General Caching with Lifetimes

Ashish Chiplunkar, Sundar Vishwanathan
{ashishc, sundar}@cse.iitb.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
December 22, 2011

Abstract

We consider the problem of caching with lifetimes, where a lifetime is specified whenever a page is loaded into the cache. The copy of a page loaded into the cache may be used to serve requests to the same page, only until its expiration time. We present a generic method to get an algorithm for caching with lifetimes, from an algorithm for caching without lifetimes. This method works for any cost model, and in online as well as offline settings. In the online (resp. offline) setting, the competitive (resp. approximation) ratio of resulting algorithm for caching with lifetimes, is one more than the competitive (resp. approximation) ratio of the original algorithm for caching without lifetimes. Using this method and the existing algorithms for caching without lifetimes, we get an $H_k + 1$ competitive randomized algorithm and a 2-approximation algorithm for standard caching with lifetimes, where $k$ is the cache size. This is an improvement over the $(2H_k + 3)$-competitive algorithm and the 3-approximation algorithm given by Gopalan et. al. [GKM+02]. We also get $O(\log k)$ competitive randomized algorithms for various subclasses of general caching such as weighted caching and the Bit and Fault models, asymptotically matching the lower bound of $H_k$ on the competitive ratio; and a 5 approximation algorithm for the general offline caching problem.

1 Introduction

In the standard caching problem, a sequence of requests to pages from the slow memory is presented to an algorithm, and the algorithm is required to serve them. The algorithm has a fast memory (cache) which can hold at most $k$ pages. To serve a request to a page, the algorithm must ensure that the page is present in its cache. If the requested page is not present in the cache, the algorithm has to load it from the slow memory, and possibly evict some other page from the cache. The algorithm has to pay a cost of one unit in such a situation, called a “cache miss” or a “page fault”. Various page eviction policies have been designed for this problem and analyzed [Bel66, ST85, FKL+91, MS91, ACN00]. The general caching problem is a variant of the standard caching problem, in which the sizes of the pages and the costs of loading them into the cache could be arbitrary. This generalization has practical significance, for instance in file caching, where different files could have different sizes, and the retrieval cost could depend on the server from where the file is retrieved. The general caching problem without lifetimes has been studied well in the past [CKPV90, MMS90, You94, You98, BBN07, BBN08, BBN10]. Various subclasses of this problem have been defined by putting various kinds of restrictions on the sizes and the costs of the pages.
The problem of caching with lifetimes has been introduced to model the scenario, where the contents of a page in the main memory could be modified by some other agents, thereby making the copy of that page in the cache inconsistent with that in the slow memory. This situation arises in applications such as the caches used in distributed systems, and in the Internet. In such applications a page comes with a “time to live” and once this time expires, the page must be discarded from the cache. In the problem of caching with lifetimes, whenever the algorithm loads the copy of a page into the cache, a lifetime for that copy is specified. This lifetime is essentially a promise that the copy of that page in the slow memory will not be modified for that long, but could be modified as soon as the lifetime is over. To serve a request, the algorithm is required to load the requested page into the cache if it is not present, or if the lifetime, of the copy of that page in the cache, is over.

The problem of caching with lifetimes has been studied only when the sizes and costs of pages are 1 [GKM+02], and only in the deterministic online setting [Kim01]. The existing algorithms for caching with lifetimes have been designed by making small modifications to the algorithms for caching without lifetimes. It is therefore natural to ask, whether an algorithm for an arbitrary subclass of general caching without lifetimes, can be easily modified, to work for the analogous subclass of the general caching problem with lifetimes. We answer this question affirmatively, and present a generic method to modify an algorithm for caching without lifetimes, to obtain an algorithm for caching with lifetimes.

The performance ratio of an algorithm is said to be \( \alpha \) if, for every input \( \rho \), the cost incurred by the algorithm to serve \( \rho \) is at most \( \alpha \) times the cost of the optimal way to serve \( \rho \). In case of randomized algorithms, the for the performance ratio to be \( \alpha \), we require that the expectation, over the runs of the algorithm, of the cost incurred by the algorithm must be at most \( \alpha \) times the optimal cost. Note that the performance ratio of an algorithm is at least 1 and an algorithm is better, if its performance ratio is closer to 1. We prove that our method results in an algorithm for caching with lifetimes, whose performance ratio is only a little worse than the performance ratio of the original algorithm for caching without lifetimes.

**Related Work: Caching without lifetimes**

For the standard caching problem without lifetimes, where page sizes and costs are 1, Belady [Bel66] proved that the algorithm which, in order to accommodate the requested page into the cache, evicts the page requested farthest in future, is optimal. However, this algorithm works when the entire request sequence is known in advance. Such algorithms are called “offline” algorithms. On the other hand, when an algorithm serves a request without the knowledge of the future requests, we say that the algorithm is “online”. Sleator and Tarjan [ST85] showed that the least-recently-used (LRU) and the marking algorithms have performance ratio of \( k \). They also showed that no deterministic online algorithm can have a better performance ratio. Following them, the performance ratio is usually referred to as “competitive ratio” in case of online algorithms.

Fiat et. al. [FKL+91] proposed the randomized marking algorithm (RMA) and showed that the competitive ratio of this algorithm is at most \( 2H_k \), where \( H_k = \sum_{i=1}^{k} 1/i \approx \ln k \). They also showed that even with randomization, no online algorithm can have competitive ratio less than \( H_k \). McGeoch and Sleator [MS91] and Achlioptas et. al. [ACN00] later gave a randomized algorithm with competitive ratio \( H_k \). Achlioptas et. al. [ACN00] also proved that the competitive ratio of RMA is exactly \( 2H_k - 1 \).

The general caching problem without lifetimes is a variant, in which the page sizes and costs
can be arbitrary. (We will refer to the original caching problem as “classical” caching.) Note that the lower bounds for the competitive ratio for classical caching, also hold for general caching. The subclass of general caching, in which only the page costs are arbitrary and the pages have unit size, is called “weighted” caching. Chrobak et al. [CKPV90] gave the $k$-competitive “Balance” algorithm and also an exact offline algorithm for weighted caching. Young [You94] developed another $k$-competitive algorithm called “Greedy Dual” which is, in some sense, a primal dual algorithm. Later he [You98] gave the “Landlord” algorithm which is $k$-competitive for general caching.

The “Bit” model, where the cost of any page is equal to its size, and the “Fault” model, where the cost of each page is 1 but its size is arbitrary, are two other subclasses of the general caching problem that are well studied. Bansal et. al. [BBN07, BBN08, BBN10] used the technique which they called “online primal-dual” to design randomized online algorithms for these models. By this technique, they obtained $O(\log k)$ competitive algorithms for weighted caching [BBN07, BBN10], and for the bit and fault models [BBN08]. They also obtained an $O(\log^2 k)$ competitive randomized algorithm for general caching. In the offline setting, Chrobak et. al. [CWMX11] proved that the problem is NP-hard in the Bit as well as the Fault model. Bar-Noy et. al. [BNBYF+01] formulated a resource allocation and scheduling problem, and gave a 4-approximation algorithm for the same. They showed that an instance of general caching problem can be seen as an instance of the problem they defined, thereby giving a 4-approximation algorithm for offline general caching. The above results are summarized in Table 1.

<table>
<thead>
<tr>
<th>Cost Model</th>
<th>Online</th>
<th>Offline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>$k$: [ST85]</td>
<td>$2H_k$: [FKL+91]</td>
</tr>
<tr>
<td>Weight</td>
<td>$k$: [CKPV90, MMS90, You94]</td>
<td>$O(\log k)$: [BBN07, BBN10]</td>
</tr>
<tr>
<td>Bit</td>
<td>$k$: [CKPV90, MMS90, You94]</td>
<td>$O(\log k)$: [BBN08]</td>
</tr>
<tr>
<td>Fault</td>
<td>$k$: [You98]</td>
<td>$O(\log k)$: [BBN08]</td>
</tr>
</tbody>
</table>

Table 1: A summary of performance ratios of known algorithms for various subclasses of the caching problem without lifetimes.

**Related work: Caching with lifetimes**

Kimbrel [Kim01] and Gopalan et. al. [GKM+02] considered the problem of caching with lifetimes and gave a $k$-competitive deterministic algorithm for classical caching. The LRULU algorithm by Kimbrel [Kim01] has been derived from the LRU algorithm whereas the algorithm by Gopalan et. al. [GKM+02] has been derived from the marking algorithm. Kimbrel [Kim01] has also given the modified version of the Landlord algorithm from [You98] for general caching with lifetimes. Gopalan et. al. have given a randomized $(2H_k+3)$-competitive algorithm, which is a modified version of RMA, for classical caching and a 3-approximation algorithm, which is a modified version of the farthest-in-future policy, for offline classical caching. We must note that the lower bounds proved for various settings for caching without lifetimes, carry over to the analogous settings of caching with lifetimes also.

**Our Contribution**
In this paper, we present a generic method to modify any algorithm for any subclass of the general caching problem without lifetimes, to get an algorithm for the analogous subclass of the general caching problem with lifetimes. The resulting algorithm is deterministic if the original algorithm is deterministic; the former is online if the latter is online. Our method thus gives a better \((H_k + 1)\)-competitive randomized online algorithm and a 2-approximation offline algorithm for classical caching with lifetimes. These are better than the corresponding algorithms from [GKM+02]. In fact, we obtain a 2-approximation algorithm for offline weighted caching. We also obtain new online randomized algorithms with competitive ratio \(O(\log k)\) for weighted caching, and the Bit and Fault models; and with competitive ratio \(O(\log^2 k)\) for general caching with lifetimes. Further, we have a 5-approximation algorithm for offline general caching with lifetimes. These results, along with the earlier results for caching with lifetimes, are summarized in Table 2.

<table>
<thead>
<tr>
<th>Cost Model</th>
<th>Online</th>
<th>Offline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>(k: [\text{Kim01}, \text{GKM}^+02])</td>
<td>(2H_k + 3: [\text{GKM}^+02])</td>
</tr>
<tr>
<td>Weight</td>
<td>(O(\log k): \text{our work})</td>
<td>(2: \text{our work})</td>
</tr>
<tr>
<td>Bit</td>
<td>(O(\log k): \text{our work})</td>
<td></td>
</tr>
<tr>
<td>Fault</td>
<td>(O(\log k): \text{our work})</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>(k: [\text{Kim01}])</td>
<td>(O(\log^2 k): \text{our work})</td>
</tr>
</tbody>
</table>

Table 2: A summary of performance ratios of known algorithms for various subclasses of the caching problem with lifetimes.

## 2 Preliminaries

Let \([n] = \{1, \ldots, n\}\) be the set of pages in the slow memory, \(k\) be the size of cache and \(m\) be the number of requests. Let \(w(p)\) be the size of page \(p\) and let \(c(p)\) be the cost of loading page \(p\) into the cache, where \(p \in [n]\). A cache state is a subset \(s\) of \(\{1, \ldots, m\}\) such that if \(i_1, i_2 \in s\) then \(p_{i_1} \neq p_{i_2}\), and \(\sum_{i \in s} w(p_i) \leq k\), where \(p_i\) is the page requested in the \(i\)'th request. Note that we have defined the cache state to be a subset of indices from the request sequence, and ensured that there is at most one copy of a page in the cache. Observe that this assumption is without loss of generality, since even with lifetimes, an algorithm can evict an old copy of a page at no cost, once it has loaded a more recent copy of the same page.

An instance of caching without lifetimes consists of the sequence \(p_1, p_2, \ldots, p_m\) of pages as input, where \(p_i \in [n]\). A sequence \(s_0 = \emptyset, s_1, \ldots, s_m\) of cache states is said to be feasible for the given input sequence, if for each \(i = 1, \ldots, m\), one of the following two conditions holds.

- \(s_i \subseteq s_{i-1}\) and there exists \(i' \in s_i\) such that \(p_i = p_{i'}\).
- \(i \in s_i\) and \(s_i \setminus \{i\} \subseteq s_{i-1}\).

The first condition holds if the algorithm has loaded a copy of \(p_i\) some time in the past, and retained it till the \(i\)'th request. The second condition holds if the algorithm loads \(p_i\) to serve the request. In both cases, algorithm can possibly evict some of the pages from the cache while
serving the request. The fact that $s_i$ is a cache state, ensures that the total size of the pages in cache is at most $k$. The algorithm incurs a cost of $w(p_i)$ only in the second case. The total cost incurred is thus given by $\sum_{i: i \in s_i} c(p_i)$. As an output, we want a feasible sequence of cache states which has minimum cost.

An instance of caching with lifetimes consists of the sequence $r_1, r_2, \ldots, r_m$ of requests, where the $i$'th request $r_i$ is the pair $(p_i, l_i)$. Here $p_i \in [n]$ is a page and $l_i \in \mathbb{Z}_+$ is the lifetime of the current copy of the page. This means that if the algorithm loads $p_i$ to serve this request then it can use the current copy to serve future requests to $p_i$ that occur within $l_i$ time units. As output, we again want a minimum cost feasible sequence $s_0 = \emptyset, s_1, \ldots, s_m$ of cache states, where we call the sequence feasible if for each $i = 1, \ldots, m$, one of the following two conditions holds.

- $s_i \subseteq s_{i-1}$ and there exists $i' \in s_i$ such that $p_i = p_{i'}$ and $l_i + l_{i'} \geq i$.
- $i \in s_i$ and $s_i \setminus \{i\} \subseteq s_{i-1}$.

In this case, the first condition holds if there is already a live copy of page $p_i$ in the cache. We say that a copy of a page is “live” if its lifetime is not yet over; otherwise we say that the copy is “dead”. As before, the cost for this sequence is $\sum_{i: i \in s_i} c(p_i)$.

We say that the lifetimes in the request sequence are monotone if for each $i_1 < i_2 \in [m]$, $p_{i_1} = p_{i_2}$ implies $i_1 + l_{i_1} \leq i_2 + l_{i_2}$. In words, for two copies of the same page, the copy which is loaded earlier dies no later than the copy which is loaded later. We will assume that the lifetimes in the input are monotone.

Note that with any instance $\mathcal{P}$ of the caching problem with lifetimes, we can associate the instance $\Pi(\mathcal{P})$ of the caching problem without lifetimes, obtained by removing the lifetimes from the input (or in other words, setting them to $\infty$). Thus $\Pi(\mathcal{P})$ is in some sense a “relaxation” of $\mathcal{P}$. Let $\text{OPT}(\mathcal{P})$ denote the cost of the optimum solution for the instance $\mathcal{P}$. Then it is easy to see that for an instance $\mathcal{P}$ of the caching problem with lifetimes,

$$\text{OPT}(\mathcal{P}) \geq \text{OPT}(\Pi(\mathcal{P}))$$

(1)

For an instance $\mathcal{Q}$ of the caching problem without lifetimes, let $\Pi^{-1}(\mathcal{Q}) = \{\mathcal{P} \mid \Pi(\mathcal{P}) = \mathcal{Q}, \text{and the lifetimes in } \mathcal{P} \text{ are monotone}\}$.

In the offline setting of the caching problem, the entire input is given to the algorithm before it starts running. On the other hand, in the online setting, only $[n]$, $k$, $w$, and $c$ are given as input initially. The page $p_i$ and the lifetime $l_i$ are given at time $t$, and the algorithm is required to output $s_i$ right away, without the knowing the future input.

### 3 The generic modification

For a cache state $s$, let $q(s)$ denote the set of pages in the cache, when its state is $s$. Thus, $q(s) = \{p_i \mid i \in s\}$. Let $\mathcal{U}$ be a set of instances of the caching problem without lifetimes and $\mathcal{U}' = \{\mathcal{P} \mid \Pi(\mathcal{P}) \in \mathcal{U}\}$. We overload notation and say $\mathcal{U} = \Pi(\mathcal{U}')$ and $\mathcal{U}' = \Pi^{-1}(\mathcal{U})$. Suppose $\mathcal{A}$ is an algorithm for $\mathcal{U}$. Consider the algorithm $\mathcal{A}'$ (Algorithm 1) for $\mathcal{U}'$ which uses $\mathcal{A}$ as a subroutine. In the description of the algorithm, $s_i$ is the cache state returned by $\mathcal{A}$ in response to the $i$'th request $p_i$, and $s'_i$ is the cache state returned by $\mathcal{A}'$ in response to the request $(p_i, l_i)$.

Algorithm $\mathcal{A}'$ essentially mimics $\mathcal{A}$ and maintains the same set of pages in the cache. However, the cache of $\mathcal{A}'$ could have more recent copies of pages than the cache of $\mathcal{A}$. Whenever $\mathcal{A}$
Algorithm 1 $A'$: modified version of $A$ for caching with lifetimes

\begin{align*}
\text{s}_0, \text{s}'_0 & := \emptyset \\
\text{for } t = 1 \text{ to } m \text{ do} \\
& \{ \text{Invariant: } q(s_{i-1}) = q(s'_{i-1}) \} \\
& \text{Read } (p_t, l_t). \\
& \text{Run } A \text{ to obtain } s_t. \\
& \text{if } A \text{ loaded } p_t \text{ at the current request then} \\
& \quad s'_t := \{ t' \mid t' \in s'_{i-1} \text{ and } p_{t'} \in q(s_t) \} \cup \{ t \} \{ \text{the “replace” step} \} \\
& \text{else if the copy } t' \in s'_{i-1} \text{ of page } p_t \text{ is dead then} \\
& \quad s'_t := s'_{i-1} \setminus \{ t' \} \cup \{ t \} \{ \text{the “refresh” step} \} \\
& \text{else} \\
& \quad s'_t := s'_{i-1} \\
& \text{end if} \\
& \text{end for}
\end{align*}

evicts the copies of some pages from its cache and loads the current copy, $A'$ evicts the (possibly more recent) copies of the same pages from its cache and loads the current copy. If $A$ has a copy of the currently requested page $p_t$ in its cache (and decides to use it), $A'$ checks whether the copy of $p_t$ in its cache is live. If not, $A'$ evicts the dead copy and loads the current copy $t$.

The following observations are immediate from the description of Algorithm 1.

1. The invariant $q(s_{i-1}) = q(s'_{i-1})$ is maintained and the algorithm always returns a feasible solution.

2. $A$ in online (resp. deterministic, polynomial time), then $A'$ is online (resp. deterministic, polynomial time).

Theorem 2 asserts that the performance ratio of $A'$ is only a little worse than the performance ratio of $A$. To prove the theorem, we require Lemma 1, which gives a lower bound on the number of times any algorithm must fault on a page.

**Lemma 1** (Independent Set Bound). For every page $p \in [n]$, any feasible output must load page $p$ at least as many times, as the size of the largest independent set of requests to page $p$.

**Proof.** Suppose there is a feasible solution in which the number of times page $p$ is loaded, is less than the size of the largest independent set of requests to $p$. Then there must be a copy of $p$ which was used to serve two requests – $r_{t_1}$ and $r_{t_2}$ in the independent set, where $t_1 < t_2$. Suppose that copy is loaded at time $t$ (to serve $r_t$). This copy must be live till $t_2$ and hence $t \leq t_1 < t_2 \leq t + l_t$. Due to the monotonicity assumption about the lifetimes we have $t + l_t \leq t_1 + l_{t_1}$. But then $t_2 \leq t_1 + l_{t_1}$. Thus the intervals $[t_1, t_1 + l_{t_1}]$ and $[t_2, t_2 + l_{t_2}]$ overlap, and hence $r_{t_1}$ and $r_{t_2}$ cannot both be in an independent set.

**Theorem 2.** If $A$ is an $\alpha$-competitive (resp. $\alpha$-approximation) algorithm for $U$ then $A'$, as defined by Algorithm 1 is an $(\alpha + 1)$-competitive (resp. $(\alpha + 1)$-approximation) algorithm for $U' = \Pi^{-1}(U)$.
Proof. Fix an instance $\mathcal{P}' \in \mathcal{U}'$ defined by the sequence $\rho = (r_1, \ldots, r_m)$ of requests, where $r_i = (p_i, l_i)$. We divide the (expected) cost incurred by $\mathcal{A}'$ into two parts - $C_1$, the (expected) cost due to the “replace” step and $C_2$, the (expected) cost due to the “refresh” step. Thus, $C_1$ is equal to the (expected) cost incurred by $\mathcal{A}$ on the instance $\mathcal{P} = \Pi(\mathcal{P}') \in \mathcal{U}$ defined by the sequence $p_1, \ldots, p_m$ of pages. Due to the fact that $\mathcal{A}$ is an $\alpha$-competitive / $\alpha$-approximation algorithm, and inequality (1) we have

$$C_1 \leq \alpha \cdot \text{OPT}(\mathcal{P}) \leq \alpha \cdot \text{OPT}(\mathcal{P}')$$  \hspace{1cm} (2)$$

To bound $C_2$, fix a page $p \in [n]$ and consider the set of requests $R_p$ to $p$, on which $\mathcal{A}$ executed the “refresh” step. We claim that $R_p$ must be an independent set (with probability 1). Suppose not; let $r_1, r_2 \in R_p$, $t_1 < t_2$ be such that $\mathcal{I}_{t_1} = [t_1, t_1 + l_{t_1}]$ and $\mathcal{I}_{t_2} = [t_2, t_2 + l_{t_2}]$ intersect. Then $t_2 \leq t_1 + l_{t_1}$ i.e. the copy of $p$ loaded at time $t_1$ stays live till $t_2$. If the algorithm retains this copy till time $t_2$, then it will not refresh to serve $r_{t_2}$. So suppose the algorithm evicts the copy loaded at time $t_1$, before time $t_2$. Since the algorithm executed the refresh step at $t_2$, there must be a dead copy of $p$ in the cache at that time. Suppose that dead copy was loaded at time $t$. Then $t + l_t < t_2$. Now, since the algorithm maintains at most one copy of any page in the cache at any time, $t > t_1$. But now we have $t + l_t < t_2 \leq t_1 + l_{t_1}$. This contradicts the assumption that lifetimes are monotone.

Since $R_p$ is an independent set for every $p$, any feasible solution must fault at least $|R_p|$ times on requests to page $p$, due to Lemma 1. Thus (with probability 1) we have $\text{OPT}(\mathcal{P}') \geq \sum_{p \in [n]} c(p)|R_p|$. But $C_2$ is equal to (the expected value, over the runs of $\mathcal{A}'$, of) $\sum_{p \in [n]} c(p)|R_p|$. Therefore

$$C_2 \leq \text{OPT}(\mathcal{P}')$$ \hspace{1cm} (3)$$

Finally, the (expected) cost incurred by $\mathcal{A}'$ is $C_1 + C_2 \leq (\alpha + 1) \text{OPT}(\mathcal{P}')$, due to inequalities (2) and (3). \hfill \Box

4 New results and improvements

Following are the new algorithms, and the algorithms improving the older ones, that result by using various algorithms for caching without lifetimes, as subroutine $\mathcal{A}$ in Algorithm 1.

1. Classical caching; randomized online setting: Using the $H_k$ competitive algorithms due to [MS91, ACN00] as $\mathcal{A}$, $\mathcal{A}'$ becomes an $(H_k + 1)$-competitive randomized algorithm for classical caching with lifetimes. This improves the $(2H_k + 3)$-competitive randomized algorithm due to [GKM+02].

2. Weighted caching and the Bit and Fault models; randomized online setting: Using the $O(\log k)$ competitive online-primal-dual algorithms from [BBN07, BBN08, BBN10], we have an $O(\log k)$-competitive randomized algorithm for weighted caching, and the Bit and Fault models with lifetimes. This asymptotically matches the lower bound of $H_k$ in all cases.

3. General caching; randomized online setting: Using the $O(\log^2 k)$ competitive algorithm from [BBN08], we have an $O(\log^2 k)$-competitive randomized algorithm for general caching with lifetimes. (In fact, we have an $O(\log k)$-competitive algorithm for general caching with lifetimes, asymptotically matching the lower bound of $H_k$, if and only if there is an $O(\log k)$-competitive algorithm for general caching without lifetimes.)
4. **Weighted caching; offline setting:** Using the exact algorithm for weighted caching form [CKPV90], we get a 2-approximation algorithm for the weighted caching (and hence classical caching) with lifetimes. This improves the 3-approximation algorithm due to [GKM+02], for the classical caching.

5. **General caching; offline setting:** Using the 4-approximation algorithm given by [BBNBYF+01], we have a 5-approximation algorithm for the general caching (and hence for the Bit, Fault models) with lifetimes.

5 Concluding remarks

We have presented a general method to modify an algorithm for any variant of caching problem without lifetimes, to get an algorithm for the same variant, in the presence of lifetimes. Our method introduces only an additive 1 in the performance ratio of the original algorithm. With this, we have online algorithms with competitive ratios asymptotically matching the lower bounds, in almost all subclasses of caching with lifetimes. In case of general caching with lifetimes, we will have an \( O(\log k) \) competitive randomized online algorithm, as soon as we find such an algorithm for general caching without lifetimes. For the offline problems, we do not know whether weighted caching with lifetimes is NP-hard. Similarly, we do not know whether we can have a better approximation algorithm for general caching with lifetimes, but we will have one as soon as we find a better approximation algorithm for general caching without lifetimes.

References


