Meta-Learning for Fast Adaption in Caching Networks

Dheeraj Narasimha, Dileep Kalathil, *Senior Member, IEEE*, and Srinivas Shakkottai, *Senior Member, IEEE*

***Abstract*— With the proliferation of short form high quality video content, it has become increasing important to find light weight and efficient edge caching algorithms that can quickly adapt to changing trends. In this context we study an online caching problem where a set of users are connected to a set of caches. The users request files from these caches over a time horizon. These requests arrive sequentially, the sequence of requests are divided into tasks that have a certain degree of similarity. This similarity is leveraged so that we may learn the best policy for a new task using a very small number of sequential requests. We characterize the task averaged regret incurred in this setting, showing an improvement of D/D^* where D is the diameter of the set of cache configurations and D^* is a measure of task similarity. We provide the same theoretical guarantees under both a distributed and smoothed setting. Further, we validate our algorithm on trace based data as well as on synthetic data sets. In the trace based data sets we do not assume any inherent task structure or estimate of D^*. These simulations show not only fast adaptation to new incoming tasks but also improved performance in highly non-stationary request settings.**

***Index Terms*— Communication systems, communication networks, content distribution, caching, online convex optimization (OCO), online learning, transfer learning.**

I. Introduction

As PHONES have grown smarter, the data consumed by the average phone user has grown exponentially. While the set of all files (the library) may be very large, the request for files at a particular time are typically drawn from a much smaller set. For example, the demand for YouTube videos often show strong correlations among users in a certain proximity to each other [1]. This setting is most appropriately modeled using edge caching where we have large central repositories but much smaller file servers at different locations. A user arrives to these locations to request files sequentially. These file requests are highly dynamic and hence, require adaptive policies to serve these requests. Policies such as Least frequently used (LFU) and Least recently used (LRU) often fail to keep up with demand and as a result encounter huge miss rates.

The need for adaptability motivates the creation of policies that can quickly change as the underlying request distribution changes. One perspective that has recently gained much attention is the online convex optimization (OCO) approach to learning what to cache [2], [3], [4]. Here, system is framed as a bipartite caching problem where the users and caches are sets of nodes in a graph. The users do not interact with each other and the caches (representing the smaller file servers) are only connected to the main library of files. An incoming user request is served by all file servers that she can locally connect with to the best of their ability, any request that isn't served by the local servers is served by the central library garnering no reward in the process. In the coded cache setting, the utility is then proportional to the (weighted) fraction of the file received by the user from the local file servers.

While the OCO approach is consistent with an arbitrary popularity model, it often takes a large set of requests in order to adapt to changes. Periodic changes over batches of requests are fairly common in streaming services such as YouTube, which sees daily popularity upswings that can drop down soon afterwards as other items become popular [1]. The gradient ascent approach typically used in OCO would result in a "sluggish" response to such shifts, causing many cache misses as the algorithms learns the appropriate items to cache starting from its current state. Essentially, naïve OCO ignores the relationships between batches of requests over time, and so does not initialize the cache periodically to a good initial configuration that positions it to quickly learn the optimal cache configuration to support requests in the current batch.

The fundamental question that drives our work is whether it is possible to choose an initialization such that for a new batch of requests, a cache may learn the best caching configuration in a short amount of time (using few samples)? Effectively, the goal would be to extract information from the batches of requests that have arrived previously to learn a good initialization from which to learn the best cache configuration for a new batch in a "few-shot learning" manner over only a small set of requests.

The general approach on "learning to learn" falls under the framework of "meta" learning [5], [6], under which a good initialization is learned, from which learning for the current problem may be achieved efficiently. In the caching context, the set of requests is divided up into batches referred to as "tasks," i.e., each task consists of a sequence of requests. The objective of "meta" learning is to learn the best initialization

Received 25 September 2023; revised 1 April 2024 and 24 July 2024; accepted 17 September 2024; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor L. Huang. This work was supported in part by the National Science Foundation (NSF) under Grant 1955696, Grant CRII-CPS-1850206, and Grant NSF-1719384; and in part by Army Research Office (ARO) under Grant W911NF-19-1-0367 and Grant W911NF-19-2-0243. The work of Dileep Kalathil was supported by NSF under Grant 2312978 and Grant 2045783. *(Corresponding author: Dheeraj Narasimha.)*

Dheeraj Narasimha was with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77840 USA. He is now with INRIA, 38058 Grenoble, France (e-mail: dheeraj.narasimha@inria.fr).

Dileep Kalathil and Srinivas Shakkottai are with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77840 USA (e-mail: dileep.kalathil@tamu.edu; sshakkot@tamu.edu).

Digital Object Identifier 10.1109/TNET.2024.3478853

over multiple tasks, so that our learner may use a very small set of samples to quickly arrive at the optimal policy for a new task. As is typical in the online convex optimization setting, we will primarily be interested in the performance of our algorithm with respect to the best policy in hindsight. However, when this setting is specialized to the "meta" problem, we must simultaneously learn both the best policy and the best initialization in hindsight.

A. Main Results

Our main contributions are as follows: We frame the online learning problem of caching as a meta learning problem where a fixed sequence of requests of size T belong to a *task* and we consider M tasks arriving in a sequential fashion. This formulation allows us to re-frame the problem of learning the best cache configuration as a problem of learning the best configuration using a relatively small number of samples. The task based framework requires us to minimize a notion of the so-called "task averaged regret" instead of the traditional notion of regret, which compares the cost suffered at each time with that of the optimal cache configuration. We provide a complete characterization of the task averaged regret in the bipartite setting. Our results recover an order $\mathcal{O}(\sqrt{T})$ regret with an additional $\mathcal{O}\left(\frac{\log M}{M}\right)$ term in the case when our tasks are completely dissimilar. On the other hand, if D is the diameter of the set of cache configurations and D^* is a measure of task similarity, then we show an improvement of order D^*/D in the case when the previous tasks help us learn the new task.

We use the meta framework to prove the efficacy of our algorithm under two variants of the bipartite caching problem. Our first setting is the distributed setting where each of the edge caches can be updated individually, and the second is the "smoothed" setting where we constrain the permissible changes in our cache configuration between updates, so as to not allow large changes in the caches at any one time. In both of these cases, we are able to show the same regret scaling improvements under the meta learning framework.

We test our algorithm using large trace based simulations, as well as over synthetic data sets. Crucially, we demonstrate that even simply by segmenting the data set into batches of equal size i.e, when the algorithm is neither aware of some inherent task structure or possible similarities between the incoming requests, we are able to outperform both traditional algorithms like LRU and LFU, as well as the naive online gradient ascent algorithms considered in earlier work. While our analysis uses a known value of task similarity in our computation, our simulation results makes no such assumption, instead we estimate a value for task similarity while sequentially learning the optimal configuration.

B. Related Work

Caching as a problem was first studied in the context of paging. The question was: which files must one store in fast memory locations in a computer under a variety of assumptions on the request distribution. One of the first algorithms in this domain was Belady's algorithm [7]. The algorithm assumes that all future requests are known (this is termed the "offline setting") and chooses to evict the item in the cache which is furthest in the future. Belady's algorithm is optimal in the offline finite time horizon setting. Several modifications to Belady's algorithm have been made for the case when the future of incoming requests is unknown. Two such algorithms of note which have been widely used are LFU and LRU. LFU is a policy which evicts the least frequently used cache item, where the frequency is calculated over all past requests. Under the assumption of stationary request distributions, the LFU policy is known to be the optimal stationary policy [8]. Similarly, the LRU policy evicts the item in the cache which was least recently used.

Over the years, various further variants of both LFU and LRU policies have been proposed. For instance, variants of LFU and LRU including CLIMB, TTL approximations and RANDOM [9], [10], [11], [12], have been studied. There has also been work on studying caching performance under the model of requests being generated by a stationary process that can change from time to time [8], [13]. While many of these policies are more adaptable, since the choice of parameters are fixed apriori, they suffer from poor adversarial guarantees under non-stationary request distributions.

With the advent of content distribution networks for video streaming and a host of other data intensive services, particularly in the wireless network setting the problem of caching has gained renewed significance. The problem under these settings is typically modeled as a bipartite caching problem where the nodes from one of the disconnected sets belong to the set of caches and the other set of nodes is the set of users [14], [15]. Since the demands of the users can be arbitrary, we frame this problem in an adversarial framework. This setting was considered with coded caching as an online convex learning problem to minimize the regret over a time horizon [2], [16]. Under such a setting, it is known that the online gradient ascent to achieve a regret of $\mathcal{O}(\sqrt{T})$ regret. Several variants of the algorithm, including using mirror descent [17], regret guarantees under prediction [3], [4] and a stronger characterization of regret in the bipartite setting [18] have emerged since.

There has also been work on caching in the "dynamic" regret setting, such as [19]. It is well known that under such a setting it is impossible to achieve an adversarial regret better than $O(T)$. Hence, the paper employs a constraint on how much the tasks can vary over time in order to compute a sub-linear regret algorithm. This constraint means that, on average, the variation of the tasks tends to 0. We do not wish to limit our requests in this manner and hence do not employ this approach.

In this paper, we return to the bipartite caching network but we frame the problem in a *meta* setting [5], [6]. In the meta framework, we have a sequential arrival of requests from users that are divided into tasks (batches of requests). If one considers the requests in a task, one may compute the best static configuration in hindsight. This leads us to a problem in two parts, with the first being to minimize the regret by achieving the optimal cache configuration within a given task. In the second part (the "meta" portion) we are required to learn the right initialization of a cache at the beginning of any task that allows us to quickly arrive at the best configuration.

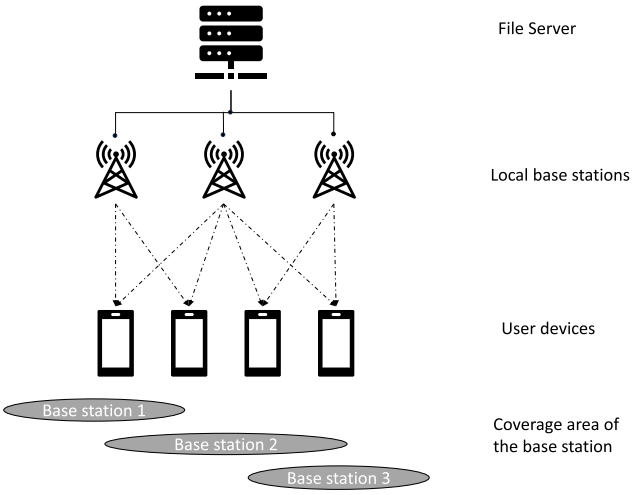
We show that, if our tasks are similar in terms of the optimal configuration for each of them being near each other, then we can use the previous tasks to learn an initialization that allows us to learn the new task much faster than a naïve approach that views the set of all requests as a single giant task. To this end, we show an improvement of regret by a factor of D^*/D, where D is the maximum distance of possible cache configurations while D^* is a task similarity distance that will be formalized in future sections. These improvements are similar to those seen in other meta learning settings [5], [6].

A different notion of "meta" algorithms comes from [3] and [4] where they assume the system has access to a prediction oracle with some known accuracy. The main result of these works is the improvement in regret going so far as to beat the best in hindsight policy under these prediction models. The "meta" aspect of the algorithm is used when there are many prediction oracles and one must assign weights to them based on their quality of predictions. In contrast to their work, we do not assume access to such an oracle.

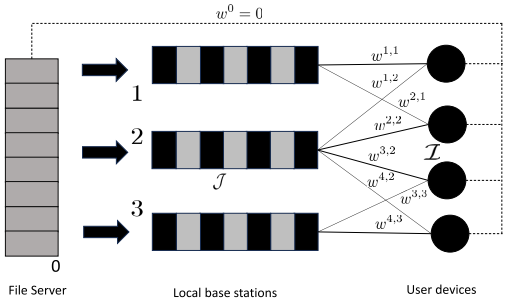
II. Problem Formulation

Network: We consider a caching network where a set of users, denoted as $\mathcal{I} := \{1, 2, \dots I\}$, is connected to a set of caches, denoted as $\mathcal{J} := \{1, 2, \dots J\}$. Each user can be thought of as a mobile edge device and each cache can be thought of as a small-cell base station serving the users nearby. Each user $i \in \mathcal{I}$ is connected to a set of neighbouring caches $\mathcal{J}_i \subset \mathcal{J}$. We focus on a bipartite caching network, where $\mathcal{I}$ and $\mathcal{J}$ form the (left and right) set of vertices of the graph and (i, j) is an edge in this graph if and only if $j \in \mathcal{J}_i$. The library of files are denoted by $\mathcal{N} := \{1, 2 \dots N\}$. In order to efficiently serve the users with data intensive requests (such as streaming videos), the base stations maintain a cache with files that are most likely to be requested by the nearby devices. For the sake of simplicity, we assume that each file occupies one unit of cache and each cache has a capacity C which is much smaller than N. If a file requested by a user is not available with the local caches connected to that user, then that file has to be fetched from a central server. This server can be thought to be located at a macro-cell base station. We denoted this central server with index 0.

File requests: We assume that the file requests arrive sequentially, one at a time, at discrete intervals of time. We also assume that requests are grouped into M batches, where each batch consists of T requests. In the following, we call each batch a "task", a terminology that is consistent with the meta-learning literature. When the context is clear, we will abbreviate time slot t of task m as (t, m). As mentioned in the introduction, the idea of dividing requests into batches aims at exploiting possible correlations across batches, and is fundamental to the "meta" formulation. We will make this explicit later in this section. At each time slot t of task m, exactly one user, $i_{t,m}$, generates a request for exactly one file. We denote this file request as $\mathbf{r}^i_{t,m} = e(n)$ for file n. Here, $e(n)$ is a vector of dimension N whose n^{th} entry is 1 and the other entries are 0. Note that, for any $\tilde{i} \neq i_{t,m}$, $\mathbf{r}^{\tilde{i}}_{t,m} = \mathbf{0}$. In what follows, we will assume that the requests are generated adversarially much like in [2]. This implies that our results hold even in the worst case.



(a) A simple illustration of four users with their devices connected to three base stations



(b) A bipartite graph representation of the four-user and three-base-station system

Fig. 1. A simple illustration of the bipartite caching problem.

Caching: We assume that the caching is performed using maximum distance separable codes, i.e., the files are broken up into chunks. The coding scheme allows the user to decode a file as long as it gets a certain number of chunks (say F), with high probability. This means that a cache may store a fraction of the chunks allowing the user to partially recover the file, and the user can then query only the remaining chunks instead of the whole file. Therefore, this form of coding scheme, for sufficiently large F, allows us to assume that a cache stores a fraction of a file instead of a binary value. To this end, we will use $y^{n,j}_{t,m} \in [0, 1]$ to denote the fraction of file $n \in \mathcal{N}$ stored in cache j at time t for task m. Due to the cache capacity constraints, we have $\sum_{n \in \mathcal{N}} y^{n,j}_{t,m} \leq C$. For cache j, we use the following N dimensional vector to denote the configuration at time t for task m, $\mathbf{y}^j_{t,m} := \{y^{1,j}_{t,m}, y^{2,j}_{t,m}, \dots y^{N,j}_{t,m}\}$.

Routing: If user i requests file n at time t for task m, then we must decide how much of this request is fulfilled by the neighboring caches $\mathcal{J}_i$. We denote the vector corresponding to this routing scheme for cache $j \in \mathcal{J}_i$ by $\mathbf{z}^j_{t,m} = \{z^{1,j}_{t,m}, z^{2,j}_{t,m} \dots z^{N,j}_{t,m}\}$. A feasible routing vector $\mathbf{z}^j_{t,m}$ must satisfy basic restrictions: we cannot route more from the cache than is available, i.e, $z^{n,j}_{t,m} \leq y^{n,j}_{t,m}$ for all j and n. Also, $\sum_{j \in \mathcal{J}_i \cup \{0\}} z^{n,j}_{t,m} = 1$, implying that the files that are not available in the neighbouring caches are procured from the central server denoted as 0. If i is the only user that makes the request, then for any $j \notin \mathcal{J}_i$, we set $\mathbf{z}^j_{t,m}$ to 0. Note that our routing vector corresponds to the amount of a

file served by a given cache for a request and is indexed in terms of the cache and not the user. Finally, we will use $\mathbf{z}_{t,m} = \{\mathbf{z}^1_{t,m}, \mathbf{z}^2_{t,m} \dots \mathbf{z}^{|\mathcal{J}|}_{t,m}\}$ to denote our routing vector at (t,m). The set of feasible routing vectors is denoted as $\mathcal{Z}(\mathbf{y}_{t,m})$. Note, this set depends explicitly on the cache configuration at (t,m).

When the network is only tasked with providing a single file and gains no additional benefit by providing more than the requisite F chunks is known as the *inelastic setting*, see [18] and the references therein for more context. In later sections in the paper we will encounter the *elastic* setting where a user benefits proportional to the number of chunks received, hence, one may set $z^{n,j}_{t,m} = y^{n,j}_{t,m}$.

Utility: For a given cache configuration $\mathbf{y}_{t,m}$ and a feasible routing vector $\mathbf{z}_{t,m}$, we define the utility of serving user request(s) at (t,m) as $\sum_{i=1}^{|\mathcal{I}|}\sum_{j=1}^{|\mathcal{J}|}\sum_{n=1}^{|\mathcal{N}|} w^{i,j} r^{i,n}_{t,m} z^{n,j}_{t,m}$, where $w^{i,j}$ is the weight associated with the edge (i,j). Note, the devices always have direct access to the file server, however, the weight associated with any connection to the file server is *always* 0, we denote all such edges with a universal $w^0 = 0$ (see fig 1 for example). The edge weights depend on a variety of factors such as the latency, connection consistency, and the number of other users connected to a given cache. At each (t,m), for a given $\mathbf{y}_{t,m}$, the instantaneous utility $f_{t,m}(\mathbf{y}_{t,m})$ is then defined by selecting the best routing,

$$f_{t,m}(\mathbf{y}_{t,m}) \triangleq \max_{\mathbf{z}_{t,m}\in\mathcal{Z}(\mathbf{y}_{t,m})} \sum_{i=1}^{|\mathcal{I}|}\sum_{j=1}^{|\mathcal{J}|}\sum_{n=1}^{|\mathcal{N}|} r^{i,n}_{t,m} w^{i,j} z^{n,j}_{t,m}. \quad (1)$$

Figure (1) illustrates the system model described so far in this section. The first figure is a sketch of a four user three base station network with a file hosting server at the back end. The second figure is a bipartite representation of the sketch in the first figure. It should be noted that our results carry over unchanged if the weights are time varying as well as item dependent. The former is pertinent if the user is moving and hence, changing the underlying connection graph (the graph still remains bipartite). The latter corresponds to weights associated with different cache requests from different applications. Henceforth, we will not specify such weights, since the notation becomes extremely cumbersome otherwise.

Cache placement: First, at (t,m), an adversary picks our request vector $\mathbf{r}_{t,m}$. Next, given our cache configuration $\mathbf{y}_{t,m}$, the request generates a utility to our network through equation (1). Finally, based on the utility received, the cache may now update its configuration $\mathbf{y}_{t+1,m} := \{\mathbf{y}^j_{t+1,m}\}_{j\in\mathcal{J}}$. We do not restrict the update vector to the missing file, nor do we restrict the size of the update i.e, $\|\mathbf{y}_{t+1,m} - \mathbf{y}_{t,m}\|$ (we will address this in a later section). In order to compute an update, it may use the history of requests up to and including time slot (t,m).

Role of meta learning: We will now take a moment to distinguish between the conventional online convex optimization approach and the "*meta*" approach. Let us consider the problem at the end of $m-1$ tasks, the following information is available to us; we have the history of all file configurations chosen by our caches, $\mathcal{J}$, as well as the history of all requests made to this point, $H_{m-1} := \{\mathbf{y}_{1,1}, \mathbf{y}_{2,1}\dots\mathbf{y}_{T,1}, \mathbf{y}_{1,2}, \mathbf{y}_{2,2}\dots\mathbf{y}_{T,2}\dots\mathbf{y}_{T,m-1}\} \cup \{\mathbf{r}_{1,1}, \mathbf{r}_{2,1}\dots\mathbf{r}_{T,1}, \mathbf{r}_{1,2}, \mathbf{r}_{2,2}\dots\mathbf{r}_{T,2}\dots\mathbf{y}_{T,m-1}\}$. Further, using this history, for each task $\hat{m} \in \{1,2,\dots m-1\}$ we know the corresponding best static configuration in hindsight $\mathbf{y}^*_{\hat{m}}$. Therefore, we wish to use $\{H_{m-1}, \{\mathbf{y}^*_{\hat{m}}\}_{\hat{m}=1}^{m-1}\}$ to find an initialization $\mathbf{y}_{1,m}$ so that we may learn, $\mathbf{y}^*_m$, using a relatively small number of samples. To this end, the previous tasks will provide no benefit to us if they provide no new information to us over the previous $m-1$ rounds. We therefore define a notion of task similarity as follows:

Definition 1: Let there exists a convex subset $\mathcal{Y}^* \subset \mathcal{Y}$ from which the best configuration in hindsight, $\mathbf{y}^*_m$, are drawn. $\bar{\mathcal{Y}}^*$ is the closure of this set. Then,

$$D^* = \sup_{\boldsymbol{x}^*,\boldsymbol{y}^*\in\bar{\mathcal{Y}}^*} \|\boldsymbol{x}^* - \boldsymbol{y}^*\|_2 \quad (2)$$

is the maximum distance between any two points in $\bar{\mathcal{Y}}^*$.

In the worst case, if $D := \sup_{\mathbf{x},\mathbf{y}\in\mathcal{Y}} \|\mathbf{x}-\mathbf{y}\|_2$ is the diameter of the set of configurations, then, we will have $D^* = D$. However, in the case when $D^* < D$, we will demonstrate a benefit proportional to the reduction in distance due to task similarity with an additional vanishing term in M. Note, here, $\|\cdot\|_2$ refers to the L2 norm *i.e.* for any d dimensional vector x, the L2 norm of x given by $\|x\|_2 := \sqrt{x_1^2 + x_2^2 + \dots x_d^2}$.

We may now define the regret in accordance with [5], which we call task averaged regret:

Definition 2: Task averaged regret, denoted by $\bar{R}$, is the average regret incurred by a file configuration over a set of tasks M when each task has T sequential samples i.e.,

$$\bar{R}(T\times M) := \frac{1}{M}\left(\sum_{m=1}^{M}\sum_{t=1}^{T}\left[f_{t,m}(\mathbf{y}^*_m) - f_{t,m}(\mathbf{y}_{t,m})\right]\right) \quad (3)$$

where $\mathbf{y}^*_m$ is the best configuration for a task in hindsight i.e, the single configuration that incurs the most utility in hindsight.

Note, this notion of regret dominates the best configuration in hindsight where we treat the $T\times M$ total slots as one giant task with y^* being the best task in hindsight over the entire time horizon. Further, this provides a major point of distinction between the previous online caching problems such as [2], [4], and [17] and our own caching problem. Instead of having one adversary over a large time horizon, one may view this problem as a series of distinct adversaries (under suitable constraints) with relatively small time horizons for us to specialize our cache configuration. Finally, we note that the meta regret differs from the dynamic regret. The dynamic regret compares $y_{t,m}$ with $y^*_{t,m}$ an omniscient adversary who knew of *all* incoming functions before hand and individually picks the best policy *for each function*. It is not hard to see that such an adversary would generate $O(T)$ regret without any restrictions. Here, we restrict ourselves to an adversary who can only pick one configuration for a *task of T requests* instead.

Concretely, we may summarize our overall objective into two parts for an incoming task. First, we would like to learn the best initial configuration that allows us to learn the best configuration for this task using the fewest requests. Second, we would like to choose a sequence of configurations that minimizes the regret incurred from this task. We will look to minimize the task averaged regret $\bar{R}(T\times M)$ in order to achieve the objectives described above. The remainder of this

section will be spent on a few definitions that will feature in our main results.

To this end, for task m, we would like to choose our initialization Φ_m such that, $\|\mathbf{y}_m^* - \Phi_m\|_2^2$ is as small as possible in order to minimize the regret.

The average distance over M sequential tasks is then given by:

$$\frac{1}{M}\left(\sum_{m=1}^{M}\|\mathbf{y}_m^* - \Phi_m\|_2^2\right)$$

The meta problem of finding the best initialization in hindsight that minimizes this error over the tasks. Let Φ^* denote this value i.e,

$$\Phi^* := \arg\min_{\Phi\in\mathcal{Y}} \frac{1}{M}\left(\sum_{m=1}^{M}\|\mathbf{y}_m^* - \Phi\|_2^2\right) \tag{4}$$

Formulating this problem as an online convex learning problem of computing the best initialization in hindsight, we may now define our *meta regret* for M tasks as follows:

$$R_{meta}(M) := \frac{1}{M}\sum_{t=1}^{M}\left[\|\Phi_m - \mathbf{y}_m^*\|_2^2 - \|\mathbf{y}_m^* - \Phi^*\|_2^2\right] \tag{5}$$

With the problem setup firmly in mind, the following section will describe the main results for our paper.

III. Meta Learning for a Bipartite Caching Network

This section highlights our main results on meta learning for caching. We will begin with a reformulation of the bipartite caching problem and a brief summary of the results from [2] and [16]. We will then quickly move on to our main results and their consequences.

At time (t, m) a user $i = i_{t,m}$ generates a request vector, for a file n can be written as, $\mathbf{r}_{t,m}^i := e(n) \in \{0,1\}^N$ where $e(n)$ is the vector that is 1 at location n and 0 in all other locations. Further, the full request vector $\mathbf{r}_{t,m} = \{\mathbf{r}_{t,m}^i\}_{i=1}^I$ is the $I \times N$ dimensional vector created by stacking the vectors together. Note, since we have exactly one user making a request for one file at a time, $\mathbf{r}_{t,m}^{\hat{i}} = 0$ the zero vector for any $\hat{i} \neq i$

Next, we can similarly describe the caching vector at time t for task m; $\mathbf{y}_{t,m}^j = [0,1]^N$ is the vector containing the fraction of files from the library such that it satisfies the capacity constraints $\|\mathbf{y}_{t,m}^j\|_1 \leq C$. Then, $\mathbf{y}_{t,m} := \{\mathbf{y}_{t,m}^j\}_{j=1}^J$ is the $J \times N$ dimensional vector associated with stacking the set of J cache configurations at time slot t. Recall, we assign weights $w^{i,j}$ to the connection between users i and cache j. For a fixed i and j,

let $W_{i,j} = w^{i,j}\mathbb{1}_{N\times N}$ where $\mathbb{1}_{N\times N}$ is the N dimensional identity matrix. Further, let $\mathbb{W}$ be the $(I \times N) \times (J \times N)$ block matrix whose elements are comprised of W_{ij} i.e,

$$\mathbb{W} := \begin{bmatrix} W_{11} & W_{12} & \dots & W_{1J} \\ W_{21} & W_{22} & \dots & W_{2J} \\ \vdots & \vdots & \dots & \vdots \\ W_{I1} & W_{I2} & \dots & W_{IJ} \end{bmatrix}$$

Using this new notation, for an incoming request from user i, with feasible routing vector $\mathbf{z}_{t,m} := \mathbf{z}(\mathbf{y}_{t,m}, \{w^{i,j}\}) \in [0,1]^{J\times N}$ being the quantity of file n taken from cache j to service a request $\mathbf{r}_{t,m}^i$ we get,

$$f_{t,m}(\mathbf{y}_{t,m}) = \left(\mathbf{r}_{t,m}\right)^T \mathbb{W}\mathbf{z}_{t,m} =: \langle \mathbf{r}_{t,m}, \mathbf{z}(\mathbf{y}_{t,m}, \{w^{i,j}\})\rangle_{\mathbb{W}} \tag{6}$$

It is not hard to show, given a request $\mathbf{r}_{t,m}^i = e(n)$ the optimal routing vector $\mathbf{z}_{t,m}(\{\mathbf{y}_{t,m}^j\}_{j\in\mathcal{J}_i}, \{w^{i,j}\}_{i,j})$ can be found as follows: for each j attached to i, order the weights $w^{i,j}$ in decreasing order, denoted by θ. So we have, $w^{i,\theta(1)} \geq w^{i,\theta(2)} \dots \geq w^{i,\theta(|\mathcal{J}_i|)}$. Now, the routing vector for the K^{th} cache in the set of ordered caches, $\theta(K)$ is set to

$$z^{\theta(K),\hat{n}} = \begin{cases} \min\{\mathbf{y}_{t,m}^{\theta(K),\hat{n}}, 1 - \sum_{k=1}^{K-1}\mathbf{y}_{t,m}^{\theta(k),\hat{n}}\} \text{ for } \hat{n} = n \\ 0 \text{ otherwise} \end{cases} \tag{7}$$

and the 0 vector for all other caches that are not directly connected to the user i. Intuitively, we try to route the file from the cache with the most value to fulfill an incoming request. If the most valuable cache is unable to completely fulfill this request we move on to the next most valuable cache and try to fulfill the remaining request and so on until the file request is completed to the best of the network's ability. Since this is the minimum of two linear functions, the routing function is concave. Hence, a linear combination of these functions must also be concave. Thus, even though the routing maybe non-differentiable in $\mathbf{y}$; it is possible to compute the super gradient. The following lemma [lemma 1, [2]] states this result concretely.

Lemma 1: The utility at slot t for task m, $f_{t,m}(\mathbf{y})$ for the set $\mathcal{Y}$, is concave in y. Hence, a super gradient exists for all $y \in \mathcal{Y}$.

We may now directly compute the super gradient $\mathbf{g}_{t,m} = \partial f_{t,m}(\mathbf{y}_{t,m})$ in order to bound it. Note, $z^{n,\theta(K)}(\mathbf{y}_{t,m}) = 1 - \sum_{k=1}^{K-1}\mathbf{y}_{t,m}^{n,\theta(k)} > 0$ if and only if the K^{th} most valuable is the last cache to serve a request $r_{t,m}$. If $z^{n,\theta(K)}(\mathbf{y}_{t,m}, \{w^{i,j}\}) = \mathbf{y}_{t,m}^{\theta(K),n}$, then the K^{th} cache completes as much of the file as it can serve and $z^{n,\theta(K)} = 0$ otherwise. We will denote the case where every cache attached to i, i.e, $j \in \mathcal{J}_i$ tries its best to serve the request, $z^{n,\theta(k)}(\mathbf{y}_{t,m}, \{w^{i,j}\}) = \mathbf{y}_{t,m}^{\theta(k),\hat{n}}, k \in \{1, 2 \dots |\mathcal{J}_i\}$ by **Case I**. When the cache configuration is unable to serve the request, that is, for any $K \leq |\mathcal{J}_i|$, $z^{n,\theta(K)} = 1 - \sum_{k=1}^{K-1}\mathbf{y}_{t,m}^{n,\theta(k)} > 0$ by **Case II**. For Case II, let $K_{t,m}$ denote the least valuable cache that serves the request at time (t, m). Using (6) and (7), the supergradient can be written as:

$$g_{t,m}^{n,\theta(k)} = \begin{cases} w^{i,\theta(k)} & \text{for Case I} \\ w^{i,\theta(k)} - w^{i,\theta(K_{t,m})} & \text{for Case II} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

By (8) the gradient can be bounded for any $j \in \mathcal{J}_i$ by,

$$|g_{t,m}^{n,j}| \leq w^{i,j} \leq \|w\|_\infty$$

Algorithm 1 Meta Updated Online Bipartite Caching

1: Input: sequential requests of size M starting from 1 to T, $\{\{r_{1,1},\dots r_{T,1}\},\{r_{1,2},\dots r_{T,2}\}\dots\{r_{1,M}\dots r_{T,M}\}\}$, Bipartite Network with users $\mathcal{I}$ and cache $\mathcal{J}$, similarity distance D^*
2: Let Φ_1 be an arbitrary configuration, $\eta = \frac{D^*}{\sqrt{T}\|w\|_\infty\sqrt{d_{max}}}$
3: **repeat**
4: initialize $\mathbf{y}_{1,m} \leftarrow \Phi_m$
5: **repeat**
6: compute $z(\mathbf{y}_{t,m})$ using (7)
7: compute $g_{t,m}$ using $r_{t,m}$ and (8)
8: $\mathbf{y}_{t+1,m} := \Pi_{\mathcal{Y}}(\mathbf{y}_{t,m} + \eta g_{t,m})$
9: Increment t by 1
10: **until** $t > T$
11: $\mathbf{y}^*_m \leftarrow \arg\min_{y\in\mathcal{Y}} \sum_{t=1}^{T} f_{t,m}$
12: $\Phi_{m+1} = \Pi_{\mathcal{Y}}(\Phi_m - \alpha_m(\Phi_m - \mathbf{y}^*_m))$
13: Increment m by 1
14: **until** $m > M$
15: **return**

So, computing the L2 norm for $g_{t,m}$ will now give us:

$$\sqrt{\sum_{k=1}^{\mathcal{J}_i} |g_{t,m}^{n,\theta(k)}|^2} \le \sqrt{\|w\|_\infty^2 |\mathcal{J}_i|} \le \|w\|_\infty \sqrt{|\mathcal{J}_i|}$$
$$\|g_{t,m}\|_2 \le \|w\|_\infty \sqrt{d_{max}}$$

where d_{max} is the maximum number of caches a user, i is attached to i.e, $d_{max} = \max_{i\in\mathcal{I}} |\mathcal{J}_i|$.

The bipartite meta caching algorithm is presented in Algorithm (1). Our algorithm has an inner loop minimizes the in-task regret (line 7 and 8) using regular gradient ascent (specifically *projected gradient ascent*) while an outer loop finds the best initialization, Φ_m for each task (line 11 and 12). Note, in line 8, $\Pi_{\mathcal{Y}}$ refers to the projection to the convex set $\mathcal{Y}$. By computing an appropriate initialization we reduce the distance between the initialization picked by the algorithm and the best cache configuration for any given task, which in turn reduces the number of samples taken to find the optimal cache configuration.

Before describing our first result, we will state a useful result first proposed by [20].

Lemma 2: [Theorem 3.3, [21]] Given the regret formulation (5), using Algorithm 1 with step size $\alpha_m = \frac{1}{\kappa m}$, for some constant $\kappa > 0$ achieves,

$$R_{meta}(M) \le (D^*)^2 \sum_{m=1}^{M} \alpha_m = (D^*)^2 \sum_{m=1}^{M} \frac{1}{\kappa m} \tag{9}$$

where due to task similarity we know D^* is an upper bound on $\|\Phi_m - \mathbf{y}^*_m\|$.

We leverage the lemma above in the following theorem to show that the regret for the meta bipartite caching problem yields an order $\mathcal{O}(\sqrt{T})$ regret, indicating that as T grows large, the cache configuration converges to the optimal single configuration in hindsight.

Theorem 1: Suppose D^* is the task similarity distance described in Definition 1; then, Algorithm 1 gives a task averaged regret (recall Definition 2) bounded above as:

$$\bar{R}(T\times M) \le \frac{(D^*)^2}{\kappa}\frac{1+\log M}{M} + D^*\|w\|_\infty\sqrt{Td_{max}}$$

where κ is as defined in Lemma 2.

Proof: We begin by writing out the Task averaged regret,

$$\bar{R}(T\times M) := \frac{1}{M}\sum_{m=1}^{M}\sum_{t=1}^{T}\Big(f_{t,m}(\mathbf{y}^*_m) - f_{t,m}(\mathbf{y}_{t,m})\Big).$$

Now, rewriting the regret for task, m we get

$$R_m(T) = \sum_{t=1}^{T}\Big(f_{t,m}(\mathbf{y}^*_m) - f_{t,m}(\mathbf{y}_{t,m})\Big).$$

From the concavity of $f_{t,m}$, we have,

$$R_m(T) \le \sum_{t=1}^{T}\langle \mathbf{g}_{t,m}, (\mathbf{y}^*_m - \mathbf{y}_{t,m})\rangle.$$

From the contractive property of projection operators we have,

$$\begin{aligned}\|\mathbf{y}_{t+1,m} - \mathbf{y}^*_m\|_2^2 &= \|\Pi_{\mathcal{Y}}(\mathbf{y}_{t,m} + \eta\mathbf{g}_{t,m}) - \mathbf{y}^*_m\|_2^2\\ &\le \|\mathbf{y}_{t,m} + \eta\mathbf{g}_{t,m} - \mathbf{y}^*_m\|_2^2.\end{aligned}$$

Expanding the L2 norm gives us,

$$\begin{aligned}\|\mathbf{y}_{t,m} + \eta\mathbf{g}_{t,m} - \mathbf{y}^*_m\|_2^2 \le \|\mathbf{y}_{t,m} - \mathbf{y}^*_m\|_2^2\\ + 2\eta\langle \mathbf{g}_{t,m}, (\mathbf{y}_{t,m} - \mathbf{y}^*_m)\rangle + \eta^2\|\mathbf{g}_{t,m}\|_2^2.\end{aligned}$$

Rearranging we get,

$$\begin{aligned}\|\mathbf{y}_{t+1,m} - \mathbf{y}^*_m\|_2^2 - \|\mathbf{y}_{t,m} - \mathbf{y}^*_m\|_2^2\\ \le 2\eta\langle \mathbf{g}_{t,m}, (\mathbf{y}_{t,m} - \mathbf{y}^*_m)\rangle + \eta^2\|\mathbf{g}_{t,m}\|_2^2.\end{aligned}$$

Summing the terms telescopically from $t = 1$ to T we get,

$$\begin{aligned}\|\mathbf{y}_{T+1,m} - \mathbf{y}^*_m\|_2^2 - \|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2\\ \le 2\eta\sum_{t=1}^{T}\langle \mathbf{g}_{t,m}, (\mathbf{y}_{t,m} - \mathbf{y}^*_m)\rangle + \eta^2\sum_{t=1}^{T}\|\mathbf{g}_{t,m}\|_2^2.\end{aligned}$$

Rearranging the terms and substituting our bounds for the gradient gives us,

$$\begin{aligned}2\eta\sum_{t=1}^{T}\langle \mathbf{g}_{t,m}, (\mathbf{y}^*_m - \mathbf{y}_{t,m})\rangle \le \|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2 - \|\mathbf{y}_{T+1,m}\\ - \mathbf{y}^*_m\|_2^2 + \eta^2 T\|w\|_\infty^2 d_{max}.\end{aligned}$$

Since, $\|\mathbf{y}_{T+1,m} - \mathbf{y}^*_m\|_2^2 > 0$ we get,

$$\begin{aligned}\sum_{t=1}^{T}\langle \mathbf{g}_{t,m}, (\mathbf{y}^*_m - \mathbf{y}_{t,m})\rangle\\ \le \frac{\|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2}{2\eta} + \frac{\eta T\|w\|_\infty^2\sqrt{d_{max}}}{2}.\end{aligned}$$

Finally, we have:

$$\begin{aligned}R_m(T) \le \sum_{t=1}^{T}\langle \mathbf{g}_{t,m}, (\mathbf{y}^*_m - \mathbf{y}_{t,m})\rangle\\ \le \frac{\|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2}{2\eta} + \frac{\eta T\|w\|_\infty^2 d_{max}}{2}.\end{aligned}$$

Now substituting our bound into the task averaged, we have,

$$\bar{R}(M \times T) \leq \frac{1}{M} \sum_{m=1}^{M} \frac{\|\mathbf{y}_{1,m} - \mathbf{y}_m^*\|_2^2}{2\eta} + \frac{\eta T \|w\|_\infty^2 d_{max}}{2}.$$

Let Φ^* be the quantity defined in (4), then the RHS can be rewritten as follows,

$$\bar{R}(T \times M) \leq \frac{1}{2\eta M} \sum_{m=1}^{M} \left(\|\mathbf{y}_{1,m} - \mathbf{y}_m^*\|_2^2 - \|\mathbf{y}_m^* - \Phi^*\|_2^2 \right) + \frac{1}{2\eta M} \left(\sum_{m=1}^{M} \|\mathbf{y}_m^* - \Phi^*\|_2^2 + \eta^2 T \|w\|_\infty^2 d_{max} \right).$$

Concretely, one may now decompose the task averaged regret $\bar{R}(T \times M)$ in terms of the *meta regret* (5) and *per task regret*.

$$\bar{R}(T \times M) \leq \frac{1}{2\eta} R_{meta}(M) + \frac{1}{2\eta M} \left(\sum_{m=1}^{M} \|\mathbf{y}_m^* - \Phi^*\|_2^2 + \eta^2 T \|w\|_\infty^2 d_{max} \right)$$

Finally, updating our initialization for each task m using OGA with step size, $\eta\alpha_m$ from Lemma (2), we get:

$$\bar{R}(T \times M) \leq \frac{(D^*)^2}{\kappa} \frac{1 + \log M}{M} + \frac{1}{2\eta M} \times \left(\sum_{m=1}^{M} \|\mathbf{y}_m^* - \Phi^*\|_2^2 + \eta^2 T \|w\|_\infty^2 d_{max} \right)$$

Substituting our task similarity distance D^* into the second term in the equation above,

$$\bar{R}(T \times M) \leq \frac{(D^*)^2}{\kappa} \frac{1 + \log M}{M} + \frac{(D^*)^2}{2\eta} + \frac{\eta T \|w\|_\infty^2 d_{max}}{2}$$

Now letting $\eta = \frac{D^*}{\sqrt{T}\|w\|_\infty \sqrt{d_{max}}}$ gives us:

$$\bar{R}(T \times M) \leq \frac{(D^*)^2}{\kappa} \frac{1 + \log M}{M} + D^* \|w\|_\infty \sqrt{T d_{max}}$$

■

A few remarks are now in order for our formulation and solution for the meta bipartite caching problem.

Remarks: We note that there are two reasons the meta algorithm is advantageous in the bipartite setting but may not be so helpful in the single cache setting. Firstly, note, the naïve algorithm uses a step size of order $\frac{D}{\sqrt{T}}$ where $D = \sqrt{2|\mathcal{J}|C}$. In the single cache setting, J is 1, hence, unless the request distribution varies greatly, we do not expect D^* to be very different from the D. However, in the Bipartite setting J can grow very large and hence, we find that D^* can be significantly smaller than D, especially in cases where there are similarities between tasks across different caches. Secondly, note the multiplicative factor of $\|w\|_\infty \sqrt{d_{max}}$. This value is 1 in the single cache setting but can grow large depending on the topology of the bipartite graph. Hence, even small improvements in the value of D^* over D get amplified by this factor.

The cases highlighted in (8) imply that the gradient changes discontinuously when moving from one case to another. As a result our objective function is not differentiable everywhere and it is more appropriate to write that the values described in (8) *belong* to the super gradient. For clarity of presentation we do not make the distinction, further, the bounds on the gradient and thus, the bounds on the regret still hold true.

In principle one can replace the simple OGA algorithm in the inner loop with any other algorithm that minimizes regret. This can include algorithms such as mirror ascent or "follow the perturbed leader". In all these cases, it is not hard to see that our "meta" approach will provide a similar improvement in performance.

Finally, the inner product formulation for our utility allows us to immediately see how to handle multiple requests from multiple users. The routing problem for multiple requests devolves into a routing problem for each user which can be handled using (7). While the bound on our gradient will change by the number of requests, it is not hard to see that the inner product formulation of (6) does not change at all.

IV. Smoothed Problem

Throughout the paper, we have considered the bipartite caching problem when there is no replacement cost. This means that a cache can potentially change its entire configuration between the arrival of two consecutive requests. This is an impractical assumption although it was very useful in setting up the regret bounds for out meta bipartite caching problem. We now turn to a more practical consideration where we attach a cost for changes in cache configuration. We modify the utility for our cache as follows:

$$\tilde{f}_{t,m}(\mathbf{y}_{t,m}) = \langle \mathbf{r}_{t,m}, z(\mathbf{y}_{t,m}, \{w^{i,j}\}) \rangle_{\mathbb{W}} - \beta \|\mathbf{y}_{t,m} - \mathbf{y}_{t-1,m}\|$$

Note, due to the equivalence of norms, we are free to choose norms depending on the context. More importantly we need to distinguish between the smooth problem during the T sequential samples in a task and the cost during M initializations as part of the meta problem. We assume that at the beginning of a new task, the initialization can be arbitrary and incurs no additional cost.

Remark: The notion of smoothing relates to a trade-off between a dynamic notion of regret and a static notion of regret. When β is low, the best dynamic policy in hindsight over a time horizon will vary greatly over time; since, there is relatively little incentive to stick to the current action. When β tends to infinity, changes in the policy result in a heavy price, hence, we prefer a static policy in hindsight in keeping with the idea of regret used in this paper. A more detailed discussion on this topic can be found in [22]. In this section we are primarily looking at the problem from the latter perspective where changing cache configuration often is not a preferable outcome.

Theorem 2: The meta OGA algorithm 1 for the smoothed problem with η equals $\frac{D^*}{\sqrt{T(1+\beta\|w\|_\infty\sqrt{d_{max}}L_2)}}$ gives us a task averaged regret of

$$\bar{R}(T \times M) \leq D^* \sqrt{T(1 + \beta \|w\|_\infty L_2 \sqrt{d_{max}})} + \frac{(D^*)^2}{\kappa} \frac{1 + \log M}{M}$$

Proof: We will restrict ourselves to the $\sum_{t=1}^{T} \|\mathbf{y}_{t,m} - \mathbf{y}_{t-1,m}\|$ smoothing terms for our task since we have already

shown in the proof of 1 that the in task regret for task m denoted by R_m with step size η; is bounded above by

$$R_m(T) := \sum_{t=1}^{T} f_{t,m}(\mathbf{y}^*_m) - f_{t,m}(\mathbf{y}_{t,m}) \leq \frac{\|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2}{2\eta} + \frac{\eta T}{2}$$

Recall, the bound on the supergradient is $\|g_{t,m}\|_2 \leq \|w\|_\infty \sqrt{d_{max}}$. Next, recall for a single task m by Algorithm 1, we have $\mathbf{y}_{t+1,m} = \mathbf{y}_{t,m} + \eta g_{t,m}$. Therefore, we have,

$$\sum_{t=1}^{T} \|\mathbf{y}_{t,m} - \mathbf{y}_{t-1,m}\| \leq \eta \sum_{t=1}^{T} \|g_{t,m}\|$$

Due to the equivalence of any two norms $\|\cdot\|_a, \|\cdot\|_b$, there exist L_1, L_2 such that $L_1\|x\|_a \leq \|x\|_b \leq L_2\|x\|_a$. Hence, for an arbitrary norm $\|.\|$, we may bound the term above by,

$$\sum_{t=1}^{T} \|\mathbf{y}_{t,m} - \mathbf{y}_{t-1,m}\| \leq \|w\|_\infty \sqrt{d_{max}} L_2 T \eta$$

Adding the two bounds together for each task one obtains,

$$\sum_{t=1}^{T} \tilde{f}_{t,m}(\mathbf{y}^*_m) - \tilde{f}_{t,m}(\mathbf{y}_{t,m}) \leq \frac{\|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2}{2\eta} + \frac{\eta T}{2} + \beta \|w\|_\infty \sqrt{d_{max}} L_2 T \eta$$

Now, we can once again take a closer look at the task averaged regret,

$$\bar{R}(T \times M) \leq \frac{1}{M} \sum_{m=1}^{M} \frac{\|\mathbf{y}_{1,m} - \mathbf{y}^*_m\|_2^2}{2\eta} + \frac{\eta T}{2}(1 + \beta \|w\|_\infty \sqrt{d_{max}} L_2)$$

As before, we introduce the best initialization in hindsight Φ^* to decompose the problem into the problem of computing the meta regret and the additional per task regret mediated by the task similarity. This yields,

$$\bar{R}(T \times M) \leq \frac{(D^*)^2}{\kappa} \frac{1 + \log M}{M} + \frac{\eta T}{2}(1 + \beta \|w\|_\infty \sqrt{d_{max}} L_2) + \frac{(D^*)^2}{2\eta}$$

Setting η to $\frac{D^*}{\sqrt{T(1+\beta\|w\|_\infty \sqrt{d_{max}} L_2)}}$ we have:

$$\bar{R}(T \times M) \leq \frac{(D^*)^2}{\kappa} \frac{1 + \log M}{M} + D^* \sqrt{T(1 + \beta \|w\|_\infty \sqrt{d_{max}} L_2)}$$

■

Remark: Once again we see a benefit of $\sqrt{2|\mathcal{J}|C}/D^*$ compared to an off the shelf algorithm. This factor plays an even more significant role in this context because not only do we reduce the scaling factor due to the number of caches $|\mathcal{J}|$ but in general the factor L_2 depends on the dimension of our problem. In our case, the dimension of our problem is the library size N. This means that the smoothed problem using the "off the shelf" strategy amplifies the effect of the number of caches by a factor that scales with the library size.

For completeness we include the following corollary of the theorem, note when we have a single task it is straightforward to extrapolate this result, similar results have been previously observed in [23] and [24].

Corollary 1: Algorithm 1 with a single task $M = 1$ incurs a $\mathcal{O}(\sqrt{T})$ regret for the smoothed problem.

V. Elastic Bipartite Caching and a Distributed Algorithm

The bipartite caching problem that we have described so far has what we call *inelastic* file demand, see [18]. A user request in this case is satisfied so long as she receives one file form the network. We now turn to the *elastic* case. Recall, from Section II an elastic request gains additional utility by gaining a bigger fraction of the file. Examples of such file requests can be found in streaming applications where a greater fraction of file chunks enable end users to see smoother videos. The utility becomes directly proportional to the number of file chunks received by the user. The routing vector in this setting is no longer constrained by $\sum_{j \in \mathcal{J}_i} z^{j,n}_{t,m} \leq 1$ where we explicitly avoid using $\{0\}$ to indicate the total requests from the network but not the main file server. The utility function remains the same, however, the lack of the constraints means we may set $z^{j,n}_{t,m} = y^{j,n}_{t,m}$ for any cache in the neighborhood of the user i, $j \in \mathcal{J}_i$ making the request at time t for task m.

The following result can be considered as a corollary to our main result. We remove the routing constraint $\sum_{j \in \mathcal{J}_i} z^{n,j}_{t,m} \leq 1$ from the inelastic setting. If $n = n_{t,m}$ is the file request at time slot t for task m, then, it is straightforward to find the routing policy $\mathbf{z}_{t,m}$:

$$\mathbf{z}^{j,n}_{t,m} = \begin{cases} \mathbf{y}^{j,n}_{t,m} & \text{when } n = n_{t,m} \text{ and } j \in \mathcal{J}_i \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

This means that the utility function can now be written as,

$$f_{t,m}(\mathbf{y}_{t,m}) = \langle \mathbf{r}_{t,m}, \mathbf{z}(\mathbf{y}_{t,m}, \{w^{i,j}\}) \rangle_{\mathbb{W}} = \langle \mathbf{r}_{t,m}, \mathbf{y}_{t,m} \rangle_{\mathbb{W}}$$

Note, even though z^j is not equal to y^j for all caches j, since r is only non zero for those caches in the neighborhood of i i.e, $\mathcal{J}_i$, it is not hard to see that the two quantities are equal. This observation makes the routing as well as the gradient computation simpler than the *inelastic setting*. It is not hard to see that such a result is true even under replacement costs.

The form described above is differentiable in the domain $\mathcal{Y}$ and concave. With a slight abuse of notation, let $\partial f_{t,m} = \mathbf{g}_{t,m}$ be the gradient at time slot t for task m. Then, if $n = n_{t,m}$ is the file requested at time t for task m, we can explicitly characterize the gradient as follows:

$$g^{\hat{n},j}_{t,m} = \begin{cases} w^{i,j} & \text{for } j \in \mathcal{J}_i \\ 0 & \text{otherwise} \end{cases}$$

The OGA update follows as:

$$\mathbf{y}_{t+1,m} := \Pi_{\mathcal{Y}}(\mathbf{y}_{t,m} + \eta \mathbf{g}_{t,m})$$

Now, it is important to note that the derivative $g^{\hat{n},j}_{t,m}$ is independent of $g^{\hat{n},\tilde{j}}_{t,m}$ for any two $j, \tilde{j}$ in $\mathcal{J}$. Further, the projection for each cache can also be distributed across each cache, giving us the following update to any cache j:

$$\mathbf{y}^j_{t+1,m} := \Pi_{\mathcal{Y}^j}(\mathbf{y}^j_{t,m} + \eta \mathbf{g}^j_{t,m}) \tag{11}$$

Hence, the OGA update for each cache is simply $g_{t,m}^{j,n} = w_{i,j}^n$ for $j \in \mathcal{J}_i$ and 0 otherwise. Further, note that the bounds on the gradient remain unchanged. This leads us to the following corollary

Corollary 2: Algorithm 1 with the distributed updates given by (11) gives a task averaged regret is bounded above as:

$$\bar{R}(T \times M) \leq \kappa \frac{1 + \log M}{M} + D^* \|w\|_\infty \sqrt{T d_{max}}$$

Remark: This is in contrast to the inelastic bipartite caching problem described above. Recall, $K_{t,m}$ was the least valuable cache that served an incoming request at (t, m). This cache either completes the request (described by Case II) or it offers all its chunks to fulfill as much of the request as possible. However, if each cache is given the weight of the least valuable cache for a file request, each cache may compute $w^{i,\theta(k)}$ according to Equation (8). This enables a distributed computation of the gradients under minimal central control.

VI. Simulations

We broadly break our simulations into four categories: *Bipartite caching*, *Single cache setting*, *Synthetic data traces* and *Real world data traces*. The first two settings pertain to the system model while the next two pertain to the user request generation. We will combine them and compare policies that are appropriate for these settings in order to highlight our algorithm. Before we proceed we briefly describe the data sets below.

Trace data set: For the purposes of making a fair comparison, we set our cache size similar to those of [2] with a single cache of size 900 with a library size of 3000, hence, we have a cache that is 30% of the library size. We use average hit rate as our metric of comparison since this is our primary interest. We use a window of 100 samples to smooth the data curves. We evaluate our policies on a number of data sets; YouTube request data [25], Movie lens data set for Movie ratings and IBM web search data requests. Note, we perform the meta initialization step once every 100 requests. This means that the algorithm is unaware of any inherent "task" structure in the data set and simply uses a partition of the data set in order to perform meta caching for the purposes of the trace based simulation. Further, we make no attempt to estimate the task similarity prior to the simulation, all estimates are made during the simulation run.

Synthetic data set: In this setting the data is generated using a zipf distribution with parameter $\alpha = 2.5 + W$. Here W is a uniform random variable between $[-d, d]$. By choosing an appropriate d we can control the similarity between our tasks. We set d to 1.25 to ensure that our sipf distribution does not diverge but can occasionally have a very high variance. Note, in the synthetic setting, the naïve OGD algorithm is also aware of the task structure and chooses its step-size to find the best policy in hindsight *per task*. In the following subsections simulation results for the two systems are shown.

Through our simulations we would like to answer the following questions:

1) So far, we assumed that D^* was a known quantity and the task structure was well defined. Can we simply divide the incoming requests into tasks and estimate the value of D^* using these tasks?
2) In a single cache setting, the value of D is relatively small. Hence, unless the user requests vary dramatically, the value of D/D^* is close to one in this setting; is there any benefit thatcomes from using our approach over a more naïve gradient ascent approach? We denote this approach throughout by the naïve OCO.
3) While Meta caching might lower meta regret, does this indeed translate to an improvement in the weighted hit rates of the caches? Further does this advantage remain if the task size is increased while keeping the number of tasks the same?
4) As a designer of an edge caching system, what practical considerations can we take based on the results we have obtained? Given the cache size is there an optimal way to divide the incoming tasks? How does cache size influence our performance?

Single Cache setting:

In the single cache setting we compare the following algorithms,

1) *Meta OCO:* This is the algorithm presented in the paper, we convert the problem of conventional online learning into a meta learning problem simply by dividing the data sets into batches of approximate size 100. Since we do not have access to D^* apriori in our Meta algorithm we begin with an estimate for $D^* = 2$. Next, we increase this estimate by multiplying a factor $\gamma = 1.05$ each time it fails to bound the difference between the best policy in hindsight for a given task and the initialization.
2) *naïve OCO:* We use conventional online gradient ascent (OGD) without assumptions on the underlying data structure [2] as a benchmark for performance in the coded caching setting.
3) *LFU:* The least frequently used policy evicts items in the cache that have been used the least so far. It is a well known algorithm that can be shown to be optimal when the distribution is stationary throughout a given trace. We use this benchmark to firstly show the efficacy of OCO policies to adapt to stationary request distributions over time. Secondly, a poor performance of an LFU policy indicates that the distribution of trace requests is non stationary.
4) *LRU:* The least recently used policy is popular policy that evicts the cache item which was used least recently. This policy performs is known to have excellent performance on real world data even though it suffers from poor adversarial guarantee.

Trace data set:

In the movie lens case and the web browsing case, LRU, naïve OCO and the meta algorithm all perform nearly identically however, the LFU policy performs poorly. This indicates that the distribution is highly non-stationary and it is optimal to simply discard all previous requests for a set of new requests. In the YouTube request setting we see that the meta algorithm dominates all the other algorithms while the naïve policy eventually reaches the performance of the LRU policy. We investigate the YouTube request set further in the single cache setting at the end of the simulation section.

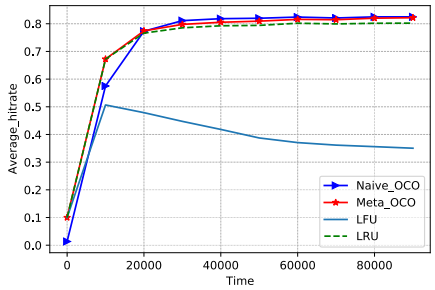


Fig. 2. Single cache movie lens rating.

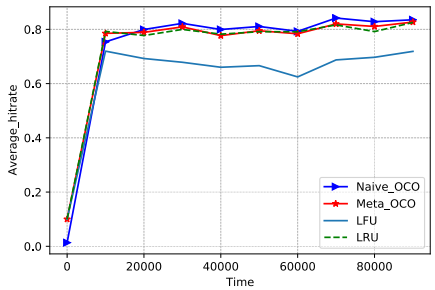


Fig. 3. Single cache IBM web browsing.

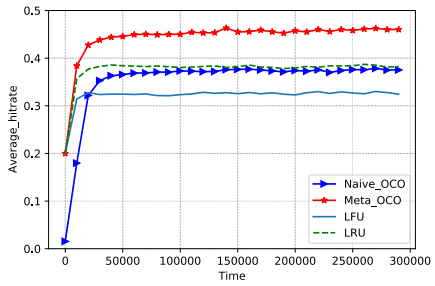


Fig. 4. Single cache YouTube data trace.

Synthetic data set: Our synthetic data results in fig (5a and 5) show the variation of performance in the single cache setting with respect to both the number of requests and the number of tasks.

Unsurprisingly, all the algorithms perform relatively well in this setting. Note, for a fixed set of requests, the meta algorithm performs slightly poorer than the naïve algorithm, possibly due to the extra $\frac{\log M}{M}$ term. As M grows large this difference fades away and both caches encounter the same hit-rate. In the single cache setting, we should note that our value of cache size is small and the weight is set to 1. This in turn means that unless the underlying distribution is varied wildly, the relative value of D/D^* will be close to 1. This explains not only the comparable performance of both the online learning algorithms but also the more conventional algorithms (LFU and LRU).

Bipartite Caching:

To begin with, we fix the bipartite graph as follows. We have 3 caches and 4 users. The weights for the three caches are $[1, 2, 100]$, this means that every item sourced from the third cache gives a utility of 100, similarly items sourced from the second cache give utility 2 and 1 for items from the first cache. Here we are trying to obtain the exact same setting as the one used in [2] so that we have a one to one comparison of our policies,

1) *Meta OCO:* Once again, the meta algorithm suggested in this paper is run without prior knowledge of the task similarity.
2) *Naïve OCO:* As before we simulate the online gradient ascent algorithm to maximize utility unaware of the task structure. We treat the sequential requests as a single long sequence.
3) *LRU:* We use an LRU policy to select which item to evict in our caches. Since, conventional mLRU [26] does not specify how to handle weights, we select the adjacent cache for eviction in proportion to its weights in the event of a miss. As mentioned previously, the LRU policy suffer from poor adverserial guarantees which might explain its relatively poor performance in this setting.
4) *LFU:* As stated above, we modify the LFU policy in the single cache setting to the Bipartite setting by selecting adjacent caches proportional to the weights in the event of a cache miss.

Trace Data: The trace data in figures 6, 7 and 8 was generated for a single cache setting, in order to convert the problem into the bipartite setting, we assign requests among the 4 users uniformly at random.

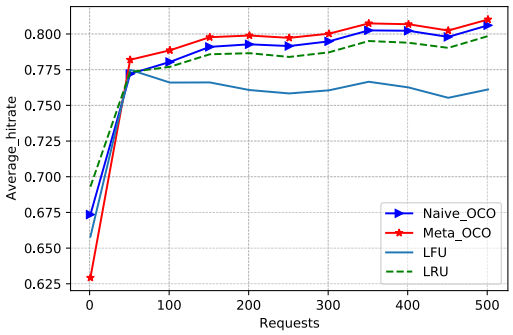
Comparing the results from the single caching setting, it is clear that both the online gradient algorithms perform far better than the more conventional counterparts. Further, one can look at the IBM web search results and the movie ratings data and see that unlike in the single caching setting, the meta algorithm has a noticeable improvement in performance compared to the naïve algorithm. This fact is largely explained by the *increased value of* D which makes the similarity between the tasks far more relevant.

Synthetic Data: The synthetic results for the bipartite caching are shown in figure 9a and 9b. Here we set the number of tasks to 100 and vary the number of requests per task for figure 9a and fix the number of requests to 20 and vary the number of tasks for figure 9b.

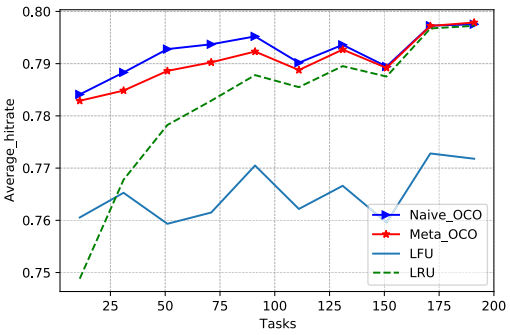
We compare our results to M-LRU and the best policy in hindsight. As can be seen, the meta algorithm quickly learns a good policy while the naïve OCO algorithm takes a relatively large number of requests for each task to catch up to the meta algorithm.

Practical Considerations:

In this final subsection, we provide some heuristics that might be helpful for cache design. As the real world trace data shows us, one may obtain superior performance simply by dividing the data into batches of equal size and running the meta algorithm. In a practical scenario, as seen in our trace simulations, one is able to control the task size easily. What is the best task size to select to see the greatest improvement in the performance? Another parameter we might be interested in is the cache size in terms of the library size. We show the variation of performance with changing task size for the Youtube data set in the single cache setting. We measure the average hit rate over all samples for a given tasks size. As seen in figure 10a, our X axis in this case is the number of tasks, since data length is fixed, our task size is inversely proportional



(a) Average hit rate as a function of requests



(b) Average hit rate as a function of tasks

Fig. 5. Synthetic data simulation for the single cache problem.

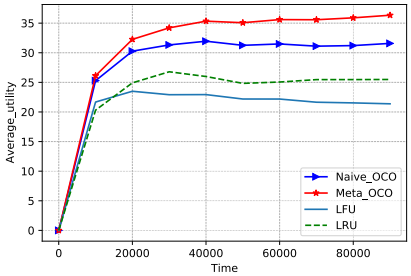


Fig. 6. Bipartite Caching - Movie ratings.

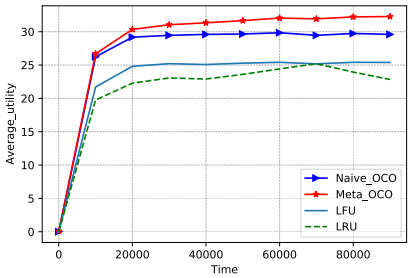


Fig. 7. Bipartite Caching Web browsing.

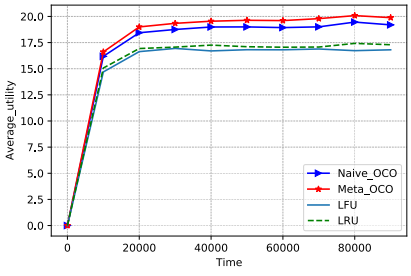


Fig. 8. Bipartite Caching YouTube traces.

to the number of tasks. Our data length is of the order of 10^5 samples. This means that when the number of samples per task is close to the cache size we see a sharp increase in performance followed by a slight decrease as we continue to decrease the task size. As the task size becomes smaller, the best in hindsight policy simply becomes the set of unique requests within a task. The slight decrease in performance may be explained by poor meta initialization updates combining the previous initialization with a configuration with a drastically different $L1$ norms. Finally, note, when the number of tasks is set to 1, our algorithm is indistinguishable from the naïve OCO algorithm. Next, we show the performance of our algorithm as a function of the cache size. The X axis in figure 10b indicates the cache size as a fraction of the library size. Starting at about 10% of the library size we increase the fraction all the way to 90%. As expected, all the caching algorithms improve as we increase the cache size, however, it is worth noting that for relatively small cache sizes, the meta algorithm far outperforms both the naïve OCO algorithm and the LFU policy. This benefit is most felt between $10-30\%$ of the library size and gradually decreases as we reach the $80-90\%$ range.

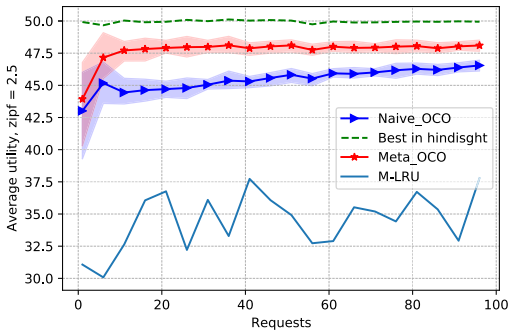
Further examination of the YouTube simulation results (Figure:4)

Finally, we end this section by investigating the results from the trace simulations seen in Figure 4 more closely.

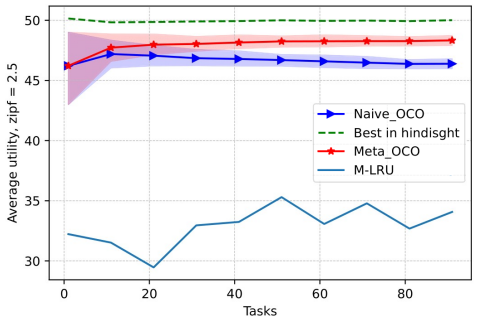
Varying Step-size in the Naive case: The YouTube dataset was collected in a university campus setting [25] around specific dates of college events. Hence, the value of D^* was relatively low across tasks due to highly correlated requests around the time frame. We believe that such trends in requests will become more significant with the rise in popularity of short form video content. This rapidly changing request pattern may also explain the performance of LRU close to the naïve OCO approach.

To substantiate this claim, in figure 11 we vary the step size of the naïve OCO algorithm under a variety of different conditions and compare it to the "meta" algorithm. We measure the average performance of all our variants for the complete data set for comparison. The "blue" line corresponds to a Naïve OCO algorithm with $\frac{c}{M\times T}$ step-size, here we vary the constant till the step size approaches 1. The "green" line corresponds to a naïve OCO algorithm with step-size $\frac{1}{t^{0.25}}$, which is the optimal step-size to achieve sub-linear regret in the setting of [19]. We refer the reader to our remark in the section I on why we believe this step-size is unable to compete with our meta-algorithm. Finally the "cyan" line corresponds to a step-size $\frac{1}{t^{0.33}}$ which was added as an additional point of comparison for a naïve OCO algorithm with time varying step size.

As seen in the figure, the meta-learning algorithm performs better than all these variants of the OGD algorithm. We believe that in this case, the "meta" initialization of our algorithm allows the cache to "reset" at regular intervals, which is not available to the other OCO algorithms. As stated in the description of our algorithm in the beginning of this section, we estimate the value of D^* over the tasks by multiplying

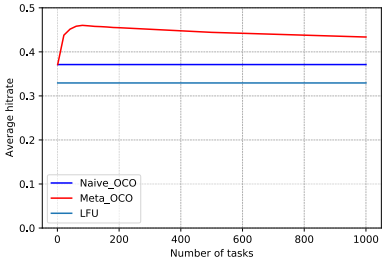


(a) Weighted utility vs requests per task

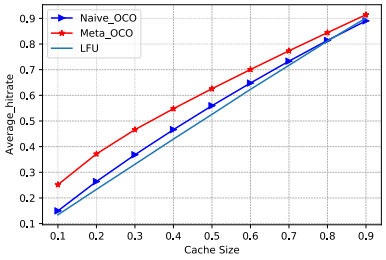


(b) Weighted utility vs tasks with fixed requests

Fig. 9. Synthetic data simulation for bipartite caching problem.



(a) Average hit rate vs number of tasks



(b) Average hit rate vs cache size

Fig. 10. Simulation figures for a few different design choices.

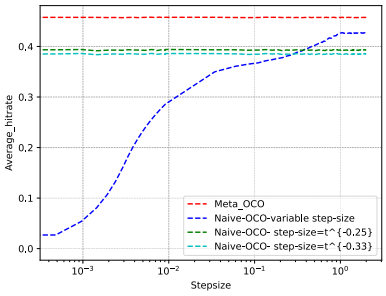


Fig. 11. Trace results for single cache YouTube data set with variations on step-size.

our previous estimate by $\gamma = 1.05$ every time our previous distance estimate falls short. For the trace data above, we are able to estimate a distance value of $D^* = 21.78$. Note, the value of $D = \sqrt{2C} \approx 42.42$. Hence, in this case D^*/D is roughly 0.5.

Resetting in meta initialization : We examine the effect of meta-initialization on the configuration. While our library contains 3000 files, we examine the file configuration of the first 50 files to make the change in configuration more legible to the reader. We also restrict our snapshot to the first few tasks

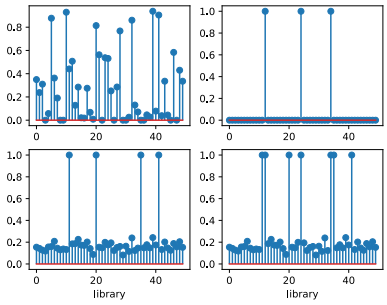


Fig. 12. Configuration of the first 50 files in cache. Top left: Naïve OCO algorithm, Top right: Task best in hindsight, Bottom left: Meta algorithm before initialization, Bottom Right: Meta algorithm after initialization.

so that our new initialization changes dramatically, recall, our step size $\alpha_m \propto \frac{1}{m}$ for the meta initialization process.

Figure 12 shows, the meta initialization in action. The meta policy chooses to "reset" its configuration by combining its configuration from the previous task with the best in hindsight policy. This allows it to adapt to adversarial requests when the next task changes the request sequence dramatically while the naive algorithm changes its own configuration "sluggishly" due to much smaller step size and lack of resets. Note, the many additional files that the naïve algorithm holds on to while the best in hindsight configuration completely discards them.

VII. Conclusion

In this paper, we propose a meta algorithm for caching. The algorithm uses the task structure in order to use a small number of samples to quickly adapt to a new incoming task. As we demonstrate in the simulations, this task structure need not be explicitly specified in the data set. Naïvely dividing the data set into batches is sufficient to apply our meta algorithm. While our paper thoroughly explores the idea of meta learning using task averaged regret in a variety of settings, it remains an open question how other ideas of task similarity can be used to improve caching performance. Of particular interest is [27] which explicitly looks at the problem of learning lower dimensional representations in the linear setting.

Acknowledgment

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

[1] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas.*, 2007, pp. 1–14.

[2] G. S. Paschos, A. Destounis, L. Vigneri, and G. Iosifidis, "Learning to cache with no regrets," in *Proc. IEEE Conf. Comput. Commun.*, Paris, France, Apr. 2019, pp. 235–243.

[3] N. Mhaisen, G. Iosifidis, and D. Leith, "Online caching with no regret: Optimistic learning via recommendations," 2022, *arXiv:2204.09345*.

[4] N. Mhaisen, A. Sinha, G. Paschos, and G. Iosifidis, "Optimistic no-regret algorithms for discrete caching," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 3, pp. 1–28, Dec. 2022.

[5] M. Khodak, M.-F. F. Balcan, and A. S. Talwalkar, "Adaptive gradient-based meta-learning methods," in *Proc. Annu. Conf. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Vancouver, BC, Canada: Curran Associates, Dec. 2019, pp. 5915–5926.

[6] M.-F. Balcan, M. Khodak, and A. Talwalkar, "Provable guarantees for gradient-based meta-learning," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2019, pp. 424–433.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[8] A. Bura, D. Rengarajan, D. Kalathil, S. Shakkottai, and J.-F. Chamberland, "Learning to cache and caching to learn: Regret analysis of caching algorithms," *IEEE/ACM Trans. Netw.*, vol. 30, no. 1, pp. 18–31, Feb. 2022.

[9] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 7, pp. 1305–1314, Sep. 2002.

[10] N. Gast and B. Van Houdt, "Asymptotically exact TTL-approximations of the cache replacement algorithms LRU(m) and h-LRU," in *Proc. 28th Int. Teletraffic Congr.*, vol. 1, Sep. 2016, pp. 157–165.

[11] D. S. Berger, P. Gland, S. Singla, and F. Ciucu, "Exact analysis of TTL cache networks," *Perform. Eval.*, vol. 79, pp. 2–23, Sep. 2014.

[12] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Upper Saddle River, NJ, USA: Prentice-Hall, 1973.

[13] J. Li, S. Shakkottai, J. C. S. Lui, and V. Subramanian, "Accurate learning or fast mixing? Dynamic adaptability of caching algorithms," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 6, pp. 1314–1330, Jun. 2018.

[14] M. M. Amble, P. Parag, S. Shakkottai, and L. Ying, "Content-aware caching and traffic management in content distribution networks," in *Proc. 30th IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Shanghai, China, 2011, pp. 2858–2866.

[15] N. Abedini and S. Shakkottai, "Content caching and scheduling in wireless networks with elastic and inelastic traffic," *IEEE/ACM Trans. Netw.*, vol. 22, no. 3, pp. 864–874, Jun. 2014.

[16] G. S. Paschos, A. Destounis, and G. Iosifidis, "Online convex optimization for caching networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 625–638, Apr. 2020.

[17] T. Si Salem, G. Neglia, and S. Ioannidis, "No-regret caching via online mirror descent," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2021, pp. 1–6.

[18] R. Bhattacharjee, S. Banerjee, and A. Sinha, "Fundamental limits on the regret of online network-caching," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 48, no. 1, pp. 15–16, Jul. 2020.

[19] S. Zhou et al., "Caching in dynamic environments: A near-optimal online learning approach," *IEEE Trans. Multimedia*, vol. 25, pp. 792–804, 2023.

[20] S. Shalev-shwartz and S. M. Kakade, "Mind the duality gap: Logarithmic regret algorithms for online optimization," in *Proc. 22nd Annu. Conf. Adv. Neural Inf. Process. Syst.*, vol. 21, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds., Vancouver, BC, Canada: Curran Associates, Dec. 2008, pp. 1–8.

[21] E. Hazan, *Introduction to Online Convex Optimization*. Cambridge, MA, USA: MIT Press, 2022.

[22] N. Chen, J. Comden, Z. Liu, A. Gandhi, and A. Wierman, "Using predictions in online optimization: Looking forward with an eye on the past," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Sci.*, 2016.

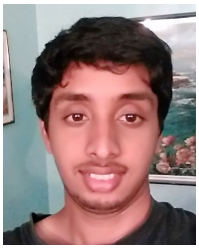
[23] J. Comden, S. Yao, N. Chen, H. Xing, and Z. Liu, "Online optimization in cloud resource provisioning: Predictions, regrets, and algorithms," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 47, no. 1, pp. 47–48, Dec. 2019.

[24] L. Zhang, W. Jiang, S. Lu, and T. Yang, "Revisiting smoothed online learning," in *Proc. Neural Inf. Process. Syst.*, 2021, pp. 13599–13612.

[25] M. Zink, K. Suh, Y. Gu, and J. Kurose, "Watch global, cache local: YouTube network traffic at a campus network: Measurements and implications," in *Proc. SPIE*, vol. 6818, Jan. 2008, Art. no. 681805.

[26] A. Giovanidis and A. Avranas, "Spatial multi-LRU: Distributed caching for wireless networks with coverage overlaps," 2016, *arXiv:1612.04363*.

[27] N. Tripuraneni, C. Jin, and M. Jordan, "Provable meta-learning of linear representations," in *Proc. 38th Int. Conf. Mach. Learn.*, vol. 139, M. Meila and T. Zhang, Eds., Jul. 2021, pp. 10434–10443.



Dheeraj Narasimha received the Ph.D. degree from Arizona State University in 2021. He was a Post-Doctoral Scholar with Texas A&M University. He is currently a Post-Doctoral Researcher with INRIA, Grenoble. His research interests include restless multi-armed bandits, Markov decision processes, mean field games, online learning, and graph processes.

Dileep Kalathil (Senior Member, IEEE) received the Ph.D. degree from the University of Southern California in 2014. He was a Post-Doctoral Scholar with the University of California at Berkeley till 2017. He joined Texas A&M University in 2017, where he is currently an Associate Professor of computer engineering with the Department of Electrical and Computer Engineering. Together with Srinivas Shakkottai, he co-directs the Learning and Emerging Networked Systems (LENS) Laboratory. His research interests include reinforcement learning, control theory, game theory and their applications in intelligent transportation systems, renewable energy systems, and cyber-physical systems. He has received the NSF CRII Award as well as the NSF Career Award in 2021.

Srinivas Shakkottai (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana–Champaign in 2007. He was a Post-Doctoral Scholar in management science and engineering with Stanford University in 2007. He joined Texas A&M University in 2008, where he is currently a Professor of computer engineering with the Department of Electrical and Computer Engineering. His research interests include caching and content distribution, wireless networks, learning and game theory, as well as network data collection and analytics. He was a recipient of the Defense Threat Reduction Agency Young Investigator Award in 2009 and the NSF Career Award in 2012. He received research awards from Cisco in 2008 and Google in 2010. He received an Outstanding Professor Award in 2013, the Select Young Faculty Fellowship in 2014, and the Engineering Genesis Award from Texas A&M University in 2019.