PAGING PROBLEM WITH DIVERSITY CONSTRAINTS

CARLOS SEBASTIÁN BOZO LÓPEZ

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA,
MENCIÓN MODELAMIENTO E INTELIGENCIA ARTIFICIAL

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN MODELAMIENTO MATEMÁTICO DE DATOS

PROFESOR GUÍA:
WALDO ELÍAS GÁLVEZ VERDUGO

PROFESOR CO-GUÍA:
VÍCTOR IGNACIO VERDUGO SILVA

UNIVERSIDAD DE O'HIGGINS
DIRECCIÓN DE POSTGRADO
ESCUELA DE INGENIERÍA
MAGISTER EN CIENCIAS DE LA INGENIERÍA

RANCAGUA, CHILE
NOVIEMBRE, 2024

Abstract

In this research, we study a variant of the classical paging problem that incorporates diversity constraints. To solve the variant, we formulate a linear problem which can be efficiently rounded without increasing the cost of the solution, providing a efficient way to solve the offline problem, where the results were validated through numerical simulations even for more general settings. Finally, for the online case, we study marking algorithms and their competitive ratio against the optimal algorithm.

Resumen

En ésta investigación estudiamos una variante del problema clásico de paging que incorpora restricciones de diversidad. Para resolver dicha variante, formulamos un problema lineal que puede ser redondeado de manera eficiente sin incrementar el costo de la solución, entregando una manera eficiente de resolver el problema para el caso offline, donde los resultados fueron validados a través de simulaciones numéricas incluso para casos mas generales. Finalmente, para el caso online, estudiamos algoritmos de marking y su ratio competitivo versus el algoritmo óptimo.

Espero te sientas orgulloso papá.

Acknowledgments

First, I would like to express my deepest gratitude to my advisors, Victor and Waldo, for their unwavering support, guidance, and encouragement throughout the course of my research. Their insight and feedback have been invaluable, and this work would not have been possible without their expertise and mentorship.

I would also like to thank the members of my thesis committee, Arturo and José, for their time and willingness to assist in this process.

I am immensely grateful to the Universidad de O'Higgins for providing the professors and academics who were integral to this journey. Each of them has played a crucial role in shaping my academic path.

I would also like to extend my deepest appreciation to my family for their endless support and patience. To my parents, Sara and Roberto, and my stepfather, Juan—thank you for always believing in me and encouraging me to follow my dreams. To my friends, Mateo and Fernanda—thank you for standing by my side and providing much-needed balance throughout this journey.

Finally, I want to thank everyone who was part of this process, including those whose brief presence in my life contributed to my personal growth.

Thank you all.

Contents

List of Figures

Chapter 1

Introduction

1.1 Getting Started

In the realm of computer science, there exists a complex problem known as the paging problem. This problem arises in a computer system that is equipped with two distinct types of memory: a smaller, faster cache memory, and a larger, slower main memory. The system receives a sequence of page requests that need to be displayed. However, to display a page, it must be present in the cache memory. If it is not, a page already in the cache must be evicted, which incurs a cost for the transfer.

The goal of the paging problem is to devise a policy that minimizes the total number of evictions, thereby satisfying the page requirements dictated by the sequence. This intriguing problem was first introduced by Sleator and Tarjan in 1985 [12], who proposed a policy to address it. Since then, the problem has been extensively studied, with numerous variants being explored. These variants consider different models of request arrivals and the information available to the decision-maker. See, for instance, the survey from Irani [9].

In many real-world scenarios, such as logistics and memory management, pages are not just arbitrary data but are categorized into broader classes or types. This thesis aims to incorporate an additional dimension into the paging problem: diversity. We would like to add this in order to maintain a fair representation of the universe of requests in the cache. For example, if we are working with rapid access memory (RAM) in a PC, we would like to keep a navigator, a folder explorer instance and a document processor in memory; this way, we preserve processes covering different tasks.

Therefore, this project seeks to address the paging problem with a dual focus: minimizing cost while maintaining a cache that is diverse in terms of page types. This adds a layer of complexity to the problem, making it a compelling subject for exploration and study.

Furthermore, the concept of diversity in the paging problem opens up an entire world of research opportunities. It could lead to the development of new algorithms that balance cost and diversity. It could also pave the way for a deeper understanding of how diversity impacts system performance. Moreover, the findings from this project could have implications beyond the paging problem, potentially influencing other areas of computer science where diversity plays a crucial role. This project, therefore, stands at the intersection of theoretical computer science and practical system design, aiming to contribute valuable insights to both fields.

Through this study we are relying in an important part of practical implementations. So we can observe and investigate further on the notions given by the numerical implementations. This practical part was implemented in Python using Google Colab as the environment to work in, many algorithms and attempts were proved wrong by this computed solutions, and many work has been done building code.

We represent a solution to a given instance by showing the fast access memory (cache) at each iteration of the algorithm, this means, we explore the schedules built for each method. The amount of iterations is defined through all this study as the length of the sequence as we define what is called *reduced schedules* shown in section 3.3, it can be seen that some instances will require less iterations or "changes" and still be optimal because on some iterations the cache is not modified.

The optimal schedule does often prefer to eliminate pages that are not worth keeping, even if doing this does not decrease the cost, assuming the restriction used is keeping at most k pages on cache and not limiting the solution to have a full cache at all times, this results interesting as it shows that there are moments where the pages stored that are necessary are less than k, and that there are some iterations that require the cache to be full and some that do not.

1.2 Our Results

The primary objective of the project is to understand the challenges involved in constructing effective paging solutions that also maintain a rich and diverse cache. Achieving this balance is crucial for optimizing performance and ensuring fairness across different applications. Below, we outline the specific goals and expected outcomes of our research.

We study the impact of incorporating diversity constraints on the quality of solutions obtained by various classic models and algorithms.

We also analyze how diversity measures, particularly the well-known richness diversity indices [10], affect the performance of traditional paging algorithms. By integrating these measures, we expect to observe improvements in the balance and fairness of resource allocation, leading to more robust and equitable solutions. This provides insights into the

trade-offs between maintaining diversity and achieving optimal paging performance. Also, we investigate the complexity of the paging problem with diversity constraints, focusing on both hardness and approximation results.

We define computational boundaries of this enriched paging problem, identifying that it does not fall into the NP-hard category, and explore the feasibility of polynomial time algorithms. By establishing these complexity results, we got a better understanding of the limitations and potential of various algorithmic approaches, guiding future research and development efforts.

We develop effective solutions using both linear programming and combinatorial approaches.

- Linear programs: We formulate linear programs that incorporate diversity constraints, enabling us to solve the paging problem to optimality. These linear programs serve as benchmarks for evaluating the performance of heuristic and algorithmic solutions. By reaching optimal schedules through these formulations, we set a high standard for practical implementations.
- Algorithms: We design and implement algorithms that are not only optimal but also efficient in terms of runtime. These algorithms perform well within a reasonable time frame, making them practical for real-world applications. We explore various heuristic and exact approaches to achieve this goal, finally developing a novel algorithm that leverages the unique structure of the problem.

We design competitive algorithms for the online variant of the paging problem, where page requests are revealed incrementally over time. These algorithms make effective paging decisions in real-time, adapting to new requests as they arrive. These algorithms are evaluated based on their competitive ratios, comparing their performance to the optimal offline solutions.

These results conclude in significant implications for various fields, including computer science, logistics, and resource management. By enhancing our understanding of diversity in paging problems and developing advanced algorithms to address these challenges, we can improve the efficiency and fairness of systems that rely on dynamic resource allocation.

1.3 Related Work

We first study the classical paging problem to review on the well-known results for this problem. For this purpose, we rely pretty much on the survey of competitive paging [9], where we can check on various interesting definitions and results.

The classical paging problem involves managing a cache of fixed size k to minimize the number of page faults. In the offline setting, the entire sequence of page requests is known in advance, allowing for optimal decisions to be made.

The optimal offline paging algorithm, also known as the Longest Forward Distance (LFD) algorithm, is a theoretical algorithm that always makes the best possible decision in terms of minimizing page faults. LFD algorithm evicts the page that has the longest forward distance, based on the complete knowledge of the page request sequence, this means that to evict a page, the algorithm looks at the sequence and evicts the page that is the last to be requested among pages in the cache. This algorithm was first introduced by Belady [3].

On the other hand, the online variant issues the problem when the sequence is totally unknown, which means that we do not know what pages are going to be requested in the future. There also exist semi-online problems, where just some part of information about the sequence is unknown.

There are lower bounds on the competitive ratio for certain classes of deterministic paging algorithms in the worst-case scenario. These bounds help establish the difficulty of designing highly competitive algorithms.

For the online setting we highlight the following results:

- Deterministic algorithms: There are multiple deterministic algorithms that have been developed to solve the online paging problem, each with its strengths and limitations.
  - **First-In-First-Out (FIFO) Algorithm**: This is a simple and widely used deterministic algorithm that evicts the oldest page in the cache when a page fault occurs. While easy to implement, FIFO may suffer from the "Belady's Anomaly," where increasing the number of page frames (cache size) results in an increase in the number of page faults for a given memory access pattern.
  - **Least Recently Used (LRU) Algorithm**: Another algorithm used commonly approximates the optimal offline algorithm by evicting the least recently used page when a page fault occurs. LRU is known for its good performance in capturing temporal locality but can be challenging to implement efficiently due to its need of tracking accessed pages.

  Both of these algorithms were shown by Sleator and Tarjan [12] to be worse than the optimal offline algorithm by a factor of k, but not more, this means they are k-competitive.

  There is also introduced the so called *Marking algorithms* (which fall under the deterministic algorithms family), along with these, it is stated that any method in this family of algorithms is k-competitive[9].

It is fundamental to describe what this family of algorithms look like for this problem. As an analogue of the classic paging problem marking algorithms, we later study the marking algorithms for diverse paging.

Definition 1.1 *(Marking algorithm) We define a marking algorithm for the diversity constrained variant as follows:*

1. ***Marking**. Every time a page is requested, it gets marked.*
2. ***Serving**. If the requested page is not in cache, some unmarked page is removed to make space for the request.*
3. ***End of a phase**. If the requested page is not in cache and every page is marked, we finish the current phase, erase all the marks and start a new phase.*

Notice that now a phase can end either because k pages were marked, or because the removal of any unmarked page would violate the diversity constraint. This implies that there is no lower bound on the number of pages requested by phase, but it is still possible to prove that marking algorithms are $O(k)$-competitive, which is the best possible for deterministic online algorithms due to the existing lower bound for classical online paging [12]. This proof can be found on the section 3.6.3.

- Randomized algorithms: In addition to deterministic algorithms, there are randomized algorithms designed to address the challenges of the online paging variant. These consist in choosing randomly which page to evict under certain case criteria.

  – **Random Replacement Algorithm**: It is natural to develop an algorithm that uniformly at random chooses which page to evict, so we use a random process to discard pages. While simple and easy to implement, its performance is analyzed in expectation, and it may not guarantee strong worst-case bounds. This algorithm has a competititve ratio of k. [9]

  There is also the randomized marking type of algorithm, which simply evicts an arbitrary non-marked page on the serving stage. [9]

There are various important variants of the paging problem, each adding different complexities and considerations. Below, we describe some of the most significant ones.

- **Weighted Paging**: The key difference in this variant is that pages have weights associated with them, meaning some pages are more costly to evict than others. In weighted paging, each page is assigned a numerical weight reflecting its importance or eviction cost. The goal is to minimize not only the total number of page faults but instead the weighted cost of these faults. This variant introduces an additional layer of complexity, as the algorithm needs to balance the importance of pages with their associated weights.

Weighted paging is particularly relevant in scenarios where not all pages contribute equally to the overall performance or cost, and optimizing for weighted page faults becomes critical for efficient resource management. For instance, in a web server environment, frequently accessed resources might be assigned higher weights due to their impact on user experience. Effective strategies for weighted paging ensure that high-cost pages are evicted less frequently, thereby optimizing overall system performance, using LRU in weighted paging guarantees a $O(k)$ competitive ratio. For a deeper dive into this topic, refer to works such as [2].

- **Multi-Server Paging**: This variant involves more than one server (cache), where pages are distributed across these servers, incurring some cost for moving them. In the multi-server paging problem, the presence of multiple independent servers introduces a spatial dimension to the caching challenge. The algorithm must decide how to distribute pages among different servers, considering factors such as server capacities, communication costs, and the overall goal of minimizing the total cost associated with page faults. For this variant, LRU takes the name of distributed LRU, which guarantees a competitive ratio of $O(s \cdot k)$ [13], where s is the number of servers, while there is a more efficient way described as Work Function algorithm, achieving a competitive ratio of $O(k)$. [14]

  This variant is common in distributed systems and content delivery networks (CDNs), where multiple caching servers enhance scalability and responsiveness. Efficient strategies for redistributing pages among servers while accounting for the associated costs are crucial in optimizing the performance of such multi-server environments. These strategies need to ensure that the most frequently accessed pages are placed on the most appropriate servers, reducing latency and balancing the load across the network.

- **Min-Max**: The Min-Max variant of the paging problem focuses on minimizing the maximum number of times any page is not in cache when requested. Unlike traditional paging, which aims to minimize the total number of page faults or weighted faults, Min-Max paging seeks to ensure that the page faults are distributed evenly among the requests. This approach is particularly useful in scenarios where faults can damage the pages or their access speed. For this variant, every algorithm is $\Omega(\log n)$-competitive.

  Min-Max paging is relevant in real-time systems and mission-critical applications where predictable performance is crucial. The algorithm must be designed to handle the worst-case sequence efficiently, ensuring that the maximum cost incurred by any page is minimized. This often involves robust and adaptive strategies that can cope with a wide range of request patterns, providing consistent performance under varying conditions. This is studied in works such as [5].

- **Paging with succinct predictions**: This study is based on learning-augmented paging from the perspective of requiring the least possible amount of predicted information. More specifically, the predictions obtained along- side each page request are limited to one bit only. They consider two natural such setups:

- Discard predictions, in which the predicted bit denotes whether or not it is "safe" to evict the page.
- Phase predictions, where the bit denotes whether the current page will be requested in the next phase (for an appropriate partitioning of the input into phases).

The authors of [1] develop algorithms for each of the two setups that satisfy all three desirable properties of learning-augmented algorithms – that is, they are consistent, robust and smooth – despite being limited to a one-bit prediction per request. They also present lower bounds establishing that the algorithms presented are essentially best possible, they also establish that there is no deterministic algorithm with a competitive ratio smaller than $O(\log k)$.

1.3.1 Diversity

As it is for diversity, there are multiple notions based on different measures that will be chosen based on the requirements of the setting. Some types of diversity are:

- **Richness**: Richness is one of the most straightforward measures of diversity. It simply counts the number of different categories or species present in a dataset or solution. For this measure, we demand a particular minimum number of species, denoted as:

$$\sum_{i=1}^{S} \mathbb{1}[p_i > 0] \geq \delta$$

  Where S is the total number of species, δ is the required minimum of total species present and p_i is the proportion of individuals belonging to the i-th species. While richness does not consider the rarity of species or other more complex demands, it ensures a basic level of diversity by maintaining a minimum number of distinct categories. This measure is often used in ecological studies to assess biodiversity, and in our context, it ensures a baseline variety in the pages kept in memory or items stored in inventory.

- **Shannon Index** The Shannon Index, also known as Shannon-Wiener or Shannon-Weaver Index, is a more sophisticated measure of diversity. It takes into account both richness (the number of different species) and evenness (the distribution of individuals across species). The Shannon Index is calculated as:

$$H' = -\sum_{i=1}^{S} p_i \ln(p_i)$$

This index is widely used in various fields, from ecology to computer science, to quantify the uncertainty or entropy in a dataset. In the context of paging, using the Shannon Index can help ensure not only a variety of pages but also a balanced distribution, avoiding scenarios where a few categories dominate the cache.

- **Group Fairness**: Group Fairness introduces both lower and upper bounds on the representation of each species or category in a solution. This means we set constraints to ensure that no species is underrepresented or overrepresented, this can be expressed as:

$$L_i \leq p_i \leq U_i \quad \forall i \in S$$

Where L_i and U_i are the lower and upper bounds tied to the number of species respectively. In the context of paging, this can be used to maintain a balanced representation of different types of pages, preventing scenarios where some pages are evicted too frequently while others are kept in memory excessively. Group Fairness ensures equitable access and representation, making it particularly useful in scenarios where fairness across groups is a priority.

A slight relaxation of this variant has been considered in the context of Paging, known as Paging with reserve constraints [8]. Where the authors present a 2-approximation algorithm for the offline case, while for the online setup, they develop an $O(\log(k))$-competitive fractional algorithm, and then show how to convert it online to a randomized integral algorithm with the same guarantee.

- **Hill Numbers**: Hill Numbers represent a family of diversity indices that generalize traditional diversity measures like species richness and Shannon entropy. Denoted as D_q where q is a positive real number, Hill Numbers provide a parameterized way to measure diversity, emphasizing the contribution of rare species. The formula for Hill Numbers is:

$$D_q = \left(\sum_{i=1}^{S} p_i^q \right)^{1/(1-q)}$$

Different values of q result in different diversity measures

  - $q = 0$ corresponds to richness (number of species)
  - $q = 1$ corresponds to the exponential of Shannon entropy, which balances richness and evenness.
  - $q = 2$ corresponds to the inverse of Simpson's index, which gives more weight to common species.

Hill Numbers allow for flexibility in emphasizing different aspects of diversity. In the context of paging, this could be used to adjust the balance between maintaining a variety of pages and ensuring that frequently used pages are adequately represented.

Some of these notions can be found on earlier works as [10]. This can be implemented on combinatorial algorithms, leading to different solutions and leveraging the absence of diversity to gain an advantage.

Some relevant results in this field are:

- Diversity helps prevent premature convergence to suboptimal solutions. By introducing and maintaining diverse solutions in the population, algorithms are less likely to get stuck in local optimal, enabling continued exploration for better solutions.
- A direct implication from adding diversity to a combinatorial problem is that as we are adding more constraints, we will be working on a direct subset of the non-diverse solutions, leading to enhanced solution finding. But this could make the optimal algorithm be more "difficult" to build.
- Algorithms that solve diversity requirements are more likely to adapt to the difficulty of the optimization problem. They can dynamically adjust their search strategies based on the complexity of the problem, leading to improved performance on challenging instances.

1.3.2 Paging with Diversity Constraints

While there are few work on paging with diversity constraints, we have got enough results of both fields in order to get tools to facilitate the building of a mix-up of both aspects.

It results that a new line of research has begun to investigate how to incorporate Hill diversity notions into classical combinatorial optimization problems[11, 7]. This emerging field of study could provide valuable insights and methodologies that could be applied to the proposed inclusion of diversity for the paging problem, as they could be homologated to our researches.

As well as Hill's has been incorporated, there are papers such as [8] which are applying restrictions on the paging problem that can be seen as diversity measures, particularly to the reserves notion. So it will also be fundamental to complement their work with our investigation, and we hope to have some findings that could be useful on this area of research.

1.4 Applications

The paging problem, while originally formulated in the context of computer memory management, has a broad range of applications in various real-world scenarios. Here, we explore

some of these contexts, highlighting how the principles of paging and cache management can be effectively applied to optimize performance and efficiency.

1. **Memory management**: As the problem is defined, its application to computer memory management is straightforward and crucial. Key scenarios include:
    - **Operating Systems**: In modern operating systems, efficient memory management is critical. Paging algorithms help in deciding which pages to keep in the faster, but limited, RAM and which to store in slower disk storage. By optimizing these decisions, operating systems can significantly improve performance and responsiveness.
    - **Application Performance**: For applications that handle large datasets or perform complex computations, effective paging ensures that frequently accessed data remains readily available, reducing latency and improving user experience.
    - **Database Systems**: Databases often manage vast amounts of data, and efficient caching strategies are essential to ensure quick access to frequently queried data, thereby enhancing overall performance.
2. **Logistics**: The principles of the paging problem extend naturally to various logistical operations, where managing resources efficiently is paramount.
    - **Retail Inventory Management**: In a supermarket, shelves serve as the fast memory with limited space, while warehouses act as slow memory with larger capacity. The cost here is associated with transporting items between these two locations. An effective paging strategy can help ensure that high-demand items are readily available on shelves, reducing restocking times and improving customer satisfaction.
    - **Supply Chain Management**: In broader supply chain operations, managing the storage and transportation of goods across multiple locations can benefit from paging algorithms. These algorithms can help in deciding which items to keep in transit or in regional warehouses to optimize delivery times and reduce costs.
    - **E-commerce Fulfillment**: For online retailers, deciding which items to keep in fulfillment centers (fast memory) versus central warehouses (slow memory) is critical. Efficient paging strategies can minimize shipping times and costs, enhancing the overall customer experience.
3. **Network Caching**: In the realm of computer networks, the paging problem is highly relevant for managing data caches:
    - **Content Delivery Networks** (CDNs): CDNs cache web content at various locations to reduce latency and bandwidth usage. Efficient paging algorithms ensure that the most frequently accessed content is available close to the end users, improving load times and reducing server load.

- **Web Browsers**: Browsers cache web pages and resources to speed up loading times for frequently visited sites. Effective cache management strategies, derived from paging principles, can significantly enhance browsing experience by reducing redundant network requests.

4. **Cloud Computing**: In cloud computing environments, resource management is a critical aspect that can benefit from paging algorithms:

   - **Virtual Machine (VM) Management**: Cloud providers often need to manage the allocation of VMs across physical servers. Efficient paging strategies can help in deciding which VMs to keep active on faster resources and which to migrate or hibernate, optimizing performance and resource utilization.
   - **Data Storage and Access**: Cloud storage solutions manage large datasets across different tiers of storage with varying speeds and costs. Paging algorithms can help in deciding which data to keep on high-speed storage and which to archive, balancing cost and performance effectively.

Additionally, results such as bounds, measures of algorithm competitiveness and even dual problem findings can be useful to acquire deeper understanding of the problem's intricacies.

Chapter 2

Preliminaries

2.1 Paging Problem

In this section, we review the paging problem as it is defined in the literature and we include our own modeling.

2.1.1 Classical Paging Model

The classic paging problem is defined as follows:

1. **Parameters**:
    - k corresponds to the length of the fast memory (cache).
    - m is the number of total pages.
    - n corresponds to the total number of iterations.
    - σ is defined by a sequence $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ of page requests.
2. **Variables**:
    - p_j is the page requested on iteration j.
    - $x_{i,j}$ is a binary variable set to 1 if the page is in cache on iteration j and zero otherwise.
    - f_j is a binary variable that captures the cost of each iteration.

3. **Linear program**:

$$
\begin{aligned}
\min \quad & \sum_{j=0}^{n} f_j & & \\
\text{s.t} \quad & x_{p_j,j} = 1 & & \forall j \in [0,\dots,n] \\
& \sum_{i=0}^{m} x_{i,j} = k & & \forall j \in [0,\dots,n] \\
& f_j \geq 1 - x_{p_j,j-1} & & \forall j \in [1,\dots,n] \\
& x_{i,j} \in \{0,1\} & \forall i \in [0,\dots,m], & \forall j \in [0,\dots,n] \\
& f_j \in \{0,1\} & & \forall j \in [0,\dots,n]
\end{aligned}
$$

2.1.2 Our Model

We model the classical paging problem in a different way respect to the previous literature, based on a perspective that looks into the objective function as a basic concept that ends being a maximum that we had to linearize. The problem is defined as follows:

1. **Variables**:
   - p_j will be the page requested on iteration j.
   - $x_{i,j}$ is a binary variable that is 1 if the page i is on iteration j and zero if not.
   - $f_{i,j}$ is the cost of the page i on the iteration j.
2. **Constraints**:
   (a) First, we have to limit the amount of pages we can have in cache to k, so we apply the following constraint
   $$\sum_{i=1}^{m} x_{i,j} = k$$
   Over every iteration.
   (b) Next we need to satisfy the requests, naturally we would add
   $$x_{p_j,j} = 1$$
   But since we are looking for a complementary slackness formulation, so we add the following:
   $$x_{p_j,j} \leq 1$$
   $$-x_{p_j,j} \leq -1$$
   For every iteration.

(c) We linealize the objective function from

$$f_{i,j} = \max\{0, x_{i,j} - x_{i,j-1}\}$$

So we need to add the corresponding constraints in order to get a linear problem,

$$0 \leq f_{i,j}$$

$$x_{i,j} - x_{i,j-1} - f_{i,j} \leq 0$$

For every page and iteration.

(d) For the first iteration we need to count the first pages

$$x_{i,1} - f_{i,1} \leq 0$$

(e) Finally we add the non-negative constraint to $x_{i,j}$ and $f_{i,j}$.

$$0 \leq x_{i,j}$$
$$0 \leq f_{i,j}$$

For every page and iteration.

3. **Objective**: As the classic paging problem is defined, we count as cost just the faults, so the objective will be

$$\sum_j f_{p_j,j}$$

this is just counting the cost of the pages in the sequence on the iteration they are needed, which means we only focus on the requests, and the linearization we stated previously prevents -1 values from appearing in the objective.

4. **Lp relaxation**:

$$\begin{aligned}
\min \quad & \sum_j f_{p_j,j} && && (2.1)\\
\text{s.t} \quad & \sum_{i=1}^{m} x_{i,j} = k && \forall j \in [n]\\
& x_{p_j,j} \leq 1 && \forall j \in [n]\\
& -x_{p_j,j} \leq -1 && \forall j \in [n]\\
& x_{i,j} - x_{i,j-1} - f_{i,j} \leq 0 && \forall i \in [m], \forall j (n \geq j > 1)\\
& x_{i,1} - f_{i,1} \leq 0 && \forall i \in [m]\\
& x_{i,j}, f_{i,j} \geq 0 && \forall i \in [m], \forall j \in [n]
\end{aligned}$$

2.2 Diversity Constraints

We add the variable $y_{t,j}$ which will be 1 if the color t is present in the cache on iteration j and zero otherwise.

Known diversity approaches: As it was described previously, there are multiple types of diversity that have been studied. We will focus mainly on the following three.

- **Richness**: This notion could be said to be the most simple, it requires to have a minimum total of different types, so it does not ensure the conservation of species. Let δ be the minimum number of different species required through the solution, and let $y_{t,j}$ be a binary variable such as it takes the value of 1 if category t is present in the solution at time j and 0 otherwise. With this in mind, we can provide the following constraint:

  $$-\sum_{t} y_{t,j} \leq -\delta$$

  This models the requirement needed.

- **Reserves**: Unlike Richness, this notion has more control over each specie, this because we are asking to have at least certain number of a kind individually for every type. Take μ_t as the minimum number of elements of type t that need to be present during all the solution, then this can be formulated as

  $$\mu_t \leq \sum_{i \text{ of color } t} x_{i,j}$$

- **Group Fairness**: This is an extension of Reserves that apart from the lower bounds, it adds upper bounds to each specie, so it prevents having a great number of elements of the same kind. Assume d_t is the maximum number allowed of the same type t, then we can formulate this notion with two parts.

  $$\mu_t \leq \sum_{i \text{ of color } t} x_{i,j}$$

  $$-d_t \leq -\sum_{i \text{ of color } t} x_{i,j}$$

2.3 Paging with Richness Constrains

In the diverse paging problem with richness requirements, an instance is described as follows. We have $n \in \mathbb{N}$ number of iterations or periods, and m pages $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$. The

value σ_j is the page requested at iteration $j \in [n]$. There are $q \in \mathbb{N}$ different types, and each page is of a unique type $t \in [q]$. The set $\mathcal{P}_t \subseteq \{p_1, \ldots, p_m\}$ contains the pages of type t, and $\mathcal{P}_1, \ldots, \mathcal{P}_q$ is a partition of $\mathcal{P}$. The cache size is $k \in \mathbb{N}$, and $\delta \in \mathbb{N}$ is the richness parameter.

A *schedule* for the problem is represented by a sequence $S_1, \ldots, S_n \subseteq \{p_1, \ldots, p_m\}$ where S_j is the *cache configuration* at iteration j, and represents the pages in the cache at that moment. The schedule is *feasible* if the following properties hold:

1. **(Paging)** For every iteration $j \in [n]$, we have $\sigma_j \in S_j$, and $|S_j| = k$.
2. **(Richness)** For every iteration $j \in [n]$, we have $|\{t \in [q] : S_j \cap \mathcal{P}_t \neq \emptyset\}| \geq \delta$.

Condition 1 states the same requirements for the classic paging problem: at every iteration, the page requested must be in the cache, and the number of stored pages in the cache is exactly k. Condition 2 summarizes the diversity requirement: at every iteration, the total number of types present in the cache is at least δ. The schedule cost is the total number of times a page is added to the cache, namely, $\sum_{j=2}^{n} |S_j \setminus S_{j-1}| + |S_1|$, which also accounts for the number of page faults since $|S_1| = k$. We denote by $\text{cost}(S)$ the cost of a schedule S, and our goal is to find the feasible schedule of minimum cost, that is,

$$\min \Big\{ \text{cost}(S) : S \text{ is a schedule satisfying 1 and 2} \Big\}. \tag{2.2}$$

We call problem (2.2) the *k-server formulation* of the diverse paging problem.

Constraints: We formulate the paging problem with richness constraints by including 3 new constraints to our formulation of the classical problem, which are:

1. We have to satisfy the richness diversity measure, so we apply the following constraint,

$$-\sum_{t} y_{t,j} \leq -\delta$$

   Where this t is the color, we use this for every color and iteration.

2. Now we need to link the y and x variables, so we add

$$y_{t,j} - \sum_{\text{i of color t}} x_{i,j} \leq 0$$

   This will ensure that:

   - We can't have a $y_{t,j}$ value of 1 if there is no page of that color t in the cache.
   - We have $y_{t,j}$ to be at least 1 if there is some page of that color.

   This applies for all colors and iterations.

3. We are not interested in $y_{t,j}$ taking values greater than 1, as we define this variable to be the binary representation of the presence of the color. So we add

$$y_{t,j} \leq 1$$

for all iterations and colors.

4. Finally, we need $y_{t,j}$ to be non-negative for every color and iteration.

Linear relaxation: For the main model we use, the objective function accounts the initial cache in a different way than the first model we constructed, this makes the problem less constrained.

$$\begin{aligned}
\min \quad & \sum_{i=1}^{m}\sum_{j=2}^{n} f_{i,j} + \sum_{i=1}^{m} x_{i,1} && && (2.3)\\
\text{s.t} \quad & \sum_{i=1}^{m} x_{i,j} = k && \forall j \in [n]\\
& x_{p_j,j} \leq 1 && \forall j \in [n]\\
& -x_{p_j,j} \leq -1 && \forall j \in [n]\\
& -\sum_{t} y_{t,j} \leq -\delta && \forall t \in [q], \forall j \in [n]\\
& y_{t,j} - \sum_{\text{i of color t}} x_{i,j} \leq 0 && \forall t \in [q], \forall j \in [n]\\
& x_{i,j} - x_{i,j-1} \leq f_{i,j} && \forall i \in [m], \forall j (n \geq j > 1)\\
& x_{i,j}, y_{i,j}, f_{i,j} \geq 0 && \forall i \in [m], \forall j \in [n]
\end{aligned}$$

Reserves model: The below corresponds to the formulation of the paging problem with the memory reserve restrictions instead of using the richness measure.

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m}\sum_{j=2}^{n} f_{i,j} + \sum_{i=1}^{m} x_{i,1} && && (2.4)\\
\text{s.t} \quad & \sum_{i=1}^{m} x_{i,j} \leq k && \forall j \in [n]\\
& x_{p_j,j} \leq 1 && \forall j \in [n]\\
& -x_{p_j,j} \leq -1 && \forall j \in [n]\\
& \mu_t \leq \sum_{i \text{ of color } t} x_{i,j} && \forall t \in [q], \forall j \in [n]\\
& y_{t,j} - \sum_{i \text{ of color } t} x_{i,j} \leq 0 && \forall t \in [q], \forall j \in [n]\\
& 0 \leq f_{i,j} && \forall i \in [m], \forall j \in [n]\\
& x_{i,j} - x_{i,j-1} - f_{i,j} \leq 0 && \forall i \in [m], \forall j (n \geq j > 1)\\
& x_{i,j}, f_{i,j} \geq 0 && \forall i \in [m], \forall j \in [n]
\end{aligned}
$$

First, we start by noticing that (2.3) defines the relaxation of its integer counterpart $[IP_\delta]$ (defined by giving binary values to the variables $x_{i,j}$ and $y_{t,j}$).

Lemma 2.1 *For every instance of the diverse paging problem, the following holds:*

1. *For every feasible schedule S, there exists an integer solution (x, y, f) that is feasible for (2.3), and such that* $\text{cost}(S) = \sum_{j=2}^{n}\sum_{i=1}^{m} f_{i,j} + \sum_{i=1}^{m} x_{i,1}$.
2. *For every integer feasible solution (x, y, f) of (2.3), there exists a feasible schedule S such that* $\text{cost}(S) \leq \sum_{j=2}^{n}\sum_{i=1}^{m} f_{i,j} + \sum_{i=1}^{m} x_{i,1}$.

In particular, the value of the k-server formulation (2.3) is equal to the optimal value of its integer counterpart.

Proof. Consider a schedule S, and we define x as follows: for every $j \in [n]$, let $x_{i,j} = 1$ for every page $p_i \in S_j$, and zero otherwise. Condition 1 guarantees that x satisfies the first, second and third constraints in (2.3). For every $j \in [n]$ and every type $t \in [q]$, let $y_{t,j} = \min\{1, \sum_{i:i\in\mathcal{P}_t} x_{i,j}\} = \min\{1, |S_j \cap \mathcal{P}_t|\}$. The sixth constraint is satisfied by construction, and condition 2 implies that

$$\sum_{t=1}^{q} y_{t,j} = \sum_{t=1}^{q} \min\{1, |S_j \cap \mathcal{P}_t|\} = |\{t \in [q] : S_j \cap \mathcal{P}_t \neq \emptyset\}| \geq \delta,$$

that is, the fifth constraint is also satisfied. Finally, for every $j \in \{2, \dots, n\}$ and every $i \in [m]$, let $f_{i,j} = \max\{x_{i,j} - x_{i,j-1}, 0\}$. In particular, the fourth constraint, and non-negativity, is satisfied, and furthermore, we have $\sum_{j=2}^{n} |S_j \setminus S_{j-1}| + |S_1| = \sum_{j=2}^{n} \sum_{i=1}^{m} f_{i,j} + \sum_{i=1}^{m} x_{i,1}$. This proves 1.

Consider a feasible integer solution (x, y, f) in (2.3), and for each iteration $j \in [n]$, let $\overline{S}_j = \{p_i \in \mathcal{P} : x_{i,j} = 1\}$. The first, second and third constraints in (2.3) guarantee that the sequence $\overline{S}_1, \dots, \overline{S}_n$ meets condition 1.

By the fifth constraint, the size of $Q_j = \{t \in [q] : y_{t,j} = 1\}$ is at least δ, and since $\sum_{i:p_i \in \mathcal{P}_t} x_{i,j} = |\overline{S}_j \cap \mathcal{P}_t|$, by the sixth constraint, we have $\overline{S}_j \cap \mathcal{P}_t \neq \emptyset$ for every $t \in Q_j$, which implies $|\{t \in [q] : \overline{S}_j \cap \mathcal{P}_t \neq \emptyset\}| = |Q_j| \geq \delta$. Then, condition 2 is satisfied, and $\overline{S}$ is a feasible schedule. By construction, for every $j \in \{2, \dots, n\}$ and $i \in [m]$, we have $f_{i,j} \geq \max\{x_{i,j} - x_{i,j-1}, 0\} = |\overline{S}_j \setminus \overline{S}_{j-1}|$, thus $\text{cost}(\overline{S}) \leq \sum_{j=2}^{n} \sum_{i=1}^{m} f_{i,j} + \sum_{i=1}^{m} x_{i,1}$. This proves 2. □

We will use the following theorems as a base to prove some of our base results:

Theorem 2.2 *Let $A \in \mathbb{R}^{m \times n}$. If A is totally unimodular, then*

$$P = \{x \in \mathbb{R}^n : Ax \leq b,\ x \geq 0\} \text{ is integral } \forall b \in \mathbb{Z}^m$$

Theorem 2.3 (Ghouila-Houri) *$A \in \mathbb{R}^{m \times n}$ is totally unimodular if and only if for every subset of the rows $R \subseteq [m]$, there is a partition $R = R_1 \cup R_2$ such that for every $j \in [n]$*

$$\sum_{i \in R_1} A_{i,j} - \sum_{i \in R_2} A_{i,j} \in \{-1, 0, 1\}$$

2.4 Integral Paging Formulation

2.4.1 Total Unimodularity of the Classical Problem

An important result we got is that we have found an integer formulation for the paging problem. We state the following:

Theorem 2.4 *The formulation shown in (2.3) without richness constraints is integer.*

Proof. Let us label the restrictions:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{m}\sum_{j=2}^{n} f_{i,j} + \sum_{i=1}^{m} x_{i,1} \\
\text{s.t} \quad & x_{p_j,j} \leq 1 && \forall j \in [n] && (2.5)\\
& -x_{p_j,j} \leq -1 && \forall j \in [n] && (2.6)\\
& \sum_{i=1}^{m} x_{i,j} \leq k, \text{ and } -\sum_{i=1}^{m} x_{i,j} \leq -k && \forall j \in [n] && (2.7)\\
& x_{i,j} - x_{i,j-1} - f_{i,j} \leq 0 && \forall i \in [m], \forall j (n \geq j > 1) && (2.8)
\end{aligned}$$

For this proof, we will use Theorem 2.3.

Let us define R as an arbitrary set of constraints. If we check the rows corresponding to our LP, we will see that the constraints (2.5) and (2.6) have only one non-zero coordinate which is either 1 or -1, hence, we can always assign them to one required partition R_1 or R_2 in a consistent way after partitioning constraints (2.7) and (2.8). Meanwhile, the columns corresponding to $f_{i,j}$ have only one non-zero coordinate each, which is a -1, so they do not play a role when partitioning the rows. Furthermore, every extreme point of 2.3 is also an extreme point of the LP where we replace the first constraint by $\sum_{i=1}^{m} x_{i,j} \leq k$. Therefore, by this observations, we can restrict ourselves to analyzing the following matrix, which corresponds to the first family in (2.7) and (2.8) restricted to the variables $x_{i,j}$:

$$\left[\begin{array}{cccc|cccc|c|c}
-1 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & & \\
0 & -1 & \dots & 0 & 0 & 1 & \dots & 0 & \multicolumn{2}{c}{\multirow{3}{*}{$\dots$}} & \multirow{3}{*}{$\mathbf{0}$} \\
0 & 0 & \ddots & 0 & 0 & 0 & \ddots & 0 & & \\
0 & 0 & \dots & -1 & 0 & 0 & \dots & 1 & & \\
\hline
\multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{4}{c|}{\multirow{2}{*}{$\ddots$}} & \multicolumn{2}{c}{\multirow{2}{*}{$\ddots$}} & \multirow{2}{*}{$\mathbf{0}$} \\
\multicolumn{4}{c|}{} & \multicolumn{4}{c|}{} & \multicolumn{2}{c}{} & \\
\hline
\multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{4}{c|}{\multirow{2}{*}{$\dots$}} & -1 & 0 \dots 0 & 1 & 0 \dots 0 \\
\multicolumn{4}{c|}{} & \multicolumn{4}{c|}{} & 0 & -1 \dots 0 & 0 & 1 \dots 0 \\
\multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{4}{c|}{\multirow{2}{*}{$\dots$}} & 0 & 0 \ddots 0 & 0 & 0 \ddots 0 \\
\multicolumn{4}{c|}{} & \multicolumn{4}{c|}{} & 0 & 0 \dots -1 & 0 & 0 \dots 1 \\
\hline
1 & 1 & 1 & 1 & \multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{2}{c}{\multirow{2}{*}{$\dots$}} & \multirow{2}{*}{$\mathbf{0}$} \\
\multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{4}{c|}{} & \multicolumn{2}{c}{} & \\
\multicolumn{4}{c|}{\multirow{2}{*}{$\vdots$}} & \multicolumn{4}{c|}{\multirow{2}{*}{$\vdots$}} & \multicolumn{2}{c}{\multirow{2}{*}{$\ddots$}} & \multirow{2}{*}{$\vdots$} \\
\multicolumn{4}{c|}{} & \multicolumn{4}{c|}{} & \multicolumn{2}{c}{} & \\
\multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{4}{c|}{\multirow{2}{*}{$\mathbf{0}$}} & \multicolumn{2}{c}{\multirow{2}{*}{$\dots$}} & \multirow{2}{*}{$\mathbf{0}$} \\
\multicolumn{4}{c|}{} & \multicolumn{4}{c|}{} & \multicolumn{2}{c}{} & \\
\hline
\multicolumn{4}{c|}{} & \multicolumn{4}{c|}{} & \multicolumn{2}{c}{} & 1 \; 1 \; 1 \; 1
\end{array}\right]$$

Let $A^{(i,j)}$, with $i \in \{1, \ldots, n-1\}$ and $j \in \{1, \ldots, n\}$, denote the sub-matrix induced by the rows $(i-1)n+1, \ldots, in-1$ and columns $(j-1)n+1, \ldots, jn-1$. Observe that the sub-matrices with non-zero coordinates are all of the form $A^{(i,i)}$, which are negative identity matrices, and $A^{(i,i+1)}$, which are identity matrices. After that, we have n rows representing (2.8).

Consider now the rows corresponding to constraints (2.8) from top to bottom. We will prove by induction on $i \in \{1, \ldots, n-1\}$ the following: the rows from R among the first jn rows of the matrix can receive signs so that, for any $j \leq i$, the sum of the columns $i(n-1)+1, \ldots, in-1$ together with the corresponding row in (2.8) is in $\{-1, 0, 1\}$, and the entries in $A^{(j,j+1)}$ multiplied by the already assigned signs are either all non-negative or all non-positive. This way, we can conclude that the system is totally unimodular thanks to the Ghoulia-Houri theorem, and thus (2.3) is integral.

In the base case $j = 1$, if the first row in (2.8) is in R, we fix the signs of rows defining block $A^{(1,1)}$ so that they cancel out with the ones in (2.8); if the first row in (2.8) is not in R, then we give any sign (but the same) to every row in block $A(1,1)$ and the property is satisfied.

For the inductive step, assume that so far the first kn rows have signs so that the hypothesis is fulfilled. If the $(k+1)$-th row in (2.8) is not in R, we can assign signs to rows $kn+1, \ldots, (k+1)n-1$ in R so that they cancel out with previous rows in $A^{(k,k+1)}$ if needed, and since the entries of $A^{(k,k+1)}$ are all non-negative (resp. non-positive), we can give signs to the remaining rows so that the entries in $A^{(k+1,k+2)}$ are all either non-negative or non-positive. On the other hand, if the $(k+1)$-th row in (2.8) is in R, we first give a sign to that row so that it cancels out with block $A^{(k,k+1)}$; since the entries in the block were all non-negative (resp. non-positive), the sum of the rows in $A^{(k,k+1)}$ and the row from (2.8) with its corresponding sign will all be non-positive (resp. non-negative), and then we can proceed the same way as in the previous case to give signs to rows $kn+1, \ldots, (k+1)n-1$ in R. Finally, for block $A^{(n-1,n)}$, we give a sign to the n-th row of (2.8) to cancel out with that block (which is either all non-negative or non-positive thanks to the previous construction). □

2.4.2 Total Unimodularity of the Richness Problem

We prove that the formulation (2.3) does not satisfy total unimodularity:

Proof. Consider an instance of diverse paging with richness parameter $\delta = 2$, three requests, and three pages divided into two types. The rows corresponding to the first, third, and fifth sets of constraints in (2.3), restricted to variables $x_{i,j}$, define the following matrix:

$$
\left[\begin{array}{ccc|ccc|ccc}
-1 & 0 & 0 & 1 & 0 & 0 & & & \\
0 & -1 & 0 & 0 & 1 & 0 & & \mathbf{0} & \\
0 & 0 & -1 & 0 & 0 & 1 & & & \\
\hline
 & & & -1 & 0 & 0 & 1 & 0 & 0 \\
 & \mathbf{0} & & 0 & -1 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & -1 & 0 & 0 & 1 \\
\hline
-1 & 0 & 0 & & & & & & \\
0 & -1 & -1 & & \mathbf{0} & & & \mathbf{0} & \\
\hline
 & & & -1 & 0 & 0 & & & \\
 & \mathbf{0} & & 0 & -1 & -1 & & \mathbf{0} & \\
\hline
 & & & & & & -1 & 0 & 0 \\
 & \mathbf{0} & & & \mathbf{0} & & 0 & -1 & -1 \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}\right]
$$

To conclude, we use the following theorem by Camion [4]: A matrix with coefficients in $\{-1, 0, 1\}$ is totally unimodular if and only if the sum of the elements in each Eulerian[1] square submatrix is a multiple of 4. The proposition follows since the submatrix defined by the gray non-zero coordinates in the matrix above is Eulerian but the total sum of its coordinates is 2. □

2.5 Algorithms

We know that there is a LP for finding an optimal integer solution, so the next step is to explore and build an algorithm for finding optimal solutions in a reasonable time.

2.5.1 Introduction to Algorithms

We want to solve (2.3) with an algorithm, we suspect there must be a way to solve the problem in a reasonable time (polynomial) via an algorithm different than those used by linear program solving optimizers, so first we explore some natural extensions of the *Belady's algorithm*[3] which we will be calling LFD to include diversity constraints, specifically richness notions, as this algorithm runs in polynomial time, it makes sense that a variant could also work in this case.

[1] A matrix is Eulerian if each row and each column has even sum.

This leaded us to develop an algorithm (CLFD) which at a first glance, seemed to reach the optimal solutions, but was unable to optimally solve every instance of the problem, it failed under specific circumstances.

With this in mind, we made another algorithm (CELFD) which reached optimal solutions for that cases where CLFD failed. Sadly, it failed where CLFD was optimal, and there is no way of easily differentiating the cases where each one reaches the best solution to make a composed algorithm that chooses the best one for every instance.

Nonetheless, we show both algorithms we explore practically and the instances where they fail.

2.5.2 CLFD

First, we took the classic paging algorithm and we redefine it to handle diversity in the most natural way. We call it CLFD (Colored Longest Forward Distance), and the difference is that doing LFD as we know it, may break diversity. To fix this, we establish the algorithm can't evict pages that break diversity at each iteration, this results in an algorithm that guarantees not breaking diversity at any point at the cost of optimality.

Algorithm 1 CLFD

Require: x_{i0}, y_{t0}, c_0
Ensure: x, y, c or infeasible instance

```
j ← 1
for j ≤ n do
    x_ij ← x_ij−1                          ▷ We start the iteration with the last cache
    if x_pjj ≠ 1 then                      ▷ if the request is not in cache
        x_pjj ← 1
        for i such that x_ij = 1 do        ▷ For each page in cache
            if x_ij ≠ x_pjj then           ▷ If it isn't the page we loaded
                if evicting page i breaks diversity then
                    if it is the last option then
                        return infeasible instance
                    else
                        continue
                else
                    if page i is not in sequence then      ▷ Remove page i
                        x_ij ← 0
                        if there are no other pages of the same color then
                            y_tij ← 0
                    else
                        if i is the LDF page among the cache then      ▷ Remove page i
                            x_ij ← 0
                            if there are no other pages of the same color then
                                y_tij ← 0
                        else
                            continue
    j ← j + 1
```

As it was expressed, it fails under certain cases. This due to taking a different path than the optimal does because at some point it evicts a different page, making that at some point later it is limited to not evict the page the optimal replaces to preserve diversity. We present the proof with an example of an instance where it fails.

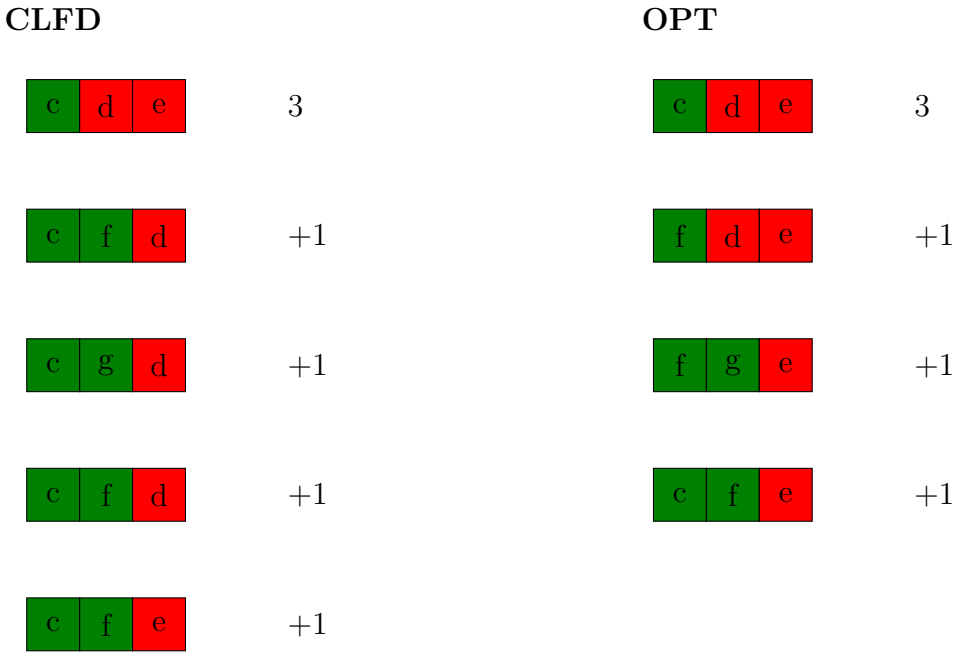
As our results showed that we failed to build an optimal algorithm, we state the following:

Lemma 2.5 *CLFD is not optimal.*

Proof. Assume the following instance:

$$\begin{aligned} \sigma &= cdefdgcfe \\ m &= 5 \\ n &= 9 \\ k &= 3 \\ \delta &= 2 \end{aligned}$$

Where c, f, g have color 0 and d, e have color 1. Then, we have to pay 3 for the first three pages cde. From this point, the optimal algorithm and CLFD start making different changes.



We can see CLFD generates a cost of 7, while the optimal algorithm costs 6. Repeating this sequence increments the cost by 4/3. □

With this proof, we discard the idea that this variation of LFD could be our desired approach, although it doesn't have a much elevated cost than the optimal, it is not what we are looking for.

Since this kind of instances are problematic for our algorithm, we explore a different approach which is more complex and sophisticated with the hope that we can find our desired result.

2.6 Examples

In the following plot, we can see both the solution of the linear problem presented, and an algorithm called CELFD, which turned out to be the worst performing algorithm that we explored. There is consistency between both graphs, making that the number of iterations shown on the left also corresponds to the number of the iteration at all times in the second plot. The colors are ordered and correspond to the page shown, with this in mind, we can have an idea of the richness of the solution by only looking at the colors displayed. The cost of each solution is shown next to the name of the solution method. For this pair of schedules, the instance associated corresponds to the following:

- Cache size $k = 6$
- Total number of colors $q = 4$
- Number of pages $m = 10$
- Number of iterations $n = 50$
- Minimum colors at each iteration $\delta = 4$
- Sequence $\sigma = 90679037742087513506295664472452737960903234853587$

Although it seems like a pretty particular instance, where it has a long sequence and a good variety of pages and colors, this is where interesting things start to happen, so it is a more general presentation rather than a small instance.

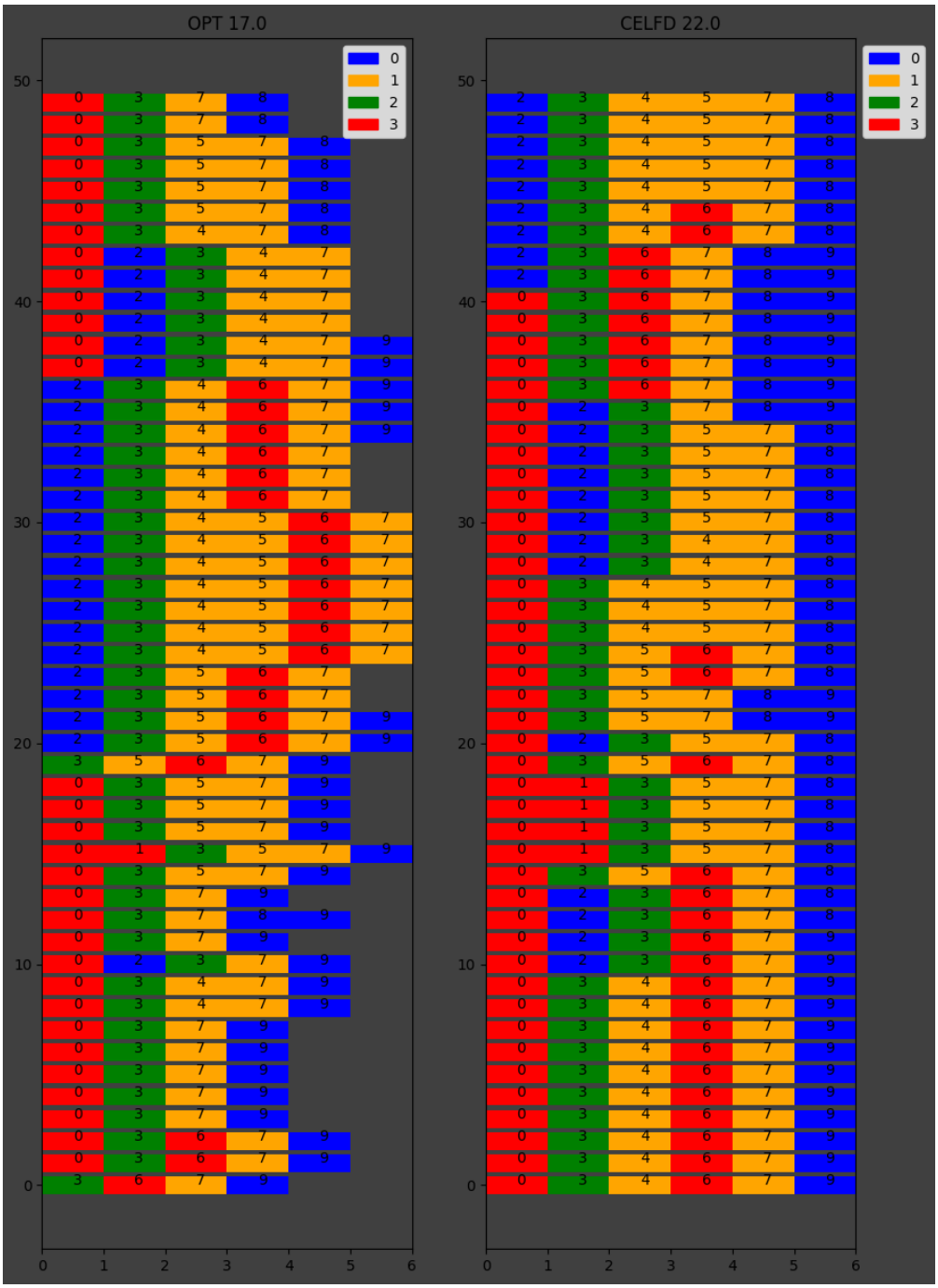


Figure 2.1: On the left, we have the optimal schedule for the instance presented, where the colors of each page are represented by the colors of the blocks, and the page is represented with the number inside the rectangle. The initial cache is on the bottom, while the final cache is on top, above this, we can see the cost for the algorithm. On the right, we have the schedule for the CELFD algorithm, with the same plot structure.

Chapter 3

Offline Paging with Richness Constraints

3.1 Linear programs

We explore the idea of linear programs applied to our problem, a pretty important part of this research is based exactly on this concept.

In this section and the problem in general, we will be using linear programs (LPs) to describe our problem due to this being the most easy and useful way, apart from the advantages we currently have to find solutions for this kind of problems.

We found that (2.3) falls under an integer set of solutions, so this concept will be particularly useful for our research.

We call a linear solution subset the family of solutions that emerge from a linear program, which is characterized by the absence of non-linear equations. This subset is particularly valuable because it can be calculated in a reasonable time frame, offering a significant computational advantage over non-linear solutions. Linear programming techniques leverage the simplicity and structure of linear relationships, enabling the use of efficient algorithms such as the simplex method (not in worst-case scenario) or interior-point methods. These algorithms are capable of handling large-scale problems with numerous variables and constraints, providing solutions quickly and reliably. By solving the LP, we can retrieve a solution from this subset.

The linear solution subset is not only computationally efficient but also offers a degree of predictability and stability that is often absent in non-linear programming. Non-linear equations can introduce complexities such as multiple local minima, non-convex regions,

and sensitivity to initial conditions, all of which can complicate the search for an optimal solution. In contrast, linear programming ensures that if a solution exists, it will be found on the boundary of the feasible region, and this solution will be globally optimal.

Moreover, the linear solution subset is highly applicable across various domains, including operations research, economics, engineering, and logistics. Problems such as resource allocation, production planning, transportation, and network flow can often be modeled effectively using linear programming. By focusing on linear solutions, we can simplify these complex problems and develop robust, scalable solutions that are practical for real-world implementation.

We call an integer solution subset the family of solutions that arise from solving an integer program, where the decision variables are constrained to take on only integer values. This subset is crucial in many real-world applications where solutions must be discrete rather than continuous. Despite the additional complexity introduced by these integer constraints, modern algorithms and computational techniques have made it possible to calculate this subset in a reasonable time for many practical problems.

Integer programming combines the linear relationships of traditional linear programming with the added requirement that some or all variables be integers. This requirement introduces a combinatorial aspect to the problem, often making it more challenging to solve than its linear counterpart. However, advancements in algorithms, such as branch and bound, branch and cut, and cutting planes, have significantly improved our ability to solve integer programs efficiently.

The integer solution subset is particularly valuable because it aligns with the nature of many real-world problems. In scenarios such as scheduling, where tasks must be assigned to time slots, or in logistics, where goods must be transported in whole units, solutions must be integers to be meaningful and implementable. Integer programming ensures that these discrete requirements are met, providing practical and actionable solutions.

While the computational complexity of integer programming is generally higher than that of linear programming, recent developments in computational power and algorithmic strategies have made it feasible to solve large-scale integer programs within acceptable time frames. Techniques such as heuristic methods, metaheuristics (like genetic algorithms and simulated annealing), and advanced exact algorithms have further enhanced the efficiency and effectiveness of finding optimal integer solutions.

Moreover, the integer solution subset offers several advantages, including the ability to model and solve problems with binary decisions (e.g., yes/no decisions) and other discrete choices that are common in various fields like finance, operations research, manufacturing, and telecommunications. This subset provides solutions that are not only optimal but also practically implementable in settings where fractional solutions would be impractical or impossible.

We propose the classical paging problem as a linear program in a different manner than the literature, exploring a new way to see the problem.

It is already proven that paging can be formulated as a linear program, which allows us to leverage the computational efficiencies associated with linear programming techniques. Moreover, an important and somewhat less recognized result is that paging can also be formulated as an integer program. This formulation opens up the application of a range of powerful integer programming techniques, enhancing our ability to find optimal solutions under specific constraints.

The ability to represent paging as an integer program is particularly significant. Integer programming allows for the precise modeling of scenarios where decisions must be discrete, such as in caching where the state of memory pages (either being in the cache or not) is inherently binary. By defining paging as an integer program, we can apply advanced optimization methods like branch and bound, branch and cut, and cutting plane methods, which are tailored to handle the combinatorial nature of integer constraints.

In the context of paging, where decisions involve choosing which pages to retain in cache and which to evict, the integer programming formulation ensures that these decisions are made optimally within the constraints of memory capacity and access patterns. This approach provides a more accurate and effective solution compared to approximate or heuristic methods that may not guarantee optimality.

To illustrate the practical application and benefits of this approach, we present the following Theorem, which demonstrates why paging can be defined as an integer problem. The proof will represent the structure and properties of the integer program, showing how it captures the essence of the paging problem.

We already proved that paging can be expressed in a linear program manner, now it is time to explore the linear program that can be obtained by adding diversity constraints to our main formulation. At the end we will also prove that this linear program with richness constraints can also be formulated as an integer program.

Apart from adding richness constraints we could explore more notions such as Shannon Index, group fairness, memory reserves and Hill numbers. But the contents of this research are focused primarily on richness.

3.2 Exchange Decompositions

Definition 3.1 *Given a schedule S, an exchange decomposition of iteration j of S, with $J \in \{2, \ldots, n\}$ is a sequence $e_1, \ldots, e_{T_j}$, with $T_j = |I_j|$, of pairs in $O_j \times I_j$ such that $(O_j \cup I_j, \{e_1, \ldots, e_{T_j}\})$ is a perfect matching. An exchange decomposition $\mathcal{M}$ of S is a*

sequence $M_2, \ldots, M_n$ *where every* M_j *is an exchange decomposition of iteration* j *in* S, $j \in \{2, \ldots, n\}$. *We denote by* $\text{out}(e)$, $\text{in}(e) \in \mathcal{P}$ *the pages so that* $e = (\text{out}(e), \text{in}(e))$ *for each exchange* e *in the decomposition.*

An exchange decomposition captures a one-by-one sequence of steps, called *exchanges* of how pages enter and leave the cache. See Figure 3.1 for an example of an exchange decomposition of an iteration.

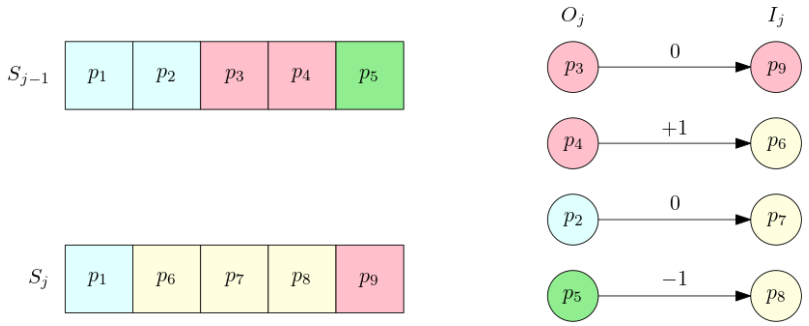


Figure 3.1: After applying the aforementioned procedure, we get an exchange decomposition of iteration j defined by $M_{j,1} = \{(p_3, p_9)\}$, $M_{j,2} = \{(p_4, p_6), (p_2, p_7)\}$, and $M_{j,3} = \{(p_5, p_8)\}$. The ordered sequence of exchanges is $e_1 = (p_9, p_3), e_2 = (p_6, p_4), e_3 = (p_7, p_2)$, and $e_4 = (p_8, p_5)$.

Definition 3.2 *The cache configuration history induced by an exchange decomposition* $\mathcal{M}$ *of* S *is the sequence* $R_0, \ldots, R_T \subseteq \mathcal{P}$, *with* $T = \sum_{j=2}^{n} |I_j|$, *such that* $R_0 = S_1$, *and* $R_r = R_{r-1} \cup \{\text{in}(e)\} \setminus \{\text{out}(e)\}$, *where* e *is the* r*-th exchange in* $\mathcal{M}$ *sorted according to iterations and the corresponding order inside the exchange decompositions of the iterations.*

The cache configuration history corresponds to the sequence of subsets of pages in the cache after every step. We say that an exchange decomposition is *feasible* if every set in the cache configuration history has richness at least δ, i.e., $|\{t \in [q] : R_r \cap \mathcal{P}_t \neq \emptyset\}| \geq \delta$ for every $r \in [T]$. The following proposition shows that a feasible exchange decomposition always exists.

The first thing to explore is the decomposition of a schedule, which will be the first step to find an algorithm. This is based on elaborating a manner to modify evicted pages in order to build a feasible schedule that does not reduce diversity when it is not necessary, this means, optimizing the schedule. We can also view this as constructing a perfect matching between pages. For this purpose we state the following:

Proposition 3.3 *Given a schedule* S, *there exists a feasible exchange decomposition of* S, *and it can be computed in time* $\text{poly}(m, n)$.

To prove Proposition 3.3, we will map a schedule to an exchange decomposition by constructing, for every iteration $j \geq 2$, a perfect matching M_j in the complete bipartite graph with node-partition given by O_j and I_j according to the following procedure:

1. For each type t, construct a maximum cardinality matching $T_{j,t}$ between the pages of type t in O_j, and the pages of type t in I_j. We call $M_{j,1}$ the set of edges given by $\cup_{t=1}^{q} T_{j,t}$, and we denote by $O_{j,1}$ and $I_{j,1}$ the pages in O_j and I_j, respectively, that are not matched in $M_{j,1}$.

2. We greedily construct a matching $M_{j,2}$ between $O_{j,1}$ and $I_{j,1}$ as follows: Let initially $M_{j,2} = \emptyset$, and S' be the cache configuration obtained after applying the exchanges from $M_{j,1} \cup M'$ to S_{j-1}. As long as there exists a page $p_o \in O_j \setminus O_{j,1}$ and a page $p_i \in I_j \setminus I_{j,1}$ such that $S' \cup \{i\} \setminus \{o\}$ has richness larger or equal than the richness of S', we add (p_o, p_i) to $M_{j,2}$, remove p_o, p_i from the current graph, update S' and continue with the procedure. We denote by $O_{j,2}$ and $I_{j,2}$ the pages in $O_j \setminus O_{j,1}$ and $I_j \setminus I_{j,1}$, respectively, that are not matched in $M_{j,2}$.

3. We let $M_{j,3}$ be an arbitrary perfect matching between $O_{j,2}$ and $I_{j,2}$.

See Figure 3.1 for an example of the matching procedure execution. This way, we obtain an exchange decomposition for each iteration $j \in \{2, \dots, n\}$, and hence an exchange decomposition of S.

Proof. Consider the exchange decomposition obtained from the previous procedure. To see that it is feasible, observe that, since S is a schedule, for each $j \in \{2, \dots, n\}$ condition 2 ensures that $|\{t \in [q] : S_j \cap \mathcal{P}_t \neq \emptyset\}| \geq \delta$, meaning that each iteration starts with a cache of richness at least δ. Then, after applying all the exchanges of the iteration, the richness will also be at least δ: By construction, exchanges of an iteration are sorted so that there is a first set of exchanges, $M_{j,1} \cup M_{j,2}$, that does not decrease the richness, and then a second set $M_{j,3}$ of exchanges that decrease richness, meaning that every intermediate cache configuration has richness at least δ in the iteration, proving the claimed result. □

3.3 Reduced Schedules

Definition 3.4 *A schedule is reduced if for every iteration $j \geq 2$, the unique page that might be added to the cache is σ_j, that is, $S_j \setminus S_{j-1} \subseteq \{\sigma_j\}$. The combinatorial formulation of the diverse paging problem is obtained by further imposing that every schedule must be reduced, namely,*

$$\min \Big\{ \text{cost}(S) : S \text{ is a reduced schedule satisfying 1 and 2} \Big\}. \tag{3.1}$$

We remark that when $\delta = 0$, we recover in (3.1) the classic paging problem. Furthermore, for the classic paging problem, it is known that the k-server and the combinatorial formulations are equivalent, as the value of (3.1) is equal to the value of (2.2).

Definition 3.5 *We define an operation $Op(a \to b)$ when page a evicts some page b, making it correspond to the pair of inserted and evicted.*

Definition 3.6 *We say two operations $Op1(p_1 \to p_2)$ and $Op2(p_3 \to p_4)$ can be matched when a swap is possible such as they now evict each others page, in this case the match would be $Op1(p_1 \to p_3)$ and $Op2(p_2 \to p_4)$.*

In this section we will prove that our classic paging problem formulation can be solved by an integer schedule, where we define and restrict our variables x and y to be binary.

This results being one of the main results of this research, which leaded us to the idea that adding richness constraints could also end in an integer program.

The next step is meant to build a reduced optimal schedule in polynomial time, because if we can reduce a schedule that we know is optimal for the diverse paging problem, it makes easier the construction of an optimal integer schedule.

Theorem 3.7 *Assume a feasible schedule S for an instance of the diverse paging problem, then, we can find a reduced schedule S' in polynomial time such that* $\text{cost}(S') \leq \text{cost}(S)$. *In particular, the value of the k-server formulation (2.2) is equal to the value of the combinatorial formulation (3.1)*

Proof. To prove this theorem, we start by decomposing a non-reduced schedule as follows: for every iteration $j = \{2, \ldots, n\}$, we have a set $I_j = S_j / S_{j-1}$ of pages that entered the cache due to request j, and a set $O_j = S_{j-1} / S_j$ of pages that were evicted from cache due to request j. Notice that I_j and O_j are always disjoint and $|I_j| = |O_j|$ as we have to maintain k pages in cache.

Consider the feasible exchange decomposition $\mathcal{M}$ of S constructed by Proposition 3.3.

For each iteration $j \in \{2, \ldots, n\}$, we say that an exchange e in M_j is *non-reduced* if $\text{in}(e) \neq \sigma_j$, i.e., it adds to the cache a page that is not the request σ_j at iteration j. We show how to construct a new schedule such that its exchange decomposition has no non-reduced exchanges.

Consider the exchange $e_{j^\star, i^\star} \in M_{j^\star}$, with $j^\star \in \{2, \ldots, n\}$ and $i^\star \in \{1, \ldots, |I_{j^\star}|\}$, that is the last non-reduced exchange in the decomposition. We construct a new schedule S' and its corresponding exchange decomposition $\mathcal{M}'$ by *delaying* the execution of $e_{j^\star, i^\star}$ as long as it is non-reduced, according to the following iterative procedure:

1. If $j^\star = n + 1$, we simply remove $e_{j^\star, i^\star}$ from the sequence, obtaining a schedule where this last exchange does not occur.
2. If $j^\star \leq n$ and $i^\star < |I_{j^\star}|$, let $e_{j^\star, i^\star + 1}$ be the exchange of $M_{j^\star}$ in position $i^* + 1$, and

consider the following four cases:

(a) If $\text{out}(e_{j^\star,i^\star}) = \text{in}(e_{j^\star,i^\star+1})$, i.e. the exchanges evict a page in cache and include it back immediately after in the same iteration, then we define a new exchange decomposition for iteration $j^\star$, namely $M'_{j^\star}$, where we remove both $e_{j^\star,i^\star}$ and $e_{j^\star,i^\star+1}$ from $M_{j^\star}$, and include a new exchange $e' = (\text{out}(e_{j^\star,i^\star+1}), \text{in}(e_{j^\star,i^\star}))$ in that position (adapting the labels of the exchanges accordingly). The delaying procedure later continues delaying e', as it is still non-reduced.

(b) If $\text{in}(e_{j^\star,i^\star}) = \text{out}(e_{j^\star,i^\star+1})$, i.e. the exchanges place a page in cache and remove it immediately after in the same iteration, then we define a new exchange decomposition for iteration $j^\star$, namely $M'_{j^\star}$, where we remove both $e_{j^\star,i^\star}$ and $e_{j^\star,i^\star+1}$ from $M_{j^\star}$, and include a new exchange $e' = (\text{out}(e_{j^\star,i^\star}), \text{in}(e_{j^\star,i^\star+1}))$ in that position (adapting the labels of the exchanges accordingly). Since $e_{j^\star,i^\star}$ was the last non-reduced exchange (and therefore $\text{in}(e_{j^\star,i^\star+1}) = \sigma_{j^\star}$), the delay ends as the new exchange becomes reduced.

(c) If none of the above happens, let $M'_{j^\star}$ be the exchange decomposition for iteration $j^\star$ obtained by swapping $e_{j^\star,i^\star}$, and $e_{j^\star,i^\star+1}$, i.e., $M'_{j^\star}$ is defined as $e'_{j^\star,i} = e_{j^\star,i}$ for every $i \notin \{i^\star, i^\star+1\}$, $e'_{j^\star,i^\star} = e_{j^\star,i^\star+1}$, and $e'_{j^\star,i^\star+1} = e_{j^\star,i^\star}$. If the obtained exchange decomposition is feasible, we replace $M_{j^\star}$ by $M'_{j^\star}$ and continue with the delay procedure.

(d) Otherwise, if the obtained exchange decomposition in (c) is not feasible, we then define another exchange decomposition $M''_{j^\star}$ as $e''_{j^\star,i} = e_{j^\star,i}$ for every $i \notin \{i^\star, i^\star+1\}$, $e''_{j^\star,i^\star} = (\text{out}(e_{j^\star,i^\star}), \text{in}(e_{j^\star,i^\star+1}))$, and $e''_{j^\star,i^\star+1} = (\text{out}(e_{j^\star,i^\star+1}), \text{in}(e_{j^\star,i^\star}))$;
see Fig. 3.2 for a depiction of this operation.

3. If $j^\star \leq n$ and $i^\star = |I_{j^\star}|$, i.e., it is the last exchange of the iteration, then we remove $e_{j^\star,i^\star}$ from $M_{j^\star}$ and include it as the first exchange in $M_{j^\star+1}$ (adapting the exchange labels accordingly). This also modifies the schedule, as now S_j is obtained without performing exchange $e_{j^\star,i^\star}$ (while the rest remains unchanged).

Claim 1 *After each step of the previous procedure, the obtained schedule is a feasible solution to diverse paging with richness requirements, and its obtained exchange decomposition is also feasible.*

If we apply this procedure for every non-reduced operation in the aforementioned order, by Claim 1, we obtain at the end a reduced schedule whose cost is not larger than the original one since the procedure does not add new exchanges. This proves the theorem.

To prove the claim, observe that if $j^\star = n+1$, then the page that this exchange includes in the cache is not requested, and hence the exchange can be safely removed. In case 2, the schedule remains unchanged, and sub-cases (a) and (b) simply merge two operations in one that is equivalent to applying both in sequence, while sub-case (c) satisfies richness

constraints by construction. For sub-case (d), observe that this can happen only if applying exchange $e_{j^\star,i^\star+1}$ first decreases the richness from δ to $\delta-1$, and then exchange $e_{j^\star,i^\star}$ brings it back to δ (as it must be feasible after the two exchanges). This implies that $e_{j^\star,i^\star+1}$ is evicting a page of a type that disappears from the cache while including a page of a type that is already in the cache, and $e_{j^\star,i^\star+1}$ is evicting a page of a type that does not disappear from cache while including a page of a type that is absent from the cache. After performing the operation in (d), then both new exchanges $(\text{out}(e_{j^\star,i^\star}), \text{in}(e_{j^\star,i^\star+1}))$ and $(\text{out}(e_{j^\star,i^\star+1}), \text{in}(e_{j^\star,i^\star}))$ do not change the number of types in the cache (see Fig. 3.2). Finally, case 3 changes the schedule but not the cache configuration history, so it remains feasible. □

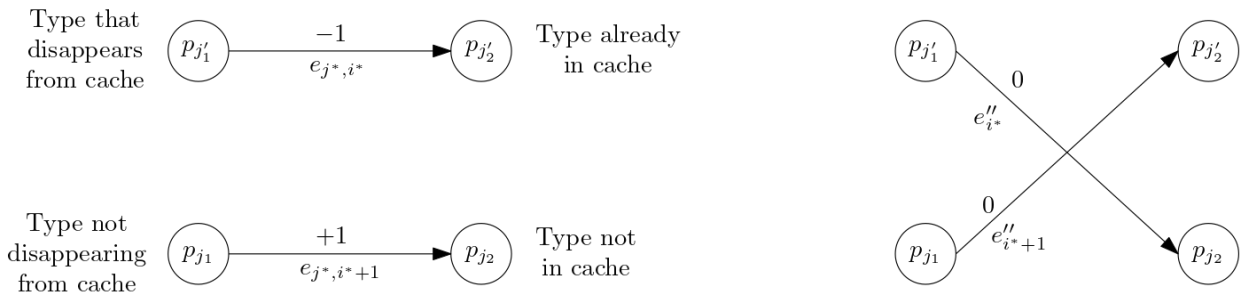


Figure 3.2: Depiction of case 2 (d) in the proof of Theorem 3.7, where numeric labels on the edges represent the change of richness induced by the exchange. If that happens, we can rematch the edges and maintain feasibility.

Definition 3.8 *We say that a feasible solution (x,y,f) of (2.3) is reduced if for every iteration* $j = \{2, \ldots, n\}$*:*

$$x_{ij} - x_{ij-1} > 0 \text{ only for } p_j = \sigma_j$$

In the second step, we show that any feasible solution to the linear program (2.3) can be modified in polynomial time and at no extra cost to get a reduced fractional solution.[1]

3.4 Fractional Diverse Paging

As we saw in the previous result, we will be looking into fractional solutions as a way to generalize the problem, but at some point we will discover a way to transform a fractional schedule into an integer schedule.

We will also explore the previous notion of exchange decompositions but this time for fractional schedules, analogous to the integer case, there is a way to decompose a fractional schedule.

[1]The cost of a feasible solution (x, y, f) in (2.3) is the objective value $\sum_{j=2}^{n}\sum_{i=1}^{m} f_{i,j} + \sum_{i=1}^{m} x_{i,1}$.

Definition 3.9 *Given a fractional schedule x, a fractional exchange decomposition M_j of iteration j, with $j \in \{2, \dots, n\}$, is a sequence $e_1, \dots, e_{T_j}$, for some $T_j \leq |I_j||O_j|$, of pairs in $O_j \times I_j$, and a function $w : \{e_1, \dots, e_{T_j}\} \to [0, 1]$, such that:*

1. $\sum_{e \in M_j : \text{in}(e) = p_i} w(e) = x_{i,j} - x_{i,j-1}$ *for every* $p_i \in I_j$,
2. $\sum_{e \in M_j : \text{out}(e) = p_i} w(e) = x_{i,j-1} - x_{i,j}$ *for every* $p_i \in O_j$,

where we denote by $\text{out}(e), \text{in}(e) \in \mathcal{P}$ *the pages so that* $e = (\text{out}(e), \text{in}(e))$ *for each exchange e in the decomposition. A fractional exchange decomposition $\mathcal{M}$ of x is defined as a sequence $M_2, \dots, M_n$, where each M_j is a fractional exchange decomposition of iteration j.*

See Figs. 3.3 and 3.4 for examples of fractional exchange decompositions. They capture, roughly speaking, how the cache changes as a sequence of fractional exchanges.

Definition 3.10 *The cache configuration history induced by a fractional exchange decomposition $\mathcal{M}$ of a fractional schedule x is the sequence $R_0, \dots, R_{|\mathcal{M}|}$, with $T = \sum_{j=2}^{n} |I_j|$, such that $R_0 = (x_{1,1}, \dots, x_{m,1})$, and R_r is obtained by replacing $x_{i,j}$ by $x_{i,j} + w(e)$, and also $x_{o,j}$ by $x_{o,j} - w(e)$ in R_r, where $e = (p_i, p_o)$ is the r-th exchange in $\mathcal{M}$ sorted according to iterations and the corresponding order inside the exchange decompositions of the iterations.*

Proposition 3.11 *Given a fractional schedule x, there exists a feasible fractional exchange decomposition of x, and it can be computed in time* $\text{poly}(m, n)$.

Proof. We first provide a procedure that constructs the fractional exchange decomposition without polynomial running time guarantees, and then modify it to obtain the desired result.

Let $\varepsilon > 0$ be a value such that $|x_{i,j} - x_{i,j-1}|$ can be written as an integer multiple of ε, for every $i \in [m]$ and $j \in \{2, \dots, n\}$, and such that $1/\varepsilon \in \mathbb{N}$. Then, for each $j \in \{2, \dots, n\}$, we will construct an auxiliar exchange decomposition where each page $p_i \in I_j$ is replaced by $\ell_{i,j} = (x_{i,j} - x_{i,j-1})/\varepsilon$ many copies of the page, that we call *ε-pages*, having the same type as i; we also replace pages $p_i \in O_j$ by $\ell'_{i,j} = (x_{i,j-1} - x_{i,j})/\varepsilon$ many ε-pages of the same type of page p_i. Hence, an exchange of the form $(p_i, p_{i'})$ over a cache configuration $(x_1, \dots, x_m)$ would correspond to decreasing x_i by ε and increasing $x_{i'}$ by ε.

If we proceed this way for every i and j, we obtain a (integral) diverse paging instance, with the only difference that each ε-page contributes ε to the cache richness (instead of one), and up to a total of at most one for each type. This implies that we can apply Theorem 3.7 to this instance and obtain a reduced schedule, but adapted for the modified contribution to richness. Indeed, we decompose iterations into a feasible exchange sequence of ε-pages analogously as in Proposition 3.3, following the procedure, except that in case (2) instead of considering the usual richness, we check whether the fractional cache configuration

$(x_1, \ldots, x_m)$ obtained after executing the exchange satisfies that $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} x_i\}$ did not decrease.

Since the obtained fractional schedule is feasible and leaves the exchanges that decrease richness at the end of each iteration, we obtain a feasible fractional exchange decomposition of ε-pages. We can easily retrieve a feasible fractional exchange decomposition for x by merging the exchanges between ε-pages that correspond to equal pairs of pages, and defining the weights as the number of merged exchanges in each case.

Instead of explicitly constructing the exchanges defined by all the ε-pages and then merging, which might lead to exponential running time due to ε being restrictively small, we will immediately construct the merged fractional exchanges at each step.

Now it is possible to describe the modifications to reach polynomial time. We construct the fractional exchange decomposition of each iteration j as follows:

1. We initialize a cache configuration $r = (x_{1,j-1}, \ldots, x_{m,j-1})$. We check whether there are pages $p_i \in I_j$ and $p_o \in O_j$ of the same type such that $\min\{x_{i,j} - r_i, r_o - x_{o,j}\} > 0$, and create an exchange $e = (p_i, p_o)$ with $w(e) = \min\{x_{i,j} - r_i, r_o - x_{o,j}\}$, update the cache configuration r, and continue.

2. If there is no pair of pages as in the previous case, we check whether there are pages $p_i \in I_j$ and $p_o \in O_j$ such that the fractional cache configuration obtained after performing an exchange $e = (p_i, p_o)$ with $w(e) = \varepsilon$ for a value $\varepsilon > 0$ arbitrarily small, namely $(r'_1, \ldots, r'_m)$, satisfies that $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r'_i\} \geq \sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r_i\}$. We then set $\alpha \leq 1$ as the maximum possible value such that, if we perform exchange $e = (p_i, p_o)$ with $w(e) = \alpha$ over r obtaining $(r'_1, \ldots, r'_m)$, it holds that $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r'_i\} \geq \sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r_i\}$, and then create exchange $e = (p_i, p_o)$ with $w(e) = \min\{x_{i,j} - r_i, r_o - x_{o,j}, \alpha\}$; we update the cache configuration and continue.

3. If there is no pair of pages as in the previous cases, we take an arbitrary pair $p_i \in I_j$ and $p_o \in O_j$ satisfying that $\min\{x_{i,j} - r_i, r_o - x_{o,j}\} > 0$, and create an exchange $e = (p_i, p_o)$ with $w(e) = \min\{x_{i,j} - r_i, r_o - x_{o,j}\} > 0$; we update the cache configuration and continue.

If we perform this procedure for each iteration, we obtain our desired fractional exchange decomposition at the end, which is furthermore feasible because we leave exchanges that decrease richness at the end of each iteration.

Observe that, by construction, every time a fractional exchange is created due to the first case, some value $r_i - x_{i,j}$ becomes zero; if it is created due to the second case, either some value $r_i - x_{i,j}$ becomes zero, or some value $\sum_{i\in\mathcal{P}_t} r_i$ becomes one; if some exchange is created in the third case, some value $r_i - x_{i,j}$ becomes zero again. Consequently, this procedure creates $O(n(m+q))$ exchanges and therefore has a polynomial running time. □

Theorem 3.12 *For every feasible solution (x, y, f) of (2.3), it is possible to compute in polynomial time a reduced feasible solution such that its cost is at most the cost of (x, y, f).*

Proof. Consider the feasible fractional exchange decomposition $\mathcal{M}$ of x constructed by Proposition 3.11. If we assume this exchange decomposition is defined for ε-pages, then we can construct a new schedule and a feasible fractional exchange decomposition for it analogously to the proof of Theorem 3.7, delaying non-reduced exchanges in reverse order, but with the only difference that we take into account the richness as $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} x_{i,j}\}$.

Indeed, if an exchange breaks feasibility as $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} x_i\} < \delta$, it is because we perform first an exchange that evicts an ε-page of a type with at most $1/\varepsilon$ many ε-pages in the cache and includes a ε-page of a type with at least $1/\varepsilon$ many ε-pages in the cache, and then we perform an exchange that evicts a ε-page of a type with at least $1/\varepsilon$ many ε-pages in the cache, and includes a ε-page of a type with at less than $1/\varepsilon$ many ε-pages in the cache. If we cross these exchanges, then we can continue delaying exactly as in Theorem 3.7.

To guarantee polynomial time, we construct the fractional exchange decomposition of each iteration j as follows:

1. We initialize a cache configuration $r = (x_{1,j-1}, \ldots, x_{m,j-1})$. We check whether there are pages $p_i \in I_j$ and $p_o \in O_j$ of the same type such that $\min\{x_{i,j} - r_i, r_o - x_{o,j}\} > 0$, and create an exchange $e = (p_i, p_o)$ with $w(e) = \min\{x_{i,j} - r_i, r_o - x_{o,j}\}$, update the cache configuration r, and continue.

2. If there is no pair of pages as in the previous case, we check whether there are pages $p_i \in I_j$ and $p_o \in O_j$ such that the fractional cache configuration obtained after performing an exchange $e = (p_i, p_o)$ with $w(e) = \varepsilon$ for a value $\varepsilon > 0$ arbitrarily small, namely $(r'_1, \ldots, r'_m)$, satisfies that $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r'_i\} \geq \sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r_i\}$. We then set $\alpha \leq 1$ as the maximum possible value such that, if we perform exchange $e = (p_i, p_o)$ with $w(e) = \alpha$ over r obtaining $(r'_1, \ldots, r'_m)$, it holds that $\sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r'_i\} \geq \sum_{t\in[q]} \min\{1, \sum_{i:p_i\in\mathcal{P}_t} r_i\}$, and then create exchange $e = (p_i, p_o)$ with $w(e) = \min\{x_{i,j} - r_i, r_o - x_{o,j}, \alpha\}$; we update the cache configuration and continue.

3. If there is no pair of pages as in the previous cases, we take an arbitrary pair $p_i \in I_j$ and $p_o \in O_j$ satisfying that $\min\{x_{i,j} - r_i, r_o - x_{o,j}\} > 0$, and create an exchange $e = (p_i, p_o)$ with $w(e) = \min\{x_{i,j} - r_i, r_o - x_{o,j}\} > 0$; we update the cache configuration and continue.

If we perform this procedure for each iteration, we obtain our desired fractional exchange decomposition at the end, which is furthermore feasible because we leave exchanges that decrease richness at the end of each iteration.

Observe that, by construction, every time a fractional exchange is created due to the first case, some value $r_i - x_{i,j}$ becomes zero; if it is created due to the second case, either some value $r_i - x_{i,j}$ becomes zero, or some value $\sum_{i \in \mathcal{P}_t} r_i$ becomes one; if some exchange is created in the third case, some value $r_i - x_{i,j}$ becomes zero again. Consequently, this procedure creates $O(n(m+q))$ exchanges and therefore has a polynomial running time. □

3.5 Schedules: Fractional to Integer

As an important step, we need to transform a fractional schedule into an integer schedule, for this purpose, we will be looking at a way for doing this in a way that does not affect the cost of the solution, which is fundamental for keeping an optimal schedule.

Proposition 3.13 *Let x be a reduced fractional schedule such that the following holds:*

1. *$x_{i,1} \in \{0,1\}$ for every $i \in \{1,\dots,m\}$, and*
2. *for every $j \in \{2,\dots,n\}$, we have $|O_j| \leq 1$.*

Then, x is integral.

Proof. We will prove by induction on the iterations that the cache configuration history consists of integral solutions. The base case comes from 1. For the inductive step, assume that the cache configuration at iteration j is integral. Since the fractional schedule is reduced, we have two cases: If $I_{j+1} = \emptyset$, no exchange is performed and the cache configuration is integral; if $|I_{j+1}| = 1$, since the cache configuration at iteration j was integral, it means the exchange is including a page that was not present in the cache configuration, and then due to 2 and the first constraint of (2.3), some page is evicted entirely from cache, obtaining an integral cache configuration. □

Theorem 3.14 *For every reduced feasible solution (x, y, f) of (2.3), it is possible to compute in polynomial time an integral feasible solution such that its cost is at most the cost of (x, y, f).*

Proof. We will first ensure the integrality of the initial cache configuration. Recall that $\sigma = \sigma_1 \dots \sigma_n$ is the request sequence, and let us define a new request sequence $\sigma' = \sigma_1\sigma_1\sigma_2 \dots \sigma_n$ (i.e., a request sequence that repeats the first request). It is not difficult to see that they both have the same optimal cost, as any feasible solution for σ can simply maintain the cache configuration for the second request of σ'.

We will modify x so as to satisfy σ' as follows. Consider the pages i such that $x_{i,1} \in (0,1)$, and let us call this set P_1. Among them, let $p_{i^\star}$ be the page that is requested again for the

first time at the latest. We will then create a new schedule $\tilde{x}$ to satisfy σ', initially equal to the aforementioned modification of x to satisfy σ', but we set $\tilde{x}_{i^\star,1} = 0$ and increase the value of the variables of a subset of pages in $P_1 \setminus \{p_{i^\star}\}$ so as to obtain back a feasible schedule. To do so, we distinguish three cases: if removing $p_{i^\star}$ from the initial cache does not violate the richness constraint, then we can increase an arbitrary subset of variables; if removing $p_{i^\star}$ from the cache breaks richness and its corresponding variable $y_{t,1}$ was originally one, due to $x_{i^\star,1}$ being fractional, there must be other pages of the same type in $P_1 \setminus \{p_{i^\star}\}$, and we can increase them to satisfy richness, and then increase any other page if needed; finally, if removing $p_{i^\star}$ from cache breaks richness and its corresponding variable $y_{t,1}$ was less than one, then due to $x_{i^\star,1}$ being fractional, there must be other types in the initial cache different to the type of $p_{i^\star}$ such that the corresponding variable $y_{t,1}$ is also strictly smaller than one (hence all the pages of that type with $x_{i,1} > 0$ are in P_1), and we can increase these variables to satisfy the richness constraint, and any other variable after that if needed.

Then, for the second σ_1 request in σ', we set all the values $\tilde{x}_{i,2}$ to be $x_{i,1}$, meaning that $p_{i^\star}$ is back in cache and the increased values in the previous operation are decreased back.

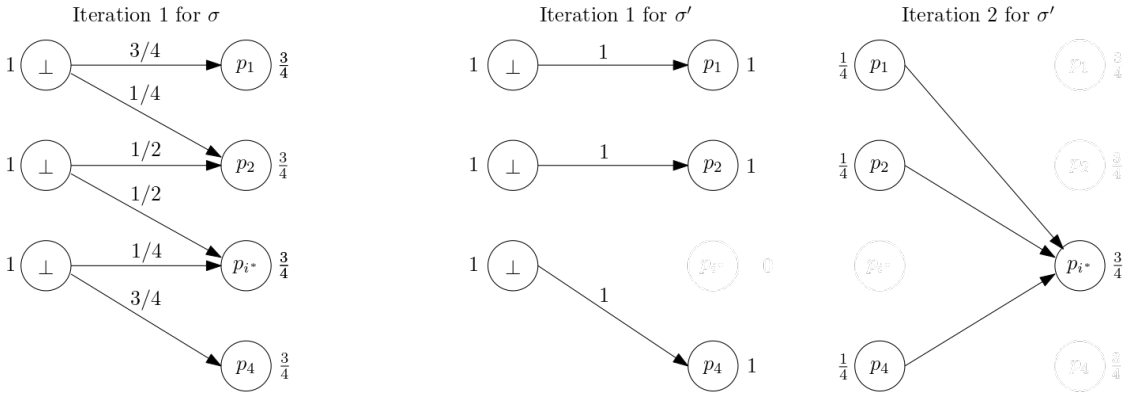


Figure 3.3: Depiction of the procedure to obtain an integral initial cache in Theorem 3.14. (Left) Fractional pages in the initial cache of the original solution are interpreted as fractional exchanges; nodes on the left represent the empty slots in the cache, and their labels represent the amount by which they leave the cache; nodes on the right represent pages entering the cache with their corresponding amounts by which their variables increase, and edges represent fractional exchanges. (Right) Decomposition of the initial operations into two iterations, so as to reduce the second one by means of Theorem 3.12.

See Fig. 3.3 for a depiction of the procedure. After doing this change, we obtain a feasible solution for σ' with one less fractional page in the initial cache compared to x, but that may not be reduced. Consequently, we reduce it by means of Lemma 3.12, obtaining a fractional reduced schedule with one less fractional page in the initial cache for σ'. Furthermore, observe that the second iteration will become irrelevant for σ' ($p_{i^\star}$ is not requested in the first iteration as $x_{i^\star,1}$ is fractional and the schedule is feasible), and if we remove it then we obtain a reduced fractional schedule with one less fractional page in the initial cache for σ.

In terms of cost, note that including page $p_{i^\star}$ back in the iteration corresponding to σ_2 has

a cost of $x_{i^\star,1}$, but then after reducing the schedule we obtain a solution that has cost smaller by at least $x_{i^\star,1}$ since, due to the choice of $p_{i^\star}$, when any page from $P_1 \setminus \{p_{i^\star}\}$ is requested, its variable has to be increased by a smaller value due to the extra amount coming from $x_{i^\star,1}$ that was transferred to them, decreasing the cost associated to that request; since $p_{i^\star}$ is not requested before any page in $P_1 \setminus \{p_{i^\star}\}$ and the original schedule was reduced, the total cost of the new reduced solution is not larger than the original one. Thus, repeating this procedure leads us to a fractional reduced schedule with an integral initial cache whose cost did not increase, i.e., 1 is satisfied.

To ensure now that in each iteration at most one page leaves the cache, we can proceed in a similar way. Let j be the first iteration where more than one page leaves the cache, namely, $p_{i_1}, \ldots, p_{i_k}$. We consider a modified request sequence $\sigma' = \sigma_1 \ldots \sigma_{j-1}\sigma_j\sigma_j\sigma_{j+1} \ldots \sigma_n$ (i.e., we repeat the request σ_j). Let $p_{i^\star}$ be the page, among the ones that are leaving the cache in iteration j, that is requested first again in the future. We will not remove $p_{i^\star}$ from cache (meaning that we set $\tilde{x}_{i^\star,j} = x_{i^\star,j-1}$), and instead evict other pages from $\{p_{i_1}, \ldots, p_{i_k}\} \setminus \{p_{i^\star}\}$ compensating for $x_{i^\star,j-1} - x_{i^\star,j}$ while preserving the richness constraint. Note first that since we have a reduced fractional schedule with an integral initial cache, the cache has been integral up to this point. If we look at the set of pages from the cache, plus the requested page σ_j, this set can have richness at least $\delta + 1$ or δ. In the first case, removing any of the pages in $\{p_{i_1}, \ldots, p_{i_k}\} \setminus \{p_{i^\star}\}$ further to compensate for $x_{i^\star,j-1} - x_{i^\star,j}$ satisfies the richness constraint; in the second case, all the pages that are being removed do not decrease the richness, and hence we can also remove any of them further to compensate for $x_{i^\star,j-1} - x_{i^\star,j}$.

After that, in the second request σ_j, we include back these pages by evicting $p_{i^\star}$ for a value of $x_{i^\star,j-1} - x_{i^\star,j}$, and then we reduce the schedule by using Theorem 3.12. See Fig. 3.4 for a depiction of the procedure. This way, we obtain a fractional reduced schedule with an integral initial cache, and we evict one less page in iteration j. In terms of cost, including back the pages has a cost of $x_{i^\star,j-1} - x_{i^\star,j}$, but then reducing produces a schedule whose cost is smaller by at least the same amount because now when $p_{i^\star}$ is requested there is no need to increase the variable as it is already one (due to being requested first in future among the removed pages in iteration j). If we continue like this, condition 2 is satisfied, and by Proposition 3.13, we end up with an integral solution to (2.3) whose cost is not higher than the original one. This concludes the proof. □

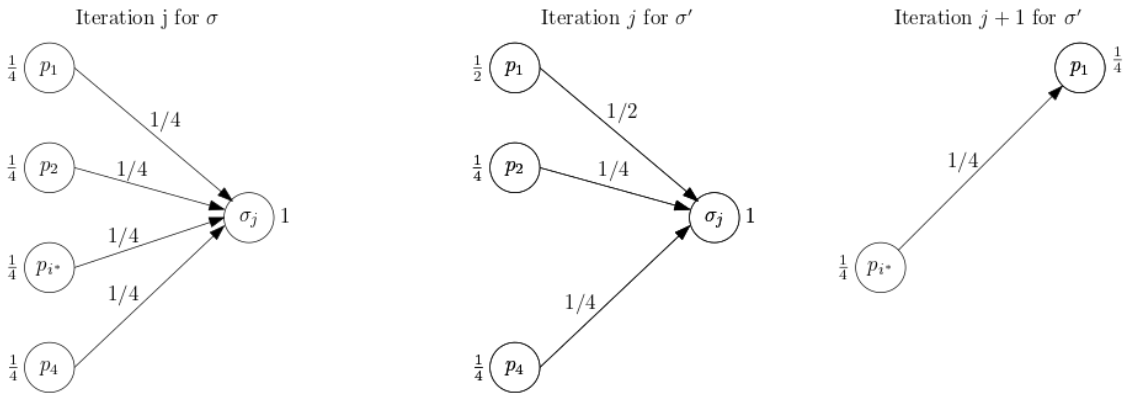


Figure 3.4: Depiction of the procedure to reduce the number of pages evicted in each iteration in Theorem 3.14. (Left) Iteration j for sequence σ, following the same notation as Figure 3.3. (Right) Decomposition of the iteration into two iterations, so as to reduce the second one by means of Theorem 3.12.

3.6 Online Paging

3.6.1 Competitive Algorithms

As for the online algorithms, results natural to find a way to compare between multiple algorithms including the optimal algorithm, so in this section we will take a look at this kind of measures.

We need a notion to compare online algorithms, the most natural way uses the offline optimal solution to measure how good the online algorithm is. So we need to define the following:

Definition 3.15 (Competitiveness) *An algorithm A is called c-competitive if there exists a constant b such that on every request sequence σ we have that*

$$cost_{k,A}(\sigma) \leq c \cdot cost_{k,OPT}(\sigma) + b$$

Where $cost_{k,OPT}(\sigma)$ resembles the cost of the optimal algorithm with cache size k on a sequence σ.

We also need an indicator to measure the faults respect to the optimal, so we define:

Definition 3.16 (Competitive ratio) *We define the competitive ratio of an algorithm A as the ratio between the number of page faults incurred by A to the number of faults incurred bu OPT in the worst-case scenario.*

We also define the competitive ratio such as for any paging algorithm A, the competitive

ratio is defined as the ratio of the number of page faults incurred by A to the number of page faults incurred by OPT, in the worst-case scenario.

3.6.2 Deterministic Algorithms

We also study deterministic algorithms for diverse paging, exploring their definitions, characteristics, and performance metrics.

To understand deterministic algorithms in the context of diverse paging, we first need to establish some foundational concepts.

Definition 3.17 (Decision rule) *A decision rule for the diverse paging problem is a rule that determines which page is evicted based on specific criteria. This rule dictates the behavior of the algorithm in a precise and predictable manner*

Definition 3.18 (Deterministic algorithm) *An algorithm is considered deterministic if it operates based on a decision rule. This implies that no matter how many times the algorithm is executed on a given instance, it will always produce the same result.*

This means a deterministic algorithm is fully predictable.

Our study also includes establishing a lower bound for the performance of deterministic algorithms in diverse paging scenarios.

Lemma 3.19 *If A is any deterministic online paging algorithm, let k be the cache size and σ any page request sequence, we have that:*

$$\text{cost}_{k,A}(\sigma) \geq k$$

Proof. In the worst-case scenario, we have a sequence with k different pages, so the algorithm must have a cost for each one, as processing the first k requests has a cost equal to the number of pages, we cannot have less cost. □

This lemma indicates that the cost (in terms of page faults) of any deterministic algorithm on any sequence of page requests is at least k, the size of the cache. This sets a fundamental performance baseline for such algorithms.

3.6.3 Marking Algorithms

It is needed to research the competitiveness of the marking algorithms, so we state the following:

Theorem 3.20 *Any marking algorithm is $O(k)$-competitive*

Proof. Let $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ be the request sequence. A marking algorithm will divide σ into phases $1, \ldots, T$. We will lower bound the cost induced by an optimal solution phase by phase, where we distinguish two cases:

- If the phase finished due to every page in the cache being marked, this means that $k+1$ different pages have been requested during this phase; consequently, the optimal solution must have evicted some page during this phase, paying a cost of at least 1.
- If the phase finished due to not being able to evict any unmarked page, this means that the set of marked pages in the phase plus the incoming request does not satisfy the diversity constraint; consequently, the optimal solution cannot have them all in cache during the whole phase and must have evicted some page, paying a cost of at least one.

Notice that any marking algorithm pays a cost of at most k per phase, and we conclude that it is k-competitive. □

3.6.4 Optimal algorithm

As we saw earlier, our variations of LFD do not work for the problem, hence, we use a procedure based on our findings to construct an optimal algorithm for the problem, although that is more difficult to implement, it leads to the following statement:

Theorem 3.21 *There exists a poly(n, m, q) algorithm to find an optimal schedule for the diverse paging problem with richness requirements* (2.2).

Proof. Let $(x^\star, y^\star, f^\star)$ be an optimal solution of the linear program (2.3), and we denote by $\mathrm{OPT}_{\mathrm{LP}}$ its value. By Lemma 3.12, we can compute in polynomial time a reduced feasible solution such that its objective cost is at most $\mathrm{OPT}_{\mathrm{LP}}$, and by Lemma 3.14 from this reduced feasible solution we can compute in polynomial time an integral feasible solution (X, Y, F) in (2.3) with objective cost $\sum_{j=2}^{n}\sum_{i=1}^{m} F_{i,j} + \sum_{i=1}^{m} X_{i,1} \leq \mathrm{OPT}_{\mathrm{LP}}$.

By Lemma 2.1, we have $\mathrm{MTS} = \mathrm{OPT}_{\mathrm{IP}}$, where MTS is the value of the k-server formulation (2.2), and $\mathrm{OPT}_{\mathrm{IP}}$ is the optimal value of the integer program $[\mathrm{IP}_\delta]$. Since (X, Y, F) is feasible for $[\mathrm{IP}_\delta]$, we have $\sum_{j=2}^{n}\sum_{i=1}^{m} F_{i,j} + \sum_{i=1}^{m} X_{i,1} \geq \mathrm{OPT}_{\mathrm{IP}}$. In general, $\mathrm{OPT}_{\mathrm{IP}} \geq \mathrm{OPT}_{\mathrm{LP}}$, and therefore we conclude that the objective cost of (X, Y, F) is equal to $\mathrm{OPT}_{\mathrm{IP}}$ and equal to $\mathrm{OPT}_{\mathrm{LP}}$. Finally, to get the optimal schedule for (2.2), we define $S_1, S_2, \ldots, S_n$ such that $S_j = \{p_i \in \mathcal{P} : X_{i,j} = 1\}$. □

We used all the theoretical proofs presented earlier in order to reach the optimal algorithm we were looking for, the important result apart from an approach like that existing is that it runs on polynomial time, proving the problem is not NP-Hard, which could be expected as the classic paging problem can also be solved on polynomial time, so we can conclude that the richness diversity measure does not affect the order of the running time for the optimal algorithm.

Chapter 4

Computational experiments

For our computational tests addressing the paging problem with diversity constraints, we utilized Python as the programming language. Python's ease of use and robust debugging capabilities made it an ideal choice for implementing and testing our algorithms.

To solve the linear program (LP) formulated for the problem, we employed the Gurobi optimization framework. Gurobi is renowned for its efficiency in solving large-scale linear and integer programming problems, making it a perfect fit for our needs. The computations were executed in a Google Colab environment, which provided several advantages:

- **Speed and efficiency**: Google Colab leverages cloud computing resources, ensuring that our calculations are performed as quickly as possible.
- **Accessibility and collaboration**: The cloud-based environment facilitates remote work and collaboration, allowing team members to share and review results seamlessly.

We chose a Python notebook environment for our development and testing processes. This choice was motivated by several factors:

- **Structured workflow**: Notebooks provide a structured environment that allows for incremental development and testing. Code, documentation, and results can be interleaved, making the workflow more intuitive and easier to manage.
- **Interactive debugging**: The interactive nature of notebooks makes it easier to debug code, test different scenarios, and visualize intermediate results.
- **Documentation and presentation**: Notebooks serve as both a development tool and a presentation medium, combining executable code with rich text, equations, and visualizations.

For visualizing the results of our algorithms and LP solutions, we utilized Matplotlib, a high-level plotting library in Python. Matplotlib's capabilities enabled us to create detailed and precise plots for each instance solution, facilitating a clear and comprehensive analysis of the outcomes. Some key benefits included:

- **High-Level plotting**: Matplotlib provides a straightforward interface for creating a wide range of plots, from simple line graphs to complex multi-plot figures.
- **Customization**: The library offers extensive customization options, allowing us to tailor the visualizations to highlight specific aspects of the solutions.
- **Integration with Notebooks**: Matplotlib integrates seamlessly with Google Colab, enabling us to display visualizations directly alongside the code and analysis.

To structure the code we used multiple functions for each operation needed (create models, solve models, calculate gaps, plot solutions, etc). So at the end we just called functions in a linear code to perform all needed steps.

4.1 Linear Program Results

We will be using the instance shown in page 26 for this example.

We know this is the optimal for the instance we are looking into because:

1. The model (2.3) we are using represents the problem of paging with diversity (richness constraints).
2. The optimizer ensures to find an optimal for the model given.

So we are sure this is the optimal for the instance, moreover, we can calculate the optimal for any given instance.

The first thing we noticed is that solving the model always retrieved an integer solution, we tested for around 100.000 different random instances satisfying the definition for a diverse paging problem, and the optimal schedule always showed integer, so this led us to the idea of an optimal integer solution being possible for any given instance.

The second thing we noticed was that when the restriction of having k pages was relaxed to be at most k pages, the optimal schedule did not use all the space at every iteration, so this awakened the concept of some pages not being relevant at some points, we can confirm

this by simply looking at the start of the schedule, where we only need to cover the richness constraint and do not lose cost for not having future pages in the cache.

As long as we are allowed to introduce dummy pages, we can convert any non-feasible instance into a feasible instance by adding a dummy page of the color missing, which will always incur in some cost due to the dummy page having to be always in cache, the solution will be equivalent to reducing the cache size to $k - 1$ and decreasing δ to $\delta - 1$.

See Figure 4.1 for the solution to the instance, where we can confirm all mentioned points.

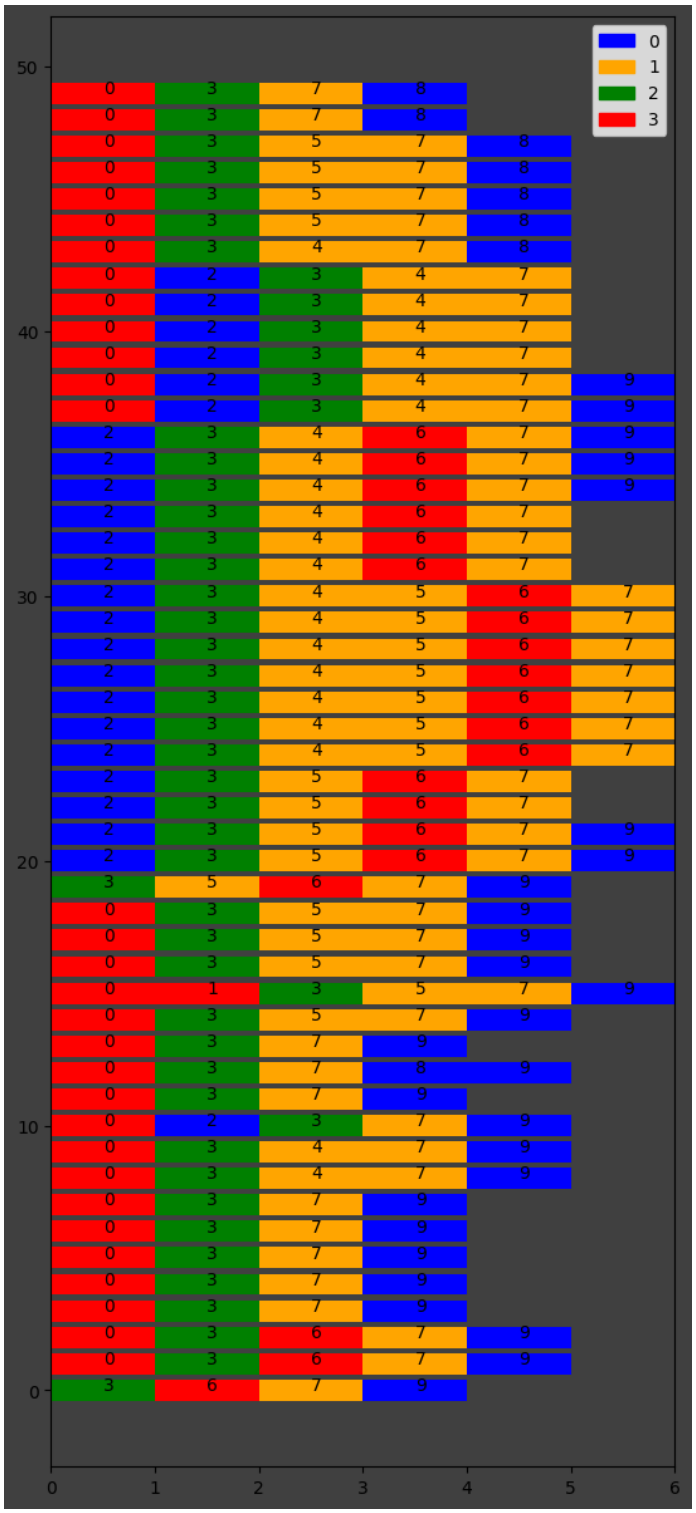


Figure 4.1: Optimal schedule for some arbitrary instance.

4.2 Algorithm Comparison

As for the more studied CLFD algorithm, we know that it never breaks diversity by construction, although we can see that it takes different paths compared to the optimal schedule, in most cases the cost is the same, but there are some cases that at some point it makes a different eviction compromising richness later and having to keep a different page than the optimal does, hence, the cost increases as it limited its moves. See Figure 4.2 for the graphical description of this point.

Another difference is that at the start of the instance, by construction CLFD decides to include the first k pages that satisfy richness, unlike the optimal algorithm it starts with a full cache and evicts more pages later, making use of a not totally reduced schedule. Although, it satisfies being a reduced schedule at some point, specifically it always end with δ pages as it does not require more than that to satisfy the conditions for the problem.

Like for the instance we decided to display, there are many other instances where the cost coincides with the optimal cost, this for the small instances explored at first was inducing us to think that CLFD was optimal, but as we tried to prove this claim and explored more complicated instances, we eventually found an instance where our algorithm was sub-optimal and after an analysis we were able to determine a simple instance where CLFD failed (the one shown in Lemma 2.5).

This intuition of sub-optimality could be hinted by the comparison shown in the first point, but it was not fully clear as there could be multiple optimal solutions. With this in mind we decided to explore different approaches for our goal of developing an optimal algorithm, our next step landed on CELFD as we were also looking for a more theoretical way.

As CELFD quickly showed to be sub-optimal, this branch was discarded and we fully focused on the theoretical way, which finally led us to the optimal algorithm we presented on the previous section.

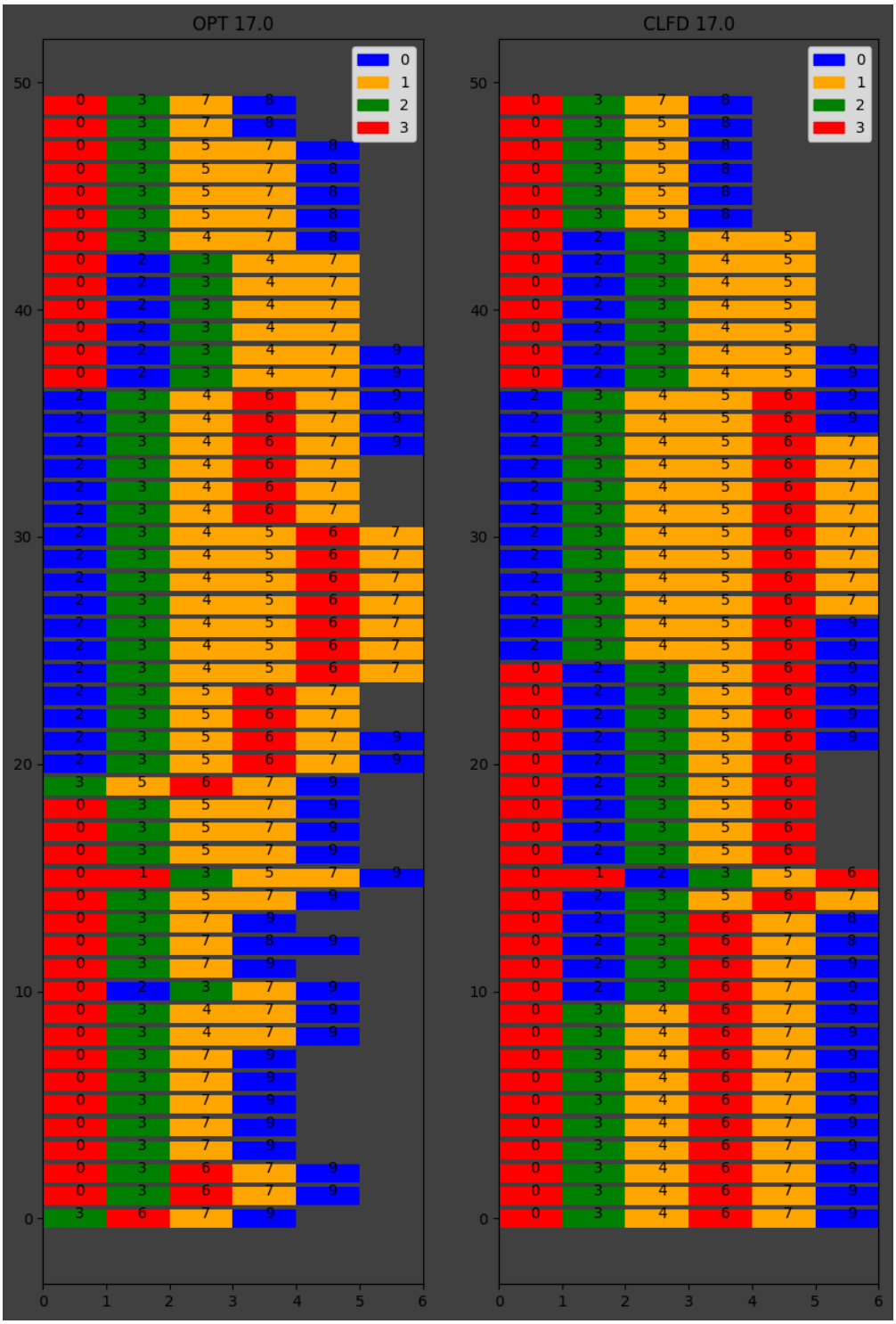


Figure 4.2: Optimal schedule compared to the CLFD algorithm schedule.

Chapter 5

Conclusions and future work

5.1 Conclusions

In this section, we will discuss our findings and their impact on the current body of knowledge in the field of paging problems, both offline and online. Our research has provided valuable insights and advancements in understanding and solving these problems, particularly with the inclusion of diversity constraints.

5.1.1 Offline Paging

For the offline paging results, our research dived into two main areas: linear programs and algorithms.

Our investigation into linear programs revealed that the classical paging problem can indeed be formulated as an integer linear program. This is a significant finding, as integer programming offers a robust framework for solving discrete optimization problems. Despite its potential, this approach is not widely used or reviewed in the literature, highlighting an area ripe for further exploration. The integer formulation is especially valuable when no other algorithmic solution exists for the problem at hand.

Additionally, our study on the paging problem with richness constraints demonstrated that it also results in an integer program. This suggests that in practical scenarios, optimal fractional schedules are rare, and solutions will predominantly be integer-based. This conclusion depends heavily on the optimizer used, as different optimizers may yield varying results.

In our algorithmic exploration, we developed two algorithms that, while not optimal for

the paging problem with diversity constraints, opened the way for discovering the optimal algorithm. The optimal algorithm we identified runs in polynomial time and is highly efficient. As it addresses the paging problem with diversity constraints this means it also shows potential for solving a wide range of memory allocation problems involving various types. This has significant implications for real-world logistics and resource management problems, where optimal memory allocation is crucial.

5.1.2 Online Paging

For the online paging results, our research focused on three main axes: marking algorithms, deterministic algorithms, and randomized algorithms.

We evaluated the competitiveness of marking algorithms and established a clear understanding of their performance in online paging scenarios. Our findings provide a benchmark for what can be expected from this family of heuristics. These results are valuable for practitioners seeking reliable, near-optimal solutions in dynamic and unpredictable environments.

In the realm of deterministic algorithms, we identified a lower bound that serves as a baseline for the best possible outcomes achievable with this approach. This baseline is crucial for benchmarking and improving existing deterministic algorithms, guiding future research towards more effective and efficient solutions.

Our exploration of randomized algorithms, while less conclusive, underscored the challenges inherent in developing these types of solutions. The inherent unpredictability and complexity of randomized algorithms make them difficult to design and analyze. However, this also points to an exciting avenue for future research, as advancements in this area could lead to breakthroughs in handling highly dynamic and uncertain paging scenarios.

Overall, our research has significantly advanced the understanding of paging problems, particularly with the incorporation of diversity constraints. The findings offer new theoretical insights and practical algorithms that can be applied to a variety of real-world problems. By establishing the feasibility of integer programming in these contexts and developing efficient algorithms, we have provided tools that can enhance memory allocation and resource management across numerous applications. The implications of this work extend beyond theoretical interest, promising tangible improvements in fields such as logistics, computer systems, and operations research.

5.2 Future Work

In this section, we outline potential directions for future research based on our findings. These directions aim to build upon the current study, exploring new dimensions of diversity constraints and advancing algorithmic development in both offline and online paging contexts.

5.2.1 Diversity Extensions

Future work should explore additional notions of diversity beyond the richness constraints discussed in this study. Potential directions include:

- **Shannon Index**: This measure of entropy could be used to quantify and maximize the uncertainty or diversity within the cache, ensuring a more evenly distributed set of pages.
- **Memory Reserves**: Allocating specific portions of the cache to different types of pages could ensure balanced usage and prevent any single type from dominating the cache space.
- **Hill Numbers**: A family of diversity indices that generalize richness, allowing for different emphases on rare or common pages, could provide a more nuanced approach to maintaining diversity.

As for the memory reserves notion, we explored some behaviors as the *Caching with Reserves* [8] paper can be seen as paging with diversity constraints, and found some results such as it also seems to be integer (unlike the Min-Max variant) and the linear program we created by adding the restriction to our formulation seems to represent the problem. A future work is to verify if is there some way to build an optimal algorithm similar to the one for the richness constraints, such as it is optimal and operates under polynomial time.

5.2.2 Algorithms

Future research could focus on refining and improving the marking algorithms used for online paging problems:

- **Adaptive Marking Algorithms**: Developing adaptive algorithms that can dynamically adjust their marking strategies based on real-time analysis of access patterns and cache performance.

- **Enhanced Competitiveness**: Investigating techniques to enhance the competitiveness of marking algorithms, ensuring they perform closer to the optimal under a wider range of scenarios.

Further work is needed to advance deterministic algorithms for both offline and online paging problems:

- **Optimal Bounds**: Research could focus on tightening the lower bounds identified in this study, pushing towards discovering the best possible deterministic algorithms.
- **Scalability**: Ensuring that deterministic algorithms scale efficiently with increasing problem sizes and complexity, particularly in high-demand environments.

To provide a comprehensive analysis of paging algorithms, it is essential to dive into randomized algorithms. Similar to deterministic algorithms, we need to define what constitutes a randomized algorithm.

Definition 5.1 (Randomized algorithm) *A randomized algorithm is an algorithm that incorporates random decisions into its execution process. This means that for a given instance, the output can vary between different runs due to the inherent randomness in the decision-making process.*

Unlike deterministic algorithms, the behavior of a randomized algorithm is not entirely predictable.

For the randomized case, it is not that easy to adapt Fiat's proof [6] to this variant as we could have more and shorter phases.

As we explored, a randomized algorithm selects the page to evict based on a random process. This inherent randomness means that such algorithms do not guarantee the maintenance of diversity during each step of their execution. The lack of a systematic decision-making process can lead to situations where the diversity of the cache is compromised, as the eviction decisions are not explicitly guided by diversity constraints.

Mainly, we can conclude that a badly defined randomized algorithm can lead to breaking the diversity constraints, making it useless for this purpose, so there has to be some kind of restriction referring to the types for it to keep pages that removing would not be removed in a feasible schedule, this makes the phases be shorter, which implies more phases and a complicate handling of them.

The primary challenge in defining randomized algorithms for the diverse paging problem lies in balancing randomness with diversity preservation. Here are some of the specific issues faced:

- **Lack of Predictability**: Since randomized algorithms do not follow a fixed rule, their behavior can be unpredictable. This unpredictability makes it difficult to ensure that diversity is maintained across different runs and instances.
- **Sub-optimal diversity**: Randomized decisions can lead to sub-optimal outcomes where diversity is not preserved. For instance, consecutive random evictions might remove a variety of page types, leading to a homogeneous cache content, which contradicts the goal of maintaining diversity.
- **Performance variability**: The performance of randomized algorithms can vary significantly across different instances. While they might perform well on average, certain instances could result in poor diversity maintenance and higher page fault rates.
- **Complex analysis**: Analyzing the performance and guarantees of randomized algorithms is more complex compared to deterministic ones. Probabilistic methods are required to evaluate their expected behavior, which can be mathematically intensive and less intuitive.

To address these challenges, future work could focus on developing hybrid algorithms that incorporate both deterministic rules and randomized elements. Such hybrid approaches could use randomness for some decisions while applying diversity-preserving heuristics for others. Additionally, techniques such as:

- **Biased Randomization**: Introducing biases in the random selection process to favor evictions that are less likely to reduce diversity.
- **Randomized Rounding**: Using linear programming solutions to guide the random decisions, rounding fractional solutions in a way that maintains diversity.
- **Adaptive Randomization**: Adjusting the degree of randomness based on the current state of the cache and the diversity levels, ensuring that critical diversity thresholds are not violated.

These strategies can help in designing randomized algorithms that better balance the need for diversity with the benefits of randomization.

The development of randomized algorithms remains a challenging yet promising area for future research:

- **Algorithm Design**: Investigating new design strategies for randomized algorithms that balance the trade-offs between performance and unpredictability.

- **Performance Guarantees**: Establishing robust performance guarantees for randomized algorithms, providing clearer expectations for their behavior under various conditions.
- **Hybrid Approaches**: Exploring hybrid algorithms that combine deterministic and randomized strategies to leverage the strengths of both approaches.

Bibliography

[1] Antonios Antoniadis, Joan Boyar, Marek Eliáš, Lene M. Favrholdt, Ruben Hoeksma, Kim S. Larsen, Adam Polak, and Bertrand Simon. Paging with succinct predictions, 2022. URL https://arxiv.org/abs/2210.02775.

[2] Nikhil Bansal, Christian Coester, Ravi Kumar, Manish Purohit, and Erik Vee. Learning-augmented weighted paging, 2020. URL https://arxiv.org/abs/2011.09076.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi: 10.1147/sj.52.0078.

[4] Paul Camion. Characterization of totally unimodular matrices. *Proceedings of the American Mathematical Society*, 16(5):1068–1073, 1965.

[5] Ashish Chiplunkar, Monika Henzinger, Sagar Sudhir Kale, and Maximilian Vötsch. Online min-max paging, 2022. URL https://arxiv.org/abs/2212.03016.

[6] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, dec 1991. doi: 10.1016/0196-6774(91)90041-v. URL https://doi.org/10.1016%2F0196-6774%2891%2990041-v.

[7] Waldo Gálvez and Víctor Verdugo. Approximation schemes for packing problems with p-norm diversity constraints. In *LATIN 2022: Theoretical Informatics: 15th Latin American Symposium, Guanajuato, Mexico, November 7–11, 2022, Proceedings*, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-20623-8. doi: 10.1007/978-3-031-20624-5_13. URL https://doi.org/10.1007/978-3-031-20624-5_13.

[8] Sharat Ibrahimpur, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Caching with reserves, 2022.

[9] Sandy Irani. *Competitive analysis of paging*, pages 52–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-68311-7. doi: 10.1007/BFb0029564. URL https://doi.org/10.1007/BFb0029564.

[10] Tom Leinster. Entropy and diversity: The axiomatic approach. *CoRR*, abs/2012.02113, 2020. URL https://arxiv.org/abs/2012.02113.

[11] Sebastian Perez-Salazar, Alfredo Torrico, and Victor Verdugo. Preserving diversity when partitioning: A geometric approach, 2021. URL https://arxiv.org/abs/2107.04674.

[12] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), feb 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL https://doi.org/10.1145/2786.2793.

[13] Rahul Vaze and Sharayu Moharir. Paging with multiple caches, 2016. URL https://arxiv.org/abs/1602.07195.

[14] Wenming Zhang, Huan Hu, Yongxi Cheng, Hongmei Li, and Haizhen Wang. The work function algorithm for the paging problem. *Theoretical Computer Science*, 930: 100–105, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2022.07.014. URL https://www.sciencedirect.com/science/article/pii/S0304397522004327.