Adaptive Caching with Follow The Perturbed Leader Replacement Policy

Mikkel Mäkelä
Supervisor(s): Georgios Iosifidis, Tareq Si Salem
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Caching is a widely relevant problem in the world of ever-growing online traffic. In recent years, online learning methods have inspired algorithms that outperform more traditional, widely used policies such as LRU and LFU. Furthermore, some of these newly proposed policies have proven upper regret bounds, offering guarantees on arbitrary request sequences. In this paper, we inspect one such policy named Follow the Perturbed Leader (FTPL), which we found to perform reasonably well among different traces but showcases poor ability to adapt to changing file popularities. We propose a modification to the algorithm which addresses this issue, and although it comes with no performance guarantees, we leverage the expert problem to maintain such a guarantee. More precisely, we utilize different configurations of FTPL, including the one with a tight upper bound, as experts in an Incrementally Adaptive Weighted Majority (IAWM) algorithm. Our findings include simulation results on the MovieLens trace, as well as multiple synthetic traces, some with adversarial properties. We record these in a setting where a single user is connected to a single cache, which is then connected to a server, but also in a bipartite network where multiple clients execute traces against multiple caches.

1 Introduction

Since its inception, internet traffic has been growing exponentially year-over-year [1]. Increasing network loads have adverse impacts on both server and client side parties, resulting in elevated costs and higher latencies. Caching is a widely adopted mechanism to alleviate some of these issues by placing more frequently requested content closer to the end users, thereby reducing overall network load as well as delivering a better experience.

Traditional caching techniques emerged from local computing problems, where a subset of the data is placed in smaller, faster memory to reduce average access times. Some of the most commonly used recency and frequency-based algorithms, such as Least Recently Used (LRU) and Least Frequently Used (LFU), originate from such problems. These algorithms were constrained by hard computational performance requirements because cache misses in low-level settings e.g. translation lookaside buffers (TLB) often yield very few additional CPU cycles [2]. Such constraints, however, are less applicable to web caching because response times are significantly higher due to I/O overhead. More recently, driven by these relaxed constraints, novel intelligent caching algorithms that make cache replacement decisions inspired by machine learning methods have been proposed for use in computer networks [3].

The field of Online Convex Optimization (OCO) has had a significant impact on caching research, inspiring many state-of-the-art algorithms [4–6]. An online learning setting involves a decision maker who tries to predict events at every time slot. After a prediction, the actual event gets revealed, and the decision maker receives some reward proportional to the correctness of the prediction [7]. This framework can model many practical problems, such as caching. Content delivery networks, for example, usually have caches in different geographical areas with drastically different request patterns that change over time. No sound assumptions can therefore be made against the order of incoming requests, raising the need for algorithms that make decisions based on the historical sequence rather than assumed statistical properties. A non-adaptable policy cannot be robust in such an uncertain setting, and if these request sequences become adversarial, deterministic policies such as LRU and LFU fare particularly severely (see section 4). Mathematical worst-case performance guarantees are desirable to counter against request sequence uncertainty, and we provide them against an optimal static cache configuration chosen with hindsight, called regret. However, comparing against a benchmark with hindsight knowledge can be difficult, especially in the presence of an adversary. Nonetheless, under the OCO framework, it was shown that it is indeed possible to design a policy that achieves sublinear regret [8], i.e., on average, the policy experiences at most the equivalent loss as the benchmark with complete knowledge of requests.

Recently, [6] proposed a Follow the Perturbed Leader (FTPL) caching policy with a tight regret upper bound. It works by caching most requested objects but adds some noise to the decision process. Although it performs well in multiple settings, a shortcoming of this algorithm is its inability to adapt to changing object popularities, resulting in low performance in several realistic scenarios (see section 4). We propose an extension to FTPL that adds an aging dimension, which reduces the relevance of once popular objects that lose traffic over time. A similar notion can be found in time-to-live (TTL) caching [9]. Furthermore, we pair multiple configurations of this new technique in an Incrementally Adaptive Weighted Majority (IAWM) expert framework algorithm from [10] and show that we maintain a sublinear regret bound.

In practice, web caches often deliver more value when deployed as a network with nodes organized into geographical regions [12]. Therefore, good network performance is a desirable property for any web caching policy. Figure 1 from [11] shows an abstract example of a topology where multiple users have access to small caches, which can download files from the remote server. Users can use any nearby cache to sat-

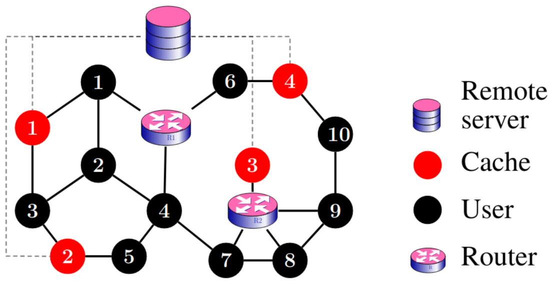


Figure 1: Caching network from [11], where users can access caches by routing requests along routers or other users.

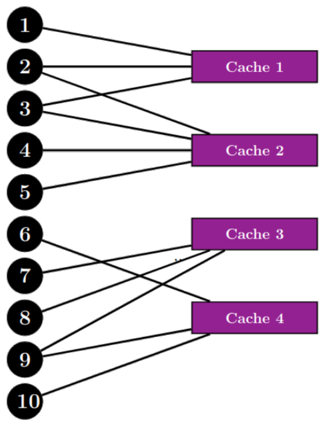


Figure 2: Bipartite network from [11], which is a reduction from figure 1 if we assume that users can access any cache within two hops.

isfy their requests. An expensive request to the remote server is necessary if none of the connected caches contain the requested file. In theoretical scenarios, it often helps to simplify such networks into bipartite networks, where a connection exists between a user and a cache if the cache was accessible to the user in the original topology. Assuming that users in figure 1 can access any cache within two hops, we arrive at the bipartite network in figure 2. FTPL retains sublinear regret in a bipartite caching network if reward functions are assumed to be linear (see section 2.2 for a reward function definition) [6]. A sublinear regret bound was also found for the non-linear case in [11] with slight alterations to the algorithm. Expanding on this, we show that our adaptive FTPL also performs well in a bipartite setting and retains its single-cache bound in a network.

The paper is structured as follows. Section 2 will formally define the caching problem for both single and bipartite systems, section 3 lists the new contributions of the paper, section 4 discusses the experimental setup and findings, section 5 reflects on the ethical considerations of the research, and finally, section 6 draws the conclusions.

2 Problem Description

Before giving a formal definition of a hierarchical caching network, it helps to begin with a single cache. Section 2.1 defines caching in a setting where the only entities are a server, a cache, and a client. Section 2.2 extends this to a bipartite caching network.

2.1 Single cache

We consider caching over a catalog of N files denoted by the set $\mathcal{N}$. The system contains one cache whose maximum size is C. In most cases, $\mathcal{N}$ is significantly larger than the cache (i.e $C \ll N$).

For simplicity, we think of time as slots of indivisible units, represented by the sequence $\{1, 2, ..., T\}$, with T being the time horizon. At every time slot, the client requests one file, one-hot-encoded as vector x^t, picked by an adversary from the following discrete request space.

$$\mathcal{X} = \left\{ x \in \{0,1\}^N \mid \sum_{n=1}^{N} x_n = 1 \right\} \tag{1}$$

This request is stored by a policy π, which records a cache hit when the file was present, or a miss if it was not. After every incoming request, the policy selects a new cache configuration y^{t+1} from the following cache configuration space.

$$\mathcal{Y} = \left\{ y \in \{0,1\}^N \mid \sum_{n=1}^{N} y_n \leq C \right\} \tag{2}$$

A configuration y is an N-dimensional vector where $y_f^t = 1$ if the cache stores file $f \in \mathcal{N}$ at time slot t. We denote the policy as a sequence of mappings $\{\pi^t\}_{t=1}^T$, which at time t uses the map π^t to map the sequence of past requests $\{x^1, x^2, ..., x^T\}$ to the following cache configuration y^{t+1}, i.e.,

$$\pi^t : \mathcal{X}^t \to \mathcal{Y} \tag{3}$$

A reward function $q : \mathcal{X} \times \mathcal{Y} \to \{0, 1\}$ measures the performance of a policy. It returns 1 if the requested file is present in the cache and 0 if it is not.

$$q(x, y) = x \cdot y \tag{4}$$

The cumulative reward of policy π at time T is Q_T^π, the sum of rewards over all time slots up to and including T.

$$Q_T^\pi = \sum_{t=1}^{T} q(x^t, y^t) \tag{5}$$

2.2 Multi-level caching

To expand our definition from one cache to a network of caches, we introduce a new set, $\mathcal{J}$, which denotes all J caches in the system. We represent the configuration of some cache $j \in \mathcal{J}$ at time t as y_j^t. Every cache can store an equal amount of C files to increase convenience without loss of generality. Set $\mathcal{I}$ represents all users that make requests, and the function $\varphi(i)$ will give us all caches connected to user $i \in \mathcal{I}$. Time will remain slotted, with one request per user per slot. We now denote requests by x_i^t for some user $i \in \mathcal{I}$. In a model inspired by [13], every cache is accessible by d clients, which we call the degree of the network. If we think of this model more abstractly, we can easily see that it can model a hierarchical graph of any complexity, such that there is a connection between a user and a cache if any path connects them.

The performance measure of this network is the cumulative sum of rewards at every time slot. We can define the reward function in two ways, depending on the type of content we cache. For example, in [6], the user is expected to get some constant reward for every cache that can serve its request, a realistic scenario when all caches hold layers of the coded content and the benefit of the user (the final video quality, for example) is dependant on how many codes they can retrieve. Coded content would result in the following linear reward function.

$$q(x^t, y^t) = \sum_{i \in I} x_i^t \cdot (\sum_{j \in \varphi(i)} y_j^t) \tag{6}$$

Paper [6] proved sublinear regret under this reward function when every node in the network runs a decentralized instance of FTPL.

In many situations, however, files are indivisible entities, and having more than one copy creates no additional value to clients. A reward function that captures this will award a reward of 1 to a user when some reachable cache has the file and 0 if it does not. The following reward function defines this behavior.

$$q(x^t, y^t) = \sum_{i \in I} x_i^t \cdot \min \left\{ 1, \sum_{j \in \varphi(i)} y_j^t \right\} \tag{7}$$

The LeadCache algorithm introduced in [11] is a modification of FTPL that achieves sublinear regret in this setting. Since our paper is an extension [6], we consider our algorithm in the context of a linear reward function, particularly relevant in modern times where coded video makes up the majority of traffic [1].

2.3 Regret

Intuitively, regret is the maximum difference between the total reward of an optimal stationary cache configuration y^* (chosen with hindsight) and the actual dynamic cache configuration selected by some policy π. This OCO performance metric enables us to prove upper bound guarantees in the presence of an adversary. A sublinear regret bound implies that in the worst-case scenario, the policy performs as well as the optimal static configuration. We now give the formal definition of regret in a single cache system with a time horizon of T.

$$R_T^\pi = \sup_{\{x_1, x_2, \ldots, x_T\} \in \mathcal{X}^T} \left\{ \max_{y^* \in \mathcal{Y}} \sum_{t=1}^{T} q(x^t, y^*) - Q_T^\pi \right\} \tag{8}$$

In a bipartite network, the reward is dependent on multiple users and caches, so we require a corresponding regret definition. We compare the cumulative sum of all user rewards against those achieved by an optimal static cache configuration chosen with hindsight. In our bipartite system, the cache configuration is a 2-dimensional $J \times N$ vector. We define the optimal configuration as y^*, enabling us to define regret for a bipartite network formally. First, we define the reward of some policy π with users $\mathcal{I}$ and time horizon T.

$$Q_{T,\mathcal{I}}^\pi = \sum_{i \in \mathcal{I}} \sum_{t=1}^{T} q(x_i^t, y_i^t) \tag{9}$$

Then, we can define regret as the maximum difference between the static optimal reward and the actual reward.

$$R_{T,\mathcal{I}}^\pi = \sup_{\{x_1, x_2, \ldots, x_T\} \in \mathcal{X}^T} \left\{ \max_{y^* \in \mathcal{Y}} \sum_{i \in \mathcal{I}} \sum_{t=1}^{T} q(x^t, y^*) - Q_{T,\mathcal{I}}^\pi \right\} \tag{10}$$

3 Contributions

This section summarizes the new contributions of the paper. Section 3.1 describes the proposed adaptive FTPL replacement policy, and section 3.2 describes the expert IAWM policy.

3.1 FTPL Algorithm

The FTPL algorithm proposed in [6] uses file request counts to make cache admission decisions. Requests at any time slot are equally relevant and increment the count of the corresponding files by 1. Authors from [6] proved the following expected upper regret bound.

$$\mathbb{E}_{\{\mathbf{y_t}\}_{t \geq 1}}(R_T^{\text{FTPL}}) \leq 1.51 (\log N)^{1/4} \sqrt{CT} \tag{11}$$

Although FTPL captures fixed popularity traces well, it is less suited for traces with changing popularities. Imagine, for example, a trace where C files appear equally often for the first T slots, followed by another different C equally frequently appearing files until time slot $2T$. FTPL would never (assume a large T with insignificant noise) cache the latter files, failing to achieve a high hit ratio in this seemingly trivial setting. If this pattern were to continue indefinitely, the hit ratio would approach zero.

We propose an extension in the form of a discount rate d in the range $(0, 1.5)$, which lowers (or increases) the relevance of files depending on how old their requests are. At every time slot t, we first update an N-dimensional count vector c by $c_{t+1} = dc_t$. Then, to bring the total vector sum back to t, we increment the value at the requested file by the following.

$$c_{t+1,f} = c_{t+1,f} + t - d(t-1) \mid x_f^t = 1 \tag{12}$$

When d is < 1, counts are reduced exponentially on every round, and a growing number increments the requested file at each time slot. This behavior is suitable for traces with changing file popularities. Cyclical patterns, on the other hand, are captured by $d > 1$, which exponentially increases the counts of all files and then decreases the count of the requested file by an increasing amount on every round. Maintaining the same cumulative sum to FTPL keeps gaussian noise effective for dealing with adversarial patterns. We knowingly omit the upper bounds from equation 11 but regain a different sublinear bound in section 3.2. Algorithm 1 presents FTPL with a discount rate.

Algorithm 1 FTPL with discounting

```
counts ← 0
$\eta \leftarrow \frac{1}{(4\pi \log |N|)^{\frac{1}{4}}} \sqrt{\frac{T}{C}}$
for $t \leftarrow 1$ to $T$ do
    counts ← $d$ * counts
    counts$_f$ ← counts$_f$ + $t - d(t-1)$
    Sample $\mathbf{y}_t \sim \mathcal{N}(0, \eta)$
    perturbed counts ← perturbed counts + $\mathbf{y}_t$
    SortDescending(perturbed counts)
    Load first $C$ files to cache from perturbed counts
end for
```

3.2 Expert Algorithm

Adaptability to different traces is a desirable quality of a caching algorithm since request patterns can often be unpredictable. The expert problem, an online learning problem that decides which oracle's advice to follow, can be adapted to create a policy that works well in various settings by choosing appropriate experts for different patterns. The modified FTPL algorithm defined in section 3.1 requires a constant recency factor, making it suitable for one specific pattern. When we use multiple instances of FTPL with different discount rates as experts, we arrive at an algorithm that can adapt to arbitrary sequences.

In the experts' problem, some event occurs at every time slot t. An oracle receives predictions from a set of experts Π with size $|\Pi|$ and makes a prediction $\hat{y}$ based on the advice. Afterward, the oracle observes the actual event y and updates the losses for every expert by a function $l_{\pi,t} = l_{\pi,t-1} + l(y, \hat{y}) \mid \pi \in \Pi$. Algorithms vary in how they calculate the loss and how they select advice from experts given their losses.

An significant shortcoming of our proposed FTPL is that sublinear regret no longer applies when $d \neq 1$. With a reasonable cost, however, we can place the original FTPL algorithm into an expert algorithm with proven sublinear regret and keep a tight upper bound. Moreover, this approach gives a good balance between practical adaptability and adversarial guarantees.

We choose to use the IAWM algorithm first proposed in [10], which enjoys a tight state-of-the-art upper bound. Each expert is associated with a weight that gets reduced proportionally to others when they make a mistake. IAWM is outlined in algorithm 2. The algorithm randomly selects an expert proportionally to their weights and thus begins to favor the expert with the optimal discount rate over time. Our adaptive FTPL inherits its regret bound, defined below [14].

$$R_T^{\text{IAWM}} \leq (2.83 + o(1))\sqrt{L^* \ln |\Pi|} \tag{13}$$

It is sublinear with respect to the smallest loss L^*, which in our case is the difference between the number of possible hits JT and the hits of the best policy (the best reward Q^*), defined as the following.

$$L^* = \min_{\pi \in \Pi} \sum_{t=1}^{T} l(y^t, \hat{y}_\pi^t) = TJ - Q^*, \tag{14}$$

where $Q^* = \max_{\pi \in \Pi} \sum_{t=1}^{T} q(x^t, y_\pi^t)$.

With a tight bound for IAWM, we can prove the sublinearity of our adaptive FTPL:

$$\begin{aligned} R_T^{\text{Overall}} &= \max_{y^* \in \mathcal{Y}} \sum_{t=1}^{T} q(x^t, y^*) - \sum_{t=1}^{T} q(x^t, y^t) \\ &= \max_{y^* \in \mathcal{Y}} \sum_{t=1}^{T} q(x^t, y^*) - \max_{\pi \in \Pi} \sum_{t=1}^{T} q(x^t, y_\pi^t) \\ &\quad + \max_{\pi \in \Pi} \sum_{t=1}^{T} q(x^t, y_\pi^t) - \sum_{t=1}^{T} q(x^t, y^t) \\ &= \min_{\pi \in \Pi} R_T^\pi + R_T^{\text{IAWM}} \\ &\leq R^{\text{FTPL(d=1)}} + R_T^{\text{IAWM}} \\ &= \mathcal{O}(\sqrt{T}) + \mathcal{O}(\sqrt{T}). \end{aligned} \tag{15}$$

Algorithm 2 IAWM

```
$w_{j,0} \leftarrow 1$
$L_{j,0} \leftarrow 0$
$\pi \in \Pi$
**for** $t \leftarrow 1$ to $T$ **do**
    $L^*_{t-1} \leftarrow \min_{\pi \in \Pi}(L_{\pi,t-1})$
    $\epsilon_t \leftarrow \min\{\frac{1}{4}, \sqrt{2\frac{\ln P}{L^*_{t-1}}}\}$
    $a_t \leftarrow \frac{1}{1-\epsilon_t}$
    $W_t \leftarrow \sum_{\pi=1}^{P} a_t^{-L_{\pi,i-1}}$
    $W_{j,t} \leftarrow \frac{a_t^{-L_{\pi,t-1}}}{W_t}$
    SELECTEXPERT($\pi$, probabilities $= w_t$)
    REVEALREQUEST
    INCREMENTLOSSESBY(1)
**end for**
```

4 Results

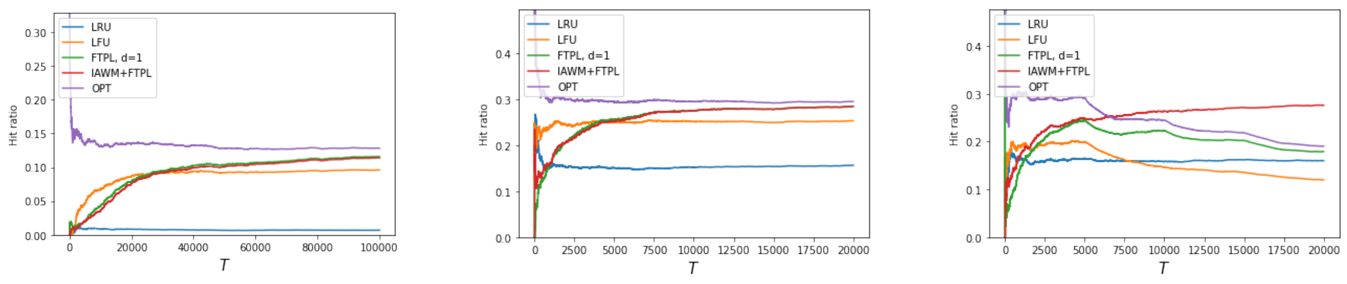
Simulations were run on various traces and system topologies to evaluate the proposed algorithm. Section 4.1 gives an overview of the simulator and the setup of experiments; section 4.2 presents the findings.

4.1 Experimental setup

The source code for the simulator is written in Python3 and can be accessed online at [15]. It contains the FTPL policy described in algorithm 1 and LFU and LRU for comparison. In addition, it includes our adaptive FTPL policy with IAWM from algorithm 2. Multiple policies can be run concurrently through an included simulation runner utility, which returns and plots various statistics. A Jupyter notebook [16] provides an interface to interact with the simulator.

Compatible datasets fall into two categories: single cache datasets containing one continuous trace and multi-cache datasets containing one trace for each client. Both have corresponding interfaces that the runner expects. The source code includes the ability to process the MovieLens dataset [17], as well as synthetic datasets from [18] (including adversarial traces). In a single cache setting, MovieLens movie id-s are filtered to be below $100 * C$, a measure that results in a cache that is 1% of the catalog size. All synthetic traces are run on caches at 10% of the catalog. In the bipartite scenario, a trace is assigned to every user. We use two partitions of the MovieLens trace, two partitions of a synthetic fixed popularity trace, and two adversarial traces. Cache sizes are chosen to be at 10% of the catalog. We consider two different edge patterns, an optimal and an adversarial one. In the optimal setting, users that share trace partitions connect to the same cache, while users with adversarial traces also share a cache. In the adversarial topology, every user with a partition is paired with some adversarial trace to reduce the likelihood of patterns emerging. Performance on both MovieLens and synthetic traces in single and bipartite settings be seen in section 4.2.

Single cache simulations plot data about every policy's (including static optimal) average regret and hit ratio at every time slot. For multi-cache simulations, we plot the



(a) MovieLens

(b) Synthetic fixed popularity

(c) Synthetic changing popularity

Figure 3: Hit ratios for each time slot for non-adversarial realistic traces.

average reward instead. We choose to time average metrics because doing so amplifies learning curves, making it easier to interpret and compare results.

4.2 Findings

Non-adverserial traces

Figure 3 shows the performance of LRU, LFU, the optimal fixed strategy, FTPL from [6], and our proposed IAWM and FTPL combination. We obtained these results on non-adversarial traces in a single-cache setting. The discount rates for the experts are hand tuned to find optimal ranges. Figure 3a showcases performance on the MovieLens trace, while figures 3b and 3c feature synthetic traces that we use to showcase some statistical properties.

All algorithms perform similarly on the MovieLens and fixed catalog traces, with FTPL slightly ahead in both. Perhaps interestingly, LFU significantly outperforms both FTPL and IAWM at the start, but we can attribute this to the perturbations, which have a disproportionately large effect initially. As time increases, FTPL and IAWM emerge as leaders. The flatness of the optimal line shows precisely how fixed the popularity catalog is for the MovieLens trace, explaining why discounting does not offer adaptive FTPL a significant advantage in this scenario. Since we can see a slight decrease in the line on close inspection, it is, in theory, possible that there exists an optimal discount rate $\neq 1$, but finding this requires running more adaptive FTPL instances than we had at our disposal.

The non-adversarial changing popularity catalog trace from figure 3c displays the power of our adaptive FTPL algorithm, which significantly outperforms its competitors. The original FTPL algorithm fails to consider that once popular items are not requested anymore, something that often happens in realistic scenarios. A similar situation could, for instance, arise in the case of new Netflix releases that initially get high traffic, but as more and more users have already seen them and new, more relevant content starts to appear, their popularities fade over time. We should also note that the hit ratio of adaptive FTPL is growing over time rather than decreasing as it does for other algorithms.

Adverserial traces

Figure 4 illustrates how the chosen algorithms perform on synthetic adversarial traces, which aim to reduce performance intentionally. Figure 4a showcases an oscillator trace that cyclically requests the same sequence of files. The changing oscillator in figure 4b follows multiple such sequences at once and switches to different sequences periodically. The sliding popularity trace from figure 4c cyclically changes the popularity patterns of different files.

Adaptive FTPL performs similarly to FTPL on the oscillator trace in figure 4a. The trace intends to degrade LRU and LFU, which both receive no hits. However, storing popular files is a viable strategy due to a repeating pattern, leading to a reasonable performance in FTPL. Adaptive FTPL can gain an edge here by introducing a > 1 discount rate, which elevates the likelihood that files that have not been used for some time get admitted.

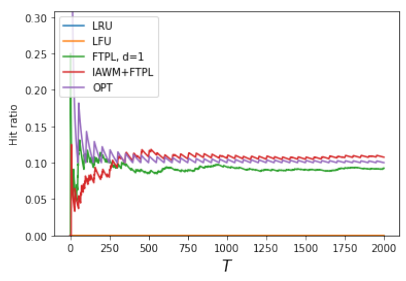
Both LRU and LFU perform very well relative to FTPL in the changing oscillator trace in figure 4b. While FTPL must remain close to the static optimum, an alternative approach is far more effective. The trace is quite complex, and we do not understand why adaptive FTPL performs so well. We configured it with experts from an extensive discount rate range.

Like the oscillator trace, the sliding popularity trace from figure 4c delivers no hits for either LRU or LFU. FTPL and IAWM perform similarly and approximate the optimal static strategy. Failure of adaptive FTPL to improve on FTPL might be caused by the patterns in the trace being simply too complex to address with a discounting mechanism.

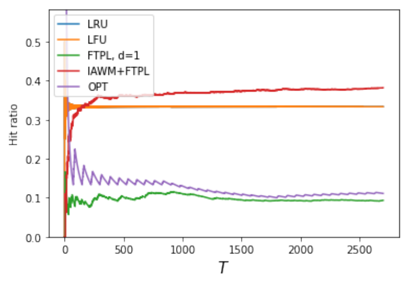
Bipartite setting

Figure 5 presents different algorithms' average rewards in a bipartite setting assuming a linear reward function. Figure 5a illustrates that when users with similar traces execute their requests against the same cache, our adaptive FTPL algorithm can capture the emerging patterns and significantly outperform other algorithms. While every other algorithm, even the static optimal, is dropping in average reward over time, adaptive FTPL keeps learning and improving.

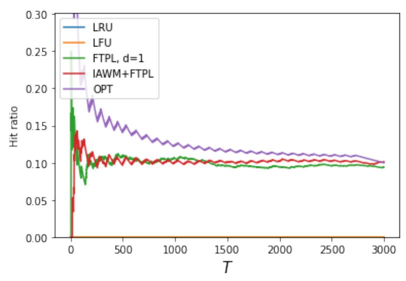
Figure 5b shows how different policies perform with edges between users and caches that aim to degrade performance intentionally. A significant difference from the previous setting is the smaller amount by which adaptive FTPL outperforms other policies. Furthermore, it fails to outperform the static benchmark. A likely cause is the absence of patterns to which our FTPL can adapt. When many dissimilar patterns



(a) Oscillator

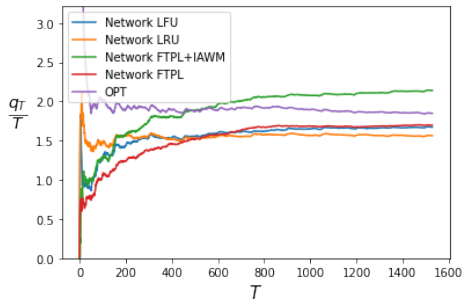


(b) Changing oscillator

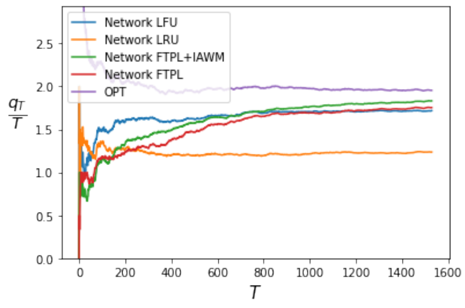


(c) Sliding popularity

Figure 4: Hit ratios for each time slot for synthetic adversarial traces.



(a) Optimal edges



(b) Adversarial edges

Figure 5: Average reward at each time slot for different edge configurations in a bipartite network.

merge into one, the sequence starts to act more and more randomly. This finding highlight the importance of connecting users with similar traces to the same caches. In addition to the obvious benefit of a higher popularity overlap, it allows our algorithm to perform at its best since it can learn an appropriate discount rate. Matching users with caches is an issue that deserves consideration for future research.

5 Responsibility

Improving caching performance is a relevant issue that can significantly reduce technology-related costs and improve services. Lowering barriers to entry can make technology more accessible to people with a less fortunate socioeconomic status, enabling them to acquire more products and warrant new business models that cater to them. Growth in the standard of living is not restricted to these groups and can, of course, be felt in all classes of the society.

With many clear benefits, caching also proposes ethical issues. How often to update objects in the cache was offered as a dilemma in [19]. Refreshing the cache less often increases the likelihood that the user receives inaccurate, outdated data. In some cases, this might have severe consequences. On the other hand, refreshes are expensive and degrade performance. Although doing cache refreshes is not a part of the problem description or the contributions discussed in this paper, it deserves some consideration. The more relevant items a cache holds, the higher the impact of this question.

To ensure that the claimed benefits of our new replacement policy are unbiased, we ran them on various traces. On some occasions, not unexpectedly, it was slightly beat by the original FTPL policy. Furthermore, we admit that in a realistic setting, our proposed expert framework approach might be very performance intensive (many FTPL instances must be run concurrently) and that some other machine learning technique might be more appropriate for learning the discount rate. Finally, we also admit that the range for discount rates was slightly tuned for every figure to showcase our algorithm at its best.

6 Conclusion

We propose a modification to the FTPL caching replacement policy from [6] to make it more adaptive to changing file popularities. We gain adaptability by introducing a discount rate that can lower or increase the relevance of files based on the ages of their requests. The algorithm maintains sublinear regret, learns the optimal discount rate by using the expert framework with the IAWM algorithm from [10], and uses FTPL algorithms with various discount rates, including 1, as experts. We empirically show that such adaptability can significantly improve performance in a single cache setting and a bipartite network, especially when file popularities change over time. Future research should be directed towards learning the optimal discount rate via some reinforcement learning technique as the algorithm is run instead of having these rates preconfigured. It would leave IAWM with two experts, the original FTPL with a tight upper bound and an expert that continuously optimizes the discount rate. Doing this would further increase adaptability and reduce the computational overhead that comes from running multiple FTPL instances. Another relevant research topic is optimally matching users with caches in networks to direct traces with similar patterns to the same caches. We show that such connections increase learnable patterns and improve overall performance.

References

[1] T. Barnett, S. Jain, U. Andra, and T. Khurana, "Cisco visual networking index (vni) complete forecast update, 2017–2022," *Americas/EMEAR Cisco Knowledge Network (CKN) Presentation*, pp. 1–30, 2018.

[2] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing tlb and memory overhead using online superpage promotion," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 176–187.

[3] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *Journal of Network and Computer Applications*, vol. 181, p. 103005, 2021.

[4] G. S. Paschos, A. Destounis, and G. Iosifidis, "Online convex optimization for caching networks," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 625–638, 2020.

[5] T. S. Salem, G. Neglia, and S. Ioannidis, "No-regret caching via online mirror descent," in *ICC 2021-IEEE International Conference on Communications*. IEEE, 2021, pp. 1–6.

[6] R. Bhattacharjee, S. Banerjee, and A. Sinha, "Fundamental limits on the regret of online network-caching," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 2, pp. 1–31, 2020.

[7] S. Shalev-Shwartz *et al.*, "Online learning and online convex optimization," *Foundations and Trends® in Machine Learning*, vol. 4, no. 2, pp. 107–194, 2012.

[8] S. Müller, O. Atan, M. Van Der Schaar, and A. Klein, "Smart caching in wireless small cell networks via contextual multi-armed bandits," in *2016 IEEE International Conference on Communications (ICC)*. IEEE, 2016, pp. 1–7.

[9] J. Gwertzman and M. I. Seltzer, "World wide web cache consistency." in *USENIX annual technical conference*, vol. 141, 1996, p. 152.

[10] P. Auer, N. Cesa-Bianchi, and C. Gentile, "Adaptive and self-confident on-line learning algorithms," *Journal of Computer and System Sciences*, vol. 64, no. 1, pp. 48–75, 2002.

[11] D. Paria and A. Sinha, "Leadcache: Regret-optimal caching in networks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 4435–4447, 2021.

[12] H. Che, Z. Wang, and Y. Tung, "Analysis and design of hierarchical web caching systems," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, vol. 3. IEEE, 2001, pp. 1416–1424.

[13] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[14] R. Yaroshinsky and R. El-Yaniv, *Smooth online learning of expert advice*. Citeseer, 2001.

[15] M. Mäkelä, "Caching simulator for CSE3000," https://github.com/mikkel-makela/caching-simulator, 2022.

[16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[17] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[18] Y. Li, T. Si Salem, G. Neglia, and S. Ioannidis, "Online caching networks with adversarial guarantees," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 3, pp. 1–39, 2021.

[19] G. Barish and K. Obraczke, "World wide web caching: Trends and techniques," *IEEE Communications magazine*, vol. 38, no. 5, pp. 178–184, 2000.