IEEE ICCC 2023 1570914470

Edge Caching Based on Deep Reinforcement Learning

1st Farnaz Niknia
Department of Electrical Engineering and Computer Science
York University
Toronto, Canada
fniknia@yorku.ca

2nd Ping Wang
Department of Electrical Engineering and Computer Science
York University
Toronto, Canada
ping.wang@lassonde.yorku.ca

3rd Aakash Agarwal
Electronic and Communication Engineering
IIIT-Naya Raipur
Naya Raipur, India
aakash19101@iiitnr.edu.in

4th Zixu Wang
School of Computer Science and Engineering
South China University of Technology
Guangdong, China
2571493168@qq.com

2023 IEEE/CIC International Conference on Communications in China (ICCC) | 979-8-3503-4538-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCC57788.2023.10233394

***Abstract*—The use of mobile apps has caused a lot of data to be transmitted repeatedly, which has put a strain on the networks. One solution to this problem is caching, which stores data closer to the user to reduce data transfer and delay. Typically, when designing a caching policy, only a limited set of file features, such as freshness and popularity, are taken into account in the majority of prior studies. Nevertheless, it is crucial to incorporate factors like importance and size since highly valuable files may hold more significance compared to freely accessible ones. In this paper, we modeled caching problem using Semi-Markov Decision Process that considers file features such as popularity, lifetime, size, and importance. Then, we used a Reinforcement Learning algorithm called Double Deep Q-Learning to solve the Semi-Markov Decision Process. Simulation results show that the proposed method outperforms an existing caching method in terms of cache hit rate and total utility in various settings. This method is the first to comprehensively consider all file features, making it more practical for real-world scenarios.**

***Index Terms*—Reinforcement Learning, Caching, Semi-Markov Decision Process**

I. Introduction

The rapid growth of mobile device applications has led to a significant increase in redundant data transmission, as a large number of users access contents from a data center. This surge in traffic has placed a heavy burden on backhaul links and backbone networks [1]. Caching at the edge router has become a promising solution for mitigating redundant traffic transfer and decreasing transmission delay, as evidenced by several recent studies [2]. Generally speaking, caching methods exploit the history of requested contents to detect popular files and cache them. Existing caching schemes can be categorized as reactive and proactive [3] [4]. Reactive caching decides whether or not to cache a file after it has been requested [5] [1] [6] [7]. Proactive methods, however, utilize historical data to predict the future popularity of files and cache them in advance even when they are not requested [8] [9] [10]. The major shortcoming of proactive caching is that it may lead to a low cache-hit ratio if the cached content has a short lifetime. Furthermore, caching content that might not be accessed in advance requires storage space, which can only be obtained by removing previously cached content [3]. Thus, reactive methods seem to be a better option for this problem.

The caching problem can be modeled using the Markov Decision Processes (MDPs) framework [11] [12], which involves sequential decision-making by an agent to determine a policy that maximizes the total expected reward over time. Given the model of the environment (reward function and transition probabilities), the MDP can be solved using Bellman optimality equations [13]. In most real-world problems, however, the model of the environment is not available beforehand. Therefore, model-free Reinforcement Learning (RL) schemes are used, enabling the agent to learn by trial and error without relying on statistical knowledge of the environment [13] [14].

Most existing work uses MDP to model wireless caching problem [15] [16]. However, in an MDP formulation, caching decisions are made at the beginning of equal time intervals [12]. Since caching decisions are made upon a request's arrival, there must be a new request at the beginning of each time interval. However, in real-world scenarios, requests may arrive at the edge router at random times. Semi-Markov Decision Process (SMDP) [11] [12] [13] is a better option for this kind of problem that the transition between states happen at a non-constant time interval. Similar to MDP, SMDP is a mathematical framework used to model decision-making problems but the difference is that it is more general than MDP since in an SMDP the state transition times may be fixed, exponentially distributed, or totally random [11]. For more explanation, MDP is a special case of SMDP where state transition times are equal. The proposed approaches in [17] is an example of exploiting SMDP for wireless caching.

Furthermore, current caching policies tend to focus on specific content features and overlook other significant aspects like size and importance. Considering the fact that critical

files may possess higher value compared to freely available ones, it becomes essential to include the importance of files in caching strategies. Additionally, characteristics such as size and lifetime play crucial roles in caching policies. For instance, a widely popular movie with a short lifespan and large file size might not be a suitable candidate for caching if it necessitates removing multiple files from the cache, as it could expire before any requests for that specific file are fulfilled.

Motivated by all of these, in this paper, we proposed a reactive caching scenario using Semi-Markov Decision Process (SMDP) [11] [12], which is capable of making decisions at any random time that a request for a content arrives at the edge router. The reason our work in this paper is included in the reactive category is that caching decisions are made after a file is requested. Our contributions in this paper can be summarized as follows: (1) Firstly, we modeled the caching problem by SMDP to capture the continuous-time nature of real-world scenarios such as request arrivals. Then, we proposed a discounted reward Double Deep Q-Learning (DDQL)-based caching method which utilizes the past popularity data to shape a caching policy while considering future states of the system and uncertainties. (2) In addition to popularity, we considered various features for files such as lifetime, size, and importance. To the best of our knowledge, this is the first caching method that comprehensively considers all these features of files in shaping its caching policy, making our method more practical for real-world scenarios. (3) We propose to assign a utility value to the cached files and use it to decide which file to remove when the cache is full. In this way, we can manage the cache utilization efficiently. (4) We simulated the problem and investigated the performance of our proposed method in terms of cache hit rate and total utility under different settings. We compare our method with a recent DRL-based caching method [6] (to the best of our knowledge, this is the most suitable existing work for comparison). The experimental results show that our proposed method outperforms the existing DRL-based caching in all experiments.

In the following, we will briefly review the literature in the section II. Then, we will present our system model in section III and propose a method in section IV. Finally, we will investigate the parameter settings and represent numerical results in V.

II. Related Work

Generally speaking, caching approaches fall into two main categories: proactive and reactive. In the following, we will review a number of existing approaches for each category and investigate their possible shortcomings.

A. Reactive caching

A case study is presented in [5], where authors first assign a priority to each file based on their "virality", and then, solve an optimization problem to minimize the average content access delay. However, solving an optimization problem at the end of each time slot results in both time and computational overheads. In [1] a DRL-based agent is designed with the aim of maximizing the caching benefit rate. The benefit rate is calculated using latency and the content request frequency. When a new request arrives at the core network, its popularity is predicted by the recommendation system-based content popularity prediction model. The authors in [6] proposed an actor-critic DRL-based approach that maximizes a utility function. In [7], authors deployed a version of the actor-critic algorithm named Proximal Policy Optimization (PPO) to maximize hit rate while minimizing energy consumption.

B. Proactive caching

The authors of [8] addressed the slow start issue in proactive caching, which refers to the time needed to collect request history for each new content. They use Euclidean distance to determine the similarity of new files with existing files, assuming a new file is popular if it's similar to a popular file. However, this approach results in heavy computational overhead as the Euclidean distance of new files must be calculated with all existing files. The proposed work in [9] not only supports service differentiation but also achieves convergence to the optimal cache content placement strategy by maximizing cache hits. The authors in [10] incorporated a dynamic k-nearest neighbor algorithm into the actor-critic algorithm, enabling the DRL algorithm to operate in an action space of varying size.

III. System Model

In this section, we first describe the system model for caching problem including a data center, edge router's cache, end users, file types, and their characteristics. Then, we describe the dynamics of user request arrivals and file expiration.

A. System architecture

In this paper, we consider a general network scenario where the end users access the Internet through an edge router and the edge router connects the data center on one side and the end users on the other side.

1) Data center: The cloud data center is assumed to have sufficient space to store all contents [8]. A copy of a file will be created upon receiving a request from an end user. The requested copy will be sent to the user via the edge router through the Internet network.

2) Edge router's cache: Edge router has a limited capacity of M to cache files. If the requested file f_r is already cached in the edge router, the current request will be fulfilled by a copy of f_r from the edge router rather than the data center. In the rest of this paper, we will use "cache" as a short form of "edge router's cache".

3) End users and file requests: End users are devices that request files from the data center based on their needs and preferences [8]. $\mathcal{U} = \{u_1, u_2, u_3, \ldots, u_u, \ldots, u_U\}$ is a set of users 1 to U connected to the edge router. we denote requests by $\mathcal{G} = \{g_1, g_2, g_3, \ldots, g_g, \ldots, g_G\}$ where g_g is the g^{th} request regardless of which user generated it. These requests are responded to in the order that they have been created.

4) File types and characteristics: Files are contents generated by a variety of sources (like cameras, sensors, and computers) and stored in the data center. We denote files by $\mathcal{F} = \{f_1, f_2, f_3, \ldots, f_f, \ldots, f_F\}$ where f_f refers to the f^{th} type of file. Due to the underlying structure and intent to use, each file type may have distinct characteristics such as popularity [18], lifetime [19], size, and importance. In the following, we will provide further explanations and real-life examples for each one:

Popularity: refers to the number of times a file may be requested by users [18]. The popularity of files can vary due to several reasons such as user demand and preferences. For example, popular video files, such as recent movies or trending videos, may have a higher demand compared to less popular files, such as old content.

Lifetime: is a value that indicates how long the file is valid for use from the time of generation [6]. For instance, many mobile user applications, such as location-based services require regular and timely location updates to be delivered to application servers [20]. Upon generating a file, its lifetime is determined and stored along with the file.

Size: Depending on type and content, files may have different sizes. For example, a file containing a movie will likely be larger than a text file.

Importance: The importance of a file indicates how valuable a file is to users. It may depend on several factors such as content, purpose, and availability. Popular movies or online courses are examples of valuable files while free to all contents e.g. free ebooks, music, or videos, have less value for users to pay for them. In this paper, we considered that importance is a fixed value for each file type and as the same as lifetime and size, it is known when the file is cached.

It is important for a caching policy to take all the features of files into account simultaneously rather than focusing on only one or ignoring the others. For instance, the most popular file may have the least importance in the network. For example, trending music gains tremendous popularity and dominates the music charts, while the emergency broadcast is significantly more important for the safety and well-being of people. Thus, solely focusing on popularity may not meet all requirements. Another scenario is when a highly popular file with a short lifetime, large size, and low importance is cached, which may not be as beneficial as caching a less popular file with a longer lifetime, smaller size, and higher importance. In this paper, we consider all four characteristics of each file. $\boldsymbol{c} = (c_1, c_2, c_3, \ldots, c_f, \ldots, c_F)$ represents a popularity value for each type, e.g. c_f is the popularity of file type *f*. In other words, c_f shows the number of times that the corresponding file type f is requested in T. Popularities follow a Zipf distribution with the parameter η. The popularity of file type f is [21]:

$$f_f = \frac{\frac{1}{f.\eta}}{\sum_{\omega=1}^{F} \frac{1}{\omega.\eta}},\ 1 \leq f \leq F \tag{1}$$

We denote the lifetime, size, and importance of files by $\boldsymbol{l} = (l_1, l_2, l_3, \ldots, l_f, \ldots, l_F)$, $\boldsymbol{z} = (z_1, z_2, z_3, \ldots, z_f, \ldots, z_F)$ and $\boldsymbol{i} = (i_1, i_2, i_3, \ldots, i_f, \ldots, i_F)$, respectively.

5) Utility of cached files: For each file in the cache, we assign a value called utility. The utility of a file is a function of its freshness and importance. In this work, freshness is the age of a file normalized by its lifetime at time *t*:

$$h^f(t) = (t - w_g^f)/w_l^f, \quad 0 \leq h^f(t) \leq 1 \tag{2}$$

where t is the current time, w_l^f and w_g^f represent the lifetime and generation time of file type *f* respectively. $h^f(t)$ indicates the freshness of file f at time *t*. We utilized a non-linear function to obtain the utility of file type *f* at time *t*:

$$y_f(t) = (-e^{\left(h^f(t) + \log(curve)\right)} + UT_{\max} + curve).i_f \tag{3}$$

$y_f(t)$ is the utility of file type f and Curvature (curve) is calculated as follows:

$$curve = (UT_{\max} - UT_{\min}) \ / \ (e - 1) \tag{4}$$

Where UT_{max} and UT_min denote the maximum and minimum value of utilities.

B. System uncertainties

Caching introduces uncertainties such as random request arrivals and unknown future impact of requests on the cache. These uncertainties affect the edge router's decision-making, and it's crucial to consider their potential future impact. Therefore, the edge router should be able to adapt its decisions to the dynamic nature of real-world caching systems. The following sections will explain each of these uncertainties and the assumptions made.

1) Random inter-arrivals of requests: Requests for content in real-world scenarios can arrive randomly due to several factors, such as user behavior and network conditions. We use a Poisson process to model the arrival of requests, which is a widely adopted model that can well characterize the user behaviors of generating requests [22]. It's assumed that the edge router has no prior knowledge of this Poisson process or its parameter, e.g., the average request rate (η).

2) Unknown effect of upcoming requests on the cache: The system consists of F types of files, each with unique characteristics such as popularity, lifetime, size, and importance. Caching a file may have an unknown impact on subsequent requests since there is no prior knowledge of upcoming content requests. Caching a large file may necessitate evicting several other files from the cache, directly impacting future hit rates. Conversely, caching a highly popular file can increase the hit rate and help fulfill more requests from the edge router.

Uncertainties have a significant impact on the performance of any caching policy. In the next section, we explain how our caching policy considers these uncertainties and propose an approach to optimize the hit rate in the long run.

IV. The DDQL-based Caching

The decision-making problem of caching files on an edge router can be modeled using the Markov Decision Processes (MDP) framework [11], which includes five components: state, action, system dynamics, reward function, and policy.

However, since the caching problem involves continuous-time arrivals of requests and non-exponential durations for state transitions, the Semi-Markov Decision Processes framework is proposed as a more suitable model [11]. In the following, we elaborate on the SMDP formalism and its elements.

A. SMDP formalism

a semi-Markov decision process is defined by a tuple $(\mathcal{S}, \mathcal{A}, J, R, \pi)$ where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. *J*, *R*, and π are the transition probabilities, reward function, and policy, respectively.

1) States of the System: the system state at time t is denoted by $s(t) \in \mathcal{S}$ which contains five components:

$$s(t) = \{Mem(t), \boldsymbol{d}(t), \boldsymbol{b}(t), \boldsymbol{b}(t) \circ \boldsymbol{y}(t), \boldsymbol{b}(t) \circ \boldsymbol{z}(t)\}, \quad (5)$$

$Mem(t)$ indicates the unoccupied portion of cache memory and the remaining components reflect the popularity, utility, and sizes of files in the cache, respectively. ***Mem(t)*** is defined as:

$$Mem(t) = \left(M - \sum_{f=1}^{F} \boldsymbol{B}(t) \circ \boldsymbol{Z}(t) \right) / M, Mem(t) \in \mathcal{M} \quad (6)$$

$\mathcal{M}$ is a set of all possible values for ***Mem(t)***. The vector $\boldsymbol{d(t)} = (d_1(t), d_2(t), d_3(t), \ldots, d_f(t), \ldots, d_F(t)), \boldsymbol{d}(t) \in \mathcal{D}$ is the number of times that each file type has been requested in recent *N* requests, while the vector, $\boldsymbol{y(t)} = (y_1(t), y_2(t), y_3(t), \ldots, y_f(t), \ldots, y_F(t)), \boldsymbol{y}(t) \in \mathcal{Y}$, indicates the utility values for each file type and $\boldsymbol{z(t)} = (z_1(t), z_2(t), z_3(t), \ldots, z_f(t), \ldots, z_F(t)), \boldsymbol{z}(t) \in \mathcal{Z}$, is the set of sizes of all file types. The vector $\boldsymbol{b(t)} = (b_1(t), b_2(t), b_3(t), \ldots, b_f(t), \ldots, b_F(t)), \boldsymbol{b}(t) \in \mathcal{B} = \{0, 1\}^F$, is a binary vector, where a value of 0 indicates that a file is not cached and a value of 1 indicates that the file is already cached. $\mathcal{D}, \mathcal{Y}, \mathcal{Z}$ and $\mathcal{B}$ are sets of all possible for ***d***, ***y***, ***z***, and ***b***. The symbol $\circ$ represents the Hadamard product of matrices, also known as the element-wise product [23].

2) Actions: if the agent decides to cache the requested file, $a(t)$ will be set to 1, otherwise, it is set to 0. $A = \{0, 1\}$ represents all of the possible actions that can be taken in all states.

The caching problem may encounter situations where the cache is full and cannot store additional data. In such cases, the file with the lowest utility is removed first, and if adequate space is still not available, the next file with the lowest utility is removed, and this process continues until enough space is made available in the cache.

3) Dynamics of the System: In SMDP formulation, since the transition times are variant, there is also a distribution over how long it will take to transit to the next state. See [14] for more details.

4) Instant reward and the long-term goal: we define the instant reward as follows:

$$r(t) = \left((\boldsymbol{b}(t) \circ \boldsymbol{d}(t)) \ (\boldsymbol{b}(t) \circ \boldsymbol{y}(t))^T \right) - \ Mem(t) * 100 \quad (7)$$

The first term is the weighted utility of the already cached files, i.e., the utility of each cached file is multiplied by its popularity within past *N* trials. As explained above, the second term refers to the unused portion of the cache. In simpler terms, we determine the worth of each file in the cache by multiplying its popularity over the last *N* attempts with its current utility. Then, we subtract the unused portion of the cache from the overall value. The rationale behind this is to encourage the agent to maximize the utility of the cache capacity in order to help maximize the hit rate. According to the reward function, the long-term system goal is to cache files in a way that the average accumulated worth of files in each trial is maximized in the long run.

5) The policy: determines which action should be selected in each system state in order to reach the long-term goal.

B. The Double Deep Q-Learning algorithm (DDQL)

Double-deep Q-learning is a reinforcement learning algorithm used to learn Q-values for each (s, a) in order to shape a decision-making policy. Using this algorithm, the agent takes an action in each state following the policy generated by DDQL, then transits to a new state and receives reward r. This is called a transition and it is shown with tuple $< s, a, s', r, \eta >$. When a transition is completed, the corresponding tuple will be stored in a replay buffer with size Λ. If the buffer is full, the oldest existing tuple will be replaced with the newly generated one. Then, a batch of transitions is randomly sampled to update a neural network called Q-network. The output of this neural network is the approximated Q-value of (s, a) which is denoted by $\hat{Q}(s_k, a_k)$. Then, Q(*s, a*) is calculated using the most recent observation (transition):

$$Q(s,\ a) = r + \ \gamma^{\tau} Q\left(s',\ \underset{a' \ \epsilon \ \mathcal{A}}{\operatorname{argmax}} Q(s',\ a';\ \theta)\ ;\ \theta'\right) \quad (8)$$

where $\operatorname{argmax}_{a' \ \epsilon \ \mathcal{A}} Q(s',\ a';\ \theta)$ gives a' which is the action with maximum Q-value in state s'. $\operatorname{argmax}_{a' \ \epsilon \ \mathcal{A}} Q(s',\ a';\ \theta)$ is obtained by Q-network with weight vector $\boldsymbol{\theta}$. Also, $Q(s',\ \operatorname{argmax}_{b \ \epsilon \ \mathcal{A}} Q(s',\ b;\ \theta);\ \theta')$ is the Q-value of action a' in state s' obtained by another network called target-network with weight vector $\boldsymbol{\theta}'$. Target network is a copy of Q-network and it is updated every ζ time steps by copying $\boldsymbol{\theta}$ into $\boldsymbol{\theta}'$.

The Q-network minimizes a Mean Squared Error loss (MSE) which results in updating $\boldsymbol{\theta}$ as follows:

$$\theta_{k+1} = \theta_k + \psi \left(Q_{k+1}(s_k, a_k) - \hat{Q}(s_k, a_k) \right) \phi(s_k) \quad (9)$$

$\phi(s_k)$ represents the state features, *k* and $\psi/2$ are the trial number and the gradient step size.

V. Simulation settings and experimental results

In this section, we first report our setup for experiments and the tools that we used to implement the whole experiment. Then, we compare simulation results with existing work in terms of performance metrics and discuss the results.

A. Simulation platform

We used Python 3 programming language to simulate our system model and train the DRL agent. To facilitate neural network implementation, training, and exploitation, we utilized Tensorflow [24] platform.

B. Parameters and settings

We implemented the DRL with a neural network containing 2 layers and 16 nodes in the hidden layer. The weights of the neural networks are randomly initialized to values between -0.1 and 0.1 with a bias equal to 0.1. On top of that, ReLU [14] is used as the activation function. We considered 50 file types and a catch with a capacity of 10000. Lifetimes, importance values, and sizes of files are randomly generated from [10, 30], [0.1, 0.9], and [100, 1000], respectively. Moreover, the batch size is 64, UT_{max} and $UT_{constant}$ are 1.5 and 0.1. ζ also is set to 100.

C. Baseline

To the best of our knowledge, there is no existing work that takes all four features, i.e., popularity, lifetime, importance, and size, into account in shaping their caching policy. Therefore, we proposed an algorithm for caching issue with [6] where authors take different popularities and freshness of files into account. We believe this is a recent work which is most relevant to ours. We refer to [6] as Caching Transient Data (CTD) in the result comparison.

D. Performance metrics

We evaluate our proposed approach in terms of:

1) **Hit count:** Once the policy has converged, we evaluate its performance by testing it on 1000 additional user requests. The number of times that the user was served with a file from the cache is defined as the hit count. The goal of any caching policy is maximizing the hit count.
2) **Total utility:** If the requested file is in the cache, its current utility will be added to the total utility. A higher total utility is desired since it indicates that the corresponding approach keeps files with higher utilities in the cache.

E. Results

In this section, we represent a comparison of our simulation results with the existing approach [6] in terms of total utility and hit count.

1) Impact of different popularity rates: Fig.1a and Fig.2a show the total utility and hit counts out of 1000 trials for varying Zipf parameters (η) respectively. When η is close to 0 the files have close popularity and when η is close to 1, the distribution is highly skewed, with a few items being very popular and the rest having low popularity. In this case, caching the most popular files boost the hit count. For this reason, increasing η from 0 to 1 results in an increase in the hit count. In addition to this, when η is close to 1, there is a higher probability to cache a fresher version of files, thus, the total utility increases when η gets larger.

2) Impact of different request arrival rate: Fig.1b and Fig.2b represent the total utility and hit counts for different inter-arrival times respectively. λ represents the average request rate. A higher value of λ results in a shorter inter-arrival time and higher hit counts. Shorter inter-arrival times lead to a higher probability of requested files being in the cache before they expire.

3) Impact of different cache sizes: A larger cache size allows for more files to be cached, resulting in a higher hit count and fulfilling more requests from the cache rather than the data center. Comparing Figs 2c and 1c, we can observe that an increase in the hit count does not lead to a higher total utility. This is because when a file remains in the cache for an extended period, requests for that file are fulfilled from the cache, leading to a higher hit count. However, this prolonged stay in the cache may not result in higher utility.

Based on experimental results, our proposed approach outperforms [6] for several reasons. First, the DRL agent in [6] receives an instant reward only based on the availability and freshness of the requested file in the cache. On the contrary, our proposed approach comprehensively considers various features for files, including popularity, utility, freshness, and size of all cached files, in the instant rewards.

Second, the reward function in [6] only focuses on the current requested file rather than considering all previously cached files. This is important since the other cached files are the results of previous decisions, thus, their current status is a delayed reward for previous actions. Since the DRL agent is foresighted (it takes the future state of the system into account), considering the delayed reward helps the agent to obtain a better vision of the impact of current actions on the future of the system.

Moreover, the state definition in [6] includes the number of requests for currently cached files but it is insufficient to encourage the agent to cache popular files. The proposed approach addresses this issue by including the request history of files in the reward function.

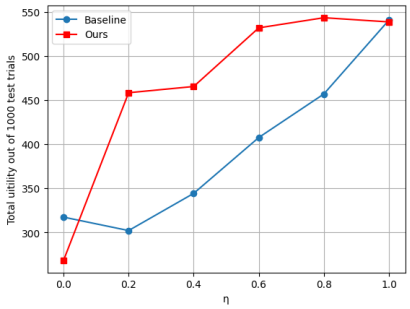
VI. Conclusion

In this paper, we modeled the caching problem using SMDP and proposed a DDQL-based method to capture the continuous-time nature of request arrivals, considering various file features such as lifetime, size, and importance. Moreover, we simulated our proposed caching method and compare it with an existing DRL-based caching method. The experimental results showed that our method outperforms the existing caching approach in terms of cache hit count and total utility. Overall, the proposed method shows promising results for real-world caching scenarios.

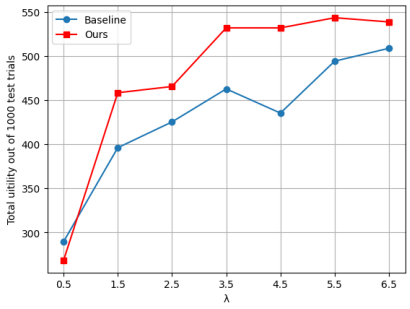
References

[1] Y. Liu, J. Jia, J. Cai, and T. Huang, "Deep reinforcement learning for reactive content caching with predicted content popularity in three-tier wireless networks," *IEEE Transactions on Network and Service Management*, 2022.

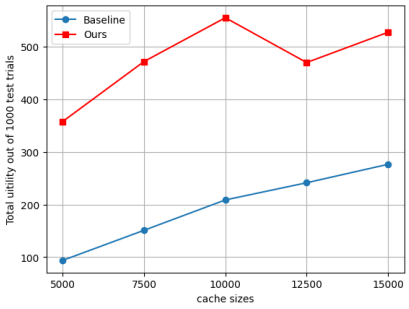
[2] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, IEEE, 2010.



(a) Total utility for different values for η

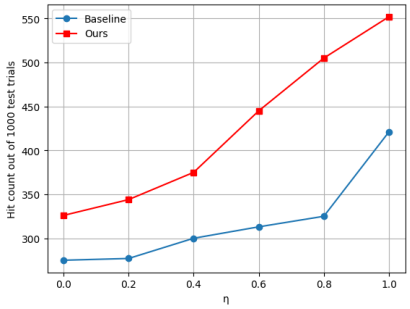


(b) Total utility for different values for λ

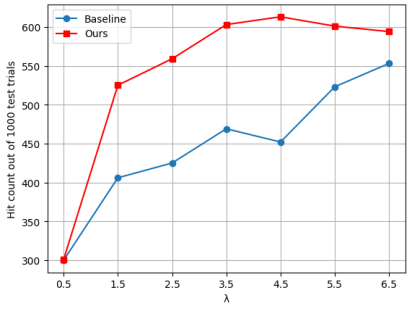


(c) Total utility for different cache sizes

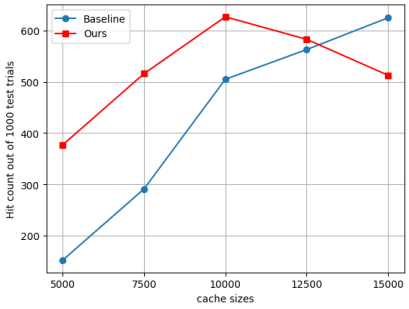
Fig. 1: Total utility for a) different values for η, b) different values for λ c) different cache sizes.



(a) Hit count for different values for η



(b) Hit counts for different values for λ



(c) Hit counts for different cache sizes

Fig. 2: Hit count for a) different values for η, b) different values for λ c) different cache sizes.

[3] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in edge networks: A comprehensive survey," *arXiv preprint arXiv:2006.16864*, 2020.

[4] H. Ahlehagh and S. Dey, "Video-aware scheduling and caching in the radio access network," *IEEE/ACM Transactions on networking*, vol. 22, no. 5, pp. 1444–1462, 2014.

[5] S. Ahangary, H. Chitsaz, M. J. Sobouti, A. H. Mohajerzadeh, M. H. Yaghmaee, and H. Ahmadi, "Reactive caching of viral content in 5g networks," in *2020 3rd International Conference on Advanced Communication Technologies and Networking (CommNet)*, pp. 1–7, IEEE, 2020.

[6] H. Zhu, Y. Cao, X. Wei, W. Wang, T. Jiang, and S. Jin, "Caching transient data for internet of things: A deep reinforcement learning approach," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2074–2083, 2018.

[7] H. Wu, A. Nasehzadeh, and P. Wang, "A deep reinforcement learning-based caching strategy for iot networks with transient data," *IEEE Transactions on Vehicular Technology*, vol. 71, no. 12, pp. 13310–13319, 2022.

[8] X. Wei, J. Liu, Y. Wang, C. Tang, and Y. Hu, "Wireless edge caching based on content similarity in dynamic environments," *Journal of Systems Architecture*, vol. 115, p. 102000, 2021.

[9] S. Müller, O. Atan, M. van der Schaar, and A. Klein, "Context-aware proactive content caching with service differentiation in wireless networks," *IEEE Transactions on Wireless Communications*, vol. 16, no. 2, pp. 1024–1036, 2016.

[10] Z. Zhang, Y. Yang, M. Hua, C. Li, Y. Huang, and L. Yang, "Proactive caching for vehicular multi-view 3d video streaming via deep reinforcement learning," *IEEE Transactions on Wireless Communications*, vol. 18, no. 5, pp. 2693–2706, 2019.

[11] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[12] R. S. Sutton, "Between mdps and semi-mdps: Learning, planning, and representing knowledge at multiple temporal scales," 1998.

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[14] F. Niknia, V. Hakami, and K. Rezaee, "An smdp-based approach to thermal-aware task scheduling in noc-based mpsoc platforms," *Journal of Parallel and Distributed Computing*, vol. 165, pp. 79–106, 2022.

[15] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis, "Optimal and scalable caching for 5g using reinforcement learning of space-time popularities," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 180–190, 2017.

[16] J. Gu, W. Wang, A. Huang, H. Shan, and Z. Zhang, "Distributed cache replacement for caching-enable base stations in cellular networks," in *2014 IEEE International Conference on Communications (ICC)*, pp. 2648–2653, IEEE, 2014.

[17] N. Zhang, W. Wang, P. Zhou, and A. Huang, "Delay-optimal edge caching with imperfect content fetching via stochastic learning," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 338–352, 2021.

[18] S. Li, J. Xu, M. Van Der Schaar, and W. Li, "Popularity-driven content caching," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[19] L. Thomas, S. Gougeaud, S. Rubini, P. Deniel, and J. Boukhobza, "Predicting file lifetimes for data placement in multi-tiered storage systems for hpc," in *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pp. 1–9, 2021.

[20] S. Vural, N. Wang, P. Navaratnam, and R. Tafazolli, "Caching transient data in internet content routers," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1048–1061, 2016.

[21] N. Golrezaei, A. G. Dimakis, and A. F. Molisch, "Wireless device-to-device communications with distributed caching," in *2012 IEEE International Symposium on Information Theory Proceedings*, pp. 2781–2785, IEEE, 2012.

[22] H. Gomaa, G. G. Messier, C. Williamson, and R. Davies, "Estimating instantaneous cache hit ratio using markov chain analysis," *IEEE/ACM transactions on Networking*, vol. 21, no. 5, pp. 1472–1483, 2012.

[23] R. A. Horn, "The hadamard product," in *Proc. Symp. Appl. Math*, vol. 40, pp. 87–169, 1990.

[24] T. Developers, "Tensorflow," *Zenodo*, 2022.