An Approach Based on Reinforcement Learning for Efficient Process Cache Management in Operating Systems

B.Tech Project Report

Submitted in partial fulfillment of
the requirements for the award of the degree
of
Bachelors of Technology
in
Department of Computer Science and Engineering
by

Sumanyu Muku, Vaibhav Kumar, Vikas Tomar
(2K16/CO/322, 2K16/CO/346, 2K16/CO/352)

under the guidance of

Dr. Rajni Jindal
Head of department
Department of Computer Science and Engineering

DEPTT. OF COMPUTER SCIENCE AND ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY,DELHI
December, 2019

CERTIFICATE

This is to certify that Project Report entitled **An Approach Based on Reinforcement Learning for Efficient Process Cache Management in Operating Systems** submitted by Sumanyu Muku (2K16/CO/322), Vaibhav Kumar (2K16/CO/346), and Vikas Tomar (2K16/CO/352) for partial fulfillment of the requirement for the award of degree Bachelors Of Technology (Computer Science and Engineering) is a record of the candidate work carried out by them under my supervision.

Dr. Rajni Jindal
Head of Department,
Project Guide,
Department of Computer Science and Engineering
Delhi Technological University

DECLARATION

We hereby declare that the project work entitled **An Approach Based on Reinforcement Learning for Efficient Process Cache Management in Operating Systems** which is being submitted to Delhi Technological University, in partial fulfillment of requirements for the award of degree of Bachelor Of Technology (Computer Science and Engineering) is a bonafide report of B.Tech Thesis carried out by us. The material contained in the report has not been submitted to any university or institution for the award of any degree.

Sumanyu Muku (2K16/CO/322),
Vaibhav Kumar (2K16/CO/346),
Vikas Tomar (2K16/CO/352)

ACKNOWLEDGEMENT

We would like to express our gratitude and appreciation to all those who gave us the support to complete this project. A special thanks to our mentor and project guide, **Dr. Rajni Jindal** , Head of Department, Computer Science and Engineering, Delhi Technological University for her inspiring guidance, constructive criticism and valuable suggestion throughout this project work. We also extend our sincere thanks to all our friends who have patiently helped us in accomplishing this undertaking.

Sumanyu Muku (2K16/CO/322),
Vaibhav Kumar (2K16/CO/346),
Vikas Tomar (2K16/CO/352)

Contents

List of Figures

1 Abstract

Adaptive Cache replacement strategies have shown superior performance in comparison to classical strategies like *LRU* and *LFU*. Some of these strategies like *Adaptive Replacement Cache (ARC)*, *Clock with Adaptive Replacement (CAR)* are quite effective for day to day applications but they do not encode access history or truly learn from cache misses. We propose a reinforcement learning framework, *RLCaR* which seeks to tackle these limitations. We use TD 0 model-free algorithms like Q-Learning, Expected SARSA and SARSA to train our agent to efficiently replace pages in cache in order to maximize the cache hit ratio. We also developed a memory cache simulator in order to test our approach and compare it with LRU and LFU policies.

2 Introduction

Operating systems like Linux, Unix and Windows implement caching techniques for quick access of memory pages. Cache is a special portion allocated in the memory or disk for quick access of data. Pages stored in the cache need to be replaced for allocation of new pages. This process of evicting existing pages from cache to make space for new pages is done by a page replacement policy like LRU (Least recently used page) or LFU (Least frequently used page). If the page requested by operating system is currently allocated in cache, it is said to be a cache hit. If the page is not allocated in cache, it needs to be allocated. Such a scenario is called a cache miss. In order to maximize performance, cache replacement is done in a way to maximize the cache hit ratio.
In this project we present a Reinforcement Learning based cache replacement technique. We call it RLCaR - Reinforcement Learning Cache Replacement. We train an agent with a goal to optimize the cache hit ratio. Agents learns an optimal policy π^* which helps it optimize (maximize) an objective (cache hit ratio). We use reinforcement learning to train our system rather than conventional machine or deep learning because this way, we can create a generalized agent that can work in various scenarios without relying on a plethora of data to train and evaluate. We describe our contributions as follows:

- A novel reinforcement learning for the task of page replacement
- A cache simulation model to train and test our approach
- A probabilistic simulation for operating system's page request routine

- Comparative analysis of LRU, LFU and RLCaR

The structure of this project report is as follows: We first present all the preliminaries required to understand our work, followed by our approach, results and code. Finally, we mention the future work and conclude our report.

3 Related work

There have not been many works which utilize learning the changing patterns in cache replacement mechanism. [Vietri et al., 2018] is one of the few works which try to learn the eviction strategy. It tries to achieve a balance between *recency* and *frequency*. It uses both LRU and LFU policies for its functioning. **LeCAR** inviolves regret minimization using reinforcement learning. This algorithm maintains and learns a probability distribution for both LRU and LFU. Weights are associated W_{LRU} and W_{LFU} to denote the regret associated when a page is evicted using the particular mechanism e.g if a miss occurred due to the eviction of a page by LFU then the weights are updated in the way such that next time more importance is given to eviction of the page by LRU. [Surbhi, 2018] is another work which tries to employ UCB1 algorithm for page cache replacement. It tries to tackle the problem of loss of past access information and caching of large data with infrequent access patterns. This algorithm treats the page cache replacement as a multi arm bandit problem. It tries to achieve a balance between exploration and exploitation. It can both greedily or randomly remove a page based on reward obtained. The reward depends on two parameters:- 1) n_j which is the number of times a page j has been evicted and 2) T which is the number of times an action is taken and a reward is received. The eviction strategy of [Alabed, 2019] treats each cached object as a state of the Reinforcement Learning Algorithm. When the cache is full and a page eviction is required this algorithm stops cache execution and analyzes all the states. It then chooses the action of evicting an object or not based on past access patterns. If a page eviction later results in a cache miss then the agent is penalized with a high negative reward otherwise it is given a positive reward.

4 Preliminaries

In this section we'll be discussing about the pre-requisites required to understand and go through this report, in order to maximise the take-away. Reinforce-

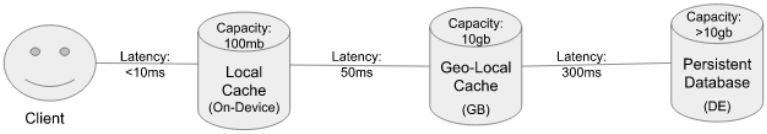


Figure 1: General Caching Mechanism

ment Learning for efficient Cache management requires one to have a knowledge certain basic concepts such as: Page Cache Management, Eviction/Page Replacement, Least Recently Used and Least Frequently used page replacement algorithms. In the following section, we'll cover Reinforcement learning, Markov Decision Process, Model free and Model Dependent learning, Q-learning, SARS and Expected SARSA. We've tried to cover each topic in sufficient depth so that the application of our project becomes as much clear as possible.

4.1 Page cache management

Cache Memory is a type of memory which is used to speed up the data access process. Caching tries to achieve a balance between speed and size. Cache memory is small but the latency produced while accessing data is the least of all storage devices. Page cache is a type of disk cache which is responsible for caching pages which are derived from accessing files in the secondary storage. The operating system is responsible for managing the page cache using page memory management. The kernel keeps the page cache in the unused section of the main memory. Cache Management task is performed by a Cache Manager (CM). Cache manager involves following tasks: 1) deciding whether to cache an object or not 2) time for which an object should be cached 3) which object to remove when the cache is full.

4.2 Caching Strategy

Caching strategy is used to determine whether an object is supposed to be cached or not. If the requested page is found in the cache then there is a cache hit otherwise it is a cache miss. When a cache miss is occurred them the missed page is to be retrieved from the secondary slower storage. These are also known as cache coherence strategies. There are 3 kinds of these strategies: 1) Write Through 2) Write Back 3) Write on Read.

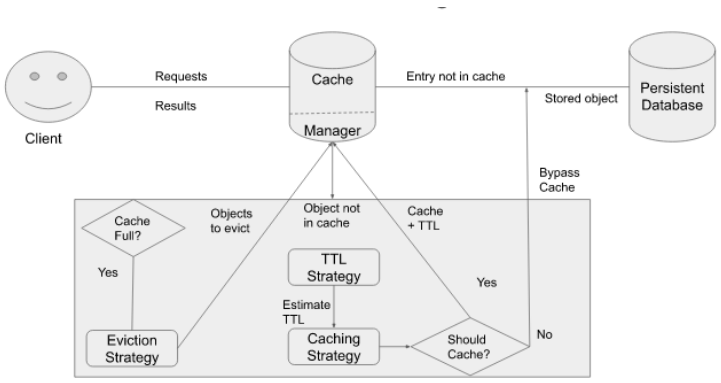


Figure 2: Cache Management Flowchart

4.2.1 Write Through

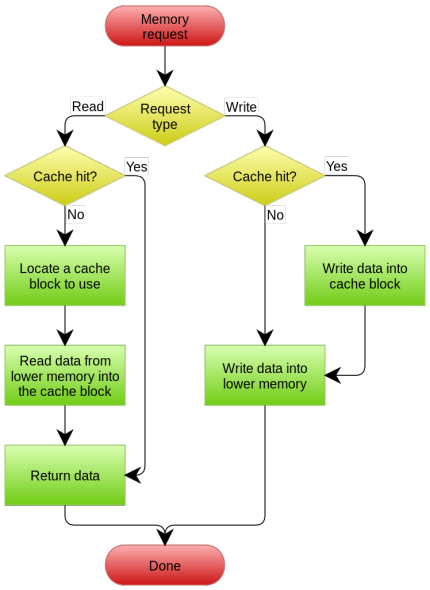
In write through the data is written to both cache and the backend slower storage. It is good for processes that have read after write access patterns. However, it is quite slow to write as write operation occurs simultaneously at two places.

4.2.2 Write Back

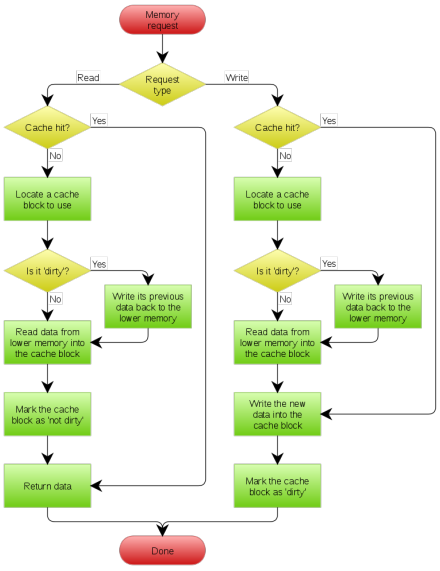
Write back is a more complex strategy which dosen't write to the back-end immediately. It keeps track of locations which were written over and marks them as dirty, When the pages are evicted from cache then they are written to the back-end store. Read operations on an average tend to be slower as if a altered page is to be read then first it is written over to the backend store and then is read from there. It is also known as lazy write.

4.2.3 Write on Read

In this strategy the data is brought into the cache when a read operation is invoked. When there is a cache miss then the data is read from the slower backend and is also brought into the cache at the same time. The data object remains in the cache until TTL. There is always a chance of cache miss in this mechanism when a read operation is called after a write operation.



(a) Write Through Mechanism.



(b) Write Back Mechanism

4.3 TTL Strategy

Time To Live (TTL) strategy is used to ensure data freshness and make sure not to clutter cache with unwanted objects. TTL is an integer which denotes

the time for which an object is kept in the cache after which it is evicted. This strategy is used in order to tackle the problem of data staleness. Data staleness occurs when an object is updated in the backend storage but not in the cache or when the write operation somehow fails in cache but succeeds in the backend store. Underestimating the value of TTL can lead to various cache misses whereas overestimating it can clutter it with unwanted objects.

4.4 Eviction Strategy

A cache eviction strategy is used when the cache is full. It is used to determine which data object to remove to make space for another data object so that cache hit ratio remains maximum. There are various cache eviction strategies out there. Some of the following have been explained below:-

4.4.1 First In First Out (FIFO)

FIFO algorithm is used for evicting objects which were first inserted in the cache irrespective of the access patterns. It is useful for events which are linear in occurrence e.g. it is common to go back to a page which was visited recently but it is uncommon to go to a page which was accessed quite earlier.

4.4.2 Least Recently Used (LRU)

LRU evicts the data item which was least recently used. It keeps track of the access history of each data item. It is a popular algorithm due to its simplicity and applicability in real world scenarios. Generally "age bits" are tracked and the LRU algorithm makes decision on the basis of these "age bits".

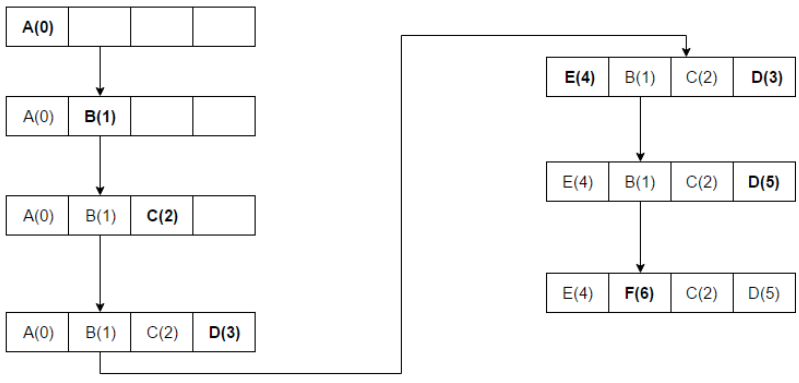


Figure 4: Demonstration of LRU for a sequence of data items

4.4.3 Least Frequently Used (LFU)

LFU keeps track of the frequency of the data items. It evicts those items which have been least frequently used. It is similar to LRU, but instead of keeping track of the access history of a data item it keeps track of how many times the item was accessed. It is useful in situations where popular data item is accessed throughout the application life cycle. For e.g. the icon of a website.

4.4.4 CLOCK

The CLOCK algorithm is similar to second chance algorithm but instead of moving the data items again and again to the end of the list, it implement the cache as a circular list. Each data item has a R bit. The hand or the iterator of the circular list points towards the oldest page. When a page fault occurs, then the hand checks the R bit of the data item it is referencing to. If the R bit is 0 them the page is evicted. If the R bit is 1 then the bit is cleared and the hand is moved to the next data item. This continues until it finds a data item whose bit has been cleared and it evicts that page. This algorithm tries to find a page which is both old and unused in the clock interval.

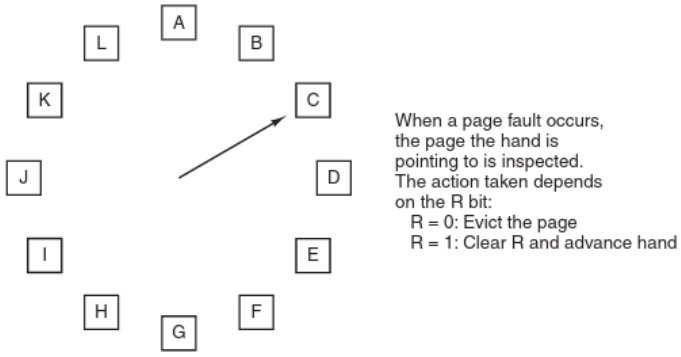


Figure 5: Demonstration of Clock Algorithm

4.4.5 Most Recently Used (MRU)

MRU algorithm discards the most recently used item in contrast to LRU. It is useful in applications which are processed sequentially once.

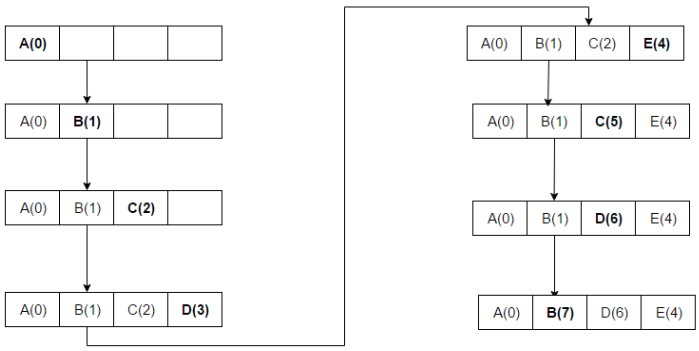


Figure 6: Demonstration of MRU algorithm for a sequence of data items.

4.4.6 Random Replacement (RR)

RR algorithm randomly selects a data item for eviction. It dosen't keep any information regarding the access history of the data item.

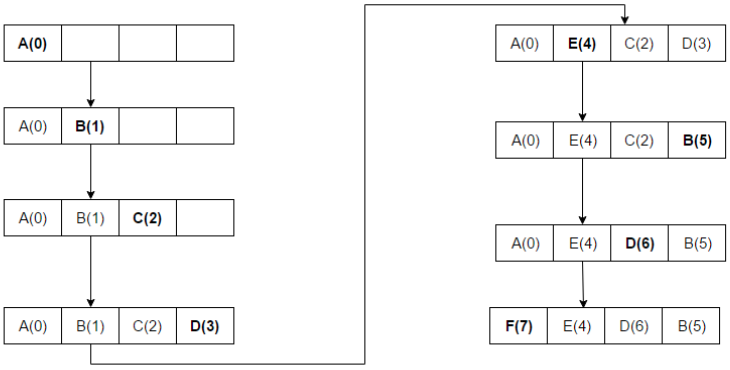


Figure 7: Demonstration of RR algorithm for a sequence of data items.

4.4.7 Low Inter-reference Recency Set (LIRS)

LIRS is an improvement over LRU algorithm. It uses reuse distance as its metric for choosing the page for eviction. Reuse distance is the number of distinct pages accessed between two references of a page. For a page which was accessed for the first time, the reuse distance is infinite. In order to take recency into consideration, the LIRS algorithm uses larger value between reuse distance and recency of a page. This metric is known as *RD-R*. So pages are ranked according to the RD-R and the worst one is then evicted.

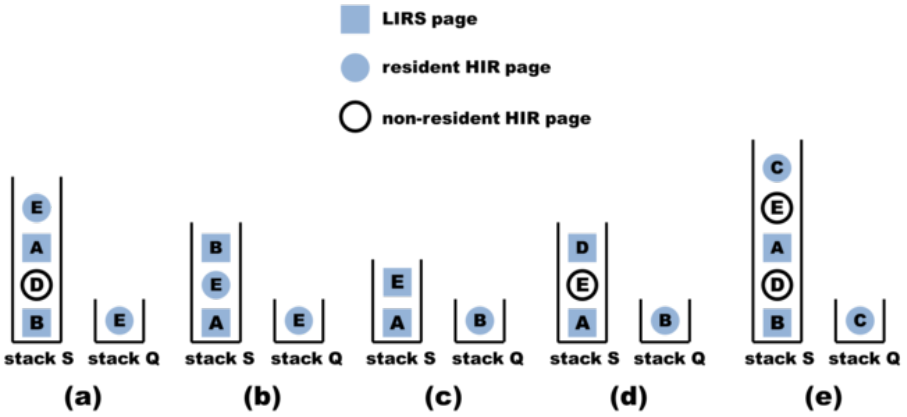


Figure 8: Demonstration of LIRS Mechanism

4.4.8 2Q

[Johnson and Shasha, 1994] proposed the 2Q page replacement algorithm. 2Q uses two buffers:- 1) a secondary buffer (A_s) 2) a main buffer (A_m). When a page is referenced for the first time it is put at the head of the A_s. When a page present in A_s is referenced then it is put at the head of A_m queue which is itself a LRU queue. When a page is to be removed from A_m the least recently used one is removed and put in the A_1 queue. From the A_s queue, the page present at the tail is removed.

4.4.9 Adaptive Replacement Cache

The ARC algorithm was proposed by [Megiddo and Modha, 2003].The ARC algorithm employs multiple queues for tracking both recency and frequency. The queues used in this algorithm are as follows:

- A T_1 queue for caching recent entries.
- A T_2 queue for caching entries on the basis of frequency.
- A B_1 queue which is a ghost queue for keeping the metadata of the pages evicted from T_1.
- A B_2 queue which is similar to B_2 queue but is used for storing the metadata for T_2.

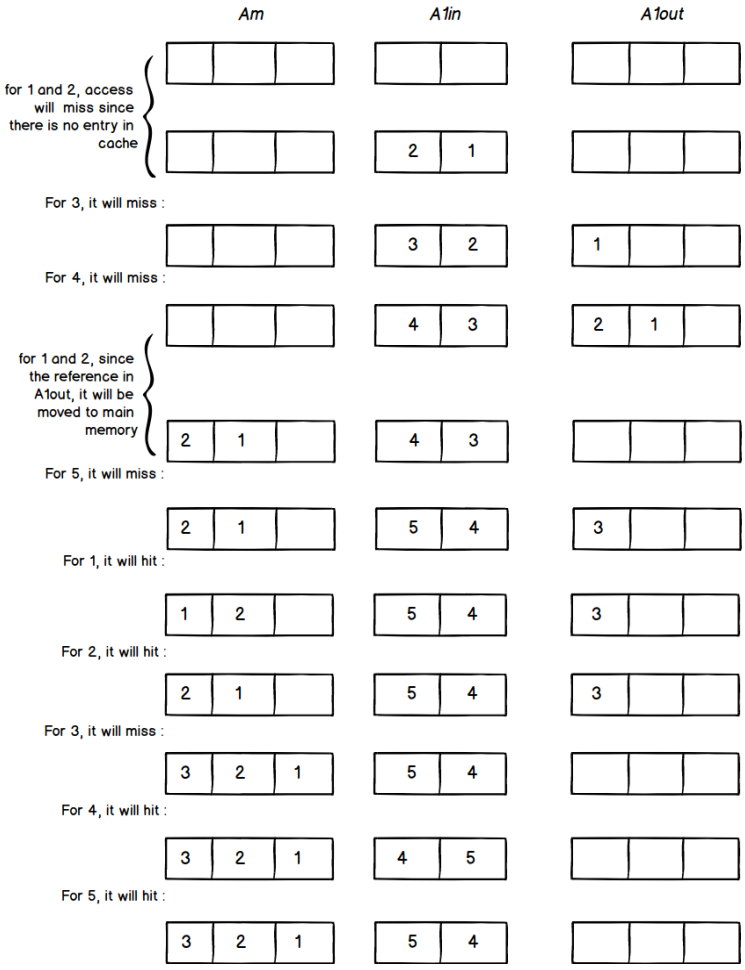


Figure 9: Demonstration of 2Q algorithm.

4.4.10 Clock with Adaptive Replacement

The CAR algorithm was proposed by [Bansal and Modha, 2004]. CAR like the ARC algorithm uses multiple lists for tracking recency and frequency. It also ensures that there is scan resistance and lock contention is achieved. The T_1 and T_2 queues used in this case are circular lists.

5 Reinforcement Learning

Machine Learning is broadly divided into three categories: Supervised Learning, Unsupervised Learning and Reinforcement Learning. Supervised learning involves a function that maps an input value to an output target value based on

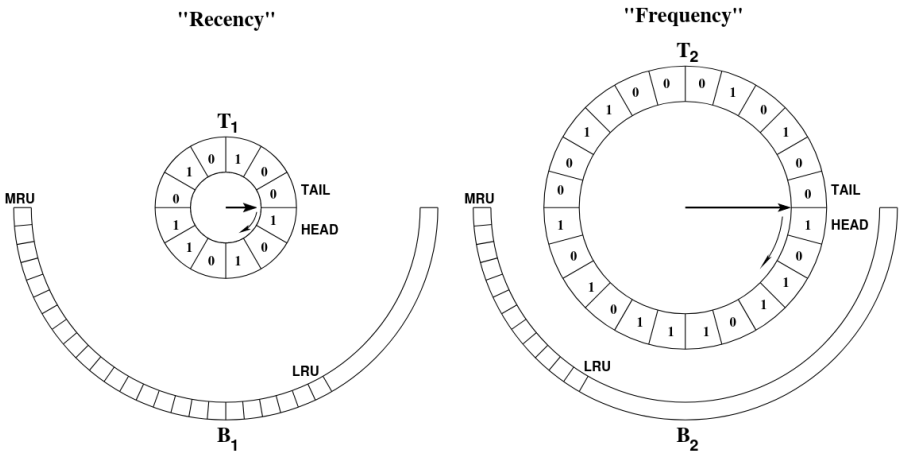


Figure 10: Demonstration of the CAR algorithm.

example input-output pairs of value. In other words we are provided with a labeled training data. In contrast to this, unsupervised learning methods aims to extract relevant features from unlabelled data, encapsulating their useful properties with minimum errors. The third category, which we'll be discussing in detail in this report further is called the Reinforcement Learning, which has an agent that learns about the constraints and decision in real-time while interacting with the environment.

In the past few years, Reinforcement Learning (RL) or adaptive (or approximate) dynamic programming (ADP), came up as an effective tool for evaluating complicated sequential decision-making problems. Although many influential studies that were conducted in this area within the artificial intelligence (AI) network, more recently, it has been a point of attraction for optimization researchers because of many notable achievement memories from operations control. The fulfilment of Reinforcement Learning is because of its sturdy mathematical backgrounds.

A recurrent problem in Machine Learning involves making the agent learn when and what action to perform in order to achieve the desired goal or complete certain task in a given environment. During the learning phase the agent constantly interacts with the

5.1 Reinforcement Learning Problem

According to [Andrew, 1999], Reinforcement Learning is defined as a learning method where the learner or the agent constantly interact with the environment in an hit and trial manner and aims at minimizing the errors encountered. Unlike other machine learning methods, the agent isn't told beforehand what actions to perform. Instead, it is the responsibility of the agent itself to explore and exploit it's environment and figure out the suitable action. This process is derived by the rewards obtained by the agent and the goal of the agent in every step is to maximise the future rewards(or statistically speaking, the highest sum of expected reward), usually in search for a objective or a target space which is represented numerically by a large reward.

Every reinforcement learning problem possess similar modules: an agent or learner, a reward function or teacher, a performance measurement depicting how well the agent is doing based upon tests represented numerically by rewards and few more variable.

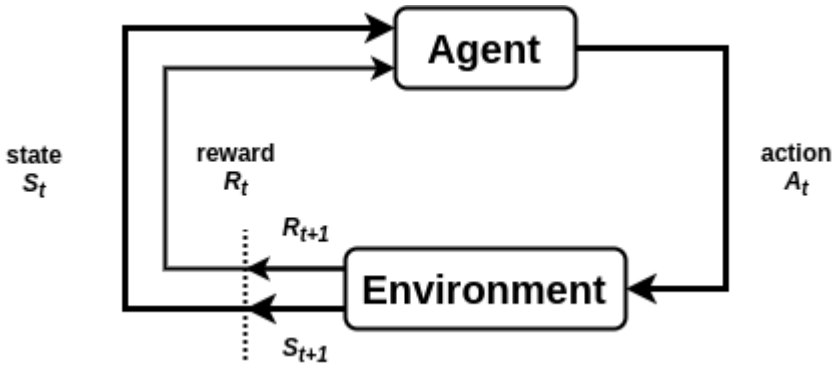


Figure 11: The reinforcement learning framework

Figure 11 depicts the fundamental concept of reinforcement learning. At every individual time step t = 1,2,3,... the agent being in certain state $s_t \in S$ and takes one of the possible available actions $a_t \in A(s)$. The action will cause a change in environment and takes it to a new state $s_t + 1$. In addition to this, a reward $R_t + 1$ is received from the environment which serves as a feedback about the goodness of the taken actions a_t in state s_t.

Alongwith agent and environment, there are four additional sub-components of a reinforcement learning system: a *policy*, a *reward function*, a *value function*, and optionally, a *model* of the environment.

5.1.1 Environment

An environment comprises a world for the agent or the learner to act and learn. For example in any arcade game, our player would be an agent and it's surroundings(say a maze) in the particular level or episode would be considered as it's environment. In order to define an event within the environment, we've considered the state to be a vector.

5.1.2 Policy

A policy specifies the way of behaving for the learning agent at any given time. Roughly speaking, a policy maps the various states of the environment to possible actions that can taken when the agent is in those states. At times the policy may be basic function or simply a lookup table, whereas in some cases it may involve extensive computation such as a proper search process. The policy is the heart and soul of a reinforcement learning agent as it alone is enough to determine behaviour of the agent. Policies can be either deterministic or stochastic.

Deterministic policy is formulated as:

$$\pi(s_t) = a_t \quad , \quad s_t \in S \; and \; a_t \in A \tag{1}$$

Stochastic policy is formulated as:

$$\pi(a_t|s_t) = P[A = a_t|S = s_t] \quad , \quad s_t \in S \; and \; a_t \in A \tag{2}$$

where $P[A = a_t|S = s_t]$ is the probability of taking particular action conditioned on being in some state and lies between 0 and 1. S and A are the set of states and possible actions respectively.

5.1.3 Rewards/Goal

In a Reinforcement learning methods, an agent will learn by reinforcement or hit and trail. A negative reward or a penalty will lead to and unwanted reactions. Conversely, a series of positive rewards leads towards a good policy. According to [Andrew, 1999], the aim of any agent must be to maximise the accumulate sum of rewards:

$$R_t = \Re_t + \Re_{t+1} + \Re_{t+2} + \Re_{t+3} + ... + \Re_T \tag{3}$$

where R_t denotes the return, t a particular time step upto T. For stochastic environment, the goal is to maximise the expected value of the return within the task. The above equation can be rewritten as follow:

$$R_t = \sum_{k=0}^{T} \Re_{t+k+1} \tag{4}$$

However the future reward may have a different present value. This rate is referred in as discount factor. A discount factor can be defined as a learning value, varying from $0 \leq \gamma \leq 1$. Each future reward at time stamp t will be discounted by γ^{k-1}. If γ approaches zero, the reward in the near future to the present state is considered more valuable by the agent. Conversely, if γ is close to one, then the agent treats future rewards with equal importance and behaves greedily. The return for a specific policiy can be formulated as:

$$R_t = \sum_{k=0}^{T} \gamma^k \Re_{t+k+1} \tag{5}$$

5.1.4 Reward Function

Reward function defines the goal of any reinforcement learning problem. It maps each input state(or state-action pair) in an environment to a single numerical value, called a reward, depicting the innate preference of that state. The ultimate goal of the reinforcement learning agent is to maximise the rewards it receives in the long run. The reward function defines how good or bad certain actions can be for our agent. The utmost important characteristic of the reward function is that it cannot be changed by the agent. It may, however, serve the basis of altering the policy of our system. For example, an action chosen by policy might yield a low reward which will lead to change in policy so that some other action is chosen in that situation in the future.

5.1.5 Discounted Rewards

In order to train an effective agent, the goal should be to maximise the reward in the long run instead of just caring about the immediate returns. The *discounted expected return* is defined as the cumulative sum of future returns and can be

found in (6). The discount factor $\gamma \in [0, 1]$ tells how to weight the distant rewards.

$$R_t = R_{t+1} + \gamma Rt + 2 + \gamma^2 R_{t+2} + ... = \sum_{k=0}^{\inf} \gamma^k R_{t+k+1} \tag{6}$$

It can be divided into two categories, which are as follow:

- Episodic: We can divide the training process into episodes. Episode is considered to be over when the agent reaches a terminal state, which resets the scene and the agent has to start again in the next episode. The immediate reward for terminal state is 0 and cam be reached in T finite time steps.
- Continuous: We cannot form episodes as it is a continuous ongoing process such that T = inf.

5.1.6 Value Function

While the reward function defines what's good in an immediate sense, the value function defines what would be good for the agent in the far sight. Roughly speaking, the value of the state is the total amount of reward an agent can expect to collect over the steps taken in future, starting from the present state. Whereas rewards determine the immediate, inherent preference for the states, values determine the long term favourablity of state after considering the states that are likely to follow, and the rewards that will be received from those states. For example, a state yielding a low immediate reward might have high value because of it being regularly followed by states of high rewards and vice-versa.

In a system following a policy π, the value V_π is give by the expected rewards from that state. Mathematically, it is expressed as:

$$V_\pi(state) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = state] \tag{7}$$

5.1.7 Model

Model of the environment imitates the behaviour of the environment itself. For instance, given a state and action, the model is capable of predicting the resultant state and reward. Generally, models are used for planning, i.e., the ways of deciding on a way of action by taking into account the possible prospect situation before they actually happen.Earlier reinforcement learning system were

explicitly trial-and-error learners; where they did something that was almost opposite of planning.

5.1.8 The Bellman Equation

Mathematician Richard Bellman derived the very famous Bellman Equation which allow us to start solving Markovian Decision Process(MDP). The Bellman equation is omnipresent in Reinforcement Learning and allow us to understand various Reinforcement Learning algorithms working. The importance of Bellman Equation is that it let us express the value of states as value of other state. It provides a recursive relation between current states and successive states, i.e., if we know the value of s_{t+1}, we can calculate the value of s_t.

Before we quote the Bellman Equations, we must understand how P and R are defined. We define P and R as follows:

$$P^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a) \tag{8}$$

P is the transitional probability, i.e., if we start at state s and take action a we end up in state s' with probability $P^a_{ss'}$.

$$R^a_{ss'} = E[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a] \tag{9}$$

$R^a_{ss'}$, is another way of writing the expected(or mean) reward that can be received when starting from state s, taking action a, and moving into state s'.

Finally with these in hand we can formulate the Bellman Equation. The Bellman Equation for state value function is denoted as follow:

$$v_\pi = \sum_a \pi(a|s) \sum_{s',r} P^a_{ss'}[R^a_{ss'} + \gamma v_\pi(s'] \tag{10}$$

For action value function it is denoted as follows:

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')] \tag{11}$$

5.2 Markov Decision Process

In order to compare real world tasks and theoretical problems, a common framework is required and hence, we need to mathematically formalize this decision making process. According to *Markov Assumption or Property*, an indepen-

dence of past and future states, means that the state and the behaviour of the environment of the environment at time step t are not affected by the past agent-environment interactions $a_1, ..., a_{t-1}$.

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...) = P(s_{t+1}|s_t, a_t) \tag{12}$$

In short, the future must be independent of the past given the present. If the reinforcement learning task can fulfill the *Markov Assumption*, it can be formulated as five-tuple $MarkovDecisionProcess(S, A, P^a_{ss'}, R^a_{ss'}, \gamma)$.

- Set of states S.
- Set of Actions A. *A(s)* is the set of available actions in the state s.
- Transitional probabilities $P^a_{ss'} : (S \times A \times S) \rightarrow [0, 1]$. It is the probability of the transition from s to s' when taking action a in state s at time step t.
- Reward Probabilities $P^a_{ss'} : (S \times A \times S) \rightarrow IR$. It defines the immediate reward the agent will receives after the transition from s to s'.
- Discount factor $\gamma \in [0, 1]$ for computing the *discounted expected* return.

Markov Decision Process is finite if the states set S and actions set A are finite. MDP formally describes a given environment for reinforcement learning given that the environment is fully observable, i.e., the current state completely characterise the process.

Markov Decision Process is finite if the states set S and actions set A are finite. MDP formally describes a given environment for reinforcement learning given that the environment is fully observable, i.e., the current state completely characterise the process.

5.2.1 Temporal Learning (λ)

This method is very analogous to natural learning agent, in which the farther in the past a decision was made, the less its value V_{t-k} is reinforced by the current reward signal r_t as shown in the figure below.

Temporal learning algorithms(Q-learning, SARSA) can learn directly from the fresh experiences without the requirement of specific model of the environment. The cost of being able to regularly vary the value function is paid by

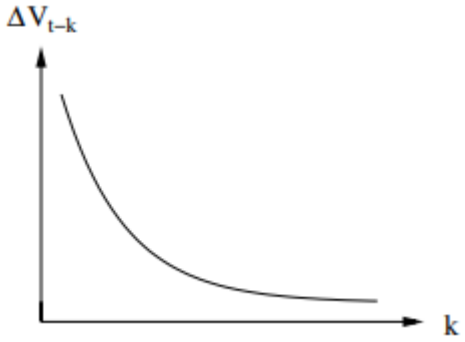
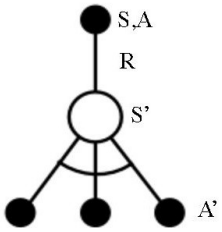


Figure 12: Effect of a delayed reward signal

the necessity of updating the estimated values based on other learnt estimations. There is no need to disregard any episodes as it can learns from every transition(state, reward, action, next state, next reward).

5.2.2 Q-learning

Q-Learning is a groundbreaking solution to Markov Decision Process. It is a Temporal Difference (TD) learning algorithm. It is an off-policy algorithm which uses the greedy policy to update the Q-Table. Compared to Mote-Carlo methods where Q-Table is updated at end of every episode, here Q-Table is continuously updated at every timestep.



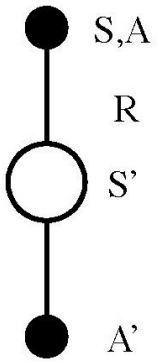
$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

Figure 13: Backup diagram for Q-Learning

The figure 13 represents the backup diagram for Q-Learning and its update rule beneath it. Source: Sutton and Barto.

5.2.3 SARSA

SARSA – an acronym for State-Action-Reward-State-Action is an on-policy learning algorithm. It learns by taking actions in accordance with the current policy rather than the greedy policy. SARSA updates and store the Q-Values for every state-action pair in a Q-Table as follows:



$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma Q(S', A') - Q(S, A)\right)$$

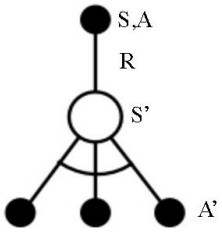
Figure 14: Backup diagram for SARSA

The figure 14 represents the backup diagram for SARSA and its update rule beneath it. Source: Sutton and barto.

5.2.4 Expected SARSA

Expected SARSA is similar to SARSA with the difference that it takes the expected value of Q-Values for every possible action in a particular state to update the Q-Table. The update equation of Expected SARSA is as follows:

The figure 15 represents the backup diagram for Expected SARSA and its update rule beneaMarkov Decision Process is finite if the states set S and actions set A are finite. MDP formally describes a given environment for reinforcement learning given that the environment is fully observable, i.e., the current state completely characterise the process.



$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

Figure 15: Backup diagram for Expected SARSA

5.3 Classes of Algorithms

Given all the inputs, we need to decide how the agent will learn the optimal policy. There are three main categories of reinforcement learning algorithms which are briefly discussed below.

5.3.1 Model Based Learning

A model based reinforcement learning algorithms takes the state, action, reward sequence as an input and then learn the transition probability $P^a_{ss'}$.Once learnt, the transitions can be used to solve the Markov Decision Process which give the optimal value function Q^*, which in turn gives the optimal policy π^*.

5.3.2 Model Free Learning

In some problems, it is not required at all to learn the transitional probability $P^a_{ss'}$. In such cases, the agent instead learns the state values or the state-action values which corresponds to the optimal policy of the problem, without having to learn the state transitions.

5.3.3 Policy Search

This is the last class of the reinforcement learning algorithms which takes in a policy and feeds it to a policy update equation. The update equation now modifies the original policy based on the state, action, reward during the learning process.

6 Our approach

For this section, we propose a reinforcement learning based optimizing solution for the cache replacement policy - RLCaR (Reinforcement Learning Cache Replacement). In particular, we want to train an agent to learn an optimal policy (π^*) to efficiently replace frames in the memory cache in order to maximize a certain optimization goal metric. In our approach, we aim to maximize the cache hit ratio.

6.1 Reinforcement Learning formation

6.1.1 State space

Intuitively, a state in our environment represents the operating system's cache allocation structure and related statistics. In order to inform the agent we use two heuristics - Least recently used (LRU) based and Least frequently used (LFU) based statistics. More formally, a state in our system is an n-dimensional tuple as follows:

$$S = (\vartheta_1, \vartheta_2...., \vartheta_n) \tag{13}$$

here. n is the size of cache. It denotes the maximum number of pages that can allocated at once in the memory cache. Each of $\vartheta_i, \forall i \in [1, n]$ is again a 2-dimensional tuple as follows:

$$\vartheta_i = (\psi_i, \chi_i) \tag{14}$$

where, ψ_i is a counter for the number of times page i was accessed. And χ_i is a counter for the number of timesteps since the page i has not been accessed. The purpose was adding these counters was that ψ_i will help the agent to make decisions using the LFU heuristic while χ_i will help it in making cache replacement decisions using the LRU heuristic.

6.1.2 Action space

The task of the agent is to learn an optimal policy to maximize the cumulative reward function. For this, the agent has to make decisions regarding the eviction or cache replacement policy. Hence, if the number of unique pages is k, the agent has to choose anyone of these k pages to evict as an action. Therefore, the action

space is a set as follows:

$$A = \{a_i | \forall i \in [0, k-1]\} \tag{15}$$

On choosing action a_i, the i^{th} page is evicted from the cache to make space for a new page.

6.1.3 Reward

The reward function has a paramount significance for guiding the agent to learn an optimal policy π^*. We want to maximize the hit ratio of out cache replacement system, thus we reward the agent with a positive reward on taking an action that causes a cache hit, while a negative reward is given for a cache miss. It is possible that the agent takes an action to replace a page that is not even allocated in the cache. In this case, the agent is given a heavily negative reward. Mathematically, the reward function is formalized as:

$$R(s_i, a_i) = \begin{cases} \begin{cases} +1, & Cache\ hit \\ -1, & Cache\ miss \end{cases} & i \in C(s_i) \\ -10, & i \notin C(s_i) \end{cases}. \tag{16}$$

Where, s_i and a_i are the current state and action. $C(s_i)$ represents the cache set of state i or simple the list of all the pages currently allocated in the cache in the state s_i. λ represents the magnitude of reward. A positive reward of $+\lambda$ is given for cache hit while a negative reward of $-\lambda$ is given for a cache miss. A heavily negative reward $-\lambda$ is given in case the agent tries for evict a page from the cache that doesn't exist in the cache. This is to heavily penalize the agent on making such grave mistakes. In our experiments, we have used the following values of these constants:

$$\lambda = 1 \tag{17}$$

and,

$$\Lambda = 10 \tag{18}$$

The agent should learn an optimal policy (π^*) by getting positive reward for taking actions that cause cache hits and getting negative rewards for actions causing cache miss.

6.1.4 Environment

In reinforcement learning, the agent learns through active interactions with an environment. For our problem statement, the ideal environment would the actual operating system's cache handling routine. However, training an agent in such a practical setting is impudent. We, therefore, create a simple probability based simulator for cache management. Our simulator consists of two modules - the environment and the OS. We have created the environment by extending the OpenAI gym environment class. Using OpenAI gym's base class for our own custom environment makes our approach open sourced and more robust. For the Operating system module, we have created a simple simulation of the operating system system call that randomly requests access to certain page as follows:

$$\alpha_i \sim \wp \tag{19}$$

Where, α_i is the requested page id sampled from a custom uniform distribution $\wp$ which is represented as follows:

$$\wp = (p_1, p_2, ..., p_k) \tag{20}$$

where k is the number pages and p_i is the probability of the OS requesting for page i. Since p_i represents probability:

$$\sum_{i=1}^{k} p_k = 1 \tag{21}$$

and,

$$p_i \in [0, 1] \forall i \in [0, k] \tag{22}$$

It is reasonable to sample the page requests from a probability distribution because operating systems make calls to certain pages more often than others. Following this principle, it is plausible to assume that the page requests made by operating system follows a certain distribution.

7 Result

This section outlines the results of our work for Reinforcement Learning Cache Replacement (RLCaR). We first present the reward optimization using three common model-free TD 0 RL algorithms - Q-Learning, SARSA and Expected

SARSA. Following this, we do a comparative analysis between RLCaR, LRU and LFU based on number of cache hits per episode.

7.1 Reward Maximization

We train an agent to maximize our reward function as illustrated in the section 6.1.3. We train the agent using three TD 0 algorithms:

1. Q-Learning - Depicted by blue color in Figure 16
2. SARSA - Depicted by blue orange in Figure 16
3. Expected SARSA - Depicted by green color in Figure 16

Figure 16 is a plot of episodic reward gains. We ran the agent training for total 1,000 episodes and plot each episode's cumulative discounted reward (as used in the Bellman Equation) on the Y-Axis. The expected cumulative reward that is to be maximized is:

$$R = E[\sum_{i=1}^{T} \gamma * r_i] \tag{23}$$

here, γ is the discount factor, r_i is the reward gain at i^{th} timestep. T is the episode length in number of timesteps. It should be noted that an episode terminates after running for T timesteps in our system. For our experiments, we have initialized these hyperparameters as follows:

$$T = 4, T \in [0, \infty) \tag{24}$$

and for the γ (discount factor),

$$\gamma = 0.9, \gamma \in [0, 1] \tag{25}$$

We justify the choice of discount factor as $\gamma = 0.9$ by stating that the task of the agent is to replace cache in a way to maximize the hit ratio. The future rewards (future cache hits) are equally important as the current reward (current cache hit). Therefore, to promote exploration and give global optimization based importance to cache hits at every timestep, we set the value of γ close to 1. However, we are yet to do an experimental selection of the value of discount factor by varying it from 0 to 1.

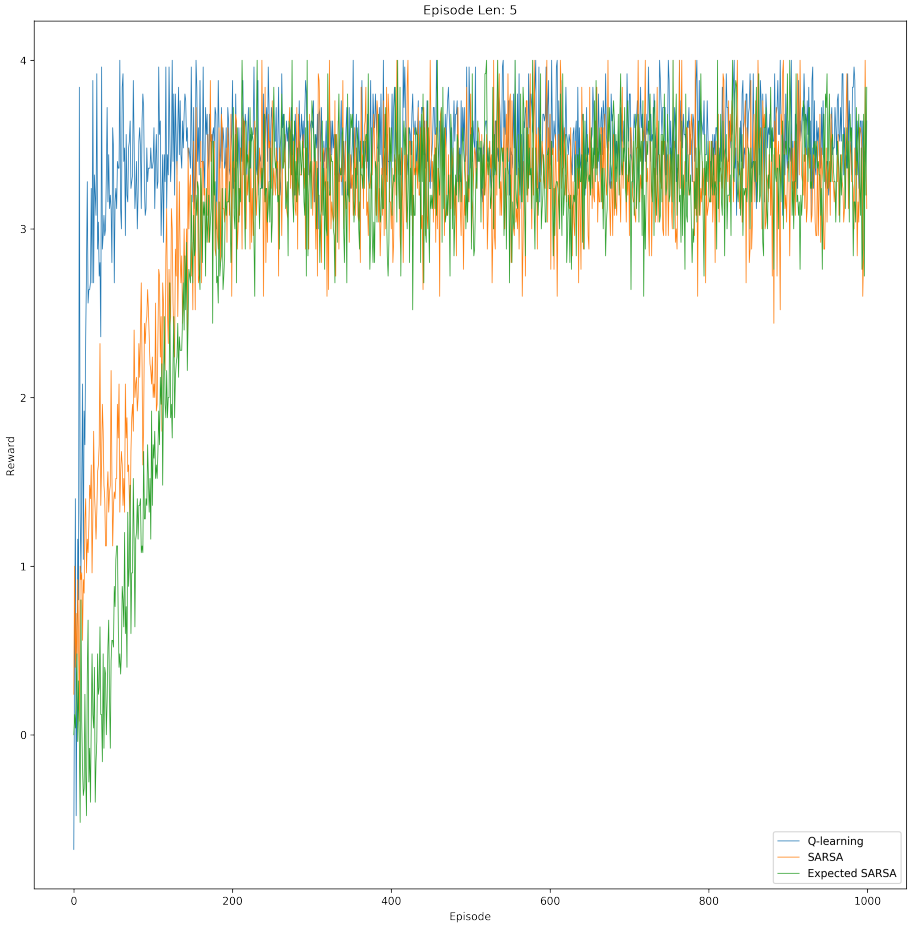


Figure 16: Reward Curve

7.2 Cache hits

This section outlines the results for the percentage of cache hits for each of the three - LRU, RLCaR and LFU cache replacement policies. To account for stochasticity of operating system page request, we perform each of our experiment 100 times and plot the mean, high and low of cache hit percentage in an error-bound line curve. The quantity on the y-axis of the following curves is the cache hit ratio which is defined as follows:

$$\theta = 100 * \frac{|Cache\ hits|}{L} \tag{26}$$

where $|Cache\ hits|$ is the number of cache hits and L is the episode length. Higher value of θ entails better cache replacement strategy.

7.2.1 LFU

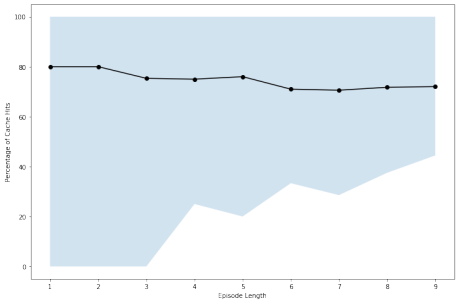


Figure 17: LFU Cache hits

As observed in Figure 17, the percentage of cache hits goes down with an increase in episode length. This explained by the incremental stochasticity of environment. For a longer length of episode, the operating systems makes more number of requests to pages in memory. Since each of this request is stochastic, it adds up to increasing the overall stochasticity of the system, thereby reducing the performance.

7.2.2 LRU

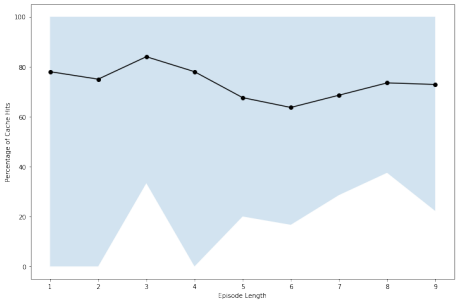


Figure 18: LRU Cache hits

Figure 18 shows the cache hit percentage with varying episode length of the system. It follows a trend similar to the LFU cache replacement policy.

7.2.3 RLCaR

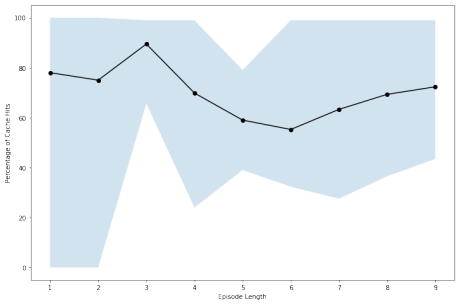


Figure 19: RLCaR Cache hits

Figure 19 shows the cache hit percentage with varying episode length of the system for our reinforcement learning based cache replacement policy. A general trend of decrease in percentage of cache hit can be observed with an increasing episode length. Again, stochasticity of environment can be used to explain this trend. However, drawing inferences from TD 0 RL based algorithms is notoriously difficult and requires a careful mathematical analysis which we leave for future work.

7.3 Comparative Analysis

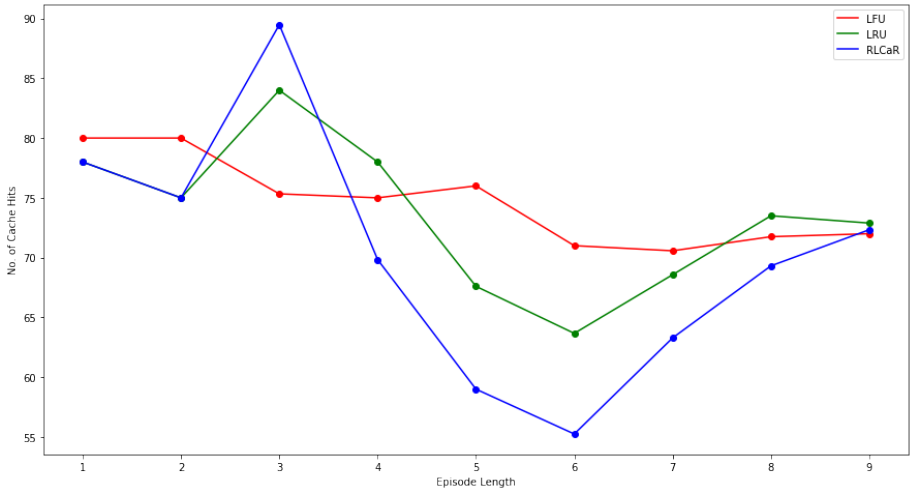


Figure 20: Comparitive analysis of LFU, LRU and RLCaR

Figure 20 shows a comparison between the three - LRU, LFU and RLCaR cache replacement policies. It can be observed that our algorithm performs

only a little worse than LRU and LFU page replacement policies. Also, for certain time lengths, RLCaR seem perform better than both LRU and LFU cache replacement algorithms. The state space of such a cache replacement environment is huge and continuous. We expect to get much better results on approximating the policies using an Artificial Neural Network (ANN). We leave that for the future work, for now.

8 Code

8.1 OS page request module

```
import gym
import numpy as np
import pandas as pd
import pickle
from collections import defaultdict

P = [0.3, 0.3, 0.2, 0.15, 0.05]

class OS():
    """
    Simulate a simple OS cache handler
    """
    def __init__(self, limit, n_pages):
        super(OS, self).__init__()
        self.limit = limit
        self.n_pages = n_pages
        if len(P) != self.n_pages:
            raise Exception("Size mismatch for P and n_pages")

    def init_pages(self):
        pages = {}
        NT = defaultdict(int)
        for i in range(self.limit):
            #page_id = self.get_id()
            page_id = i #For now let it be sequential
            lu = 0 #No. of timesteps ago this page was accessed ~LRU
```

```
            nt = 1 #No. of times this page was accessed ~LFU
            page = [lu, nt]
            NT[page_id] += 1 # Update NT dict
            pages[page_id] = page
        return pages, NT

    def get_id(self):
        return int(np.random.choice(np.arange(self.n_pages), p=P))
```

8.2 CacheEnv

```
import gym
import numpy as np
import pandas as pd
import pickle
from collections import defaultdict
from os_sim import OS

LIMIT = 3
N_PAGES = 5
EPS_LEN = 3
POS_REW = 1
NEG_REW = -1
HEAVY_NEG_R = -10

class CacheEnv(gym.Env):
    metadata = {'render.modes': ['human']}
    def __init__(self, limit=LIMIT, n_pages=N_PAGES, eps_len=EPS_LEN):
        super(CacheEnv, self).__init__()
        self.limit = limit
        self.n_pages = n_pages
        self.eps_len = eps_len
        self.os = OS(limit, n_pages)
        self.pages, self.NT = self.os.init_pages()
        self.total_hits = 0
        self.timestep = 0 #counter; if this reaches eps_len, return done=True
        self.done = False
```

```
27          self.new_page_id = -1
28
29      def step(self, action, test=False):
30          """
31          First OS will send a page id (randomly from distribution P)
32          based on the action choose to evict a page
33          allocate this page id cache inplace of the 'action' id
34          """
35          self.timestep += 1
36          if self.timestep >= self.eps_len:
37              self.done = True #Episode reached its end
38          new_page_id = self.os.get_id() #This is page requested by the OS
39          self.new_page_id = new_page_id #Store for debugging
40          reward, hit = self.allocate_cache(action, new_page_id)
41          if hit:
42              observation = f"This was a hit, OS asked for: {new_page_id}"
43              self.total_hits += 1
44          else:
45              observation = f"This was not a hit, OS asked for: {new_page_id}"
46          return self.pages, reward, self.done, observation
47
48      def reset(self):
49          self.pages, self.NT = self.os.init_pages()
50          self.total_hits = 0 #Intuitive
51          self.done = False
52          return self.pages
53
54      def render(self, mode='human'):
55          pass
56
57      def close(self):
58          pass
59
60      def if_allocated(self, id):
61          """
62          returns true if 'id' is allocated a cache currently
63          """
```

```
64          if id in self.pages.keys():
65              return True
66          return False
67
68      def allocate_cache(self, action, id):
69          """
70          remove page 'action'
71          add page 'id'
72          """
73          action = int(action)
74          id = int(id)
75          hit = False #Page hit or not?
76          self.NT[id] += 1
77          # For all the pages except id, increament their lu counter
78          for page_id in self.pages.keys():
79              if page_id == id:
80                  continue
81              else:
82                  self.pages[page_id][0] += 1
83
84          if action not in self.pages.keys():
85              #Agent asked to remove a page that wasn't even allocated
86              return HEAVY_NEG_R, hit
87
88          if self.if_allocated(id):
89              hit = True #HIT!
90              page = self.pages[id]
91              page[0] = 0
92              page[1] += 1
93              self.pages[id] = page
94              reward = POS_REW #pos reward for hit
95          else:
96              self.pages.pop(action) #Remove page 'action'
97              self.pages[id] = [0, self.NT[id]] #Add page 'id'
98              reward = NEG_REW #neg reward for no hit
99
100         return reward, hit
```

```
101
102 if __name__ == "__main__":
103     env = CacheEnv()
104     env.reset()
```

8.3 Qlearning

```
1  import gym
2  import numpy as np
3  import time
4
5  def init_q(s, a, type="ones"):
6      """
7      initialize the Q-Table
8      """
9      if type == "ones":
10         return np.ones((s, a))
11     elif type == "random":
12         return np.random.random((s, a))
13     elif type == "zeros":
14         return np.zeros((s, a))
15
16
17 def epsilon_greedy(Q, epsilon, n_actions, s, train=False):
18     """
19     Epsilon greedy policy for exloration-exploitation tradeoff
20     """
21     if train or np.random.rand() >= epsilon:
22         action = np.argmax(Q[s, :])
23     else:
24         action = np.random.randint(0, n_actions)
25     return action
26
27 def qlearning(alpha, gamma, epsilon, episodes, max_steps, n_tests, render = False, test=Fals
28     """
29     A simple Q-Learning algorithm
30     More info: https://github.com/TimeTraveller-San/RL_from_scratch
```

```
    """
    env = CacheEnv() # Init using default args
    n_states, n_actions = env.observation_space.n, env.action_space.n
    Q = init_q(n_states, n_actions, type="ones")
    timestep_reward = []
    for episode in range(episodes):
        print(f"Episode: {episode}")
        s = env.reset()
        a = epsilon_greedy(Q, epsilon, n_actions, s)
        t = 0
        total_reward = 0
        done = False
        while t < max_steps:
            if render:
                env.render()
            t += 1
            s_, reward, done, info = env.step(a)
            total_reward += reward
            a_ = np.argmax(Q[s_, :])
            if done:
                Q[s, a] += alpha * ( reward  - Q[s, a] )
            else:
                Q[s, a] += alpha * ( reward + (gamma * Q[s_, a_]) - Q[s, a] )
            s, a = s_, a_
            if done:
                if render:
                    print(f"This episode took {t} timesteps and reward: {total_reward}")
                timestep_reward.append(total_reward)
                break
    if render:
        print(f"Here are the Q values:\n{Q}\nTesting now:")
    if test:
        test_agent(Q, env, n_tests, n_actions)
    return timestep_reward
```

9 Future work

Eventually we want to use the experience that we learn from this system to run on a deep reinforcement learning function approximator that will provide us a decision of which page we should evict. The experience that we get over a period of time, makes this unsupervised learning problem into a supervised learning problem. Current Linux cache behaves mostly like LRU. However, it is beneficial sometimes to behave like LFU. The current algorithm for eviction is somewhere in between but certainly not the best. Hopefully function approximators could give better results. There has not been much research in applying AI techniques to caching. Hopefully because this is very much an empirical problem rather than a theoretical one, function approximators might give us a better solution than currently exists. One problem in using neural network based reinforcement learning for kernel level caching decisions that is very outright is that, all frameworks run in userspace whereas caching decisions in the kernel are taken in the kernel space. Alternatively, we could apply this approach to user space caches and apply simpler kernel level functions for policy gradient based reinforcement learning to the problem of caching. We could also use techniques to evaluate what policies work better and switch between these policies accordingly on the basis of their access patterns.

10 Conclusion

Our work is an novel attempt to learn the variations in data traffic during file read and write operations and model these variations for effective page caching. Our method encapsulates the caching problem as a MDP (Markov Decision Process) and makes use of RL (Reinforcemnt Learning) to optimally choose its actions depending on the fluctuations encountered in the environment in order to maximize the cache hit ratio.

References

[Alabed, 2019] Alabed, S. (2019). Rlcache: Automated cache management using reinforcement learning.

[Andrew, 1999] Andrew, A. M. (1999). Reinforcement learning: An introduction by richard s. sutton and andrew g. barto, adaptive computation and

machine learning series, mit press (bradford book), cambridge, mass., 1998, xviii+ 322 pp, isbn 0-262-19398-1,(hardback,£ 31.95). *Robotica*, 17(2):229–235.

[Bansal and Modha, 2004] Bansal, S. and Modha, D. S. (2004). Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA. USENIX Association.

[Johnson and Shasha, 1994] Johnson, T. and Shasha, D. (1994). 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB*.

[Megiddo and Modha, 2003] Megiddo, N. and Modha, D. S. (2003). Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA. USENIX Association.

[Surbhi, 2018] Surbhi, P. (2018). Better caching using reinforcement learning. `https://github.com/csurbhi/UCB1-algorithm-for-better-caching`.

[Vietri et al., 2018] Vietri, G., Rodriguez, L. V., Martinez, W. A., Lyons, S., Liu, J., Rangaswami, R., Zhao, M., and Narasimhan, G. (2018). Driving cache replacement with ml-based lecar. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, pages 3–3, Berkeley, CA, USA. USENIX Association.