The Pair Cache Problem

by

SayedMohammadAmin Khodaee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

© SayedMohammadAmin Khodaee 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis investigates the pair cache problem, a unique variation of the classic cache replacement problem where each element is stored in two pages, and the cache only needs one of these pages to respond to a query. The thesis formalizes the definition of the pair cache problem, explores relevant historical background, and investigates both offline and online cases. In the offline case, we show that approximating the problem with a factor less than two is NP-hard, while in the online case, we extend the FIFO algorithm and prove that it works as a 4-factor approximation. The proposed pair cache scheme has practical applications in systems where data retrieval times are slow, and most processes require information stored in two different data sources. The goal of a pair cache management algorithm is to minimize the number of retrievals by deciding which page to retrieve on a miss and which page to discard when the cache is full.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor J. Ian Munro, for his unwavering support and guidance throughout the course of my master's thesis. His expertise, patience, and encouragement were instrumental in moving the research forward, especially during the most challenging times. I am incredibly fortunate to have had him as my mentor.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In this thesis, we delve into a unique variation of the classic cache replacement problem that we call the pair cache problem. Unlike the usual cache replacement problem where each element is stored on a single page, in the pair cache problem, each element is stored on two pages, and the cache only needs one of these pages to respond to a query. This requires the cache replacement algorithm to not only make decisions about which page to discard to make room for new pages but also which page to retrieve to respond to a query.

The introduction of this new aspect of the problem increases its complexity. The traditional Belady's algorithm[1] is efficient in solving the offline cache replacement problem, but we will show that the offline pair cache problem is NP-hard, meaning it probably cannot be solved in polynomial time. Moreover, we prove that finding a polynomial algorithm that approximates the offline pair cache problem with a constant factor of less than two is hard, i.e. the 2-factor approximation is NP-hard if the unique games conjecture[4] is true. This conjecture is well known in computational complexity theory. However, we propose the "FPIFO" algorithm("first pair in first out"), an extension of the classic first in first out algorithm, that shows promising results in the online version of the pair cache problem, where page requests are not known in advance. We prove that it offers a 4-factor approximation, retrieving four times the optimal with four times as much cache.

The proposed pair cache scheme has practical applications in systems where data retrieval times are slow and most processes require information stored in two different data sources. Imagine a third-party service interacting with a vast social media network, seeking

to optimize the number of requests it makes to the network for shortest-path trees among users, without compromising on storage limits or user privacy. The third-party service is allowed to query the social media network for a shortest-path tree from a vertex at a fixed cost and is tasked with responding to a sequence of shortest-path queries among pairs of vertices. The service is allowed to only store a limited number of the given shortest-path trees at a time so that it does not learn about the whole network over time. The shortest path between two vertices can be easily found using a shortest-path tree rooted from either of the two ends. The pair cache algorithm can come into play and store the allowed number of shortest-path trees, minimizing the number of shortest-path tree queries while responding to all the shortest path queries. Later in the thesis, we will explore more applications by giving a general example of graph algorithms that can be assisted with a pair cache algorithm when running on a large graph.

The pair cache problem, while having practical implications, is also of theoretical significance. We will show that the vertex cover problem—an established computational problem concerning the covering of graphs— can be reduced to the offline pair cache problem.

The journey of this thesis begins by formally defining the pair cache problem, followed by a historical overview of the cache replacement problem and related issues such as the vertex cover in the section on related works. From there, we delve deeper into the relationship between the pair cache problem and the vertex cover problem. We then examine the offline pair cache problem and establish its computational difficulty. Lastly, we probe into the online version of the pair cache problem, propose the FPIFO algorithm, and illustrate its efficacy. Along the way, we will further illustrate practical applications of the pair cache problem ensuring that our theoretical investigation translates to real-world utility.

1.2 Problem

The cache problem is a common issue faced by operating systems in order to speed up the process of querying elements stored in the main memory. The main memory stores elements on pages of equal size and each element is associated with only one page. To speed up the process, the operating system uses a small and fast cache as an intermediary memory.

When a query is made, if the page associated with the required element is present in the cache, the query is considered a “hit”, and the result is returned almost instantly. However, if the page is not present in the cache, the query is considered a “miss”. In this case, the system retrieves the page from the main memory, which is time-consuming, and

stores it in the cache before responding to the query. Cache replacement policies manage the cache memory by deciding which page to discard when a new page should be retrieved and the cache is full.

The pair cache problem is a variation of the cache problem, where elements are stored on two pages, instead of just one. In order to respond to a query, the cache must have access to at least one of the two pages. If either of the two pages is present in the cache, the query is considered a "hit". Otherwise, it is considered a "miss", and the system must retrieve at least one of the two pages, store it in the cache, and then respond to the query. The goal of a pair cache management algorithm is to minimize the number of retrievals by deciding which page to retrieve on a miss and which page to discard when the cache is full.

For example, consider a scenario where we are storing information on a complete graph. This graph is divided into n pages, with each page representing a node. Each edge in the graph is represented as an element, and there are $n(n-1)/2$ elements in total.

Let us denote a page as p_i and an element as $e_{i,j}$, where $0 < i < j <= n$. Each element $e_{i,j}$ is stored on two pages: p_i and p_j.

Now, let us assume that our cache can store k pages, where k is less than or equal to $n-2$. If the cache currently stores pages $p_1, p_2, ..., p_k$, and we receive three queries for elements $e_{1,k}, e_{k-1,k+1}, e_{k+1,k+2}$, the cache can respond to the first two queries using the pages it currently stores. However, for the third query, the cache will need to retrieve either p_{k+1} or p_{k+2} from the main memory, which means it will need to discard one of the pages it currently stores.

In this scenario, the pair cache management algorithm's goal is to decide which page to retrieve and which page to discard in order to minimize the number of retrievals from the main memory.

It is important to note that the problem is agnostic to the elements themselves and only focuses on the pages that the elements are on. From this point forward, we will represent a query as a combination of the two pages that contain the queried element.

1.3 Notation and Problem Statement

In this section, we will formally define the pair cache problem and introduce the notation that we will use throughout this thesis.

A sequence of queries, denoted as $Q : q_1, q_2, ..., q_n$ is given and an empty cache with a capacity of k is available. The algorithm is tasked with answering the queries one by one.

The query q_i is specified by a pair of pages in the notation $(a_i|b_i)$, i.e., q_i is asking for an element which is in both pages a_i and b_i.

To respond to each query, the algorithm can perform the following operations:

- Retrieve(p): This operation retrieves page p from the main memory and stores it in the cache. This operation is only possible if there is sufficient unused space in the cache. For example, if the cache is full and a query for a page not in the cache is made, the algorithm would need to perform a Discard operation before it could Retrieve the new page.
- Discard(p): This operation removes page p from the cache to create space for a subsequent Retrieve operation.
- Respond(p): This operation answers the query using page p ($p = a_i$ or $p = b_i$), provided that page p is present in the cache.

The goal of the algorithm is to answer all queries while minimizing the frequency of the "Retrieve" operation.

There are two variants of the pair cache problem:

- Offline: In this variant, the sequence of queries is known in advance. The algorithm must manage the cache to respond to the queries one by one. This variant might be used in scenarios where the queries can be predicted, such as in a batch processing system.
- Online: In this variant, the queries are presented one by one after the algorithm has responded to the preceding query. This variant might be used in scenarios where the queries cannot be predicted, such as in a real-time system.

Understanding both variants is crucial. The offline version offers insights into the theoretical aspects of the problem, its computational complexity, and the inherent challenges in optimization. The online version, on the other hand, is more directly applicable to real-world scenarios, and understanding its behaviour is key to devising efficient, practical cache management strategies.

We also categorize the efficiency of an algorithm into three levels:

- Optimal: An optimal algorithm minimizes the number of retrievals. This is the ideal level of efficiency, but it may not always be achievable in practice.
- t-factor approximation: A t-factor approximation algorithm uses a cache that is t times larger and incurs t times the number of retrievals compared to an optimal algorithm. This level of efficiency might be acceptable in scenarios where a larger cache is available.
- α, β-factor approximation: An α, β-factor approximation algorithm uses a cache that is α times larger and incurs a maximum of β times more retrievals compared to an optimal algorithm. This level of efficiency might be acceptable in scenarios where a balance between cache size and retrieval frequency is needed.

The idea of an approximation to the optimal is natural, where approximation means just more cache misses. At first blush, giving a larger cache for the online approximation seems strange, until an old observation is recalled. Suppose we have a cache of size $n-1$, then an online method could remove the element required next at each step, whereas an offline method would require at most one miss for each n steps. For this reason Sleator and Tarjan[9] allowed the online approximation method to have twice the cache than the offline optimal. We consider even more "cache flexibility" for our online approximation, making comparisons between the two methods more fair.

1.4 Theorems

This section outlines the primary theorems established in this thesis. In Section 3.1, we demonstrate the similarities of the pair cache problem and the vertex cover problem. The first theorem shows that the pair cache problem can be viewed as a generalization to the vertex cover problem.

Theorem 1. *Let $G(E, V)$ be a graph with edges $e_i = (v_i, u_i)$. The graph has a vertex cover of size k if and only if an offline pair cache of size k can respond to the sequence of queries $Q : (v_1|u_1), (v_2|u_2), ..., (v_{|E|}|u_{|E|})$ with at most k retrievals.*

In Section 3.2, we prove that the offline pair cache problem's optimal solution is NP-Hard. We further extend this result by demonstrating that finding a polynomial 2-factor approximation for the problem is also a hard task.

Theorem 2. *The task of determining an optimal solution for the offline pair cache problem is NP-Hard.*

Theorem 3. *If the unique games conjecture holds true, then approximating the offline pair cache problem with a constant factor less than 2 is NP-Hard.*

Section 3.3 focuses on the online variant of the pair cache problem. We begin by extending the FIFO algorithm to the pair cache problem by retrieving the pair of pages when it misses. Furthermore we illustrate that FPIFO(first pair in first out) provides a 4-factor approximation and construct worst case sequences in which it does not work significantly better than that.

Theorem 4. *The FPIFO algorithm provides a 4-factor approximation for the online pair cache problem.*

Theorem 5. *In the worst case, if the cache size of the FPIFO algorithm is* 4 *times larger, it uses* $\beta = 4 - o(1)$ *times retrievals compared to an optimal cache baseline.*

In Section 3.4, we extend the results from Section 3.3 to the α, β-factor approximations.

Theorem 6. *For constants* $\alpha, \beta > 2$*, if the cache size of the FPIFO algorithm is* α *times larger and uses* β *times retrievals compared to an optimal cache baseline, then* $\beta \leq \frac{2\alpha}{\alpha-2}$.

Theorem 7. *Under worst-case conditions, if FPIFO's cache size is* α *times larger, provided that* $\alpha > 2$ *and the condition* $(\alpha * k) \mod 2 = 0$ *is satisfied, the number of retrievals used by FPIFO in comparison to the optimal cache baseline is:* $\beta = \frac{2\alpha}{\alpha-2} - o(1)$

Chapter 2

Related Work

This thesis undertakes the task of studying the pair cache problem, a unique variant of the cache replacement problem, where each item resides on two pages, and the cache only needs to retrieve one of these pages to satisfy a query. We propose an extension of the first in first out (FIFO) algorithm and evaluate its effectiveness for this problem. Furthermore, we investigate the complexity of finding an optimal algorithm, referred to as a MIN algorithm, for this problem. Our investigation builds upon previous research on the classical cache replacement problem and its variants. In proving the complexity, we utilize principles from the Vertex Cover problem and Unique Games Conjecture.

2.1 Cache Replacement Problem and Algorithms

We begin by reviewing the classical cache replacement problem. This problem encompasses maintaining a cache of fixed size and determining which item in the cache to replace when a new item arrives, and the cache is at capacity. The aim is to minimize cache misses—instances where a requested item is not in the cache.

Various cache replacement algorithms have been proposed in the literature, including Least Recently Used (LRU), First-In First-Out (FIFO), Least Frequently Used (LFU), and Random Replacement (RR), each offering different performance characteristics and suitability to various types of workloads.

A classic algorithm for the offline cache replacement problem is Belady's MIN algorithm, proposed by Belady in 1966[1]. This algorithm replaces the item with the furthest future use—the item whose next reference has the largest distance. Belady's algorithm offers

optimal performance for a given cache size and workload. It is also simple to comprehend and implement; however, its main limitation is the requirement of knowledge of the entire sequence of requests in advance, a condition often unfeasible in practice.

In scenarios where a query specifies when the next page request will occur, Belady's can be efficiently implemented using a balanced tree. This allows for identifying a page on a hit (by accessing the leftmost item) or replacing a page (by removing the rightmost item), then adding a new page on a miss—all in logarithmic time. However, if the focus is solely on the misses, a more streamlined data structure like a self-organizing priority queue can be utilized to process the miss in logarithmic time. This approach mirrors the techniques discussed in [11] and [10].

LRU, a widely used online cache replacement algorithm, operates by keeping track of the least recently used item in the cache and replacing it with the new item when the cache is full. LRU has been proven to have good worst-case performance, with cache misses being at most k times the optimal, where k is the cache size [12].

FIFO is another online cache replacement algorithm. It operates with a queue, replacing the oldest item with the new item when the cache is full.

FIFO and LRU both miss at most $n/(n - n_{MIN} + 1)$ times the optimal if n and n_{MIN} are the size of the cache in the LRU/FIFO and MIN algorithm respectively[12].

We can represent both these algorithms as an α, β-factor approximation (α: Cache size ratio, and β: Miss ratio), using the following equations:

$$\alpha = n/n_{MIN}$$

$$\beta = n/(n - n_{MIN} + 1) = \alpha * n_{MIN}/(\alpha * n_{MIN} - n_{MIN} + 1) < \alpha * n_{MIN}/((\alpha - 1) * n_{MIN} + 1)$$

Here, if $\beta >= \alpha/(\alpha - 1)$, FIFO and LRU serve as α, β-factor approximation algorithms. Assuming $\alpha = 2$, They offer 2-approximations.

Having reviewed the classical cache replacement problem and its algorithms, we now move to discuss a variant of this problem, the pair cache problem. The unique feature of this problem is that each item is stored on two pages, and the cache only needs to retrieve one of these pages to respond to a query. This thesis aims to investigate the complexity of finding an optimal offline algorithm (MIN) for the pair cache problem and proposes extensions of FIFO and LRU algorithms for the online version of the pair cache problem.

2.2 Vertex Cover Problem and Unique Games Conjecture

Our journey to understand the complexity of the pair cache problem brings us to the Vertex Cover problem—a classical optimization problem in computer science, and the Unique Games Conjecture (UGC).

The Vertex Cover problem is a classical optimization problem in computer science that has been extensively studied. It involves finding a minimum-size subset of vertices $S \subseteq V$ from a given undirected graph $G = (V, E)$ such that every edge in E is incident to at least one vertex in S. This problem is NP-complete, suggesting it is computationally difficult to find an exact solution within polynomial time[3], but approximations within a factor of the optimal solution are achievable. The question of how well we can approximate the Vertex Cover problem has been the subject of much research.

The Unique Games Conjecture (UGC), on the other hand, is a central hypothesis in computational complexity theory that relates to the Vertex Cover problem. The conjecture states that it is NP-hard to distinguish between instances of a certain type of constraint satisfaction problem (CSP) that has some assignments that satisfy most constraints and instances in which at most a small fraction of the constraints can be satisfied. We describe UCG as Khot[4] formulated it in the unique label cover context.

Given a bipartite graph $G = (V, E)$, each vertex $v \in V$ is associated with a label set L_v from a finite domain $[k]$ (where k is a positive integer). Each edge $(u, v) \in E$ has an associated permutation $\pi_{u,v} : L_u \to L_v$. A labeling of the graph assigns a label from L_v to each vertex v.

An assignment of labels to the vertices satisfies an edge (u, v) if the label assigned to vertex u maps to the label assigned to vertex v under the permutation $\pi_{u,v}$.

The conjecture posits that for a sufficiently small constant $\epsilon > 0$, it is **NP**-hard to distinguish between the following two cases:

1. There exists a labeling that satisfies $1 - \epsilon$ fraction of the edges (i.e., almost all of them).

2. Every labeling satisfies at most an ϵ fraction of the edges (i.e., almost none of them).

UCG has direct implications for the approximability of various optimization problems, including the Vertex Cover problem. Significant findings in this area include Khot's work,

showing that assuming UGC to be true, approximating the Vertex Cover problem within a constant factor less than $\sqrt{2}$ is NP-hard[4]. Khot and Regev further strengthened this result, demonstrating that approximating the Vertex Cover problem within any factor less than 2 is NP-hard, under the assumption of UGC's truthfulness[6].

Since its introduction, UGC has been extensively studied in the literature, and many works have explored its consequences for various optimization problems. Khot and Regev showed that UGC implies the hardness of approximating fundamental problems like Max Cut, Max 2-Sat and MAX-2LIN[5].

Overall, UGC is an important hypothesis that has led to significant advances in the theory of approximation algorithms. Its implications for the Vertex Cover problem demonstrate the difficulty of the problem and the challenges involved in designing efficient approximation algorithms for it.

In the next chapter, we aim to show the reducibility of the Vertex Cover problem to the offline pair cache problem and examine the implications of the Unique Games Conjecture on approximating the pair cache problem.

2.3 k-server Problem and the CNN Problem

Many cache problem variants have been explored in academic research over the past decades. One such variant is the k-server problem, a generalization of the cache problem and a fundamental issue in the field of online algorithms and competitive analysis. This problem focuses on efficiently moving k servers to service requests that appear online at the points of a metric space, with the objective to minimize the total distance travelled by the servers. The problem was first defined by Manasse, McGeogh, and Sleator, and it has been a major driving force for the development of the area of online algorithms [9].

The k-server problem is defined by an initial configuration and a sequence of requests. A solution is a sequence of configurations such that each request is included in the corresponding configuration. The cost of a solution is the total distance travelled by the servers. An online algorithm computes each configuration based only on the past, and the central question in competitive analysis is how good an online algorithm is compared to an optimal algorithm which is based also on future requests[7]. A cache problem with a cache of size k can be viewed as a k-server problem in a uniform metric space.

The k-server problem and its variants, including the CNN problem, are crucial in shaping our understanding of the pair cache problem. In the CNN problem(named after a

scenario in which the Cable News Network crew are trying to shoot scenes on Manhattan), a server services requests in the Euclidean plane but only needs to move to a point that lies in the same horizontal or vertical line with the request. This problem is applicable in situations where the server's exact location is not as crucial as being in the request's general vicinity[8]. The pair cache problem can be viewed as a CNN problem in uniform distances, albeit the CNN problem is primarily evaluated with a low number of servers (up to two).

These cache problem variants have considerably aided academic progress in online problems, finding practical applications in database systems and disk management. This research suggests that the pair cache problem, given its unique characteristics, can contribute to this progress as well.

In the next chapter, we will delve deeper into the pair cache problem, exploring its computational complexity and proposing an extension of the FIFO algorithm that performs well for this problem.

Chapter 3

Methods

3.1 Pair Cache and Vertex Cover

In this section, we delve into the relationship between the pair cache scheme and the vertex cover problem. Understanding the interplay between these two provides a unique perspective into the theoretical essence of this new variant of the cache problem and its computational complexity.

The discussion begins with the offline version of the pair cache problem. We establish that the well-known minimum vertex cover problem can be reformulated as the pair cache problem. This is stated formally in the following theorem:

Theorem 1. *Let $G(E, V)$ be a graph with edges $e_i = (v_i, u_i)$. The graph has a vertex cover of size k if and only if an offline pair cache of size k can respond to the sequence of queries $Q : (v_1|u_1), (v_2|u_2), ..., (v_{|E|}|u_{|E|})$ with at most k retrievals.*

Proof. The two problems have a deep-seated similarity. Each edge in the graph can be interpreted as an element shared by two pages, i.e., the two vertices at its ends. Suppose an offline pair cache of size k can respond to queries Q, by retrieving at most $k' \leq k$ pages $P = p_1, p_2, ..., p_{k'}$. Each element queried in Q represents an edge in graph G, which is responded to by a page in P corresponding to one of the edge's endpoints. This implies every edge in G has one of its endpoints in P, which is the definition of a vertex cover for G.

Conversely, let $C = c_1, c_2, ..., c_k$ be a vertex cover for G. Every query in Q encompasses the endpoints of an edge in G, and at least one endpoint will be found in C. The cache

of size k can retrieve all items in C to respond to all queries without needing to remove any element in the process. Thus, the cache can respond to all queries with at most k retrievals. □

This theorem concludes that the offline pair cache problem is as complex as the minimum vertex cover problem. It offers significant insights into the computational complexity of the pair cache problem, and it paves the way for further exploration into this topic in the following section.

Next, we introduce an online variant to the vertex cover problem that we call the "edge serving problem". This problem finds its relevance to the online pair cache problem. Unlike the static set of vertices chosen in the vertex cover problem to cover a given set of edges, the edge serving problem revolves around maintaining a dynamic set of vertices of limited size, aiming to cover requested edges one at a time.

Here is an illustration of the edge serving problem: consider a scenario where a city is under air-bombing, with one road being targeted each day. As a defence general, you have k anti-bomb squads at your disposal. A road $r = \{v, u\}$ can be protected if a squad is deployed at intersection v or u. A spy reports the targeted road to you at midnight every day. As squad movements entail high costs, the goal is to minimize the number of times you relocate a squad while ensuring the city's protection.

To put it formally, in the edge serving problem, we have a sequence of marked edges Q : $e_{v_1,u_1}, e_{v_2,u_2}, ..., e_{v_t,u_t}$ revealed one by one, and an initially empty vertex set S of maximum size k. At time i, the edge e_{v_i,u_i} is marked. The algorithm should ensure the edge e_{v_i,u_i} at time i is covered by including either v_i or u_i in the vertex set S. The challenge is to minimize the number of times a vertex is added to the set.

The edge serving problem can be seen as equivalent to the pair cache problem where vertices are pages, edges are elements, and the goal is to maintain a cache of size k (similar to the vertex set) that serves a sequence of pair queries (like edges) while minimizing the number of retrievals (akin to adding a vertex to the set).

This variation of the vertex cover problem sheds light on the practical applications of pair cache problems. For instance, in maintaining a memory of size $k << n$ that can store the adjacency lists of k vertices in a large complete graph of n vertices. To cater to an algorithm's needs that request one edge data s_i at each step i, an online pair cache algorithm can be employed to maintain a set of k adjacency lists at minimum cost.

In conclusion, we have successfully bridged the gap between the offline pair cache problem and the vertex cover problem. We introduced the edge serving problem as an online

version of the vertex cover problem, discussed its real-world implications, and established its equivalence to the online pair cache problem. In the subsequent sections, sections 3.2 and 3.3, we will delve into the computational complexity of the offline and online pair cache problems.

3.2 Offline Pair Cache Problem

For the offline case of the standard cache problem, Belady's algorithm provides an optimal solution. The strategy is straightforward: discard the page that will not be required for the longest period in the future. Despite its simplicity, it works effectively. If provided with the subsequent usage time of the queried page (as elaborated in Section 2.1), it can execute each query in logarithmic time. However, when it comes to the pair cache problem, things become considerably more complicated.

Extending Belady's algorithm to the pair cache problem involves answering a complex question: when will each page be needed again, considering whether it will truly be necessary the next time it is requested? This predicament leads to a branching problem: which page should be used to respond to each query?

By considering all possible choices, we realize that the pair cache problem can be reduced to an exponential number of standard cache problems by determining the page to use for each query.

Theorem 2 further supports our argument by demonstrating that the pair cache problem is NP-Hard. The result is interesting as it suggests our initial intuition about the complexity of the problem was not an overestimation.

Theorem 2. *The task of determining an optimal solution for the offline pair cache problem is NP-Hard.*

Proof. Building upon Theorem 1, any minimum vertex cover problem can be transformed into an offline optimal pair cache problem. Given that the minimum vertex cover problem is a known NP-Hard problem, the same complexity carries over to the offline optimal pair cache problem. □

In the preceding Theorem, we substantiated that the offline pair cache problem poses a greater challenge than the regular offline cache problem. Proceeding forward, we aim to broaden this comparison in the ensuing theorem by demonstrating that the offline pair cache problem, even armed with complete foreknowledge of future queries, does not find

itself more straightforward than the online cache problem, which is resolvable utilizing LRU with a 2-factor approximation. The complexity of the offline pair cache problem, as affirmed by theorems 1 and 2, is analogous to that of the Vertex Cover problem. Therefore, proving this theorem should not present an inordinate challenge.

Theorem 3. *If the unique games conjecture holds true, then approximating the offline pair cache problem with a constant factor less than 2 is NP-Hard.*

Proof. As demonstrated by Theorem 1, a minimum vertex cover problem can be reduced to an offline pair cache problem. Hence, any algorithm that can solve the offline pair cache problem with $c * MIN$ retrievals can equivalently solve the minimum vertex cover problem with $c * MIN$ vertices. It is known that approximating the minimum vertex cover with a factor less than 2 is NP-Hard, assuming the unique games conjecture holds true [6]. The same complexity, therefore, applies to the offline pair cache problem. □

To conclude, the aforementioned theorems establish the intricacy of the offline pair cache problem. Compared to the standard cache problem, it presents more sophisticated challenges. Moreover, its complexity is akin to well-recognized NP-Hard problems such as the Vertex Cover problem. Hence, our initial suspicion about the problem's complexity was well-founded. Even with complete knowledge of future queries, the offline pair cache problem remains as complex as the online version of the standard cache problem solvable with the LRU's 2-factor approximation.

3.3 Online Pair Cache Problem

In the previous section, we explored the offline pair cache problem. Now, let us turn our attention to the online pair cache problem, where future query information is unknown. To establish a comparison, remember that in the standard cache problem, both FIFO and LRU algorithms solve the online case with a 2-factor approximation, which is twice the optimal solution achieved in the offline case (as discussed in section 2.1). As we delve into the online pair cache problem, a natural question to ask is if a 4-factor approximation is possible, which would be twice as bad as the lower bound we proved for the offline case.

An α, β-approximation cache replacement algorithm, given α times more cache, can use up to β times more retrievals than the optimal algorithm. We will discuss the bounds considering unequal α and β in section 3.4. For now, we focus on a version of the problem where $\alpha = \beta = 4$ and aim to extend a classic cache replacement algorithm for achieving a 4-factor approximation.

FIFO and LRU are the most natural algorithms to consider extending since they perform well in the standard cache replacement problem. However, adapting these algorithms to the pair cache context is not straightforward.

The FIFO algorithm, when it misses, removes the page that has been retrieved earliest before retrieving the required page. However, in a pair cache scenario, it becomes complicated to decide which page to retrieve when a query is missed. To navigate this challenge, we propose the first pair in first out (FPIFO) algorithm, a simple method where we consider pages as pairs that are retrieved and removed together. Specifically, for a queried element, if none of its pages is present in the cache, we retrieve both pages, and to free up space, we remove a pair of pages that were retrieved together.

Algorithm 3.3 outlines the operation of the FPIFO algorithm when a query $q : (a|b)$ arrives, where the query q is present on pages a and b.

Algorithm 1 First pair in first out (FPIFO) Cache Algorithm

```
1: Input: A query q with pages (a, b), cache as a queue.
2: Output: cache after processing the query.
3: function FPIFO_Handle_Query(q)
4:     if a ∉ cache and b ∉ cache then
5:         if cache is full then
6:             cache.pop_front()
7:             cache.pop_front()              ▷ Remove the oldest pair of pages
8:         end if
9:         cache.push_back(a)
10:        cache.push_back(b)                 ▷ Add the new pair of pages
11:    end if
12:    return cache                           ▷ The output cache includes either a or b
13: end function
```

Similarly, we can extend the LRU algorithm to the pair cache context using the same technique. The standard LRU removes the page that has not been used for the longest time. In a pair cache scenario, we can consider pages as pairs that are retrieved and removed together. This means that for a queried element, if none of its pages is present in the cache, we fetch both pages. To free up space, we remove a pair of pages that were retrieved together. We maintain a "used time" for each pair, which records the last time any of the two pages in a pair were used.

Algorithm 2 outlines the operation of the least recently used pair (LRUP) algorithm when a query $q : (a|b)$ arrives, where the query q is present on pages a and b.

Algorithm 2 Least Recently Used Pair (LRUP) Cache Algorithm

Input: A query q with pages (a, b).
function LRUP_HANDLE_QUERY(q)
    if $a_i \notin$ cache and $b_i \notin$ cache **then**
        RETRIEVE((a_i, b_i))
        $(a_i, b_i)_{\text{usedtime}} \leftarrow i$
        if cache is full **then**
            $(a_{\min}, b_{\min}) \leftarrow \arg\min_{(a,b)\in\text{cache}}(a, b)_{\text{usedtime}}$
            REMOVE($(a_{\min}, b_{\min})$)
        end if
    else if $a_i = a_j$ and $b_i = b_t$ and $(a_j, b_j) \in$ cache and $(a_t, b_t) \in$ cache **then**
        Choose $x \in \{j, t\}$ arbitrarily
        $(a_x, b_x)_{\text{usedtime}} \leftarrow i$
    else if ($a_i = a_j$ or $b_i = a_j$) and $(a_j, b_j) \in$ cache **then**
        $(a_j, b_j)_{\text{usedtime}} \leftarrow i$
    end if
end function

While it seems more natural to prove a good approximation with the LRUP algorithm, we found it challenging to prove that it can achieve a 4-factor approximation. However, the FPIFO algorithm is simpler to analyze and can achieve our 4-factor approximation goal more easily.

Theorem 4. *The FPIFO algorithm provides a 4-factor approximation for the online pair cache problem.*

Proof. Our aim is to prove that on any query sequence $Q, |Q| \geq 1$, FPIFO with a cache of size $4k$, retrieves at most $4k/(k+1) < 4$ times the number of elements retrieved by an optimal algorithm (which we call the MIN algorithm), which has a cache of size k.

Starting from the end, we break the query sequence into subsequences $S_0, ..., S_m$ such that MIN has $k+1$ misses in each, and S_m is the subsequence that includes the last query. If MIN had exactly $m * (k+1)$ misses, S_0 would be empty. Note that while breaking the sequence, every subsequence begins with a miss by MIN.

In S_0, both caches are empty at the start. Thus, if FPIFO misses a query, MIN would miss it as well. So if MIN retrieves $0 \leq l \leq k$ elements, then FPIFO would retrieve at most $2l$ elements. Observe that $2l/l = 2 \leq 4k/(k+1)$.

In $S_t(t > 0)$, MIN misses (and retrieves) $k + 1$ times. We define $GoldenSet_t$ as the set of elements that MIN uses to respond to the queries in S_t. By demonstrating Lemmas 1-5, we prove that FPIFO retrieves at most $2k$ times while responding to S_t, each time a pair that includes a unique member of $GoldenSet_t$. Thus, it retrieves at most $4k/(k+1)$ times the number of elements retrieved by MIN while responding to S_t.

Adding all the retrievals in $S_0, ..., S_m$, FPIFO retrieves at most $4k/(k+1)$ times the number of elements retrieved by MIN.

Lemmas 1-5 address critical characteristics of the $GoldenSet_t$ and the retrieval behaviour of FPIFO during $S_t(t > 0)$. Lemma 1 bounds the size of $GoldenSet_t$, Lemma 2 explains the retrieval policy of FPIFO, Lemma 3 establishes that once a pair is retrieved, it is not removed during the same sequence, Lemma 4 confirms that each element in $GoldenSet_t$ is retrieved at most once, and Lemma 5 caps the number of retrievals FPIFO makes during any sequence S_t.

Lemma 1. *The size of the set $GoldenSet_t$ is less than or equal to $2k$.*

Proof. According to our definition, MIN, having k spaces in the cache, misses the first query in S_t and retrieves a new element. After that query, it can contain at most k elements in the cache, and it needs to retrieve up to k additional elements during S_t. These $2k$ elements are the only ones that MIN can use to respond to the queries in S_t. Consequently, these are the only elements that can constitute the $GoldenSet_t$. □

Lemma 2. *During S_t, FPIFO retrieves a pair (a_i, b_i) only if either $a_i \in GoldenSet_t$ or $b_i \in GoldenSet_t$, and neither of them is already in the cache.*

Proof. FPIFO algorithm follows a policy where it does not retrieve a pair if it can answer the query using an element already in the cache. Suppose FPIFO retrieves a pair (a_i, b_i). This pair is included in the queries in S_t and to answer this query, MIN has to use at least one of the two elements. Hence, as per the definition of $GoldenSet_t$, it should include that element. □

Lemma 3. *Once FPIFO retrieves a pair (a_i, b_i) during S_t, it does not remove it from the cache during S_t.*

Proof. Assume $p = (a_i, b_i)$ is the first pair that FPIFO both retrieves and removes during S_t. This pair must have been retrieved earlier than all other $2k - 1$ pairs that are in the cache (consider that the cache size is $4k$). Hence all other pairs should have been retrieved during S_t as well. Based on Lemma 2, any of these pairs (including p) must include a

unique element in $GoldenSet_t$(if not unique, then the pair retrieved later should have not been retrieved in the first place). Therefore, all members of $GoldenSet_t$ must be in the cache, which means there cannot be a query that FPIFO cannot answer using elements already in the cache. Therefore, there is no need for FPIFO to remove any pair to make space for new retrievals. □

Lemma 4. *FPIFO retrieves each element $a_i \in GoldenSet_t$, at most once during S_t.*

Proof. Based on Lemma 3, the algorithm does not remove a retrieved pair during S_t. Thus, when a_i is retrieved in a pair, it will remain in the cache during S_t. Based on Lemma 2, the algorithm does not retrieve it again if it is in the cache. □

Lemma 5. *FPIFO retrieves at most $4k$ elements during S_t.*

Proof. Based on Lemma 2, the algorithm only retrieves pairs that include an element in $GoldenSet_t$. Based on Lemma 4, it does not retrieve a member of $GoldenSet_t$ more than once. Thus it cannot retrieve more than $2|GoldenSet_t|$ elements. Based on Lemma 1, it can be at most $4k$ elements. □

To summarize, this proof demonstrates that, regardless of the query sequence, the FPIFO algorithm retrieves at most $4k/(k+1)$ times the number of elements retrieved by the optimal MIN algorithm. □

Theorem 5 proposes that in the worst-case scenario, the first pair in first out (FPIFO) algorithm uses $\beta = 4$ times more retrievals compared to an optimal cache baseline if its cache size is 4 times larger. The FPIFO algorithm can thus be deemed as a 4-factor approximator for the pair cache problem. We conjecture that no other algorithm can do significantly better than FPIFO. We formulate this later in conjecture 2. In the next theorem, we show that FPIFO does not challenge our conjecture.

Theorem 5. *In the worst case, if the cache size of the FPIFO algorithm is 4 times larger, it uses $\beta = 4 - o(1)$ times retrievals compared to an optimal cache baseline.*

Proof. Let us consider any constant $\beta = 4 - \epsilon, (\epsilon > 0)$, any $c \in \mathbb{N}$, and any $k > \frac{(4c+8)}{\epsilon \cdot c}$. We aim to construct a sequence of queries Q of length $l = c * 2k^2 + 2k + 1$, such that FPIFO with a cache of size $4k$ uses more than β times retrievals compared to MIN, the optimal cache baseline with a cache of size k.

To achieve this, we construct a sequence where FPIFO misses all $c*2k^2+2k+1$ queries and MIN misses almost half, $ck^2+(c+2)k+1$ times. By design, FPIFO retrieves twice the number of misses.

The constructed sequence uses $2k+1$ elements located in $4k+2$ different pages, queried in rotation. For the sequence $Q:(a_1|b_1),(a_2|b_2),...,(a_l,b_l)$, we define $a_i=((i-1) \mod (2k+1))+1$ and $b_i=-a_i$.

FPIFO misses all queries since no two pages are used in two queries with a distance of less than $2k+1$. ($2k$ misses in between, fills the $4k$ sized cache with other pages). On the other hand, MIN faces a simpler problem: $2k+1$ elements are queried that do not share a page, thus it does not matter which page it brings in on a miss. This effectively reduces the problem to a normal cache replacement problem, and MIN can apply Belady's algorithm on the a_i sequence to achieve the minimum number of retrievals. Using Belady's, MIN can effectively use the k sized cache to hit about k queries in each $2k+1$ queries, i.e., approximately half the number of queries.

Delving into the details of how MIN works, it is not straightforward to hit exactly k queries in each $2k+1$ queries. This is because it might need to remove one of the k values it has cached from the past $2k+1$ queries before using it to hit a query, in order to create room for a retrieval required to respond to a missed query. This observation necessitates a detailed analysis of MIN's behaviour to accurately count the misses.

We analyze how MIN manages the cache to respond to the queries using Table 3.1. We process MIN in $ck+1$ query batches (numbered from 0 to ck) in order. Batch $b>0$, operates as shown in row $((b-1) \mod k)+1$ (row 0 for batch 0).

In row r, we have simulated Belady's algorithm, responding to the queries of any batch b (where $r=((b-1) \mod k)+1$, given the cache final state after processing the previous batch, represented on row $r-1$. (The cache is empty when responding to queries of batch 0). While following Belady's, we observe that the sequence will be off by k compared to the table, when the algorithm wants to respond to the queries from row 1 right after the queries of row k. (last page queried in row k is $k+1$ and we should expect a query for page $k+2$, however page 1 is the first page queries in in row 1). This happens when $b \neq 1$ and ($b \mod k)=1$. To continue simulating the algorithm as the table has, we need to ensure the cache final state is as mentioned in row 0 after processing row k. Thus, after simulating batch $b-1$ ($b \neq 1$ and $b \mod k=1$), we shift all the numbers in the a_i sequence and in the cache by $+k$. Then, the cache state would be like row 0, and the queries in the next batch would be like row 1.

Now that we have described the construction of the sequence, we can count the number of misses and retrievals in FPIFO and MIN.

Row	a_i sequence	Query count	MIN: miss count	MIN: cache final state
0	1, 2, ..., 2k+1	2k+1	2k+1	1, 2, ..., k-1, 2k+1
1	1, 2, ..., 2k+1	2k+1	k+1	1, 2, ..., k-2, 2k, 2k+1
...	...	...	...	...
k-1	1, 2, ..., 2k+1	2k+1	k+1	k+2, k+3, ..., 2k+1
k	1, 2, ..., k+1	k+1	k+1	k+2, k+3, ..., 2k, k+1
Shift	+k	-	-	1, 2, ..., k-1, 2k+1

Table 3.1: The table shows the a_i sequence used in each batch, the number of times the MIN misses in the batch, and what would be the MIN's cache at the end of processing the batch. The queries start from row 0 and go towards row k. They return and start again from row 1 for $(c-1)$ times. To simplify simulation we shift all the numbers queried or stored in the cache before each returning to row 1.

FPIFO cannot hit any query as no page is used twice in a distance of less than $2k+1$, so it retrieves $2 * l = 2 * (c * 2k^2 + 2k + 1)$ times.

As Table 3.1 shows at row 0, MIN misses the first $2k+1$ queries that are in batch 0, and thus retrieves $2k+1$ times.

Batches $b > 0$, go through rows $1, 2, ..., k$ for c times. MIN misses $ck(k+1)$ queries and retrieves $ck(k+1)$ times.

The following math shows us that FPIFO uses more than $4-\epsilon$ times retrievals compared to MIN in the constructed sequence.

$$FPIFO_{\text{retrievals}} = c \cdot 4k^2 + 4k + 2 > 4k(ck+1) = 4k(ck+c+3) - (4kc+8k)$$

$$MIN_{\text{retrievals}} = ck(k+1) + (2k+1) \leq k(ck+c+3)$$

$$\frac{FPIFO_{\text{retrievals}}}{MIN_{\text{retrievals}}} > \frac{4k(ck+c+3) - (4kc+8k)}{k(ck+c+3)} = 4 - \frac{4c+8}{ck+c+3}$$

$$k > \frac{(4c+8)}{\epsilon \cdot c} \rightarrow (4c+8) < \epsilon \cdot ck < \epsilon \cdot (ck+c+3) \rightarrow \frac{4c+8}{ck+c+3} < \epsilon$$

$$\frac{FPIFO_{\text{retrievals}}}{MIN_{\text{retrievals}}} > 4 - \epsilon \rightarrow \frac{LRU(4k)}{MIN(k)} = 4 - o(1)$$

As Theorem 4 shows that FPIFO uses at most 4 times the retrievals, we conclude that the constructed sequence is a worst case example and $4 - o(1)$ is a tight worst case bound. □

Now that we have established that the FPIFO algorithm cannot achieve an approximation factor smaller than 4, a natural question arises: Can the LRUP algorithm achieve a better approximation and thereby challenge Conjecture 2? Unfortunately, it can be easily deduced that the worst-case sequence constructed in Theorem 5 is also applicable to the LRUP algorithm, with exactly the same proof and bound. Thus, Conjecture 2 remains unchallenged.

3.4 General online problem

Building on the success of FPIFO in providing a 4-factor approximation for the cache problem, we delve into its generalization. Consider a scenario where the algorithm employs a cache that is α times larger and performs β times more retrievals. We call this an α, β-approximation. To put this into perspective, let us compare it with the conventional FIFO and LRU. They both works as a α, β-approximation if $\beta < \alpha/(\alpha - 1)$ in a standard cache problem.

Theorem 6. *For constants $\alpha, \beta > 2$, if the cache size of the FPIFO algorithm is α times larger and uses β times retrievals compared to an optimal cache baseline, then $\beta \leq \frac{2\alpha}{\alpha-2}$.*

Proof. We prove that on any long enough sequence of queries, FPIFO with a cache of size $\alpha * k$, retrieves at most $\beta \leq 2\alpha/(\alpha - 2)$ times the optimal algorithm with a cache of size k. The proof follows the same scheme as Theorem 4. we break the sequence into subsequences $S_0, ..., S_m$ where MIN misses $(\alpha-2)k/2+1$ times in each. For any S_t, there is a $GoldenSet_t$ of size at most $\alpha * k/2$ that MIN uses to respond to the queries. Similar to the lemmas proved in Theorem 4, FPIFO retrieves at most $\alpha * k/2$ pairs, each including a member of $GoldenSet_t$, and responds to all the queries in S_t with those, resulting in $\alpha * k$ retrievals (note that all fit in the cache). In any S_t, MIN retrieves $(\alpha - 2)k/2 + 1$ times and FPIFO retrieves $\alpha * k$ times. Thus:

$$\beta = \alpha * k/((\alpha - 2)k/2 + 1) <= 2\alpha/(\alpha - 2)$$

□

Following this, we delineate a boundary for the worst-case scenario in the context of α, β approximations, building upon Theorem 5. It is important to note that the same proof applies to LRUP as well.

Theorem 7. *Under worst-case conditions, if FPIFO's cache size is α times larger, provided that $\alpha > 2$ and the condition $(\alpha * k) \mod 2 = 0$ is satisfied, the number of retrievals used by FPIFO in comparison to the optimal cache baseline is: $\beta = \frac{2\alpha}{\alpha-2} - o(1)$*

Proof. The proof is similar to the one for Theorem 5. For any constant $\beta = \frac{2\alpha}{\alpha-2} - \epsilon, (\epsilon > 0)$, and for any $c \in \mathbb{N}$, we create a sequence of length $l = \frac{\alpha}{2} \cdot ck^2 + \frac{\alpha}{2} \cdot k + 1$ for any $k > \frac{(4c+8)}{\epsilon \cdot c}$), that an FPIFO with a cache of size $\alpha \cdot k$ uses more than β times retrievals compared to MIN, the optimal cache baseline with a cache of size k.

We construct the sequence $Q : (a_1|b_1), (a_2|b_2), ..., (a_l, b_l)$, similar to how we constructed the worst case sequence in Theorem 5. We set $a_i = ((i-1) \mod (\frac{\alpha \cdot k}{2}+1))+1$ and $b_i = -a_i$. All pages occur in queries with distance $\frac{\alpha \cdot k}{2} + 1$ and no two elements share a page.

FPIFO misses all the queries as it keeps the cache of size $\alpha \cdot k$ filled with the pages in the previous $\frac{\alpha \cdot k}{2}$ queries, which are not of any use in the next query. It retrieves both pages in a miss, so it retrieves $\alpha ck^2 + \alpha k + 2$ times in total. MIN operates like Belady's algorithm on the a_i sequence.

To simulate MIN, we use table 3.2. When responding to the sequence Q, MIN misses $ck \cdot (\frac{(\alpha-2)}{2} \cdot k + 1) + \frac{\alpha \cdot k}{2} + 1$ queries and retrieves the same number of times.

Row	a_i sequence	Query count	MIN: miss count	MIN: cache final state
0	1, 2, ..., $\frac{\alpha \cdot k}{2} + 1$	$\frac{\alpha \cdot k}{2} + 1$	$\frac{\alpha \cdot k}{2} + 1$	1, 2, ..., k-1, $\frac{\alpha \cdot k}{2} + 1$
1	1, 2, ..., $\frac{\alpha \cdot k}{2} + 1$	$\frac{\alpha \cdot k}{2} + 1$	$\frac{(\alpha-2) \cdot k}{2} + 1$	1, 2, ..., k-2, $\frac{\alpha \cdot k}{2}$, $\frac{\alpha \cdot k}{2} + 1$
...	...	...	...	...
k-1	1, 2, ..., $\frac{\alpha \cdot k}{2} + 1$	$\frac{\alpha \cdot k}{2} + 1$	$\frac{(\alpha-2) \cdot k}{2} + 1$	$\frac{(\alpha-2) \cdot k}{2} + 2$, $\frac{(\alpha-2) \cdot k}{2} + 3$, ..., $\frac{\alpha \cdot k}{2} + 1$
k	1, 2, ..., $\frac{(\alpha-2) \cdot k}{2} + 1$	$\frac{(\alpha-2) \cdot k}{2} + 1$	$\frac{(\alpha-2) \cdot k}{2} + 1$	$\frac{(\alpha-2) \cdot k}{2} + 1$, $\frac{(\alpha-2) \cdot k}{2} + 2$, ..., $\frac{\alpha \cdot k}{2}$
Shift	$+k$			1, 2, ..., k-1, $\frac{\alpha \cdot k}{2} + 1$

Table 3.2: The table simulates MIN, working like Belady's algorithm on the a_i sequence. The queries start from row 0 and go toward row k. They return and start again from row 1 for $(c-1)$ times. To simplify simulation we shift all the numbers queried or stored in the cache before each returning to row 1.

Now we compare FPIFO and MIN:

$$MIN_{\text{retrievals}} = \frac{(\alpha-2)}{2}\cdot ck^2+ck+\frac{\alpha\cdot k}{2}+1 = \frac{(\alpha-2)}{2}k(ck+1)+ck+k+1 \leq \frac{(\alpha-2)}{2}k(ck+1+\frac{6c}{\alpha-2})$$

$$FPIFO_{\text{retrievals}} = \alpha ck^2+\alpha k+2 = \alpha k(ck+1)+2 > \alpha k(ck+1+\frac{6c}{\alpha-2})-\frac{6\alpha ck}{\alpha-2}$$

$$\rightarrow \beta = \frac{FPIFO_{\text{retrievals}}}{MIN_{\text{retrievals}}} > \frac{\alpha k(ck+1+\frac{6c}{\alpha-2})-\frac{6\alpha ck}{\alpha-2}}{\frac{(\alpha-2)}{2}k(ck+1+\frac{6c}{\alpha-2})} = \frac{2\alpha}{\alpha-2}-\frac{12\alpha c}{(\alpha-2)^2(ck+1+\frac{6c}{\alpha-2})}$$

$$k > \frac{\epsilon(\alpha-2)^2+\epsilon 6c(\alpha-2)-12ac}{-\epsilon(\alpha-2)^2c} \rightarrow \epsilon((\alpha-2)^2ck+(\alpha-2)^2+6c(\alpha-2))-12ac > 0$$

$$\rightarrow \frac{12\alpha c}{(\alpha-2)^2(ck+1+\frac{6c}{\alpha-2})} < \epsilon$$

$$\rightarrow \frac{FPIFO_{\text{retrievals}}}{MIN_{\text{retrievals}}} > \frac{2\alpha}{\alpha-2}-\epsilon \rightarrow \beta = \frac{2\alpha}{\alpha-2}-o(1)$$

We showed in Theorem 6 that FPIFO uses at most $\frac{2\alpha}{\alpha-2}$ times the retrievals. So we can conclude that the constructed sequence is a worst case example and $\frac{2\alpha}{\alpha-2}-o(1)$ is a tight worst case bound. □

Chapter 4

Discussion

The essence of this master's thesis lies in its approach toward understanding caching, specifically through the introduction and formulation of the pair cache problem. The new lens applied to this problem has led to key contributions:

This chapter outlines the contributions of this thesis and provides guidance for future researchers to build upon it. Firstly, we introduced and formulated the pair cache problem as a variant of the cache replacement problem. Secondly, we established a connection between the pair cache problem and the minimum vertex cover problem. This connection highlights the theoretical significance of the pair cache problem and its computational complexity.

In addition to defining the pair cache problem, we also defined the edge serving problem as a variant of the vertex cover problem. This definition sheds light on the practical applications of pair cache problems, such as maintaining a limited memory of size $k << n$ that can store the adjacency lists of k vertices in a large complete graph of n vertices based on the queries that need to be resolved about the edges. Future research is needed to find specific problems with real-world impact that can benefit the edge serving problem formulation.

We showed that the offline pair cache problem is an NP-Hard problem and demonstrated that even with complete knowledge of future queries, it is not easier than the online version of the normal cache problem, which is solvable using LRU or FIFO with a 2-factor approximation. This result establishes a lower bound on the computational complexity of the offline pair cache problem. We believe that this lower bound is tight, and we formulate this belief in Conjecture 1.

Conjecture 1. *There is an offline pair cache algorithm that offers a 2-factor approximation in polynomial time.*

We also introduced the FPIFO and LRUP algorithms and demonstrated that FPIFO can solve the online version of the pair cache problem with a 4-factor approximation. This gives a practical solution to the pair cache problem. Additionally, we generalized the FPIFO approximation bounds and showed that it provides a α, β-approximation if $\beta < \alpha/(\alpha - 1)$. We also established that neither FPIFO nor LRUP can achieve a better approximation. Based on our findings, we posit that the FPIFO algorithm outperforms any other online algorithm, a belief we encapsulate in Conjecture 2.

Conjecture 2. *For constants $\alpha, \beta > 2$, if the cache size of an online algorithm is α times larger and it uses β times more retrievals compared to an optimal cache baseline, then $\beta \geq \frac{2\alpha}{\alpha-2}$.*

Overall, this thesis contributes to the theoretical understanding of the pair cache problem and its practical applications. It provides a foundation for the development of efficient algorithms for the pair cache problem and sheds light on the computational complexity of the problem. Future research can build upon these findings and explore further practical applications of the pair cache problem.

References

[1] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[2] Moses Charikar, Konstantin Makarychev, and Yury Makarychev. Near-optimal algorithms for unique games. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 205–214, New York, NY, USA, 2006. Association for Computing Machinery.

[3] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[4] Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 767–775, New York, NY, USA, 2002. Association for Computing Machinery.

[5] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for max-cut and other 2-variable csps? *SIAM Journal on Computing*, 37(1):319–357, 2007.

[6] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2- ε. *Journal of Computer and System Sciences*, 74(3):335–349, 2008.

[7] Elias Koutsoupias. The k-server problem. *Computer Science Review*, 3(2):105–118, 2009.

[8] Elias Koutsoupias and David Scot Taylor. The cnn problem and other k-server variants. In *STACS 2000: 17th Annual Symposium on Theoretical Aspects of Computer Science Lille, France, February 17–19, 2000 Proceedings*, pages 581–592. Springer, 2000.

[9] Mark Manasse, Lyle McGeoch, and Daniel Sleator. Competitive algorithms for on-line problems. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 322–333, 1988.

[10] Conrado Martínez and Salvador Roura. On the competitiveness of the move-to-front rule. *Theor. Comput. Sci.*, 242(1-2):313–325, 2000.

[11] J. Ian Munro. On the competitiveness of linear search. In Mike Paterson, editor, *Algorithms - ESA 2000, 8th Annual European Symposium, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1879 of *Lecture Notes in Computer Science*, pages 338–345. Springer, 2000.

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