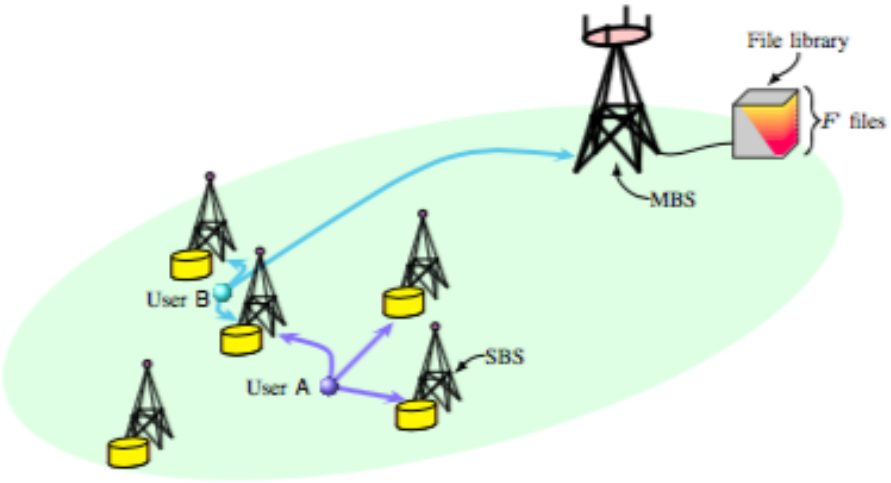
CHALMERS
UNIVERSITY OF TECHNOLOGY



Reinforcement Learning for Caching

Master's thesis in Master Programme of Communication Engineering

XIXI LIU

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019:Xixi Liu

Reinforcement Learning for Caching

XIXI LIU

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Communication Systems
Chalmers University of Technology
Gothenburg, Sweden 2020

Reinforcement Learning for Caching

XIXI LIU

Supervisor: Prof. Alexandre Graell i Amat, Department of Electrical Engineering
Co-Supervisor: Jesper Perderson, Department of Electrical Engineering
Co-Supervisor: Christian Häger, Department of Electrical Engineering
Examiner: Prof. Alexandre Graell i Amat, Department of Electrical Engineering

Master's Thesis 2020: Xixi Liu
Department of Electrical Engineering
Division of Communication Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 72 975 5242

Cover: Content delivery for a wireless network consisting of an MBS and five SBSs [1].

Typeset in LaTeX
Printed by [Chalmers University of Technology]
Gothenburg, Sweden 2020

Reinforcement Learning for Caching

XIXI LIU
Department of Electrical Engineering
Chalmers University of Technology

Abstract

It has been proposed to deploy several small base stations (SBSs) equipped with caching units to deal with communication traffic congestion. It has been proved and shown that caching content using error correcting codes (ECCs) in them can reduce the download delay and the backhaul rate. In this scenario, users download content from a file library, which is always available at a macro base station (MBS). They could download content from the SBSs, or, if that is not possible, from the MBS. During peak-traffic times, the popular files are prefetched at the SBSs. Optimizing the caching placement in order to minimize the backhaul rate is an interesting direction to explore, especially for the case of dynamic file popularity. In this thesis, reinforcement learning (RL), in particular, Q-learning, is used to find the optimal caching placement in order to minimize the backhaul rate.

Keywords: Caching, Reinforcement Learning, Markov decision process, Q-learning.

Acknowledgements

I would like to thank Prof. Alexandre Graell i Amat for offering me this interesting and promising project, as well as my co-supervisers Jesper Perderson and Christian Häger for giving me professional advice in the field of caching and reinforcement learning, respectively.

Xixi Liu, Gothenburg, February 2020

Contents

List of Figures

1
Introduction

1.1 Background

Wireless traffic is growing at an unprecedented rate, due to an increasing number of smartphones and average data volume per subscription, due to more viewing and higher resolution of video content. It is reported that the demand for wireless data is predicted to reach 136 exabytes per month by 2024, where on-demand video streaming is expected to account for 74% of the downloaded data [3]. How to provide and sustain an acceptable quality of experience (QoE) to users is thus an important problem that needs to be addressed. Because the backhaul link of the communication system can be easily congested during peak-traffic times and underutilized during the off-peak times [4], one approach is to deploy small base stations (SBSs) equipped with cache units at the end-users. During off-peak traffic hours, popular content is prefetched and cached at the SBSs. In this manner, the congestion of traffic is reduced. Such network structures are referred to as heterogeneous networks (HetNets). Besides, HetNets allow that some scarce resources, such as frequency, can be reused substantially. Since a small number of popular content will be downloaded frequently by users who wish to download at any time, fetching the expected content to SBSs during off-peak traffic times, such as night time, can be an effective way to alleviate some of the backhaul link traffic. This method is called caching.
Caching is a mature idea from the domains of web caching and content distribution networks (CDNs) [5]. CDNs include two parts, a central server and several proxy servers that are distributed geographically, as Fig. 1.1 shows. Frequently requested content at the central server will be replicated and put at the proxy servers that are closer to end-users when the communication systems are underutilized to mitigate the heavy traffic over the backhaul link. In wireless caching, the macro base station (MBS) takes the role of the central server and the SBSs correspond to the proxy servers in the CDNs scenario. The caching procedure can be divided into two phases [2]: caching placement, which happens before users' requests are revealed and content delivery, which is proceeded after users' requests are revealed.

There are several differences between caching and wireless caching.

1. The users are mobile in the wireless caching scenario. Hence, the user can connect to different caches at different times.
2. Due to the overlapping SBSs and device coverage areas, users can connect to several SBSs at any given time.

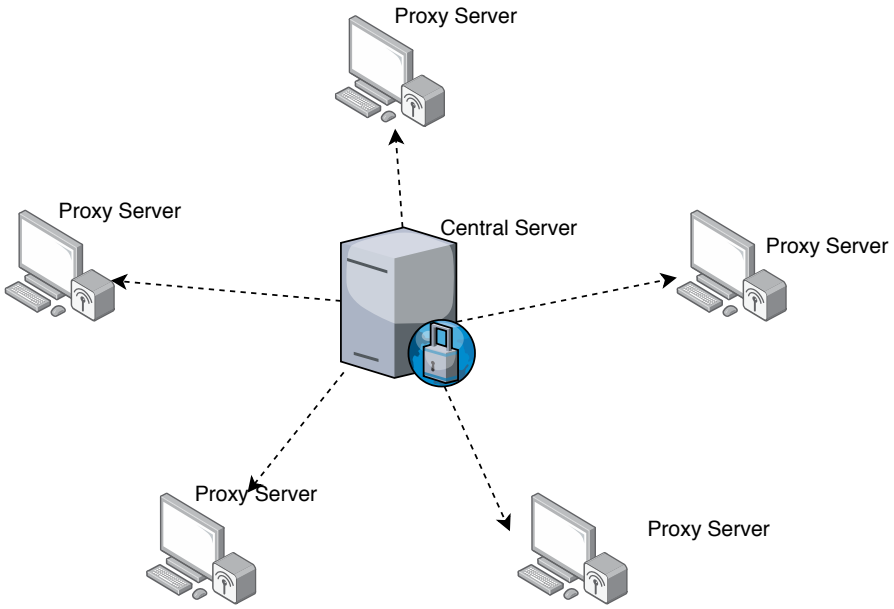


Figure 1.1: A single central server with several proxy servers.

3. The wireless medium is a broadcast medium, which implies that a transmission from a SBS can be overheard by any device within the transmission range.

Based on these three properties of wireless caching, there are two different aspects of coding for caching. One is for caching placement. Since users have access to several SBSs at the same time, codes, such as Reed-Solomon codes or Fountain codes, are proposed to decrease the backhaul rate by using the property that users can access multiple cache nodes [2, 6, 7]. The other aspect of coding for caching is content delivery. The most typical coding is index coding, which takes advantage of the inherent broadcast nature of wireless networks [4, 8].

The performance of different caching placements with maximum-distance separable (MDS) codes was analyzed in [2] compared with an uncoded scheme. Besides, [2] formulated the MDS coded caching placement as a convex optimization problem in terms of minimization of the backhaul rate. [1] formulated the optimal coded caching placement using MDS codes as a mixed linear integer programming (MILP). However, the analysis of [2, 1] is based on stationary content popularity. In reality, the content popularity is non-stationary. Finding the optimal coded caching placement for the case of dynamic content popularity is an interesting direction to explore. Currently, machine learning is widely used to deal with the problems that do not have a model or are difficult to formulate as a model. Many problems that arise in the context of content caching can be addressed using machine learning. Generally, there are two main problems, which are content popularity estimation and caching placement optimization. For example, if the phase of content popularity estimation is separated from finding an optimal caching placement, content popularity estimation can be taken as a problem that uses past observations to predict future content popularity. Machine learning, especially, supervised learning, can be used to solve such problems. As for caching placement optimization, one needs to decide which content should be cached in which node in a potentially dynamic environment to

maximize certain rewards or minimize certain costs. For example, hit rate, i.e., the probability that certain content is found in a cache node, can be maximized. Besides, the cost of refreshing cache nodes, i.e., the number of files that are cached at time slot t but not at time slot $t-1$ multiplies with a weighting parameter, can be minimized [9]. Moreover, rewards also can be a combined metric, which consists of the hit rate and the cost of refreshing cache nodes with different weights [9]. Such a problem can be formulated as a Markov decision process (MDP) and can be solved by using RL. The authors of [9] used a grouped linear prediction (GLP) model based on historical data to predict the future content popularity, and Q-learning is combined with the GLP model to decide which contents should be cached to acquire a long-term reward, which is a weighted metric that consists of hit rate and the cost of refreshing cache nodes. The data set that is used to test the proposed algorithm is from [10]. However, the drawback is that [9] only uses one cache node, which implies that no coding scheme is used. In reality, due to the deployment of SBSs, coding can be introduced to the caching placement scheme to decrease the average backhaul rate. Content popularity estimation and caching placement optimization was investigated in [11]. However, coding is not considered.
To conclude, current research has not explored the direction that integrates the file popularity estimation with finding the optimal coded caching placement.

1.2 Purpose and limitations

The thesis aims to find the optimal coded caching placement with a non-stationary content popularity by using Q-learning. The limitation is that the prediction of content popularity and the coded caching placement are decoupled, which means that the prediction of content popularity is independent of the decision of caching placement.

1.3 Thesis Outline

This master thesis report is structured as follows.
In Chapter 2, we introduce the theory on caching and reinforcement learning. In Chapter 3, we formulate the caching placement problem as an MILP problem and illustrate how to use Q-learning to solve this problem.
In Chapter 4, we present results for three scenarios. The first one is that the file popularity is simulated by Zipf distribution; the second one is using the dataset from [10]; the third one is that file popularity estimation and caching placement decision are integrated implicitly. Besides, we discuss the advantage of using Q-learning to solve the MILP problem.
In Chapter 5, we give directions for interesting future work.

2

Theory

2.1 Wireless Caching

This section summarizes the caching policies for two different scenarios in terms of the number of cache nodes used in the communication system. For the case of a single cache node, the content is cached using an uncoded scheme. The other one has several SBSs in a cell, which means users can access multiple SBSs at the same time. In the latter scenario, it is beneficial to introduce a coding scheme by exploiting overlapping coverage. ECCs can be used to increase the performance of a communication system.

2.1.1 Uncoded Caching Scheme

We are considering a scenario where a user can only have access to one cache node. The file library is dynamic, which means that the number of files is changing with time going by and the corresponding file popularity is also changing. The file library at time slot t is indicated by $\mathcal{F}_t$. Besides, one cache node can store up to M files, which is smaller than the number of files in the file library. The potential popular files will be cached at the cache node according to the caching placement at time slot $t-1$ during off-peak traffic times. When users' requests are revealed at time slot t, the number of files that the user wants to download as well as are cached at the cached node over the number of whole requests is defined as hit ratio, which is indicated by H_t. Moreover, since the caching placements of two continuous time slots are different, refreshing the cache node might cause replacement cost because of the replacement speed, network traffic and channel conditions [9].
Several caching policies are proposed to maximize the hit ratio for the case of dynamic file popularity, they are illustrated as follows.

2.1.1.1 Most Popular

The most popular policy means that the most popular M files will be cached at the cache node. The file popularity for next time slot is estimated based on the file request history. For example, the file f popularity at time slot t is calculated by the number of file f that are requested over the number of the whole file request from time slot 1 to time slot $t-1$. The most popular M files according to the calculated file popularity are cached at the cache node. It is based on the assumption that more popular files will be more likely to be requested in the future.

However, if some files are only requested extensively at one time slot and hardly requested later, the estimated file popularity of these files will be partial. Those files will be cached at the cache node for a while. When new files are appearing, this caching policy will keep ignoring the new files unless the popularity of the new files is larger than the current least popular file at the cache node. Besides, the so-called most popular files will occupy the cache node with no use.

2.1.1.2 Least Recently Used

The least recently used (LRU) policy means the least recently requested files will be replaced by the new files when the cache node is full. This policy does not consider the file popularity of the new files. This method assumes that the new files are more likely to be requested in the future. It is often used in a scenario where new files often bump up.

2.1.1.3 Least Frequently Used

The least frequently used (LFU) algorithm emphasizes that the least frequently requested files are replaced by the new files when the cache node is full. It also does not consider the file popularity of the new files. The logic of this policy is that the least frequently used files in a period will be less likely to be requested in the future. However, it still has the pitfall that the most popular one has.

2.1.1.4 Grouped Linear Prediction Model

The GLP model is a recent method that predicts file popularity by considering the age of the file. The files that appear initially at the same time slot are defined as the same age group [9]. This model is basically a linear regression model. It uses the file history to find $\Theta = (\theta_{\mathbf{1}}, \theta_{\mathbf{2}}, \theta_{\mathbf{3}}, ...\theta_{\mathbf{i}})$, which is a parameter that can be multiplied with the history of file requests to predict the future file request. i is the age of a file. The dimension of Θ depends on the number of time slots, in which file request history is taken as valid samples to predict future file popularity. It can be changed manually for specific scenarios. Finding the optimal Θ can be formulated as a linearly constrained optimization problem,

$$\begin{aligned} \min_{\theta_{\mathbf{1}},...,\theta_{\mathbf{i}}} \quad & \sum_{k=1}^{t} \sum_{f \in \mathcal{F}_k} (\hat{d}_{k,f} - d_{k,f})^2 \\ \text{s.t.} \quad & \theta_{i,j} - \theta_{i,j+1} \geq 0, \forall j \in [1 : i-1], \forall i \in \mathbb{N}^+ \\ & \theta_{i,i} \geq 0, \forall i \in \mathbb{N}^+ \end{aligned} \tag{2.1}$$

where $d_{k,f}$ represents the number times file f that is requested at time slot k. $\hat{d}_{k,f}$ represents the predicted number of times file f that is requested at time slot k. It is calculated by

$$\hat{d}_{k,f} = \theta_{\mathbf{i}} \cdot X_{k,f} \tag{2.2}$$

The files that belong to the same age group use the same $\theta_{\mathbf{i}}$, $\theta_{\mathbf{i}} = (\theta_{i,1}, \theta_{i,2}, ..., \theta_{i,i})$. $\theta_{i,j}$ represents a normalized correlation coefficient between the requests of files that

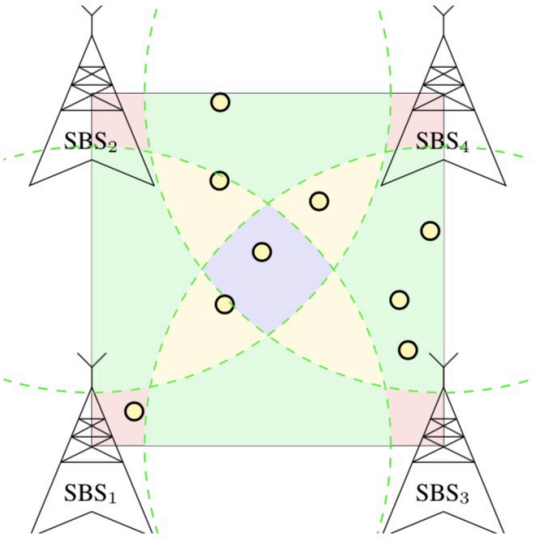


Figure 2.1: Small cells topology [2]

have the age of i at time slot t and the requests of the same files at time slot $t-j$. The reason why $\theta_{i,j} - \theta_{i,j+1} \geq 0$ is that the shorter time interval shows stronger correlation. The reason why $\theta_{i,i} \geq 0$ is that it is assumed that the files that are requested before are more likely to be requested in the future. $X_{k,f}$ represents the file history of file f at time slot k. It is calculated by

$$X_{k,f} = (d_{k-1,f}, d_{k-2,f}, ...d_{k-i,f})^T \tag{2.3}$$

The GLP model is used to acquire the predicted file requests for every file f at time slot k, $\hat{d}_{k,f}$. After that, the file popularity can be estimated based on the predicted file requests. The most popular M files are cached at the cache node.

2.1.2 Maximum-distance Separable Codes

Using the coded caching scheme is beneficial to increase the performance of the communication system. For an (n, k_c) error correction code (ECC), it encodes k_c information fragments to a codeword of n encoded packets by adding $n - k_c$ parity packets. For example, 3-repetition is a typical code, which is used to increase the resilience of the communication system. It means each file is copied three times and cached at different cache nodes. The disadvantage of this code is that it has high overhead. The storage overhead is $\frac{n}{k_c} = \frac{3}{1}$. The fault tolerance is 2 nodes failure. A suitable ECC to balance the storage overhead and fault tolerance is desirable. Among all ECC codes, MDS codes have the best guaranteed performance in terms of the trade-off between storage overhead and fault tolerance. For example, for a (9,7) MDS code, the storage overhead is $\frac{n}{k_c} = \frac{9}{7} = 1.28$. The fault tolerance is 2 nodes failure. In other words, an MDS code is the optimal ECC code that has the minimum storage overhead given a fixed fault tolerance. If a file is encoded by using an MDS code (n, k_c), the file is divided into k_c uncoded fragments and encoded to n

coded packets. A property of this MDS code is that any subset of k_c coded packets can recover the file information without any loss.

2.1.3 Coded Caching Scheme

The coded caching scheme is introduced to the wireless edge because more than one SBS is used in a cell. There will be an overlapped area where several small stations can serve, which means that users can potentially access several cache nodes at the same time. Therefore, the introduced redundancy can be used to minimize the backhaul rate.

We are considering a cellular network where a macro-cell is served by a MBS. Mobile users wish to download files from a library of N files, which are always available at the MBS through a backhaul link. The file library has popularity distribution presented by $\mathbf{p} = (p_1, p_2, .., p_N)$ where file i is requested with probability p_i. It is also assumed that N_{SBS} SBSs are deployed in a cell to serve users' requests and offload traffic from the MBS. Besides, each SBS has a cache size of M files. Users can be served by SBSs within of communication range. If the user can not recover the file from the collecting packets from SBSs, additional encoded packets from the MBS through the backhaul link need to be fetched to recover the file. The average fraction of files that are downloaded from the MBS is called backhaul rate. The cell topology is shown in Fig.2.1. Different colour regions in Fig.2.1 represent the area where a user can be served by the same number of SBSs at the same time. For example, the blue region represents the area where users can be served by 4 SBSs at the same time. The probability that a user can be served by b SBSs is represented by γ_b. The value of γ_b is equal to the percentage of the coverage area where b SBSs could serve over the whole square area. The system parameters are defined as follows:

- N_{SBS}: the number of SBSs in a cell; N_{SBS} is equal to 4 in Fig.2.1.
- N: the total number of files in the file library.
- p_i: the file i popularity, it satisfies $\sum_{i=1}^{N_{\mathsf{SBS}}} p_i = 1$.
- M: cache size. Here we assume all the files have the same size.
- r: the radius of the area that a small base station can provide.
- γ_b: the probability that a user can be served by b small base stations.

An MDS code (n, k_c) is used to encode files in this scenario. During the phase of caching placement, if the encoded packets for each file i, m_i, are cached in each cache node, the number of total encoded packets for file i is $n = (k_c + (N_{\mathsf{SBS}} - 1) \cdot m_i)$. The caching placement is expressed as $[m_1, m_2, ..., m_N]$. During the phase of content delivery, users will send file requests, the SBSs covering their locations will initially serve those users. If a user is served by b SBSs at the same time and file i is requested, the number of encoded packets that the user can collect from SBSs is $(b \cdot m_i)$. When the user downloads $(b \cdot m_i) \geq k_c$ encoded packets, the requested file can be decoded and the backhaul link will not be used. Otherwise, $(k_c - b \cdot m_i)$ uncached encoded packets need to be fetched from the MBS through the backhaul link in order to recover file i.

Four caching placement strategies are often in use. The following section intends to illustrate each of them with the assumption that each file has the same size. Besides,

it is assumed that each file is partitioned into k_c uncoded fragments, the number of encoded packets may vary after using different code rates, which depend on the caching placement strategies that are used.

2.1.3.1 Most Popular Placement

The most popular placement strategy means the most M popular files are cached at each cache node. The cache size of each SBS is M, which means those most popular M files are cached completely at each cache node. The performance is the same as the most popular policy using the uncoded caching scheme.

2.1.3.2 Uniform Placement

The uniform placement strategy is the number of cached encoded packets of each file is the same, which is independent of the file popularity. For each file in the file library, there will be $\frac{M}{N}$ encoded packets for each file that are placed randomly at each cache node.

2.1.3.3 Proportional Placement

The proportional placement strategy means that the number of cached coded packets of each file is proportional to its file popularity. The larger the file popularity is, the more cache space is allocated. More specifically, for file i, the available cache space is $p_i \cdot M$. If each encoded packet has size n_p, the corresponding number of encoded cached packets for file i is $m_i = \frac{p_i \cdot M}{n_p}$ at each cache node.

2.1.3.4 Optimal Placement

[2] formulated the optimal MDS coded caching placement in terms of the minimization of the average backhaul rate as a convex optimization problem,

$$\begin{aligned} \min_{q_1,\ldots,q_N} \quad & \sum_{i=1}^{N} \sum_{b=1}^{N_{\mathsf{SBS}}} \gamma_b p_i (1 - \min(1 - b \cdot q_i)) \\ \text{s.t.} \quad & \sum_{i=1}^{N} q_i \leq M \\ & 0 \leq q_i \leq 1, \forall i \in [1, N] \end{aligned} \tag{2.4}$$

where $q_i = \frac{m_i}{k_c}$. It can be seen from (2.4) that calculating a set of q_i to minimize the average backhaul rate equals to finding the optimal code rate for each file i. Since

$$
\begin{aligned}
q_i &= \frac{m_i}{k_c} \\
&= \frac{\frac{n-k_c}{N_{\mathsf{SBS}}-1}}{k_c} \\
&= \frac{n-k_c}{(N_{\mathsf{SBS}}-1)\cdot k_c} \\
&= \frac{1-\frac{k_c}{n}}{(N_{\mathsf{SBS}}-1)\cdot \frac{k_c}{n}}
\end{aligned}
\tag{2.5}
$$

According to the definition of code rate, $R_c = \frac{k_c}{n}$,

$$
q_i = \frac{1-R_c}{(N_{\mathsf{SBS}}-1)\cdot R_c} \tag{2.6}
$$

Due to $0 \leq q_i \leq 1$, it can be derived that the range of code rate can be chosen is,

$$
\frac{1}{N_{\mathsf{SBS}}} \leq R_c \leq 1 \tag{2.7}
$$

2.2 Reinforcement Learning

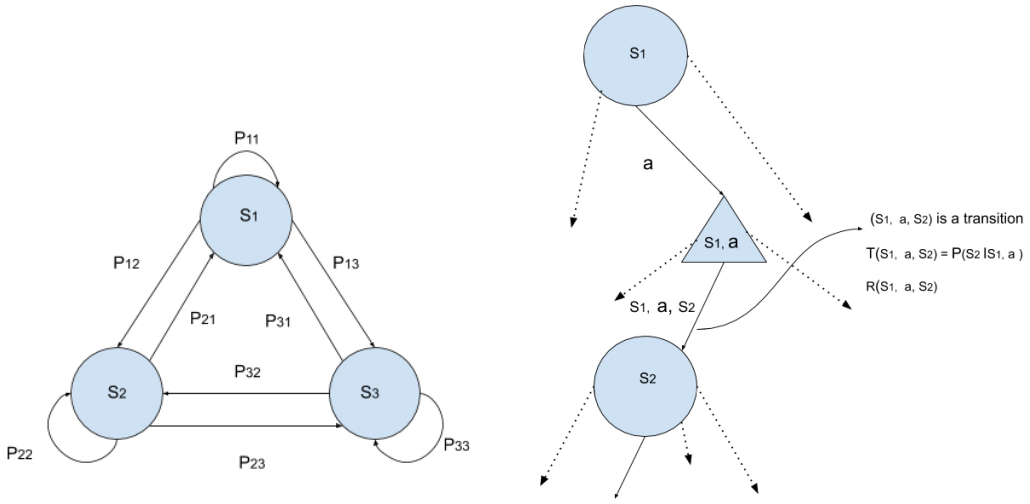
Machine learning has been one of the topics that are attracting widespread attention both in industry and academia. There are three paradigms in machine learning, which are supervised learning, unsupervised learning and reinforcement learning (RL) [12]. Supervised learning tries to learn from a set of training samples, which are labeled by a professional supervisor. It is commonly used in the classification problem [12]. Unsupervised learning tries to learn from a set of unlabeled training samples and find the hidden structure underlying the training samples. It is often used in clustering analysis [12]. The first time RL was proposed was in the late of 1979 [12], the main advantage of it is the optimal decision is made by continuously learning information from the environment. The decision made by RL will adapt to the changing of the environment and acquire a maximum long-term reward.
The common scenario that uses RL is where an agent tries to interact with the environment and find a set of state-action pairs that can get the maximum reward from a long term perspective, as Fig.2.3 shows. It is not like supervised learning, which has a training set and each labeled sample can be used to train the model [12]. The essential difference between RL and unsupervised learning is that the goal of the former is to get a sequence of state-action pairs to maximize a reward signal but the latter aims to find the hidden model structure of training samples.

2.2.1 Markov Decision Process

The Markovian property is that the future is independent of the past given the present. A Markov chain is a stochastic model, which describes a sequence of possible events in which the transition probability of each event depends only on the

state attained in the previous event, as the left plot of Fig.2.2 shows. Besides, the transition probability is independent of time. In continuous-time, it is known as a Markov process. An MDP is an extension of a Markov chain and is used to formalize a reinforcement learning task. The difference between the Markov chain and the MDP is two more parameters are introduced, which are actions and rewards. If the state space and the action space of an MDP are finite, it is called a finite MDP. The following theory part is illustrated under the assumption that the environment is a finite MDP.
An MDP is uniquely defined by:

- a set of states, $\mathbb{S}$;
- a set of actions, $\mathbb{A}$;
- transition function $T(s' \mid s, a)$, which represents a probability distribution over next possible states s' given current state s after proceeding action a;
- reward function $R(s' \mid s, a)$, which represents the reward after proceeding the action a at state s'.



(a) Markov chain **(b)** Markov decision process

Figure 2.2: Markov chain and Markov decision process

2.2.1.1 Basic Definitions

The action space is a set of all possible actions that an agent can proceed. The state space is a set of all possible states that an agent can arrive at. Reward function depends on the state that an agent is at and the action that an agent proceeds. It can be defined specifically for different scenarios. The transition function is used to represent the probability that an agent in the state s transfers to the state s'.
Take a sample $(s, a, R, s', T(s, a, s'))$ to illustrate the relationship between these four elements. An agent in the state s, after taking the action a, it will arrive at the next state s' with transition probability $T(s, a, s')$ and get the corresponding reward $Rs, a, s')$ for proceeding the action a. A table that stores the rewards of all the action-state combinations is called Q-table.

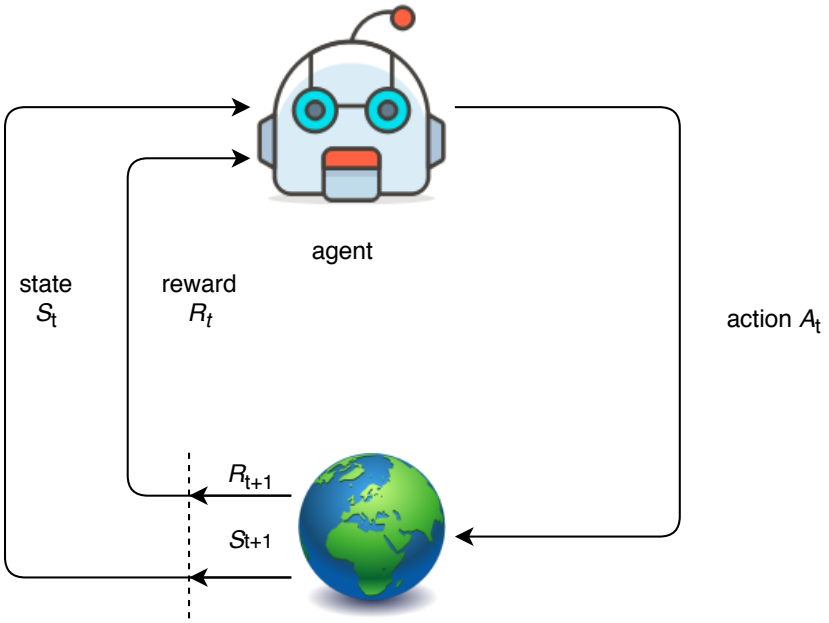


Figure 2.3: The agent–environment interaction in reinforcement learning

If the transition probability is 1 always, the MDP is deterministic. Otherwise, it is stochastic.

2.2.1.2 Value Function

In most cases, RL algorithms are related to value functions. They are functions of the state or state-action pairs and used to estimate how good it is for an agent to be in a given state or to perform a given action in a given state [12]. The criteria to evaluate how good those functions is defined in terms of expected rewards, which depends on what actions an agent will proceed in the future. Policy function π maps each state $s \in \mathbb{S}$, each action $a \in \mathbb{A}$ to the probability $\pi(a|s)$ of proceeding action a in state s [12]. Therefore, value functions are defined uniquely for specific policies. RL algorithms aim to find the optimal policy that can lead to the maximum expected rewards.
In a specific scenario, a proper definition of reward function is essential to evaluate the performance of taking action a in state s. In order to maximize the expected rewards, a value function is introduced to formulate the long term reward given a specific discount factor γ, policy π. There are two functions, the state-value function and the action-value function.
The definition of the state-value function is shown as (2.8),

$$V_\pi(s) := \lim_{T\to\infty} \mathbb{E}_\pi\Big[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \mid S_t = s\Big] \tag{2.8}$$

A similar definition of the action-value function is shown as (2.9). For the state-value function, it starts in state s and then proceeds action under policy π thereafter. However, the action-value function proceeds action a first and then proceeds actions under policy π thereafter.

$$q_\pi(s,a) := \lim_{T\to\infty} \mathbb{E}_\pi\Big[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \mid (S_t = s, A_t = a\Big] \tag{2.9}$$

One of the properties of the value function is its recursive relationship between the current state and its successor states under policy π. The relation can be expressed as a well-known equation, which is called Bellman equation.
For state-value function,

$$V_\pi(s) = \mathbb{E}_\pi\Big[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s\Big] \tag{2.10}$$

$$\mathbb{E}_\pi\Big[R_{t+1} \mid S_t = s] = \sum_a \pi(s,a) \sum_{s'} T(s' \mid s,a) R(s' \mid s,a) \tag{2.11}$$

$$\mathbb{E}_\pi\Big[\gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s\Big] = \sum_a \pi(s,a) \sum_{s'} T(s' \mid s,a)\Big(\gamma \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_{t+1} = s'\Big]\Big) \tag{2.12}$$

Plug (2.11) and (2.12) in (2.10), the Bellman equation of the state-value function can be derived as (2.13) shows.

$$\begin{aligned} V_\pi(s) &= \sum_a \pi(s,a) \sum_{s'} T(s' \mid s,a)\Big(R(s' \mid s,a) + \gamma \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_{t+1} = s'\Big]\Big) \\ &= \sum_a \pi(s,a) \sum_{s'} T(s' \mid s,a)(R(s' \mid s,a) + \gamma V_\pi(s')) \end{aligned} \tag{2.13}$$

A similar way can be used to derive the Bellman equation of the action-value function, which is shown in (2.14).

$$q_\pi(s,a) = \sum_{s'} T(s' \mid s,a)\Big[R(s' \mid s,a) + \gamma \sum_{a'} \pi(s',a') q_\pi(s',a')\Big] \tag{2.14}$$

It is worth considering how to choose the optimal policy π^* to maximize the long term reward. We can take advantage of Bellman equation. Before presenting how to find the optimal policy, optimal state-value function and optimal action-value function needs to be introduced firstly. Optimal state-value function is defined as follows [12],

$$V_*(s) = \max_\pi V_\pi(s) \tag{2.15}$$

for all $s \in \mathbb{S}$ and $a \in \mathbb{A}$.
Optimal action-value function is defined as follows [12],

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.16}$$

for all $s \in \mathbb{S}$ and $a \in \mathbb{A}$.

It has been defined that if $V_\pi > V_{\pi'}$, it means a policy π is better than policy π'.

Two types of methods are applied to finding the optimal policy in terms of transition probability. One is dynamic programming (DP), which is applied to the problem that the transition probability is known [12]. The other is temporal-difference (TD) learning, which is used to deal with the problem that the transition probability is unknown [12]. The following section will introduce them in more detail.

2.2.2 Dynamic Programming

For dynamic programming, the transition probability is known. It implies that the model of the environment is known perfectly. Therefore, DP can be taken as a performance benchmark. Policy iteration algorithm and value iteration algorithm are mostly used.

2.2.2.1 Policy Iteration

The core of the policy iteration algorithm is improving the policy at every iteration. Generally, there are two parts in the policy iteration algorithm.

1. Policy evaluation: It aims to calculate the state-value function under policy π. A common way to evaluate a policy is using the iterative policy evaluation algorithm. In the beginning, given a random policy π_0 and an initial value array V_0. For each state $s \in \mathbb{S}$, the value function is calculated based on the initial policy π_0. Thereafter, Q-table is updated by plugging the calculated $V(s)$ into the Q-table. Repeat this procedure until the difference of value function between two iterations for each state is smaller than a small positive number δ, which can be set manually. In other words, the real state-value function is acquired under policy π.

2. Policy improvement: The current policy is improved by using the present Q-table. More specifically, a policy is improved by choosing the action that leads to the maximum reward for each state based on the current Q-table. The termination of the iteration is when the policy cannot be updated. In other words, the policy is unchanged when more iterations are used.

One tricky part of policy iteration is about policy improvement. In particular, when the iterative policy evaluation algorithm is used, there are two state-value functions to store the old value V_k and new updated value V_{k+1} for each state. The condition to terminate the iteration depends on the difference between V_k and V_{k+1}. Another method is only using one V array. Once a new $V_{k+1}(s)$ is calculated, the old $V_k(s)$ will be overwritten immediately. The second method can get the converged value function quicker than that of using two value function arrays [12].

2.2.2.2 Value Iteration

The implementation of the policy iteration algorithm has two steps, the policy evaluation and the policy improvement. Besides, the policy improvement only can be proceeded after the policy evaluation is finished. Moreover, the policy evaluation

needs to reach all the states multiple times to get the converged value function under policy π. Therefore, it is desirable to come up with a new method to acquire the optimal policy. It has been proved that the policy evaluation step of the policy iteration algorithm can be truncated without losing the convergence of the policy iteration algorithm [12]. The modified algorithm is called the value iteration algorithm. It reaches all the states once and calculates state-value of all possible actions to update the state-value function for each state. The procedure is terminated when the state-value function difference between V_k and V_{k+1} is smaller than a positive number. Finally, the optimal policy can be extracted from Q-table.

2.2.3 Temporal-Difference Learning

It can be clearly seen that DP only can be proceeded with knowing the transition probability. More specifically, all possible successive states will be considered when the state-value function is updated. The charm of TD learning is that it can take advantage of the information from the environment without knowing the transition probability to get the optimal policy. The way to learn information from the environment for temporal learning is using the continuous experienced samples to update the Q-table. Thereafter, the optimal policy can be extracted from the Q-table [12]. Q-learning is an off-policy TD control algorithm if the greedy policy is not used.

2.2.3.1 Q-learning

Before illustrating what Q-learning is, a function that is similar to the action-value function is introduced. It is called Q-function and is shown in (2.17).

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a')) \tag{2.17}$$

The difference between (2.14) and (2.17) is that (2.17) takes the updated action-value function as a approximation of the optimal action-value function directly. Besides, the updated action-value function is independent of the policy being followed. (s, a, R, s') is defined as an experienced sample. In order to find the optimal policy, the Q-function is updated as (2.18) shows.

$$Q(s,a) \overset{\alpha}{\Leftarrow} R + \gamma \max_{a'} Q(s',a') \tag{2.18}$$

$\alpha \in (0,1)$ is the learning rate. In the beginning of Q-learning iteration times, α is often to be set bigger. With the learning proceeding, α, learning rate, will decrease since $\hat{Q}$ will be closed to the real Q value. $\gamma \in (0,1)$ is discount factor.
Expand (2.18),

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s',a')) \tag{2.19}$$

R is the immediate reward after proceeding action a in state s. $\hat{Q}$ is the estimated Q action-value function. With infinitely visiting (s,a), it has been proved that $\hat{Q}(s,a)$ will converge to the real $Q(s,a)$ if the learning rate satisfies (2.20).

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \sum_{t=1}^{\infty} \alpha_t^2 \leq \infty, \tag{2.20}$$

2.2.3.2 Learning Rate

In the beginning, the learning rate α is usually bigger. The initial $\hat{Q}$ value is far away from the real Q value, therefore the information from the environment is quite important for updating the $\hat{Q}$ in order to make $\hat{Q}$ closer to the real Q. With the learning procedure proceeding, the value of learning rate needs to be decreased. Since too large α may cause the $\hat{Q}$ oscillates around the real Q. In one sense, the learning rate can be explained as the learning step size. If the learning rate is too small, lots of iteration times need to be used to make Q-table converge. If it is too large, the Q-table is hard to converge.

2.2.3.3 Exploration and Exploitation

One of the challenges for RL is the trade-off between exploration and exploitation. The origin of exploration and exploitation comes from how to choose the action to get the experienced samples. If actions that can get the maximum instant reward are chosen every iteration, it is beneficial for the agent to have a reliable estimation of expected reward. However, due to the limitation of learning iteration, some possible actions that can acquire a larger reward may have no chance to be reached. Therefore, the dilemma of exploitation and exploration diminishes the performance of reinforcement learning [12]. There is a way to deal with this trade-off. With the probability of $\epsilon \in [0, 1]$, the action is chosen randomly. With the probability of $1 - \epsilon$, the action that gets the maximum reward is chosen.

3
Methods

3.1 Problem Formulation

3.1.1 Scenario

The scenario that is used in this thesis is the same as [1]. A cellular network where a macro-cell is served by a MBS. Mobile users wish to download files from a library of N files, which are always available at the MBS through a backhaul link. The file library has popularity distribution for each file, it can be presented by $\mathbf{p} = (p_1, p_2, .., p_N)$, where file i is requested with probability p_i. It is also assumed that there are N_{SBS} SBSs deployed to serve users' requests and offload traffic from the MBS. Besides, each SBS has a cache size of M files. Users are served by SBSs within communication range. If a user can not recover information of the file from the SBSs, additional encoded packets from the MBS through the backhaul link need to be fetched to recover the file. The average fraction of files that are downloaded from the MBS is called backhaul rate, denoted by R. For no caching, R=1. The cell topology that is used in this thesis is the same as [2]. Thus, the result of [2] can be taken as a benchmark to compare.

3.1.2 Coded Caching Placement

Assume that all the files have the same size. During the caching placement phase, file i is partitioned into k_i uncoded fragments. These uncoded fragments are encoded to N_{SBS} encoded packets by using an (N_{SBS}, k_i) MDS code. Thereafter, each SBS will cache one encoded packet of file i. In other words, for any file that is cached, each of SBS stores one encoded packet of that file. The difference is that the number of uncoded fragments of each file may vary. The number of uncoded fragments of file i is denoted by k_i.
Here we define $\mu_i \triangleq \frac{1}{k_i}$ for convenience, in which $k_i \in \{1, ..., N_{\mathsf{SBS}}\}$. Therefore,

$$\mu_i \in \{\frac{1}{N_{\mathsf{SBS}}}, ..., \frac{1}{2}, 1\},$$

$\boldsymbol{\mu} = \{\mu_1, \mu_2, ..., \mu_N\}$ is defined as caching placement. Since each SBS can only cache M files, it follows:

$$\sum_{i=1}^{N} \mu_i \leq M.$$

$\mathcal{C}^{\boldsymbol{\mu}}_{\mathbf{MDS}}$ is used to denote the caching scheme that uses MDS code according to caching placement $\boldsymbol{\mu}$ [1].

3.1.3 Backhaul Rate Formulation

We aim to use RL to find the optimal coded caching placement to minimize the backhaul rate, which has been formulated in [1] and defined as (3.1).

$$\begin{aligned} \min_{\mu_1,\ldots,\mu_N} \quad & \sum_{i=1}^{N} p_i \lceil \mu_i \rceil \sum_{b=1}^{N_{\mathsf{SBS}}} \gamma_b \max(0, \frac{1}{\mu_i} - b)\mu_i + \sum_{i=1}^{N} p_i \lfloor 1 - \mu_i \rfloor \\ \text{s.t.} \quad & \sum_{i=1}^{N} \mu_i \leq M \\ & \mu_i \in \{0, \frac{1}{N_{\mathsf{SBS}}}, ..., \frac{1}{2}, 1\} \end{aligned} \tag{3.1}$$

The case $\mu_i = 0$, implying that file i is not cached, is added by convention. If $\mu_i = 0$, the corresponding backhaul rate will be 1 according to (3.1). The situation that file i is not cached is represented by the second part of (3.1). The first part of (3.1) accounts for the backhaul rate of a situation that file i is encoded by using (N_{SBS}, k_i) MDS code. The code rate, $R_c = \frac{k_i}{N_{\mathsf{SBS}}}$, and the range of k_i, $k_i \in \{1, ..., N_{\mathsf{SBS}}\}$, it can be derived that the range of code rate can be chosen is,

$$R_c \in \{1, \frac{N_{\mathsf{SBS}} - 1}{N_{\mathsf{SBS}}}, ..., \frac{1}{N_{\mathsf{SBS}}}\} \tag{3.2}$$

Recall the optimal caching placement that is illustrated in section 2.1.3.4. The range of code rate that can be chosen,

$$\frac{1}{N_{\mathsf{SBS}}} \leq R_c \leq 1 \tag{2.7 revisited}$$

The optimal coded caching placement calculated in section 2.1.3.4 is taken as a benchmark. In order not to confuse the notation, the range of code rate that is considered as a benchmark is denoted as,

$$\frac{1}{N_{\mathsf{SBS}}} \leq R_{benchmark} \leq 1 \tag{3.3}$$

The reason why we can take (2.7) as a benchmark is that it represents the average backhaul rate of the same scenario as we use in our formulated problem. The only difference is that the choice of the code rate of the benchmark could be any real number from $\frac{1}{N_{\mathsf{SBS}}}$ to 1. However, the choice of the code rate of our formulated problem is specified to $\{1, \frac{N_{\mathsf{SBS}}-1}{N_{\mathsf{SBS}}}, ..., \frac{1}{N_{\mathsf{SBS}}}\}$. Therefore, the optimal coded caching placement from section 2.1.3.4 can be taken as a relaxation of our formulated problem.
The benchmark that is used in this thesis is a convex optimization problem [2], which can be solved by CVX. CVX is a Matlab-based modeling system for convex optimization [13]. Our formulated problem can be reformed as a MILP problem.
The number of SBSs that are deployed in Fig.3.1 is four. Therefore, there are four possible caching placements for each file i, which equals to $\mu_i \in \{$ 1, 1/2, 1/3, 1/4$\}$. In terms of code rate, the feasible code rate set is $R_c \in \{$ 1/4, 1/2, 3/4, 1$\}$. More specifically, $\mu_i = 1$ represents that file i can be recovered if one cache node is accessible to the users. $\mu_i = \frac{1}{2}$ represents file i can be recovered with any loss if two cache nodes are accessible to the users at the same time.

In this thesis, we successfully use greedy Q-learning to solve the MILP problem. In practice, popularity estimation of a potentially dynamic file library has to be taken into account. However, in this work, we assume a fixed file library and static file popularity.

3.2 Q-learning

Q-learning is proposed to solve the problem as (3.1) shows. Therefore, the four basic elements have to be defined.

3.2.1 State Space

The state space of our problem is defined uniquely by the number of files that are assigned in each possible μ_i. Recall the definition of caching placement, $\boldsymbol{\mu} = \{\mu_1, \mu_2, ..., \mu_N\}$. If the number of files N in the file library is larger than the number of μ_i that can be chosen, there must be some files share the same μ_i. Besides, it can be easily understood that less code rate is assigned to the files that have larger file popularity.
Due to the cell topology used, the number of SBSs is 4. Therefore, four possible $\mu_i = (1, 1/2, 1/3, 1/4)$ can be chosen for each file. Each state can be expressed as a 2×4 matrix; the first row is $(1, 1/2, 1/3, 1/4)$, the second row is the number of files that are assigned to the same μ_i. Besides, each coded caching placement is constrained by $\sum_{i=1}^{N} \mu_i \leq$ M. Since the first row of each state is the same, each state can be simply denoted by the second row for convenience. The initial state for Q-learning is $(0, 0, 0, 0)$ always. All possible states for a file library that stores N files and a cache node that could be filled up M files can be calculated by Algorithm 1.

Algorithm 1 Calculate State Space Size for a Specific Cache Size

```
1: function StateSize(N, M)          ▷ Where N - file number, M - cache size
2:     d=0
3:     for e = 0 to 2 × M do
4:         for f = 0 to 3 × M do
5:             for  g = 0 to 4 × M do
6:                 sum1 = 1/2 × e + 1/3 × f + 1/4 × g
7:                 sum2 = e + f + g
8:                 if sum1 ≤ M and sum2 ≤ N then
9:                     d=d+1
10:                end if
11:            end for
12:        end for
13:    end for
14: end function
```

Because the radius of the coverage area of a SB is equal to the side length of the cell topology , the area that a user that can be served by one SBS is 0, the probability that a user can be served by one SBS is 0. It means that $\mu_i = 1$ or¸ $i = 1$ is not

considered. In other words, the feasible set of μ_i is $\mu_i \in \{1/2, 1/3, 1/4\}$. e, f, g are the number of files that are assigned to $\mu_i = 1/2$, $\mu_i = 1/3$, $\mu_i = 1/4$, respectively. d is the size of state space for a specific file library and cache size. It will be used to investigate the convergence of Q-learning.

3.2.2 Big Action Space

A state is defined uniquely by the number of files that are assigned to the same μ_i. Besides, the number of files that are assigned to different μ_i can be increased, decreased, or kept still. Therefore, {-1,1,0} is used to represent the actions that could be proceeded at each state element. Each action can be expressed in a 2×4 matrix, the first row is (1, 1/2 ,1/3, 1/4), the second row is the corresponding action to each state element. Only the second row is used to represent action for convenience. Each element in a state could have three possible actions. Therefore, there are $3^4 = 81$ actions in total. In our case, the probability of the user can be served by one SBS is 0. Therefore, action space will have $3^3 = 27$ different actions in total.

3.2.3 Small Action Space

A known drawback of Q-learning is "the curse of dimensionality", which is the number of states (actions) often grows exponentially with the number of state (actions) variables [12]. Meantime, the number of all state-action pairs grows a lot, which means more iteration times need to be used to make Q-table converged. A smaller Q-table size may probably speed up its convergence. With this in mind, we define a new, smalller action space. It only adjusts one state element each iteration. More specifically, the proposed small action space is $(-4, -3, -2, -1, 0, 1, 2, 3, 4)$. 9 actions can be chosen in total. The absolute value of each action is the index of the element that should be adjusted in a state. Besides, if the real value is negative, the corresponding element of the current state is reduced by 1. If the real value is positive, the corresponding element of the current state is added 1. Otherwise, it is kept still. Each action will give the information about which element of a state should be adjusted and how to be adjusted.

3.2.4 Reward

$(1 - R)$ is taken as the reward, since we aim to minimize the backhaul rate R. Besides, if the next state s' violates the cache size after applying action a to the current state s, this action should be abandoned and another action should be chosen until the subsequent state does not violate the constraint of the cache size.

3.2.5 Exploration, Exploitation, Learning Rate

In this case, there is no trade-off. Therefore, exploration probability can be set to 1 at the beginning to explore all possible states. After that, ϵ is set to decrease linearly to check whether Q-learning will converge. At last, ϵ can be set to 0 and it

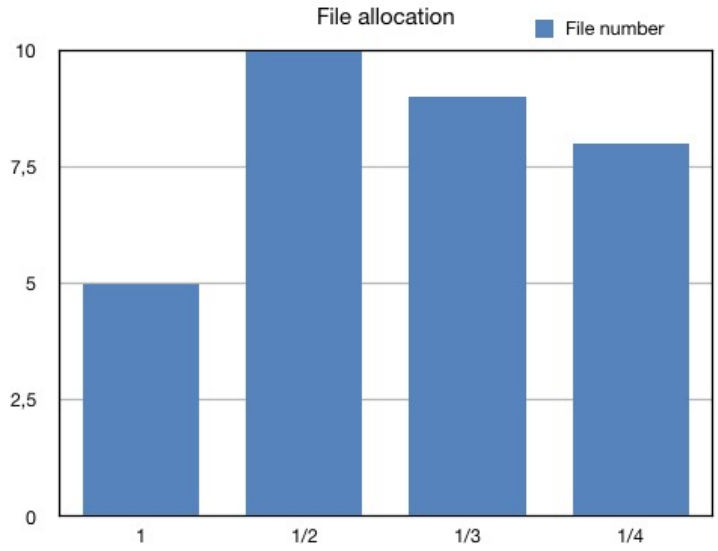


Figure 3.1: Illustration of a state

will converge around the optimal state.

Meanwhile, the learning rate α is also changing. As has been discussed in Section 2.2.3.2, the learning rate can set larger at the beginning of Q-learning. With the proceeding of Q-learning, the learning rate can be decreased since $\hat{Q}$ is closer to the real Q, which we know since the updates don't change the performance much. Then the learning step size should be small, otherwise $\hat{Q}$ will jump up and down around the real Q and never reach it. The detailed parameters values are set as given in (3.4), t is the current iteration, k is the total iteration times.

$$\begin{cases} \epsilon_t = 1, \alpha_t = 0.01 & t = 1, 2, ..., \lceil 0.75k \rceil \\ \epsilon_t = \frac{0.25 \cdot t}{k}, \alpha_t = 0.001 & t = \lceil 0.75k \rceil + 1, ..., \lceil 0.875k \rceil \\ \epsilon_t = 0, \alpha_t = 0.0001 & t = \lceil 0.875k \rceil + 1, ..., k \end{cases} \tag{3.4}$$

3.2.6 Algorithm Description

The pseudo code of proposed algorithm is tabulated in Algorithm 2.

Algorithm 2 Q-learning for MILP

function OPTIMAL STATE($N, M, \gamma_b, \mathbf{A}, \mathbf{S}, \epsilon_t, \alpha_t$)
Initialize $\hat{Q}_0(\mathbf{s}, \mathbf{a}) \leftarrow \mathbf{0}, \mathbf{s}_0 = \mathbf{0}$
for $t = 1$ to k **do**
Take action $\mathbf{a}(t)$ chosen probability by

$$\mathbf{a}(t) = \begin{cases} \arg\max \hat{Q}_{t-1}(\mathbf{s}(t-1), \mathbf{a}) & \text{w.p.} \quad 1-\epsilon_t \\ \text{random} \quad \mathbf{a} \in \mathbf{A} & \text{w.p.} \quad \epsilon_t \end{cases}$$

set $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{a}_t$
if sum(s(t))>M **then**
Go to step 2, choose an action until it doesn't violate the cache size
end if
Incure backhaul rate r can be calculated by equation 3.1,$1-R$ is taken as reward r
Update

$$\hat{Q}_t(\mathbf{s}(t-1), \mathbf{a}(t)) = (1-\alpha_t)\hat{Q}_t(\mathbf{s}(t-1), \mathbf{a}(t)) + \alpha_t\Big[r(\mathbf{s}(t-1), \mathbf{a}(t)) + \gamma_t \max_{\mathbf{a}} \hat{Q}_{t-1}(\mathbf{s}(t), \mathbf{a}))\Big]$$

end for
end function

4

Results

This chapter will show and analyze the results of two scenarios. Besides, a method that is used to increase the convergence rate of Q-learning is proposed. Moreover, the initial learning state is changed with the condition that the cache size that is allocated for each μ_i is proportional to the value of μ_i. The corresponding result is presented and analyzed. At last, the performance of a proposed algorithm that integrates implicitly finding the optimal caching placement with stationary file popularity estimation is reproduced.

4.1 Scenario 1: Zipf Distribution

The small cell topology in [2] is used. The difference between scenario 1 and scenario 2 is the distribution of file popularity. The file popularity in the first scenario is given by the Zipf distribution, which is stationary. It is given by (4.1)

$$p_i = \frac{\frac{1}{i^\delta}}{\sum_{i=1}^{N} / \frac{1}{i^\delta}} \tag{4.1}$$

where δ is the skewness of the Zipf distribution. The optimal caching placement from [2] is taken as a benchmark. The values of related parameters are the same as in [2]. The formulated problem in [2] is a convex optimization problem, which can be solved by the CVX solver.

In Fig.4.1, optimal represents the performance of caching placement that is found by CVX and ML represents the performance of the optimal caching placement using the proposed Q-learning. The red line represents the average backhaul rate that is calculated under the condition that the range of code rate $\frac{1}{N_{\mathsf{SBS}}} \leq R_c \leq 1$. The blue line represents the backhaul rate that is calculated in the circumstance that the code rate $R_c \in \{$ 1, 1/2, 3/4, 1/4$\}$.

First, it can be seen that the proposed algorithm Q-learning can achieve almost the same performance as CVX since these two curves are almost overlapped, which is exactly what we expect.

Second, it can be seen that the backhaul rate is decreased with the increase of the cache size since both curves in Fig.4.1 monotonically decrease over the cache size.

Finally, the reason why Q-learning is used to solve MILP problem is that the caching placement usually happens during the off-peak traffic time, CVX has to repetitively calculate the convex optimization at different time slots in order to find the optimal coded caching placement for the case of dynamic file popularity. However, Q-learning gives a sequence of action-state pairs. Given a state, there is always a corresponding

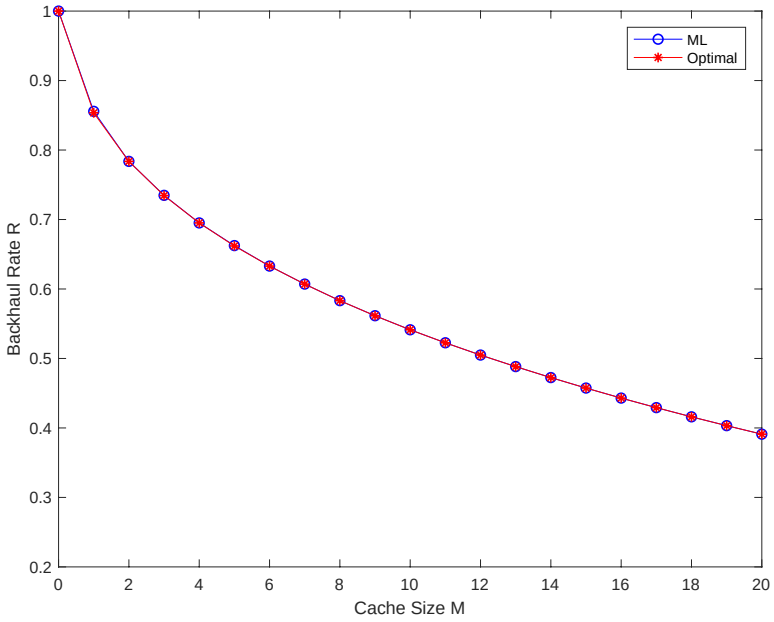


Figure 4.1: Backhaul rate as a function of cache size M, with N=200 files, δ=0.7, r=60 meters.

action that can be extracted from Q-table to be proceeded in order to achieve a good performance in the backhaul rate. Besides, for the case of dynamic file popularity, using CVX to acquire the optimal caching placement replies on a good estimation of the file popularity. However, Q-learning could update the Q-table by using the new samples from the environment. No matter how the file popularity is changing, the Q-table will adapt to the dynamic file popularity and a corresponding action always can be extracted from Q-table. After that, a suboptimal caching placement could be acquired.

4.2 Scenario 2: MovieLens

This section mainly focuses on applying the proposed Q-learning method to solving the MILP problem that occurs in reality. The file popularity in this scenario is from reality, Movie Lens, from which the file popularity is dynamic.
The file request data is from Movie Lens, which is an online recommender system created by MovieLens Research to acquire research data on personalized recommendations [10]. The recent three years of file data is used to generate 1000 time slots data. Because the data from MovieLens is daily updated, three years of data can be divided into $365 \cdot 3$ time slots. In other words, one day is taken as one time slot. The recorded data should be ordered by timestamp first and labeled by the file name before using.
The red curve and light blue curves in Fig.4.2 are the benchmarks that can be used to compare with other file popularity estimation methods. In other words, any other proposed algorithms try to achieve the same performance as the red curve or the

light blue one. Since it assumes that the future file requests are known.
First, some parameters in Fig.4.2 are listed as below,

- ML represents the performance of Q-learning.
- CVX represents the performance of CVX solver.
- 10 represents the cache size M=10.
- 20 represents the cache size M=20.
- Genie aided means tomorrow's file popularity is known, which causes the corresponding caching placement is optimal.
- Today means today's file popularity is taken as tomorrow's file popularity. Based on this assumption, the corresponding caching placement is calculated.

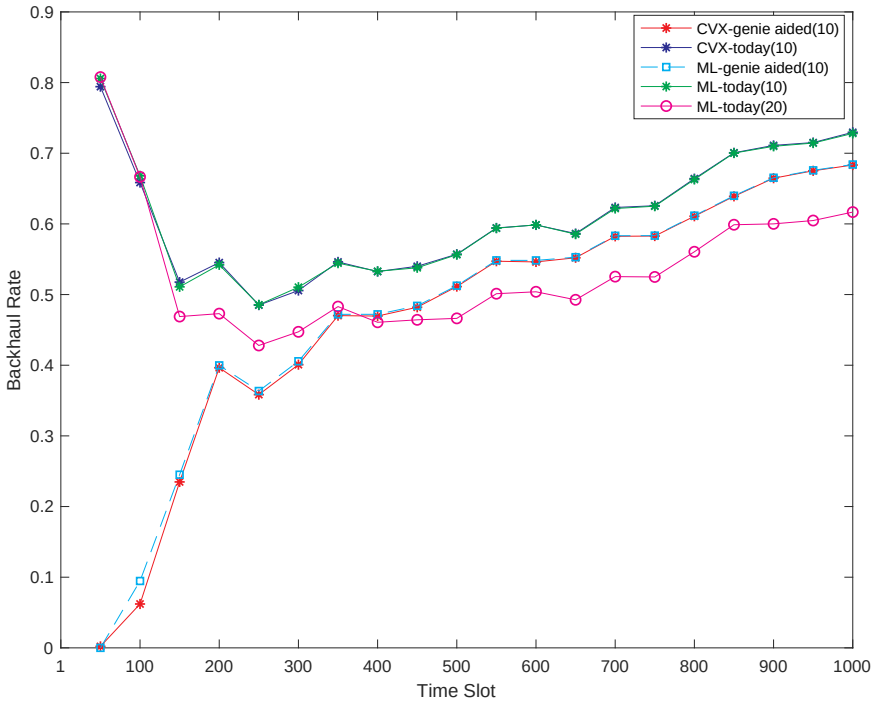


Figure 4.2: Performance comparison of CVX and Q-learning based on two kinds of file popularity estimation

Each point in Fig.4.2 is an average backhaul rate of 50 consecutive time slots. Several conclusions can be made from Fig.4.2 and the corresponding analysis is stated below,

1. If tomorrow's file popularity is assumed to be known, it can be seen by comparing the red curve and light blue curve that the performance of Q-learning is closed to CVX, which is the optimal performance that can be achieved. A similar conclusion can be applied to the scenario where today's file popularity is considered as tomorrow's file popularity.
2. The third point of the green curve is 0.0062, which is less than the corresponding point of the blue curve. Since the optimal caching placement for today's file popularity does not imply it will also be optimal for tomorrow's.

Therefore, some sub-optimal caching placement based on today's file popularity can perform better than optimal caching placement based on today's file popularity.

3. After time slots 200, all of the curves in Fig.4.2 are tending to increase since more new files are appearing but cache size is fixed.
4. Before time slots 200, no matter which methods are used, CVX or Q-learning, the backhaul rate in the scenario where today's file popularity is regarded as tomorrow's file popularity is large at the beginning since at the beginning new files bump up at the next time slot. Therefore, using today's file popularity as tomorrow's file popularity is worse. With an increment in time, more information of requested files is acquired, then the backhaul rate will decrease with a relatively more accurate file popularity prediction.

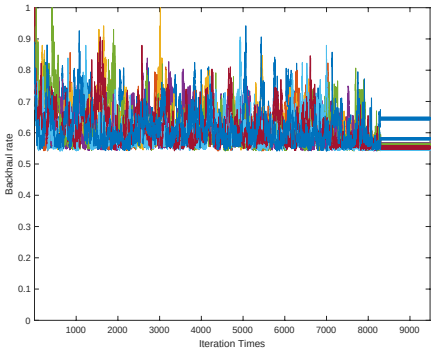
4.3 Convergence Investigation

Two action spaces are investigated to speed up the convergence rate of Q-learning. One is a big action space, which adjusts the four elements of a state at one iteration. The small action space only adjusts one state element at one iteration time. The small action space is proposed since smaller action space means less dimensionality of the Q-table and less iteration times are used to make the Q-table converged.

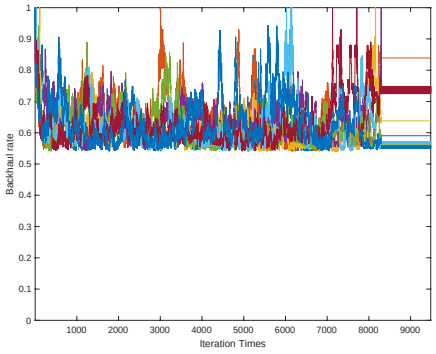
4.3.1 Big and Small Action Space

One episode is a sequence of states, actions, and rewards, which ends with terminal state or limited iteration times [12]. In this thesis, not like computer games, there is no terminal state. If iteration times are set to 5000, one episode is running out of iteration times. In order to check whether Q-learning converges to the optimal level, multiple episodes are used to achieve this goal. If all episodes generate the same converged level, which means Q-learning converges optimally. Otherwise, it doesn't.
There are two sizes of iteration times used for investigation. One is 2 times of state space size. The other is 6 times of state space size. The number of learning episodes is always 8.

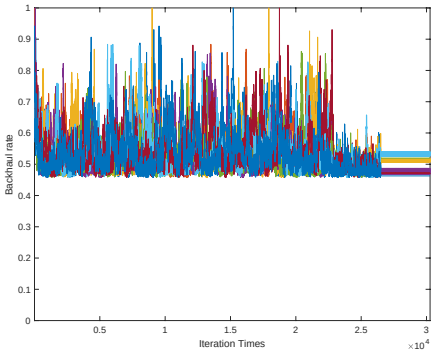
1. Small iteration times is applied to these two action spaces as Fig.4.3 shows. The left column is using big action space, the right column represents using small action space. The cache size for these three plots in each column in Fig.4.3 are 10, 15, 20 respectively.
   It seems that the big action space has better performance than the small one since using the big action space converges almost at the same level by running the same iteration times as small action space. Because Q-learning takes more iterations to explore all feasible states when the size of action space is decreased. The benefit of using a small action space is that the dimension of Q-table is smaller, the time to finish the learning procedure is smaller. The running time for the small one is 245 sec, and the big one is 563 sec.
2. Increase the iteration times to 6 times of the size of the state space. The results are displayed in Fig.4.4 shows. Both action spaces behave better than that of



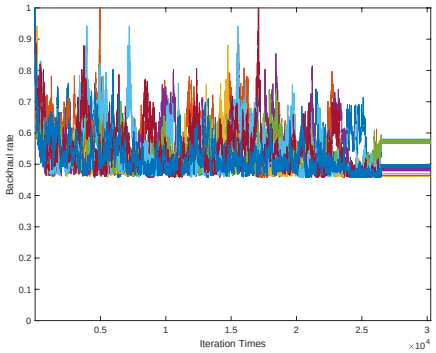
(a) Big action space, M=10



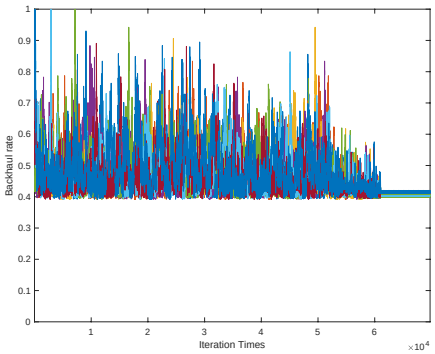
(b) Small action space, M=10



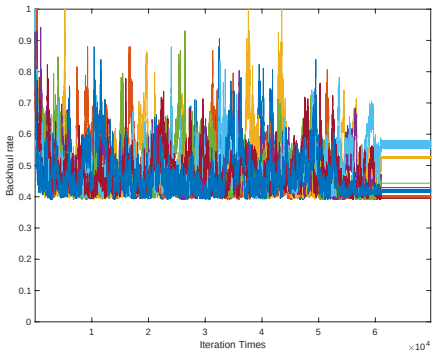
(c) Big action space, M= 15



(d) Small action space, M=15



(e) Big action space, M=20



(f) Small action space, M=20

Figure 4.3: Convergence rate of using two action space for cache size M= 10, 15, 20, respectively.

using small iteration times. Since more iteration times are used, both of them could explore all state-action pairs and converge at a similar sub-optimal level.

The small action space has a smaller dimensionality of the Q-table but still requires more iterations. One possible explanation is that the big action space requires fewer iterations to explore all possible states since it can adjust four elements in a state at one iteration. The subsequent cost is the real time to run the code for each iteration actually is longer than using the small action space. For the small action space, more iterations are used to explore the possible states since it only changes one element in a state at one iteration. The benefit of this procedure is that the real-time to run the code for each iteration is shorter.

4.3.2 Increasing the Convergence Rate

It is necessary to reach all state-action pairs for guaranteeing the convergence of Q-earning [12]. Both action spaces take the risk of reaching the same states multiple times since the exploration parameter ϵ is set to 1 to explore all possible states at the beginning of the Q-learning.

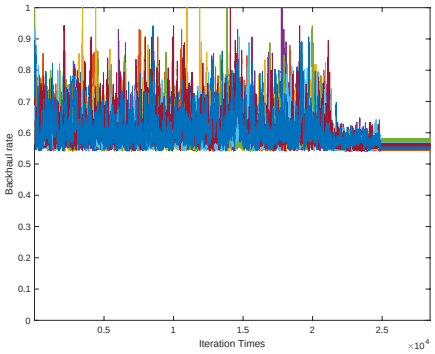
There are two ideas to speed up the convergence rate. One is constraining the action space at the beginning to reduce the times that some states are revisited. Another one is the initial state is changing to $(0, M, \lfloor \frac{3}{4} \cdot M \rfloor, M)$. The cache size that is allocated for each μ_i is proportional to the value of μ_i. Since if the file popularity is larger, it is more likely to be requested in the future. Therefore the lower code rate should be assigned to the files that have larger file popularity. Moreover, the cache space that is assigned to lower code rate should be larger in order to store more files that have larger file popularity.

4.3.2.1 Dynamic action space

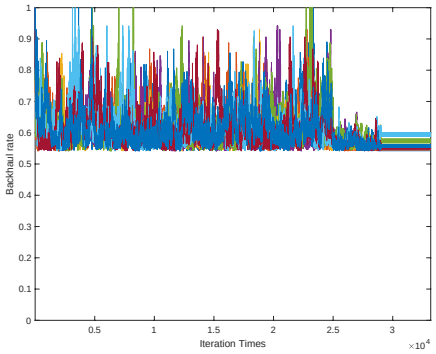
The detailed procedure is that the action space is limited at the beginning. For the big action space, any actions that include -1 are omitted. For the small action space, $(-1, -2, -3)$ are omitted.

1. The performance is illustrated in Fig.4.6. The convergence rate is faster than before in terms of iteration times. Since iteration times are 300, which is smaller compared to the iteration times that are used before.
2. It also can be seen that the big action space takes less iterations to converge. The converging point for the big action space is around 30 and for the small action space is around 60. The reason is that the big action space can adjust four elements at the same time.
3. The performance of using the big action space seems better than the small one for a fixed cache size. Take the cache size M=10 for example, the final converged level of using the big action space is lower than that of using the small action space.

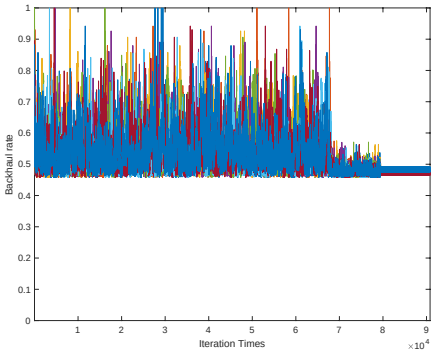
Moreover, the performance of the improved algorithm of using the small action space still has quite good performance in Fig.4.5.



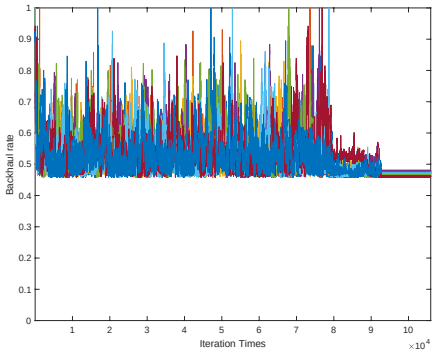
(a) Big action space, M=10



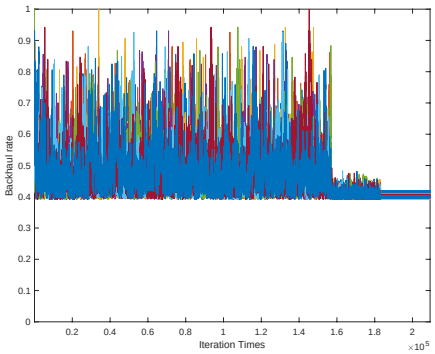
(b) Small action space, M=10



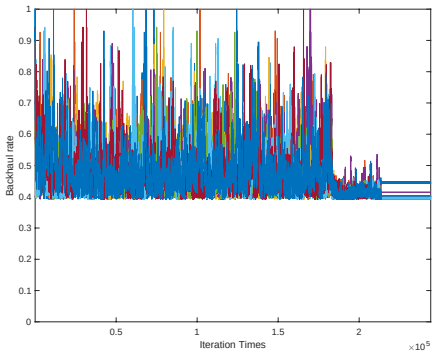
(c) Big action space, M= 15



(d) Small action space, M=15



(e) Big action space, M=20



(f) Small action space, M=20

Figure 4.4: Convergence rate of using two action space for cache size M= 10, 15, 20, respectively.

4.3.2.2 Proportional Initial State

Finally, the initial state is changed from $(0, 0, 0, 0)$ to $(0, M, \lfloor \frac{3}{4} \cdot M \rfloor, M)$ with a dynamic action space, the convergence rate curves for these two sets are illustrated as Fig.4.7.

It can be seen that initializing the algorithm with the proportional caching placement

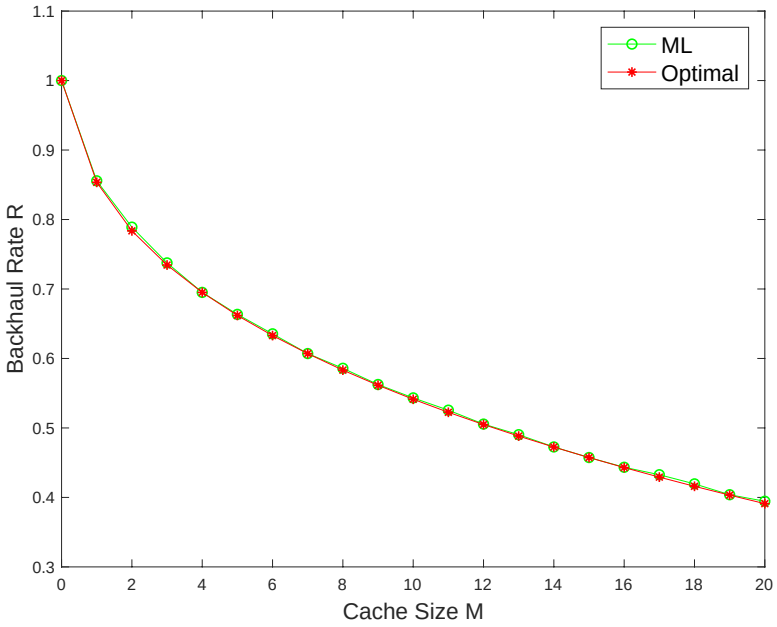
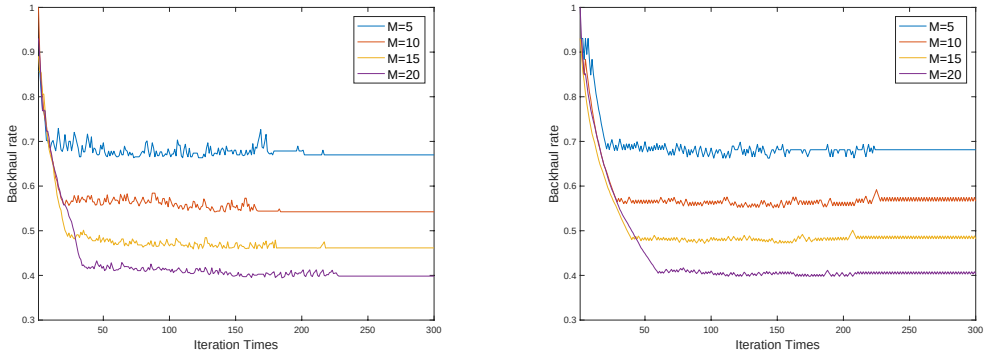


Figure 4.5: Performance of using improved algorithm



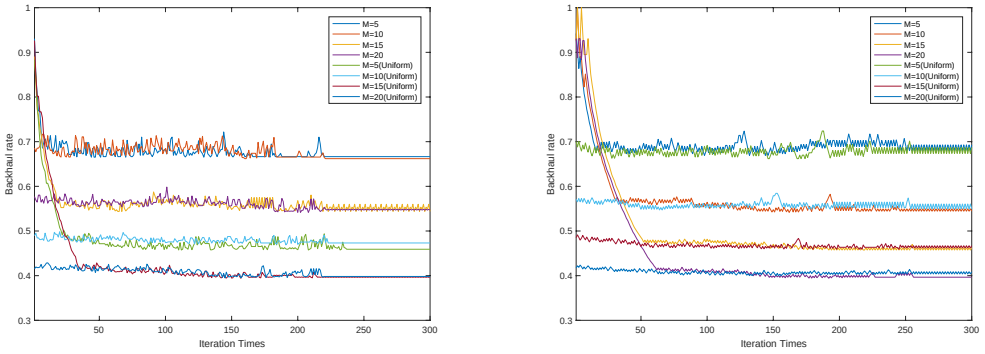
(a) Big action space, M=5,10,15,20 **(b)** Small action space, M=5,10,15,20

Figure 4.6: Convergence of two different changing action space

significantly improves the rate of convergence.

4.4 Integrated File Popularity Estimation and Caching Placement Implicitly

The results of the above two scenarios show that the proposed Q-learning can solve the formulated MILP problem. However, it still relies on a good estimation of the file popularity. Besides, it neglects the refreshment cost of cache nodes at each time slot. Therefore, the minimum backhaul rate with an optimal caching placement at each time slot does not imply that the cumulative backhaul rate within several time



(a) Big action space, M=5,10,15,20 **(b)** Small action space, M=5,10,15,20

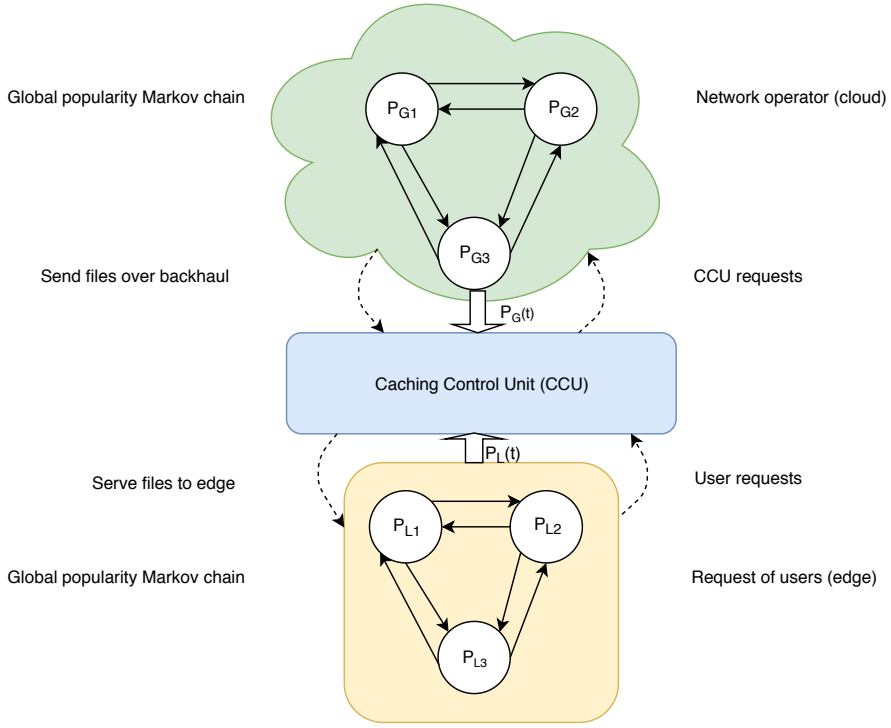
Figure 4.7: Convergence of two different changing action spaces with a proportional initial state

slots is minimized. It is interesting to investigate whether file popularity can be coupled with finding the coded caching placement in order to achieve the minimal cumulative backhaul rate. Inspired by [11], we couple the file popularity is coupled with the state space to affect the subsequent action, the coded caching placement.

We consider the scenario in [11] corresponding to a HetNet where each cell is served by an SBS that can store M files. The size of the file library is N. It is assumed that each SB is equipped with a caching control unit (CCU), which decides the files that should be cached. If users can not get the files they want, the service will be conducted by the backhaul link. The network structure is shown in Fig.4.8. Besides, it is assumed that global popularity profiles and local popularity profiles can be modeled as Markov processes. There is an underlying transition probability for global popularity and local popularity, respectively, which is assumed unknown. At every time slot, after users' requests are revealed, each SBS only has the observation of its local file popularity. The network operator could reach several SBSs and get global file popularity by concatenating the local file popularity. The network operator will transmit the global popularity to each SBS. Every SBS with CCU will decide the caching placement for the next time slot based on the global popularity, the local popularity, and the caching placement of the last time slot. The caching placement of last time slot is considered since refreshing cache nodes also deprecates the network performance.

The proposed algorithm uses Q-learning to find the optimal caching placement to minimize the cost, which has three terms to account for. The first one is the cost of refreshing cache contents; the second one is the mismatch between caching placement and local file popularity. the third one is the mismatch between caching placement and global popularity. They are weighted by λ_1, λ_2, and λ_3. An action is a binary vector with size of N, which is the number of files in the file library. 0 means the file is not cached and 1 means the file is cached. A state is defined as a concatenation of the global file popularity, the local file popularity, and the action of last time slot.

Global file popularity and local file popularity are modeled by a Zipf distribution.



(1).pdf

Figure 4.8: Schematic of network structure

The global file popularity is given by,

$$[P_G^i]_f = \frac{1}{f^{n_i G} \sum_{l=1}^{N} / l^{\eta_i^G}} \tag{4.2}$$

The local file popularity is similar. $\eta_1^G = 1$, $\eta_2^G = 1.5$, $\eta_1^L = 0.7$, $\eta_2^L = 2.5$.
It is assumed that there are two states of global file popularity and local file popularity, respectively. The assumed underlying transition probability for the global popularity and the local popularity are as follows,

$$P^G = \begin{bmatrix} p_{11}^G & p_{12}^G \\ p_{21}^G & p_{22}^G \end{bmatrix} = \begin{bmatrix} 0.8 & 0.2 \\ 0.75 & 0.25 \end{bmatrix} \tag{4.3}$$

$$P^L = \begin{bmatrix} p_{11}^L & p_{12}^L \\ p_{21}^L & p_{22}^L \end{bmatrix} = \begin{bmatrix} 0.6 & 0.4 \\ 0.2 & 0.8 \end{bmatrix} \tag{4.4}$$

We consider Q-learning is used to find the optimal caching placement by learning the underlying transition probability of global popularity and local popularity. The optimal caching placement calculated by value iteration or policy iteration with knowing the transition probability is taken as a benchmark as Fig.4.9 shows. SC1, SC2, SC3 represent different scenarios, which are specified by the value of cost parameters, λ_1, λ_2, and λ_3. The horizontal curves in Fig.4.9 represent the minimal cost in different scenarios, which is calculated by one of the DP algorithms, policy iteration algorithm. The value iteration algorithm can be used to double-check the result that is produced by the policy iteration algorithm. It can be seen that the cost

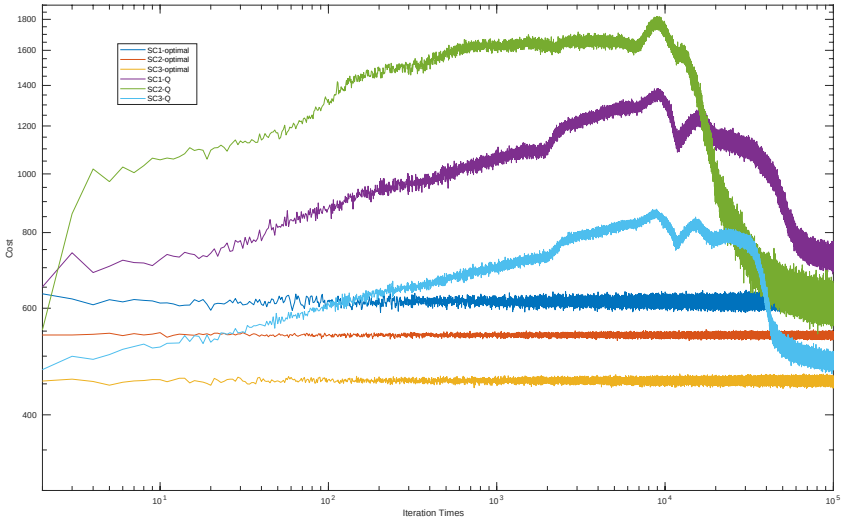


Figure 4.9: Policy iteration VS Q-learning

of caching placement that is calculated by Q-learning without knowing transition probability can reach a similar level as policy iteration one, which represents the minimal cost.

5
Conclusion

In this thesis, greedy Q-learning is proposed to solve an MILP problem, which is a function of the minimization of the backhaul rate. More specifically, the goal of solving the formulated MILP problem is to find an optimal caching placement using MDS codes in a scenario where file popularity is stationary in order to minimize the average backhaul rate. Besides, the convergence rate can be speed up by adjusting the action space and changing the initial state. It is shown that Q-learning can solve an MILP problem well, which indicates that finding the optimal coded caching placement for the case of stationary file popularity is feasible by using Q-learning. Moreover, we also reproduced [11] to show it is possible to apply Q-learning to the case of stationary file popularity by considering the cost of refreshing the cache nodes to acquire the minimum cumulative cost. An interesting area of future research is adapting the proposed Q-learning algorithm to the case of dynamic file popularity profile with considering the cost of refreshing the cache nodes in order to find the optimal coded caching placement and minimize the cumulative backhaul rate.

Bibliography

[1] S. Kumar, A. Graell i Amat, E. Rosnes, and L. Senigagliesi, "Private information retrieval from a cellular network with caching at the edge," *IEEE Transactions on Communications*, vol. 67, no. 7, pp. 4900–4912, July 2019.

[2] V. Bioglio, F. Gabry, and I. Land, " Optimizing MDS Codes for Caching at the Edge ." *IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2015.

[3] "Ericsson mobility report," *White report, Ericsson*, 2018.

[4] M. Ji, G. Caire, and A. F. Molisch, "Fundamental limits of caching in wireless d2d networks," *IEEE Transactions on Information Theory*, vol. 62, no. 2, pp. 849–869, 2015.

[5] G. Paschos, E. Bastug, I. Land, G. Caire, and M. Debbah, "Wireless caching: technical misconceptions and business barriers," *IEEE Communications Magazine*, vol. 54, no. 8, pp. 16–22, August 2016.

[6] J. Kangasharju, J. Roberts, and K. W. Ross, "Object replication strategies in content distribution networks," *Computer Communications*, vol. 25, no. 4, pp. 376–383, 2002.

[7] J. Pedersen, A. Graell i Amat, I. Andriyanova, and F. Brännström, "Optimizing mds coded caching in wireless networks with device-to-device communication," *IEEE Transactions on Wireless Communications*, vol. 18, no. 1, pp. 286–295, Jan 2019.

[8] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Transactions on Information Theory*, vol. 60, no. 5, pp. 2856–2867, 2014.

[9] N. Zhang, K. Zheng, and M. Tao, "Using grouped linear prediction and accelerated reinforcement learning for online content caching," *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, May 2018.

[10] GroupLens. Movielens latest datasets. [Online]. Available: https://grouplens.org/datasets/movielens/latest/.

[11] 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, Feb 2018.

[12] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[13] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming, version 2.1," http://cvxr.com/cvx, Mar. 2014.