Caching Connections in Matchings

Yaniv Sadeh
Tel Aviv University, Israel

Haim Kaplan
Tel Aviv University, Israel

Abstract

Motivated by the desire to utilize a limited number of configurable optical switches by recent advances in Software Defined Networks (SDNs), we define an online problem which we call the *Caching in Matchings* problem. This problem has a natural combinatorial structure and therefore may find additional applications in theory and practice.

In the *Caching in Matchings* problem our cache consists of k matchings of connections between servers that form a bipartite graph. To cache a connection we insert it into one of the k matchings possibly evicting at most two other connections from this matching. This problem resembles the problem known as *Connection Caching* [20], where we also cache connections but our only restriction is that they form a graph with bounded degree k. Our results show a somewhat surprising qualitative separation between the problems: The competitive ratio of any online algorithm for caching in matchings must depend on the size of the graph.

Specifically, we give a deterministic $O(nk)$ competitive and randomized $O(n \log k)$ competitive algorithms for caching in matchings, where n is the number of servers and k is the number of matchings. We also show that the competitive ratio of any deterministic algorithm is $\Omega(\max(\frac{n}{k}, k))$ and of any randomized algorithm is $\Omega(\log \frac{n}{k^2 \log k} \cdot \log k)$. In particular, the lower bound for randomized algorithms is $\Omega(\log n)$ regardless of k, and can be as high as $\Omega(\log^2 n)$ if $k = n^{1/3}$, for example. We also show that if we allow the algorithm to use at least $2k-1$ matchings compared to k used by the optimum then we match the competitive ratios of connection catching which are independent of n. Interestingly, we also show that even a single extra matching for the algorithm allows to get substantially better bounds.

2012 ACM Subject Classification Theory of computation → Caching and paging algorithms; Mathematics of computing → Graph coloring

Keywords and phrases Caching, Matchings, Caching in Matchings, Edge Coloring, Online Algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.120

Category Track A: Algorithms, Complexity and Games

Related Version *Extended Version*: `https://arxiv.org/abs/2310.14058` [30]

Funding The work of the authors is partially supported by Israel Science Foundation (ISF) grant number 1595-19, German Science Foundation (GIF) grant number 1367 and the Blavatnik research fund at Tel Aviv University.

1 Introduction

We define the *Caching in Matchings* online problem, on a fixed set of n nodes. Requests are edges between these node. The algorithm maintains a cache of k matchings, i.e. a k-edge-colorable graph. To serve a request for an edge (u, v) which is not in its cache (i.e. a miss), the algorithm has to insert it into one of its matchings. To do this it may need to evict the edges incident to u and v in this specific matching. Note that an evicted edge may later be re-inserted into a different matching. The algorithm has to choose which matching to use for each miss in order to minimize its total number of misses.

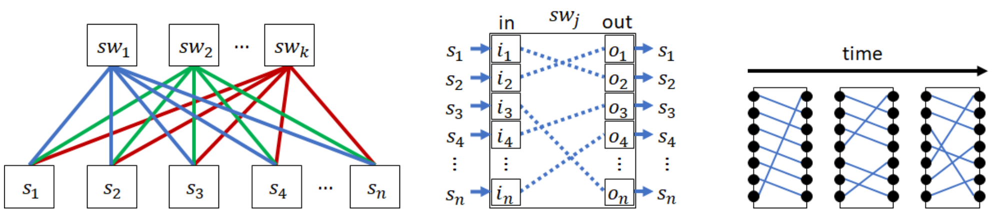


Figure 1 The physical topology that motivates our problem: n servers $s_1, \ldots, s_n$, each is connected to the in/out ports of k optical switches sw_1 through sw_k. Each switch sw_j uses mirrors to switch optical links, effectively inducing an in/out permutation, which may change over time at a reconfiguration cost of 1 per each new pairing. Abstractly, we get a bipartite graph with n nodes on each side (one per server), and each permutation is a matching that caches links.

One can look at this problem as a new variation of the online Connection Caching problem. In *Connection Caching* [20] the setup is the same, but the cache maintained by the algorithm must be a graph in which each node is of degree at most k. In case of a miss on an edge (u, v) we may choose any edge incident to u and any edge incident to v to evict. We do not have to maintain the edges partitioned into a particular set of k matchings. Thus in Caching in Matchings we are less flexible in our eviction decisions. Once we color the new edge then the two edges we have to evict are determined.

At a first glance, the two caching problems seem similar. In fact, the only difference is the added restriction of the coloring (matchings) that affects how the cache is maintained. Interestingly, it turns out that this seemingly small difference makes Caching in Matchings a much harder online problem compared to Connection Caching.

A common measure to evaluate online algorithms is their competitive ratio. We say that an online algorithm is c-competitive if its cost (in our case, miss count) on every input sequence is at most c times the minimal possible cost for serving this sequence. One would like to design algorithms with as small c as possible. The problem of Connection Caching is known to be $\Theta(k)$ (deterministic) and $\Theta(\log k)$ (randomized) competitive, and in contrast we show that the dependence on n (the number of nodes) in Caching in Matchings cannot be avoided.

The motivation to our Caching in Matchings problem comes from a data-center architecture described in [4]. In this setting we have n servers connected via a communication network which is equipped with a set O of k optical switches. Each server is connected to all the k optical switches and in each of them it is connected to both an input and an output port. Each switch is configured to implement a matching between the input and the output ports of the servers, see Figure 1. Since each server is connected to both input and output sides, the optical switches effectively induce a degree k bipartite graph with $2n$ nodes (two nodes per server). Each optical switch corresponds to a matching in our cache. It is dynamic as we can insert and evict connections from the switch, but we try to minimize these reconfigurations since they are costly (involve shifting mirrors, and down-time).

At this point we clarify that there are two "kinds" of optical switching architectures. The one which we model, as explained, is based on off-the-shelf commodity switches and is sometimes referred to as Optical Circuit Switching (OCS). Each switch is a separate box, and each box, at any time, implements a matching between its ports. We use k switches and connect every server to every switch, so this architecture induces k matching at any time. To add a connection between two servers we have to choose through which box we want to do it (choose a matching to insert it to) and then reconfigure the matching implemented by this particular box to include this edge. The other kind of switching is known as Free Space Optics (FSO) where every transmitter can point towards any receiver. When each server is connected to k transmitters and k receivers we get the standard connection caching setting. This is *not* the architecture that we model here. See Table-1 of [26] for several references and their architecture types.

Several cost models considering both communication and adjustment cost were suggested for this setting [4]. We choose to work with arguably the simplest model of paying 1 for an insertion of a new edge (formally defined in Section 2). This simple model already captures the qualitative properties of the problem. We note that the competitive results shown here can be adapted (up to constant factors) to a more complicated cost model that has additional communication costs per request. We believe that our combinatorial abstraction of this setting is natural and will find additional applications.

Here is a detailed summary of our results.

Our contributions

1. We define a new caching problem, "Caching in Matchings" (Problem 1), on a bipartite graph with n nodes on each side.[1] In this problem, the cache is a union of k matchings. When we insert an edge we pick the matching to insert it to and evict edges from this matching if necessary.
2. We show that the competitive ratio of Caching in Matchings depends not only on the cache size k as is common for caching problems, but also on the number of nodes in the network n. One might argue that since we define the cache to be k matchings, its size is $\Theta(nk)$ rather than k, so the dependency on n is not surprising. But such an argument also applies to Connection Caching [20] and in that problem the competitive ratio does not depend on n. In other words, Caching in Matchings is provably harder than Connection Caching.[2] Specifically we prove the following.
   a. An $\Omega(\max(\frac{n}{k}, k))$ lower bound on the competitive ratio of deterministic algorithms, and we give a deterministic algorithm with nk competitive ratio. For $k = O(1)$ this gives a tight bound of $\Theta(n)$ on the competitive ratio.
   b. In contrast, in the randomized case we have a larger gap. We describe an $O(n \log k)$ competitive algorithm and prove a lower bound of $\Omega(\log \frac{n}{k^2 \log k} \cdot \log k)$ on the competitive ratio. This bound is $\Omega(\log n)$ for any k, and can get as worse as $\Omega(\log^2 n)$, for example if $k = n^{1/3}$. This is in contrast to other caching problems whose randomized competitive ratio is logarithmic.[3]
3. We show that resource augmentation of almost-twice as many matchings, specifically $2k-1$ for the algorithm versus k for the optimum, allows to get rid of the dependence on n. Specifically, we show a deterministic $O(k)$ competitive algorithm and a randomized $O(\log k)$ competitive algorithm for this case. Furthermore, with $2(1+\alpha)k$ matchings we get a deterministic $O(1+\frac{1}{\alpha})$ competitive algorithm. We also show that a single extra matching already helps by allowing us to "trade" $\sqrt{n}$ for $\sqrt{k}$ in the competitive ratio. Concretely and more generally, with $h \geq 1$ extra matchings we get a deterministic $O(n^{1/2}(k/h)^{3/2})$ and a randomized $O(n^{1/2}(k/h)^{1/2} \log \frac{2k+h}{h})$ competitive algorithms. Moreover, it is even possible to reduce the dependence on n to polylogarithmic at the cost of higher polynomial dependency on k, which is beneficial for small k. Concretely, following [19], we get a deterministic $O\Big(\frac{k^6 \log n}{h} \min(k, \log n)\Big)$ and a randomized $O\Big(\big(\frac{k \log k}{h}\big)^6 \log \frac{2k+h}{h} \log^9 n\Big)$ competitive algorithms. The deterministic algorithm is not efficient.

[1] The problem makes sense on a general graph as well.
[2] In terms of the architecture, we show that the FSO architecture has a better competitive ratio than the the OCS architecture.
[3] Throughout the paper, where it matters, our logarithms are in base 2.

Our problem is a special case of a more general problem of convex body chasing in L_1. Bhattacharya et al. [11] gave a fractional algorithm for this body chasing problem with packing and covering constraints. Their fractional algorithm requires a slight resource augmentation. For a few special cases, they show how to round their fractional solution to an integral solution that does not use additional resources. Our problem is another interesting test-case of this general setting (for more details, see the appendix in [30]).

Our full list of results is summarized in Table 1. The rest of the paper is structured as follows. Section 2 formally defines the model, the notations that we use, and the caching problems. Section 3 studies in depth the Caching in Matchings problem (Problem 1). Section 4 surveys related work on caching and coloring problems, and in Section 5 we conclude and list a few open questions. Section 6 serves as an appendix that contains deferred proofs, and a few additional discussions. Due to a strict page limit, the complete appendix can be found in the extended version [30].

2 Model and Definitions

In the following we formally define two caching problems of interest, the premise of each of them is a graph with a set V of n nodes. Every turn, a new edge is requested. If it is already cached, we have a "hit" and no cost is paid. Otherwise, we have a "miss", and the edge must be brought into the cache at a cost of 1, possibly at the expense of evicting other edges. In fact, the problem that arises from [4] consists of a bipartite graph in which each server v is associated with two nodes $v^{in} \in V^{in}$ and $v^{out} \in V^{out}$, modeling its receiving and sending ports, respectively. Each among v^{in} and v^{out} can be incident to one edge in each matching.[4] Formally the problem is as follows.

▶ **Problem 1** (Caching in Matchings). Requests arrive for edges $(u, v) \in V^{in} \times V^{out}$. The cache M is a union of k matchings. When a requested edge is missing from all the matchings, an algorithm must fetch it into one of the matchings (possibly evicting other edges from this matching). In addition, the algorithm may choose to add any edge to the cache at any time (while maintaining the cache's restrictions), the cost of adding an edge to the cache is 1. It is not allowed to move an edge between matchings, but an edge may be evicted and immediately re-fetched into a new matching.

▶ Remark 1. There are other caching models in which reorganizing the cache is free, such as [16, 25]. In our model reorganizing the matchings incurs a cost. This is because we model a setting where changing the cache (physical links) is slow. In other cases accessing the slow memory is the costly operation.

We use the terminology of coloring edges when discussing Caching in Matchings (Problem 1). Recoloring an edge implies that we evict it, and then immediately fetch it back into a different matching according to the new color of the edge. Recoloring is not free, but has the same cost of standard fetching. This models, for example, the physical setting in which such a rearrangement requires reconfiguring the link in a different optical switch.

▶ **Problem 2** (Connection Caching [20]). Requests arrive for edges $(u, v) \in V^{in} \times V^{out}$. The cache M is a set of edges such that every node is of degree at most k in the sub-graph induced by M. When a requested edge is missing from M, an algorithm must fetch it (possibly

[4] Technically, the physical switch can be configured with links of the form (v^{in}, v^{out}), but it makes no sense and practically such requests do not exist. However, our algorithms can deal with all possible requests, and our lower bounds are proven without relying on such requests, so we ignore this nuance.

evicting other edges). In addition, the algorithm may choose to add any edge to the cache at any time while maintaining the degrees at most k. Adding an edge to the cache costs 1.

▶ Remark 2. Note that in both problems that we defined, an algorithm is allowed to add (fetch) and remove (evict) additional edges. Technically, it is not strictly necessary because a non-lazy algorithm can always be simulated by a lazy version that fetches an edge only when it is actually needed. This is also true for the offline optimum. We will describe non-lazy algorithms for Caching in Matchings, that recolor edges, to simplify the presentation.

To emphasize the difference between the problems see Figure 2, which shows the difference on bipartite graphs, as well as on general graphs (for the generalized problem).

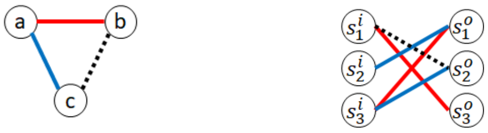


Figure 2 An example of the difference between connection caching versus caching in matchings. (left) With $k = 2$ max degree and $n = 3$ nodes, all the connections can be cached simultaneously, but not in 2 matchings, red and blue. (right) Bipartite example: Caching the edge (s_1^i, s_2^o) (i/o for in/out) is not possible without changing the matchings (red: $\{(s_1^i, s_3^o), (s_3^i, s_1^o)\}$; blue: $\{(s_3^i, s_2^o), (s_2^i, s_1^o)\}$), although both s_1^i and s_2^o only have a single connection.

The objective of an online algorithm is to minimize the number of fetched edges. We are interested in the competitive-ratio of our algorithms.

▶ **Definition 3** (Cost, Competitive Ratio)**.** Consider a specific caching problem. Let A be an online algorithm that serves requests, and let σ be a sequence of requests. We denote by $A(\sigma)$ the execution of A on σ, and $cost(A(\sigma))$ for the cost of A when processing σ.

We denote by $OPT(\sigma)$ the optimum (offline) algorithm to serve the sequence, or simply OPT when σ is clear from the context. If there exist functions of the problem's parameters (in our case: k and n) $c = c(n, k)$ and $d = d(n, k)$ such that $\forall\sigma : cost(A(\sigma)) \leq c \cdot cost(OPT(\sigma)) + d$ then we say that A is c-competitive. Note that σ may be arbitrarily long, so the "asymptotic ratio" is indeed c.

▶ Remark 4. Denote the optima for Caching in Matchings and Connection Caching by OPT_m and OPT_c, respectively. Since OPT_m implicitly maintains a connections cache as required by Connection Caching (ignore the colors), then for any sequence of edge requests σ, $cost(OPT_c(\sigma)) \leq cost(OPT_m(\sigma))$.

3 Caching in Matchings

In this section we study the problem of Caching in Matchings (Problem 1). We summarize the results of this section in Table 1. We start with upper bounds (Section 3.1), then lower bounds (Section 3.2). Then we study resource augmentation (Section 3.3). Some additional discussion on randomization is detailed in the appendix in [30].

3.1 Upper Bounds for Bipartite Graphs

In this section we prove upper bounds on the competitive ratio of algorithms for Caching in Matchings, focusing on the non-trivial case of $2 \leq k \leq n - 1$. Indeed, if $k = 1$ there are no eviction-decisions to take so the only (lazy) algorithm is the optimal one. The other

Table 1 Our bounds on the competitive ratio for Caching in Matchings (Problem 1), for $2 \leq k < n$. If $k = 1$ or $k \geq n$ optimality is trivial. Results marked with $*$ also apply to general graphs. "RA: x" shortens Resource-Augmentation, i.e., the algorithm has more matchings (x) than the optimum (k).

Result		Deterministic	Randomized	Notes
Thm. 7		$\leq nk$	$O(n \log k)$	The standard scenario
Thm. 9	$*$	.	$\Omega(\log \frac{n}{k^2 \log k} \cdot \log k)$	.
Cor. 10	$*$	$\Omega(max(\frac{n}{k}, k))$	$\Omega(\log n)$	Due to Theorems 8+9
Cor. 10	$*$	.	$\Omega(\epsilon \cdot \log n \cdot \log k)$	$k = O(n^{1/2-\epsilon})$; Due to Theorem 9
Cor. 25(1)		$O(n^{1/2}(k/h)^{3/2})$	$O(n^{1/2}(k/h)^{1/2} \log \frac{2k+h}{h})$	RA: $k + h$ for $1 \leq h \leq k$
Cor. 25(2)	$*$	$O\left(\frac{k^6 \log n}{h} \min(k, \log n)\right)$	$O\left(\left(\frac{k \log k}{h}\right)^6 \log \frac{2k+h}{h} \log^9 n\right)$	RA: $k + h$ for $1 \leq h \leq k$
Cor. 25(3)	$*$	$\leq k$	$O(\log k)$	RA: $2k - 1$
Cor. 25(4)	$*$	.	$O(\alpha^4 \log k)$	RA: $(1 + O(\frac{1}{\alpha}))k$ for $k \geq \Theta(\alpha^4 \log n)^{\Theta(\alpha \log \alpha)}$
Thm. 27	$*$	$O(1 + \frac{1}{\alpha})$	.	RA: $(2 + \alpha)k$ for $\alpha > 0$

extreme case of $k \geq n$ in bipartite graphs is also easy since we can just cache the entire graph: Number the nodes 0 to $n - 1$ on each side, and use matching i to store edges from node j to $i + j$ modulo n.

Our general technique is to reduce the problem of Caching in Matchings to Connection Caching. Our algorithm, A_m, will run a Connection Caching algorithms A_c with cache parameter k to insert requested edges into the cache. Then, layered on top of A_c, we have the "coloring component" of A_m that chooses the color of the new edge, and also recolors existing edges in order to produce a proper Caching in Matchings algorithm. A_m can be thought of as an edge coloring algorithm in the dynamic graph settings, and in this context A_c is the adversary that tells A_m which edges are inserted and which are removed (with a guarantee of bounded degree k). As a consequence we would like to use algorithms that are efficient in terms of recoloring, to achieve the best competitive results. Unfortunately, since edge coloring of graphs of bounded degree k may require $k + 1$ colors by Vizing's theorem, the dynamic graph coloring literature studies this coloring problem while typically allowing more than k colors. The number of extra colors ranges from $k + 1$ colors [29, 10], to $(1 + \epsilon)k$ colors [21, 19, 13], to $2k - 1$ colors [9, 12], and sometimes even more [8, 31] (the last citations actually study vertex coloring). Extra colors correspond to resource augmentation, which we study later in Section 3.3.

▶ Remark 5. There are known algorithms that are k competitive deterministic and $O(\log k)$ competitive randomized for Connection Caching, as studied in [20].

▶ Remark 6. Due to Remark 4 and Remark 5, it suffices to analyze the cost ratio between A_m and A_c. A ratio of ρ implies a $\rho \cdot k$ deterministic and a $O(\rho \cdot \log k)$ randomized competitive algorithms for Caching in Matchings.

▶ **Theorem 7.** There exist nk deterministic and $O(n \log k)$ randomized competitive algorithms in bipartite graphs for Caching in Matchings.

Proof. By Remark 5 and Remark 6, it suffices to show that A_m pays no more than n times compared to A_c. Whenever an edge (u, v) is requested, A_m has it cached if and only if A_c has it cached. Therefore when A_m has a miss, so does A_c. To accommodate for the edge, A_c ensures that u and v are both of degree $k - 1$ before (u, v) is inserted. Now consider how many edge recolorings are required from A_m. Nodes u and v each have at least one free color. If both have some common free color c, we are done. Otherwise, u has c_1 free and v has $c_2 \neq c_1$ free. Let P_u and P_v be the (c_1, c_2) bi-colored paths that originate in u and v

respectively. P_u and P_v must be disjoint because the graph is bipartite and does not contain odd cycles. Flipping the colors $(c_1 \leftrightarrow c_2)$ for each edge on either P_u or P_v enables A_m to insert and color (u, v). since P_u, P_v and (u, v) form a simple path in a graph with $2n$ nodes, by flipping the shorter bi-colored path, A_m colors at most n edges when inserting (u, v). ◀

3.2 Lower Bounds

Caching in Matchings is a generalization of caching, if we restrict the requests to edges of a single fixed node. Observe, therefore, that any c-competitive online algorithm for Caching in Matchings with $2 \leq k < n$ satisfies $c = \Omega(\log k)$. Moreover, if the algorithm is deterministic then $c \geq k$. The following lower bounds depend on n as well as k. These bounds hold for the non-trivial case of $2 \leq k < n$, in bipartite graphs, and therefore also hold for general graphs. Theorem 8 is proven later in this section, the proof of Theorem 9 is deferred to Appendix 6.1.

▶ **Theorem 8.** Any *deterministic* Caching in Matchings algorithm is $\Omega(\frac{n}{k})$ competitive.

▶ **Theorem 9.** *Any* Caching in Matchings algorithm is $\Omega(\log \frac{n}{k^2 \log k} \cdot \log k)$ competitive.

▶ **Corollary 10.** *Any* online algorithm for Caching in Matchings with $2 \leq k < n$ is $\Omega(\log n)$ competitive. Moreover, if $k = O(n^{1/2-\epsilon})$ for some $\epsilon > 0$, we get that any online algorithm for Caching in Matchings is $\Omega(\epsilon \cdot \log n \cdot \log k)$ competitive. If the algorithm is deterministic then the competitive ratio is $\Omega(\max\{\frac{n}{k}, k\})$.

Proof. The deterministic claim follows from the initial observation and Theorem 8. In the general case (randomized), we get $\Omega(\log n)$ from the maximum between the observation (when $k \geq n^{1/3}$) and Theorem 9 (when $k < n^{1/3}$). The $\Omega(\epsilon \cdot \log n \cdot \log k)$ bound follows from Theorem 9: If $k \leq c \cdot n^{1/2-\epsilon}$ for some constant c then $\log \frac{n}{k^2 \log k} > \log \frac{n^{2\epsilon}}{c^2 \log(cn)} = 2\epsilon \log n - 2 \log c - \log\log(cn) = \Omega(\epsilon \cdot \log n)$. ◀

We prove the lower bounds in a setting that is closer to dynamic graph coloring. Specifically, we define Problem 3 below, where we control which edges must be cached both by the algorithm and the optimum. We prove (Lemma 11 below, proven in Appendix 6.1) that lower bounds for algorithms for Problem 3 imply lower bounds for Caching in Matchings, and then study lower bounds for Problem 3.

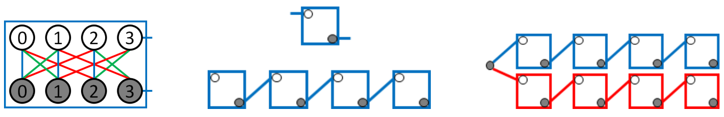
▶ **Problem 3.** Given a graph with n vertices, we get a sequence of actions that define a subset of edges at any time. Each action either adds a missing edge or deletes an existing edge. We are guaranteed that at any point in time the graph induced by existing edges, denote it (or the set of edges) by G, has a proper k-edge-coloring. An algorithm, online or OPT, must maintain k matchings, denote their union by M, such that G is a subgraph of M. For every edge that is added to a matching, the algorithm pays 1.

Note that Problem 3 is similar but not equivalent to dynamic edge coloring. On one hand a dynamic edge coloring algorithm that recolors $O(C)$ edges per update is not necessarily $O(C)$ competitive for Problem 3. The reason for this is that we allow M to contain G. By maintaining an edge in M we can avoid paying for it when it is inserted again. For example, in the proof of Theorem 8 an algorithm may do $O(1)$ worst-case recolorings per step, but its competitive ratio is $\Omega(\frac{n}{k})$ since OPT stores in M extra edges that this online algorithm keeps paying for. On the other hand, an algorithm that is $O(C)$ competitive for Problem 3 does not give a dynamic edge coloring algorithm that recolors $O(C)$ edges per update, even amortized, because it could be that both OPT and the algorithm pay a lot per edge update on some sequence, and while the ratio is $O(C)$, the absolute cost is large.

▶ **Lemma 11.** A lower bound of C on the competitive ratio of an online algorithm for Problem 3 implies a lower bound of C on the competitive ratio of an online algorithm for Caching in Matchings.

We now focus on deriving lower bounds for Problem 3. We define a *road* gadget (Definition 13) which is a connected component with a large diameter, that is also very restricted in the way it can be colored. A road is constructed from *brick* sub-gadgets (Definition 12), each of size $\Theta(k)$ nodes and $\Theta(k^2)$ edges. By connecting $r \geq 2$ roads together we get the *r-road* gadget, whose structure enforces those roads to be colored in a distinct and different way.

▶ **Definition 12** (Brick)**.** A *colorless-brick* is a union of k perfect matchings in a bipartite graph with the following structure. Each side has w nodes where w is the unique power of 2 that satisfies $\frac{w}{2} < k \leq w$. Number the nodes on each side $0, \ldots, w-1$, and number the colors $0, \ldots, k-1$. The matching of color c matches node i with node $i \oplus c$ where $\oplus$ is the bitwise exclusive-or. See Figure 3a for an example. Note that every color c indeed defines a matching that is in fact a permutation of order 2, and that $v \oplus c \neq v \oplus c'$ for any two colors $c \neq c'$ so the matchings are all disjoint. When we remove an edge from a colorless-brick, we get a *brick* whose color is associated with the color of the non-perfect matching. The two nodes of degree $k-1$ are the *endpoints* of the brick.



(a) Brick. **(b)** Road. **(c)** r-Road for $r = 2$.

Figure 3 Visualization of a *brick* (Definitions 12), a *road* and an *r-road* (Definition 13). (a) A brick for $k = 3$ ($w = 4$). The number of each node is written, and the colors are as follows: blue (0), green (1), red (2). Thus, for example, $1 \oplus red = 3$. We removed a single edge, blue $(3,3)$, thus the brick is blue. (b) A schematic way to draw a brick (top) and a road of length 4 (bottom) which is a chain of bricks connected to each other by their endpoints. The color of a road is well-defined by the color of its bricks. (c) An r-road for $r = 2$, of length 4. The node that connects to both roads is its hub. Nodes are colored by gray and white according to their side in the bipartite graph.

▶ **Definition 13** (Road, r-road)**.** A *road* of length $d \geq 1$ is an edge colored graph obtained by connecting a sequence of d bricks. Each brick is connected by an edge to the next brick in the sequence. The edge connecting two consecutive bricks is adjacent to an endpoint of each brick. Note that the color of two connected bricks must be the same since they must agree on their free color which is the color of the edge which connects them. Therefore, we define the *color of a road* to be the color of its bricks. A road has two *ends*, which are the endpoints of its first and last bricks. We also refer to r ($2 \leq r \leq k$) roads of the same length d that are all connected to a single shared node as an *r-road* of length d. The shared node is its *hub*. The edges of the hub all have different colors, therefore all the roads of an r-road have different colors. See Figure 3 for examples.

▶ **Lemma 14.** Given a brick B of color c_1, and a new color $c_2 \neq c_1$, it is always possible to recolor 3 edges to change the color of B to c_2.

Note that when we recolor B, it no longer satisfies the $\oplus$-property of Definition 12, but for convenience we still consider it as a brick. This would not affect our arguments below (by more than a constant factor) since we will make sure to always return to the original coloring (undo) before recoloring again.

Proof. Denote by u one endpoint of the brick. By definition of the matching scheme, the other endpoint is $u \oplus c_1$, and when we restrict the graph to edges of colors c_1 and c_2, we find that the path between u and $u \oplus c_1$ is of length 3: $u \oplus c_1 \to u \oplus c_1 \oplus c_2 \to u \oplus c_2 \to u$. Therefore, it suffices to flip the color of these three edges from c_2 to c_1 and vice versa. ◀

In the remainder of this section we prove the deterministic lower bound, and a simpler but weaker version of the randomized lower bound. The more involved randomized lower bound is proven in Appendix 6.1, using the same gadgets.

Proof of Theorem 8. We prove the lower bound for Problem 3. Then the theorem follows by Lemma 11. We present an adversarial construction against a given algorithm ALG.

We begin by setting aside one special node u to serve as a 2-road hub, and divide the rest of the vertices into bricks. We construct from these bricks the longest possible road, of length $N = \Theta(\frac{n}{k})$. We number the bricks in order, from 1 to N, and denote the edge between bricks i and $i+1$ by $(i, i+1)$. Let $L = \lfloor \frac{N}{3} \rfloor$. Initially we insert all the edges of all the bricks, without the edges connecting the bricks. These edges are never deleted. Our sequence has as many steps as we like as follows, based on the state of ALG, see also Figure 4:

1. Simple step: If there exist consecutive bricks i and $i+1$ of different colors, we insert the edge $(i, i+1)$. This forces ALG to recolor at least one of the bricks and pay $\Omega(1)$ (it could pay more if it recolors more bricks or does other actions). We then delete $(i, i+1)$.
2. Split step: Otherwise, all the bricks of ALG have the same color. We insert all the edges between bricks 1 through to L, and between $N+1-L$ through to N. We also insert edges from u to bricks 1 and $N+1-L$. These insertions construct a 2-road with u as its hub, that guarantees different colors for bricks 1 to L compared to bricks $N+1-L$ to N. ALG must recolor at least $L = \Omega(N)$ bricks. We then delete the edges that we inserted.

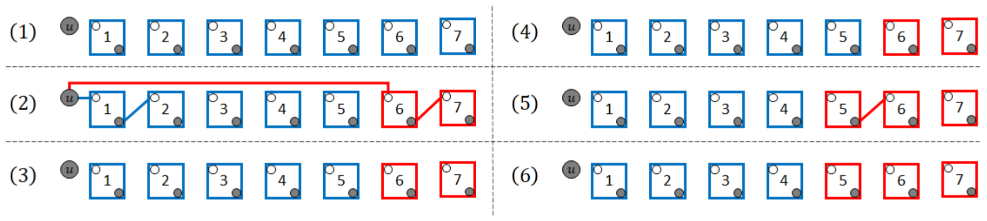


Figure 4 Visualization for the proof of Theorem 8, for $N = 7$, $L = \lfloor \frac{N}{3} \rfloor = 2$. (1)-(3) A *split step* temporarily creates a 2-road of length L out of the first and last bricks, with u as the hub, to guarantee consecutive bricks with different colors. (4)-(6) A *simple step* finds two consecutive bricks with different color (here: 5 and 6), and inserts temporarily the edge between them to enforce recoloring one of them.

In simple terms, we maintain a hole in the road which is where the color of the bricks changes (there could be multiple holes). In every step we request this hole, and once the hole disappears, the split action re-introduces a hole back near the middle of the road.[5]

Now let us analyze the costs. If the sequence contains m simple steps and s split steps, then $cost(ALG) = \Omega(m + s \cdot N)$. For OPT, we define a family of strategies B_i for $L < i < N+1-L$, to bound its cost. We define B_i to store all the edges of all the bricks, all the edges connecting them, and the edge that connects u to the first brick, except for

[5] This idea is similar to the way one can prove a deterministic lower bound for k-server [15], by always requesting a server-less location. The analogy stops here, since in our case sometimes there is no hole.

the edges that connect brick i to its neighbours, paying an initial $O(N \cdot k^2)$ cost. B_i colors all the bricks from 1 to i in one color, and all the bricks from $i+1$ to N in another color. Whenever a simple step happens in $(i-1, i)$ or in $(i, i+1)$, B_i simply recolors brick i to have the same color of the neighbour it connects to, and also inserts the connecting edge. Simple steps at other locations do not affect B_i. When a split step happens, B_i pays exactly 2: It inserts the edge that connects u to brick $N+1-L$ instead of the edge $(N-L, N+1-L)$. When the split step ends, it undoes the change, re-inserting $(N-L, N+1-L)$ instead of the edge of u. Since $L < i < N+1-L$, B_i does not have to recolor any other edge.

Thus, if we denote by m_i the number of simple steps that insert the edge $(i, i+1)$, then $cost(OPT) \leq cost(B_i) = O(m_i + m_{i-1} + s + N \cdot k^2)$. Now: $\frac{N}{3} \cdot cost(OPT) \leq (N - 2L) \cdot cost(OPT) \leq \sum_{i=L+1}^{N-L} cost(B_i) = O(m + N \cdot s + N^2k^2)$. Note that $N^2k^2 = O(n^2)$. Thus by extending the sequence such that $m + s = \Omega(n^2)$, we get: $\frac{N}{3} \cdot cost(OPT) = O(m + N \cdot s) = O(cost(ALG))$. Therefore, $\frac{cost(ALG)}{cost(OPT)} = \Omega(N)$. Recall that $N = \Theta(\frac{n}{k})$, and the claim follows. ◀

When proving the deterministic lower bound (Theorem 8) we heavily relied on determinism to know where to find neighbouring bricks of different colors. In the randomized case, we may not know where colors mismatch. Instead, we use a different and weaker construction for the randomized case. Relying on Yao's principle [32] (see also [15]), we define a distribution over sequences that is hard for any algorithm.

The following Lemma 15 is a weaker but also simpler version of Theorem 9 (proven in Appendix 6.1). Proving this lemma demonstrates our main technique.

▶ **Lemma 15.** *Any* Caching in Matchings algorithm with $2 \leq k < n$ matchings is $\Omega(\log \frac{n}{k})$ competitive.

Proof. We prove the lower bound for Problem 3. Then the theorem follows by Lemma 11. For convenience, since any action of fetching or recoloring can be lazily postponed to the next request for adding an edge to G, we assume that the algorithm does nothing else when an edge is removed.

Given a fixed k, we divide the nodes to r-roads of length d_0. We aim to have 2^h r-roads in total, for h as large as possible. We can bound h from below by noting that a brick requires at most $4k$ nodes, a road of length d_0 requires at most $4k \cdot d_0$ nodes (d_0 bricks), and an r-road requires at most $4k \cdot d_0 \cdot r + 1$ nodes (r roads and a hub). So we get that $h \geq \lfloor \log \frac{n}{4k \cdot d_0 \cdot r + 1} \rfloor$. Simplified, we get $h = \Omega(\log \frac{n}{k \cdot d_0 \cdot r})$. Eventually we will choose $r = 2$ and $d_0 = 1$ to get $h = \Omega(\log \frac{n}{k})$.

We construct the distribution over request sequences in phases. A phase begins with 2^h r-roads. Then, during each phase we have h rounds, numbered from $i = 1$ to $i = h$. In each round we pair the r-roads, and merge each pair into a single twice-longer r-road. Note that in round i there are 2^{h-i} pairs of r-roads whose length is $d_i = d_0 \cdot 2^{i-1}$. Once we get to a final single r-road of length d_h, we delete the edges that were used to connect the roads of length d_0, and insert back hub edges to re-form the initial r-roads of length d_0. Then a new phase begins.

We explain later exactly how a pair of r-roads of length d_i are merged, for now just assume that OPT pays $O(r)$ and that any algorithm pays $\Omega(r \cdot d_i)$ in expectation for such merge, and let us analyze the competitive ratio. When a phase begins, OPT can choose a consistent color for each road of the r-roads such that no further recoloring is necessary during this phase, at a cost of $O(2^h \cdot r \cdot d_0)$ by recoloring $O(1)$ edges in each brick (according to Lemma 14) and each edge that connects to a brick to the desired color. Then, throughout the rounds it pays additional $\sum_{i=1}^{h} 2^{h-i} \cdot O(r) = O(2^h \cdot r)$. Finally, when a phase ends it pays $O(2^h \cdot r)$

more to re-attach hubs when re-creating the initial r-roads of length d_0, and $O(2^h \cdot r \cdot d_0)$ more to undo any recoloring made when the phase begun.[6] Overall, OPT pays per phase $O(2^h \cdot r \cdot d_0)$. In contrast, ALG pays at least $\sum_{i=1}^{h} 2^{h-i} \cdot \Omega(r \cdot 2^{i-1} \cdot d_0) = \Omega(h \cdot 2^h \cdot r \cdot d_0)$.

Let t be the number of phases. The one-time initialization cost is $c_0 = \Theta(k^2 \cdot (2^h \cdot r \cdot d_0))$ for inserting $\Theta(k^2)$ edges per brick. Therefore, the competitive ratio that we get is $\frac{\mathbb{E}[cost(ALG)]}{cost(OPT)} \geq \frac{c_0 + t \cdot \Omega(h \cdot 2^h \cdot r \cdot d_0)}{c_0 + t \cdot O(2^h \cdot r \cdot d_0)}$. For $t = \Omega(k^2)$ we can neglect c_0 and get that $\frac{\mathbb{E}[cost(ALG)]}{cost(OPT)} = \Omega(h)$. Recall that $h = \Omega(\log \frac{n}{k \cdot r \cdot d_0})$, so we choose $r = 2$ and $d_0 = 1$ to maximize the competitive ratio and get $\Omega(\log \frac{n}{k})$, as claimed.

It remains to explain how we merge a pair of r-roads of length d, X and Y, see Figure 5 for a visual example for $r = 2$ and $k = 4$. We have r iterations, where iteration i cuts the ith road of Y away from the hub, and extends a uniformly random not yet extended road of X. OPT pays at most r for the newly introduced r edges because it can refrain from recoloring roads. As for ALG, observe that it must recolor a road if the colors of the extended road and its extension do not match. For the first two roads that we combine (one from each r-road), there is a probability of at most $\frac{1}{r}$ for the colors to agree (the probability is maximized if X and Y use the same colors for their roads, out of the k possible colors). More generally, in the ith iteration there is a probability of at most $\frac{1}{r+1-i}$ for the colors to agree, maximized if the remaining roads of Y share their colors with the not yet extended roads of X. Thus ALG recolors in expectation at least $\sum_{i=1}^{r} (1 - \frac{1}{r+1-i}) = r - H_r$ roads throughout the process, where H_r is the rth harmonic number. For $r \geq 2$, this amounts to a cost of $\Omega(r \cdot d)$ recolorings. ◀

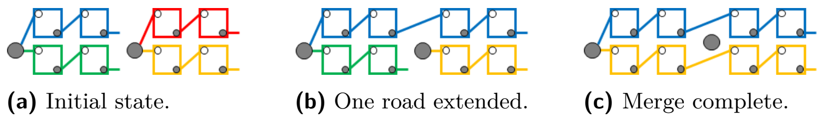

(a) Initial state. **(b)** One road extended. **(c)** Merge complete.

Figure 5 Visualization of merging a pair of 2-roads of length $d = 2$ to a single twice longer 2-road. In this example $k \geq 4$. Initially the 2-roads are disjoint. Then, we cut a road from the right 2-road and extend another road in the left 2-road. If necessary, the algorithm recolors one or more of the roads. Then we do the same for the remaining road. In the end, the hub of the cut-down 2-road is a node of degree 0.

3.3 Upper Bounds with Resource Augmentation

In this section we study upper bounds with resource augmentation. That is, we assume that the optimum still has k matchings, but our algorithm has more. Interestingly, it dramatically improves the competitive ratios, in both deterministic and randomized settings.

Recall our general approach and notations: our caching in matchings algorithm A_m implements a component of an edge-coloring algorithm, over a component of a connection caching algorithm which we denote by A_c. By Remark 6, we can divide our attention between connection caching and dynamic edge-coloring. Concretely, given $h \geq 1$ extra matchings, we maintain connection caching with $k' \equiv k + h_1$ connections per node, and maintain edge-coloring of a graph of maximum degree k', with $k' + h_2$ colors, such that $h_1 + h_2 = h$. We choose $h_2 \geq 1$ because a single extra color yields a dramatic improvement. We use either $h_2 = \lceil \frac{h}{2} \rceil$ or $h_2 = h$, Corollary 25 summarizes our choices.

[6] It is necessary to revert to the exact initial coloring before the next phase because Lemma 14 for recoloring bricks requires a very specific coloring scheme.

We begin by listing three important facts about caching and connection caching algorithms, which we use as A_c. Concretely, Lemma 16 details a deterministic algorithm for Connection Caching, and Lemma 17 together with Theorem 18 yield a randomized algorithm for Connection Caching in Corollary 19.

▶ **Lemma 16** (Corollary 8 of [20]). There is a deterministic Connection Caching algorithm with cache of size r, that is $\frac{2r}{r-k+1}$-competitive against the optimum with cache of size $k \leq r$.

▶ **Lemma 17** (Section 2.2 of [33]). Let r be the cache size of the randomized caching algorithm *MARK* [24], and let k be the cache size of the optimum. Then *MARK* is: $O(\log r)$-competitive if $r = k$; $O(\log \frac{r}{r-k})$-competitive if $\frac{e-1}{e}r < k < r$; and, 2-competitive if $k \leq \frac{e-1}{e}r$.

▶ **Theorem 18** (Theorem 7 of [20]). Let A be a $c(r,k)$-competitive caching algorithm, with additive term δ, where r and k are the cache sizes of the algorithm and the optimum, respectively. Then there is a $2 \cdot c(r,k)$-competitive algorithm for Connection Caching, with additive term $|V| \cdot \delta$ where $|V|$ is the number of nodes in the graph.

The explicit reduction and proof of Theorem 18 can be found in [30].

▶ **Corollary 19.** There exists a randomized connection caching algorithm, with cache size r compared to k of the optimum, that is $O(\log r)$-competitive if $r = k$; $O(\log \frac{r}{r-k})$-competitive if $\frac{e-1}{e}r < k < r$; and, 4-competitive if $k \leq \frac{e-1}{e}r$.

Next, we list several results for dynamic edge coloring.

▶ **Lemma 20** (Greedy, Folklore). Let G be a dynamic graph with the guarantee that its maximum degree is at most k at any time. Then we can maintain for it a $2k-1$ edge-coloring without needing to recolor any edge.

Proof. When (u,v) is inserted, both u and v are of degree at most $k-1$, thus each has at least k free colors, and they must have at least one common free color that we can use. ◀

▶ **Lemma 21.** Let G be a dynamic *bipartite* graph with the guarantee that its maximum degree is at most k at any time, and let $h \geq 1$. We can maintain a deterministic $(k+h)$-edge-coloring of G in amortized $O(\sqrt{nk/h})$ recolorings per insertion.

The proof of Lemma 21 is deferred to Appendix 6.2. It is like the proof of Theorem 7 but we only have few edges from each extra color, such that we can recolor bi-chromatic paths quickly. Periodically, we recolor the graph using only k colors and amortize this work over several operations.

The following results are particularly useful when k is small. The high probability in Theorem 23 below is for bounding the running time, not for getting a proper coloring.

▶ **Theorem 22.** Let $G' \equiv G \cup \{e\}$ be a graph with maximum degree k, such that G is $(k+1)$-edge-colored, and e is uncolored. Then there is a $(k+1)$-edge-coloring of G' which recolors only N edges in G where $N = O(k^7 \log n)$ by [19] (Theorem 3), or $N = (k+1)^6 \log^2 n$ by [10] (Corollary 6.4).

▶ **Theorem 23** (Theorem 6 of [19]). Let G be a dynamic graph such that its maximum degree never exceeds k. Then there exists a fully-dynamic algorithm that maintains a $\lceil (1+\epsilon)k \rceil$-edge-coloring with $O(\epsilon^{-6} \log^6 k \log^9 n)$ worst-case update time with high probability.

The paper [13] gives an efficient randomized edge-coloring for sufficiently large k.

▶ **Theorem 24** ([13]). Let G be a dynamic graph such that its maximum degree never exceeds k. If $k \geq (100\alpha^4 \log n)^{30\alpha \log \alpha}$, there is a fully-dynamic algorithm maintaining a $(1 + O(\frac{1}{\alpha}))k$-edge-coloring with $O(\alpha^4)$ edge recolorings in expectation per update.

Finally, we combine the various results of connection caching and edge coloring to derive the following competitive algorithms.

▶ **Corollary 25.** Given resource augmentation of extra $1 \leq h = O(k)$ matchings, that is $k + h$ for the algorithm versus k for OPT, the following competitive algorithms for caching in matchings exist:

1. $O(n^{1/2}(k/h)^{3/2})$ deterministic and $O(n^{1/2}(k/h)^{1/2} \log \frac{2k+h}{h})$ randomized competitive algorithms in bipartite graphs.
2. $O\Big(\frac{k^6 \log n}{h} \min(k, \log n)\Big)$ deterministic and $O\Big(\big(\frac{k \log k}{h}\big)^6 \log \frac{2k+h}{h} \log^9 n\Big)$ randomized competitive algorithms in general graphs. The deterministic algorithm is inefficient.
3. If $h = k - 1$ we can remove the dependence on n, yielding k deterministic and $O(\log k)$ randomized competitive algorithms in general graphs.
4. $O(\alpha^4 \log k)$ randomized competitive algorithm, where $\alpha = O(\frac{k}{h})$ and provided that $k \geq (100\alpha^4 \log n)^{30\alpha \log \alpha}$.

Proof. We use the augmentation to have extra h_2 colors for the coloring component, where $h_2 = h$ for Part-(3), and $h_2 = \lceil \frac{h}{2} \rceil$ for the rest. Part-(1) is by Lemma 21 with Lemma 16 and Corollary 19. Part-(2) is by Theorem 22 with Lemma 16, and by Theorem 23 with Corollary 19. The deterministic algorithm is inefficient because Theorem 22 only proves the existence of the stated recoloring by probabilistic arguments. Regarding the randomized part, Theorem 23 guarantees recoloring that is cheap with high probability. We can choose the constants such that the failure probability is $\leq \frac{1}{n^2}$, and fully recolor the graph if the cheap method fails. The expected number of recolorings is negligibly affected, and proper edge coloring is guaranteed. Notice that we set $\epsilon = \frac{h}{k}$. Part-(3) is by Lemma 20. Part-(4) is by Theorem 24 and Corollary 19. ◀

▶ Remark 26. Choosing $h_2 = \lceil \frac{h}{2} \rceil$ in Corollary 25 divides the augmentation into equal halves and is good enough if we do not optimize the constants, since essentially both caching and coloring components "benefit" from $\Theta(h)$ augmentation.

In Part-(4) it is simplest to think of α as a constant, in which case the requirement $k \geq f(n, \alpha)$ for the function f given in the statement, requires $k = \Omega(poly(\log n))$. However, α can also depend on k, as long as there are values of k that satisfy $k \geq f(n, \alpha(k))$. Observe that because $(\log n)^{\log n / \log \log n} = n$, for $k \leq n$ it must be that $\alpha = O(\frac{\log n}{\log \log n})$ or else the inequality cannot be satisfied. Then in particular $\alpha = o(\log n)$, and $k^{1/\tilde{\Theta}(\alpha)} \geq \log n$ (for the appropriate constants) implies $k \geq f(n, \alpha(k))$. Crudely simplified for the sake of a clean example, if $k^{\frac{1}{\alpha}} \geq \log n$ we could choose $\alpha = \frac{\log k}{2 \log \log k}$, and still have a non-empty range of applicable k values.

Finally, we improve the competitive ratio further with a larger resource augmentation.

▶ **Theorem 27.** Given $k+h$ matchings to the algorithm compared to only k matchings to the optimum, for $h \geq k - 1$, there is a deterministic algorithm that is $2(1 + \frac{k-1}{\lfloor \frac{h+3-k}{2} \rfloor})$-competitive for Caching in Matchings. In particular, with $h = (1 + \alpha)k$ extra matchings we get a competitive ratio of $O(1 + \frac{1}{\alpha})$.

The proof is in Appendix 6.2. It follows from Lemma 20 and Lemma 16.

▶ **Corollary 28.** Consider the Caching in Matchings problem where the optimum is given k matchings. There is a 6-competitive algorithm that uses $3k - 3$ matchings ($h = 2k - 3$).

A note on the running times. This work focuses on competitive analysis and therefore we do not attempt to optimize polynomial running time. We note that the algorithms in Corollary 25(1),(3) take $O(n)$ time: maintaining connection caching takes $O(k)$ time, and the coloring algorithms in Lemma 20 and Lemma 21 naively take time of $O(\Delta)$ to find a color and $O(\rho)$ to recolor a path of length ρ. The randomized algorithms of Corollary 25(2),(4) also run in polynomial time per update.

4 Related Work

Caching Problems. Caching problems have been studied in many variants and cost models. *Connection caching* is the caching variant closest to our problem. We presented it in a centralized setting, Cohen et al. [20] introduced it in a distributed setting. Albers [3] studies generalized connection caching. Bienkowski et al. [14] study connection caching in a cost model that is similar to that of caching with rejections [23]. Another related variant is *restricted caching* [16, 25] where not every page can be put into every cache slot. Buchbinder et al. [16] study the case where each page p has a subset of cache slots in which it can be cached. In our problem we also have a restriction of similar flavour, implied by the separation into matchings. We note that the cost model in [16, 25] only counts cache-misses, while we also pay for rearranging the cache.

Coloring Problems. As mentioned in Section 3, an efficient dynamic edge coloring that uses a small number of colors can be useful for competitive analysis. Subsection 3.3 covers results which we use for our advantage. The coloring literature studies the tradeoff between the number of colors, amount of recoloring (sometimes called *recourse*), and the running time of the algorithms. Some algorithms require a bound Δ on the maximum degree of the dynamic graph, while others are adaptive with respect to the maximum degree in their running time or recoloring. Literature on vertex coloring also exists, but reducing edge coloring to vertex coloring by coloring the line-graph is too wasteful in the number of colors, whether this number is parameterized by Δ, or by the arboricity of the graph as in [27]. Works on maintaining dynamically an *implicit coloring* [18, 27] cannot apply to our case because the matchings form an explicit coloring. Azar et al. [5] study dynamic vertex coloring in the context of competitive analysis.

El-Hayek et al. [22] are motivated by the same architecture as us. They solve a problem of dynamically maximizing the size of a k-edge-colorable subgraph of a dynamic graph.

Linear Programming and Convex Body Chasing in L_1. The aforementioned caching problems, like many other combinatorial problems, can be formulated as a linear program [17]. This line of research led to the development of competitive algorithms for weighted and generalized caching [1, 2, 6, 7]. A recent result of Bhattacharya et al. [11] uses linear programming with packing and covering constraints to formulate and frame the problem as convex body-chasing in L_1. They give a fractional algorithm that requires a slight resource augmentation, along with some rounding schemes to get randomized algorithms for specific problems. Our problem can be thought of as another special case of the problem considered by [11], see the appendix in the extended version [30] for this formulation and further details.

5 Conclusions and Future Work

In this paper we studied the online Caching in Matchings problem, in which we receive requests for edges in a graph and need to maintain a cache of the edges which is a union of k matchings. The problem abstracts some hardware architecture in which a datacenter is enhanced with reconfigurable optical links. Interestingly, we proved that the Caching in Matchings problem is inherently harder than the similar-looking Connection Caching problem and other caching problems. Specifically, its competitive ratio depends not only on the number of matchings k ("cache size") but also on the number of nodes in the graph. Our randomized lower bound rules out an $O(\log n)$ competitive algorithm, and the best competitive ratio we can hope for is $O(poly(\log n))$. Our lower bound for deterministic algorithms is linear in n.

We derived our algorithms by running a coloring algorithm that maintains a coloring of the cache of a connection caching algorithm. This approach is simple to describe and analyze, but inherently multiplies the competitive ratios of the two algorithms. It is natural to ask whether a "direct" algorithm for Caching in Matchings exists, and if so does it improve the competitive ratio?

Regarding resource augmentation of $h \geq 1$ extra matchings, we see that there are two interesting "discontinuities". First, immediately for $h = 1$ the competitive ratio drops to $poly(k, \log n)$, in particular "breaking" the deterministic lower bound. Second, there seems to be a point in which the competitive ratio becomes independent of n. It clearly happens for $h = k - 1$, and even sooner if k is large enough (revisit Corollary 25). These "discontinuities" beg the following two questions. First, is there an $1 \leq h < k - 1$ and an algorithm that uses h extra matchings with a competitive ratio of $O(poly(\log n))$ for any $2 \leq k < n$? Differently phrased, can we achieve a competitive ratio that is $poly(\log k, \log n)$ instead of $poly(k, \log n)$? Second, is it possible to remove the dependence on n using less than $h = k-1$ extra matchings for any k, and if so how small can h be?

A natural generalization would be to study upper bounds for Caching in Matchings in general graphs. When $k = 1$ optimality is still trivial, and when $k = n$, by Vizing's theorem, we are also optimal since we can edge-color the full n-clique with n colors. In fact, for $n \geq 2$, $k = n - 1$ colors are sufficient if and only if n is even. In the non-trivial regime $2 \leq k < n$, there exists a naive deterministic $O(n^2 k^2)$ competitive algorithm (the extended version [30] contains exlicit proofs). Resource augmentation of an extra matching ($k + 1$ colors) dramatically reduces the competitive ratio to $O(nk)$ (deterministic) and $O(n \log k)$ (randomized) by allowing us to update the coloring of the graph when a new edge is inserted according to a single step of the Misra-Gries algorithm [29],[7] or to $O(poly(k, \log n))$ as in Corollary 25(2). It is an interesting question whether the problem without augmentation is indeed that much harder in general graphs. General graphs also provide additional difficulties, such as the fact that finding minimal edge coloring for $k \geq 3$ is generally NP-complete [28].

References

1 Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *SODA*, pages 1681–1689. SIAM, 2012.

2 Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. An $O(\log k)$-competitive algorithm for generalized caching. *ACM Trans. Algorithms*, 15(1), 2018.

[7] The Misra-Gries algorithm edge-colors an uncolored graph with m edges and n nodes in m iterations. In each iteration it colors an edge and fixes the colors of previously colored edges, by recoloring $O(n)$ of them in $O(n)$ time. In our case, the graph is always fully colored up to the newly requested edge, so we get the competitive ratio of a connection caching algorithm, multiplied by $O(n)$.

3 Susanne Albers. Generalized connection caching. In *ACM SPAA*, pages 70–78, 2000.

4 Chen Avin, Chen Griner, Iosif Salem, and Stefan Schmid. An online matching model for self-adjusting ToR-to-ToR networks, 2020. `arXiv:2006.11148`.

5 Yossi Azar, Chay Machluf, Boaz Patt-Shamir, and Noam Touitou. Competitive vertex recoloring. *Algorithmica*, 85:2001–2027, 2023.

6 Nikhil Bansal, Niv Buchbinder, and Joseph (Seffi) Naor. Randomized competitive algorithms for generalized caching. In *STOC*, pages 235–244, 2008.

7 Nikhil Bansal, Niv Buchbinder, and Joseph (Seffi) Naor. A primal-dual randomized algorithm for weighted paging. *J. ACM*, 59(4), 2012.

8 Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André Renssen, Marcel Roeloffzen, and Sander Verdonschot. Dynamic graph coloring. *Algorithmica*, 81(4):1319–1341, 2019.

9 Leonid Barenboim and Tzalik Maimon. Fully dynamic graph algorithms inspired by distributed computing: Deterministic maximal matching and edge coloring in sublinear update-time. *ACM J. Exp. Algorithmics*, 24:1–24, 2019.

10 Anton Bernshteyn. A fast distributed algorithm for (Δ+1)-edge-coloring. *Journal of Combinatorial Theory, Series B*, 152:319–352, 2022.

11 S. Bhattacharya, N. Buchbinder, R. Levin, and T. Saranurak. Chasing positive bodies. In *FOCS*, pages 1694–1714, 2023.

12 Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *SODA*, pages 1–20, 2018.

13 Sayan Bhattacharya, Martín Costa, Nadav Panski, and Shay Solomon. Nibbling at long cycles: Dynamic (and static) edge coloring in optimal time. In *SODA*. SIAM, 2024.

14 Marcin Bienkowski, David Fuchssteiner, Jan Marcinkowski, and Stefan Schmid. Online dynamic b-matching: With applications to reconfigurable datacenter networks. *SIGMETRICS Perform. Eval. Rev.*, 48(3), 2021.

15 Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

16 Niv Buchbinder, Shahar Chen, and Joseph (Seffi) Naor. Competitive algorithms for restricted caching and matroid caching. In *ESA*, pages 209–221, 2014.

17 Niv Buchbinder and Joseph (Seffi) Naor. *The Design of Competitive Online Algorithms via a Primal-Dual Approach*. Now Foundations and Trends, 2009.

18 Aleksander B. G. Christiansen and Eva Rotenberg. Fully-Dynamic $\alpha + 2$ Arboricity Decompositions and Implicit Colouring. In *ICALP*, pages 42:1–42:20, 2022.

19 Aleksander Bjørn Grodt Christiansen. The power of multi-step vizing chains. In *STOC*, pages 1013–1026, 2023.

20 Edith Cohen, Haim Kaplan, and Uri Zwick. Connection caching under various models of communication. *ACM SPAA*, 2000.

21 Ran Duan, Haoqing He, and Tianyi Zhang. *Dynamic Edge Coloring with Improved Approximation*, pages 1937–1945. SIAM, 2019.

22 Antoine El-Hayek, Kathrin Hanauer, and Monika Henzinger. On b-matching and fully-dynamic maximum k-edge coloring, 2023. `arXiv:2310.01149`.

23 Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.

24 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, 1991.

25 Amos Fiat, Manor Mendel, and Steven S. Seiden. Online companion caching. In *ESA*, pages 499–511, 2002.

26 Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile reconfigurable data center interconnect. In *ACM SIGCOMM*, pages 216–229, 2016.

27 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity, 2020. arXiv:2002.10142.
28 Ian Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, 1981.
29 J. Misra and David Gries. A constructive proof of Vizing's theorem. *Information Processing Letters*, 41(3):131–133, 1992.
30 Yaniv Sadeh and Haim Kaplan. Caching connections in matchings, 2024. arXiv:2310.14058.
31 Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3):1–24, 2020.
32 Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *SFCS*, pages 222–227, 1977.
33 Neal Young. *Competitive paging and dual-guided algorithms for weighted caching and matching.* PhD thesis, Computer Science Dept., Princeton University, 1991.

6 Appendix: Deferred Proofs and Discussions

6.1 Caching in Matchings Lower Bounds (Proofs)

▶ **Lemma 11.** A lower bound of C on the competitive ratio of an online algorithm for Problem 3 implies a lower bound of C on the competitive ratio of an online algorithm for Caching in Matchings.

Proof. Let A_1 be a c-competitive algorithm for Caching in Matchings (Problem 1), with some additive term d. We show how to derive from it an algorithm A_3 that is c-competitive for Problem 3, which proves the claim. We will also use corresponding subscripts OPT_1 and OPT_3 for the optimum of each problem (with respect to a given sequence).

Given a sequence τ, Algorithm A_3 takes its decisions while processing τ by simulating A_1 on a sequence σ which is constructed as follows. We traverse τ in order, and whenever an edge is inserted, we add to σ a batch of requests which is a concatenation of $r = nk$ identical subsequences, each subsequence contains all the edges currently in G (in some arbitrary order). When an edge is deleted, we do nothing.

We now specify A_3 such that $cost(A_3(\tau))) \leq cost(A_1(\sigma))$ by having A_3 maintain its state such that it "jumps" between "check-points" in the state of A_1.

A_3 works as follows. When an edge is inserted by τ, A_3 feeds σ to A_1 until one of two things happens: either (1) the state of A_1 provides a proper coloring of G, or (2) it reaches the end of the batch that corresponds to the current edge inserted by τ. In case (1), A_3 changes its state by replaying the changes that A_1 made. Then by definition of this case, it ends up with a proper coloring of G. In case (2), we know that during the whole batch A_1 did not have a proper coloring of G, which means that in each of the r rounds it paid at least 1 for a missing edge, for a total of at least r. Rather than replaying the changes and ending up with an illegal state for A_3, we have a budget of r to completely change its state. A_3 uses half of the budget to completely empty its state and fetch all of G with some proper coloring (such coloring exists by definition of the problem). Indeed it has the budget, $|G| \leq |M| \leq \frac{nk}{2}$. The other half of its budget is used to copy the state from which A_1 continues to process σ, by flushing everything, and fetching into the cache the state of A_1. This ensures that the state of A_3 is once again identical to A_1. We showed $cost(A_3(\tau))) \leq cost(A_1(\sigma))$. It holds in the deterministic case, and also for the randomized case for every fixing of the random coins.

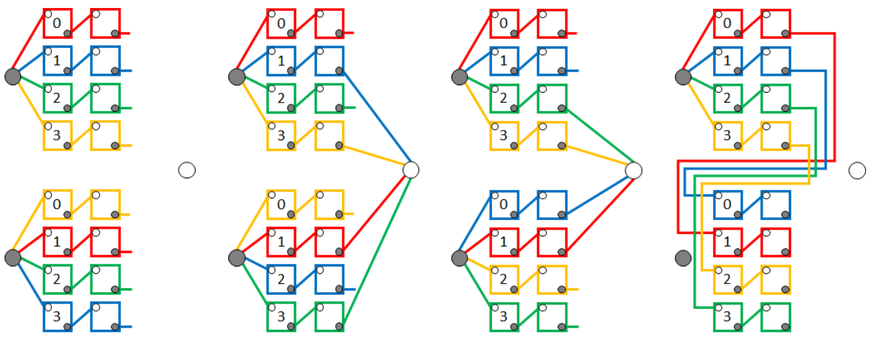
Now observe that $cost(OPT_1(\sigma)) \leq cost(OPT_3(\tau))$. Indeed, OPT_1 may simulate the behavior of OPT_3 by making changes to its own state at the beginning of each batch in σ. In conclusion, we get that $cost(A_3(\tau))) \leq cost(A_1(\sigma)) \leq c \cdot cost(OPT_1(\sigma)) + d \leq$

$c \cdot cost(OPT_3(\tau)) + d$ in the deterministic case, or similarly $\mathbb{E}[cost(A_3(\tau)))] \le \mathbb{E}[cost(A_1(\sigma))] \le c \cdot cost(OPT_1(\sigma)) + d \le c \cdot cost(OPT_3(\tau)) + d$ in the randomized case (recall that c and d were defined at the beginning of the proof). ◀

▶ **Theorem 9.** *Any* Caching in Matchings algorithm is $\Omega(\log \frac{n}{k^2 \log k} \cdot \log k)$ competitive.

Proof. This proof uses a similar high-level construction as the one used to prove Lemma 15, and the only difference is in the way we merge pairs of r-roads. Here we set $r = k$. Assume for now that when merging two k-roads of length d, OPT pays $O(k \log k)$ and any algorithm pays $\Omega(k \log k \cdot d)$ in expectation.

With this in mind, revisit the competitive analysis: when a phase begins OPT pays $O(2^h \cdot k \cdot d_0)$ to recolor bricks to their desired color, it then pays $\sum_{i=1}^{h} 2^{h-i} \cdot O(k \log k) = O(2^h \cdot k \log k)$ in the merging rounds, and finally it pays $O(2^h \cdot k \cdot d_0)$ to restore the original k-roads for the next phase. Overall, its cost is $O(2^h \cdot k \cdot (d_0 + \log k))$. In comparison, ALG pays for recoloring, in expectation, at least $\sum_{i=1}^{h} 2^{h-i} \cdot \Omega(k \log k \cdot 2^{i-1} \cdot d_0) = \Omega(h \cdot 2^h \cdot k \log k \cdot d_0)$. Assuming a sequence with $t = \Omega(k^2)$ phases, we get that the competitive ratio is $\frac{\mathbb{E}[cost(ALG)]}{cost(OPT)} = \Omega(\frac{h \cdot 2^h \cdot k \log k \cdot d_0}{2^h \cdot k \cdot (d_0 + \log k)}) = \Omega(\frac{h \cdot \log k \cdot d_0}{d_0 + \log k})$. To maximize the expression we balance and choose $d_0 = \lceil \log k \rceil$, getting $\Omega(h \log k)$. We determine h as before, except that the complicated merging technique requires a reusable extra node, so we have that $h \ge \lfloor \log \frac{n-1}{4k^2 \cdot d_0 + 1} \rfloor$. Simplified, and with $d_0 = \lceil \log k \rceil$, we get $h = \Omega(\log \frac{n}{k^2 \log k})$, therefore the competitive ratio is $\Omega(\log \frac{n}{k^2 \log k} \cdot \log k)$.



(a) Initial state. **(b)** Step 1. **(c)** Step 2. **(d)** Final merge.

Figure 6 Visualization of merging a pair of k-roads, for $k = 4$, of length $d = 2$ to a single twice longer k-road, by "negative information". (a) The roads are numbered from 0 to 3, with their number written inside their first brick. There are $\log k = 2$ steps. (b) In the first step we temporarily connect roads $\{1, 3\}$ (least significant bit 1) of the top k-road with either roads $\{0, 2\}$ or $\{1, 3\}$ of the bottom k-road, to a shared hub (the white node). In this example we connect $\{1, 3\}$, and as a results roads 2 and 3 of the bottom k-road were recolored. (c) In the second step we temporarily connect roads $\{2, 3\}$ (second bit is 1) of the top k-road with either roads $\{0, 1\}$ or $\{2, 3\}$ of the bottom k-road to a shared hub. In this example we connect $\{0, 1\}$, and as a result roads 0 and 2 of the bottom k-road were recolored. (d) Finally there is a round of "positive information" in which we simply extend each road on the top color-consistently with a road on the bottom. The consistency depends on the choices of the previous steps, and in this example it matches road x on the top with road $x \oplus 1$ on the bottom. The recoloring in this example is such that in the final extension no road is recolored.

Now we explain and analyze the merging of two k-roads, denote them by X and Y. See Figure 6 for a visual example with $k = 4$. For simplicity, let us start with k being an integer power of 2, say $k = 2^\ell$. We start with ℓ steps of "negative information" in which we reveal roads that are of *different* colors, and do so by connecting the free end of these roads to a new shared hub, denote it by v. Concretely, number the roads of X from 0 to $k-1=2^\ell-1$ and denote by $X_{i,b}$ the roads of X whose ith bit is b. We define similarly the subsets of roads for Y. In round i, we connect to v the roads $X_{i,1}$ and Y_{i,b_i} where b_i is chosen uniformly at random between 0 and 1. Note that $|X_{i,1}|=|Y_{i,b_i}|=\frac{k}{2}$ so the degree of v is k (legal). When the round ends, we delete the edges of v. Finally, in the $\ell+1$ step we produce the longer k-road with "positive information" by cutting the roads of Y from their hub and extending the roads of X, according to the unique way which does not contradict the previous ℓ steps. This way exists: road $y \in Y$ extends the road x whose binary representation is $x=y\oplus B$ where the bits of B are b_i and $\oplus$ is the bitwise exclusive-or operation.

Let us analyze the costs of OPT and ALG. Since OPT knows the correct colors in advance, it can pay at most 1 per edge that is inserted. We insert k edges per step (even if most of them are later deleted), in a total of $\ell+1$ steps. This totals to $O(k\log k)$. We argue that ALG recolors in expectation $\Omega(k)$ roads per each of the first ℓ steps. To simplify the analysis, assume that ALG recolors after learning b_i rather than being introduced online each edge of v one by one (which can only hurt ALG). Then indeed every road in $X_{i,1}$ has probability $\frac{1}{2}$ to be in a conflict of color with a road of Y_{i,b_i}, so by the linearity of expectation, we get at least $\frac{k}{2}=\Omega(k)$ road recolorings (if ALG is "reasonable", it recolors $O(k)$ roads per step, so allowing it to be semi-offline did not lose more than a constant factor). Observe that our bound for round i is not affected by previous rounds. So we conclude that ALG pays $\Omega(\ell\cdot k\cdot d)=\Omega(k\log k\cdot d)$ for recoloring in expectation.

The case of k not being a power of 2 is similar. Each road is still assigned a number, and we regard its binary representation with $\lceil\log k\rceil$ bits, but only make $\ell=\lfloor\log k\rfloor$ rounds. Note that now $X_{i,1}$ is not necessarily of size $\frac{k}{2}$, but rather might be smaller. The bias is always in favor of 0 because of how counting works, and it is such that $|X_{i,1}|\geq\frac{k-2^i}{2}$ ($i=0$ is the least significant bit). So we can still choose Y_{i,b_i} with b_i uniformly random, there is no problem to connect all the roads of $X_{i,1}$ and Y_{i,b_i} to their shared hub. Also, each road in $X_{i,1}$ still has a color conflict with probability $\frac{1}{2}$. The only thing that changes is that the expectation of road recolorings is not $\frac{k}{2}$ per round, but rather $|X_{i,1}|$ in round i. This yields at least $\sum_{i=0}^{\ell-1}|X_{i,1}|\geq\ell\cdot\frac{k}{2}-\frac{1}{2}\sum_{i=0}^{\ell-1}2^i>\ell\cdot\frac{k}{2}-\frac{2^\ell}{2}>\frac{(\ell-1)k}{2}$ road recolorings in expectation for ALG, which is still $\Omega(k\log k)$ in total. The analysis of OPT is unchanged, and its total cost is $O(k\log k)$ in total per merging a pair of k-roads (of any length). ◀

▶ Remark 29. A few notes on the proofs of Lemma 15 and Theorem 9:

1. The random choices of the adversary can be boiled-down to the random order of extending roads (in Lemma 15) and the bits b_i (in Theorem 9). The 2-roads and k-roads themselves are chosen once, and even the pairings of each merging round may be fixed.
2. For clarity, we presented it as if we need 2^h different hubs, one per r-road. In fact, we only need $h+1$ hubs if we reuse them: h of them to maintain an r-road for each unique length, and another one for the length in which we currently merge a pair of r-roads. This saving is negligible compared to the number of nodes used to compose the roads.

6.2 Caching in Matchings With Resource Augmentation (Proofs)

In this section we restate and prove the claims from Section 3.1 that we did not prove there.

▶ **Lemma 21.** Let G be a dynamic *bipartite* graph with the guarantee that its maximum degree is at most k at any time, and let $h\geq 1$. We can maintain a deterministic $(k+h)$-edge-coloring of G in amortized $O(\sqrt{nk/h})$ recolorings per insertion.

Proof. Recall the proof of Theorem 7, we will use the same idea of a color-swap on a bi-colored path. The key difference is how we use the $h \geq 1$ extra colors.

First consider $h = 1$ and denote the extra color as yellow. We allow at most y yellow edges in the graph, and if we need more, we recolor the whole graph from scratch without using yellow. Such a recoloring is possible because the graph is bipartite and every node is of degree at most k. When coloring a newly inserted edge (u, v), both u and v have at least one free color. We have three cases:

1. If u and v share a free color, including yellow: Then use this color.
2. If u does not have a yellow edge and v does (the other case is symmetric): Let c be a free color of v, and apply a color-swap of c and yellow with respect to v. This makes yellow a free color of v. Note that u is unaffected by the color-swap, because the graph is bipartite (affecting u implies that the path of the swap closes an odd cycle with the edge (u, v)). Now color (u, v) in yellow.
3. If both u and v have a yellow edge: Let c be a free color of u. Apply a color-swap of c and yellow with respect to u to make yellow a free color of u. Now apply the previous case.

We apply up to two color-swaps, each of length $O(y)$ because there are at most y yellow edges in the whole graph. Recall that we might have a global recoloring once we reach y yellow edges. We charge these recolorings to the yellow edges. Formally, we define a potential for the cache which equals $\frac{nk}{y} \cdot i$ when there are i yellow edges. Thus when we accumulate y yellow edges, the potential can pay for the global recoloring. Each insertion of an edge causes $O(y)$ recoloring and increases the potential by at most $\frac{nk}{y}$, due to possibly inserting a yellow edge (our color-swaps never increase the number of yellow edges). We conclude that the amortized cost is $O(y + \frac{nk}{y})$ per insertion. Balancing with $y = \sqrt{nk}$ gives $O(\sqrt{nk})$.

We generalize the previous logic for $h \geq 1$ by allowing each extra color to have at most y edges, and when it fills up we proceed to use the next extra color. Only when all h colors have y edges we invoke a full recoloring. The potential in this case is $\frac{nk}{hy}$ per edge, and the amortized cost is therefore $O(y + \frac{nk}{hy})$. Balancing with $y = \sqrt{nk/h}$ gives $O(\sqrt{nk/h})$. ◀

▶ **Theorem 27.** Given $k+h$ matchings to the algorithm compared to only k matchings to the optimum, for $h \geq k - 1$, there is a deterministic algorithm that is $2(1 + \frac{k-1}{\lfloor \frac{h+3-k}{2} \rfloor})$-competitive for Caching in Matchings. In particular, with $h = (1 + \alpha)k$ extra matchings we get a competitive ratio of $O(1 + \frac{1}{\alpha})$.

Proof. Let σ be a sequence of requests. Denote an algorithm A with cache parameter x as A^x, and use subscripts m for Caching in Matchings and c for Connection Caching. We have $cost(A_m^{2r-1}(\sigma)) = cost(A_c^r(\sigma))$ for any integer $r \geq 1$ by considering $r-1$ matchings as resource augmentation, such that we require no recoloring (by Lemma 20). Since this reduction halves the cache parameter, and our algorithm initially has cache of size $k + h$, we use $r = \lfloor \frac{k+h+1}{2} \rfloor$. If $k + h = 2r$, we do not use one of the colors, on purpose, to ensure using exactly $2r - 1$ colors. Taking A_c to be the algorithm that satisfies Lemma 16, $cost(A_c^r(\sigma)) \leq \frac{2r}{r-k+1} \cdot cost(OPT_c^k(\sigma)) + d$ for some fixed term d. By Remark 4, $cost(OPT_c^k(\sigma)) \leq cost(OPT_m^k(\sigma))$. Plugging everything together we get that $cost(A_m^{2r-1}(\sigma)) \leq \frac{2r}{r-k+1} \cdot cost(OPT_m^k(\sigma)) + d$, hence A_m^{2r-1} is $\frac{2r}{r-k+1} = 2(1 + \frac{k-1}{r-k+1}) = 2(1 + \frac{k-1}{\lfloor \frac{h+3-k}{2} \rfloor})$ competitive for Caching in Matchings. ◀