. *The competitive ratio of the FIFO algorithm is at most k.*

Proof. Let σ be a request sequence. Then it is possible to partition σ into n subsequences $\sigma_0, \sigma_1, \ldots, \sigma_n$ such that σ_0 contains the first accessed element and at most k cache misses, while $\sigma_1, \ldots, \sigma_n$ have exactly k cache misses. Furthermore, the last element of σ_i must be a cache miss.

Let p be the element accessed before σ_i. Then it must be a cache miss. It is possible to conclude four things about the cache and the cache misses caused by the FIFO algorithm:

1. At the start of the subsequence σ_i, the FIFO and OPT caches share at least one element, namely p.
2. The element p caused a cache miss, which implies that p entered the cache the latest. Because the FIFO algorithm evicts the element that entered the cache first, p has the lowest eviction priority.

3. For a subsequence σ_i, the FIFO algorithm will never fault twice on the same page a. Because, to fault twice on the same page, all elements in the cache should have been removed. This causes k cache faults, and requesting a again would cause another cache miss such that the number of cache misses during σ_i equals $k+1$, which is not possible by design of the subsequences.

4. For a subsequence σ_i it is not possible to fault on p during σ_i. At the beginning of σ_i, p must be in the cache. To fault on p during σ_i, p should first be removed from the cache. Due to the fact that p has the lowest eviction priority, k page faults need to occur before p is evicted. Then, faulting on p causes another cache miss, which is not possible by definition of the subsequences.

The assumption is that the cache is empty at the beginning of the sequence. Then, for the first subsequence, σ_0, the number of cache misses for the LRU algorithm equals the number of cache misses for the optimal algorithm.

Assume that the number of cache misses during σ_i using the FIFO algorithm equals f_i. The optimal algorithm can only prevent cache misses by using elements already in the cache at the start of a new request sequence. The number of elements in the cache equals k, but during σ_i FIFO will never fault on p, so there are only $k-1$ elements for which the optimal algorithm can prevent cache misses. Thus $OPT(\sigma_i)$ equals at least $f_i - (k-1) = f_i - k + 1$.

Then, for any i

$$\frac{FIFO(\sigma_i)}{OPT(\sigma_i)} \leq \frac{f_i}{f_i - k + 1} \leq \frac{k}{k - k + 1} = k.$$

And to conclude,

$$\frac{FIFO(\sigma)}{OPT(\sigma)} \leq k.$$

□

Because both the lower and upper bounds of the competitive ratio for the FIFO algorithm equal k, the competitive ratio for the FIFO algorithm equals k.

3.3. Comparing LRU and FIFO

Panagiotou and Souza [4] introduced the characteristic vector for a request sequence. Different request sequences can have the same characteristic vector. Albers and Frascaria [1] show that for any characteristic vector, the competitive ratio of LRU is better than the competitive ratio of FIFO. While LRU performs better than FIFO for most sequences, there are a few exceptions. This section explores why LRU is not always better than FIFO and how the characteristic vector defines the number of cache misses.

Definition 7 (Characteristic vector)**.** *Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_t)$ a request sequence with t requests and p distinct pages. A set of two requests σ_i, σ_j is called a pair whenever both σ_i and σ_j request the same page and are consecutive, so all pages requested between σ_i and σ_j are distinct from σ_i and σ_j. The number of distinct pages between a pair σ_i, σ_j is denoted by ℓ. For any sequence σ, given any ℓ, let C_ℓ denote the number of pairs σ_i, σ_j with exactly ℓ distinct pages in between them. The characteristic vector of a sequence σ is then defined as $c(\sigma) = (C_0(\sigma), C_1(\sigma), \ldots, C_{p-1}(\sigma))$.*

Panagiotou and Souza [4] claim that the characteristic vector defines the number of cache faults for the LRU algorithm. The following theorem establishes a relationship between the number of LRU cache misses and the characteristic vector.

Theorem 8. *For any request sequence σ, the number of cache misses caused by the LRU algorithm, $LRU(\sigma)$, equals $p + \sum_{\ell \geq k} C_\ell(\sigma)$, where p is the number of distinct pages, C_ℓ is the number of pairs with ℓ distinct elements in between, and k is the cache size.*

Proof. Let σ be any request sequence with a corresponding characteristic vector, denoted as $c(\sigma) = (C_0(\sigma), C_1(\sigma), \ldots, C_{t-1}(\sigma))$. The LRU algorithm evicts the pages whose request was least recently used. Thus, if between two requests of the same page a there are fewer than k distinct pages, a must still

be in the cache and will result in no fault. To conclude, the LRU algorithm does not fault on request pairs for which $\ell < k$. Furthermore, the LRU algorithm always faults on request pages for which $\ell \geq k$. Whenever there are at least k distinct pages between two requests to the same page a, the entire cache has been replaced, causing a fault on a. The LRU algorithm also faults on pages when they are first requested. Because the sequence contains p distinct pages, the LRU algorithm will fault on all of these p pages. So, to conclude, $LRU(\sigma) = p + \sum_{l \geq k} C_l(\sigma)$. □

The number of cache misses of the LRU algorithm can be defined by the characteristic vector due to two assumptions: (1) The LRU algorithm does not fault on request pairs for which $\ell < k$, and (2) LRU always faults on request pairs for which $\ell \geq k$. Neither of these assumptions hold for the FIFO algorithm, as shown in the following two lemmas.

Lemma 9. *The FIFO algorithm can fault on request pairs for which $\ell < k$, where ℓ is the number of distinct elements between the request pair and k the cache size.*

Proof. Consider the request sequence $1 - 2 - 1 - 3 - 1$ and the corresponding cache and eviction strategy for FIFO, as shown in Figure 3.2. The most interesting part of the cache is the last three items. This illustrates that FIFO faults on the final page, while there is only one distinct page, 3, between this request and the previous one. This shows that while $\ell < k$, the FIFO algorithm still causes a fault on this request pair. □

Request	1	2	1	3	1
Evicted				1	2
Cache – 1	1	1	1	2	3
Cache – 2		2	2	3	1

Figure 3.2: Behavior of the FIFO algorithm for the request sequence $1 - 2 - 1 - 3 - 1$

Lemma 10. *The FIFO algorithm does not have to fault on request pairs for which $\ell \geq k$, where ℓ is the number of distinct elements between the pair and k is the cache size.*

Proof. Consider the request sequence $1 - 5 - 2 - 1 - 5 - 3 - 4 - 2$ and the corresponding cache and eviction strategy for the FIFO algorithm, as shown in Figure 3.3. The request pair that is of interest is the request pair of 2. The cache size is $k = 3$ and the number of distinct pages between these requests is $\ell = 4$. However, there is no cache miss for the final 2. Thus, it is possible to conclude that $\ell \geq k$ is not a sufficient condition for a cache miss with the FIFO algorithm. □

Request	1	5	2	1	5	3	4	2
Evicted						1	5	
Cache – 1	1	1	1	1	1	5	2	2
Cache – 2		5	5	5	5	2	3	3
Cache – 3			2	2	2	3	4	4

Figure 3.3: Behavior of the FIFO algorithm for the request sequence $1 - 5 - 2 - 1 - 5 - 3 - 4 - 2$

While the assumptions that hold for the LRU algorithm do not hold for the FIFO algorithm, it is interesting to think about which assumptions do hold for FIFO. The first assumption that is true for FIFO is that for a request pair with ℓ distinct elements in between, no cache faults are only guaranteed whenever $\ell = 0$. Furthermore, FIFO must fault on a request pair whenever $\ell \geq 2k - 1$.

Theorem 11. *The largest number of distinct pages in between a request pair, ℓ, for which the FIFO algorithm never faults on a request pair, equals 0.*

Proof. For the FIFO algorithm, which page is evicted depends on which page was put first in the cache. If the page of the request pair was put in the cache first, even 1 distinct page request in between the request pair can cause a cache miss. However, if $\ell = 0$, there are no pages in between, so the request pair must be in the cache, which causes no cache miss. □

Theorem 12. *The FIFO algorithm always faults on a request pair with distance $\ell \geq 2k - 1$ and may not fault on a request pair with distance $\ell \leq 2k - 2$, where ℓ is the number of distinct pages between the request pair, and k is the cache size.*

Proof. To show that the FIFO algorithm always faults on a request pair with distance $\ell \geq 2k - 1$, it is needed to understand why the FIFO algorithm does not always fault on a request pair with distance $\ell \geq k$. This is due to the eviction order and the elements already in the cache. Assume the request pair consists of the element a. The cache consists of $k - 1$ distinct elements without a. It is possible to request all of these $k - 1$ pages without causing a cache miss. Furthermore, when the requested page a is last in eviction order, and thus entered the cache the latest, $k-1$ pages will be evicted before a is evicted. This means that it is possible to request another $k - 1$ distinct pages that have not been requested before, without evicting a. However, requesting another unique element will evict a. So, it is possible to request $2k-2$ distinct pages without a cache miss for a. However, adding another element must evict a. This shows that the FIFO algorithm must always fault on a request pair with distance $\ell \geq 2k - 1$. □

Corollary 13. *For any request sequence σ, the number of cache misses caused by the FIFO algorithm, $FIFO(\sigma)$, equals at least $p + \sum_{\ell \geq 2k-1} C_\ell(\sigma)$ and at most $p + \sum_{\ell > 0} C_\ell(\sigma)$, with p the number of distinct pages, C_ℓ the number of request pairs with distance ℓ, and k the cache size.*

Proof. This follows directly from the fact that the FIFO algorithm always faults on a request pair with distance $\ell \geq 2k - 1$, but that is also possible for FIFO to fault on request pairs with distance $1 \leq \ell < 2k - 1$. Furthermore, the FIFO algorithm must fault on the p distinct pages when they are first requested. So, the FIFO algorithm can fault on all pages with distance $1 \leq \ell < 2k - 1$, besides the guaranteed page faults for $\ell \geq 2k - 1$, so $FIFO(\sigma) \leq p + \sum_{\ell > 0} C_\ell(\sigma)$. □

4

Comparing algorithms on a complete set of sequences

The competitive ratio, as defined in Chapter 2, is the worst-case analysis of the ratio between the number of cache misses of an algorithm and the number of cache misses of the optimal offline algorithm. When both the number of possible requested pages (the universe size) and the sequence length are limited, it is possible to compute the ratio for each possible sequence. The distribution of these ratios can be compared for various algorithms. This chapter elaborates on the method and results of comparing these ratio distributions. The first section will go into more detail about the various design parameters, such as universe size, sequence length, cache size, and the chosen algorithms. The second section describes the implementation of these algorithms in Python. The final section will share the results of the various distributions of the ratio.

4.1. Selection of design parameters

The universe size, sequence length, and cache size are critical for analyzing the ratio between the number of cache misses of the algorithm and the optimal solution. The number of sequences grows exponentially with increases in universe size or sequence length. This exponential growth also exponentially increases the runtime and the memory usage. For this reason, the universe size equals 5, the sequence length equals 10 and the cache size k equals 3.

Furthermore, a selection of eviction strategies is required. The classification and algorithms from Chapter 2 were used for this purpose. The chosen eviction strategies are: (1) First In First Out, (2) Least Recently Used, (3) Least Frequently Used, (4) Segmented Least Recently Used, (5) Uniform Randomized, and (6) Recency Based Randomized.
All algorithms, except for Segmented Least Recently Used (SLRU), have the cache size as a parameter. The SLRU algorithm also needs a parameter describing which part of the cache is protected and which part is unprotected. In this case, due to the small cache size, the size of the protected cache equals 1.

4.2. Implementation

To compare all eviction strategies and the distribution of ratios, all sequences and the behavior of the eviction strategies needed to be simulated. All simulations were implemented in Python, specifically using Jupyter Notebook. First, a Queue class was created to simplify the implementation of other algorithms, such as FIFO and LRU. After that, each algorithm was implemented as functions with “cache size” and “sequence” as arguments. The SLRU algorithm also had an additional “cache size protected” argument. All possible sequences were generated using the itertools library in Python. Finally, all of components were combined to generate the cumulative distributions for the ratio between the number of cache misses generated by the eviction strategy and the optimal offline solution. The complete Python code for this can be found in Appendix A.

4.3. Results

Figure 4.1 shows the cumulative distribution of the ratio of the number of cache misses between the algorithms and the optimal solution. This is given for the complete set of sequences. All sequences have length 10 and are made with 5 distinct elements. Furthermore, the cache size is 3. Figure 4.1 shows that all algorithms perform as well as the optimal algorithm for at least 40% of the sequences. However, both random algorithms have significantly more sequences for which they perform as well as the optimal solution in comparison to the other algorithms. It is also interesting to note that the LRU algorithm has the least number of sequences for which it performs as well as the optimal offline algorithm. However, the LRU algorithm does have the lowest maximum ratio for this set of sequences, together with the FIFO algorithm. Furthermore, for this set of sequences, no algorithm has a maximum ratio higher than 2.5. This shows that, while Theorem 4 gives a lower bound for the competitive ratio, this competitive ratio is not achieved for most sequences, and for none of the sequences of length 10 with 5 distinct elements and a cache size of 3. It can be noted that in Figure 4.1 none of the algorithms

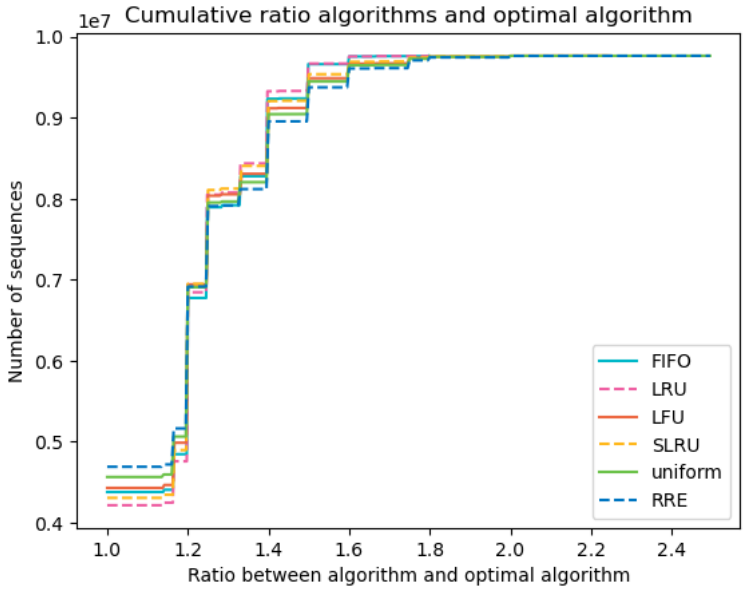


Figure 4.1: Cumulative distribution for the ratio between the cost of the algorithm and the cost of the optimal algorithm, with cache size 3, for the complete set of sequences with length 10 and 5 distinct pages.

obtain a ratio higher than 2.5. The cache size k equals 3. The algorithm has more than 3 cache misses whenever at least 4 distinct pages are requested. Each distinct requested page causes a cache miss for the optimal algorithm. The algorithm can have 10 cache misses at most, while the optimal algorithm has at least 4 cache misses in this case. Thus, the ratio between the cost of the algorithm and the optimal algorithm is lower than the competitive ratio.

5

Comparing algorithms for various sequences

In Chapter 4, the ratios between the algorithms and the optimal solution were analyzed for the complete set of sequences. This was only feasible for sequences with a limited length and a small number of distinct elements due to constraints on runtime and memory usage. In this chapter, the goal is to analyze longer sequences with more possible page requests by generating them randomly, instead of computing all possible sequences. Additionally, several types of sequences are generated based on different assumptions about request behavior. The first section introduces the design parameters, the second section discusses implementation, and the third section presents the results.

5.1. Design of parameters and sequences

In Chapter 2, several types of algorithms were introduced, along with their underlying assumptions. All of these algorithms are included in this analysis. The algorithms are: (1) First In First Out, (2) Least Recently Used, (3) Least Frequently Used, (4) Segmented Least Recently Used, (5) Uniform Randomized, and (6) Recency-Based Randomized.

These algorithms rely on three main assumptions: (1) requests are completely random, (2) more recently requested pages are more likely to be requested again, and (3) frequently requested pages are more likely to be requested again. Corresponding to these assumptions, three types of request sequences are considered: (1) randomized sequences, (2) recency-based sequences, and (3) favorite element sequences.

For the randomized sequences, all pages have an equal probability of being the next requested page. The recency-based sequences work with a look-back of 5, 10, or 20. The next requested element has an arbitrary probability of 0.2 to be chosen from the last x elements, with x equaling the look-back. Otherwise, a uniformly random element from the universe is selected. Favorite element sequences are generated similarly, but in this case, there is an arbitrary probability of 0.2 that a “favorite element” is the next requested element. The first 5, 10, or 20 elements in the universe are marked as favorite elements. Otherwise, a random element from the entire universe is chosen.

The main design parameters are universe size, sequence length, and cache size. For SLRU, the protected cache size parameter was also introduced. In this Chapter, the sequence length will be set to 1000. The other design parameters will be varied, so the cache sizes are 5, 10, and 20. The universe size equals 50, 100, or 150. The protected cache size is a percentage of the cache size, in this case 20%. The plausibility of the results depends on the sample size of sequences; larger sample sizes yield more robust results but also require significantly more memory and computation time. Considering these trade-offs, the sample size is set to 10.000 sequences per experiment. For recency-based sequences and favorite element sequences, the sequence-specific parameters, look-back and percentage of favorite elements, were also varied.

5.2. Implementation

To compare all eviction strategies for various types of sequences, their behavior was simulated using Python, specifically Jupyter Notebook. A separate file was created for each sequence type, while sharing a general structure: implementing the Queue class and algorithms, generating the sequences, computing the ratio between the solution of the algorithm and the optimal algorithm, and producing cumulative distribution plots.

The results for varying input parameters such as look-back, cache size, and universe size were generated by looping over the code multiple times. All generated data were stored in a pandas DataFrame. The complete code is provided in Appendices B, C, and D.

5.3. Results randomized sequences

Figure 5.1 shows the cumulative distribution of the ratio between the number of cache misses of each algorithm and the optimal solution for varying universe sizes, 50, 100 and 150. However, the cache size k is fixed to 5. In Figure 5.2, the cache size equals 10, and in Figure 5.3 the cache size equals 20.

Figure 5.1 shows that a larger universe size results in a smaller ratio between the algorithm's number of cache faults and the optimal number of cache faults. The expectation is that more distinct pages will lead to more cache misses. However, more distinct pages not only lead to more cache misses for the various algorithms, but also for the optimal solution. This could explain why the ratio of cache misses between the algorithms and the optimal solution is lower. Furthermore, it is interesting to note that all cumulative distributions have an "S" shape, but the LFU curve is significantly wider. This indicates that LFU has a higher proportion of low ratios but also the highest maximum ratio. For the FIFO and LRU algorithms, this is opposite; they have a low proportion of sequences with a low ratio but also have a low maximum ratio. For a larger universe size, the "S" shape compresses, reducing the difference in proportions.

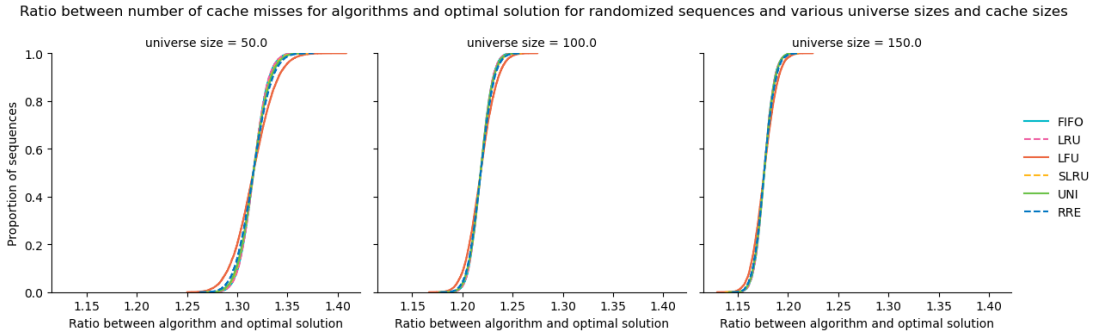


Figure 5.1: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with cache size 5, for randomized sequences with various universe sizes.

The same observations hold for Figure 5.2 and 5.3. However, the scale of the axis on which the ratio is represented differs. The ratios change for different cache sizes. When comparing these ratios in Figure 5.1, 5.2, and 5.3 for the same universe sizes, it can be seen that a larger cache size leads to a higher ratio between the algorithms and the optimal algorithm. The expectation is that a larger cache size leads to fewer cache misses and thus better performance. However, the ratio of the cost between the algorithm and the optimal solution is higher. This once again shows why the (competitive) ratio is not the best measure to determine how well algorithms perform in practice. Appendix E contains a plot for which the axes all have the same scaling, for better comparison.

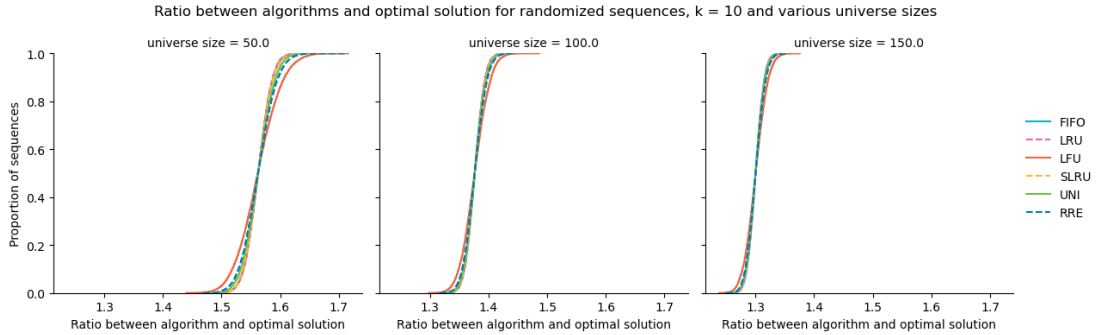


Figure 5.2: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with cache size 10, for randomized sequences with various universe sizes.

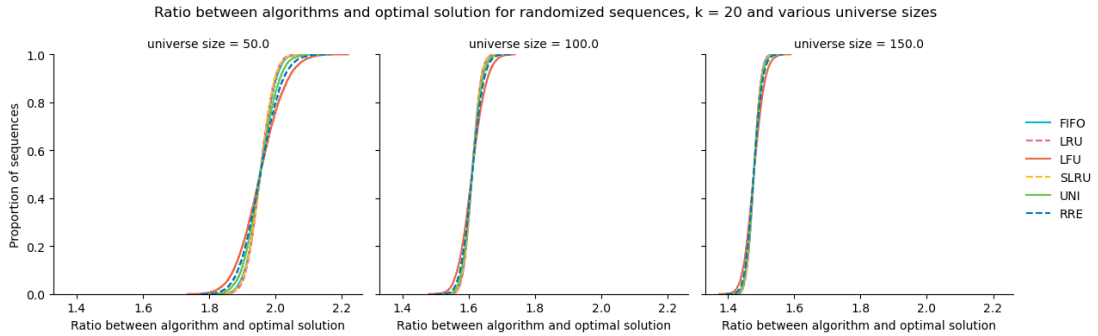


Figure 5.3: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with cache size 20, for randomized sequences with various universe sizes.

5.4. Results recency-based sequences

For recency-based sequences, three parameters were varied: cache size, universe size, and look-back, yielding 27 different combinations. Seventeen of these combinations are discussed in this section. All combinations can be found in Appendix E.

Figure 5.4 shows the cumulative distribution for sequences with fixed cache size $k = 10$ and varying universe sizes and look-back. Figure 5.5 instead fixed the universe size at 100. The middle panel of both figures corresponds to the same parameter values.

Figure 5.4 shows that the LFU performs worse than the other algorithms. This aligns with the expectations. These sequences were based on the assumption that recent elements have a higher probability of being requested again while LFU is based on the assumption that frequently requested elements are more likely to be requested again. The LRU algorithm performs the best and the Segmented LRU algorithm also performs pretty well. The randomized recency-based algorithm (RRE), which also assumes that more recently requested elements will be requested again, does not perform as well as some other algorithms that are not based on this assumption, such as FIFO and the uniformly random eviction strategy. Finally, it can once again be seen that a larger universe size leads to a smaller ratio. Furthermore, it seems that a larger look-back yields a slightly higher ratio.

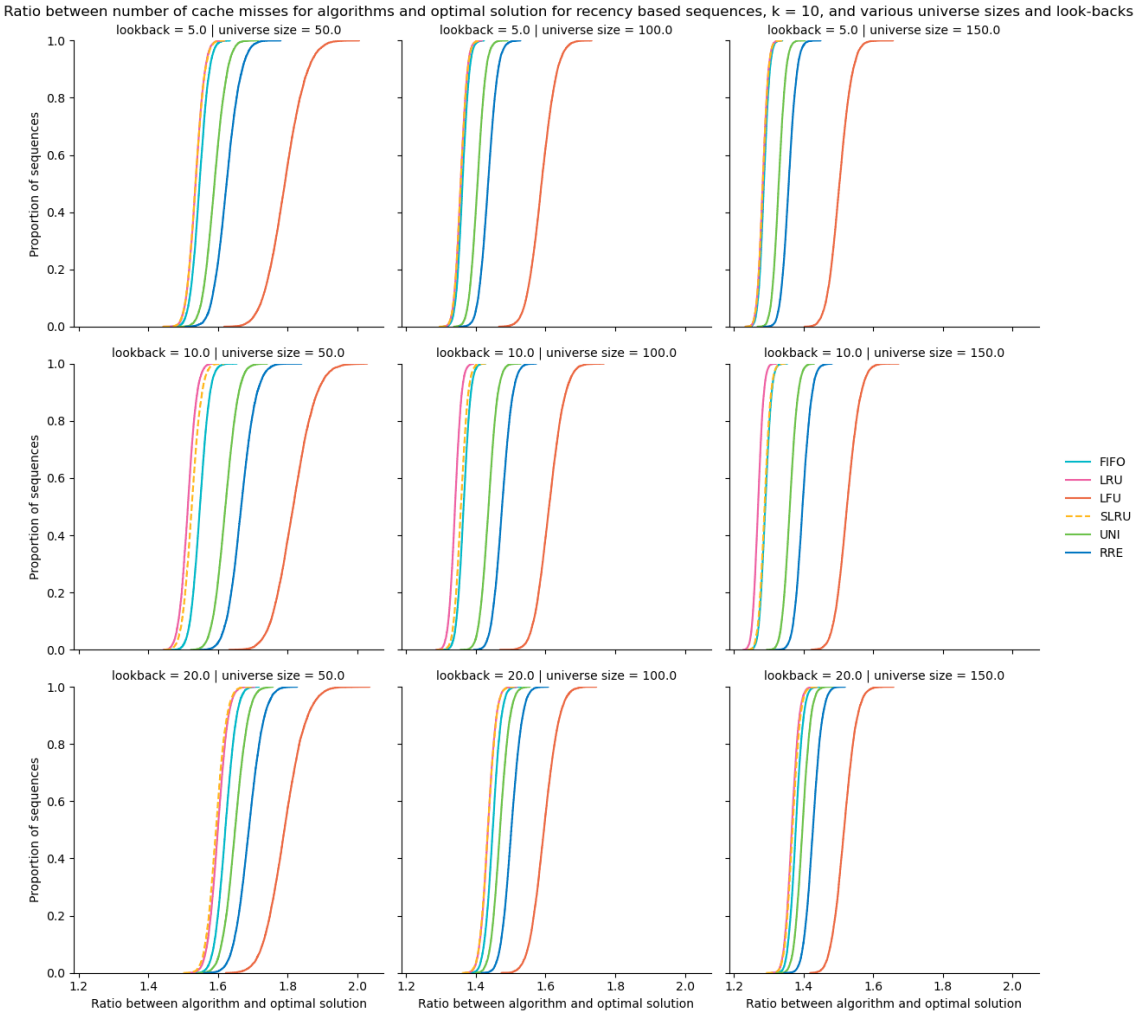


Figure 5.4: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with cache size 10, for recency-based sequences and various universe sizes and look-backs.

Figure 5.5 once again shows that a larger cache size yields a higher ratio and that a larger look-back also yields a higher ratio. Furthermore, it also shows, just like Figure 5.4, that the LFU algorithm performs the worst and that the LRU, SLRU, and FIFO algorithms perform quite well.

An explanation for the fact that a larger look-back leads to a higher ratio might be the fact that with a smaller look-back, the possibility that an item is still in the cache is greater. For example, take the LRU algorithm, which contains the last k requests. If the look-back is smaller than k, every time that a previous look-back element is requested, it is guaranteed that it must be in the cache.

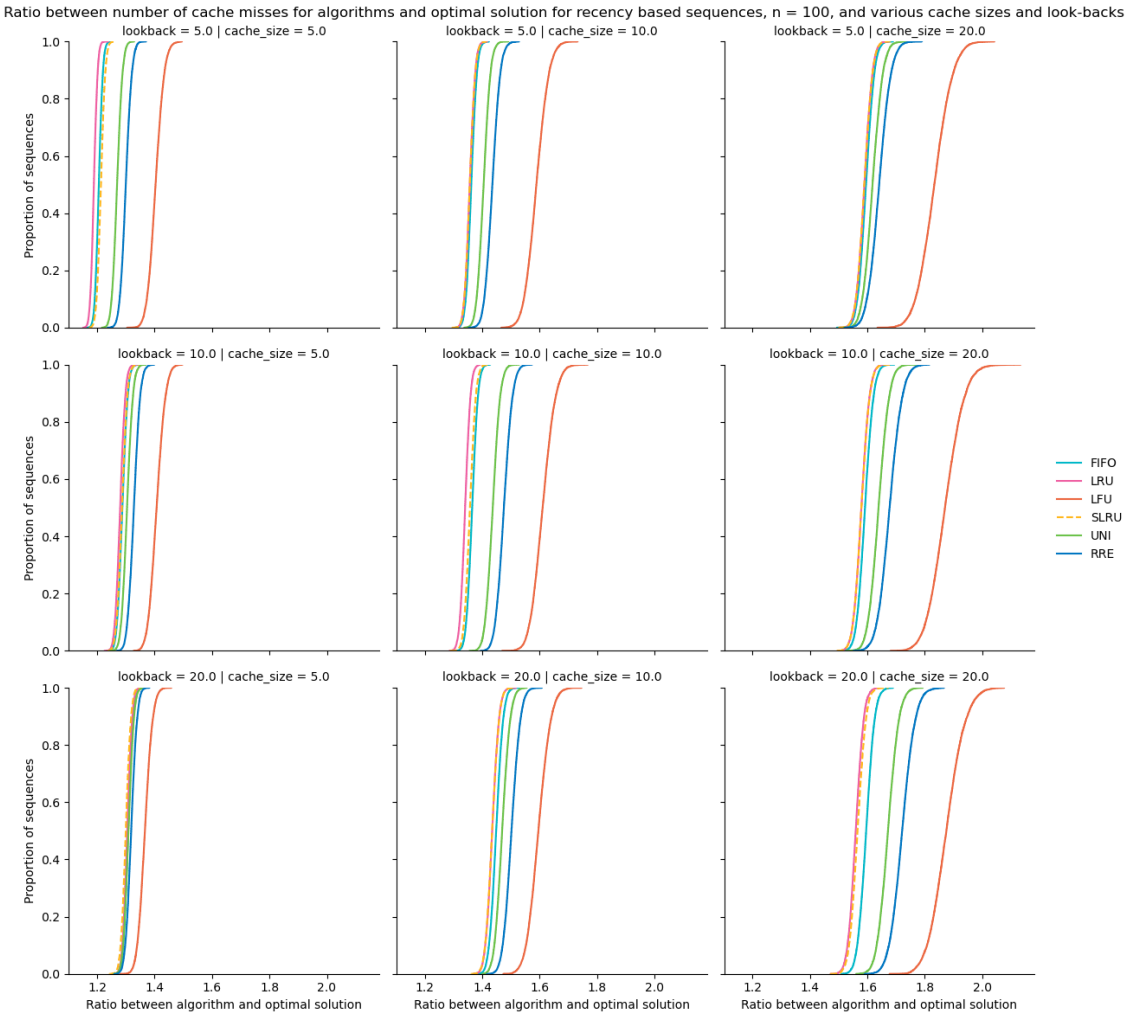


Figure 5.5: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with universe size 100, for recency-based sequences and various cache sizes and look-backs.

5.5. Results favorite element sequences

The favorite element sequences vary three parameters: the cache size, universe size, and favorite percentage. A selection of results is analyzed. The full set of results can be found in Appendix E.

Figure 5.6 shows the ratios with a fixed cache size of 10 with varying universe sizes and favorite element percentages. Unlike the recency-based sequences, the LFU algorithm performs significantly better than the other algorithms. However, the SLRU has a smaller proportion of sequences with a high competitive ratio. Furthermore, the LRU algorithm also performs relatively well for the favorite element sequences. This could be explained by the fact the LRU algorithm is based on the assumption that recently requested elements will be requested again. While not all elements recently requested will be requested again, some of the recently requested elements are favorite elements and have a higher probability of being requested again.

Figure 5.6 also shows the earlier observed trend of a lower ratio for a higher universe size. Finally, the relation between the percentage of favorite elements and the competitive ratio is more difficult to determine. The LFU curve moves closer to the other curves, but it is not clear if the competitive ratio really shifts.

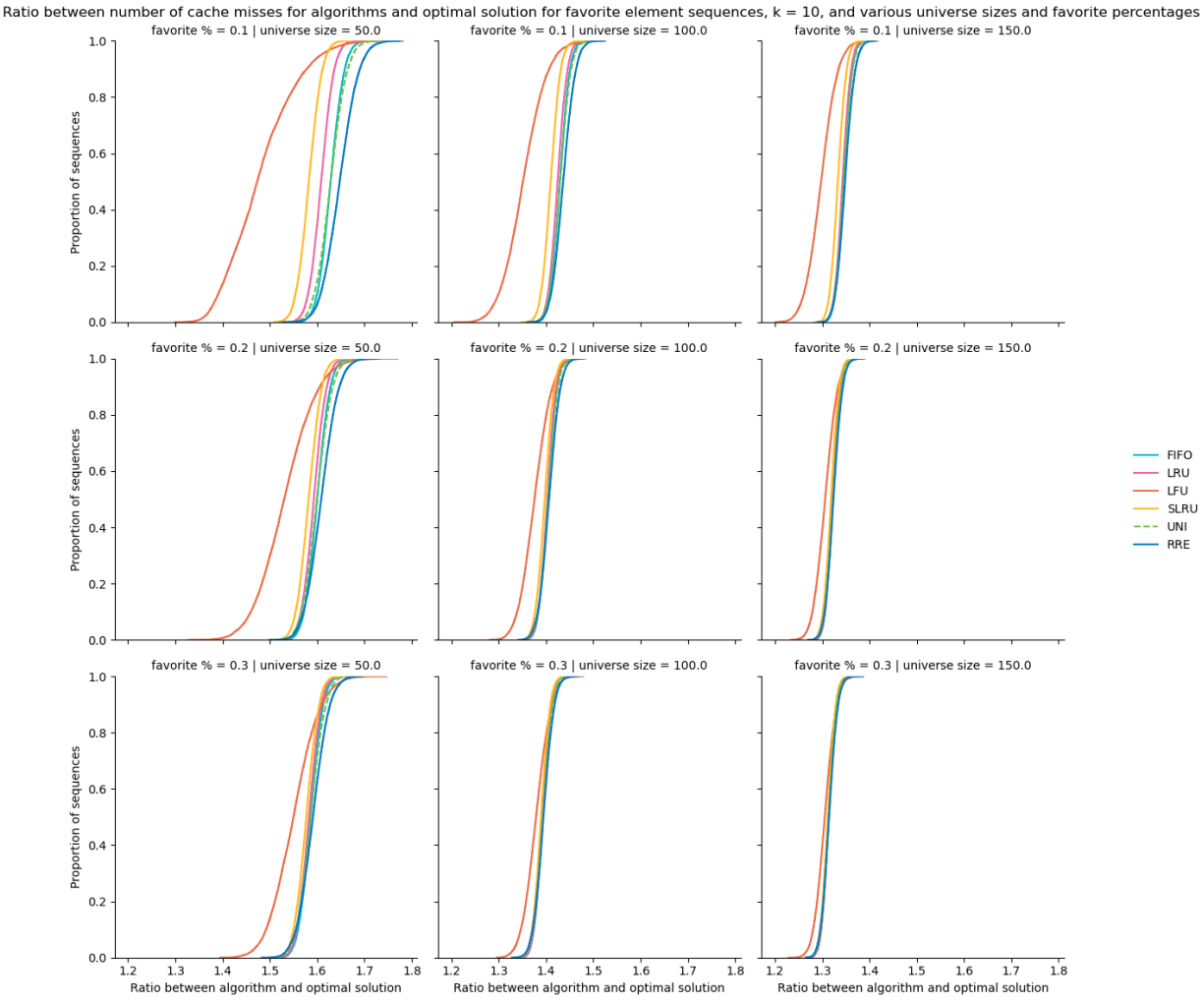


Figure 5.6: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with cache size 10, for favorite element sequences and various universe sizes and favorite percentages.

Figure 5.7 also shows a trend that has been observed earlier. This is the trend that a higher cache size yields a higher competitive ratio. Furthermore, it can be seen that the LFU, SLRU, and LRU algorithms

still perform very well, while the recency-based random algorithm (RRE) performs worse. Finally, it is not easy in this case to determine if a smaller percentage of favorite elements yields a higher or lower competitive ratio. This is examined in more detail in Figure 5.8.

Figure 5.8 once again shows that the LFU curve moves closer to the other curves and thus moves to the right. The other curves move slightly to the left. This would suggest that the LFU algorithm performs worse for a higher percentage of favorite elements, while the other algorithms perform slightly better with a higher percentage of favorite elements in this specific case.

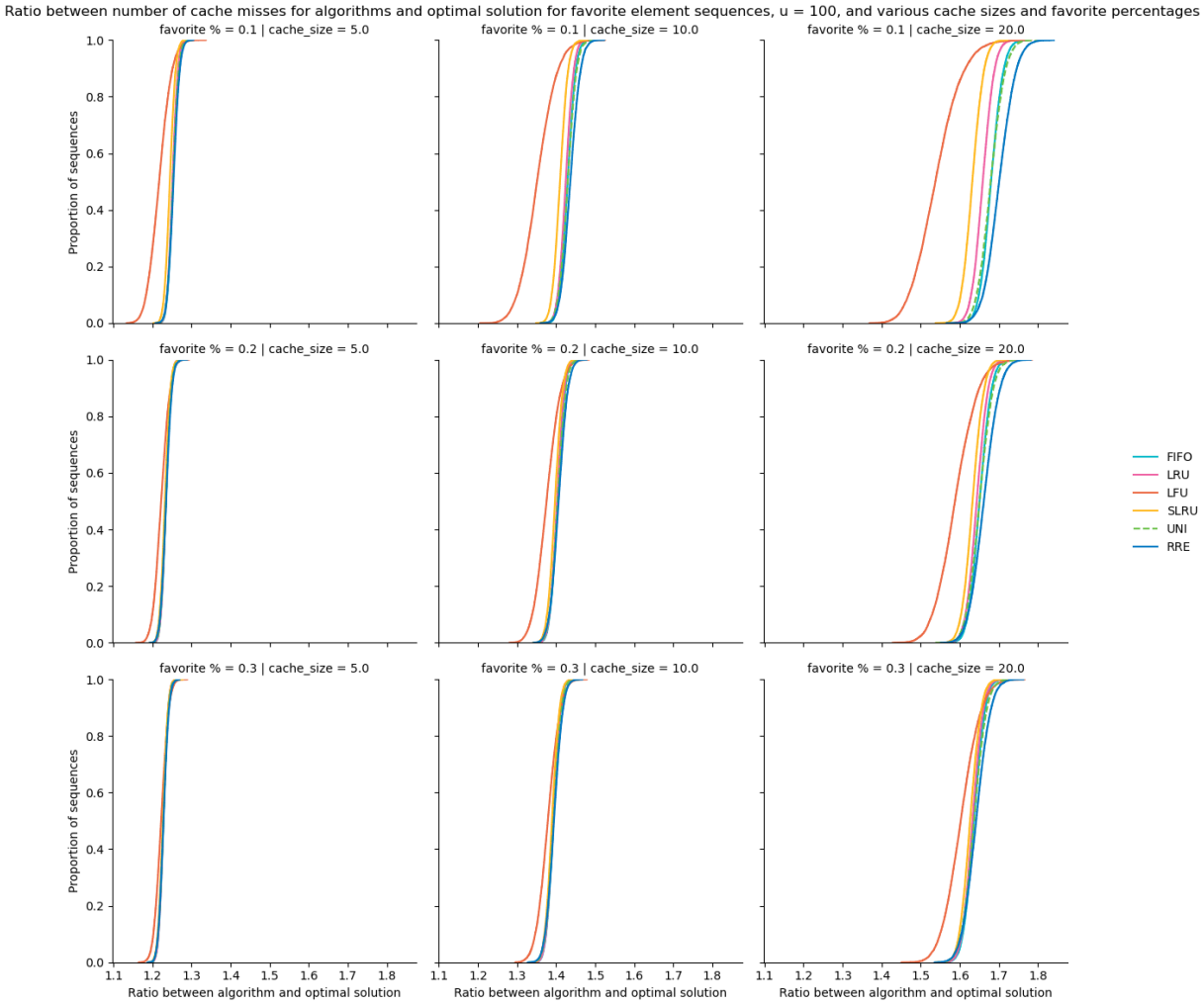


Figure 5.7: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with universe size 100, for favorite element sequences and various favorite percentages.

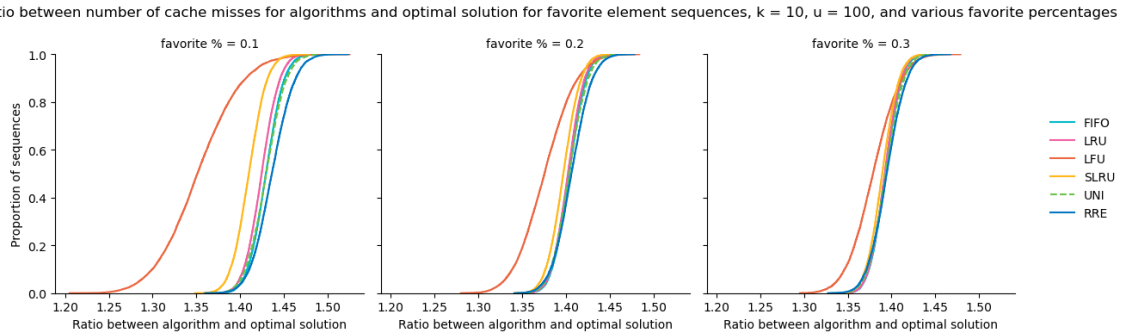


Figure 5.8: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, with cache size 10, universe size 100, for favorite element sequences and favorite percentages.

6

Conclusion and limitations

In this chapter, the results from the previous chapters are combined to answer the research question: “How do deterministic and randomized algorithms perform with differently structured request sequences in the caching problem, and how can these algorithms be compared to one another?”. The research question will be answered using the four subquestions: (1) Which metric can be used to compare different algorithms? (2) Which characteristics does this metric have for various algorithms? (3) How do algorithms compare on a complete set of sequences? (4) How do algorithms compare for various types of sequences? Furthermore, the limitations of this research will be examined, along with possible options for future research.

The most commonly used metric to compare algorithms for the caching problem is the competitive ratio. While the competitive ratio can easily be computed and does distinguish between algorithms, the LRU and FIFO algorithms are k-competitive, while most deterministic algorithms have a lower bound on the competitive ratio of k. The competitive ratio is merely a worst-case analysis. In practice, algorithms may perform significantly better than their competitive ratio suggests. Moreover, because the competitive ratio is defined over all possible sequences, it does not capture performance differences between algorithms for specific types of request patterns, such as recency-based sequences or favorite element sequences.

The competitive ratio cannot differentiate performances for various request patterns. An alternative metric, based on the characteristic vector, can solve this problem partially. The number of cache faults for LRU is defined by the characteristic vector; $LRU(\sigma) = p + \sum_{\ell \geq k} C_\ell(\sigma)$. For the FIFO algorithm, the characteristic vector only provides a lower and upper bound for the number of cache faults; $p + \sum_{\ell \geq 2k-1} C_\ell(\sigma) \leq FIFO(\sigma) \leq p + \sum_{\ell > 0} C_\ell(\sigma)$. This research does not examine how the characteristic vector defines the number of cache misses for other algorithms, but this would be interesting to explore further.

An additional advantage of the characteristic vector is that the structure of the sequence is captured within the characteristic vector. This raises the question: ”Do specific request patterns yield differently structured characteristic vectors?”. If so, and if more lower and upper bounds for the number of cache misses for algorithms are known, it might be possible to identify which algorithm causes the least cache misses for a given request pattern, using the characteristic vector. Panagiotou and Souza [4] explored this partially; instead of determining the characteristic vector for a request pattern, they designed a specific characteristic vector and computed a new and improved competitive ratio for the LRU algorithm with this specific characteristic vector.

After identifying the different metrics and their characteristics, it is insightful to see how the algorithms perform in practice. For this, six different algorithms were evaluated: (1) First In First Out, (2) Least Recently Used, (3) Least Frequently Used, (4) Segmented Least Recently Used, (5) Uniform Randomized, and (6) Recency-Based Randomized.

The algorithms were first compared on all possible sequences with 5 distinct pages and length 10, using a cache size of 3. It was interesting to note that while many research claims that the LRU algorithm performs better than the FIFO algorithm, such as Albers and Frascaria [1], the claim is that the competitive ratio for LRU for a specific characteristic vector is always better, the results show that there is a small number of sequences for which the FIFO algorithm performs better than the LRU algorithm. Furthermore, the randomized algorithms had the highest proportion of sequences for which the ratio equaled one. Due to memory and computational time limitations, the sequence length and universe size were quite small. With better computers, it would be possible to analyze the complete set of sequences for longer sequences with more distinct pages.

When comparing algorithms for these various sequence types, a noteworthy observation is that the LRU algorithm is not always the best-performing algorithm. For uniformly random and recency-based sequences, it performs best, while LFU performs worst. For favorite element sequences, however, the LFU algorithm performs best. The SLRU algorithm performs consistently well across the sequence types; this makes it an interesting algorithm for further analysis.

Because different algorithms perform better for different types of sequences, it would be useful to study user behavior before deciding which algorithm is the best fit for a specific application. For different applications users would probably show different behavior. Further research could identify user behavior patterns and recommend which type of algorithm suits this request pattern.

Finally, several general patterns emerged when varying the parameters. A larger cache size leads to a higher ratio for the number of cache misses between the algorithm and the optimal solution, while the assumption is that a higher cache size leads to fewer cache misses. The (competitive) ratio tells nothing about the absolute number of cache faults, only about the proportion to the optimal algorithm. This could also explain why a larger universe size and thus more distinct elements, leads to a lower ratio of the number of cache faults, even though it would probably lead to more cache faults. Further research could further explain these trends by comparing the number of cache faults for the algorithms and the optimal algorithm instead of the ratio between these.

Generative AI statement

For this thesis project, GenAI was used to assist with reviewing the final version of the work. All text and content have been produced by the author, but GenAI provided some feedback on spelling, grammar, and readability of the thesis. All feedback was considered carefully before changing anything. For this, Grammarly was used along with ChatGPT version 5.1. The prompt to ask feedback from ChatGPT was ”Could you please provide feedback on the following text. I specifically want feedback on spelling, grammar, and readability”.

Bibliography

[1] S. Albers and D. Frascaria. “Quantifying competitiveness in paging with locality of reference”. In: *Algorithmica* 80.12 (2018), pp. 3563–3596. DOI: 10.1007/s00453-018-0406-9.

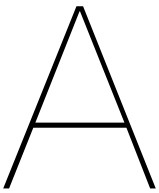
[2] L. A. Belady. “A study of replacement algorithms for a virtual-storage computer”. In: *IBM Systems Journal* 5.2 (1966), pp. 78–101. DOI: 10.1147/sj.52.0078.

[3] A. Gupta. *Advanced Algorithms: notes for CMU 15-850 (fall 2020)*. 2020, pp. 273–285.

[4] K. Panagiotou and A Souza. “On adequate performance measures for paging”. In: *Proceedings of the thirty-eighth Annual ACM Symposium on Theory of computing*. 2006, pp. 487–496. DOI: 10.1145/1132516.1132587.

[5] S. Podlipnig and L. Böszörmenyi. “A survey of Web cache replacement strategies”. In: *ACM Computing Surveys* 35 (Dec. 2003), pp. 374–398. DOI: 10.1145/954339.954341.

[6] D. Sleator and R. Tarjan. “Amortized efficiency of list update paging rules”. In: *Communications of the ACM* 28 (Feb. 1985), pp. 202–208. DOI: 10.1145/2786.2793.



Python code complete set of sequences

```
#imports
import matplotlib.pyplot as plt
import numpy as np
import math
import random

from itertools import product

#QUEUES
class MQueue:
  def __init__(self):
    self.queue = []

  def enqueue(self, element):
    self.queue.append(element)

  def dequeue(self):
    if self.isEmpty():
      return "Queue is empty"
    return self.queue.pop(0)

  def remove(self, i):
      self.queue.remove(i)

  def isEmpty(self):
    return len(self.queue) == 0

  def size(self):
    return len(self.queue)

  def contains(self, i):
      for j in self.queue:
          if j == i:
              return True
      return False

  def __str__(self):
      return str(self.queue)

#FIFO algorithm
def FIFO(sequence, cachesize):
    cache = MQueue()
    cache_misses = 0

    for i in sequence:
        if cache.contains(i):
            continue
        elif cache.size() < cachesize:
            cache_misses += 1
            cache.enqueue(i)
```

```
        else:
            evict = cache.dequeue()
            cache_misses += 1
            cache.enqueue(i)

    return cache_misses

#LRU algorithm
def LRU(sequence, cachesize):
    cache = MQueue()
    cache_misses = 0

    for i in sequence:
        if cache.contains(i):
            cache.remove(i)
            cache.enqueue(i)
        elif cache.size() < cachesize:
            cache_misses += 1
            cache.enqueue(i)
        else:
            evict = cache.dequeue()
            cache_misses += 1
            cache.enqueue(i)

    return cache_misses

#LFU algorithm
def LFU(sequence, cachesize):
    cache = []
    frequency = dict()
    cache_misses = 0

    for i in sequence:
        if i in cache:
            frequency[i] += 1
        elif len(cache) < cachesize:
            cache.append(i)
            frequency[i] = 1
            cache_misses += 1
        else:
            lowest = math.inf
            evict = None
            for j in cache:
                if frequency[j] < lowest:
                    lowest = frequency[j]
                    evict = j
            frequency.pop(evict)
            cache.remove(evict)
            cache.append(i)
            frequency[i] = 1
            cache_misses += 1

    return cache_misses

#Segmented LRU
def SLRU(sequence, cachesize_protected, cachesize_total):
    unprotected_cache = MQueue()
    protected_cache = MQueue()
    cache_misses = 0

    for i in sequence:
        if protected_cache.contains(i):
            protected_cache.remove(i)
            protected_cache.enqueue(i)
        elif unprotected_cache.contains(i):
            if protected_cache.size() < cachesize_protected:
                protected_cache.enqueue(i)
                unprotected_cache.remove(i)
            else:
                evict = protected_cache.dequeue()
                unprotected_cache.enqueue(evict)
```

```
                protected_cache.enqueue(i)
                unprotected_cache.remove(i)
        elif unprotected_cache.size() + protected_cache.size() < cachesize_total:
            unprotected_cache.enqueue(i)
            cache_misses += 1
        else:
            evict = unprotected_cache.dequeue()
            unprotected_cache.enqueue(i)
            cache_misses += 1

    return cache_misses

#Uniform algorithm
def UNIFORM(sequence, cachesize):
    cache = []
    cache_misses = 0

    for i in sequence:
        if i in cache:
            continue
        elif len(cache) < cachesize:
            cache_misses += 1
            cache.append(i)
        else:
            position = round(random.uniform(-0.5, cachesize - 0.5))
            cache_misses += 1
            cache[position] = i

    return cache_misses

#Recency based random algorithm
def RANDOM_RECENCY(sequence, cachesize):
    cache = MQueue()
    cache_misses = 0
    total_prob = cachesize*(cachesize+1)/2
    for i in sequence:
        if cache.contains(i):
            cache.remove(i)
            cache.enqueue(i)
        elif cache.size() < cachesize:
            cache_misses += 1
            cache.enqueue(i)
        else:
            n = 1
            position = random.uniform(0, total_prob)
            cache_misses += 1
            for j in cache.queue:
                if position < n*(n+1)/2:
                    cache.remove(j)
                    cache.enqueue(i)
                    break
                n += 1

    return cache_misses

#Optimal algorithm
def OPT(sequence, cachesize):
    cache = []
    cache_misses = 0
    for i in range(len(sequence)):
        element = sequence[i]
        if element in cache:
            continue
        elif len(cache) < cachesize:
            cache_misses += 1
            cache.append(element)
        else:
            future_requests_positions = dict()
            cache_misses += 1
            remaining_sequence = sequence[i::]
            for x in cache:
```

```
                future_requests_positions[x] = 0
                position = 1
                for y in remaining_sequence:
                    if x == y:
                        future_requests_positions[x] = position
                        break
                    position += 1

            evict = None

            if 0 in future_requests_positions.values():
                for x in future_requests_positions.keys():
                    if future_requests_positions[x] == 0:
                        evict = x
                        break
            else:
                sort = sorted(future_requests_positions.values())
                farthest = sort[-1]
                for x in future_requests_positions.keys():
                    if future_requests_positions[x] == farthest:
                        evict = x
                        break
            cache.remove(evict)
            cache.append(element)
    return cache_misses

#set parameters
possible_elements = [1,2,3,4,5]
length = 10
cache_size = 3

#generate data
counter = 0
data_FIFO = []
data_LRU = []
data_LFU = []
data_SLRU = []
data_UNI = []
data_RRE = []

for i in product(possible_elements, repeat = length):
    seq = i

    cache_misses_FIFO = FIFO(seq, cache_size)
    cache_misses_LRU = LRU(seq, cache_size)
    cache_misses_LFU = LFU(seq, cache_size)
    cache_misses_SLRU = SLRU(seq, 1, cache_size)
    cache_misses_UNI = UNIFORM(seq, cache_size)
    cache_misses_RRE = RANDOM_RECENCY(seq, cache_size)
    OPT_misses= OPT(seq, cache_size)

    rat_FIFO = cache_misses_FIFO/OPT_misses
    rat_LRU = cache_misses_LRU/OPT_misses
    rat_LFU = cache_misses_LFU/OPT_misses
    rat_SLRU = cache_misses_SLRU/OPT_misses
    rat_UNI = cache_misses_UNI/OPT_misses
    rat_RRE = cache_misses_RRE/OPT_misses

    data_FIFO.append(rat_FIFO)
    data_LRU.append(rat_LRU)
    data_LFU.append(rat_LFU)
    data_SLRU.append(rat_SLRU)
    data_UNI.append(rat_UNI)
    data_RRE.append(rat_RRE)

    counter += 1
    if counter % 1000000 == 0:
        print(i)

#computing cumulative
values_FIFO, base_FIFO = np.histogram(data_FIFO, bins=300)
```

```
cumulative_FIFO = np.cumsum(values_FIFO)

values_LRU, base_LRU = np.histogram(data_LRU, bins=300)
cumulative_LRU = np.cumsum(values_LRU)

values_LFU, base_LFU = np.histogram(data_LFU, bins=300)
cumulative_LFU = np.cumsum(values_LFU)

values_SLRU, base_SLRU = np.histogram(data_SLRU, bins=300)
cumulative_SLRU = np.cumsum(values_SLRU)

values_UNI, base_UNI = np.histogram(data_UNI, bins=300)
cumulative_UNI = np.cumsum(values_UNI)

values_RRE, base_RRE = np.histogram(data_RRE, bins=300)
cumulative_RRE = np.cumsum(values_RRE)

#plotting FIFO
plt.title("Cumulative ratio FIFO and optimal algorithm")
plt.xlabel("Ratio FIFO and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_FIFO[:-1], cumulative_FIFO, c='#00B8C8')
plt.show()

#plotting LRU
plt.title("Cumulative ratio LRU and optimal algorithm")
plt.xlabel("Ratio LRU and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_LRU[:-1], cumulative_LRU, c='#EF60A3')
plt.show()

#plotting LFU
plt.title("Cumulative ratio LFU and optimal algorithm")
plt.xlabel("Ratio LFU and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_LFU[:-1], cumulative_LFU, c='#EC6842')
plt.show()

#plotting SLRU
plt.title("Cumulative ratio SLRU and optimal algorithm")
plt.xlabel("Ratio SLRU and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_SLRU[:-1], cumulative_SLRU, c='#FFB81C')
plt.show()

#plotting UNI
plt.title("Cumulative ratio uniform and optimal algorithm")
plt.xlabel("Ratio uniform and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_UNI[:-1], cumulative_UNI, c='#6CC24A')
plt.show()

#plotting RRE
plt.title("Cumulative ratio recency randomized and optimal algorithm")
plt.xlabel("Ratio recency randomized and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_RRE[:-1], cumulative_RRE, c='#0076C2')
plt.show()

#combined plot
plt.title("Cumulative ratio algorithms and optimal algorithm")
plt.xlabel("Ratio between algorithm and optimal algorithm")
plt.ylabel("Number of sequences")
plt.plot(base_FIFO[:-1], cumulative_FIFO, c='#00B8C8', label = "FIFO")
plt.plot(base_LRU[:-1], cumulative_LRU, c='#EF60A3', label = "LRU", linestyle = "--")
plt.plot(base_LFU[:-1], cumulative_LFU, c='#EC6842', label = "LFU")
plt.plot(base_SLRU[:-1], cumulative_SLRU, c='#FFB81C', label = "SLRU", linestyle = "--")
plt.plot(base_UNI[:-1], cumulative_UNI, c='#6CC24A', label = "uniform")
plt.plot(base_RRE[:-1], cumulative_RRE, c='#0076C2', label =  "RRE", linestyle = "--")
plt.legend(loc="lower right")
plt.show()
```

B

Python code randomized sequences

The implementation of the Queue and algorithms is the same as in Appendix A and will not be included in this appendix.

```
#imports
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import pandas as pd
import seaborn as sns

#implementation of MQueue and algorithms

#set parameters
universe_size = [50,100,150]
sequence_length = [1000]
cache_size = [5,10,20]
cache_size_protected_percent = 0.2
iters = 10000

data2 = pd.DataFrame(columns = [”universe size”, ”sequence_length”, ”cache_size”, ”proteced %
    ”, ”FIFO”, ”LRU”, ”LFU”, ”SLRU”, ”UNI”, ”RRE”])

count = 0
for unsize in universe_size:
    for seqlength in sequence_length:
        for i in range(iters):
            seq = []
            for j in range(seqlength):
                seq.append(random.randint(0, unsize - 1))

            for k in cache_size:
                cache_misses_FIFO = FIFO(seq, k)
                cache_misses_LRU = LRU(seq, k)
                cache_misses_LFU = LFU(seq, k)
                cache_misses_SLRU = SLRU(seq, k*cache_size_protected_percent, k)
                cache_misses_UNI = UNIFORM(seq, k)
                cache_misses_RRE = RANDOM_RECENCY(seq, k)
                OPT_misses= OPT(seq, k)

                rat_FIFO = cache_misses_FIFO/OPT_misses
                rat_LRU = cache_misses_LRU/OPT_misses
                rat_LFU = cache_misses_LFU/OPT_misses
                rat_SLRU = cache_misses_SLRU/OPT_misses
                rat_UNI = cache_misses_UNI/OPT_misses
                rat_RRE = cache_misses_RRE/OPT_misses

                data2.loc[count] = unsize, seqlength, k, cache_size_protected_percent,
    rat_FIFO, rat_LRU, rat_LFU, rat_SLRU, rat_UNI, rat_RRE
                count += 1
```

```
                if count % 1000 == 0:
                    print(count)

#plot for various k and universe sizes
g = sns.FacetGrid(data2, col="cache_size", row = "universe size", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for randomized sequences and various universe sizes and cache sizes")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE", linestyle = "--")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#create dataframes for each cache size
k5 = data2.loc[data2['cache_size'] == 5]
k10 = data2.loc[data2['cache_size'] == 10]
k20 = data2.loc[data2['cache_size'] == 20]

#plot for k = 5
g = sns.FacetGrid(k5, col="universe size", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for randomized sequences and various universe sizes and cache sizes")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE", linestyle = "--")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#plot for k = 10
g = sns.FacetGrid(k10, col="universe size", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for randomized sequences and various universe sizes and cache sizes")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE", linestyle = "--")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#plot for k = 20
g = sns.FacetGrid(k20, col="universe size", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for randomized sequences and various universe sizes and cache sizes")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE", linestyle = "--")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()
```

C

Python code recency-based sequences

The implementation of the Queue and algorithms is the same as in Appendix A and will not be included in this appendix.

```
#imports
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd
import seaborn as sns

from numpy import random

#implementation of MQueue and algorithms

#set parameters
universe_size = [50, 100, 150]
sequence_length = [1000]
cache_size = [5,10,20]
cache_size_protected_percent = 0.2
lookback = [5,10,20]
iters = 10000

data2 = pd.DataFrame(columns = [”universe size”, ”sequence_length”, ”cache_size”, ”proteced %
    ”, ”lookback”, ”FIFO”, ”LRU”, ”LFU”, ”SLRU”, ”UNI”, ”RRE”])

count = 0
for unsize in universe_size:
    for seqlength in sequence_length:
        for look in lookback:
            for i in range(iters):
                seq = []
                seq.append(random.randint(0, unsize))
                for j in range(seqlength-1):
                    if random.binomial(n = 1, p = 0.2) == 1:
                        lookback_seq = seq[-look::]
                        choice = random.choice(lookback_seq)
                        seq.append(choice)
                    else:
                        seq.append(random.randint(0, unsize))

                for k in cache_size:
                    cache_misses_FIFO = FIFO(seq, k)
                    cache_misses_LRU = LRU(seq, k)
                    cache_misses_LFU = LFU(seq, k)
                    cache_misses_SLRU = SLRU(seq, k*cache_size_protected_percent, k)
                    cache_misses_UNI = UNIFORM(seq, k)
                    cache_misses_RRE = RANDOM_RECENCY(seq, k)
                    OPT_misses= OPT(seq, k)

                    rat_FIFO = cache_misses_FIFO/OPT_misses
```

```
                    rat_LRU = cache_misses_LRU/OPT_misses
                    rat_LFU = cache_misses_LFU/OPT_misses
                    rat_SLRU = cache_misses_SLRU/OPT_misses
                    rat_UNI = cache_misses_UNI/OPT_misses
                    rat_RRE = cache_misses_RRE/OPT_misses

                    data2.loc[count] = unsize, seqlength, k, cache_size_protected_percent, look, rat_FIFO, rat_LRU, rat_LFU, rat_SLRU, rat_UNI, rat_RRE
                    count += 1
                    if count % 1000 == 0:

                        print(count)
#create dataframes for each universe size
u50 = data2.loc[data2['universe size'] == 50]
u100 = data2.loc[data2['universe size'] == 100]
u150 = data2.loc[data2['universe size'] == 150]

#plot for u = 50
g = sns.FacetGrid(u50, col="cache_size", row = "lookback", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for recency based sequences, n = 50, and various cache sizes and look-backs")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#plot for u = 100
g = sns.FacetGrid(u100, col="cache_size", row = "lookback", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for recency based sequences, n = 100, and various cache sizes and look-backs")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#plot for u = 150
g = sns.FacetGrid(u150, col="cache_size", row = "lookback", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for recency based sequences, n = 150, and various cache sizes and look-backs")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#create dataframes for each cache size
k5 = data2.loc[data2['cache_size'] == 5]
k10 = data2.loc[data2['cache_size'] == 10]
k20 = data2.loc[data2['cache_size'] == 20]

#plot for k = 5
g = sns.FacetGrid(k5, col="universe size", row = "lookback", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for recency based sequences, k = 5, and various universe sizes and look-backs")
```

```
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#plot for k = 10
g = sns.FacetGrid(k10, col="universe size", row = "lookback", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for
    recency based sequences, k = 10, and various universe sizes and look-backs")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()

#plot for k = 20
g = sns.FacetGrid(k10, col="universe size", row = "lookback", height = 4)
g.fig.suptitle("Ratio between number of cache misses for algorithms and optimal solution for
    recency based sequences, k = 10, and various universe sizes and look-backs")
g.map_dataframe(sns.ecdfplot, x = "FIFO", color='#00B8C8', label="FIFO")
g.map_dataframe(sns.ecdfplot, x = "LRU", color='#EF60A3', label="LRU")
g.map_dataframe(sns.ecdfplot, x = "LFU", color='#EC6842', label="LFU")
g.map_dataframe(sns.ecdfplot, x = "SLRU", color='#FFB81C', label="SLRU", linestyle = "--")
g.map_dataframe(sns.ecdfplot, x = "UNI", color='#6CC24A', label="UNI")
g.map_dataframe(sns.ecdfplot, x = "RRE", color='#0076C2', label="RRE")
g.set_xlabels("Ratio between algorithm and optimal solution")
g.set_ylabels("Proportion of sequences")
g.add_legend()
plt.show()
```

D

Python code favorite element sequences

The implementation of the Queue and algorithms is the same as in Appendix A and will not be included in this appendix. Furthermore, the creation of smaller data frames and subplots is the same as in Appendix C, so in this appendix, only the imports and generation of sequences are included.

```
#imports
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd
import seaborn as sns

from numpy import random

#implementation of MQueue and algorithms

#set parameters
universe_size = [50, 100, 150]
sequence_length = [1000]
cache_size = [5,10,20]
cache_size_protected_percent = 0.2
fav_percent = [0.1, 0.2, 0.3]
iters = 10000

data2 = pd.DataFrame(columns = [”universe size”, ”sequence_length”, ”cache_size”, ”proteced %
    ”, ”favorite %”, ”FIFO”, ”LRU”, ”LFU”, ”SLRU”, ”UNI”, ”RRE”])

count = 0
for unsize in universe_size:
    for seqlength in sequence_length:
        for fav in fav_percent:
            for i in range(iters):
                seq = []
                for j in range(seqlength):
                    if random.binomial(n = 1, p = 0.2) == 1:
                        seq.append(random.randint(0, unsize*fav))
                    else:
                        seq.append(random.randint(0, unsize))

                for k in cache_size:
                    cache_misses_FIFO = FIFO(seq, k)
                    cache_misses_LRU = LRU(seq, k)
                    cache_misses_LFU = LFU(seq, k)
                    cache_misses_SLRU = SLRU(seq, k*cache_size_protected_percent, k)
                    cache_misses_UNI = UNIFORM(seq, k)
                    cache_misses_RRE = RANDOM_RECENCY(seq, k)
                    OPT_misses= OPT(seq, k)

                    rat_FIFO = cache_misses_FIFO/OPT_misses
                    rat_LRU = cache_misses_LRU/OPT_misses
                    rat_LFU = cache_misses_LFU/OPT_misses
```

```
                    rat_SLRU = cache_misses_SLRU/OPT_misses
                    rat_UNI = cache_misses_UNI/OPT_misses
                    rat_RRE = cache_misses_RRE/OPT_misses

                    data2.loc[count] = unsize, seqlength, k, cache_size_protected_percent,
    fav, rat_FIFO, rat_LRU, rat_LFU, rat_SLRU, rat_UNI, rat_RRE
                    count += 1
                    if count % 1000 == 0:
                        print(count)

#Creating smaller data frames and corresponding plots
```

E

All results various sequences

E.1. Results randomized sequences

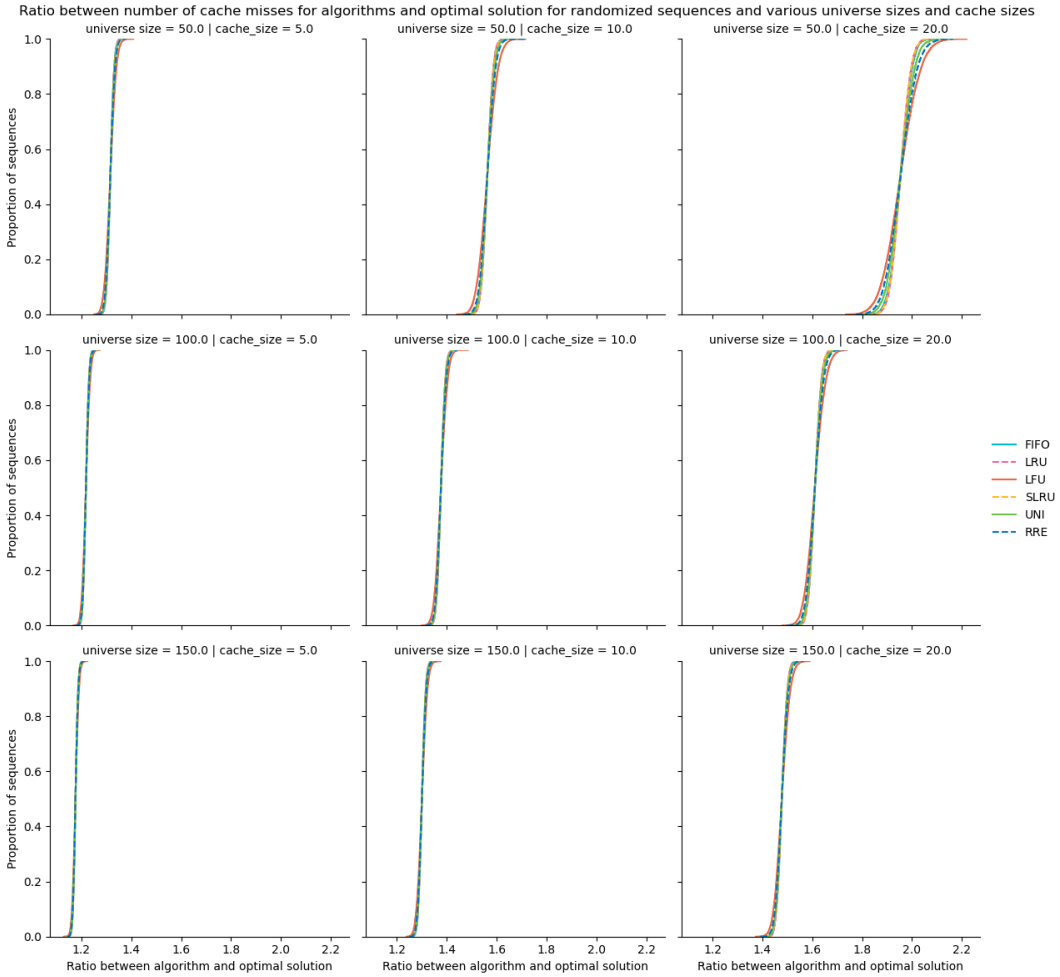


Figure E.1: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, for varying cache sizes and universe sizes.

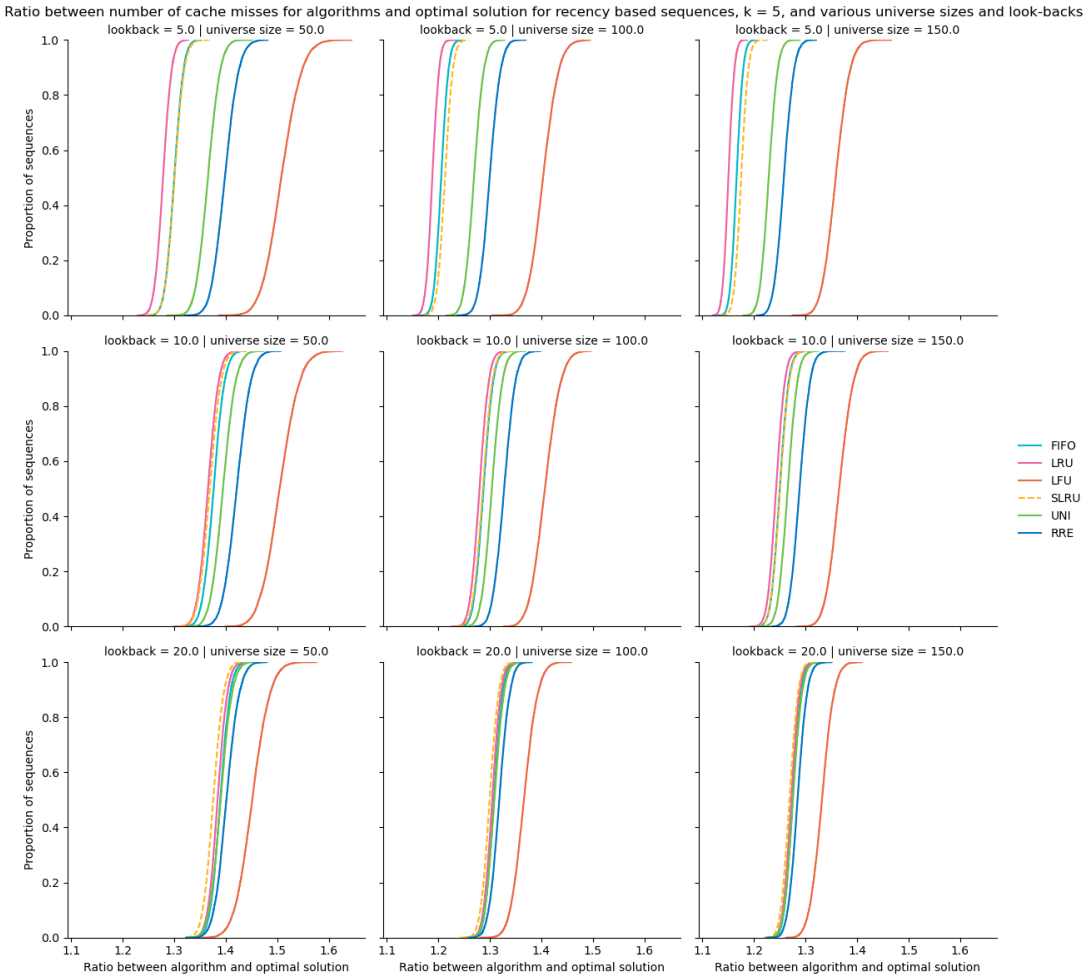


Figure E.2: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, for a fixed cache size of 5 and varying universe size and look-back.

E.2. Results recency-based sequences

E.3. Results favorite element sequences

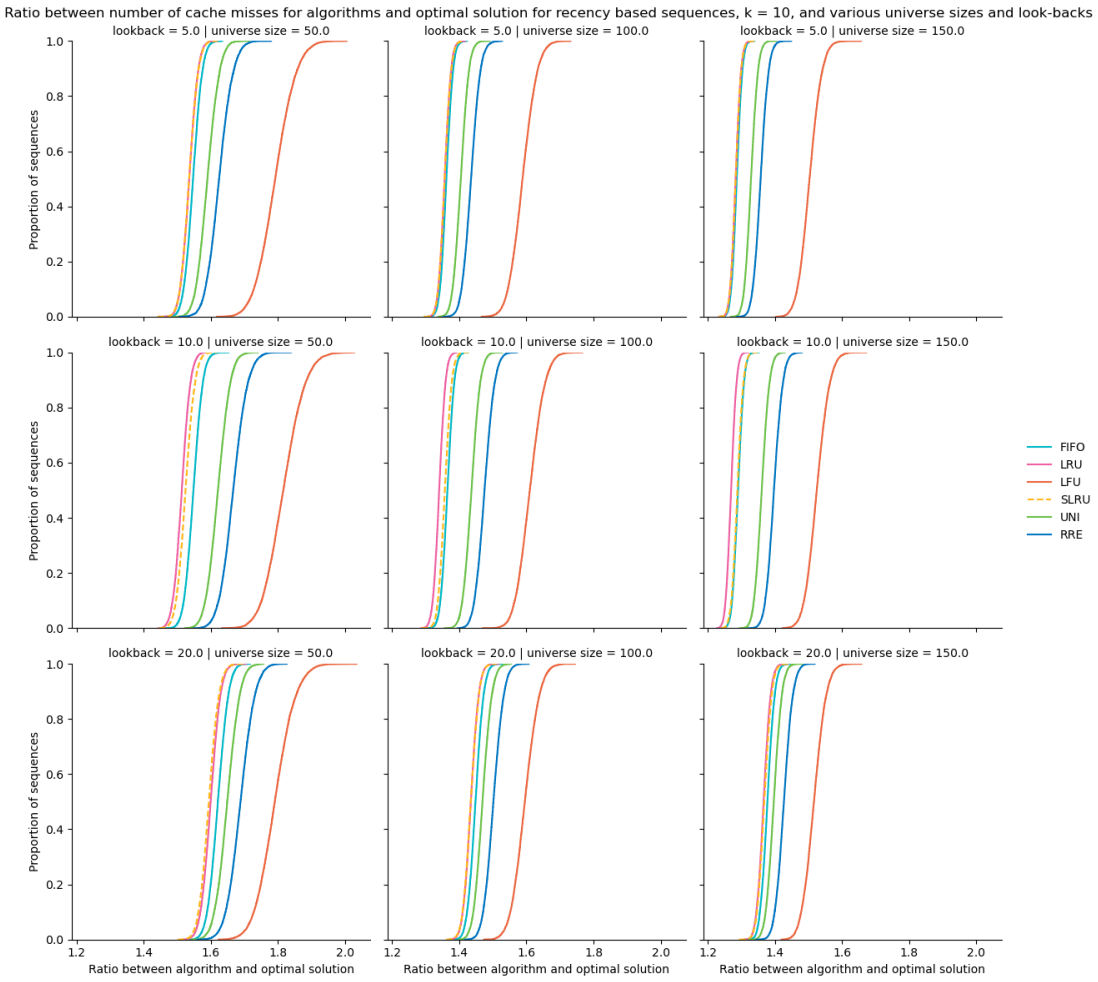


Figure E.3: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, for a fixed cache size of 10 and varying universe size and look-back.

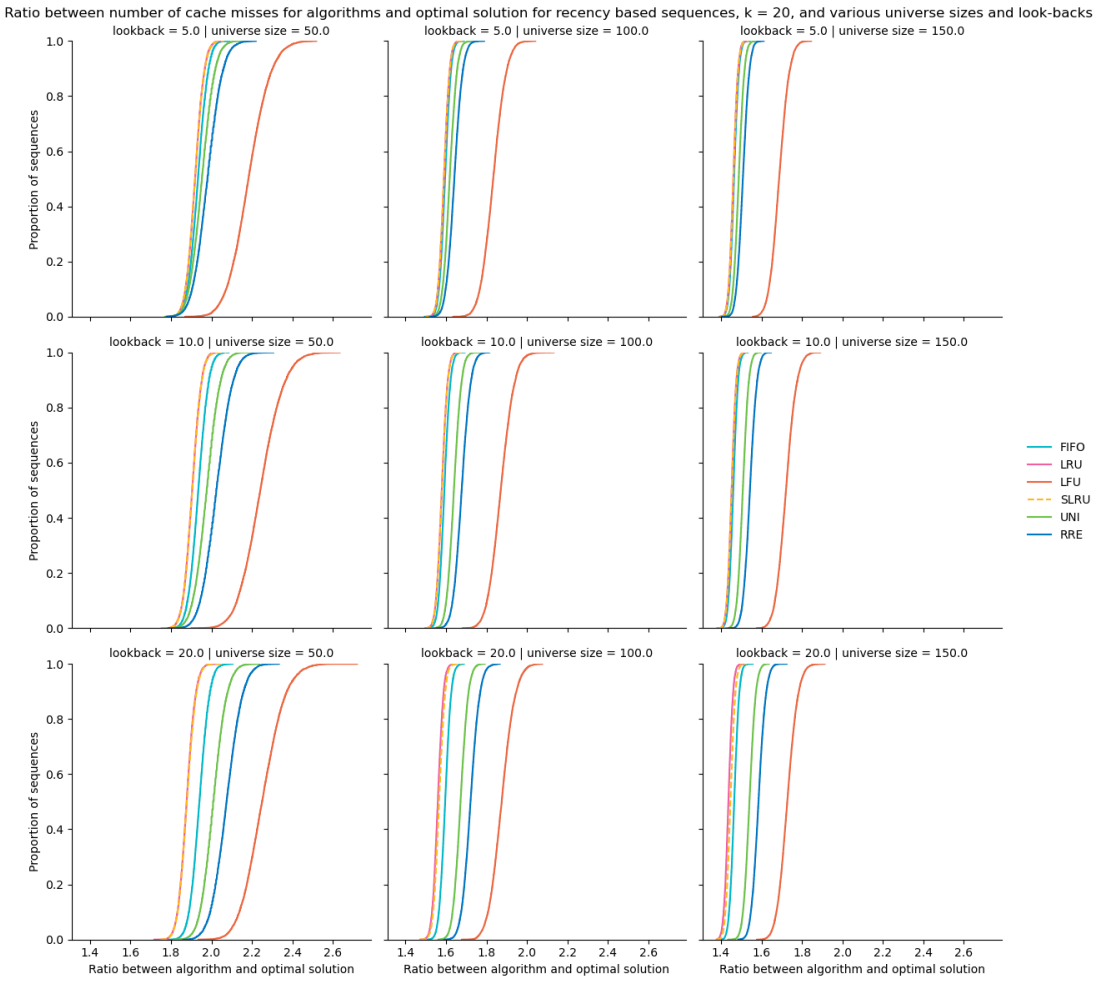


Figure E.4: Cumulative distribution for the ratio between the cost of the algorithm and the optimal algorithm, for a fixed cache size of 20 and varying universe size and look-back.

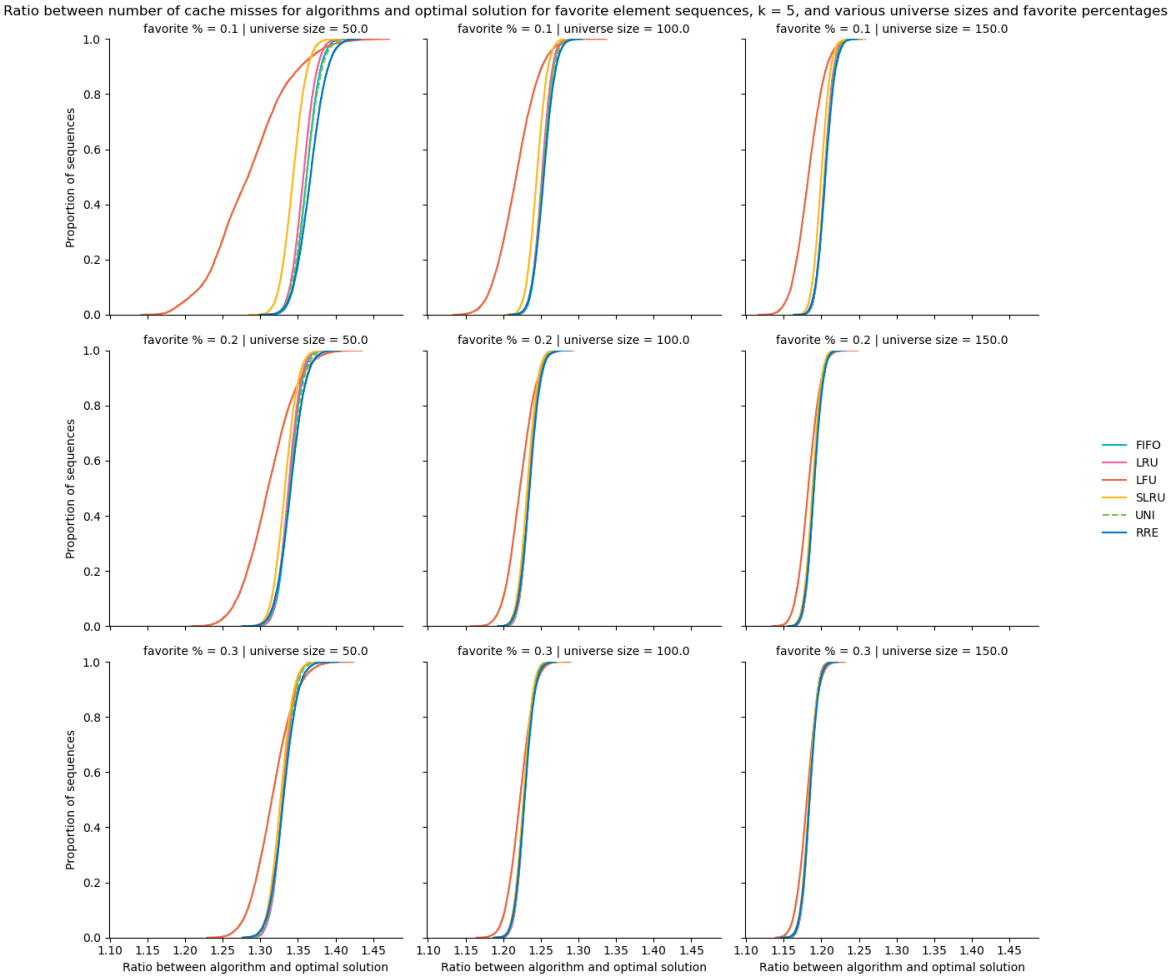


Figure E.5: Cumulative distribution for the ratio between cost of the algorithm and optimal algorithm, with cache size 5, for favorite element sequences and various universe sizes and favorite percentages.

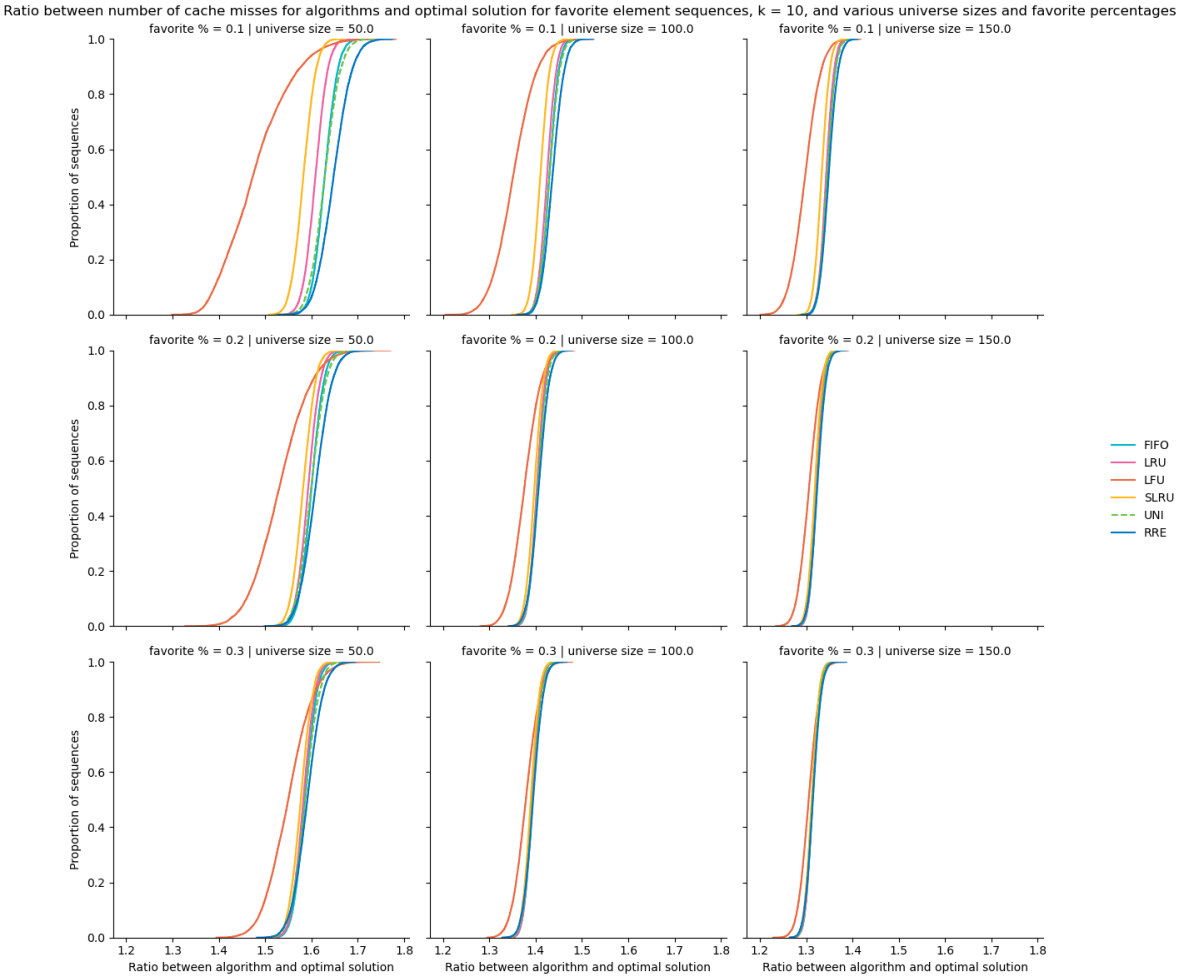


Figure E.6: Cumulative distribution for the ratio between cost of the algorithm and optimal algorithm, with cache size 10, for favorite element sequences and various universe sizes and favorite percentages.

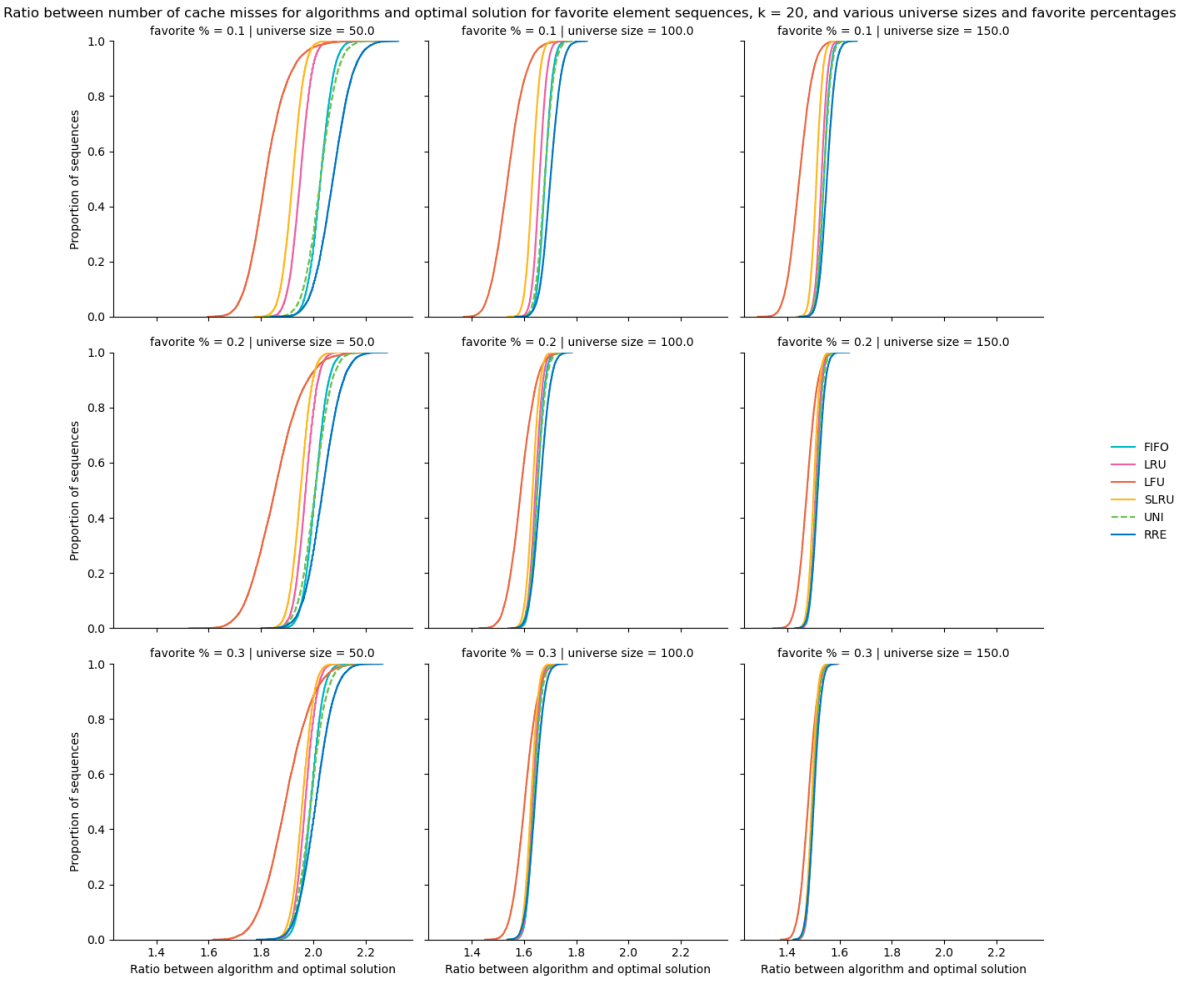


Figure E.7: Cumulative distribution for the ratio between cost of the algorithm and optimal algorithm, with cache size 20, for favorite element sequences and various universe sizes and favorite percentages.