Learning to Update: Using Reinforcement Learning to Discover Policies for List Update

by

Isabelle A. Quaye

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

Authored by: Isabelle A. Quaye
Department of Electrical Engineering and Computer Science
January 26, 2024

Certified by: Ronitt Rubinfeld
Professor
Thesis Supervisor

Certified by: Piotr Indyk
Professor
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Learning to Update: Using Reinforcement Learning to Discover Policies for List Update

by

Isabelle A. Quaye

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2024, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The use of machine learning models in algorithms design is a rapidly growing field, often termed learning-augmented algorithms. A notable advancement in this field is the use of reinforcement learning for algorithm discovery. Developing algorithms in this manner offers certain advantages, novelty and adaptability being chief among them. In this thesis, we put reinforcement learning to the task of discovering an algorithm for the list update problem. The list update problem is a classic problem with applications in caching and databases. In the process of uncovering a new list update algorithm, we also prove a competitive ratio for the transposition heuristic, which is a well-known algorithm for the list update problem. Finally, we discuss key ideas and insights from the reinforcement learning agent that hints towards optimal behavior for the list update problem.

Thesis Supervisor: Ronitt Rubinfeld
Title: Professor

Thesis Supervisor: Piotr Indyk
Title: Professor

Acknowledgments

I would like to thank my thesis advisors Professor Ronitt Rubinfeld and Professor Piotr Indyk. The impact of their counsel, support and guidance during my MEng project is immeasurable. This thesis will not be what it is without them.
I would also like to thank Sandeep Silwal for his mentorship and assistance during my MEng project. I have learnt and grown so much by working with Sandeep. The work presented in Chapter 6 is joint work with Sandeep.
I also want to thank my mum, Harriet Owusua, and my sister, Jessica Quaye, who encouraged me and kept me accountable throughout the writing process.
Most importantly, I thank God for seeing me through the ups and downs of my MEng which has culminated in this body of work.

Contents

Chapter 1

Introduction

Learning-augmented algorithms,also known as learning with predictions, constitute an algorithmic design paradigm where the algorithms makes use of predictions or advice from a machine learning model. Typically, the machine learning model is trained within the same context of the algorithm's intended application. The hope is that by delegating decision-making to a trained model, the algorithm will achieve better results depending on the context. Many learning-augmented versions of algorithms have been presented, such as learning-augmented caching algorithms[39, 25], scheduling algorithms[39, 11] and paging algorithms. However, not all algorithms lend themselves to this design paradigm. This is primarily because it is not always obvious how to decouple and delegate the decision-making process of an algorithm to a machine learning model or oracle[1]. Good candidates for learning with predictions are online and streaming algorithms[34, 41, 20, 8, 49, 33, 1, 9, 6, 15, 22], similarity search [48, 17, 18, 40, 46] and combinatorial optimization [29, 12, 33, 39, 16, 14].

[1]The two words will be used interchangeably in this thesis.

While learning-augmented algorithms seek to achieve improved results by **modifying** or **augmenting** an algorithm with advice from an oracle, the question remains whether we can delegate even more responsibility to the oracle. In that case, we go from **learning-augmented** algorithms to **learned** algorithms. Now, the machine learning model is tasked with developing an algorithm that behaves optimally. Of all the machine learning algorithms, reinforcement learning is best suited to the task and here, too, there have been many successes[19, 29, 35]. However, much like learning-augmented algorithms, not all problems can be fit within this mold. As we will see in Chapter 2, reinforcement learning has a specific formulation and only if a problem fits this formulation can we apply reinforcement learning. Even then, there are challenges that one must successfully overcome like large state spaces and non-convergence. Generally speaking, combinatorial optimization problems are excellent candidates for reinforcement learning. As a result, most data structure problems naturally lend themselves to learned algorithms design.

The list update problem (list access problem or self-organizing sequential search) involves maintaining a static list from which we look up records sequentially by walking down the list until we find the record we are looking for. During the servicing of a query, the list may be rearranged to lower the cost of future queries. Clearly, if queries are uniformly random i.e. each record is looked up with equal probability, then rearranging the list will not yield much benefit. However, if it is the case that the keys are queried with non-uniform probability, then we can reduce the cost of future queries by rearranging the list. The list update problem has been well studied over the years[13, 23, 24, 30, 37, 45, 47, 43] with many al-

gorithms, both deterministic[47, 4] and randomized[42, 5, 26, 27], being put forth. This thesis project has three major goals:

1. Discover a learned algorithm for the list update problem using reinforcement learning.

2. Compare the behavior and performance of the learned algorithm to existing list update algorithms.

3. Analyze and understand the behavior and performance of the learned algorithm relative to existing algorithms to gain insights for designing novel and competitive list update algorithms.

Throughout the chapters of this thesis, we address each of these goals as we present the major contribution of this thesis: a learned algorithm for the list update problem.

Chapter 2

Useful Background and Definitions

2.1 Reinforcement Learning

Reinforcement learning is a class of machine learning techniques or algorithms for finding an optimal policy for a Markov Decision Process(MDP). In Figure 2-1, we see the typical setup of an MDP. In reinforcement learning, the bulk of the learning by the agent is driven by the reward signal received because the goal of the agent is to maximize the total reward over all time steps. Thus, R must be designed to capture which states are more desirable than others. Later in Chapter 7, we shall see that by varying the reward function, the agent learns different policies.

Reinforcement learning algorithms can be classified as either model-based or model-free. Model-based algorithms require complete information about the environment in order to simulate and search for the best action to take. Monte Carlo Tree Search is a popular example of model-based learning. Model-free algorithms rely on the information collected by the agent's interactions to decide the best

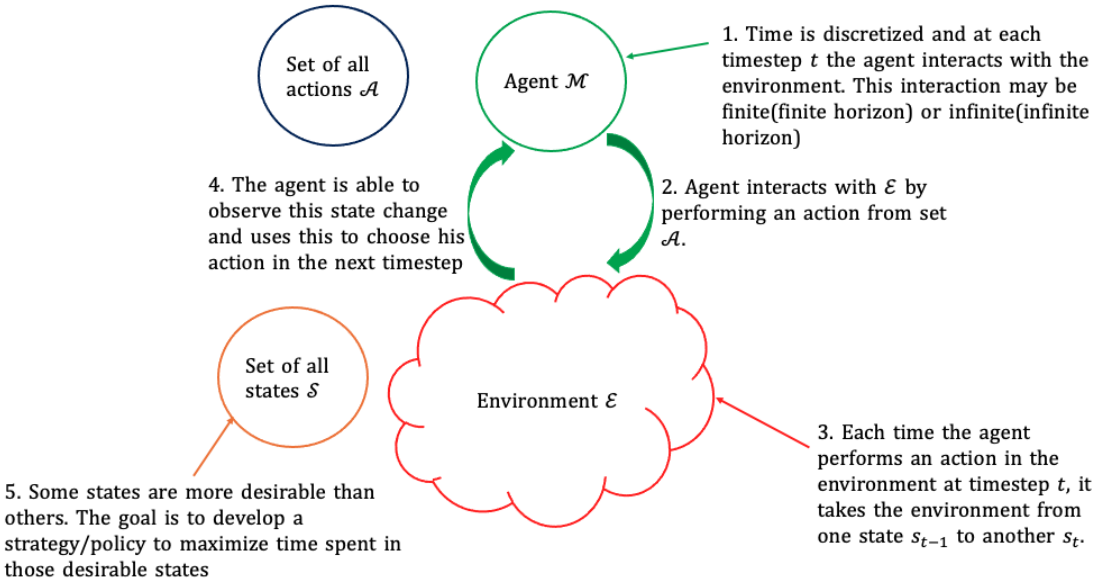

Figure 2-1: Markov Decision Process(MDP) setup.

action to take in a given state. Examples of model-free learning algorithms are policy iteration and value iteration [36].

Reinforcement learning is a powerful unsupervised learning technique, especially when the goal is to uncover optimal action(s) to take in different situations. Some applications of reinforcement learning include recommendation systems [3], autonomous vehicles [10] and more recently algorithm discovery [19, 35].

2.2 Learned Algorithms and Data Structures

To understand learned algorithms, it is useful to begin with beyond worse case analysis of algorithms and learning augmented algorithms. While worst case analysis provides strong theoretical guarantees, in practice, the worst case sel-

dom arises. This opens up the possibility of achieving better than worst case performance in practice by designing algorithms and data structures better suited to practical settings. [44] details attempts at doing just this. This is the inspiration behind data-driven algorithm design [44] and learning-augmented algorithms [44].

Learning augmented algorithms, otherwise known as learning with predictions, describes a design paradigm where an algorithm is furnished with predictions or advice from an oracle. The advice and predictions received from the oracle usually takes the place of heuristics or random bits in traditional algorithms. Learning augmented algorithms have seen huge success with optimization problems like the ski rental problem [39] and nearest neighbors [48, 17, 18, 40, 46], streaming algorithms like counting sketches [2], etc.

With learned algorithms, the machine learning model is given even more responsibility: rather than assuming the role of an advising oracle, the model determines most or all the steps of the algorithm. For learned data structures, the model is tasked with determining algorithms for all or most of the operations of the data structure. In DeepMind's 2022 Nature article [19], the authors present a learned algorithm for matrix multiplication. In fact, they present multiple novel algorithms for matrix multiplication that are empirically faster than existing matrix multiplication algorithms like Strassen's. They use reinforcement learning to first factor the matrices and then multiply the smaller matrices derived from the factorization. Learned data structures are also extremely popular with learned algorithms for indexing databases [31], learned bloom filters [38], learned algorithms for an R-trees operations [21] and many more.

Whether it be learning-augmented algorithms or learned algorithms, they have the advantage of being tailored to a specific use case and so have good empirical performance when compared with traditional algorithms. In certain cases, they also have the advantage of being able to adapt to different use cases, and so on average the good empirical performance seen in one application domain can be replicated in another with some necessary adjustments. This is especially important for online algorithms and data structures because the stream of requests, often called **workload** is not known a priori and can change over time. Having an algorithm that can adapt to a workload dynamically is therefore advantageous.

2.3 List Update Problem

In the list update problem, we are given a list of records L to which a series of queries, Q, for records will be made. We assume that all queries to L are for records found in L. The cost of each query is equal to the position or index of the record in L at the time the query is made. In between queries, records can be rearranged at a cost to improve the search time for future queries. During a query to a record in position i, moving the record to any position $\leq i$ is free(free exchange [28]). All other rearrangements incur a cost(paid exchange [28]).
The goal then is to develop a policy for maintaining the list so that the average cost of servicing queries is low, keeping in mind the cost of rearranging records in the list. Ideally, if the query frequency of each record were known, then we can achieve optimal query cost by ordering records in L by the query frequency. However, query frequencies are not known a priori and while we can approximate this by maintaining frequency counters, space constraints and the potential for counter overflows make it an unattractive solution. Instead, so-called *memory-*

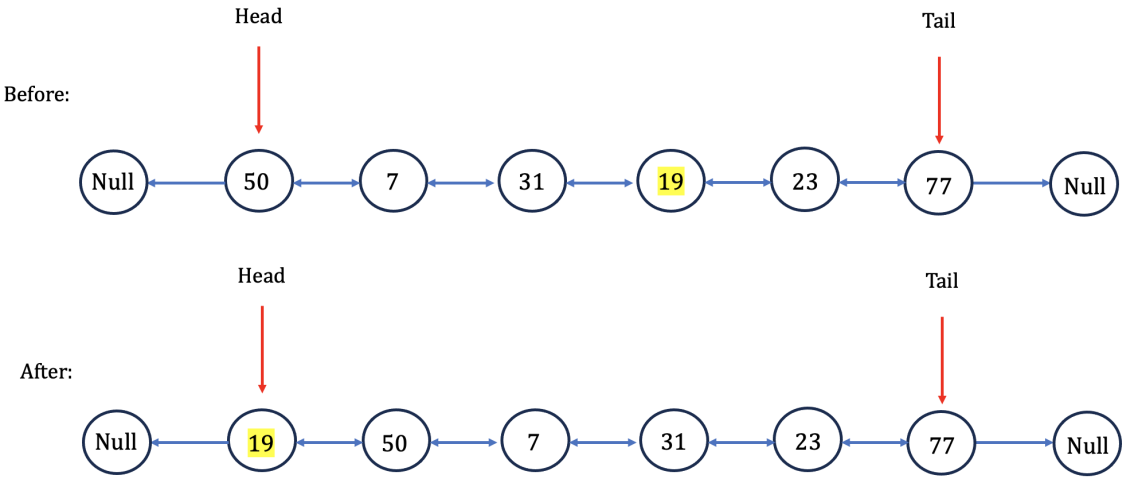

Figure 2-2: Move-to-front policy action when item 19 is accessed in L

less list update heuristics and policies that approximate this ordering are favored. Some of these heuristics are deterministic and others are randomized.

Two well-studied memory-less deterministic list update heuristics are **move-to-front** and **transposition**. With move-to-front, each time a record is queried, it is moved to the front of the list. For transposition, records are moved one position forward once accessed. We show an example for each algorithm in Figures 2-2 and Figure 2-3.

Usually, randomized list update heuristics tend to be some variation of the move-to-front heuristic with randomness added. Yet, they achieve better competitive ratios than move-to-front. For example, in the BIT algorithm [42], each record in L is initially given a random bit. While servicing a query to a record, if the record's random bit is 1, the item is moved to the front, otherwise it is kept in its original position. The random bit is toggled on each access. An illustration of

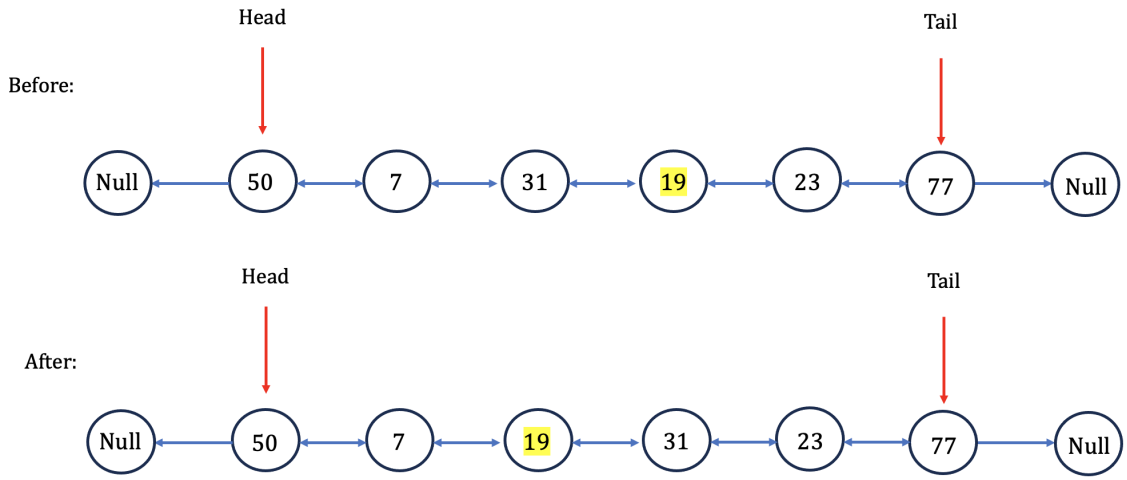

Figure 2-3: Transposition policy action when item 19 is accessed in L

its progression is shown in Figure 2-4. The BIT algorithm can be viewed as some randomized version of the move-to-front heuristic and its competitive ratio is $\frac{7}{8}$ as compared to move-to-front's 2. It seems, particularly for independent queries sampled from some skewed distribution, that a more conservative approach to moving records forward than move-to-front achieves a better competitive ratio. This same observation was made in [43] where Rivest shows that the transposition heuristic achieves a better competitive ratio than move-to-front. In general, algorithms like TIMESTAMP [4] and BIT [42] which refrain from always moving records to the front of the list on each access also have better competitive ratios.

The same observation does not hold for dependent accesses [32]. For instance, consider the scenario where we repeatedly access the last and second to last record. Then move-to-front will outperform all the other conservative algorithms, especially transposition. As a result, no one algorithm or heuristic is the best.

The list update problem is a canonical problem for online algorithm analysis be-

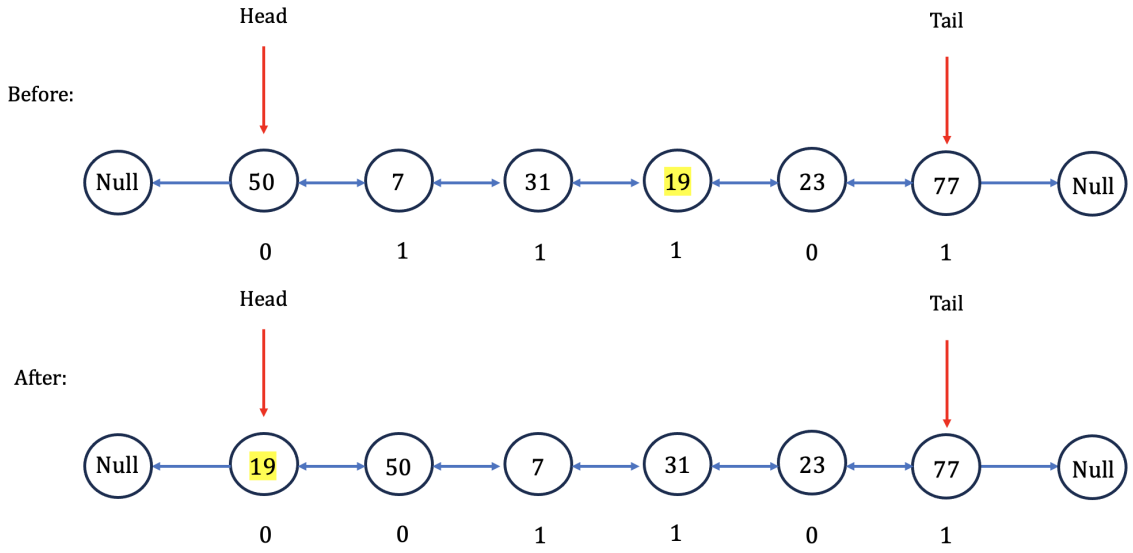

Figure 2-4: BIT algorithm action when item 19 is accessed in L.

cause of its simplicity, yet it has applications in caching and database systems [13]. Therefore, a performant list update algorithm has important practical benefits.

Chapter 3

Reinforcement Learning Agent Design

One of the primary goals of this thesis is to present a learned algorithm for the list update problem using reinforcement learning. In Chapter 2, we mentioned that there are a variety of reinforcement learning algorithms. In this chapter, we present our specific choice of algorithms and agent design. Later in chapter 7, we will discuss and justify the design choices presented here.

Besides optimality, a major goal for the learned algorithm is **adaptability**. That is, under different workloads, we want the algorithm to be competitive. This is especially important, because as we mentioned in Chapter 2, no one list update algorithm or heuristic is optimal for all query workloads. Having a single algorithm that can achieve near-optimal performance regardless of the query workload is desirable.

Explainability and interpretability is also of great importance. While the learned algorithms presented in [19] are faster than Strassen's, they are not trivial or easy to pen down. In contrast, we sought learned algorithms whose behaviour we could interpret. This way, we could analyze them to understand why it outperforms existing algorithms. We may then draw inspiration from them for new list update algorithms.

The primary reinforcement learning technique used is **model-free learning.** Specifically, we used **deep Q-learning**. In tabular Q-learning, the agent learns the Q-function, which is a function that maps a given state and an action to the value of taking that action in that state. Usually, this function will be computed through an iterative process of the agent exploring and earning rewards over time. The table is indexed by state and action pairs. However, when the state space and action space is large and complex, it is infeasible to compute the entire table. Instead, deep Q-learning uses a neural network to approximate the Q-function. Each component of the MDP for our list update problem on a list L is described below:

1. Environment: The environment comprises the current list L and the queries Q.

2. State: Ideally, the state should succinctly capture a faithful representation of the environment. To this end, the state representation at time t is the current query q_t and the current position of the record referenced by q_t. The state space, therefore, is of size $|L|^2$. We considered other state representations and a discussion of them can be found in Chapter 7.

3. Action: The set of actions consists of the item to be moved and the position

Name of hyper-parameter	Description	Value
Learning rate	Controls rate at which the neural network updates Q-values based on rewards seen	0.7
Epsilon	Controls the probability that the agent chooses a random policy vs using the Q-value learnt by model	Changes over time but increases from 0.01 to 1
Decay rate	Controls the rate at which epsilon is updated over time	1.25
Discount factor	Determines how much we discount future rewards	0.9
Batch size	Controls how large of a window from past experience the neural network should use in updating Q-values	128

Figure 3-1: Choice of reinforcement learning agent parameters

where the item should be placed. The action space also has size $|L|^2$.

4. Reward Function: At time step t where record x is queried, if x is found in position i and the agent elects to move it to position j, then the agent receives a reward according to the following function:

$$\begin{cases} -i & j \leq i \\ -j & j > i \end{cases}$$

5. Agent and Algorithm: The agent employs an epsilon greedy technique with the parameters shown in Figure 3-1 In learning the Q-function, we also use **experience replay** and **target learning** to achieve a more stable learning outcome. An experience is a tuple of a starting state, the action taken in that state, the reward earned for that action and the next state resulting from taking the action. In experience replay, as the agent takes a sequence of actions,

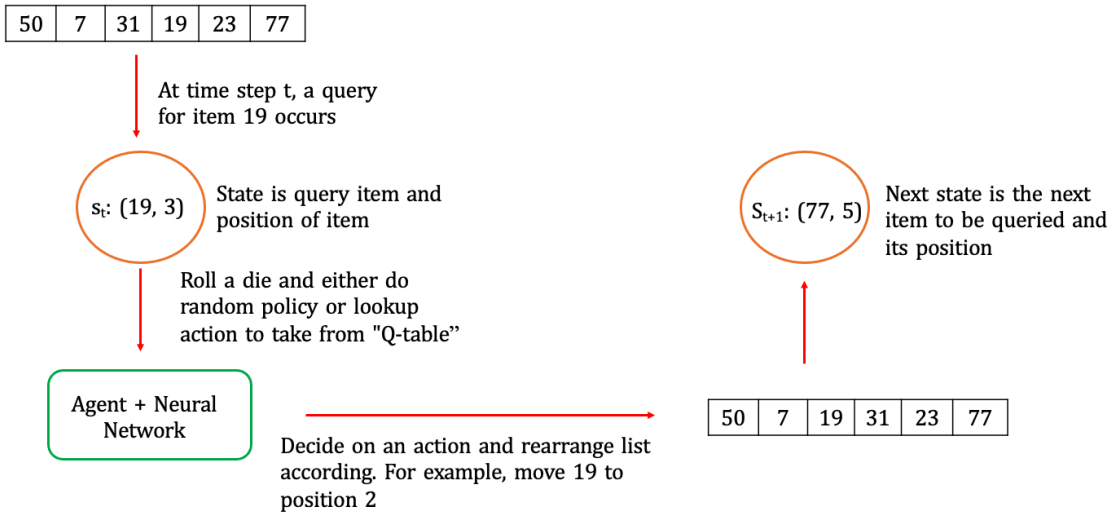

Figure 3-2: Walkthrough of MDP

we store each experience in a buffer called **experience buffer/memory**. We use a buffer size of 10000 experiences.

With target learning, we keep two copies of the neural network. One network is called the target and the other is our Q-table. All training and updates happen on the Q-table and then periodically, we transfer weights to the target network. Predictions and decisions are made with the target network which is more stable.

An example step through of the MDP can be found in Figure 3-2.

Lastly, the neural network architecture for approximating the Q-function in our deep Q-learning setup is found in Figure 3-3.

Layer Type	**Number of Units**	**Activation function**
Input Layer	$\|L\|^2$	-
Dense Layer	64 * 8	ReLu
Dense Layer	$\|L\|^2$	ReLu
Dense Layer	64*8	ReLu
Dense Layer	|A|	Linear

Figure 3-3: Deep Q-network Architecture

Chapter 4

Evaluating the Reinforcement Learning Agent

Since discovering an optimal and novel list update algorithm is the primary focus of this thesis, we evaluate the learned algorithm against existing list update algorithms. We compared it against move-to-front, transposition, order-by-access and do-nothing. We briefly explain each below:

1. Move-to-front: Each time a record is accessed it is moved to the front of the list.(See Figure 2-2)

2. Transposition: Each time a record is accessed it is swapped with the record preceding it. (See Figure 2-3)

3. Order-by-access: Maintain frequency counters for each record. Whenever a record is accessed, we update its frequency counter and shuffle it down the list until the element in front of it has a counter value greater than or equal to it.

4. Do-nothing: Each time a record is accessed, it is not moved. Under this policy, the arrangement of records in the list is not changed.

These algorithms provide useful benchmarks for evaluating the performance of the learned algorithm. In Chapter 5, where we attempt to understand the learned algorithm's behaviour, we will compare it with these algorithms too. To evaluate the learned algorithm and its chosen contemporaries, we set up the following experiment:

1. Initialize a static list L for each algorithm.

2. Generate a sequence of queries Q to be made to records in L. Q may be sampled from a distribution or constructed adversarially. We will discuss the different ways we generate Q in a later subsection of this chapter.

3. Simulate a workload over time using Q.

4. Allow the reinforcement learning agent T_{train} time steps of training so it can learn an optimal policy.

5. After it has learned a policy for the query sequence, stage a testing phase. In the testing phase, the agent does no learning. As each query comes in, it maintains L according to its learned policy. Each of the other algorithms also maintains their respective lists accordingly as queries come in.

6. During the testing phase, we measure and record cost metrics and move choices of each algorithm. A **move choice** is a pair of indices (x_1, x_2) representing the position a record was in (x_1) and the new position it was moved to (x_2).

7. Lastly, we repeat the experiment M times in order to ensure that the cost and behavior we observe is consistent. Repetitions of the experiment are called **episodes**. We allow the agent to carry over whatever it learns from episode to episode.

We conduct 80 different experiments which differ from each other in how queries are generated from episode to episode, yet the template of each experiment is roughly as described above. As we present results in this chapter and the next, there are four main types of graphs we make reference to:

1. Average cost over episodes: This graph shows, for each algorithm, the average query cost over of all queries in an episode.

2. Policy Map: This graph is a 2D heat map of move choices. Each grid/cell contains the frequency of a move choice. We construct one per algorithm. More details on how to read these graphs can be found in Chapter 5.

3. Normalized Policy Map: This graph is identical to the Policy Map except the frequency count is normalized for each column. This way, we can see the proportion of times a record in position x_1(horizontal axis) was moved to any of the $|L|$ positions of the list.

4. Transition Graph: This is a weighted directed graph. The list indices $(x_1, .., x_{|L|})$ are the vertices, the edges represent move choices of the algorithm and weights represent the frequency of move choices. More details on how to read these graphs can be found in Chapter 5.

4.1 Generating Query Sequences

As mentioned earlier, each experiment we ran differs by the manner in which the query sequence Q is generated. Figure 4-1 shows a schematic for how the query sequences were constructed. We expand on each below:

1. Distributional Generation: The sequence of queries for an episode are generated by sampling i.i.d from a given distribution. We consider three types of distributions:

   (a) **Zipfian**: Sometimes called a zeta distribution, the frequency distribution of items under Zipfian is related to their rank. Let L' be an ordering of items in L by frequency of access. Then the rank k element in L will have the following number of accesses over an episode:

   $$\frac{|Q|}{H^{\alpha}_{|L|}} \cdot \frac{1}{k^{\alpha}}$$

   where $H^{\alpha}_{|L|}$ is the harmonic sum $\sum_{i=0}^{|L|} \frac{1}{i^{\alpha}}$ and $\alpha \geq 1$ which determines the skew of the distribution. Figure 4-2 and 4-3 illustrates an example of the Zipfian distribution. An important note here is that we do not let L be L' in all experiments so we can see the policy of the learned algorithm. It is only useful to have it once or twice to test for adaptability.

   (b) **Uniform**: This is a uniform distribution over the list of items.

   (c) **Heavy/Light**: For this distribution, we designate a few items as "heavy" and others as "light". Heavy items are accessed more than light ones. For example, heavy items may comprise 95% of the query sequence, Q, and light items make up the remaining 5% of queries. The number of

heavy items can also range from 10% of $|L|$ to 50% of $|L|$. Figure 4-4 shows an example of this.

2. Adversarial Generation: Here, we generate the sequence of queries for an episode by choosing specific records to access, with the choice of record based mainly on its position in the list. The one adversarial sequence we test is a sequence that repeatedly accesses the last and second to last element, i.e. $x_{|L|}$ and $x_{|L|-1}$.

3. Episode-to-Episode variations: For both distributional and adversarial generation, we have different sets of episode-to-episode variation. We explain each below:

   (a) **No List change, No Distribution change**: As we go from episode to episode, the list L and the distribution over the list L from which we sample our queries does not change. This is a relatively easy workload for the learned algorithm because whatever it has learned in previous episodes is relevant for future epsisodes.

   (b) **No list change, Distribution change**: From episode to episode, the list L remains unchanged and hence the support of the distribution remains unchanged. The only thing that changes is the distribution frequency over the keys. So, for instance, the most frequently accessed item changes from episode to episode but the list is itself is unchanged. This experiment exists to ascertain the adaptability of the learned algorithm. If it can recognize that the most frequently accessed records have changed and modify its behavior accordingly, then it has achieved adaptability.

(c) **List change, No Distribution change**: Here, rather than change the access frequency distribution, we keep the distribution the same and change the support i.e. the list L. Again, this is a test of adaptability. We want to ensure that the learned algorithm does not merely memorize specific key values which it carries from episode to episode.

(d) **List change, Distribution change**: Here, both the list and the distribution changes. This is also a test of adaptability, to verify that the learned algorithm can adjust its policy in response to workload changes.

(e) **List change & No list change**: For the adversarial generation, we have two variants. In one variant, we change the list from episode to episode and in another variant, we do not change the underlying list L.

For simplicity, we will refer to a query sequence as either Zipfian, Uniform, Heavy/Light or Adversarial. The episode-to-episode variation will be specified where necessary. In Figure 4-5, we see a chart showing the optimal algorithm for each query sequence.

4.2 Results of Evaluation

The results presented here were obtained by running experiments on MIT CSAIL's slurm cluster. We used one cpu-per-task with 5GB of memory per task. We have three main evaluation criteria for the learned algorithm: **optimality across different query sequences**, **adaptability to changing query sequence** and **consistency at scale**. We present evaluation results for each criteria area in the sections below:

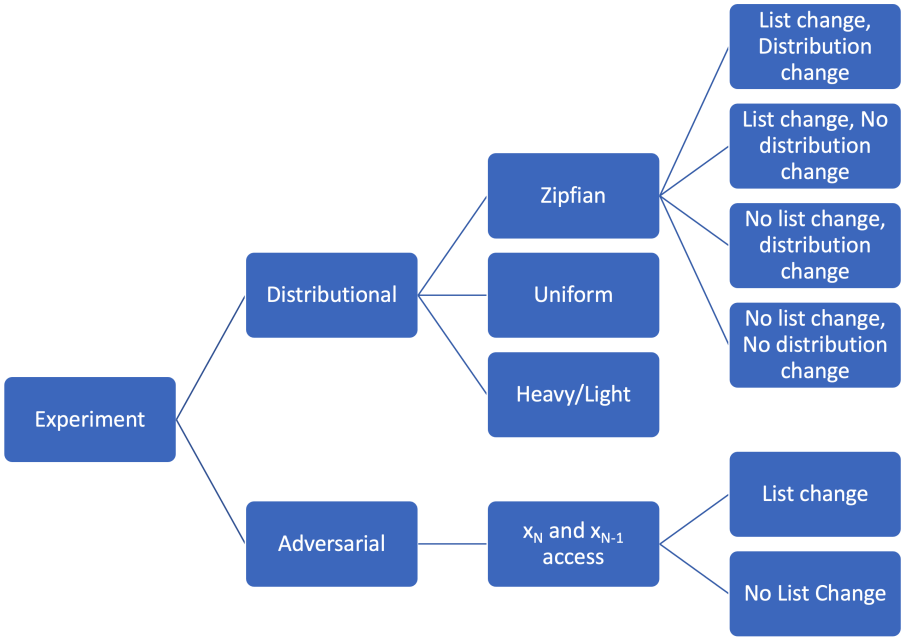

Figure 4-1: Different ways of generating Q. Note that we also conduct the same four variants for Uniform and Heavy/Light

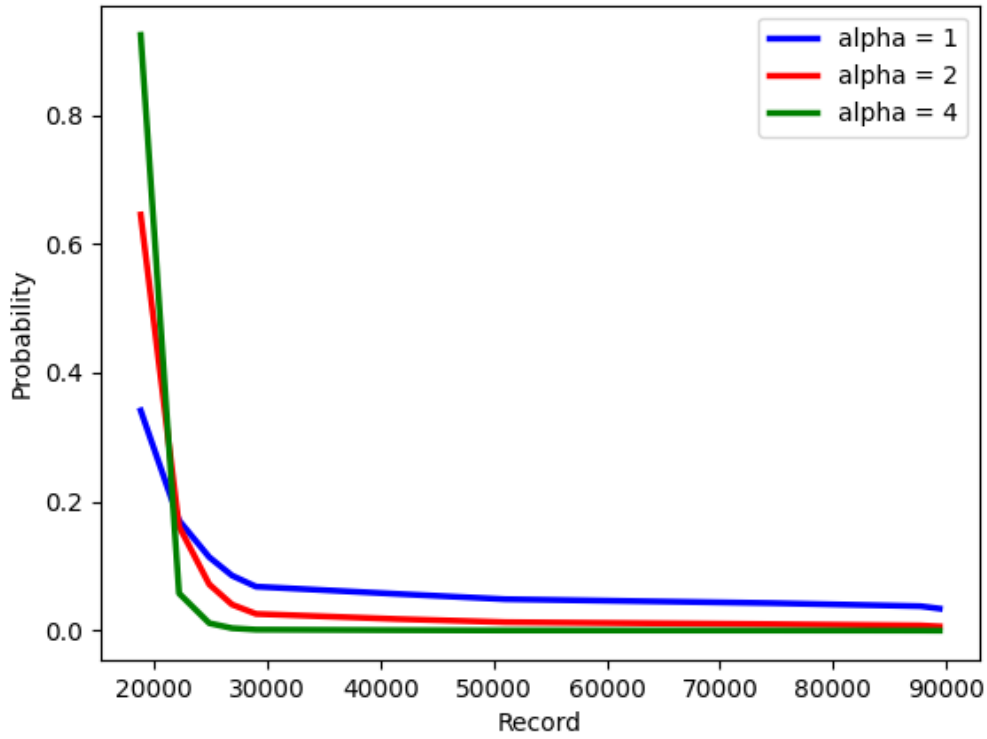

Figure 4-2: Example of a Zipfian distribution with different skews

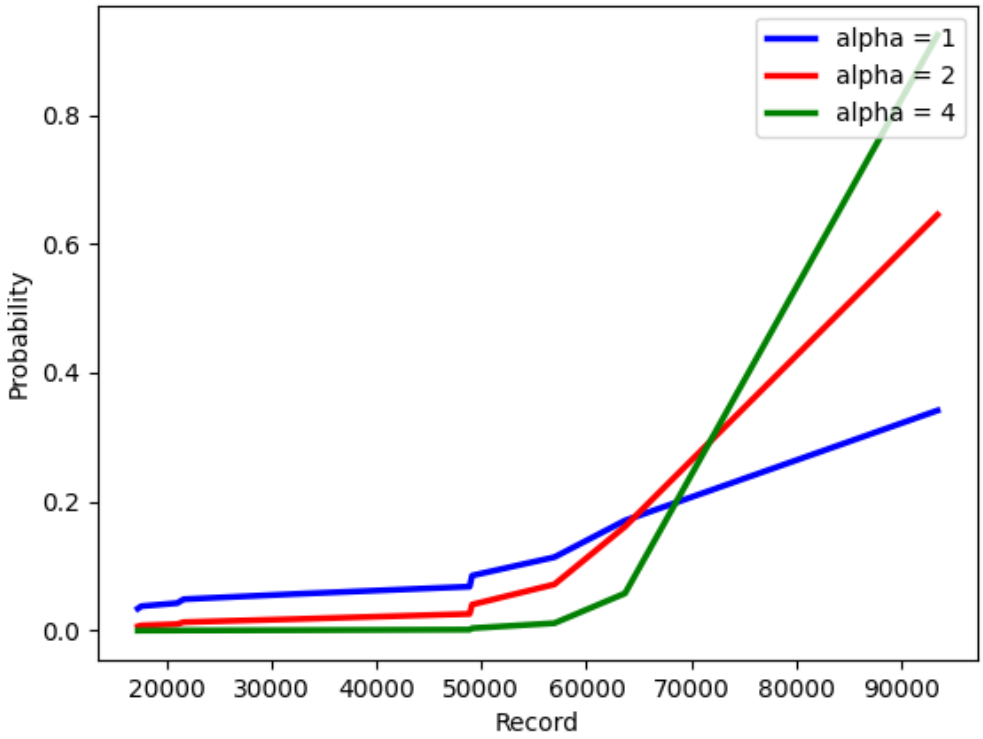

Figure 4-3: Example of a Zipfian distribution with different skews except the heavier records have larger key values.

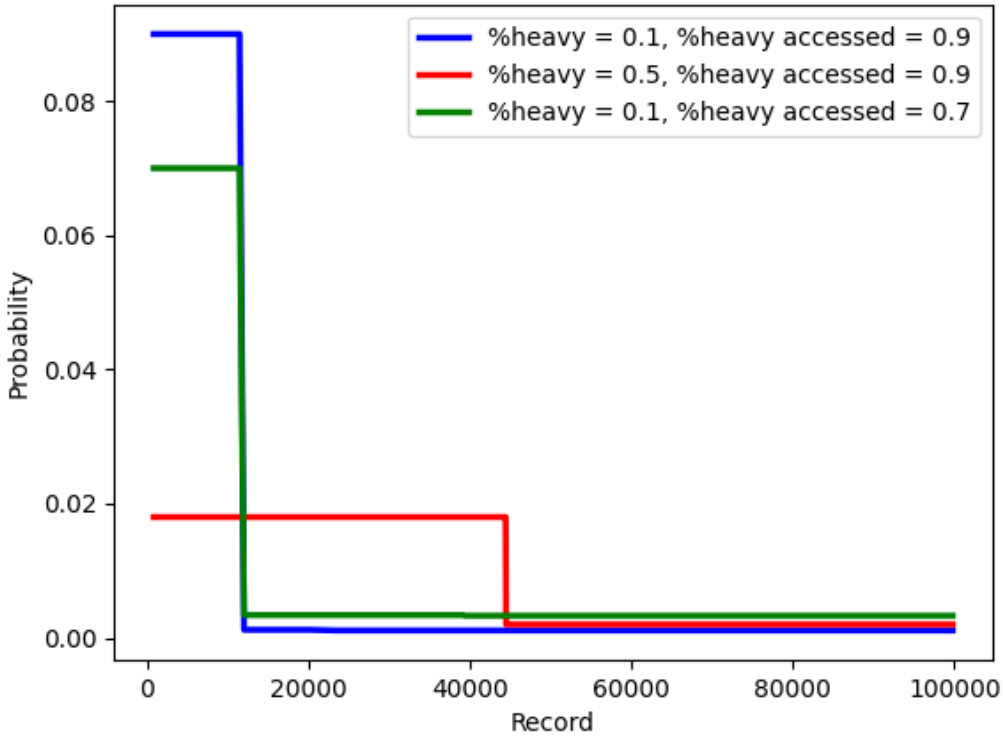

Figure 4-4: Example of a Heavy/Light distribution for a list of size 100. "%heavy" is the percentage of the list designated as "heavy" and "%heavy accessed" is the proportion of queries which constitutes heavy records.

Query Sequence	Optimal Algorithm
Zipfian	Order-by-Access
Uniform	Do-Nothing
Heavy/Light	Order-by-Access
Adverserial	Move-to-front/Order-by-Access

Figure 4-5: Optimal algorithms for each type of query sequence

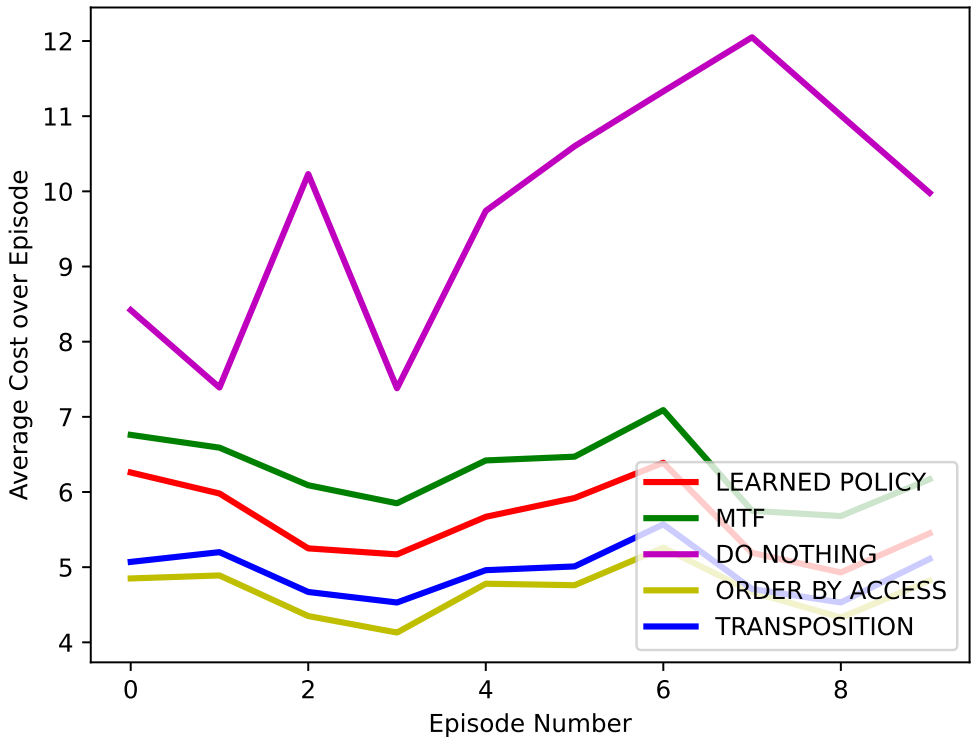

Figure 4-6: Results for Zipfian query sequence of length 1000 on list of size 20. No list change, no distribution change episode-to-episode variation.

4.2.1 Optimality Across Query Sequences

To observe optimality across query workloads, we focus on looking at performance for the episode-to-episode variation where the list and the distribution do not change. In Figure 4-6, 4-7, 4-8 and 4-9 we see the average cost of the learned algorithm on query sequences from Zipfian, Heavy/Light, Uniform and Adversarial respectively. Notice that the learned algorithm's performance is competitive with the best performing algorithm for each query sequence.

Consider for example the results for Zipfian in Figure 4-6 and the results for

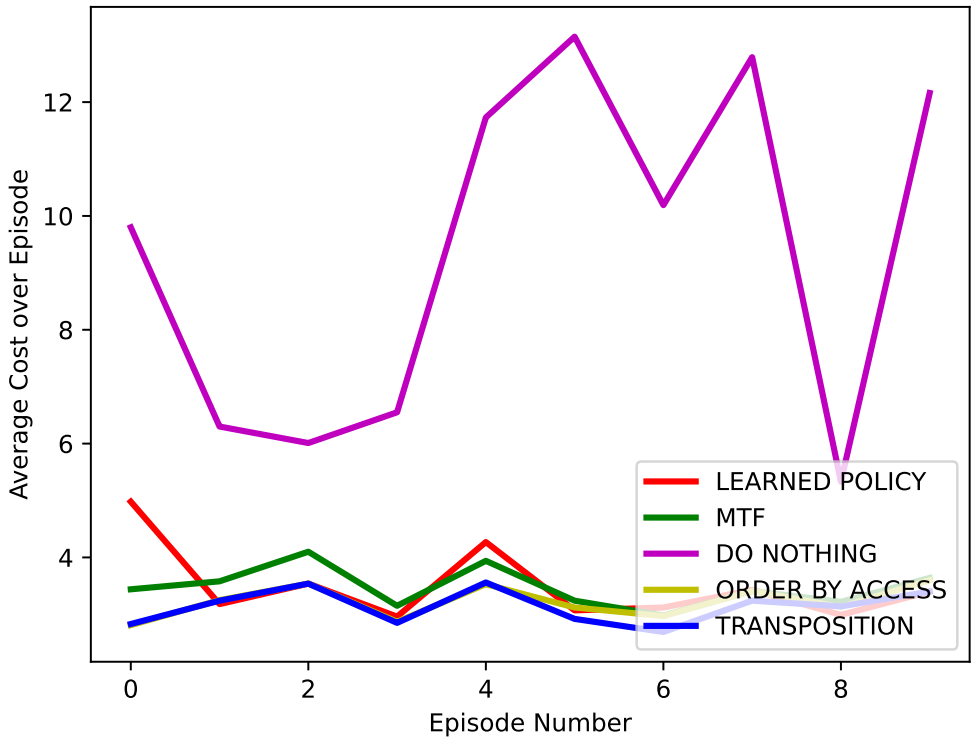

Figure 4-7: Results for Heavy/Light query sequence of length 1000 on list of size 20 10% of records are heavy and heavy records make up 75% of the query sequence. No list change, no distribution change episode-to-episode variation.

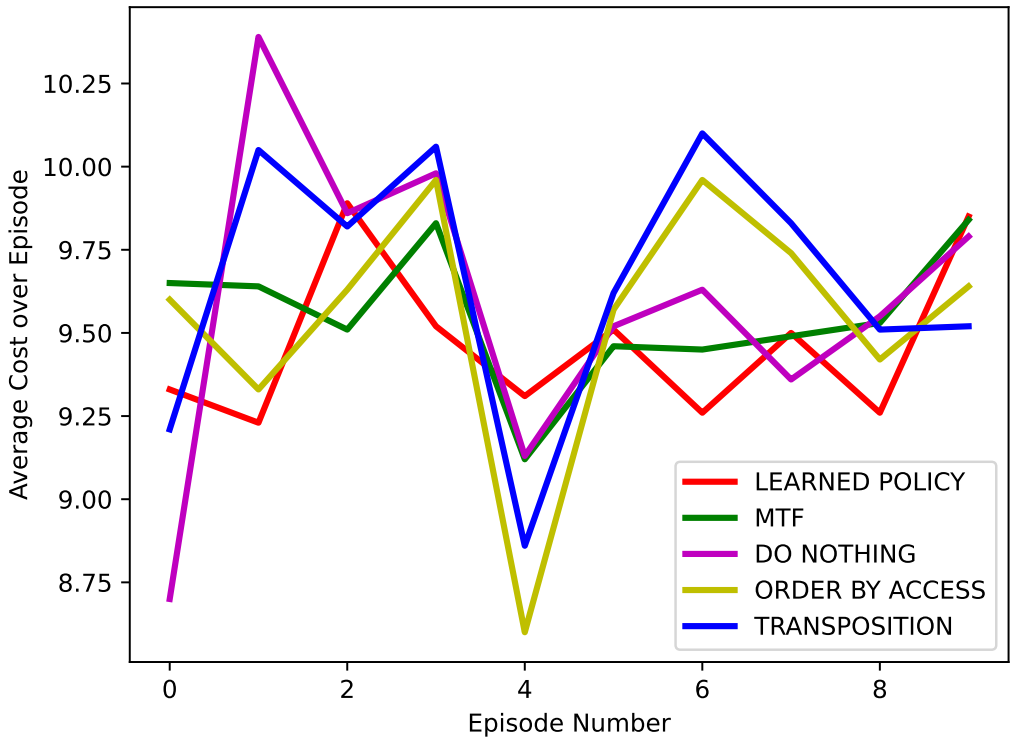

Figure 4-8: Results for Uniform query sequence of length 1000 on list of size 20. No list change, no distribution change episode-to-episode variation.

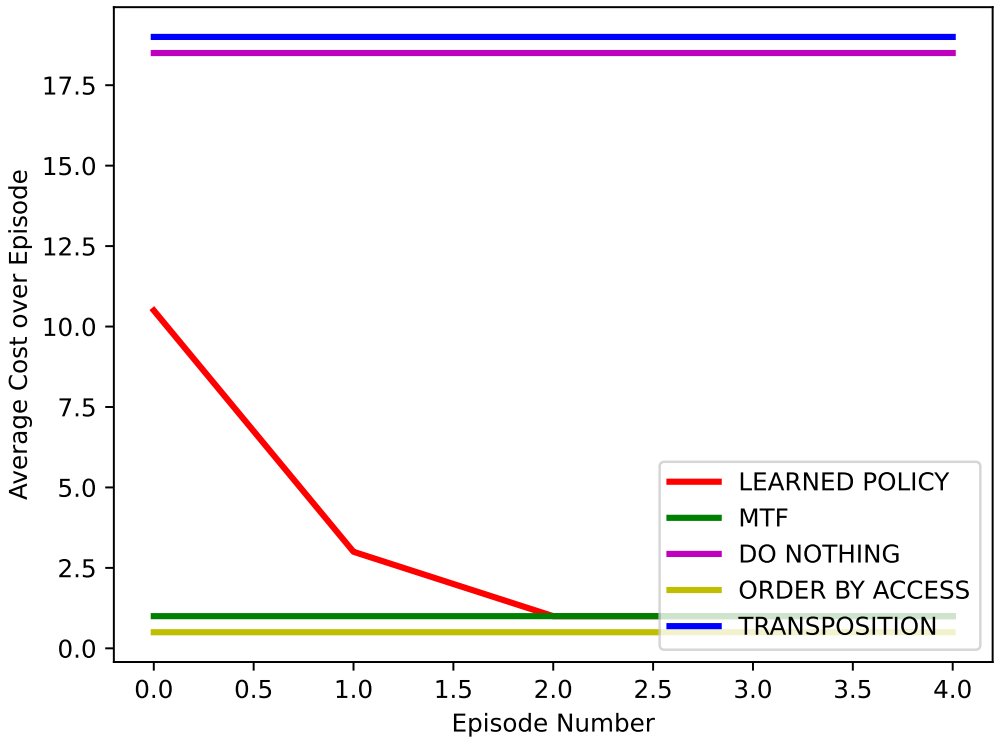

Figure 4-9: Results for Adversarial query sequence of length 1000 on list of size 20. No list change, no distribution change episode-to-episode variation.

Adversarial in 4-9. Move-to-front does well for Adversarial but is not as good as transposition for Zipfian. However, the learned algorithm is close to the best in both cases.

4.2.2 Adaptability to Changing Query Sequence

To evaluate the learned algorithm's adaptability, we consider how it performs on query sequences where the episode-to-episode variation involves the list changing or the distribution changing. If is capable of adapting to changing query workloads, then it should remain competitive with the best algorithm. In Figures 4-10, 4-11, 4-12 and 4-13 when the list changes but the distribution remains the same for a Zipfian, Heavy/Light, Uniform and Adversarial query sequence respectively, the learned algorithm still remains competitive.

A similar result is seen in Figures 4-14, 4-15, 4-16 and 4-17 for workloads where the list remains the same but the distribution over the list changes.

Lastly, when the list and distribution both change, Figures 4-18, 4-19, 4-20 and 4-21 also show that the learned algorithm remains competitive for both Zipfian and Heavy/Light query sequences. For the adversarial distribution, as the list and distribution changes, the learned algorithm does not perform as well but it is still better than transposition.

This means that even if the set of records that are frequently accessed changes, the learned algorithm will recognize this and adapt accordingly much like existing list update algorithms.

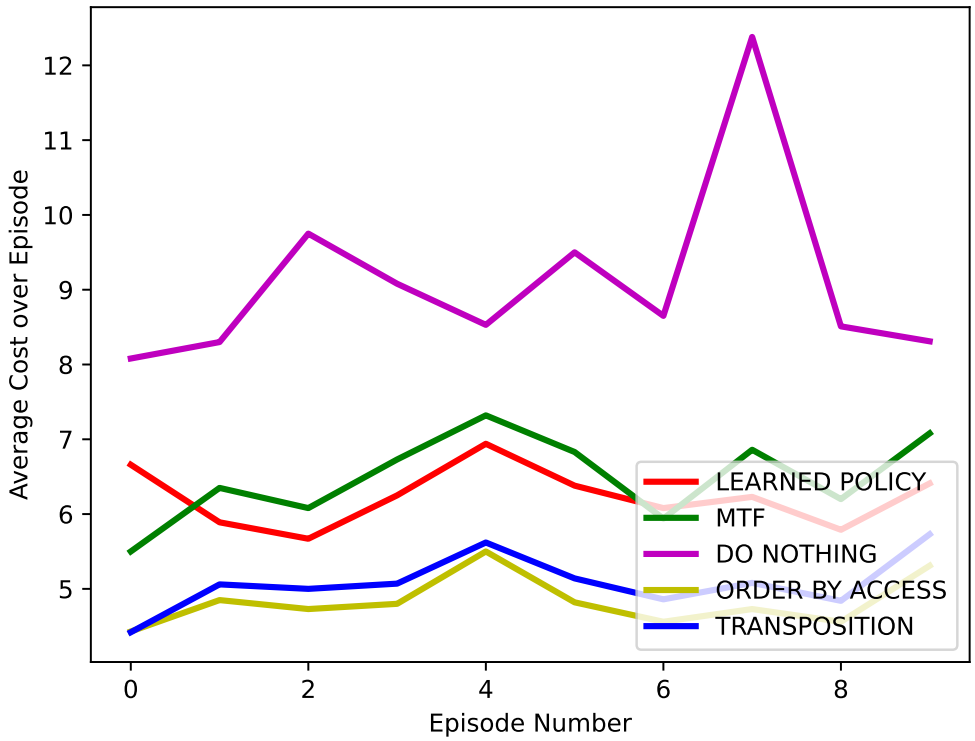

Figure 4-10: Results for Zipfian query sequence of length 1000 on list of size 20. List change, no distribution change episode-to-episode variation.

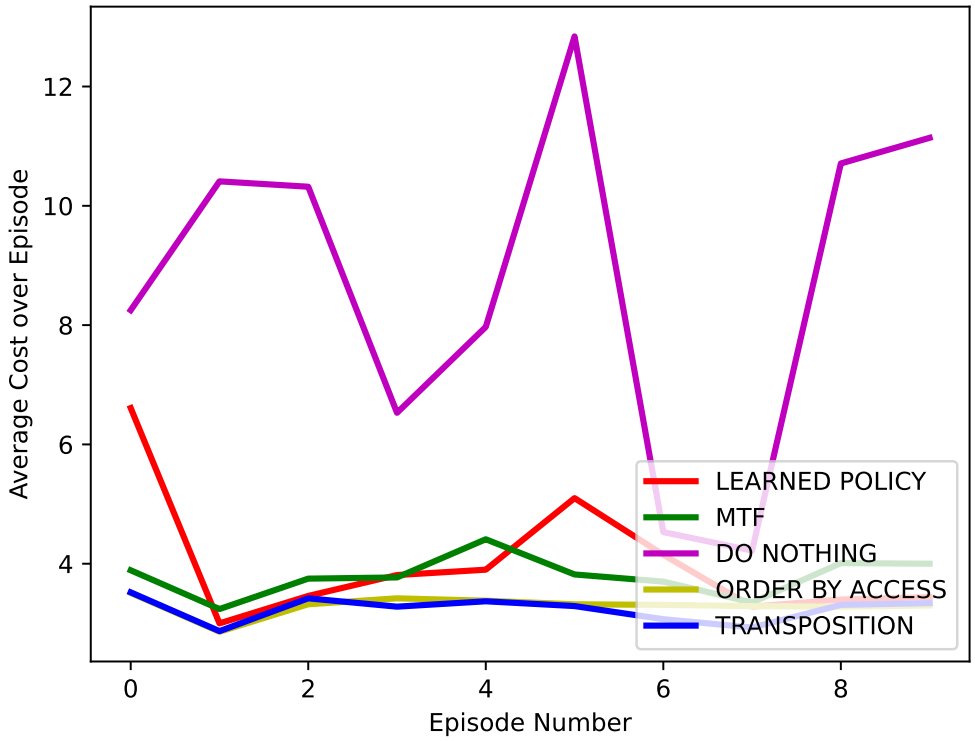

Figure 4-11: Results for Heavy/Light query sequence of length 1000 on list of size 20. 10% of records comprise 75% of the query sequence and the remaining 90% make up the rest of the query sequence. List change, no distribution change episode-to-episode variation.

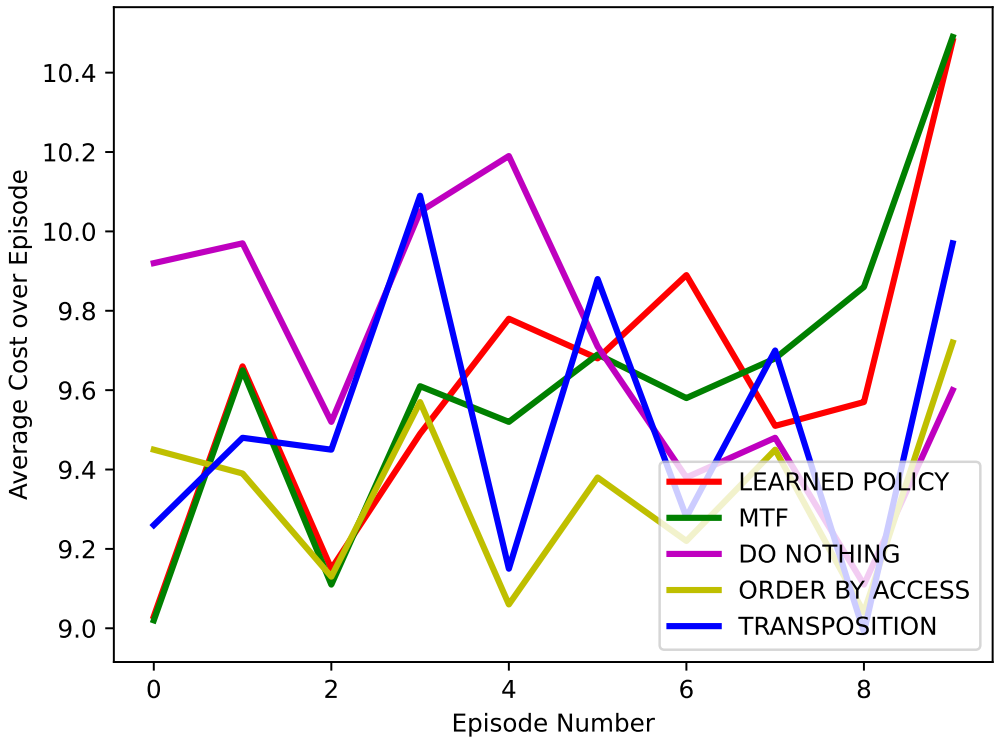

Figure 4-12: Results for Uniform query sequence of length 1000 on list of size 20. List change, no distribution change episode-to-episode variation.

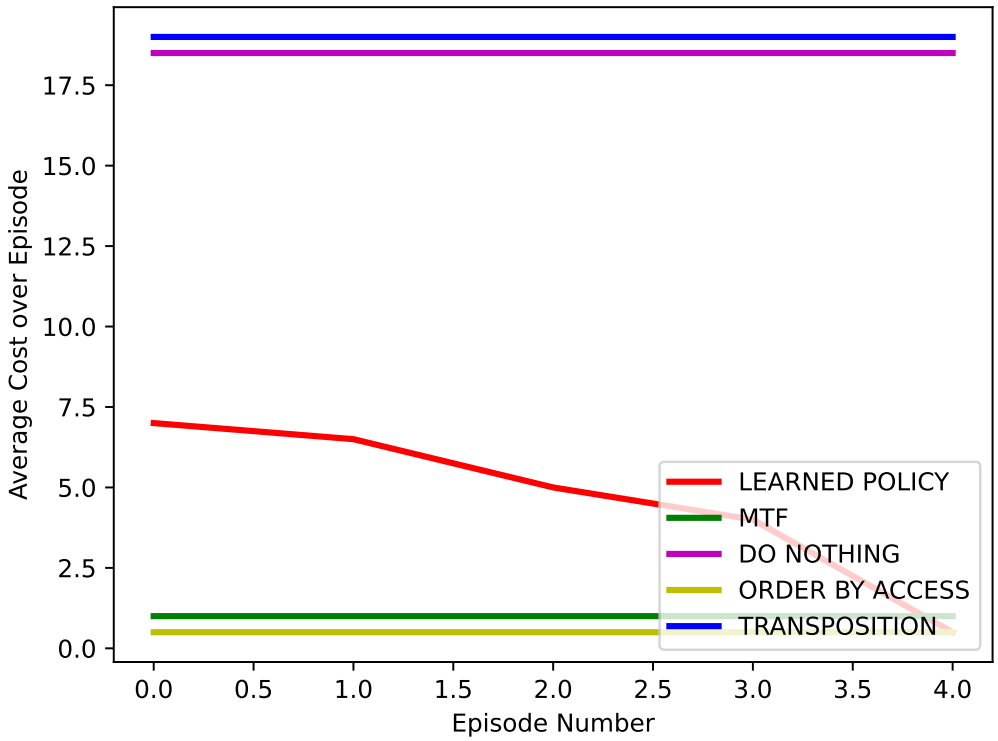

Figure 4-13: Results for Adversarial query sequence of length 1000 on list of size 20. List change, no distribution change episode-to-episode variation.

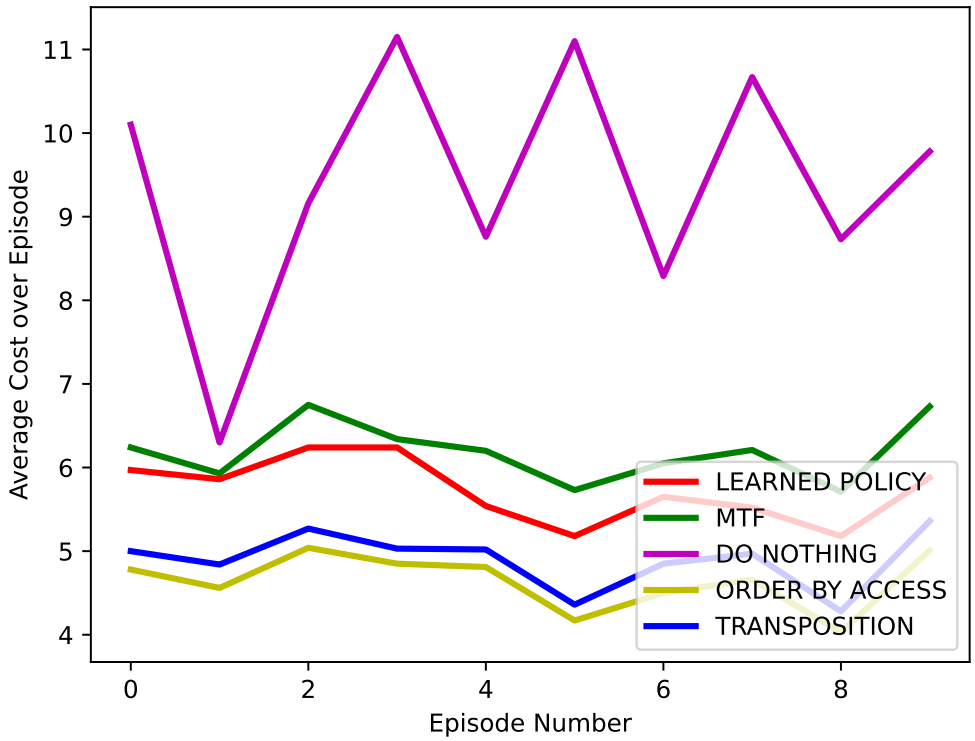

Figure 4-14: Results for Zipfian query sequence of length 1000 on list of size 20. No list change, Distribution change episode-to-episode variation.

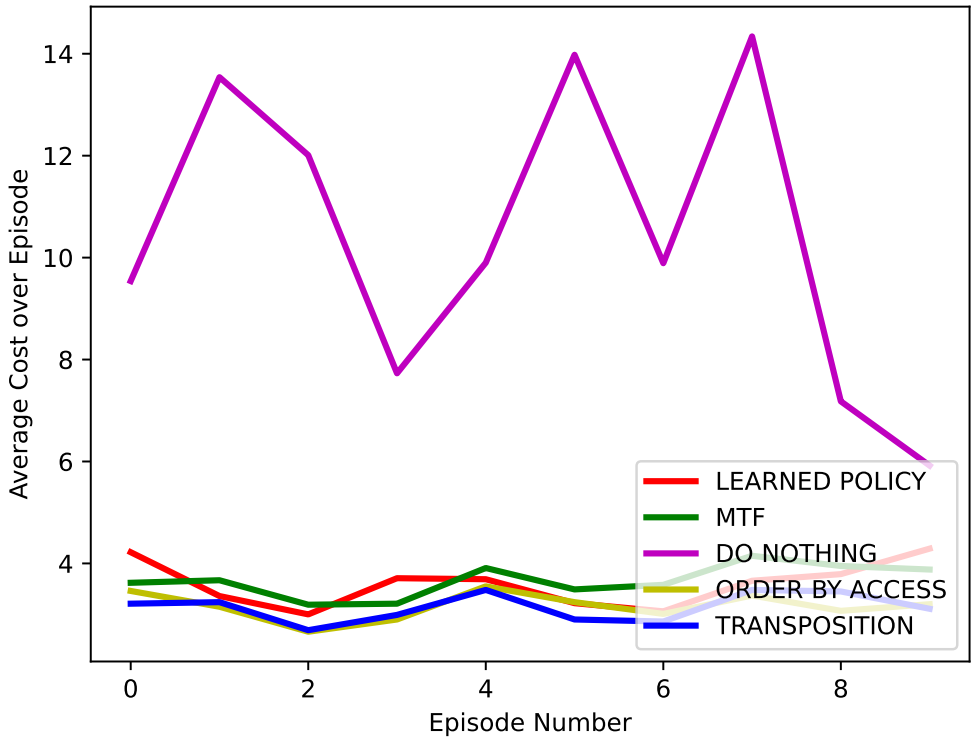

Figure 4-15: Results for Heavy/Light query sequence of length 1000 on list of size 20. 10% of records comprise 75% of the query sequence and the remaining 90% make up the rest of the query sequence. No list change, Distribution change episode-to-episode variation.

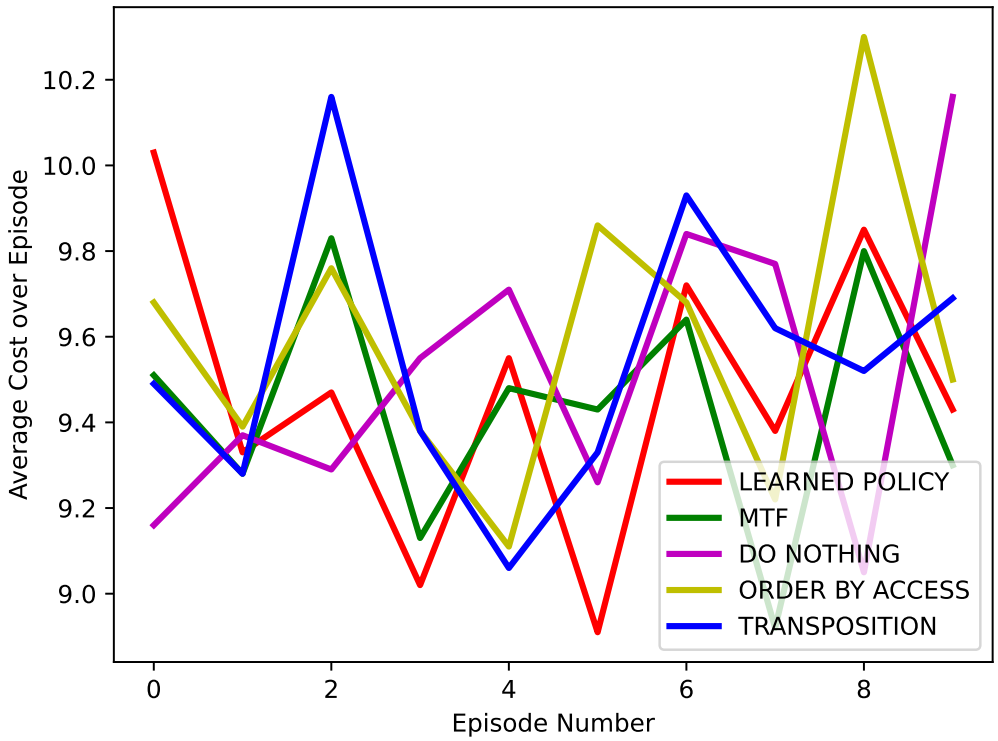

Figure 4-16: Results for Uniform query sequence of length 1000 on list of size 20. No list change, Distribution change episode-to-episode variation.

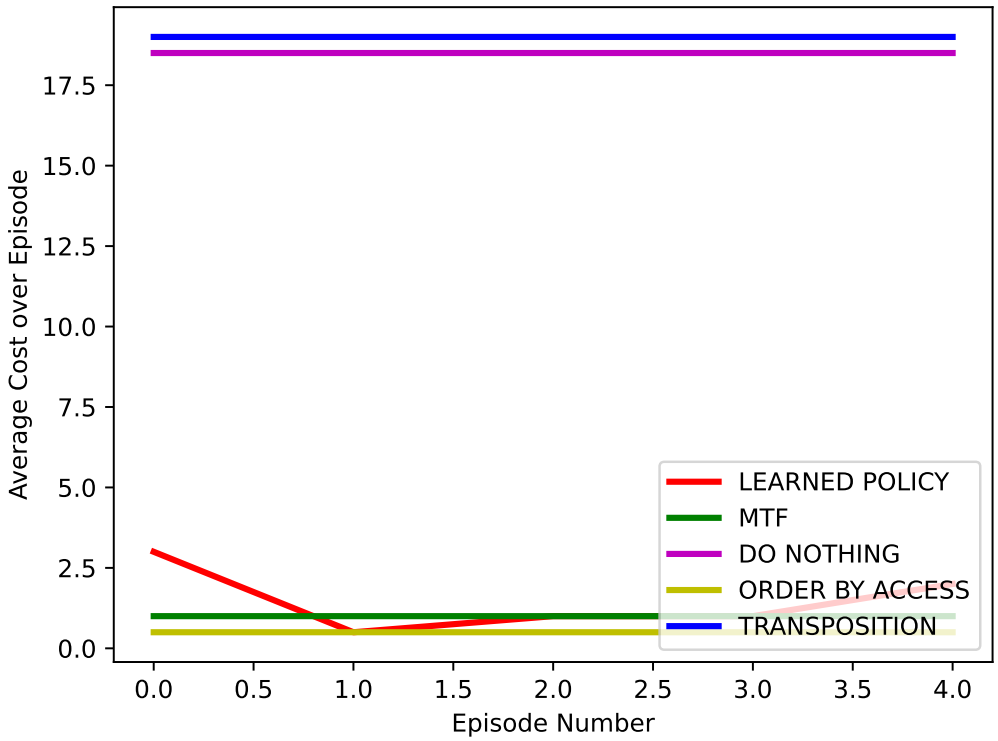

Figure 4-17: Results for Adversarial query sequence of length 1000 on list of size 20. No list change, Distribution change episode-to-episode variation.

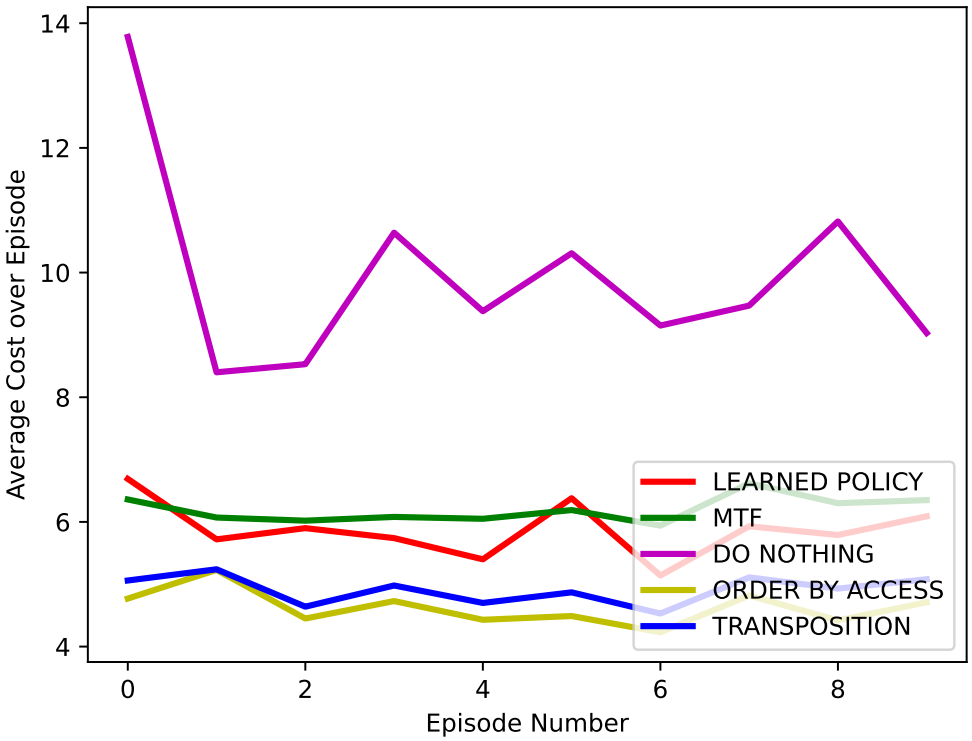

Figure 4-18: Results for Zipfian query sequence of length 1000 on list of size 20. List change, distribution change episode-to-episode variation.

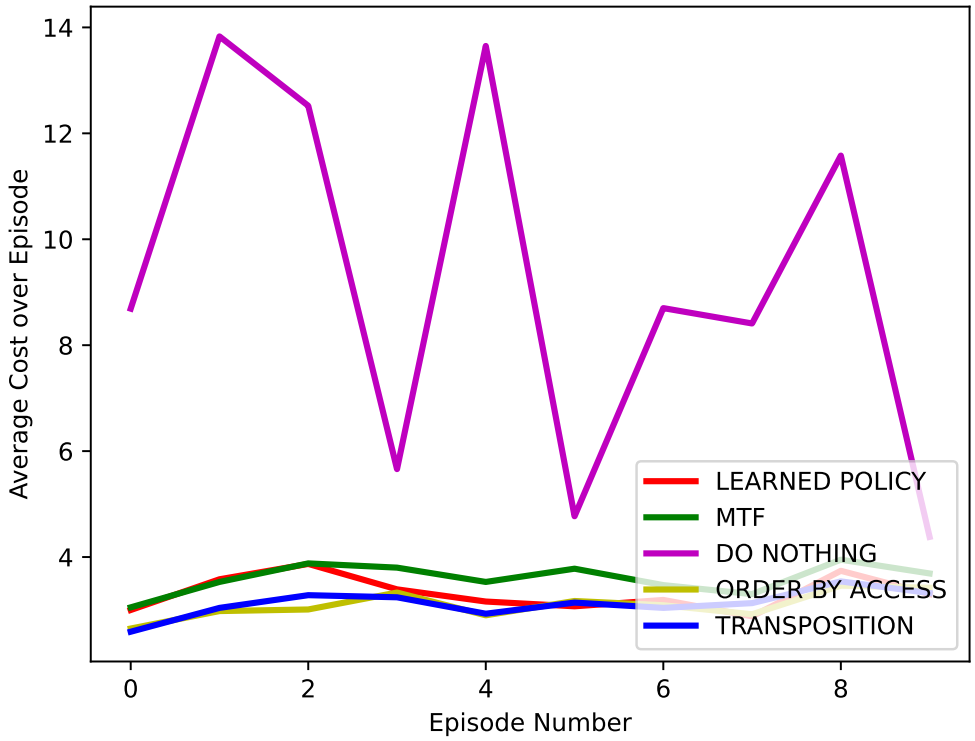

Figure 4-19: Results for Heavy/Light query sequence of length 1000 on list of size 20. 10% of records comprise 75% of the query sequence and the remaining 90% make up the rest of the query sequence. List change, distribution change episode-to-episode variation.

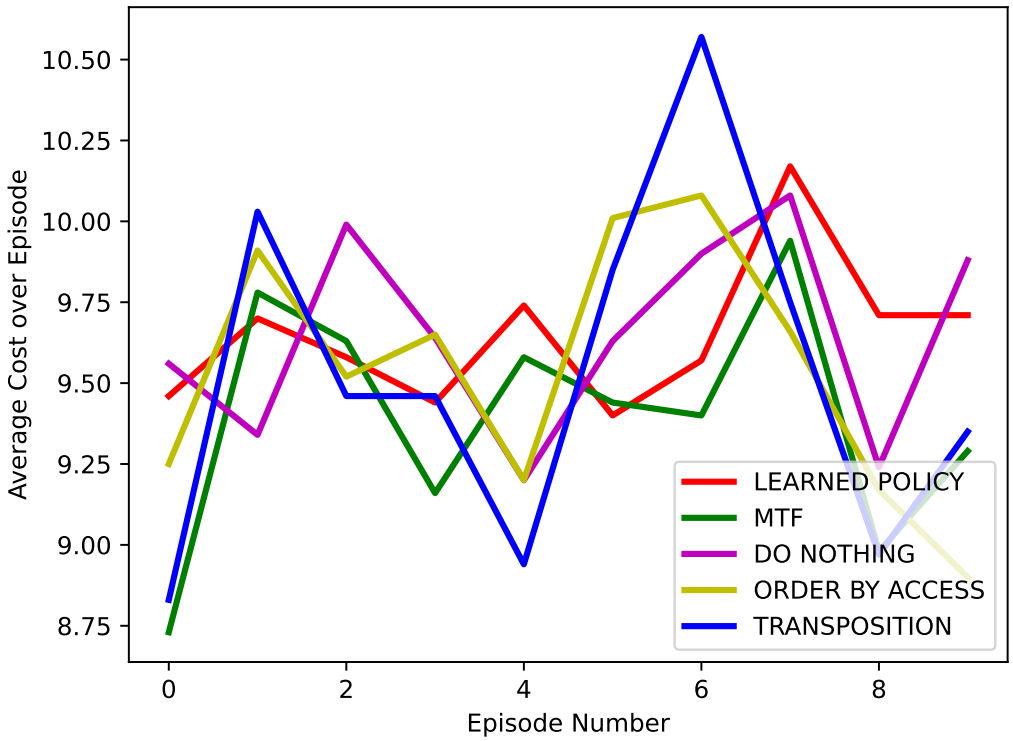

Figure 4-20: Results for Uniform query sequence of length 1000 on list of size 20. List change, distribution change episode-to-episode variation.

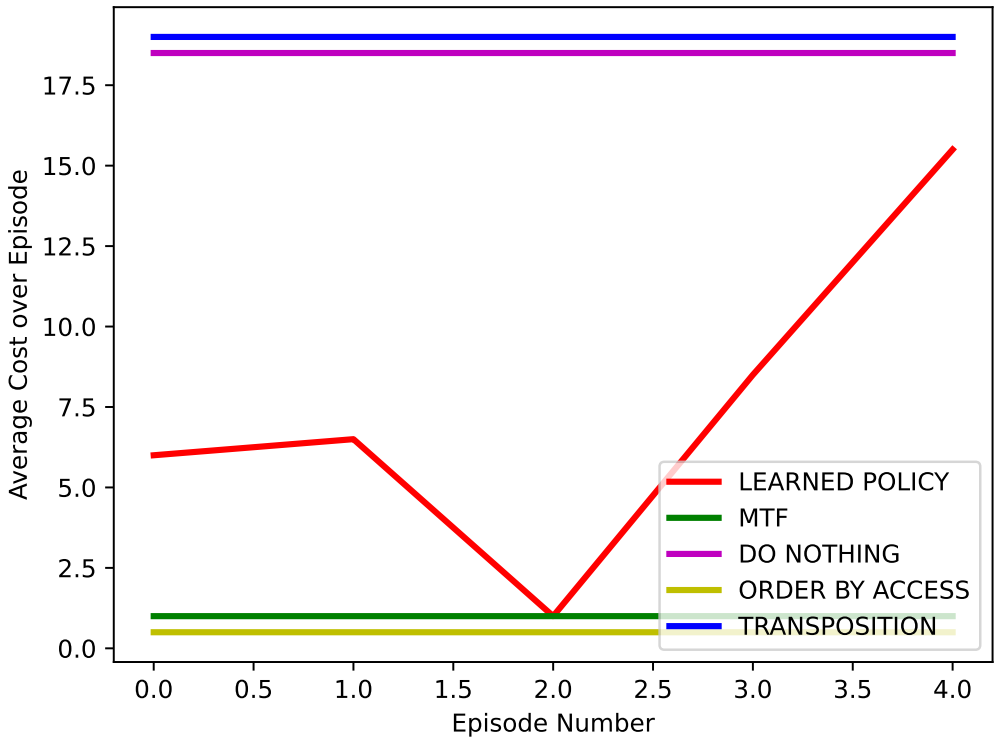

Figure 4-21: Average Search Cost for Adversarial query sequence of length 1000 on list of size 20. List change, distribution change episode-to-episode variation.

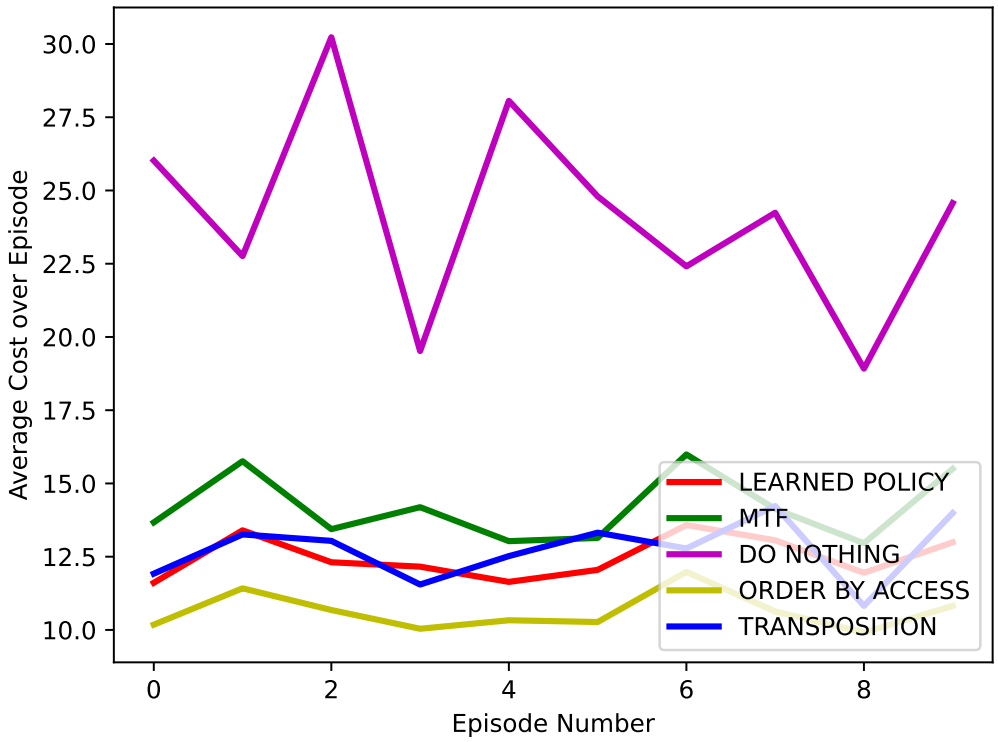

Figure 4-22: Results for Zipfian query sequence of length 1000 on list of size 50. No list change, No distribution change episode-to-episode variation.

4.2.3 Consistency of performance at scale

Another important metric we evaluate the learned algorithm on is whether its performance is consistent as the list size changes. In Figures 4-22, 4-23 and 4-24, we see the same experiment as in Figure 4-6 now on lists of size 50, 100 and 1000 respectively. The learned algorithm continues to remain competitive despite the changing list size.

Figures 4-25, 4-26 and 4-27 confirm this for Heavy/Light query sequence access, Figures 4-28, 4-29 and 4-30 confirm this for Uniform query sequences and

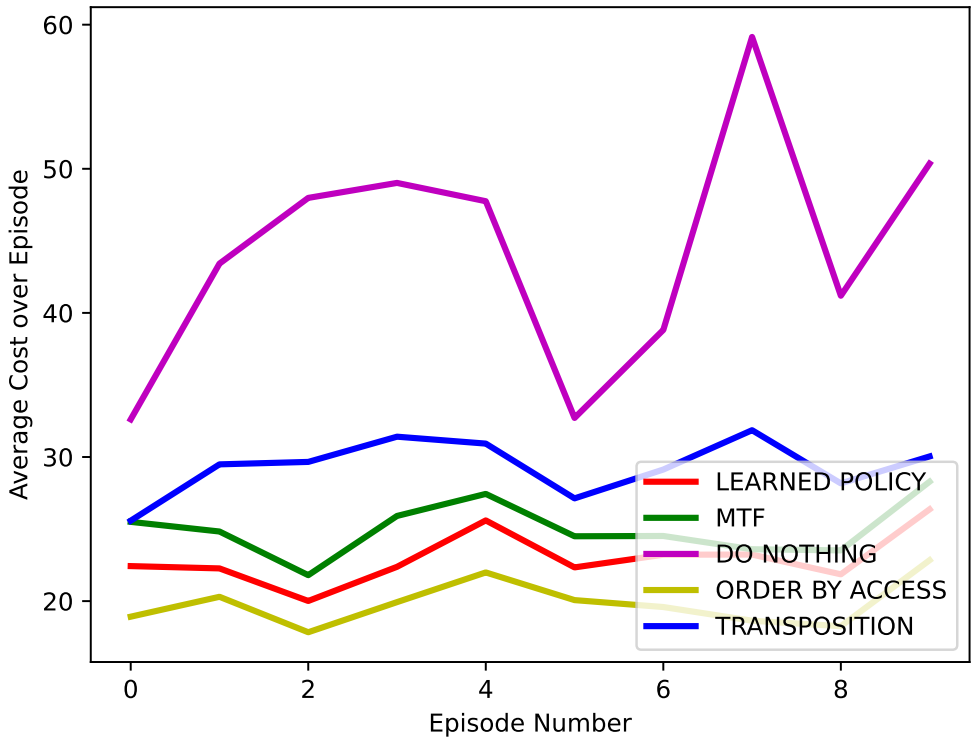

Figure 4-23: Results for Zipfian query sequence of length 1000 on list of size 100. No list change, No distribution change episode-to-episode variation.

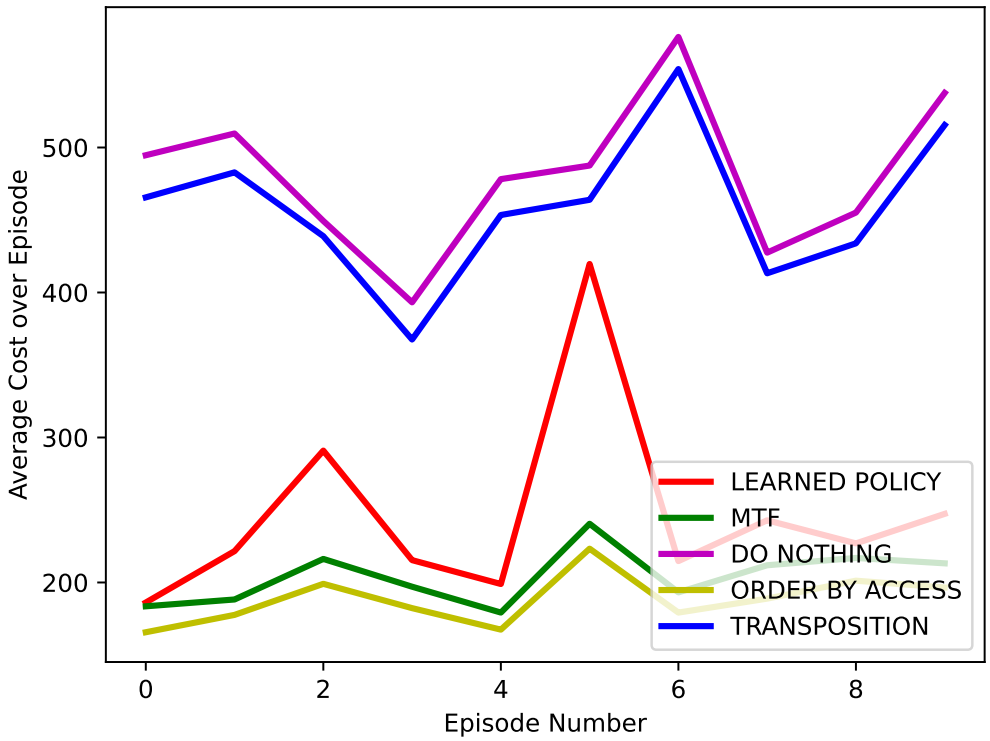

Figure 4-24: Results for Zipfian query sequence of length 1000 on list of size 1000. No list change, No distribution change episode-to-episode variation.

Figures 4-31 and 4-32 confirm it for Adversarial query sequences.

4.3 Results summary & Final notes

A link to all the results for the different variants of the experiment for each query sequence can be found in the Appendix. We make a few remarks about the results presented here:

1. In Chapter 2, we mentioned that the transposition algorithm has a lower expected search cost than move-to-front for independent accesses but in Figures 4-23, 4-24, 4-26, 4-27, 4-29 and 4-30 we see that it has a higher average cost than move-to-front. It is true that **asymptotically** transposition is better than move-to-front for independent accesses but transposition takes longer than move-to-front to converge. In the graphs referenced, there were not enough accesses for transposition's list to reach its stationary distribution, hence the higher than expected average search cost. It is likely this may be the case for the learned algorithm in certain cases(e.g. Figure 4-24). We will understand why once we analyze it's behaviour in Chapter 5.

2. As the list size increases, the performance differences between the algorithms become more pronounced.

3. Lastly, what might look like a small difference in cost, e.g. a ten percent difference in cost, could be the difference between a cache hit or miss and so these differences matter in practice.

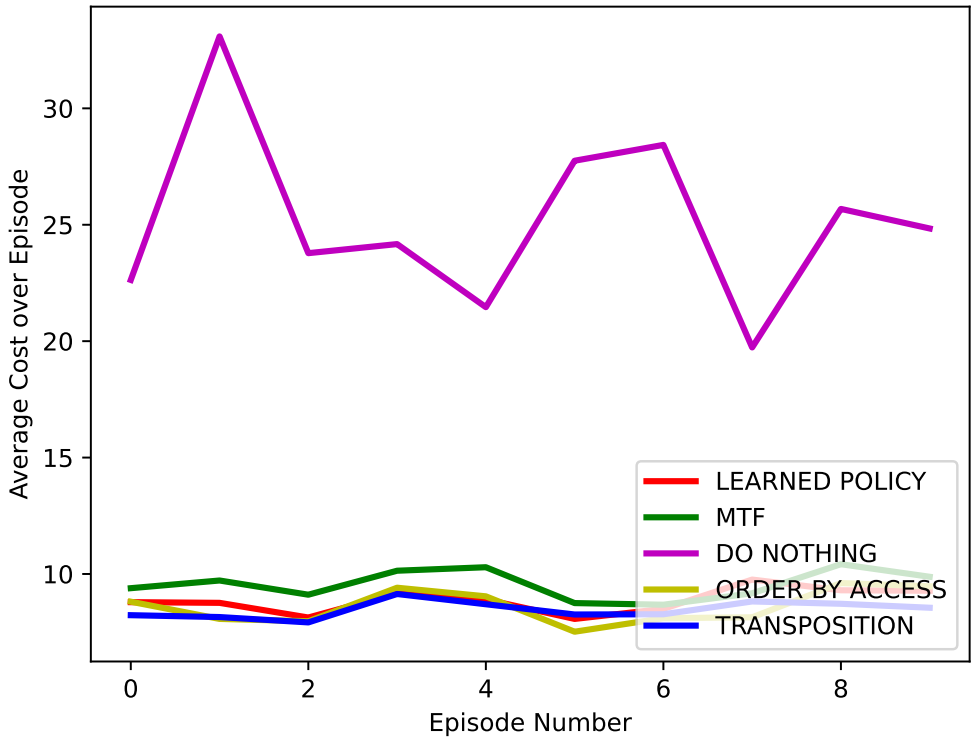

Figure 4-25: Results for Heavy/Light query sequence of length 1000 on list of size 50. 10% of records comprise 75% of the query sequence and the remaining 90% make up the rest of the query sequence. No list change, no distribution change episode-to-episode variation.

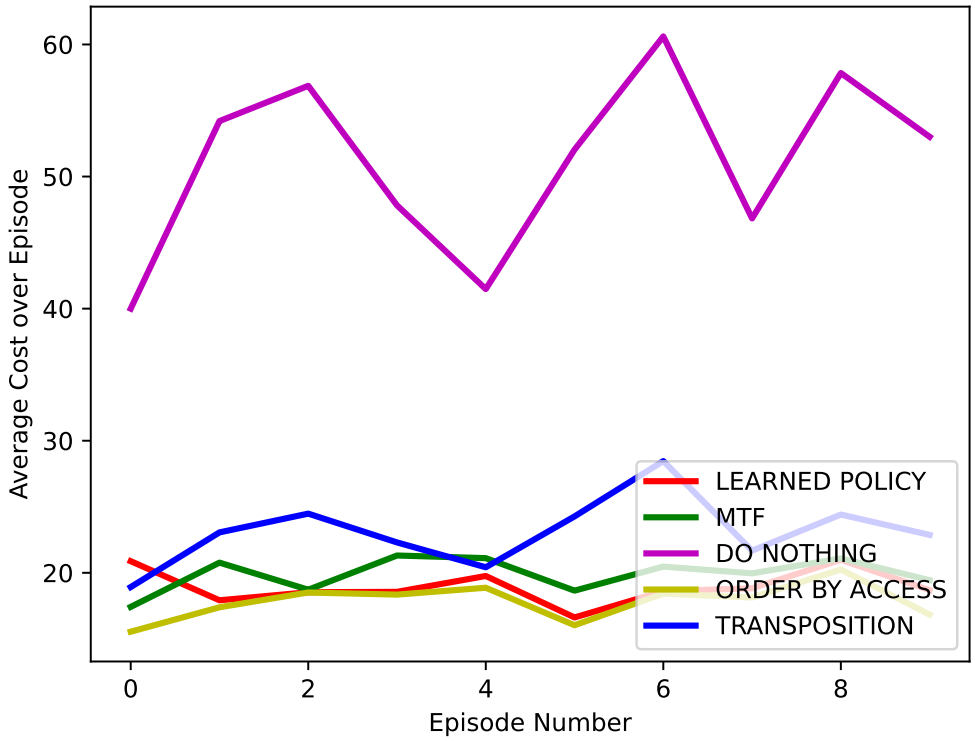

Figure 4-26: Results for Heavy/Light query sequence of length 1000 on list of size 100. 10% of records comprise 75% of the query sequence and the remaining 90% make up the rest of the query sequence. No list change, no distribution change episode-to-episode variation.

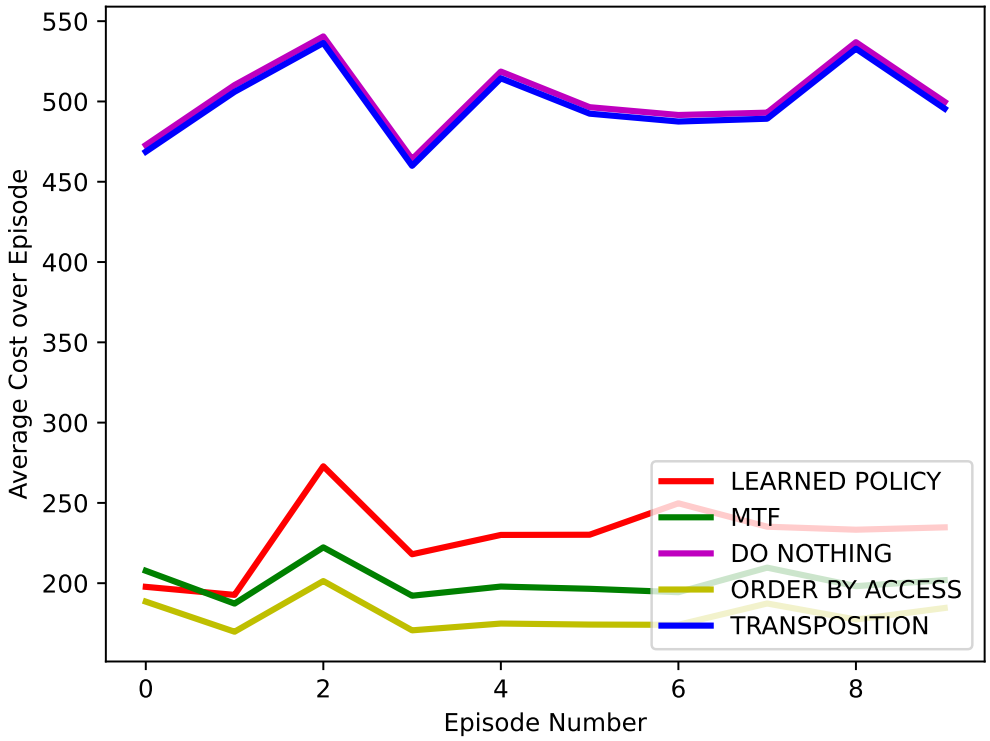

Figure 4-27: Results for Heavy/Light query sequence of length 1000 on list of size 1000. 10% of records comprise 75% of the query sequence and the remaining 90% make up the rest of the query sequence. No list change, no distribution change episode-to-episode variation.

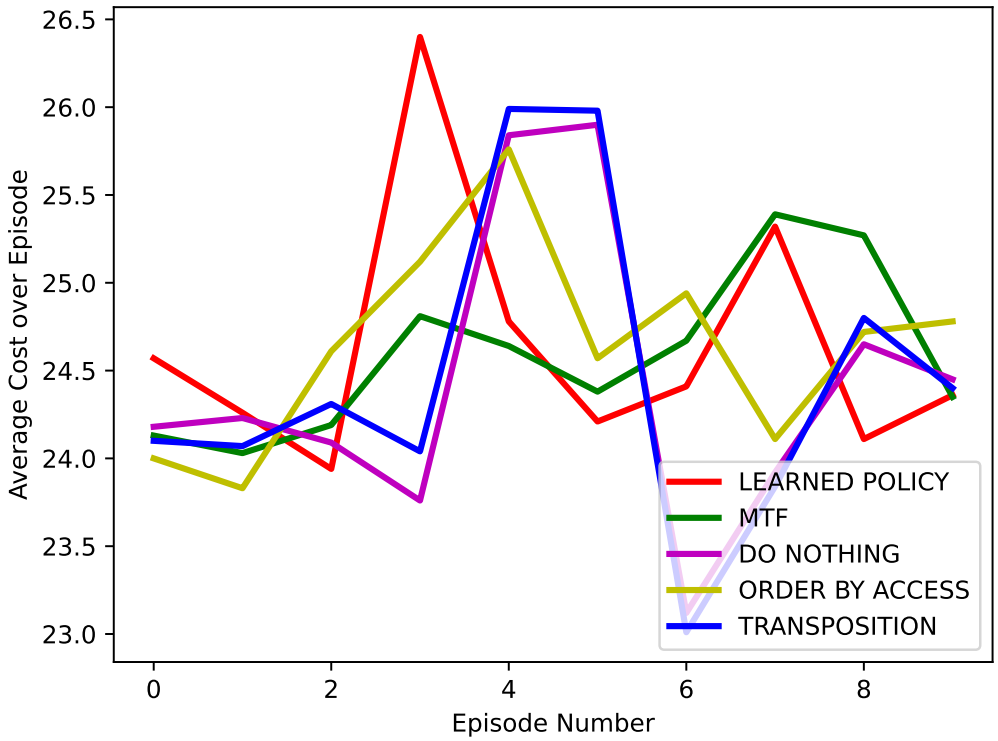

Figure 4-28: Results for Uniform query sequence of length 1000 on list of size 50. No list change, No distribution change episode-to-episode variation.

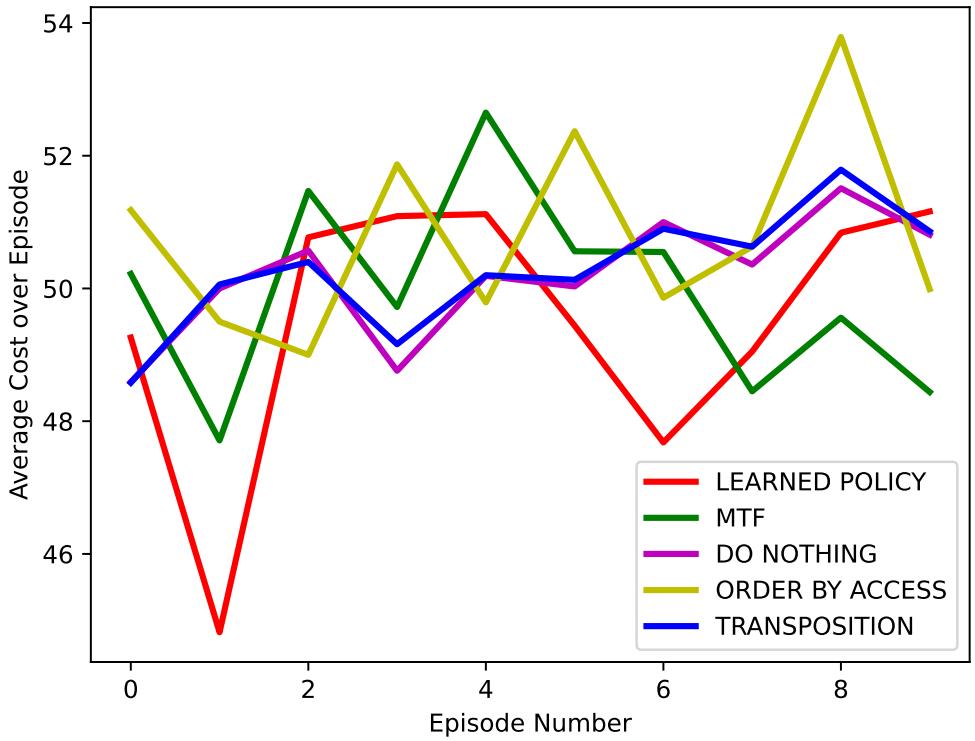

Figure 4-29: Results for Uniform query sequence of length 1000 on list of size 100. No list change, No distribution change episode-to-episode variation.

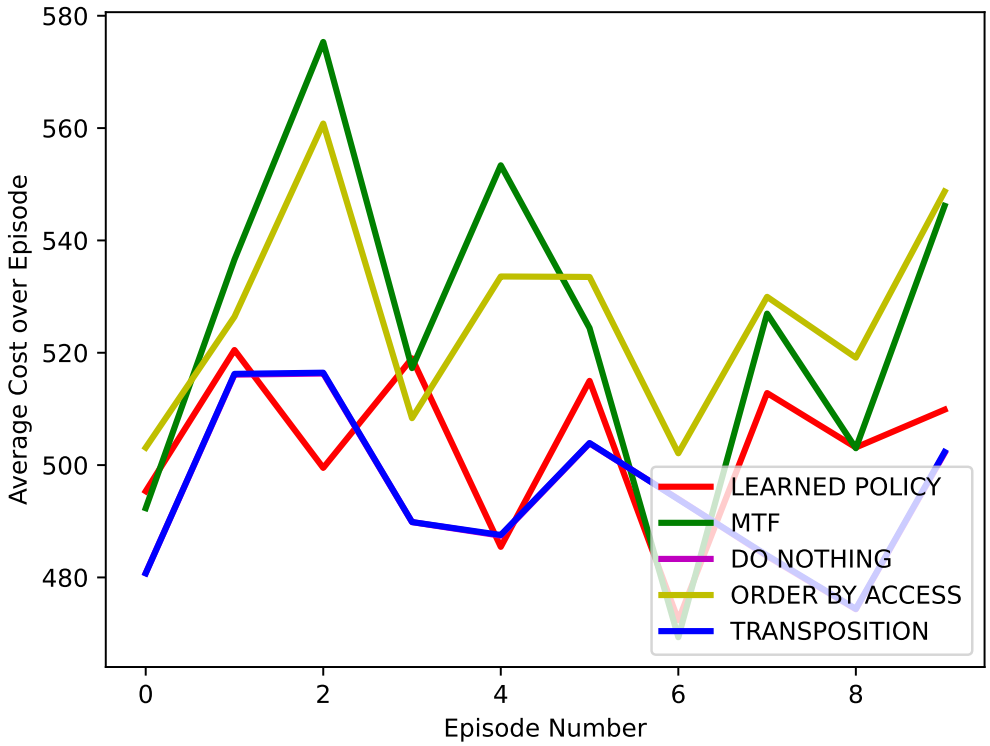

Figure 4-30: Results for Uniform query sequence of length 1000 on list of size 1000. No list change, No distribution change episode-to-episode variation.

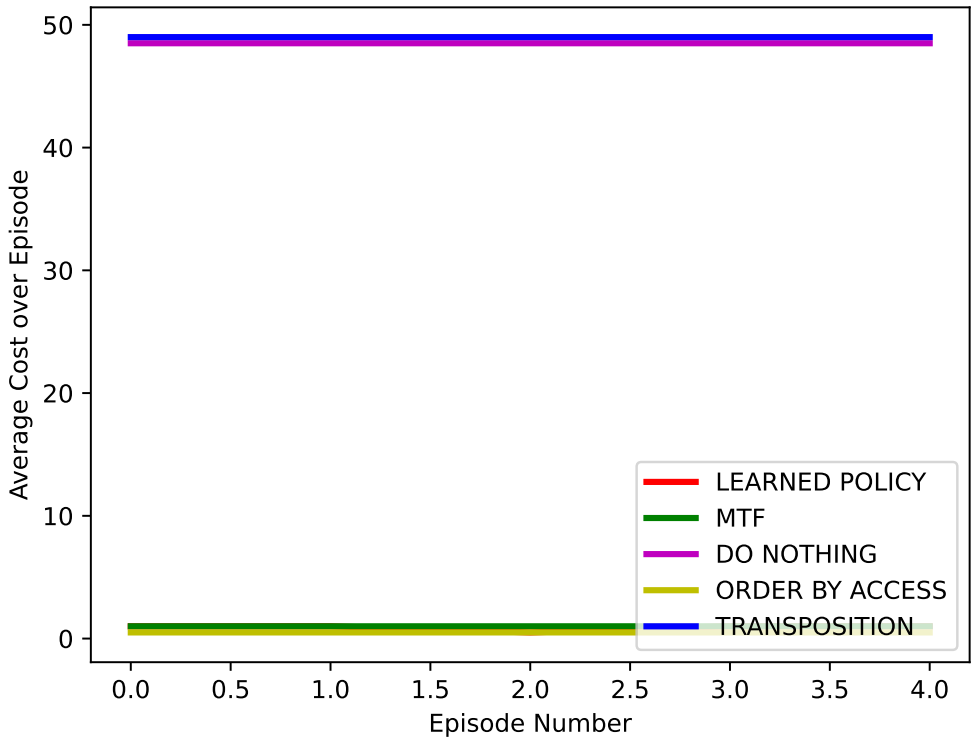

Figure 4-31: Results for Adversarial query sequence of length 1000 on list of size 50. No list change, no distribution change episode-to-episode variation.

and

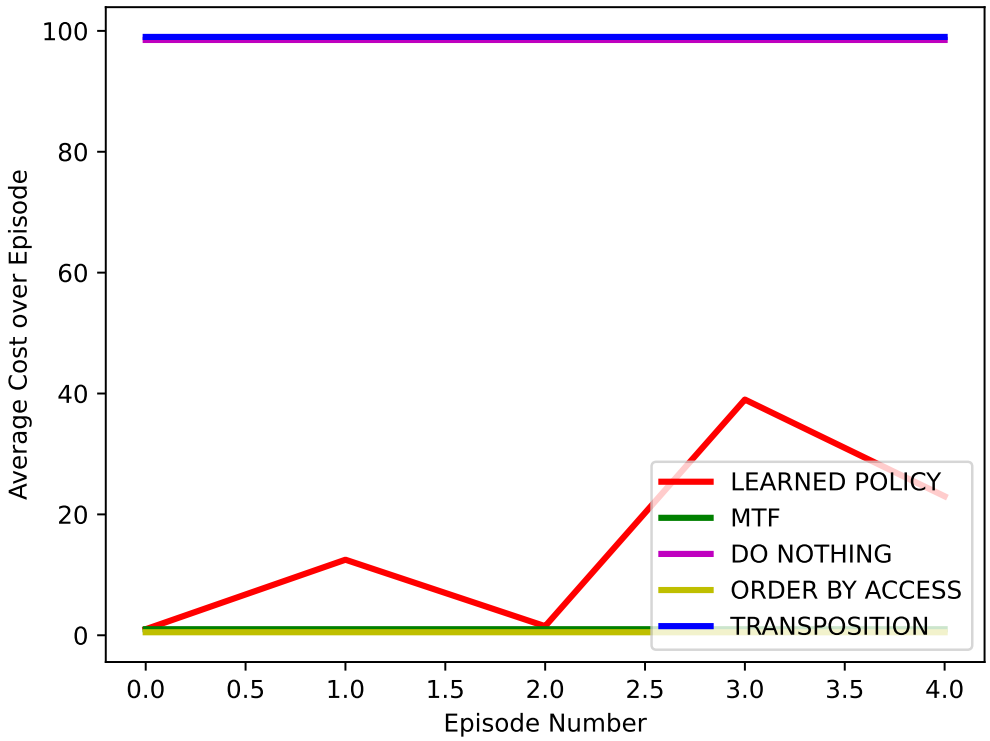

Figure 4-32: Results for Adversarial query sequence of length 1000 on list of size 100. No list change, no distribution change episode-to-episode variation.

Chapter 5

Understanding Reinforcement Learning Agent Policy

In Chapter 4, the evaluation of the learned algorithm showed that it performed well on each of our three criteria: **competitive performance across different query sequences**, **the ability to adapt to changing query sequences** and **consistent performance at scale**. We now attempt to understand the behaviour of the learned algorithm and analyze why it has a lower average search cost. The graphs and figures we use in this chapter are for lists of size 20 and 10. This is to allow for easy visualization.

5.1 Gleaning Behavior from Policy Maps

We start by studying the policy maps. Recall from Chapter 4, that policy maps show the frequency of move choices made by an algorithm. As an example, we can see the policy map for the Do-nothing algorithm in Figure 5-1. Since it never

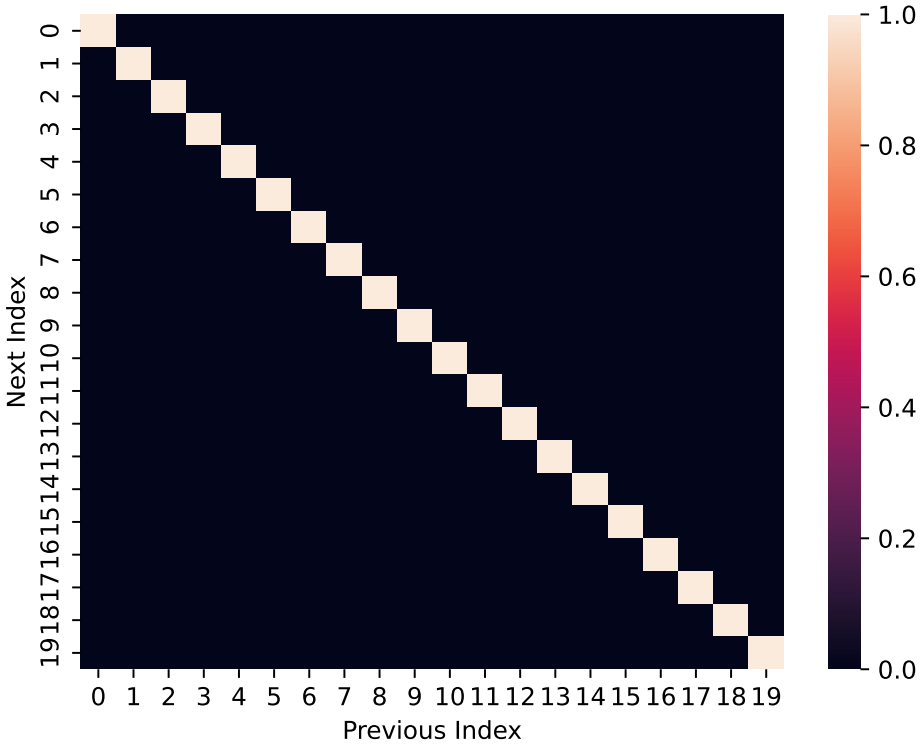

Figure 5-1: Policy Map for Do Nothing Algorithm.

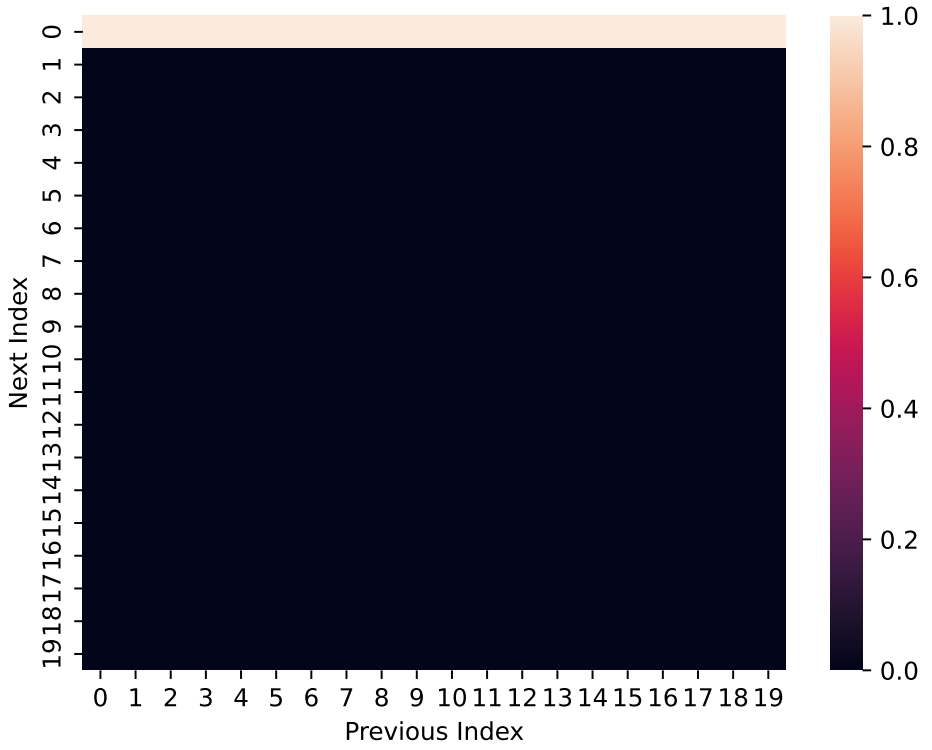

Figure 5-2: Policy Map for Move-to-Front Algorithm.

moves records, only move choices (x_1, x_2) such that $x_1 = x_2$ have non-zero frequency in the policy map. Contrast this with the policy map for move-to-front shown in Figure 5-2. In move-to-front, records are moved from their position to the front of the list, therefore only move choices (x_1, x_2) where $x_2 = 0$ have non-zero frequency in the policy map.

Let us consider the scenario when the list and the distribution remain static from episode to episode for a Zipfian query sequence. Figures 5-3, 5-4, 5-5, 5-6 and 5-7 show snapshots of the policy map over a single episode.

Each snapshot, in the order listed, represents the first 20% of accesses, the sec-

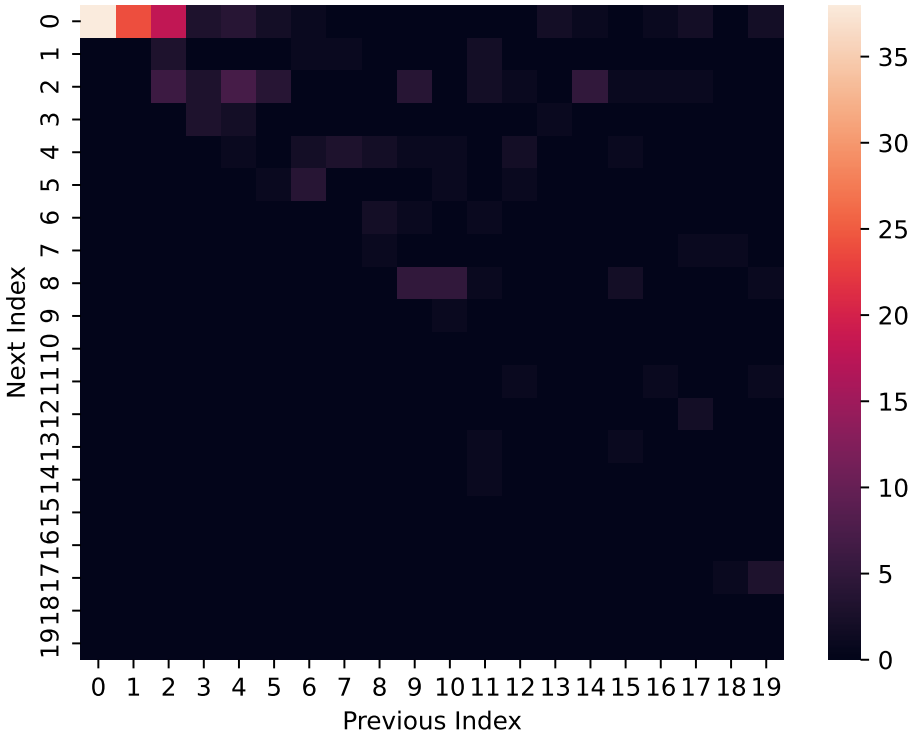

Figure 5-3: Policy Map for Learned Algorithm. Shows move choices in the first 20% of accesses in a list of size 20. Zipfian Query Sequence, No list change, No distribution change

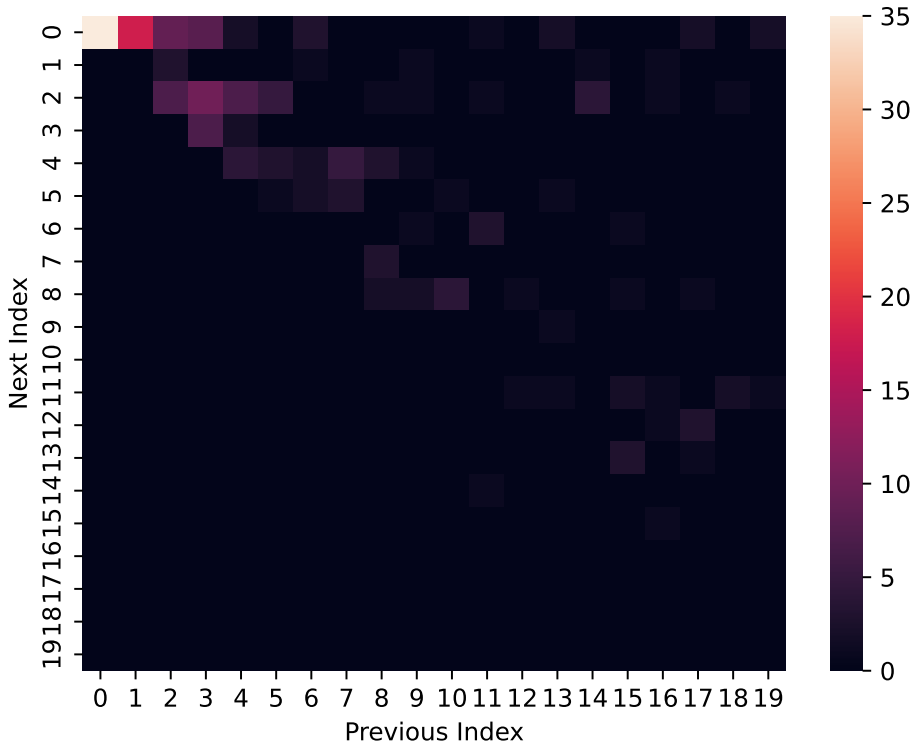

Figure 5-4: Policy Map for Learned Algorithm. Shows move choices in the second 20% of accesses in a list of size 20. Zipfian Query Sequence, No list change, No distribution change

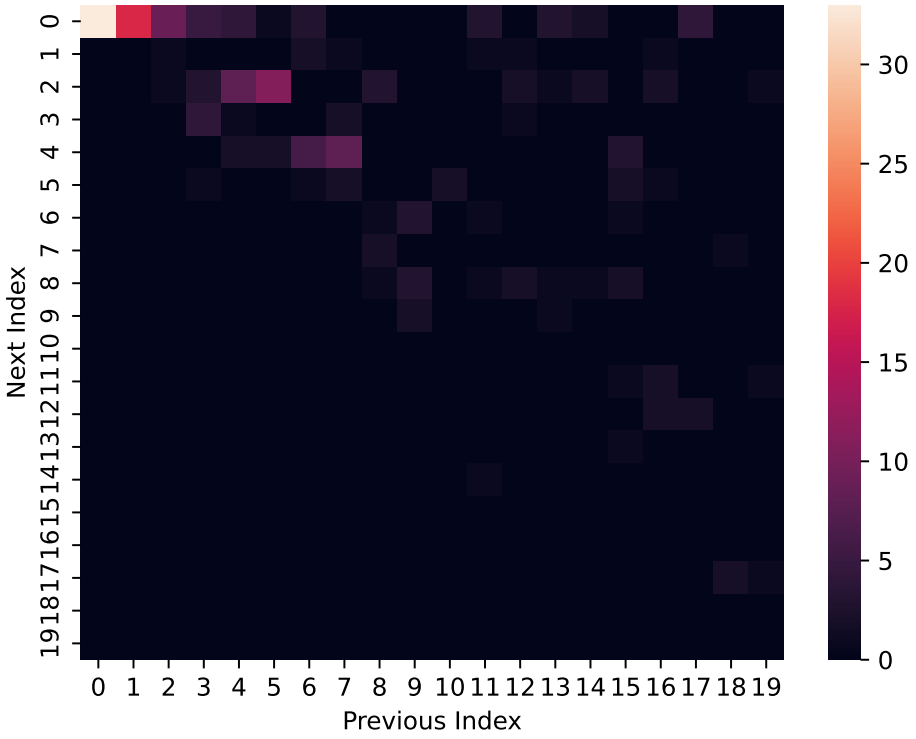

Figure 5-5: Policy Map for Learned Algorithm. Shows move choices in the third 20% of accesses in a list of size 20. Zipfian Query Sequence, No list change, No distribution change

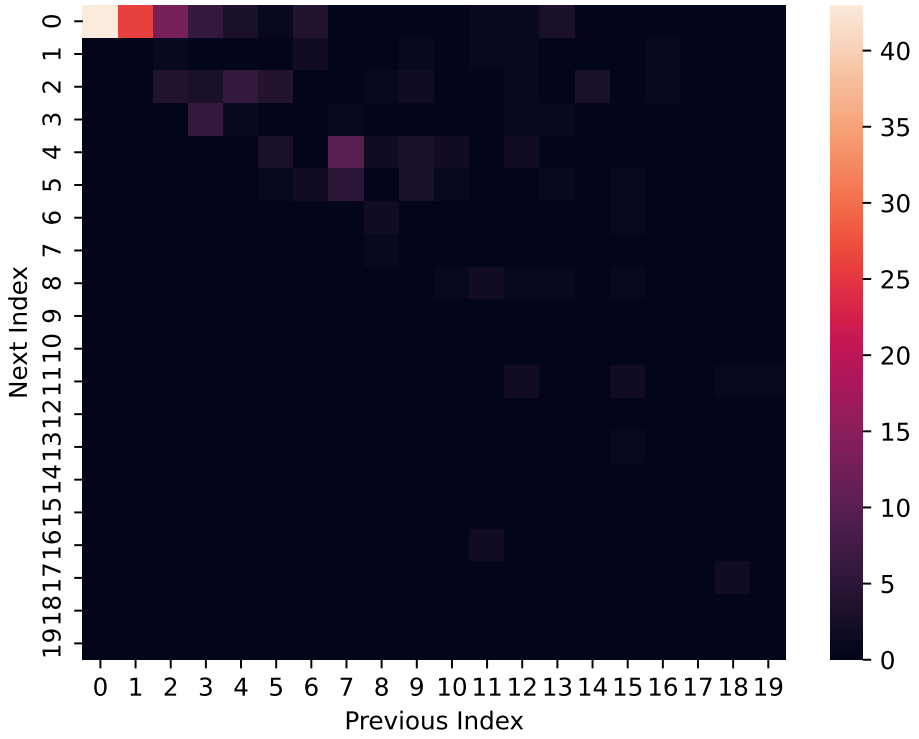

Figure 5-6: Policy Map for Learned Algorithm. Shows move choices in the fourth 20% of accesses in a list of size 20. Zipfian Query Sequence, No list change, No distribution change

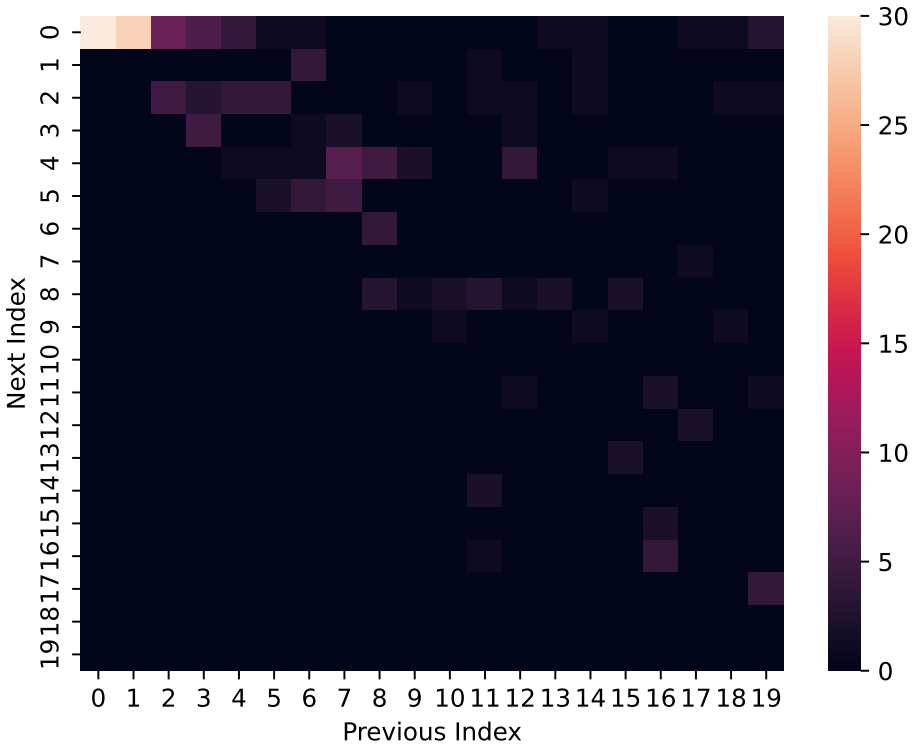

Figure 5-7: Policy Map for Learned Algorithm. Shows move choices in the last 20% of accesses in a list of size 20. Zipfian Query Sequence, No list change, No distribution change

ond 20% of accesses, etc. In the first policy map, the list has not yet reached a stationary distribution so we see that the move choices of the algorithm does not have a specific pattern yet. However, as we look to the fourth (Figure 5-6) and fifth (Figure 5-7) snapshots, we see that records found in position $0-5$ when accessed are moved to the front. When records in position $6-12$, are accessed, the farthest they are ever moved is to the 4th position. For all other records, they are either kept in the same position or moved a random number of steps forward. Based on these policy maps, we propose the following as the learned algorithm's behavior:

- Decide on a number of buckets, k, for grouping records. Denote the buckets as $b_1, ..., b_k$ starting from the front of the list to the back. For a bucket b_i, we call i the **index** of the bucket.

- Depending on the frequency of access of each record, put it in one of these buckets. More frequently accessed records are placed in buckets with lower indices and less frequently accessed buckets are placed in buckets with higher indices.

- Treat each bucket as its own small list with its own policy. Maintain buckets with lower indices according to a Move-to-Front policy. For buckets appearing later in the list, adopt a policy we call **randomized move-to-front**. In randomized move-to-front, if the record is in position i, randomly choose a position between the front of the list and i to move the record to.

We call this behavior **banded** or **bucketed** policy. In the concluding remarks of Chapter 4, we discussed the time until convergence of different algorithms. Since the learned algorithm, much like transposition, conservatively moves records towards the front of the list, it is likely that it will converge to a stationary distribu-

tion slower than move-to-front.

5.2 Verifying the Learned Algorithm's Behaviour

To verify the learned algorithm's proposed behaviour in Section 5.1, we look at four things: policy maps of the learned algorithm for Heavy/Light query sequences, policy maps of the learned algorithm for Uniform query sequences, policy maps for the different episode-to-episode variations and transition graphs.

5.2.1 Policy Map for Heavy/Light Distribution

Figures 5-8, 5-9, 5-10, 5-11 and 5-12 show snapshots of the policy map for a Heavy/Light distribution where 10% of the records are heavy and heavy records make up 90% of the query sequence. The learned algorithm's behaviour is the same as described in 5.1 above. To better appreciate the move choices, we also show normalized versions of each policy map in Figures 5-13, 5-14, 5-15, 5-16 and 5-17 Focusing on Figures 5-16 and 5-17 where the algorithm converges, it can be seen that only two records are ever moved to the front of the list since those two records are accessed 90% of the time. All other records are never moved to the front. They are only ever moved to some position between position 2 and their current position.
In this case, the learned algorithm chooses 2 buckets for grouping records. The first half follows move-to-front and the second half follows randomized move-to-front. The policy map for the Adversarial query sequence (Figures 5-18, 5-19, 5-20, 5-21 and 5-22) is also the same. Only two records are ever accessed in that case and those two records are kept in the first bucket at the very front of the list. In that first bucket, the move-to-front policy is maintained and so the only move

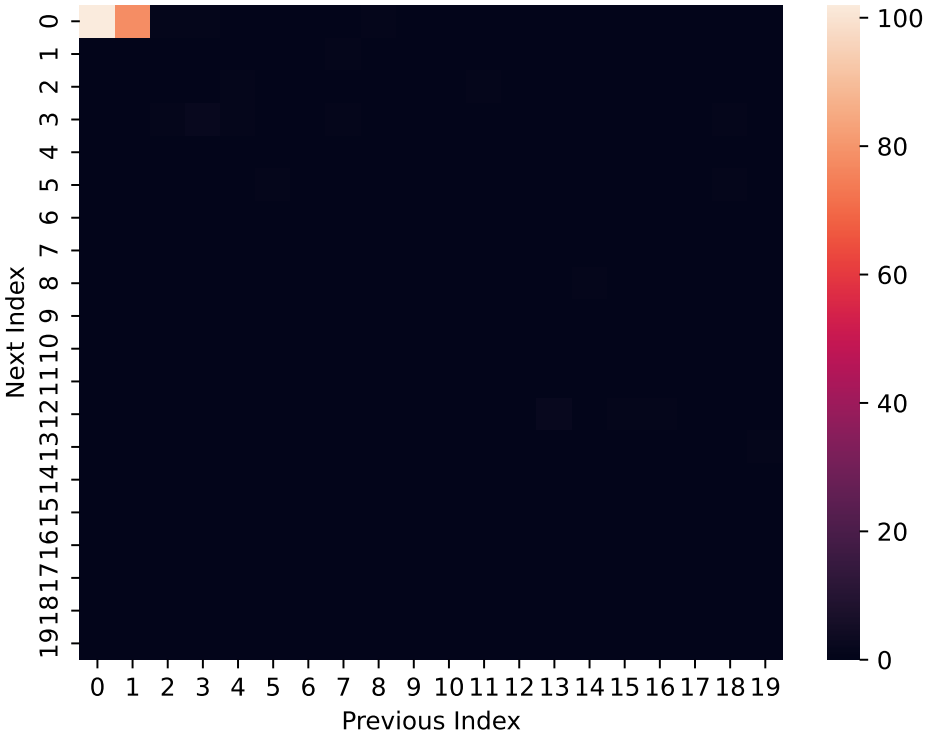

Figure 5-8: Policy Map for Learned Algorithm. Shows move choices in the first 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

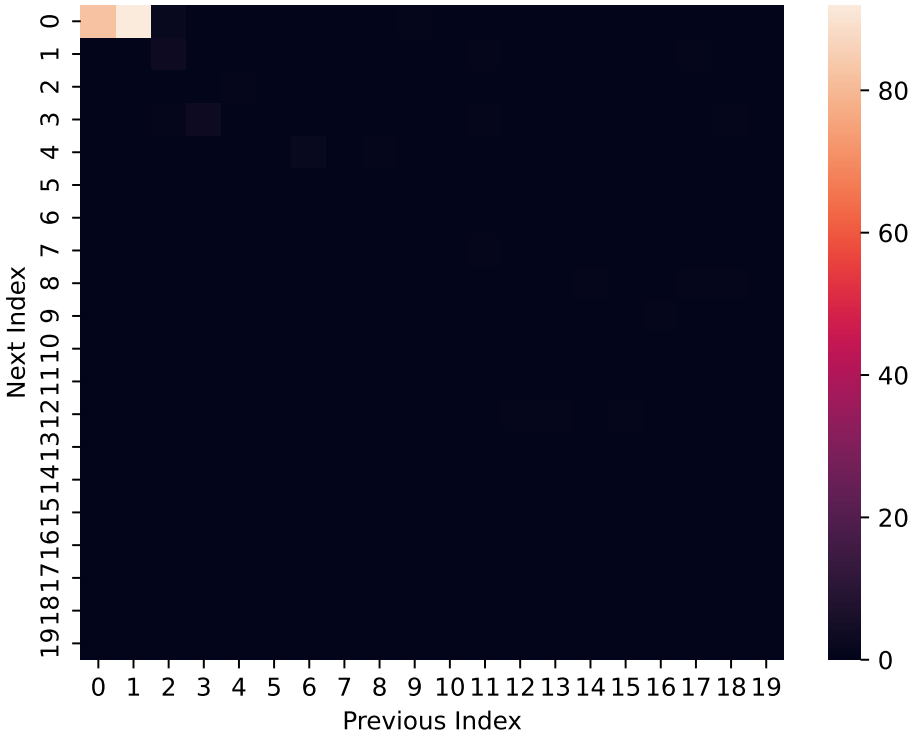

Figure 5-9: Policy Map for Learned Algorithm. Shows move choices in the second 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

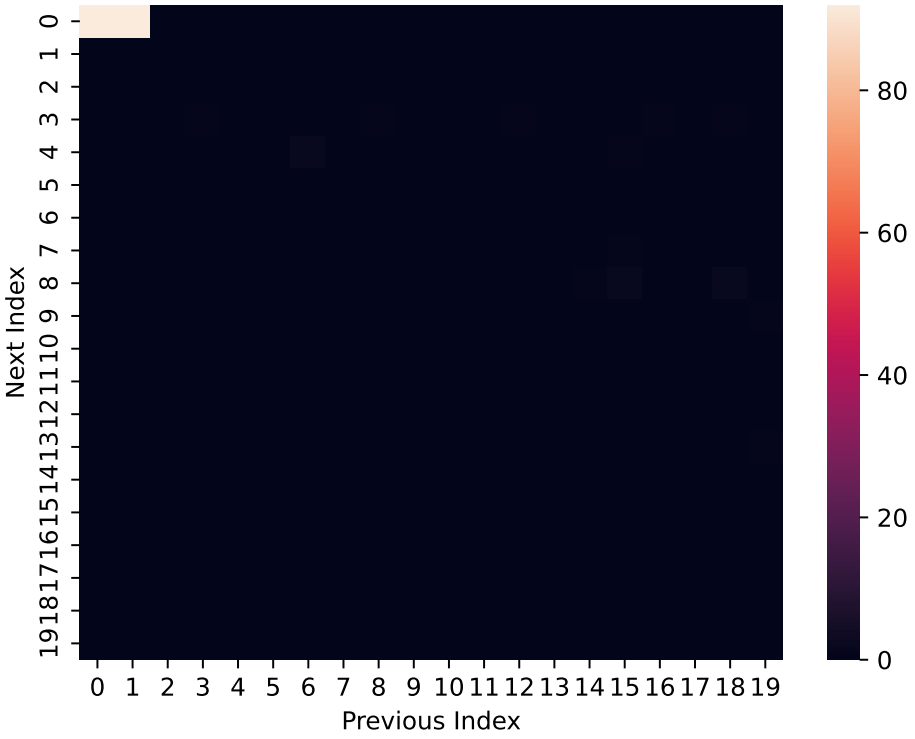

Figure 5-10: Policy Map for Learned Algorithm. Shows move choices in the third 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

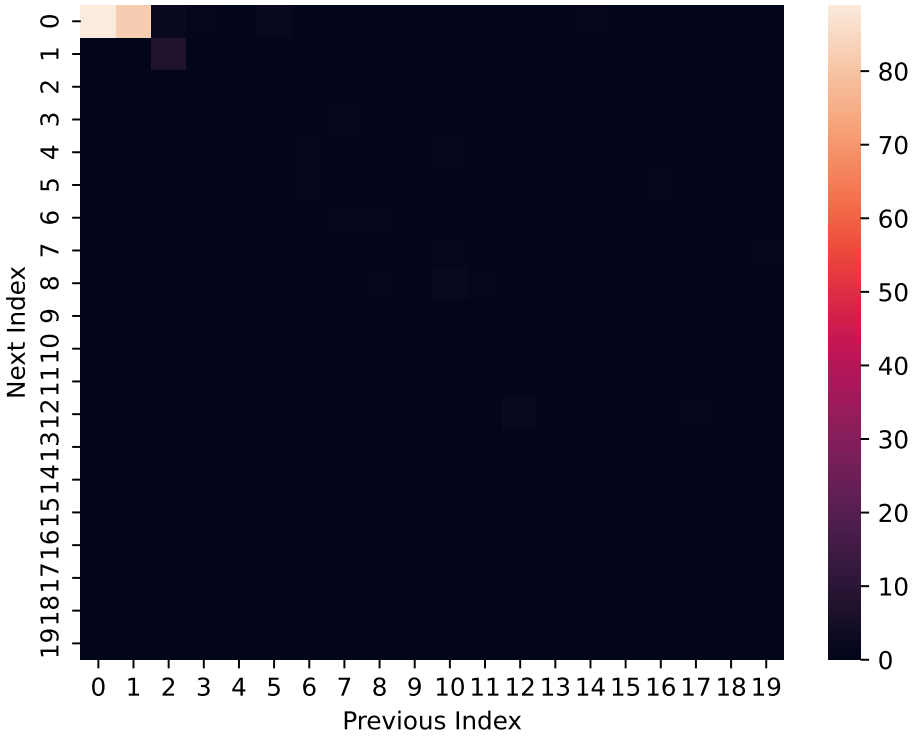

Figure 5-11: Policy Map for Learned Algorithm. Shows move choices in the fourth 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

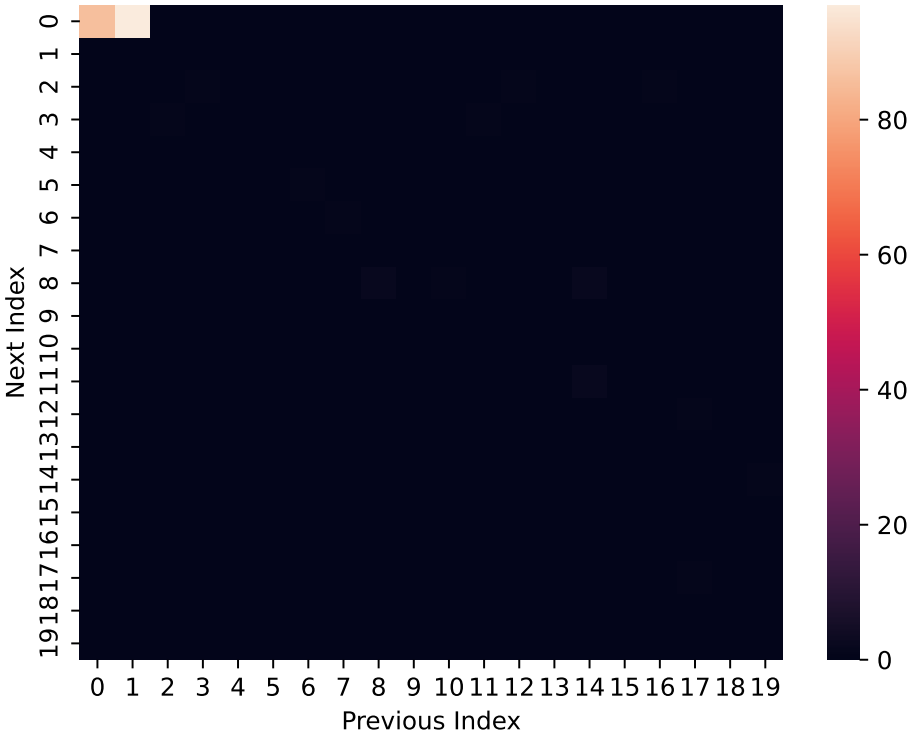

Figure 5-12: Policy Map for Learned Algorithm. Shows move choices in the last 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

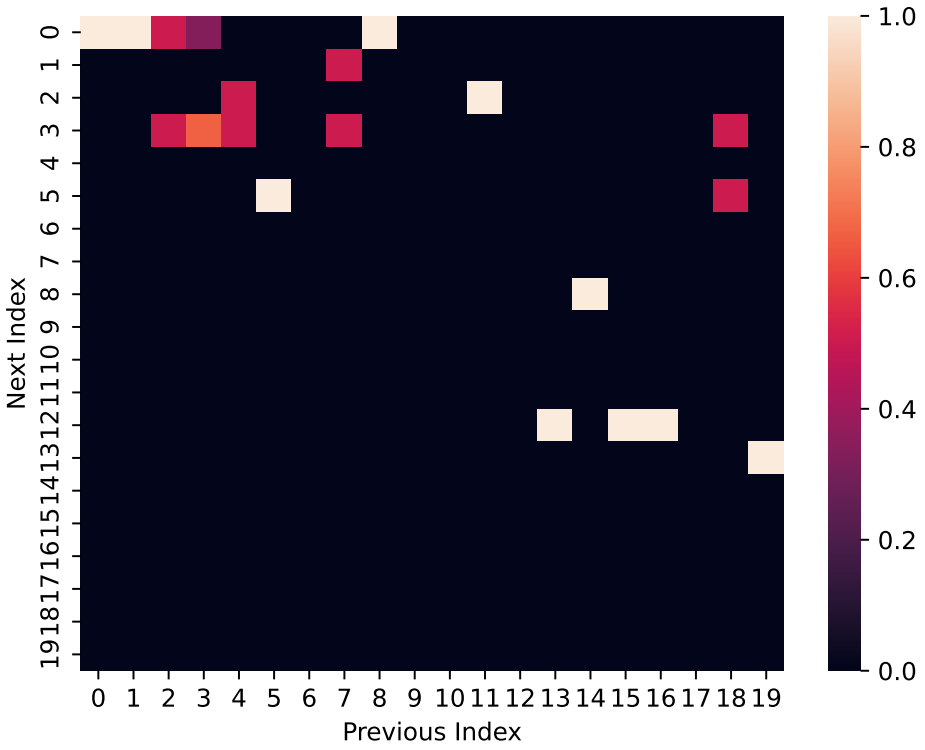

Figure 5-13: Normalized Policy Map for Learned Algorithm. Shows move choices in the first 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

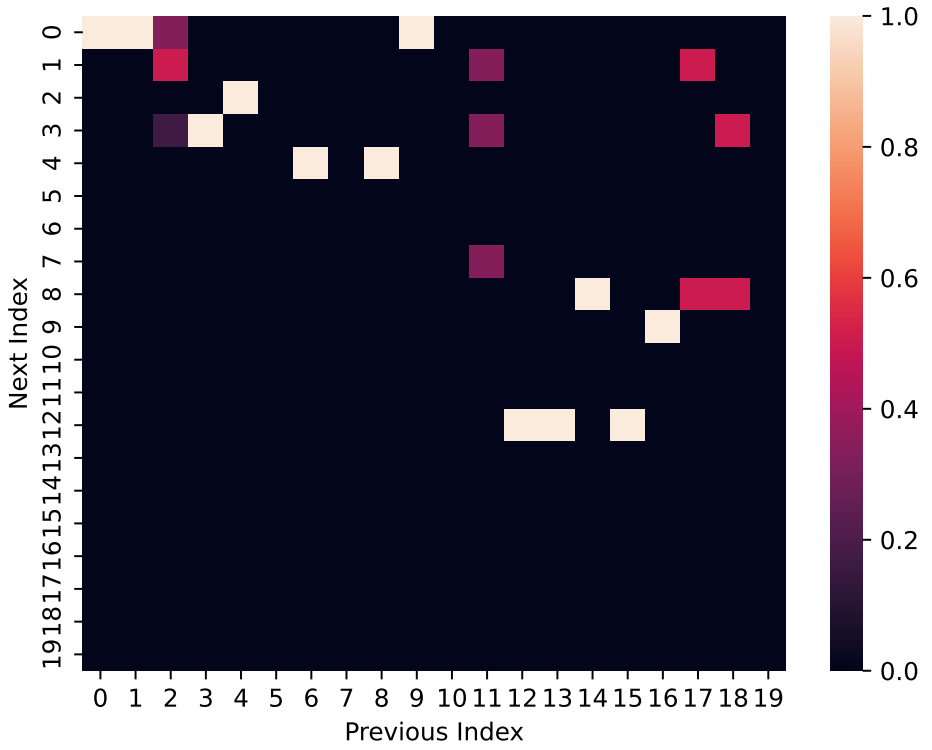

Figure 5-14: Normalized Policy Map for Learned Algorithm. Shows move choices in the second 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

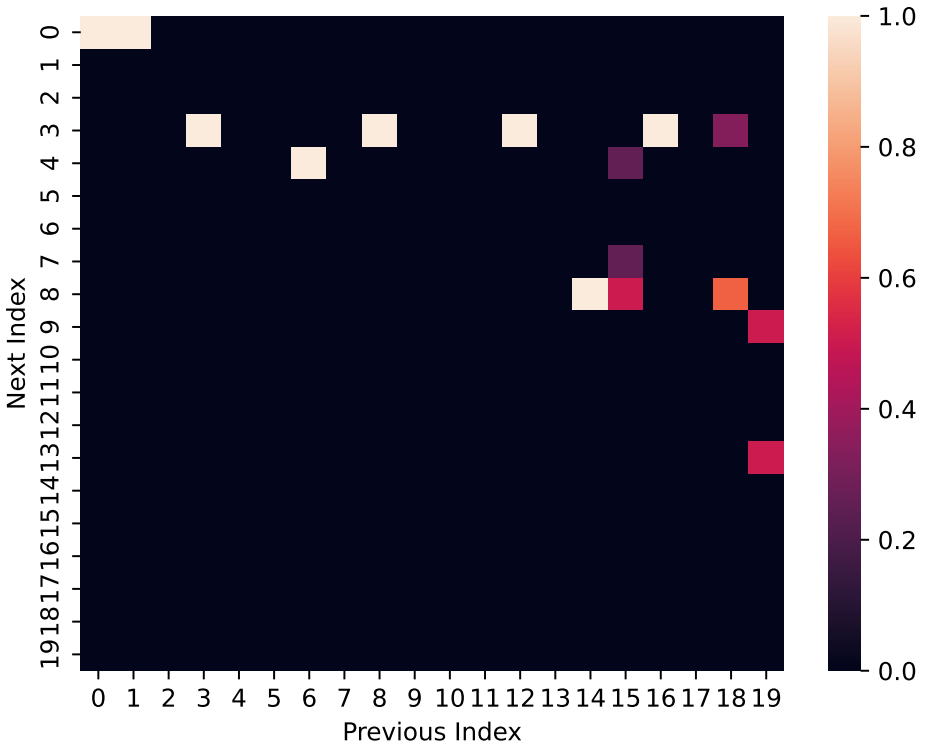

Figure 5-15: Normalized Policy Map for Learned Algorithm. Shows move choices in the third 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

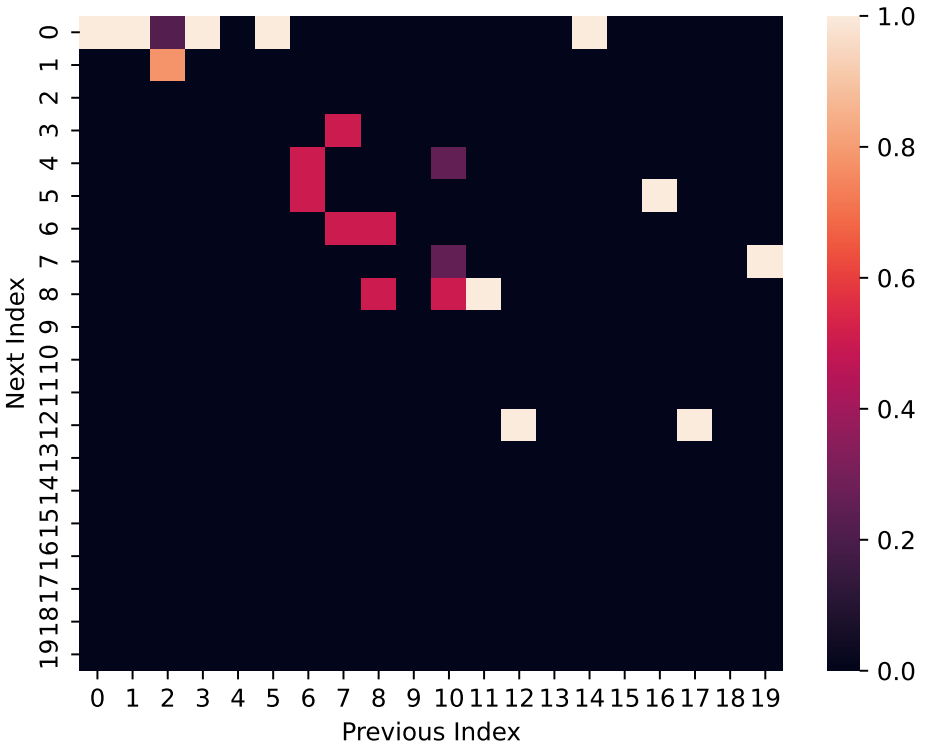

Figure 5-16: Normalized Policy Map for Learned Algorithm. Shows move choices in the fourth 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

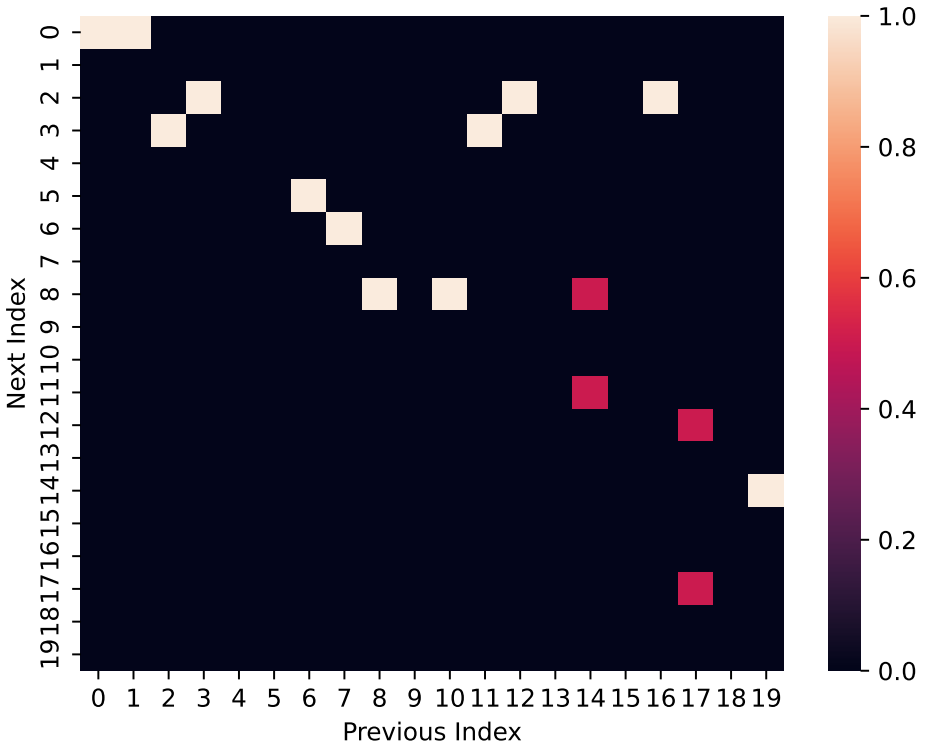

Figure 5-17: Normalized Policy Map for Learned Algorithm. Shows move choices in the last 20% of accesses in a list of size 20. Heavy/Light Query Sequence with 20% heavy, 90% heavy accessed, No list change, No distribution change

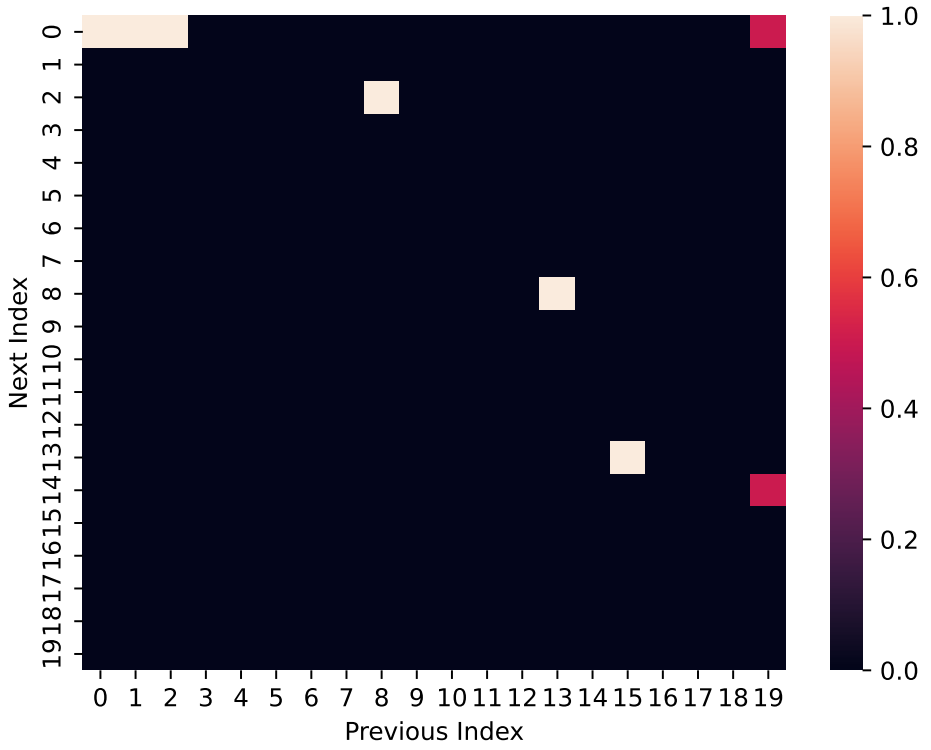

Figure 5-18: Normalized Policy Map for Learned Algorithm. Shows move choices in the first 20% of accesses in a list of size 20. Adversarial Query Sequence, No list change, No distribution change

choice is (1,0) at the stationary distribution (Figures 5-21 and 5-22).

This bucketing behaviour policy is an indication that the reinforcement learning agent discovers which items are heavy and places them closer to the front of the list.

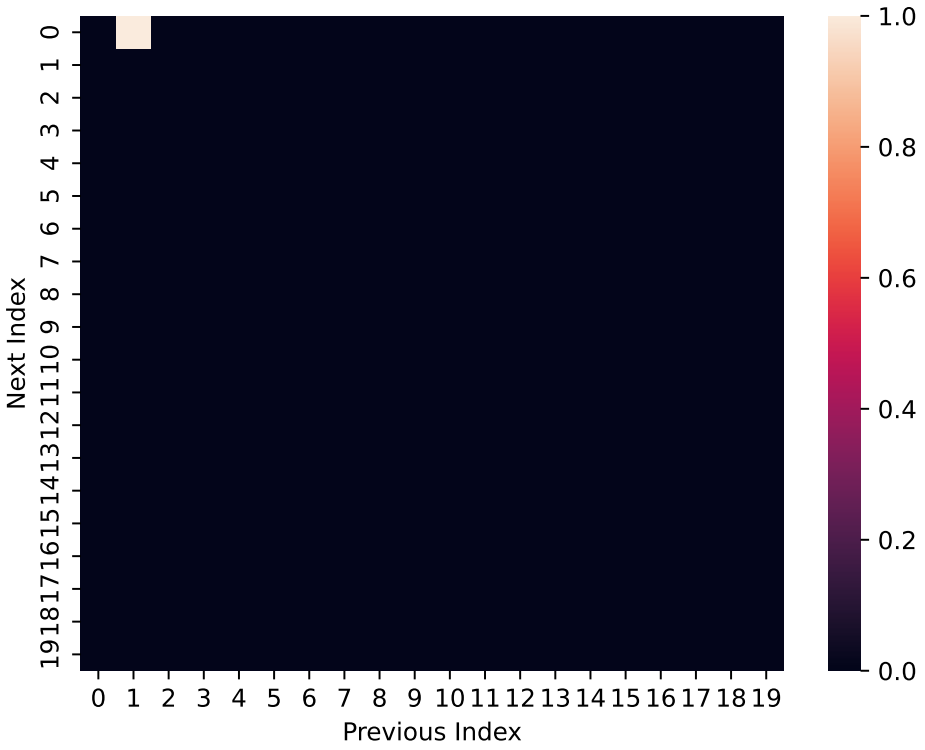

Figure 5-19: Normalized Policy Map for Learned Algorithm. Shows move choices in the second 20% of accesses in a list of size 20. Adversarial Query Sequence, No list change, No distribution change

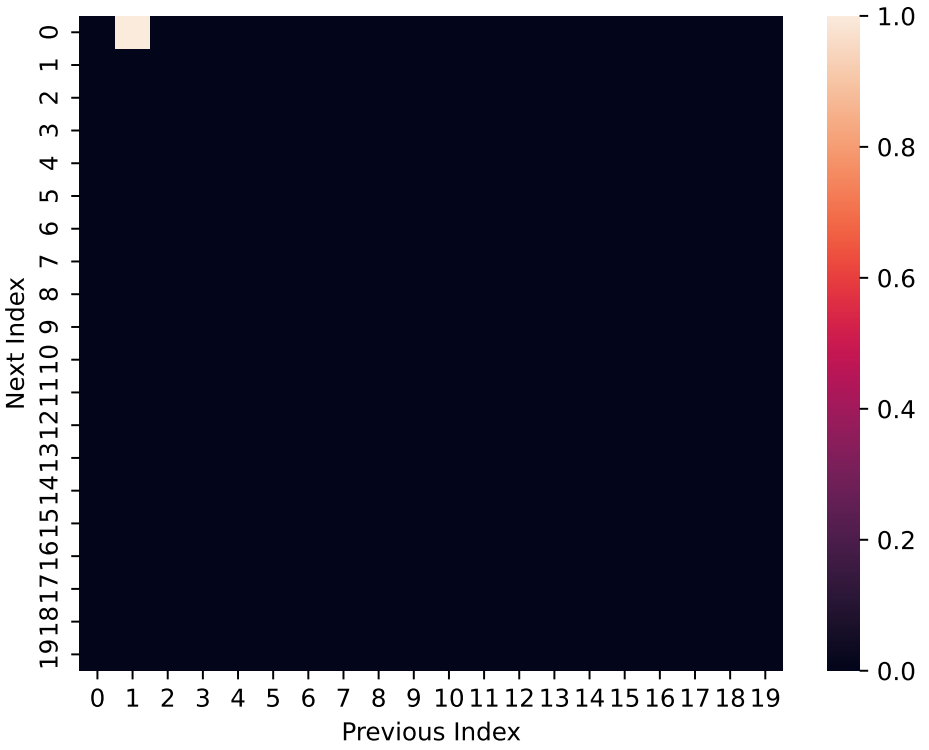

Figure 5-20: Normalized Policy Map for Learned Algorithm. Shows move choices in the third 20% of accesses in a list of size 20. Adversarial Query Sequence, No list change, No distribution change

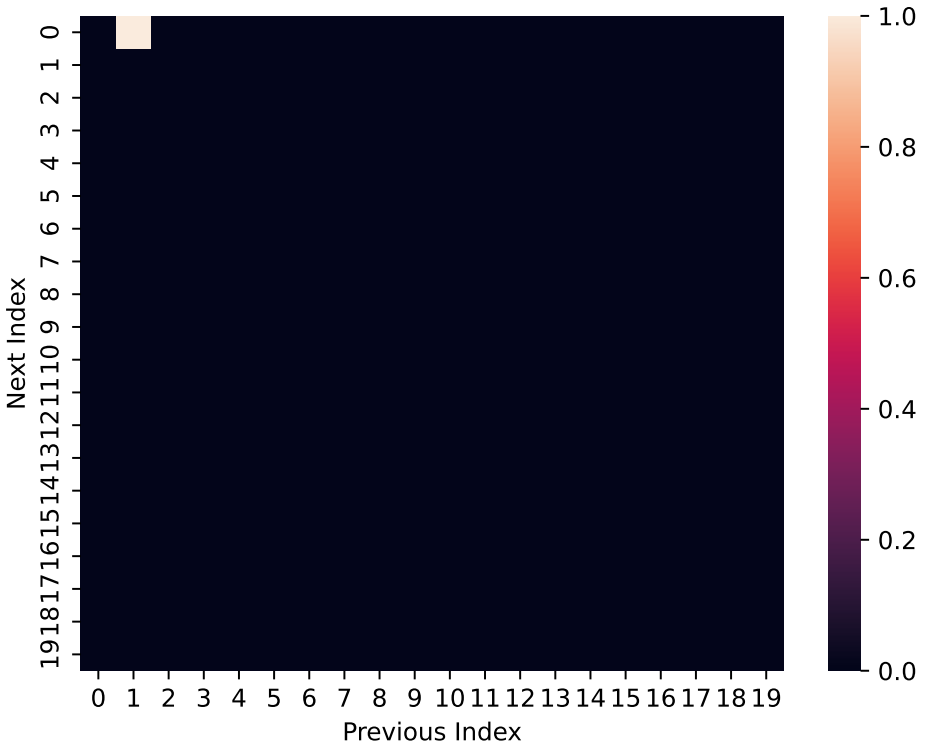

Figure 5-21: Normalized Policy Map for Learned Algorithm. Shows move choices in the fourth 20% of accesses in a list of size 20. Adversarial Query Sequence, No list change, No distribution change

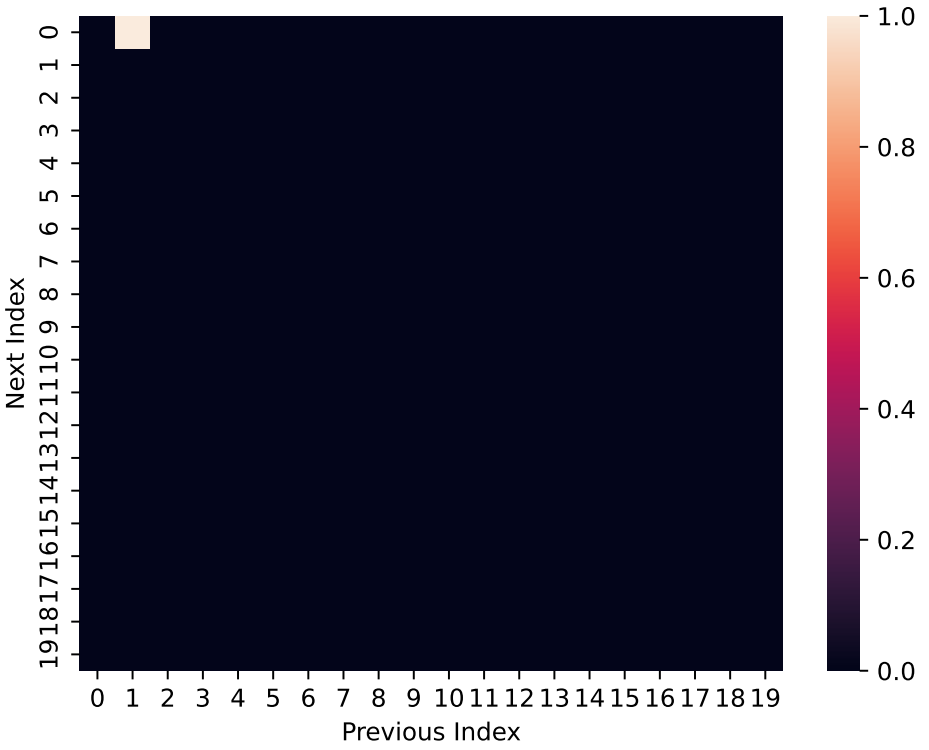

Figure 5-22: Normalized Policy Map for Learned Algorithm. Shows move choices in the last 20% of accesses in a list of size 20. Adversarial Query Sequence, No list change, No distribution change

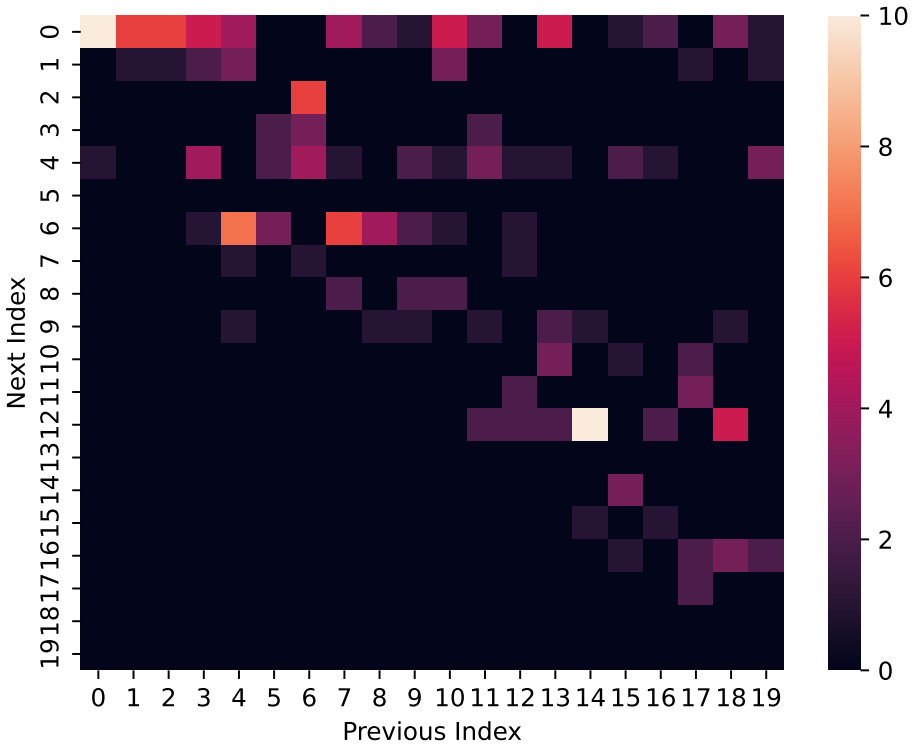

Figure 5-23: Policy Map for Learned Algorithm. Shows move choices in the first 20% of accesses in a list of size 20. Uniform Query Sequence, No list change, No distribution change

5.2.2 Policy Map for Uniform query sequence

Previously, we studied the policy map for non-uniform query sequences(Zipfian, Heavy/Light and Adversarial). We now try to decode the algorithm's behaviour when records are accessed uniformly at random. Per our description of the learned algorithm in Section 5.1, it should treat the entire list as a single bucket and follow a randomized move-to-front policy. In Figures 5-23, 5-24, 5-25, 5-26 and 5-27 this is indeed the case. For a record queried and found in position x_1, the algorithm

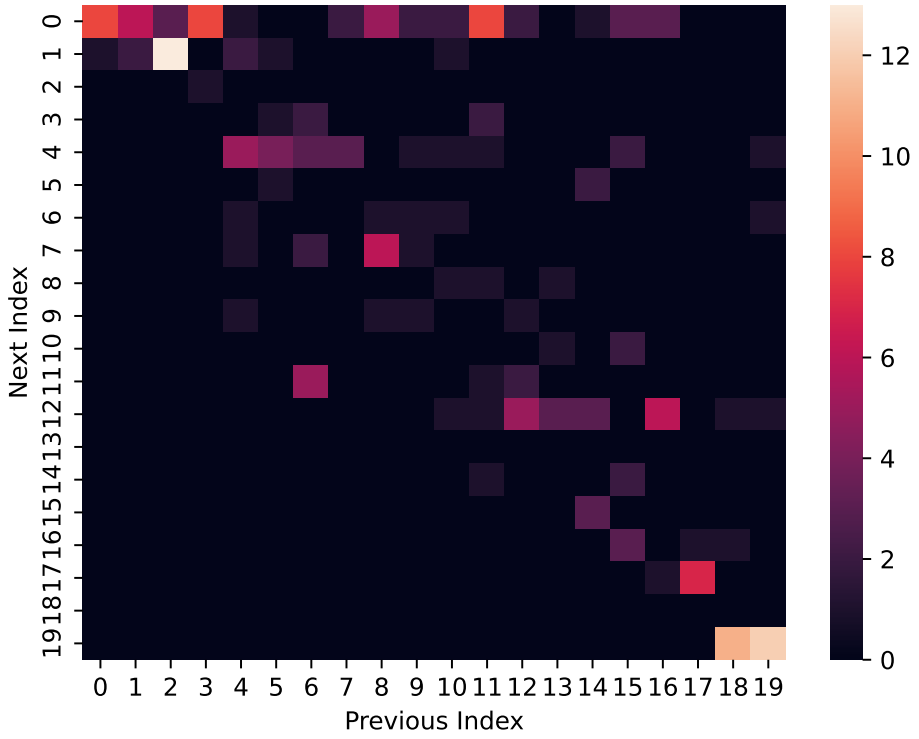

Figure 5-24: Policy Map for Learned Algorithm. Shows move choices in the second 20% of accesses in a list of size 20. Uniform Query Sequence, No list change, No distribution change

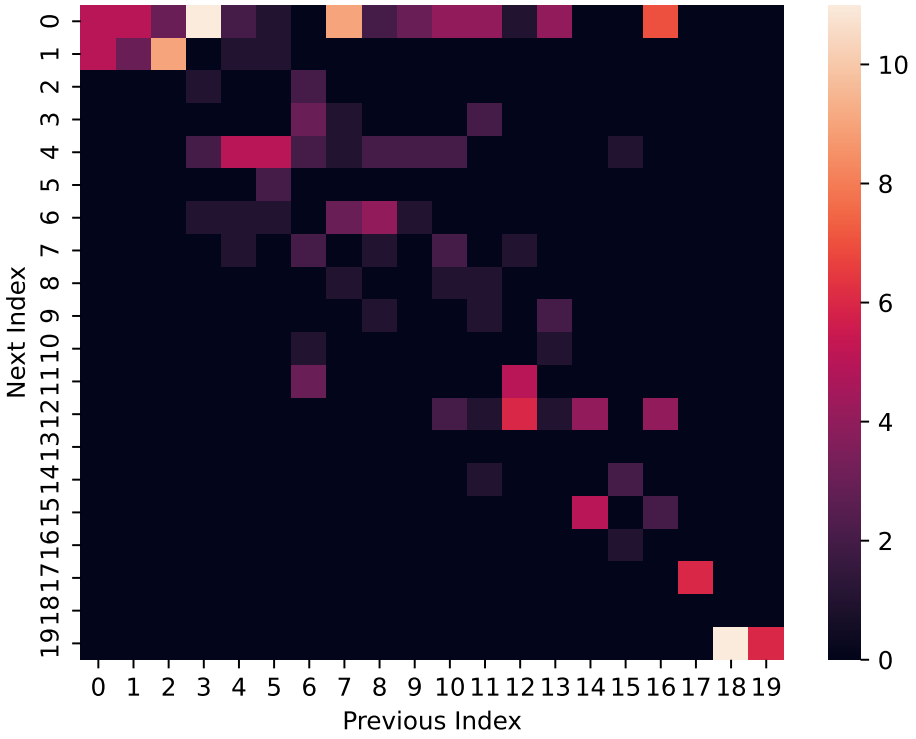

Figure 5-25: Policy Map for Learned Algorithm. Shows move choices in the third 20% of accesses in a list of size 20.Uniform Query Sequence, No list change, No distribution change

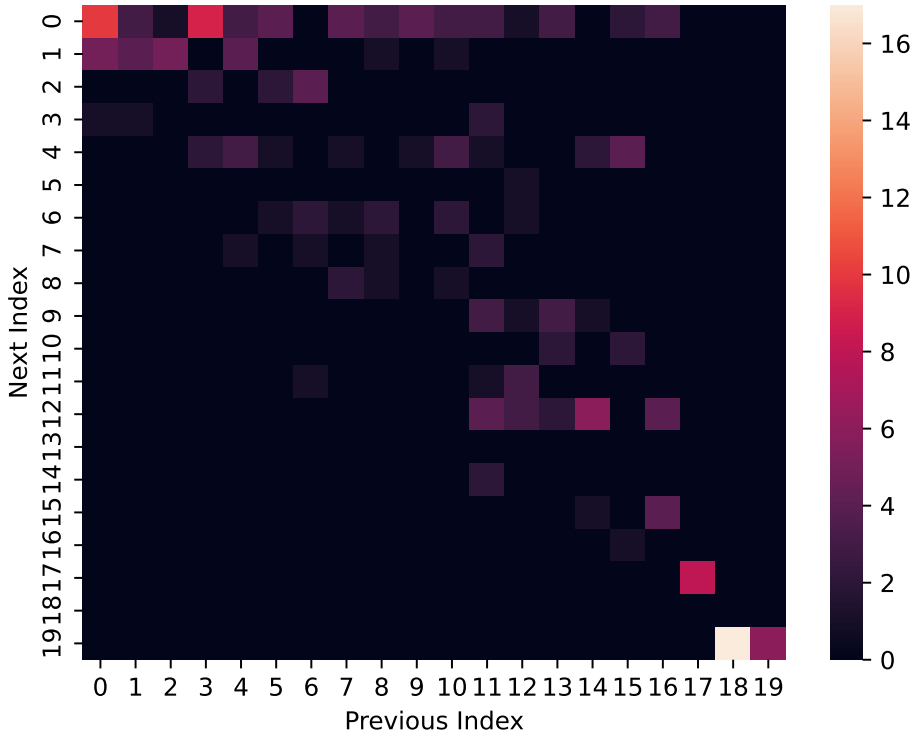

Figure 5-26: Policy Map for Learned Algorithm. Shows move choices in the fourth 20% of accesses in a list of size 20.Uniform Query Sequence, No list change, No distribution change

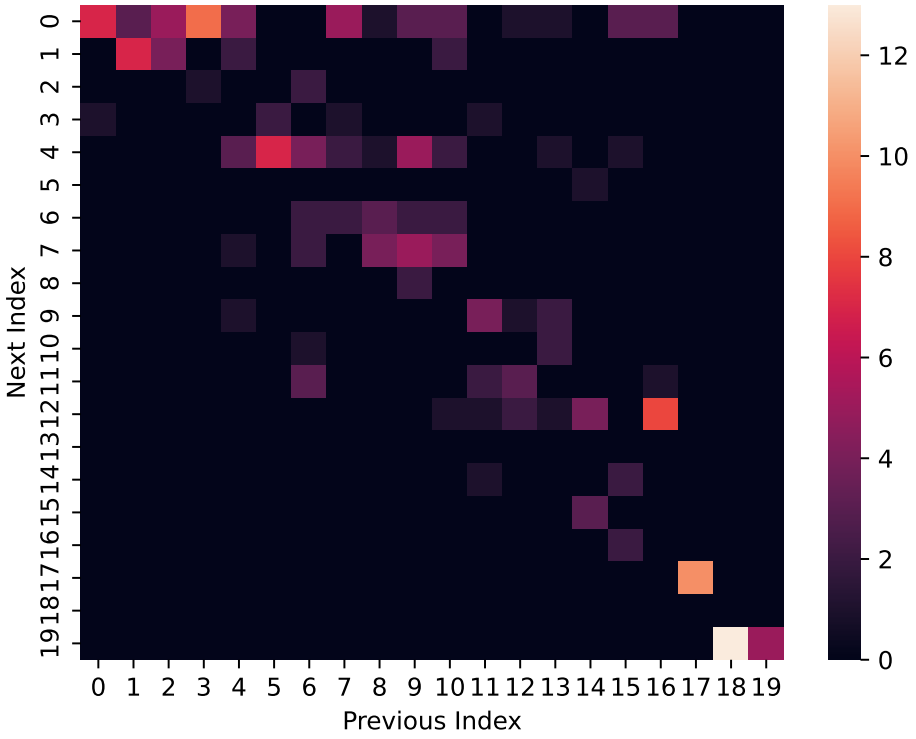

Figure 5-27: Policy Map for Learned Algorithm. Shows move choices in the last 20% of accesses in a list of size 20.Uniform Query Sequence, No list change, No distribution change

chooses some random position between 0 and x_1 inclusive to place the record.

5.2.3 Policy Map for Episode-to-Episode Variation

While evaluating the performance of the learned algorithm, we saw that despite episode-to-episode variations, the learned algorithm remained competitive. Now, we verify that the behavior of the algorithm is also consistent for different episode-to-episode variations. We show the policy map over an entire episode of Zipfian query sequences for each episode-to-episode variant in Figures 5-28, 5-29 and 5-30.

5.2.4 Transition Graphs

In Chapter 4, we described transition graphs as weighted directed graphs with positions/indices as vertices, move choices as edges and move choice frequencies as edge weights. So a move choice (x_1, x_2) will be shown as an arrow from x_1 to x_2. Self-loops indicate that the item remained in that position. In Figure 5-31, we see an example of the transition graph for the do-nothing algorithm. It contains only self-loops because records are not moved in the list. To contrast, we also show the transition graph for move-to-front in Figure 5-32. All edges terminate at the vertex 0 because in Move-to-Front, all records are moved to the front on each access. In Figure 5-33, the transition graph for the learned algorithm on a list of size 10 with Zipfian query sequence is shown. Suppose we remove the edges of weight 1. This will split the graph into three weakly connected components. One component consists of $0, 1, 2$, the second consists of $3, 4, 5, 6, 7, 8$ and the last one consists of 9. These components correspond to the buckets of the learned algorithm mentioned in section 5.1. As the episode progresses, the membership within buckets and the

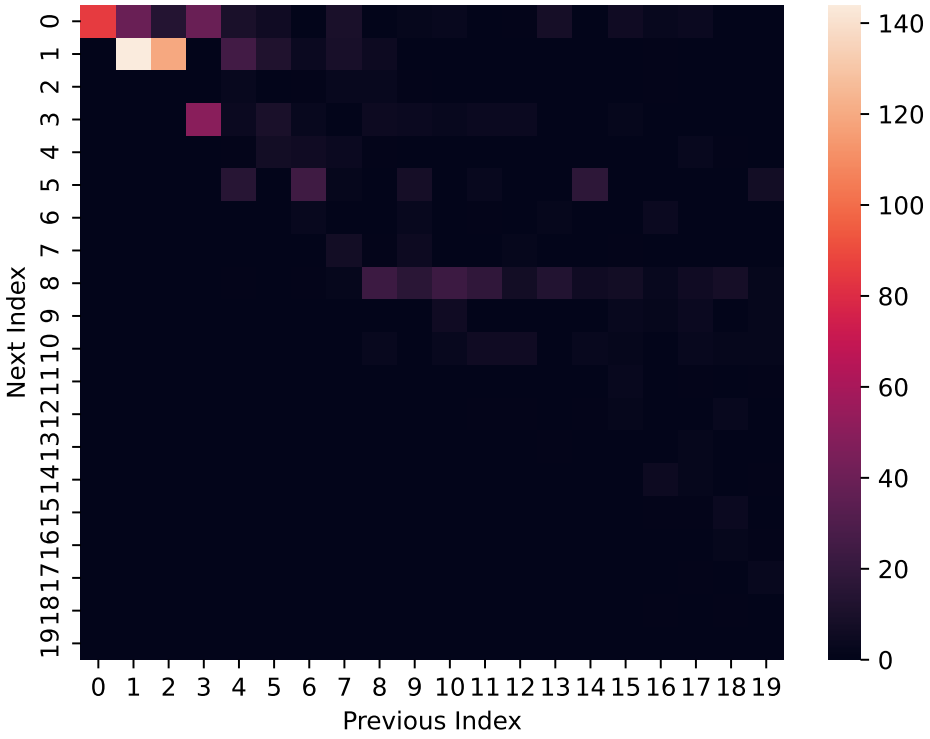

Figure 5-28: Policy Map for Learned Algorithm. Shows move choices for all accesses in a list of size 20. Zipfian Query Sequence, No list change, distribution change

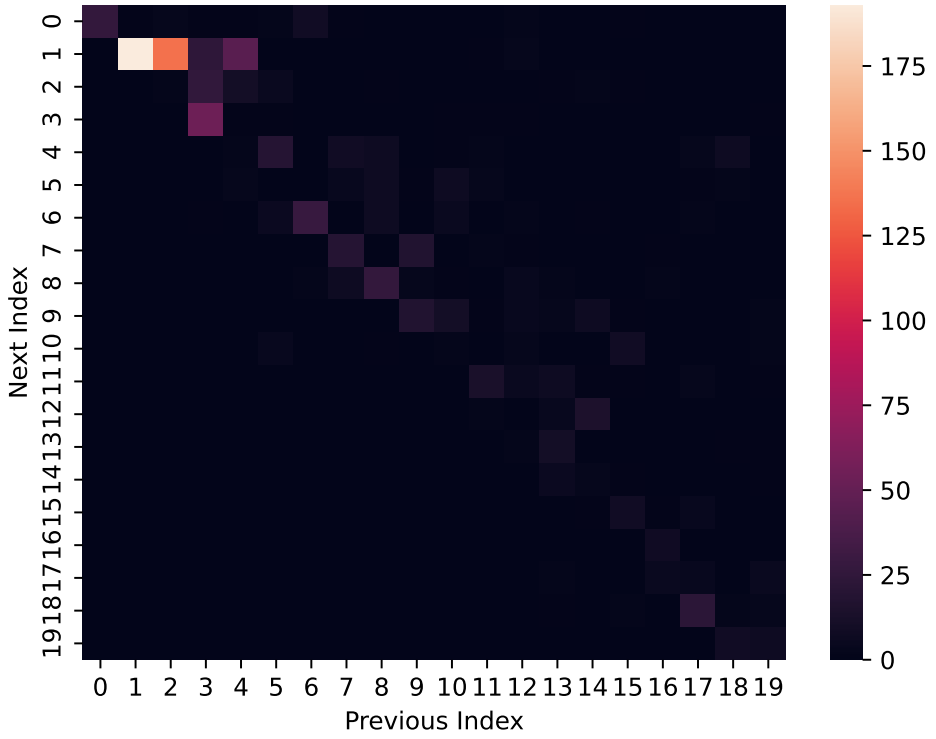

Figure 5-29: Policy Map for Learned Algorithm. Shows move choices for all accesses in a list of size 20. Zipfian Query Sequence, List change, No distribution change

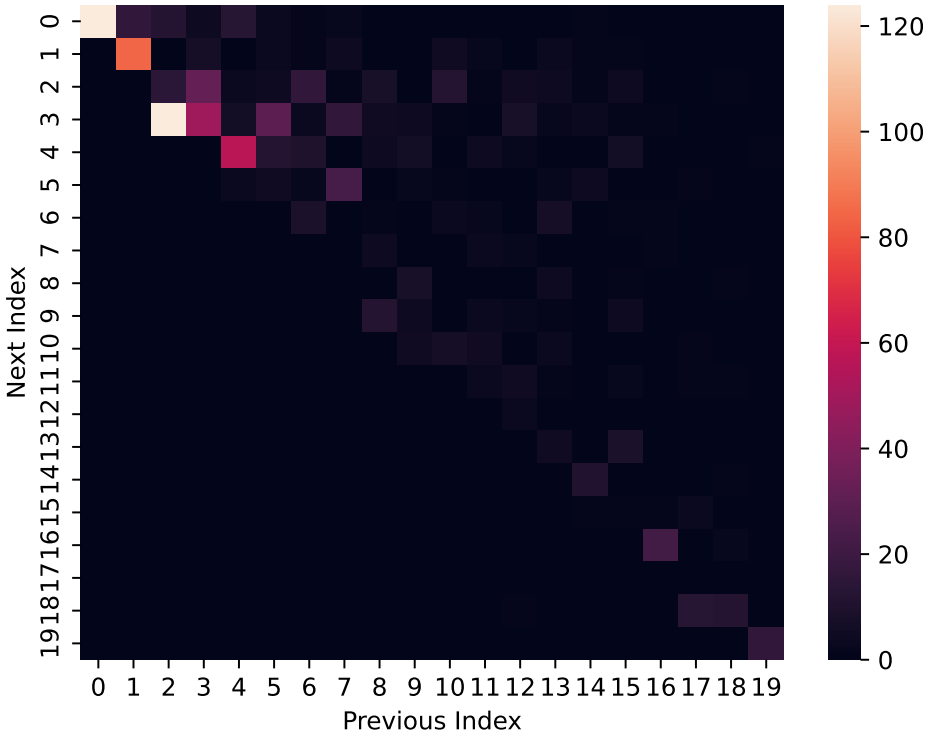

Figure 5-30: Policy Map for Learned Algorithm. Shows move choices for all accesses in a list of size 20. Zipfian Query Sequence, List change, Distribution change

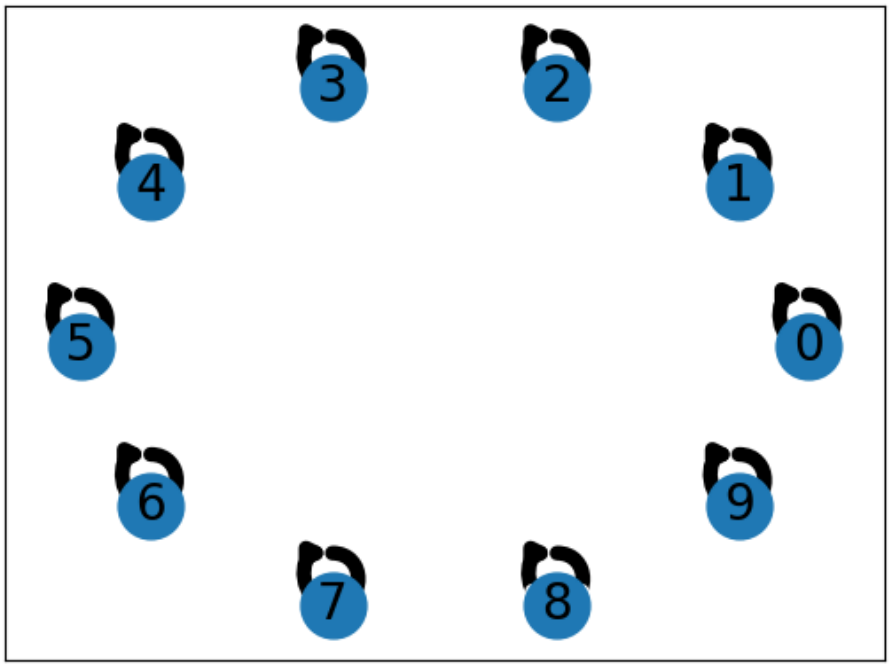

Figure 5-31: Transition Graph for Do Nothing Algorithm on list of Size 10.

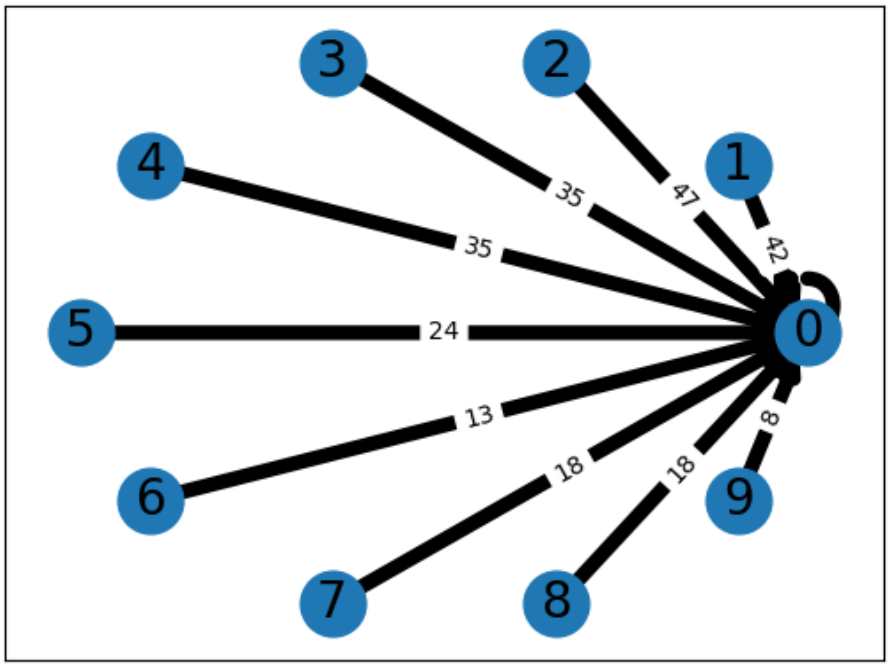

Figure 5-32: Transition Graph for Move-to-Front Algorithm on list of Size 10.

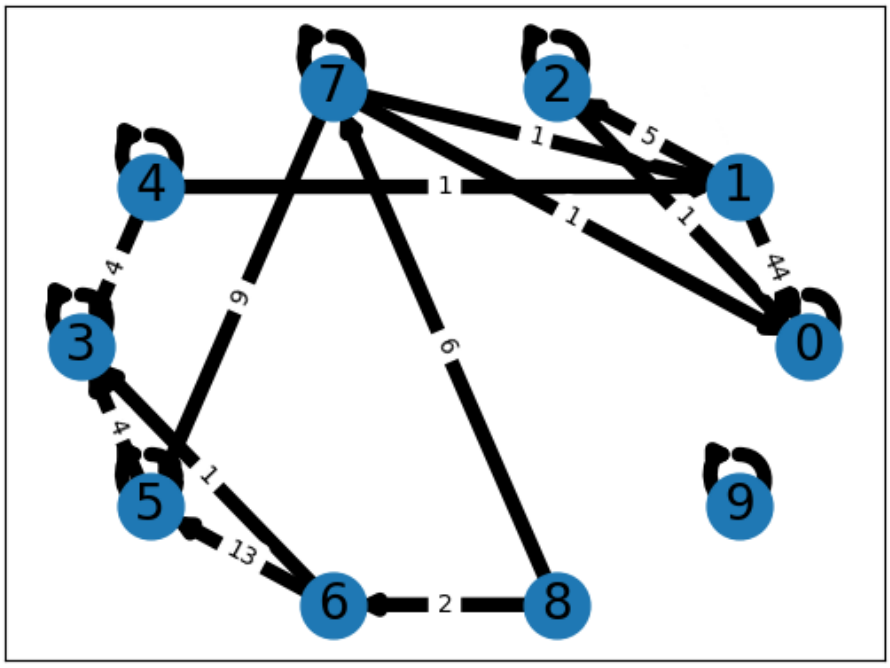

Figure 5-33: Transition Graph for Learned Algorithm on list of Size 10. Zipfian Query Sequence, No list change and No distribution change

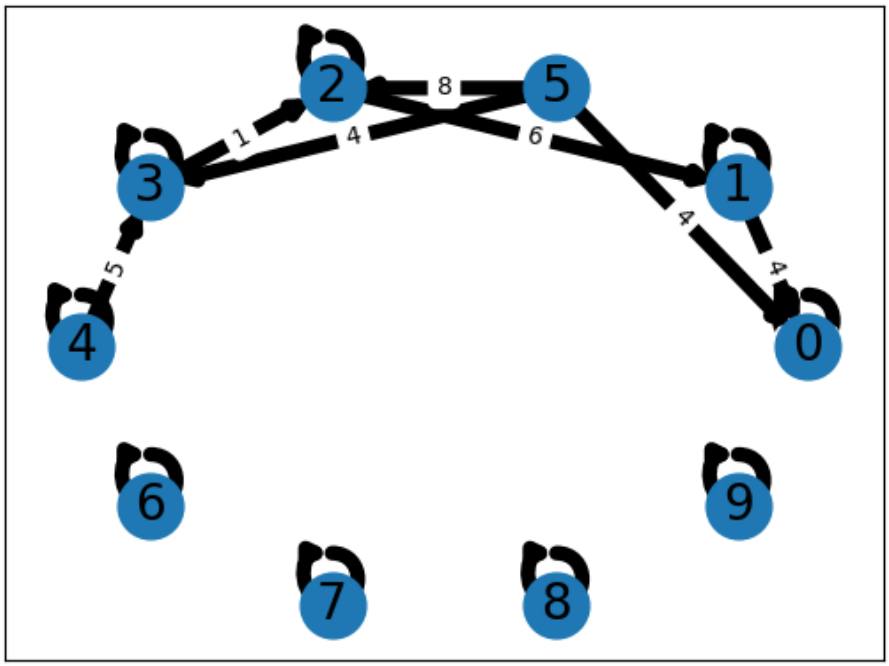

Figure 5-34: Transition Graph for Learned Algorithm on list of Size 10. Zipfian Query Sequence, No list change and No distribution change

number of buckets changes to better reflect the heaviness of records. Evidence of this can be seen in Figure 5-34 where the components are now $(0, 1, 2, 3, 4, 5)$, (6), (7), (8) and (9).

Seeing more than one component in the transition graph both confirms the bucketed nature of the learned algorithm and the fact that each bucket acts as its own small list which maintains its own policy.

5.3 Why the Learned Algorithm is Competitive

We now analyze the performance of the learned algorithm. Intuitively, the learned algorithm approximates the order-by-access algorithm by bucketing records according to their frequency of access. It places buckets with heavier records closer to the front of the list and buckets with lighter records to the back of the list. It also applies a different policy, where appropriate, within each bucket. Before we analyze its performance, we present the following definition which will also prove useful in Chapter 6.

Definition 1 *Consider a list L of n records $r_1, r_2, ..., r_n$. Let $p_1 \geq p_2 \geq ... \geq p_n$ be the access probabilities of these records. Define $b(i, j)$ to be the probability that r_i is before r_j in the stationary distribution. Then the expected search cost for record r_i is:*

$$\sum_{1 \leq j \leq n, j \neq i} b(j, i) \cdot 1$$

Taking the expectation across all records, we have that the expected search time for a record in L is:

$$\sum_{i=0}^{n} p_i \cdot \left(\sum_{1 \leq j \leq n, j \neq i} b(j, i) \cdot 1 \right)$$

For instance, consider the the simple case of the Heavy/Light distribution. The learned algorithm effectively splits L into two buckets L_L and L_R. In L_L, it performs Move-to-Front and in L_R it performs randomized move-to-front. Based on definition 1, we can decompose the summation as follows:

$$\sum_{i=0}^{n} p_i \cdot \left(\sum_{1 \leq j \leq n, j \neq i} b(j, i) \cdot 1 \right) = \sum_{a \in L_L} p_a \cdot \left(\sum_{a' \in L, a \neq a'} b(a', a) \right) + \sum_{b \in L_R} p_b \cdot \left(\sum_{b' \in L, b \neq b'} b(b', b) \right)$$

From [43] we have that for Move-to-Front, $b(j,i) = \frac{p_j}{p_i+p_j}$ so we can substitute $b(a',a) = \frac{p_{a'}}{p_{a'}+p_a}$. Because we perform randomized move-to-front in L_R and every starting permutation of L_R is equally likely, $b(j,i)$ is equal for all j,i pairs. Therefore $b(b',b) = \frac{1}{2}$. Substituting yields:

$$\sum_{a\in L_L} p_a \cdot \left(\sum_{a'\in L, a\neq a'} \frac{p_{a'}}{p_{a'}+p_a} \right) + \sum_{b\in L_R} p_b \cdot \left(\sum_{b'\in L, b\neq b'} \frac{1}{2} \right)$$

But all items in L_L are accessed with equal frequency therefore we get:

$$\sum_{a\in L_L} p_a \cdot \left(\sum_{a'\in L, a\neq a'} \frac{1}{2} \right) + \sum_{b\in L_R} p_b \cdot \left(\sum_{b'\in L, b\neq b'} \frac{1}{2} \right)$$

$$\sum_{a\in L_L} p_a \cdot \left(\sum_{a'\in L, a\neq a'} \frac{1}{2} \right) + \sum_{b\in L_R} p_b \cdot \left(\sum_{b'\in L, b\neq b'} \frac{1}{2} \right)$$

$$\leq \sum_{a\in L_L} p_a \cdot \left(\frac{|L_L|}{2} \right) + \sum_{b\in L_R} p_b \cdot \left(\frac{|L_R|}{2} \right)$$

$$\leq p_a \cdot \left(\frac{|L_L|^2}{2} \right) + p_b \cdot \left(\frac{|L_R|^2}{2} \right)$$

We perform a similar calculation for move-to-front:

$$\sum_{a\in L_L} p_a \cdot \left(\sum_{a'\in L, a\neq a'} b(a',a) \right) + \sum_{b\in L_R} p_b \cdot \left(\sum_{b'\in L, b\neq b'} b(b',b) \right)$$

$$= \sum_{a\in L_L} p_a \cdot \left(\sum_{a'\in L, a\neq a'} \frac{p_{a'}}{p_{a'}+p_a} \right) + \sum_{b\in L_R} p_b \cdot \left(\sum_{b'\in L, b\neq b'} \frac{p_{b'}}{p_{b'}+p_b} \right)$$

$$= \sum_{a\in L_L} p_a \cdot \left(|L_L| \cdot \frac{1}{2} + |L_R| \cdot \frac{p_b}{p_b+p_a} \right) + \sum_{b\in L_R} p_b \cdot \left(|L_R| \cdot \frac{1}{2} + |L_L| \cdot \frac{p_a}{p_b+p_a} \right)$$

$$= p_a \cdot \left(|L_L|^2 \cdot \frac{1}{2} + |L_L| \cdot |L_R| \cdot \frac{p_b}{p_b + p_a} \right) + p_b \cdot \left(|L_R|^2 \cdot \frac{1}{2} + |L_R||L_L| \cdot \frac{p_a}{p_b + p_a} \right)$$

Thus, move-to-front has a higher expected cost than the learned algorithm.

This analysis can be extended in many ways. For example, consider the case where we vary the ratio $\frac{p_a}{p_b}$. As $\frac{p_a}{p_b} \to 1$, the expected cost of both algorithms are exactly the same even thought the learned algorithm follows a randomized move-to-front policy. As $\frac{p_a}{p_b} \to \infty$[1], the banded algorithm outperforms move-to-front.

Interestingly, varying $\frac{|L_R|}{|L_L|}$ will have an effect on the probabilities p_a and p_b. As $\frac{|L_L|}{|L_R|} \to \infty$, $\frac{p_a}{p_b}$ decreases. Given the behaviour of the learned algorithm described in Section 5.1, records in L_R will be moved to bucket L_L and records in L_L will be moved to bucket L_R, restoring the invariant that heavy records are found in buckets with lower indices.

Another interesting consideration is generalizing to $k > 2$ buckets. This means we have a distribution where we can group elements by their frequency of access into k groups. Then, under the assumption that the learned algorithm is able to break them into k groups, the average cost of the banded policy is:

$$\sum_{k \in \mathcal{K}} \leq p_k \cdot \left(\frac{|L_k|^2}{2} \right)$$

Doing the same calculation for move-to-front:

$$= \sum_{k \in K} p_k \cdot \left(|L_k|^2 \cdot \frac{1}{2} + \sum_{k' \in K, k' \neq k} |L_k| \cdot |L_{k'}| \cdot \frac{p_{k'}}{p_{k'} + p_k} \right)$$

. In practice, the learned algorithm may use fewer than k buckets because it only

[1]We do not consider the case of $\frac{p_a}{p_b} \to 0$ because it is analogous to case where it approaches ∞

approximates the bands. Evidence of this is seen in Figure 5-7 where it only creates 3 bands.

A remarkable observation we make here is that even when we have uniform access over L, the algorithm executes randomized move-to-front over the entire list rather than move-to-front. This likely suggests that when L_L gets too large, it no longer does move-to-front. To confirm this, we look at the policy map in Figure 5-35 for a Heavy/Light distribution where 50% of the list is accessed 90% of the time. Compare this policy map to Figure 5-36 . In Figure 5-35, we see it follows a randomized move-to-front policy for the first bucket whereas for Figure 5-36, it follows a move-to-front policy for the first bucket.

5.4 The Learned Algorithm in Practice & Some Insights

One advantage of the learned algorithm over the order-by-access or other frequency count techniques is that it has finite size while still being competitive. This is because the neural network, which uses finite memory, learns which elements are heavy and which ones are light and buckets them appropriately. As we saw in Chapter 4, it is also able to identify changes in the heaviness status of an element and reorganizes the list accordingly.

On the other hand, it is not a memory-less heuristic like move-to-front or the transposition heuristic. In fact without careful design, the memory overhead can get quite large. Techniques like reducing the size of the experience buffer and the state representation help lower this memory overhead usually at the cost of performance.

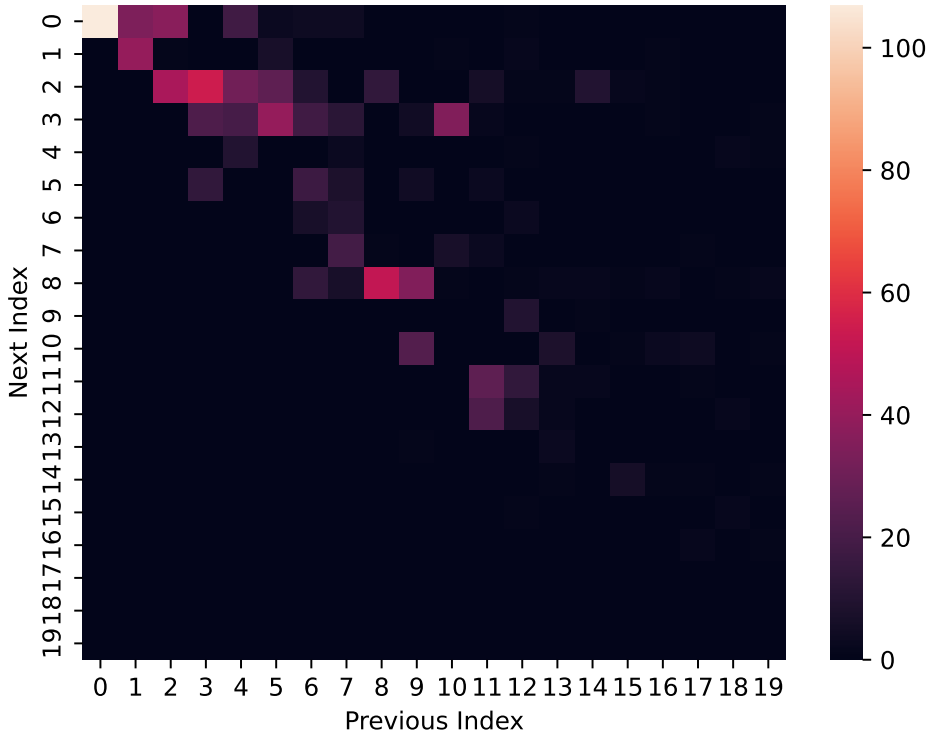

Figure 5-35: Policy Map for Learned Algorithm. Shows all move choices over an episode in a list of size 20. Heavy/Light Query Sequence with 50% heavy, 90% heavy accessed, No list change, No distribution change

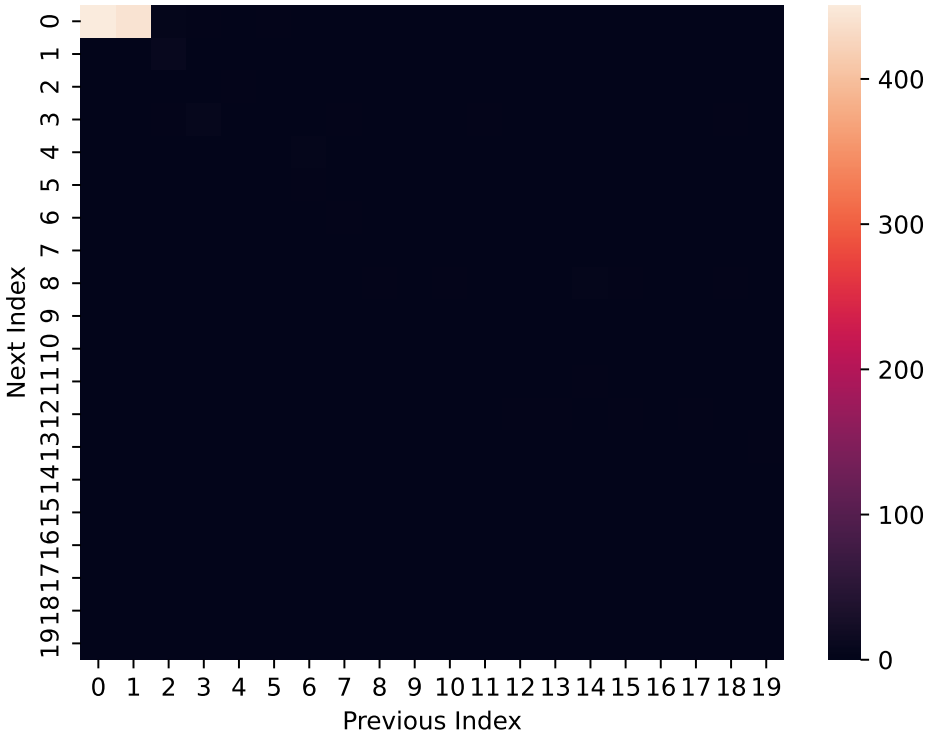

Figure 5-36: Policy Map for Learned Algorithm. Shows all move choices over an episode in a list of size 20. Heavy/Light Query Sequence with 10% heavy, 90% heavy accessed, No list change, No distribution change

Chapter 6

A New Competitive Ratio and Surprising Observations

As we saw in Chapter 5, the learned algorithm does not move **all** items to the front of the list; it adopts a more conservative approach for less frequently accessed items. This was a surprising observation, given the reward function, but confirms the notion expressed in Chapter 2.3: while move-to-front is competitive, algorithms that do not always move items to the front tend to achieve better competitive ratios.

Of particular interest is the argument presented in [43]. There Rivest proves that the transposition heuristic achieves a lower expected search cost even though no competitive ratio was explicitly computed. In this chapter, we will show that the transposition heuristic in fact achieves $(1 + o(1)) \cdot \text{OPT}$ for Zipfian query sequences.

6.1 Proof Setup and Useful Lemmas

Recall definition 1 from Chapter 5 that the expected search cost for record r_i is:

$$\sum_{1 \leq j \leq n, j \neq i} b(j,i) \cdot 1$$

Taking the expectation across all records, we have that the expected search time for a record in L is:

$$\sum_{i=0}^{n} p_i \cdot \left(\sum_{1 \leq j \leq n, j \neq i} b(j,i) \cdot 1 \right)$$

For a Zipfian distribution where $p_i \propto \frac{1}{i}$ with skew($\alpha = 1$), we can write the following for the expected search cost:

$$\sum_{i=0}^{n} \frac{1}{H_n} \cdot \frac{1}{i} \cdot \left(\sum_{i \leq j \leq n, j \neq i} b(j,i) \cdot 1 \right)$$

where H_n is the nth harmonic number. To achieve OPT, items in L must be organized from most frequently accessed to least frequently accessed. The OPT expected search cost, therefore, for a Zipfian access distribution will yield:

$$\sum_{i=0}^{n} \frac{1}{H_n} \cdot \frac{1}{i} \cdot i = \sum_{i=0}^{n} \frac{1}{H_n} = \frac{1}{H_n} \sum_{i=0}^{n} 1 = \frac{n}{H_n} = O\left(\frac{n}{\ln n} \right)$$

When we consider the stationary distribution over the permutations of L, we define $Q(\pi)$ for a permutation π of L to be the probability of π in a stationary distribution. Given a set S of permutations, we define $Q(S) = \sum_{\pi \in S} Q(\pi)$.

In [43], the following useful lemma is presented for the stationary distribution under the transposition heuristic:

Lemma 6.1.1 *Let π be a permutation where record a immediately precedes record b and let π' be the same permutation but with a and b swapped, then*

$$\frac{Q(\pi)}{Q(\pi')} = \frac{p_a}{p_b}.$$

This will be the driving lemma for the proof.

Lemma 6.1.2 *Let $b(i,j)$ be the probability that record i appears before record j in L and let $b(j,i)$ be analogously defined. Then:*

$$\frac{b(i,j)}{b(j,i)} = \frac{\sum_{\pi, i \text{ appears before } j} Q(\pi)}{\sum_{\pi, j \text{ appears before } i} Q(\pi)}.$$

Lemma 6.1.3 *Define $b(i,j)$ as in Lemma 6.1.2 but for a list L maintained under the transposition heuristic. Define $b'(i,j)$ also as in Lemma 6.1.2 but for a list maintained under the move-to-front heuristic. Then:*

$$b(i,j) > b'(j,i) = \frac{p_i}{p_i + p_j}$$

Define D_t to be the set of permutations where record j appears before record i and there are at least t other records between them. We are interested in proving the competitive ratio for the following setting:

- $i \geq \log n$
- $j \geq i + i^{0.9}$
- $0 \leq t \leq i^{0.15} := T$

An important fact which we use repeatedly is that because $j > i$, $p_j < p_i$ under a Zipfian access distribution.

6.2 Main Lemmas

For the setting described in 6.1, let π be a permutation in the set of permutations D_t (Recall that $0 \leq t \leq i^{0.15} := T$). Then we consider the following partitioning for D_t:

1. Partition 1 (D_t^1): Permutations where a record r_l, $l \geq j - 2j^{0.75}$ is immediately before j.

2. Partition 2 (D_t^2): Permutations **not** in Partition 1 where a record r_l, $l \in \{j - 1, j - 2, ..., j - j^{0.75}\}$ is somewhere before j.

3. Partition 3 (D_t^3): All other permutations not found in Partition 1 or 2.

From the description, we see that these partitions are disjoint and also cover all of D_t. Given a permutation π, we can swap records around in π to yield another permutation π'. Thus swapping records around in π gives rise to a mapping from π to other permutations. To this end, we define a mapping f for each partition of D_t as follows:

1. $f_{D_t^1}$: Swap the record immediately before record j with record j. Denote the set of permutations arising from applying this mapping to D_t^1 as $\mathcal{D}_t^1$.

2. $f_{D_t^2}$: Swap one of the records r_l, $l \in \{j - 1, j - 2, ..., j - j^{0.75}\}$ with the record immediately before j. Then swap r_l with record j. Denote the set of permutations arising from applying this mapping to D_t^2 as $\mathcal{D}_t^2$.

3. $f_{D_t^3}$: Swap record i and record j. Then swap one of the records r_l, $l \in \{j - 1, j - 2, ..., j - \frac{j^{0.75}}{2}\}$ that appears farthest after record j. Denote the set of permutations arising from applying this mapping to D_t^3 as $\mathcal{D}_t^3$.

We denote the set of all permutations after applying each partition's respective mapping as $\mathcal{D}_t$.

We now present the first building-block lemma for permutations in Partition 1 which is a consequence of Lemma 6.1.1:

Lemma 6.2.1

$$\sum_{\pi \in D_t^1} Q(\pi) \leq \left(1 + \frac{4j^{0.75}}{j}\right) \cdot \sum_{\pi' \in \mathcal{D}_t^1} Q(\pi')$$

Proof: First consider single terms $Q(\pi)$ and $Q(\pi')$ where $\pi \in D_t^1$ and $\pi' \in \mathcal{D}_t^1$. Let r_l be the record that was swapped with record j. Then from lemma 6.1.1., we see that:

$$\frac{Q(\pi)}{Q(\pi')} = \frac{p_l}{p_j}$$

But we know that $l \geq j - 2j^{0.75}$ from our description of Partition 1 and under a Zipfian access distribution, we have that $p_l \leq p_{j-2j^{0.75}}$ which yields:

$$\frac{Q(\pi)}{Q(\pi')} \leq \frac{p_{j-2j^{0.75}}}{p_j}$$

Substituting that $p_i \propto \frac{1}{i}$ for Zipfian, we get:

$$\frac{Q(\pi)}{Q(\pi')} \leq \frac{j}{j - 2j^{0.75}}$$

Rearranging and using a taylor approximation of $\frac{1}{1-c}$, we get:

$$Q(\pi) \leq \left(1 + \frac{2j^{0.75}}{j}\right) Q(\pi')$$

Since $f_{D_t^1}$ is a one-to-one mapping, we can sum over all permutations in π and π' on both sides to yield the inequality:

$$\sum_{\pi \in D_t^1} Q(\pi) \leq \left(1 + \frac{2j^{0.75}}{j}\right) \sum_{\pi' \in \mathcal{D}_t^1} Q(\pi') \leq \left(1 + \frac{4j^{0.75}}{j}\right) \sum_{\pi' \in \mathcal{D}_t^1} Q(\pi')$$

Finally, as desired, we get:

$$\sum_{\pi \in D_t^1} Q(\pi) \leq \left(1 + \frac{4j^{0.75}}{j}\right) \sum_{\pi' \in \mathcal{D}_t^1} Q(\pi')$$

We present a similar lemma for permutations in Partition 2:

Lemma 6.2.2

$$\sum_{\pi \in D_t^2} Q(\pi) \leq \left(1 + \frac{2(j^{0.75})^2}{j^2}\right) \cdot \sum_{\pi' \in \mathcal{D}_t^2} Q(\pi')$$

Proof: First denote $D_t^{2'}$ as the set of permutations obtained from performing the first swap in the mapping on permutations in D_t^2. Then, from Lemma 6.1.1, we can also write the following for $\pi \in D_t^2$ and $\pi'' \in D_t^{2'}$:

$$\frac{Q(\pi)}{Q(\pi'')} = \frac{p_l}{p_b}$$

where $l \in \{j-1, ..., j-j^{0.75}\}$ and r_b is the record immediately before record j with access probability p_b. Based on our description of Partition 2, we know that $b < j - 2j^{0.75}$ and so importantly, $p_b > p_{j-2j^{0.75}}$. From this, we get that:

$$\frac{Q(\pi)}{Q(\pi'')} \leq \frac{p_l}{p_{j-2j^{0.75}}}$$

In the same way, we can also write an equation between $Q(\pi'')$ and $Q(\pi)$ where

$\pi'' \in D_t^{2'}$ and $\pi' \in \mathcal{D}_t^2$:

$$\frac{Q(\pi'')}{Q(\pi')} = \frac{p_l}{p_j}$$

Multiplying both sides of the inequality with this new relation:

$$\frac{Q(\pi)}{Q(\pi')} \leq \frac{p_l^2}{p_j \cdot_{j-2j^{0.75}}}$$

But much like in the previous proof, $p_l \leq p_{j-j^{0.75}}$, so we get that:

$$\frac{Q(\pi)}{Q(\pi')} \leq \frac{p_{r_{j-j^{0.75}}}^2}{p_j \cdot_{r_{j-2j^{0.75}}}}$$

. Substituting for a Zipfian access distribution, we get that:

$$\frac{Q(\pi)}{Q(\pi')} \leq \frac{j \cdot (j - 2j^{0.75})}{(j - j^{0.75})^2}$$

. Rewriting $j \cdot (j - 2j^{0.75})$:

$$j \cdot (j - 2j^{0.75}) = (j - j^{0.75} + j^{0.75}) \cdot (j - j^{0.75} - j^{0.75}) = (j - j^{0.75})^2 - (j^{0.75})^2$$

This gives us finally that:

$$\frac{Q(\pi)}{Q(\pi')} \leq \frac{(j - j^{0.75})^2 - (j^{0.75})^2}{(j - j^{0.75})^2} = \left(1 - \frac{(j^{0.75})^2}{(j - j^{0.75})^2}\right)$$

Rearranging, we get that:

$$Q(\pi) \leq \left(1 - \frac{(j^{0.75})^2}{(j - j^{0.75})^2}\right) \cdot Q(\pi') \leq \left(1 + \frac{2(j^{0.75})^2}{j^2}\right) \cdot Q(\pi')$$

Summing over all permutations in D_t^2, we get:

$$\sum_{\pi \in D_t^2} Q(\pi) \leq \left(1 + \frac{2(j^{0.75})^2}{j^2}\right) \cdot \sum_{\pi' \in \mathcal{D}_t^2} Q(\pi')$$

as desired.

We present the last building block lemma for permutations in Partition 3:

Lemma 6.2.3

$$\sum_{\pi \in D_t^3} Q(\pi) \leq \left(1 - \frac{j^{0.75}}{2j}\right)^{\frac{j^{0.75}}{2}} \cdot \sum_{\pi' \in \mathcal{D}_t^3} Q(\pi')$$

Proof: First denote $D_t^{3'}$ as the set of permutations obtained from performing the first swap in the mapping on permutations in D_t^3. The first observation is that j and i are not necessarily next to each other. Therefore, we cannot write a simple ratio between $Q(\pi)$ and $Q(\pi'')$ where $\pi \in D_t^3$ and $\pi'' \in D_t^{3'}$. We must therefore apply Lemma 6.1.1 repeatedly to obtain this ratio. That is, consider intermediate permutations π_1 such that we swap record i with the record in front of it, the permutation π_2 such that we swap record i with the record in front of it again, and on and on until record j is directly in front of i. After swapping j and i, you must swap j back down to record i's old position. If we let $r_1^t, r_2^t, ..., r_t^t$ be the set of records that are between j and i, we can chain the ratios together to get:

$$\frac{Q(\pi)}{Q(\pi'')} = \frac{p_{r_t^t}}{p_{r_i}} \cdot \frac{p_{r_{t-1}^t}}{p_{r_i}} \cdot ... \cdot \frac{p_{r_1^t}}{p_{r_i}} \cdot \frac{p_{r_j}}{p_i} \cdot \frac{p_j}{p_{r_1^t}} \cdot \frac{p_{r_j}}{p_{r_2^t}} \cdot ... \cdot \frac{p_{r_j}}{p_{r_t^t}} = \left(\frac{p_{r_j}}{p_{r_i}}\right)^{t+1}$$

Applying a similar reasoning, when we attempt to swap j with the record from among $r_{j-j^{0.75}}, ..., r_{j-\frac{j^{0.75}}{2}}$ appearing farthest to its right, we can chain ratios together

to get the following ratio:

$$\frac{Q(\pi'')}{Q(\pi')} = \left(\frac{p_{r_j}}{p_{r_l}}\right)^{k+1}$$

where k is the number of records between record j and the rightmost record. Notice that records $r_{j-j^{0.75}}...r_{j-1}$ must appear after record j on the assumption that Partition 3 considers all permutations except those found in Partition 1 and 2. For the second swap, we are choosing among records $r_{j-j^{0.75}}, ..., r_{j-\frac{j^{0.75}}{2}}$ and so $p_l \geq p_{j-\frac{j^{0.75}}{2}}$ which gives:

$$\frac{Q(\pi'')}{Q(\pi')} \leq \left(\frac{p_{r_j}}{p_{r_{j-\frac{j^{0.75}}{2}}}}\right)^{k+1}$$

Substituting for Zipfian distribution and multiplying the two ratios, we get:

$$\frac{Q(\pi)}{Q(\pi')} \leq \left(\frac{i}{j}\right)^{t+1} \cdot \left(\frac{j-\frac{j^{0.75}}{2}}{j}\right)^{k+1}$$

Consider an arrangement of L where all the records $r_{j-j^{0.75}}...r_{j-1}$ appear consecutively after j with record i spliced t spots behind j. Such an arrangement presents the closest the rightmost element from among $r_{j-j^{0.75}}...r_{j-1}$ could ever be to j. Therefore $k \geq \frac{j^{0.75}}{2} - t$. Substituting this yields:

$$Q(\pi) \leq \left(\frac{i}{j}\right)^{t+1} \cdot \left(\frac{j-\frac{j^{0.75}}{2}}{j}\right)^{\frac{j^{0.72}}{2}-t+1} \cdot Q(\pi')$$

Seeing that $j - j^{0.75} > i$, we can write that:

$$Q(\pi) \leq \left(\frac{i}{j}\right)^{t+1} \cdot \left(\frac{j-\frac{j^{0.75}}{2}}{j}\right)^{\frac{j^{0.72}}{2}-t+1} \cdot Q(\pi') \leq \left(\frac{j-\frac{j^{0.75}}{2}}{j}\right)^{\frac{j^{0.72}}{2}+2} \cdot Q(\pi')$$

Simplifying and summing over permutations, we get the desired inequality:

$$\sum_{\pi \in D_t^3} Q(\pi) \leq \left(1 - \frac{j^{0.75}}{2j}\right)^{\frac{j^{0.72}}{2}+2} \cdot \sum_{\pi' \in \mathcal{D}_t^3} Q(\pi') \leq \left(1 - \frac{j^{0.75}}{2j}\right)^{\frac{j^{0.72}}{2}} \cdot \sum_{\pi' \in \mathcal{D}_t^3} Q(\pi').$$

In the next section, we put these lemmas together to prove two theorems.

6.3 Putting it all together: Competitive Ratio Results

We start with the first result:

Theorem 6.3.1 *Let $b(i,j)$ be the probability that record i appears before record j under the transposition heuristic and analogously define $b(j,i)$. Under the settings of Section 6.1, we have*

$$\frac{b(i,j)}{b(j,i)} \geq (1 - o(1)) \frac{1}{T} \cdot \left(\frac{j}{i}\right)^T.$$

Proof: Using Lemma 6.1.2, we have:

$$\frac{b(i,j)}{b(j,i)} = \frac{\sum_{\pi, i \text{ appears before } j} Q(\pi)}{\sum_{\pi, j \text{ appears before } i} Q(\pi)}.$$

We decompose the set of permutations we sum over in the denominator based on the number of elements appearing between record j and record i. Denote D_m as the set of permutations such that record j and record i are separated by m other records. Given our work in Section 6.2, we choose to split the set D_m into two groups: the set of permutations where record j and record i are separated by $\leq T$ (recall that $T = i^{0.15}$) records and permutations where they are separated by $> T$

records. We rewrite the ratio of sums as follows:

$$\frac{b(i,j)}{b(j,i)} = \frac{\sum_{\pi, i \text{ appears before } j} Q(\pi)}{\sum_{m=0}^{T} Q(D_m) + \sum_{m>T} Q(D_m)}$$

We decompose the first summation into the three partitions from Section 6.2 to yield:

$$\frac{b(i,j)}{b(j,i)} = \frac{\sum_{\pi, i \text{ appears before } j} Q(\pi)}{\sum_{m=0}^{T} Q(D_m^1) + \sum_{m=0}^{T} Q(D_m^2) + \sum_{m=0}^{T} Q(D_m^3) + \sum_{m>T} Q(D_m)}$$

Using Lemma 6.2.1 and 6.2.2, we can bound the first two sums in the denominator as follows:

$$\sum_{m=0}^{T} Q(D_m^1 \cup D_m^2) \leq T \cdot \left(1 + \frac{4j^{0.75}}{j}\right) \cdot Q(D_T) \leq T \cdot \left(1 + \frac{4j^{0.75}}{j}\right)^T \cdot Q(D_T)$$

Recall that the resulting permutations from applying D_m^1's and D_m^2's respective mappings to them keeps j in front of i and keeps them $m+1$ spots apart. For $m < T$, this ensures j and i are still within T of each other.

We now seek to use Lemma 6.2.3 to bound the third sum in the denominator. When we apply the mapping on permutations in partition 3, j is no longer in front of i. In fact, i appears before j. Therefore, it must be an expression involving a term in the numerator, since all permutations with i before j are in the numerator. Notice, that for every permutation in the numerator, there is a "counterpart" in the denominator which is exactly identical except that j and i's positions have been swapped. Therefore, we can write the following for the third summation in the denominator:

$$\sum_{m=0}^{T} Q(D_m^3) \leq \left(1 - \frac{j^{0.75}}{2j}\right)^{\frac{j^{0.75}}{2}} \cdot Q(N_{m'})$$

where $m' \geq m + \frac{j^{0.75}}{2}$. Using the approximation $\left(1 - \frac{1}{x}\right)^{ky} \leq e^{-\frac{ky}{x}}$, we write that:

$$\sum_{m=0}^{T} Q(D_m^3) \leq (e^{-\frac{j^{0.75}}{2} \cdot \frac{j^{0.75}}{2j}}) \cdot Q(N_{m'}) = (e^{-\frac{j^{2 \cdot 0.75}}{4j}}) \cdot Q(N_{m'})$$

We make two interesting observations. The first is that $m' > T$, and so if we also decompose the numerator into two sums ($\sum_{m=0}^{T} Q(N_m) + \sum_{m>T} Q(N_m)$) like we did initially with the denominator, we can write that:

$$\sum_{m=0}^{T} Q(D_m^3) \leq (e^{-\frac{j^{2 \cdot 0.75}}{4j}}) \cdot \sum_{m>T} Q(N_m) \leq T \cdot (e^{-\frac{j^{2 \cdot 0.75}}{4j}}) \cdot \sum_{m>T} Q(N_m)$$

The second observation is that we can rewrite $\sum_{m=0}^{T} Q(N_m)$ in terms of D_T. Recall that one can match each permutation in the numerator with a permutation in the denominator where the position of j and i are swapped. For N_m where $m \leq T$, we can write:

$$\sum_{m=0}^{T} Q(N_m) \leq Q(D_T) \cdot \left(\frac{p_i}{p_j}\right)^T$$

Putting all these together, we have that:

$$\frac{b(i,j)}{b(j,i)} = \frac{Q(D_T) \cdot (\frac{p_i}{p_j})^T + \sum_{m>T} Q(N_m)}{T \cdot (1 + \frac{4j^{0.75}}{j})^T \cdot Q(D_T) + \sum_{m=0}^{T} Q(D_m^3) + \sum_{m>T} Q(D_m)}$$

$$\geq \frac{Q(D_T) \cdot (\frac{p_i}{p_j})^T + \left(Te^{\frac{-j^{0.75 \cdot 2}}{4j}}\right) \cdot \sum_{m=0}^{T} Q(D_m^3)}{T \cdot (1 + \frac{4j^{0.75}}{j})^T \cdot Q(D_T) + \sum_{m=0}^{T} Q(D_m^3) + \sum_{m>T} Q(D_m)}$$

Because we want to use the fact that $\frac{a+b+c}{d+e+f} \geq \min(\frac{a}{d}, \frac{b}{e}, \frac{c}{f})$, we split the second

term in the numerator into two and express it in terms of $\sum_{m>T} Q(D_m)$ to yield:

$$\geq \frac{Q(D_T) \cdot (\frac{p_i}{p_j})^T + \left(\frac{1}{2} T e^{\frac{-j^{0.75 \cdot 2}}{4j}}\right) \cdot \sum_{m=0}^{T} Q(D_m^3) + \frac{1}{2} \cdot (\frac{p_i}{p_j})^T (\sum_{m>T} Q(D_m))}{T \cdot (1 + \frac{4j^{0.75}}{j})^T \cdot Q(D_T) + \sum_{m=0}^{T} Q(D_m^3) + \sum_{m>T} Q(D_m)}$$

Using the inequality $\frac{a+b+c}{d+e+f} \geq \min(\frac{a}{d}, \frac{b}{e}, \frac{c}{f})$ and substituting for a Zipfian distribution, we get our desired bound:

$$\frac{b(i,j)}{b(j,i)} \geq \min\left(\frac{1}{T} \cdot \left(\frac{j}{i}\right)^T \cdot \frac{1}{(1+4j^{0.75}/j)^T}, \frac{1}{2} T e^{\frac{-j^{0.75 \cdot 2}}{4j}}, \frac{1}{2} \cdot \left(\frac{j}{i}\right)^T\right)$$

$$\geq (1 - o(1)) \frac{1}{T} \cdot \left(\frac{j}{i}\right)^T$$

Theorem 6.3.2 *Suppose n is sufficiently large. Under a Zipfian access distribution, the expected lookup cost of the transposition rule is $\frac{n}{\log n}(1 + o(1))$.*

Proof In Section 6.1, we showed that the expected search cost for a particular record i:

$$\sum_{i=0}^{n} p_i \cdot \left(\sum_{1 \leq j \leq n, j \neq i} b(j,i) \cdot 1\right)$$

We will compute this sum first for $j \leq i + i^{0.9}$ and start by substituting Zipfian access distribution:

$$\sum_{i=0}^{n} p_i \cdot \left(\sum_{1 \leq j \leq n, j \neq i} b(j,i)\right) = \sum_{i=0}^{n} \frac{1}{H_n} \cdot \frac{1}{i} \cdot \left(\sum_{1 \leq j \leq i+i^{0.9}, j \neq i} b(j,i)\right)$$

$$\leq \frac{1}{H_n} \sum_{i=0}^{n} \frac{1}{i} \cdot \left(i + \sum_{i \leq j \leq i+i^{0.9}, j \neq i} b(j,i)\right)$$

$$\leq \frac{1}{H_n}\sum_{i=0}^{n} 1 + \left(\frac{1}{i}\sum_{i\leq j\leq i+i^{0.9}, j\neq i} b(j,i)\right)$$

$$\leq \frac{n}{H_n} + \frac{1}{H_n}\sum_{i=0}^{n}\frac{1}{i}\left(\sum_{i\leq j\leq i+i^{0.9}, j\neq i} b(j,i)\right)$$

Using Lemma 6.1.3 and Zipfian probability formula, we can write:

$$\leq \frac{n}{H_n} + \frac{1}{H_n}\sum_{i=0}^{n}\frac{1}{i}\left(\sum_{i\leq j\leq i+i^{0.9}, j\neq i}\frac{i}{j+i}\right) = \frac{n}{H_n} + \frac{1}{H_n}\sum_{i=0}^{n}\left(\sum_{i\leq j\leq i+i^{0.9}, j\neq i}\frac{1}{j+i}\right)$$

$$\leq \frac{n}{H_n} + \frac{1}{H_n}\sum_{i=0}^{n}\left(\int_{2i}^{2i+i^{0.9}}\frac{1}{x}\,dx\right) = \frac{n}{H_n} + \frac{1}{H_n}\sum_{i=0}^{n}\left(\ln\left(\frac{2i+i^{0.9}}{2i}\right)\right)$$

$$= \frac{n}{H_n} + \frac{1}{H_n}\sum_{i=0}^{n}\left(\ln\left(1+\frac{1}{2i^{0.1}}\right)\right) \leq \frac{n}{H_n} + \frac{1}{H_n}\cdot n \leq (1+o(1))\frac{n}{H_n}$$

We now handle the case where $j > i + i^{0.9}$. We make use of Theorem 6.2.3 which says that:

$$\frac{b(i,j)}{b(j,i)} \geq \frac{1}{2}\cdot\frac{1}{T}\cdot\left(\frac{j}{i}\right)^{T}$$

From this, we can also say that:

$$b(i,j) \geq \frac{1}{2}\cdot\frac{1}{T}\cdot\left(\frac{j}{i}\right)^{T}, b(j,i) = 1 - b(i,j) \leq 1 - \frac{1}{2}\cdot\frac{1}{T}\cdot\left(\frac{j}{i}\right)^{T} = \frac{2Ti^T}{2Ti^T + j^T}$$

We now bound the expected search cost sum as follows:

$$\sum_{i=0}^{n} p_i \cdot \left(\sum_{j>i+i^{0.9}} b(j,i)\right) = \sum_{i=0}^{n}\frac{1}{H_n}\cdot\frac{1}{i}\cdot\left(\sum_{j>i+i^{0.9}} b(j,i)\right)$$

$$\leq \frac{1}{H_n}\sum_{i=0}^{n}\frac{1}{i}\sum_{j>i+i^{0.9}}\frac{2Ti^T}{2Ti^T + j^T} = \frac{1}{H_n}\sum_{i=0}^{n}\frac{1}{i}\sum_{j>i+i^{0.9}}\frac{1}{1+\frac{j^T}{2Ti^T}}$$

$$\leq \frac{1}{H_n} \cdot \sum_{i=0}^{n} \frac{2Ti^T}{i} \int_{i+i^{0.9}}^{\infty} \frac{1}{x^T}\, dx$$

$$\leq \frac{1}{H_n} \cdot \sum_{i=0}^{n} \frac{2Ti^T}{i} \int_{i+i^{0.9}}^{\infty} \frac{1}{x^T}\, dx$$

$$\leq \frac{1}{H_n} \cdot \sum_{i=0}^{n} \frac{2Ti^{T-1}}{(T-1)\cdot(i+i^{0.9})^{T-1}}$$

$$\leq \frac{1}{H_n} \cdot \frac{2T}{T-1} \sum_{i=0}^{n} \left(\frac{1}{1+i^{-0.1}}\right)^{T-1}$$

$$\leq \frac{1}{H_n} \cdot \frac{2T}{T-1} \sum_{i=0}^{n} \left(\frac{i^{0.1}}{i^{0.1}+1}\right)^{T-1}$$

$$\leq \frac{1}{H_n} \cdot \frac{2T}{T-1} \sum_{i=0}^{n} \left(1-\frac{1}{i^{0.1}+1}\right)^{T-1}$$

Using the approximation that $\left(1-\frac{1}{x}\right)^{ky} \leq e^{-\frac{ky}{x}}$, we can write:

$$\leq \frac{1}{H_n} \cdot \frac{2T}{T-1} \sum_{i=0}^{n} \left(e^{-\frac{T-1}{1+i^{0.1}}}\right) = O\left(\frac{1}{\ln n}\right) \cdot \sum_{i=1}^{n} e^{-\Theta(i^{0.05})} = o(1)$$

Putting all of this together, we get the desired bound for the expected search cost to be $(1+o(1))\frac{n}{\ln n}$.

Notice, however, that the analysis above is for the setting we gave in Section 6.1 where $i \geq \log n$. We now consider the case where $i < \log n$:[1]

$$\sum_{i \leq \log n} p_i \cdot \left(\sum_{j > i+i^{0.9}} b(j,i)\right) \leq \frac{1}{H_n} \sum_{i \leq \log n} \sum_{j > i+i^{0.9}} \frac{1}{i+j} = \frac{1}{H_n} \sum_{i \leq \log n} H_n - H_i$$

[1]The summation here is for $j > i + i^{0.9}$ because the case when $i < \log n$ and $j < i + i^{0.9}$ was handled when we considered the case $j \leq i + i^{0.9}$

$$= \frac{1}{H_n} \cdot \left(\sum_{i \leq \log n} H_n - \sum_{i \leq \log n} H_i \right) = \log n - \frac{1}{H_n} \sum_{i \leq \log n} H_i = O(\log n)$$

which is still $(1 + o(1)) \frac{n}{\ln n}$.

While the proof above supports the conjecture presented by Rivest in [43] for a Zipfian access, a counterexample to the conjecture is offered in [7]. The counterexample, however, differs in an important way: our results prove the conjecture for Zipfian accesses on a large list. The counterexample presented focuses on a distribution where all but two of the records are heavily accessed and every other record has equal probability of access. They somewhat confirm this reasoning by stating that the transposition heuristic only performs worse when the access probabilities resemble the one in their given counterexample.

Chapter 7

Discussion of Design Choices

In Chapter 3, we presented the design of the reinforcement learning agent. In this chapter, we discuss those design choices and any trade-offs.

7.1 Choice of Reinforcement Learning Techniques

In Chapter 3 we mentioned that we used a form of model-free deep Q-learning with experience replay and target learning. We describe each choce below:

1. Model-free learning: The alternative to model-free learning is model-based learning. Given the experiments that we ran, we could have used model-based learning since we knew the distributions a priori. However, this would not be representative of the online setting we wanted to simulate for the reinforcement learning agent. Therefore, we opted for model-free learning to better capture the application domain.

2. Experience Replay: The experience memory/buffer is essential to model-free learning. In model-free learning, the agent first collects information

about the environment and then learns from that information so the information collected must be stored in a buffer. This buffer is called an **experience buffer**. A single unit of experience memory consists of a tuple of the current state s_t, the action that was taken a_t, the reward that was received r_t and the resulting state s_{t+1}. As new memories come, old ones are removed to make room where necessary.

In experience replay, we use the experience memory units to compute the "loss" of the neural network on its approximation of the optimal Q-function and then update the network accordingly. The units are chosen uniformly at random from the experience buffer to break any correlation that might arise from using sequential memory units. We discuss how we choose the right buffer size in section 7.2.

3. Target Learning: Let us consider the update equation used in deep Q-learning:

$$q_t(s,a) = 1 - \text{learning rate} \cdot q_t(s,a) + \\ \text{learning rate} \cdot \max_{a' \in A}(r_t + \text{discount factor} \cdot q_{t-1}(s,a'))$$

In the above equation, computing $\max_{a' \in A}(r_t + \text{discount factor} * q_{t-1}(s,a'))$ relies on estimates from the neural network itself. This causes the Q values to not converge and our estimates of the Q-function are unstable. To rectify this, we maintain two neural networks. We call one the Q-network and the other, the target network. The estimates used for computing $\max_{a' \in A}(r_t + \text{discount factor} * q_{t-1}(s,a'))$ come from the target network to update the Q-network's values during training. Periodically, the weights of the Q-network are transferred to the target network. This way, our Q-values converge to a

stable value.

7.2 Choosing the Right Size for the Experience Buffer

Choosing the right buffer size can have a big impact on the quality of the Q-function approximation and consequently the policy. If the buffer size is too large, the learned algorithm's memory overhead increases and we run the risk of holding on to memories which are not relevant for future inference. If the buffer size is too small, we may also lose vital examples that could inform the learned algorithm's actions in the future.
We determine the right buffer size for our setting through experimentation. The experiment we conduct is similar to the one described in Chapter 4, but we outline it here for clarity:

1. Initialize a static list L.
2. Generate a sequence of queries Q to be made to items in L. Q may be sampled from a distribution or constructed adversarially.
3. Initialize a different reinforcement learning agent for each buffer size.
4. Simulate a workload over time using Q.
5. Allow each reinforcement learning agent N time steps of training so it can learn an optimal policy.
6. Stage a testing phase for each learned algorithm.
7. During the testing phase, we measure and record cost of search for each reinforcement learning agent.

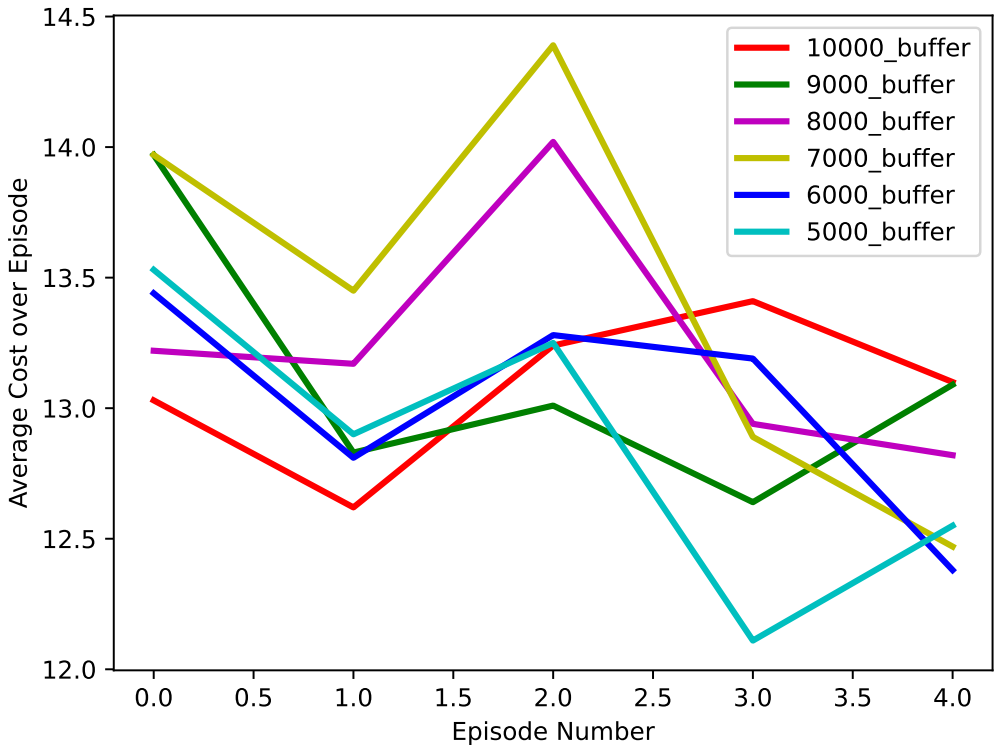

Figure 7-1: Average cost of learned algorithm on Zipfian query sequence on a list of size 50 for different buffer sizes. List remains unchanged and distribution remains unchanged from episode to episode.

8. Lastly, we repeat the experiment M times in order to ensure that the cost and behavior we observe is consistent.

In the first round of experiments, we check to see if the optimal buffer size varies from distribution to distribution. In Figures 7-1, 7-2 and 7-3 we see the average search cost of the learned algorithm for Zipfian, Heavy/Light and Uniform respectively. The buffer size 5000 appears to achieve the lowest average cost.

We also check to see if the story is true for episode-to-episode variations. Fig-

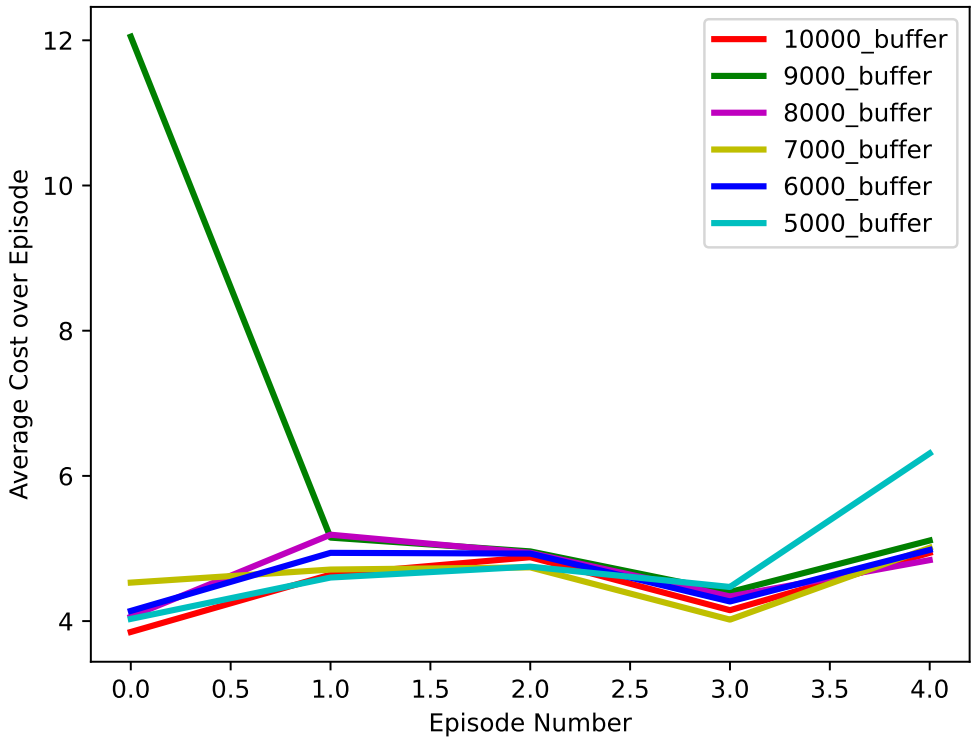

Figure 7-2: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50 for different buffer sizes. 10% heavy and heavy accessed 90%. List remains unchanged and Distribution remains unchanged from episode to episode

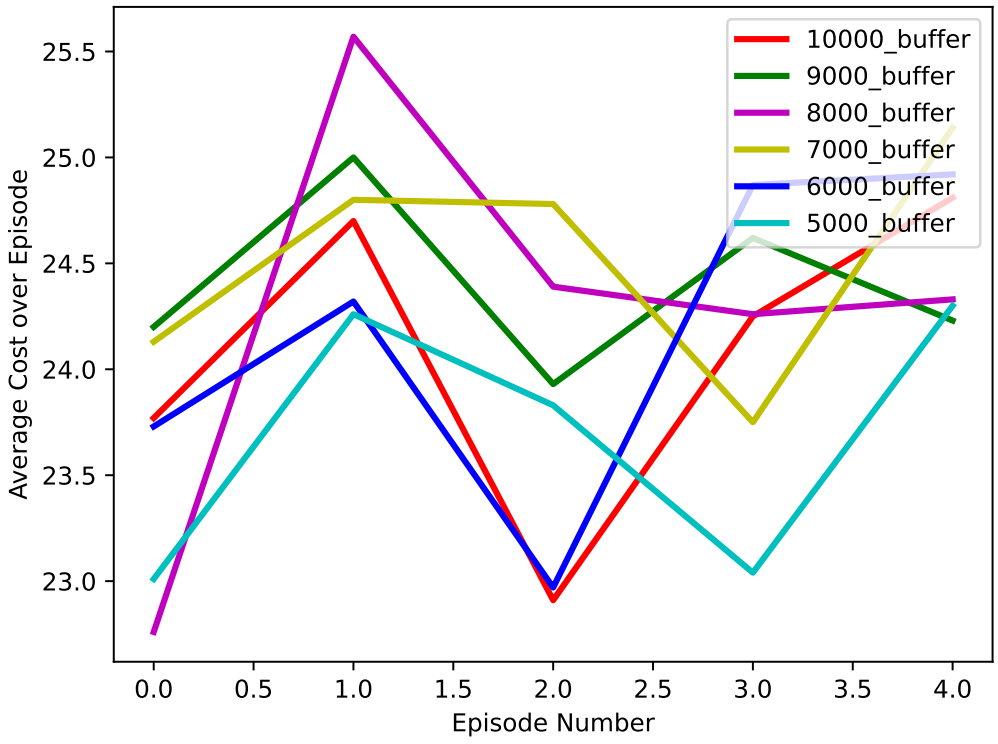

Figure 7-3: Average cost of learned algorithm on Uniform query sequence on a list of size 50. for different buffer sizes. List remains unchanged and Distribution remains unchanged from episode to episode

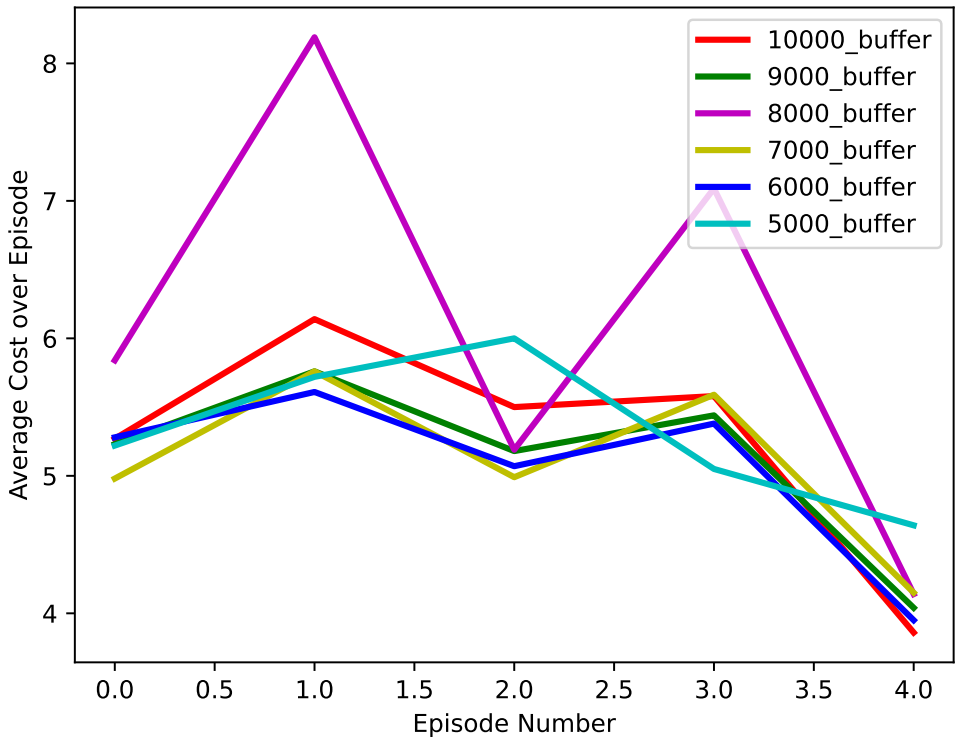

Figure 7-4: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50 for different buffer sizes. 10% heavy and heavy accessed 90%. List remains unchanged but distribution changes from episode to episode.

ures 7-4, 7-5 and 7-6 show different episode-to-episode variations of Heavy/Light query sequences. Figures 7-7, 7-8 and 7-7 show different episode-to-episode variations of Uniform query sequences. Figures 7-10, 7-11 and 7-12 show different episode-to-episode variations of Zipfian query sequences. We still see that the buffer size of 5000 maintains a competitive average cost across all the variations for each type of distribution.

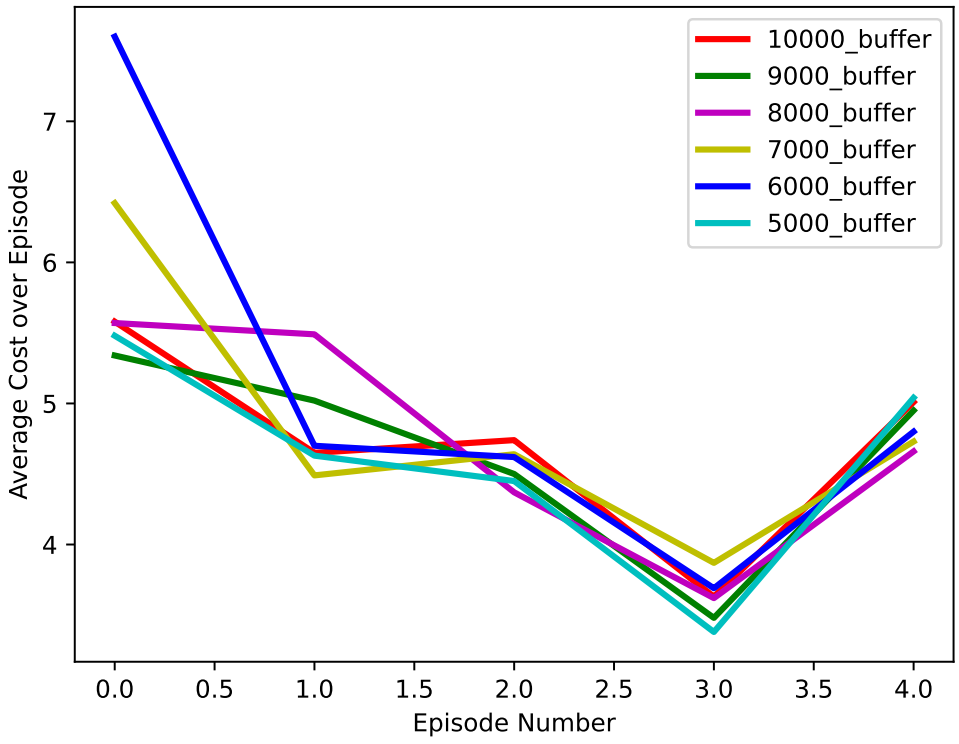

Figure 7-5: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50. for different buffer sizes. 10% heavy and heavy accessed 90%. List changes but distribution remains unchanged from episode to episode

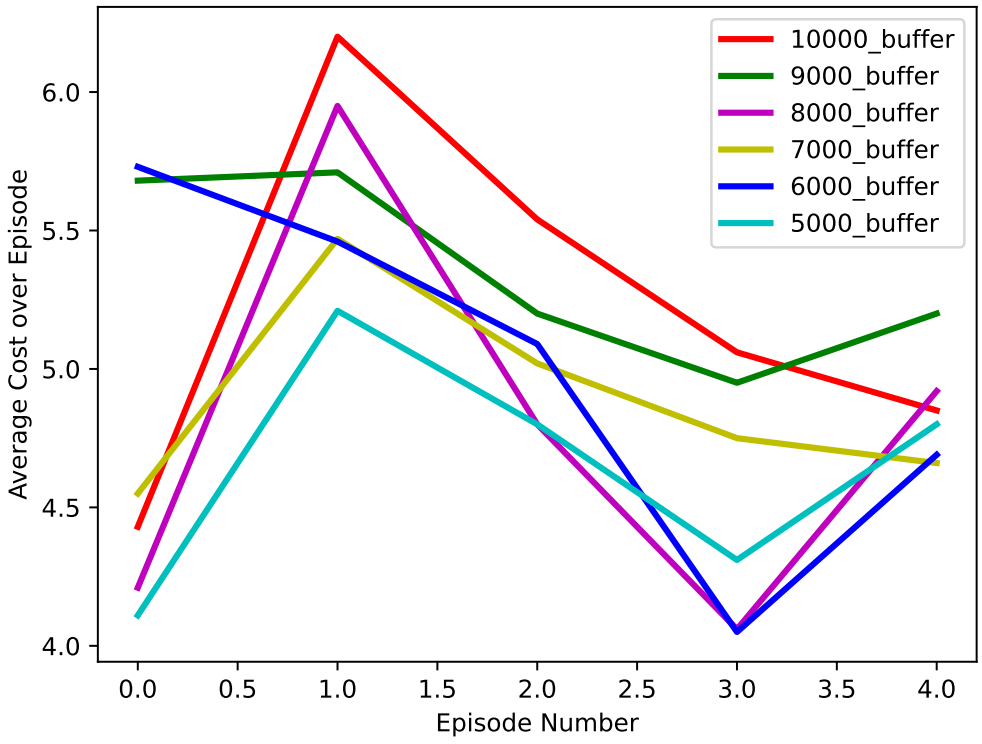

Figure 7-6: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50. for different buffer sizes. 10% heavy and heavy accessed 90%. Both the list and distribution change from episode to episode

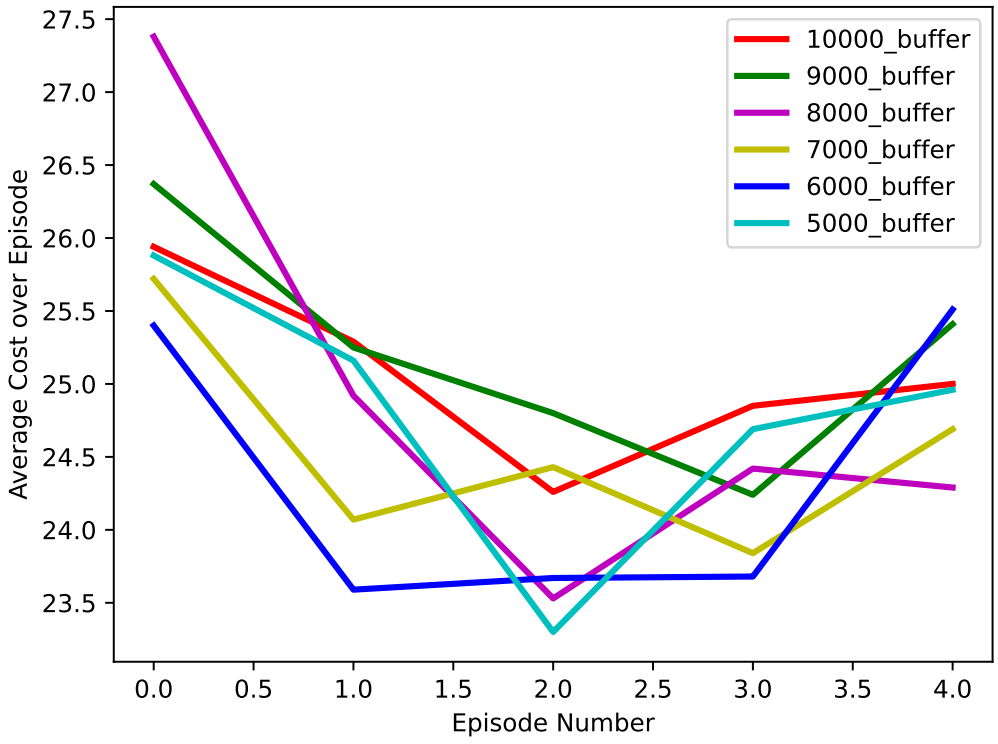

Figure 7-7: Average cost of learned algorithm on Uniform query sequence on a list of size 50. for different buffer sizes. List remains unchanged but distribution changes from episode to episode.

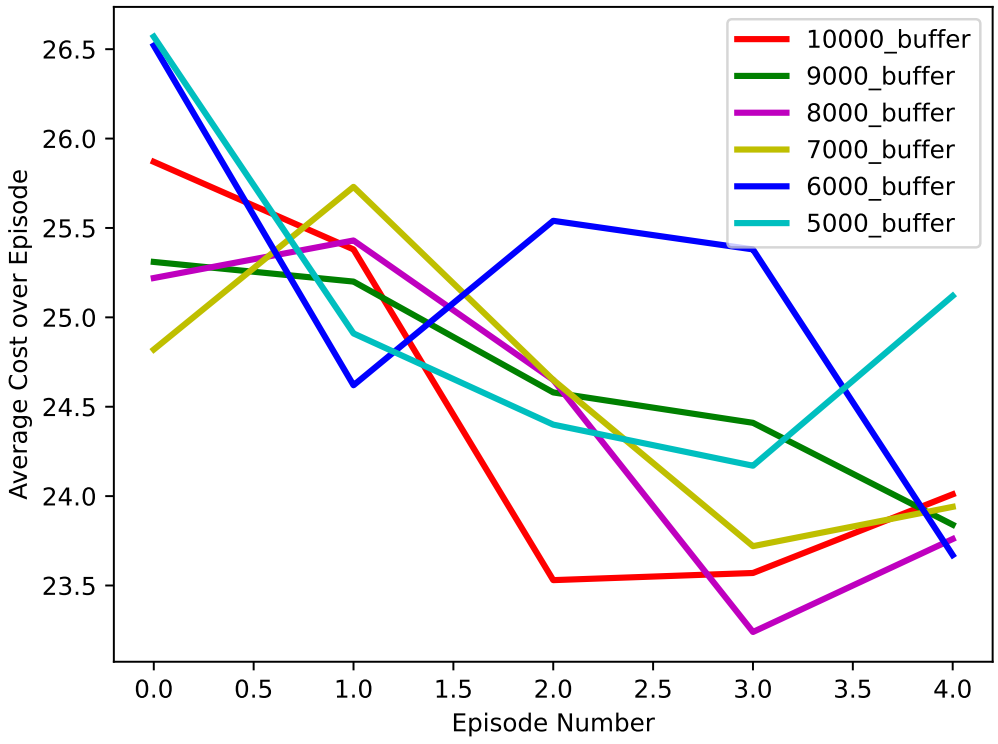

Figure 7-8: Average cost of learned algorithm on Uniform query sequence on a list of size 50. for different buffer sizes. List changes but distribution remains unchanged from episode to episode

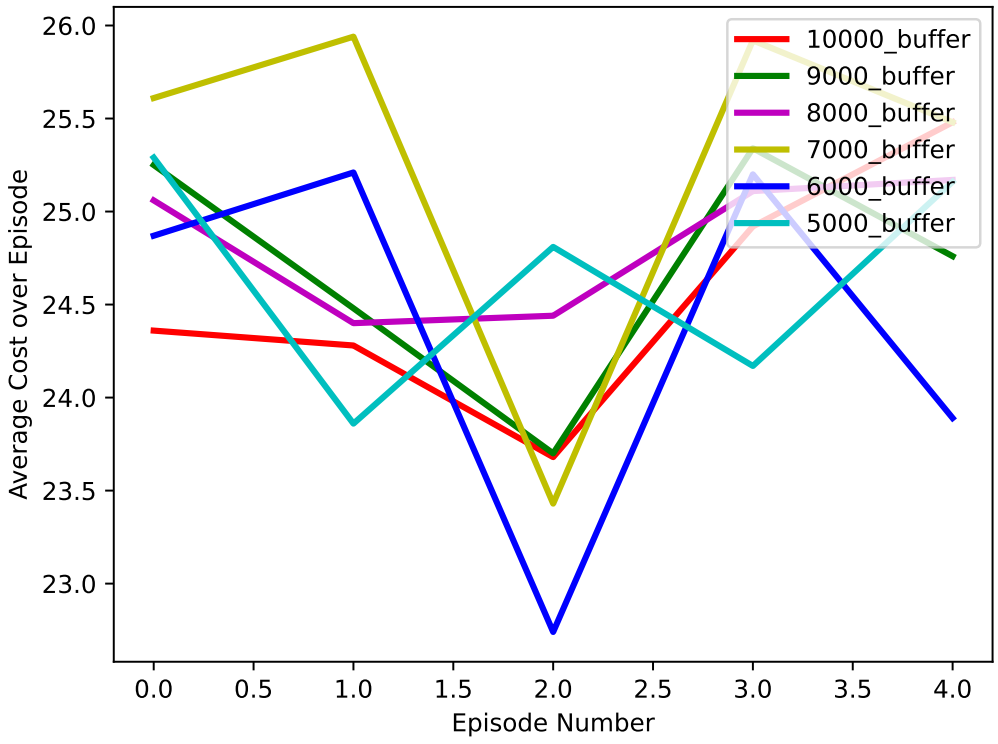

Figure 7-9: Average cost of learned algorithm on Uniform query sequence on a list of size 50. for different buffer sizes. Both the list and distribution change from episode to episode

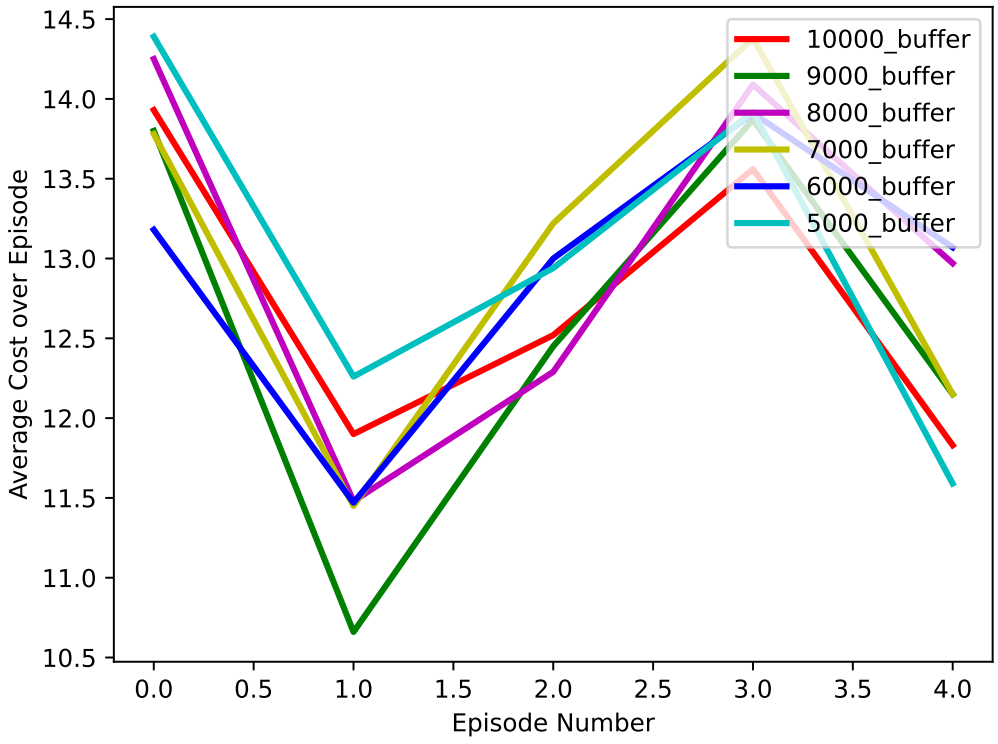

Figure 7-10: Average cost of learned algorithm on Zipfian query sequence on a list of size 50. for different buffer sizes. List remains unchanged but distribution changes from episode to episode.

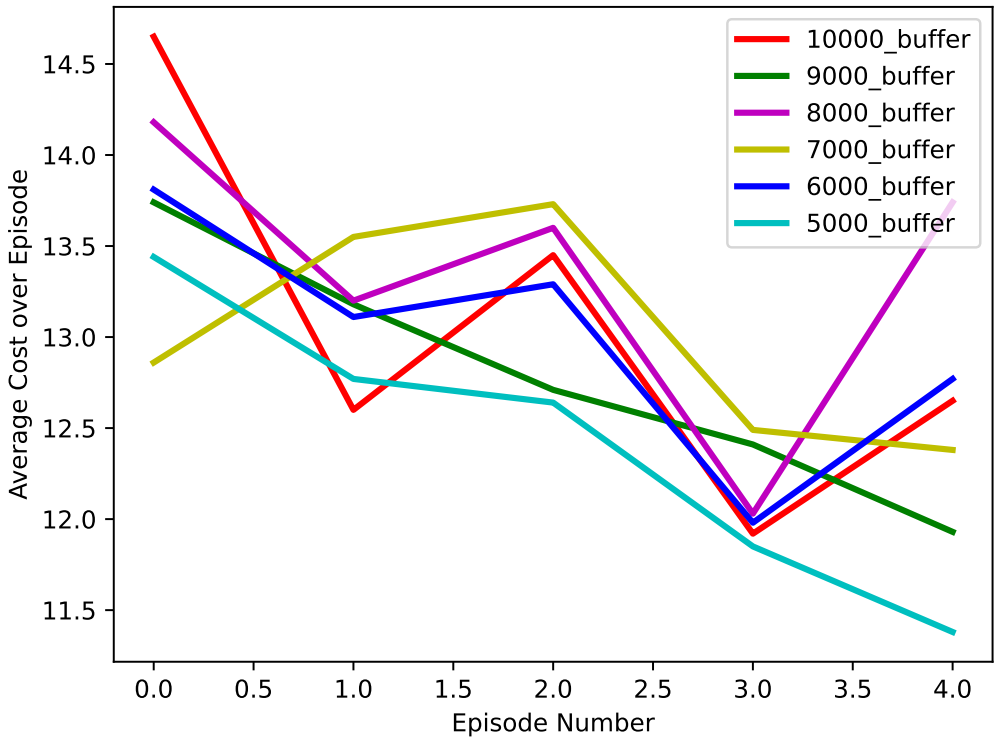

Figure 7-11: Average cost of learned algorithm on Zipfian query sequence on a list of size 50. for different buffer sizes. List changes but distribution remains unchanged from episode to episode

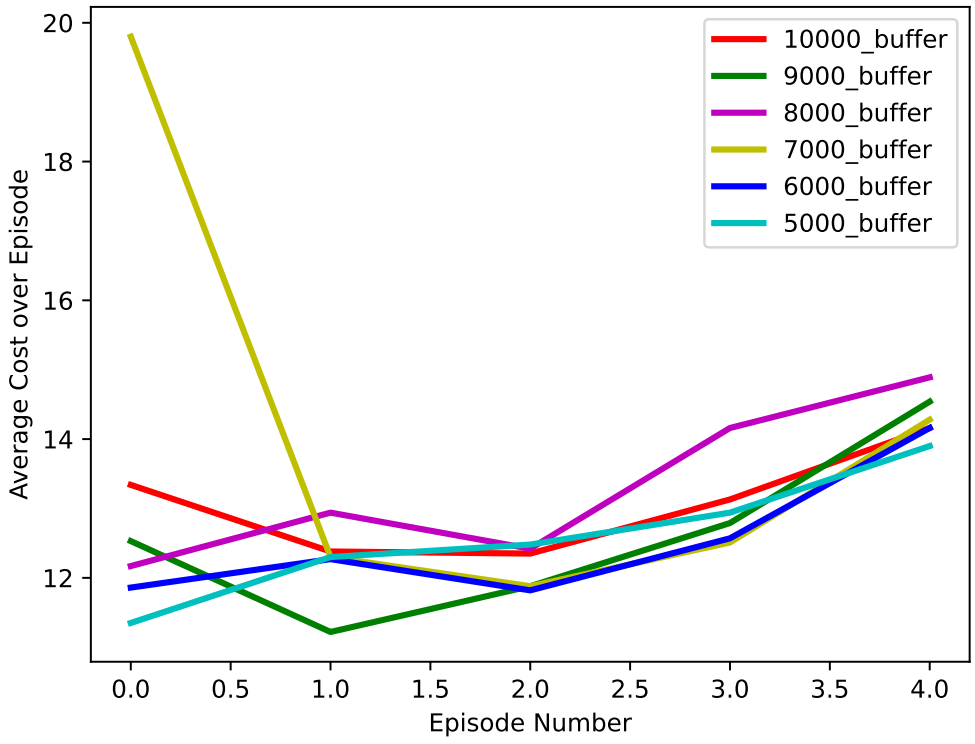

Figure 7-12: Average cost of learned algorithm on Zipfian query sequence on a list of size 50. for different buffer sizes. Both the list and distribution change from episode to episode

7.3 Choosing a State Representation

Out of all the design choices, this is perhaps the most important one. The choice of state representation is crucial to the agent's learning because a poor representation will cause the agent to incorrectly value different states of the environment leading to a poor performing policy/learned algorithm. At the same time, we want to keep the state representation small so we can scale to larger list sizes.
It is clear that the state at time step t, s_t, should capture **both** the state of the list **and** the query at that time step q_t. The different state representations we considered stemmed from the different ways we could choose to represent the state of the list. We explain each one below and in Figures 7-13, 7-14 and 7-15 we use an example to illustrate each state representation[1]:

1. Permutation Matrix: This list representation is much like a standard permutation matrix consisting of only 0s and 1s with each row and column having only a single entry being equal to 1. For a list L, the row of the permutation matrix represents the position of a record in the initial permutation of L and the column represents the position it has been moved to. This list state representation has size $|L|^2$ and the total size of the state representation is $|L|^2 + 1$. The size of the total state space is $|L| \cdot 2^{|L|}$.

2. First Item: The first record in the list L. This list state representation has size 1 and the total size of the state representation is 2. The size of the total state space is $|L|^2$.

3. Preceding Item: Given that q_t references the record in position i, the preceding item list state representation chooses the record $i - 1$ as the list state. If

[1] All size descriptions are in terms of the number of records

$i = 1$, then the preceding item list state chooses the record i itself. This list state representation has size 1 and the total size of the state representation is 2. The size of the total state space is $|L|^2$.

4. Succeeding Item: Given that q_t queries the record in position i, the succeeding item list state representation chooses the record $i + 1$ as the list state. If $i = |L|$, then the succeeding item list state chooses the record i itself. This list state representation has size 1 and the total size of the state representation is 2. The size of the total state space is $|L|^2$.

5. Preceding and Succeeding Item: This list state representation uses both the preceding and succeeding item representation. This list state representation has size 2 and the total size of the state representation is 3. The size of the total state space is $\leq |L|^3$.

6. Current Position: Given that q_t queries the record in position i, the current position list state representation chooses the index i as the list state. This list state representation has size 1 and the total size of the state representation is 2. The size of the total state space is $|L|^2$.

7. First log N: The first $\log |L|$ records of L are the list state representation. This list state representation has size $\log |L|$ and the total size of the state representation is $\log |L| + 1$. The size of the total state space is $|L| \cdot \binom{|L|}{\log |L|}$.

8. Last log N: The last $\log |L|$ records of L are the list state representation. This list state representation has size $\log |L|$ and the total size of the state representation is $\log |L| + 1$. The size of the total state space is $|L| \cdot \binom{|L|}{\log |L|}$.

9. First member of log N: Given a list L, break it up into $\frac{|L|}{\log |L|}$ sections each of

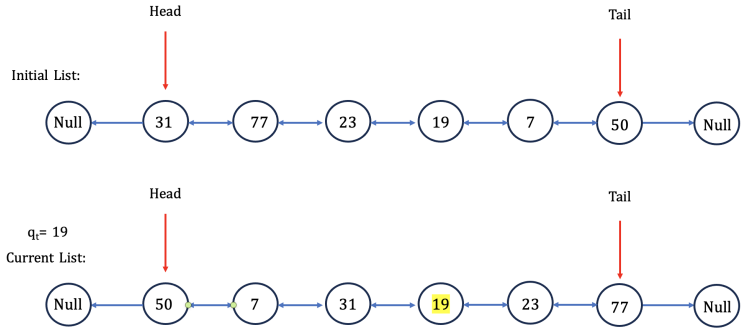

Figure 7-13: List for illustrating different list representations.

size $\log|L|$. The list state representation consists of the first record of each $\log|L|$ chunk. This list state representation has size $\frac{|L|}{\log|L|}$ and the total size of the state representation is $\frac{|L|}{\log|L|}+1$. The size of the total state space is $|L|\cdot\binom{|L|}{\frac{|L|}{\log|L|}}$.

We now look at the average cost of the learned algorithm from each of the state representations. We start as usual by looking at performance across different distributions in Figures 7-16, 7-17 and 7-18. From the results, the current position list representations seems to be the state representation that maintains a competitive performance across distributions. Looking also at performance across different episode to episode variations for both Heavy/Light(Figures 7-19, 7-20 and 7-21) and Zipfian(Figures 7-22, 7-23 and 7-24), the current position list representation remains competitive.

The experiment for comparing each reward function is identical to the process described in section 7.2. The difference is that rather than a different buffer size for each agent, the state representation is different.

List Representation Name	Example
Permutation Matrix	[0, 0, 1, 0, 0, 0 0, 0, 0, 0, 0, 1 0, 0, 0, 0, 1, 0 0, 0, 0, 1, 0, 0 0, 1, 0, 0, 0, 0 1, 0, 0, 0, 0, 0]
First Item	[50]
Preceding Item	[31]
Suceeding Item	[23]
Preceding and Succeeding Items	[31, 23]

Figure 7-14: Illustration of different list representations for list in Figure 7-13.

List Representation Name	Example
First Log N	[50, 7, 31]
Last Log N	[19, 23, 77]
First Member of Log N	[50, 19]
Current Position	[3]

Figure 7-15: Illustration of different list representations for list in Figure 7-13 continued.

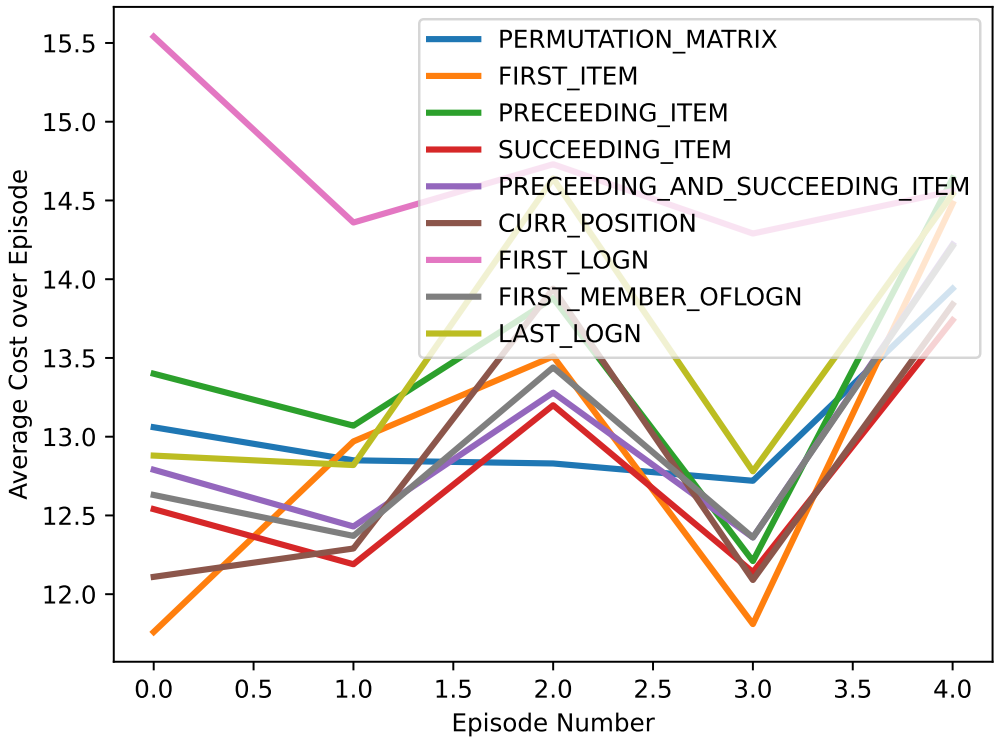

Figure 7-16: Average cost of learned algorithm on Zipfian query sequence on a list of size 50 for different state representations. List remains unchanged and Distribution remains unchanged from episode to episode.

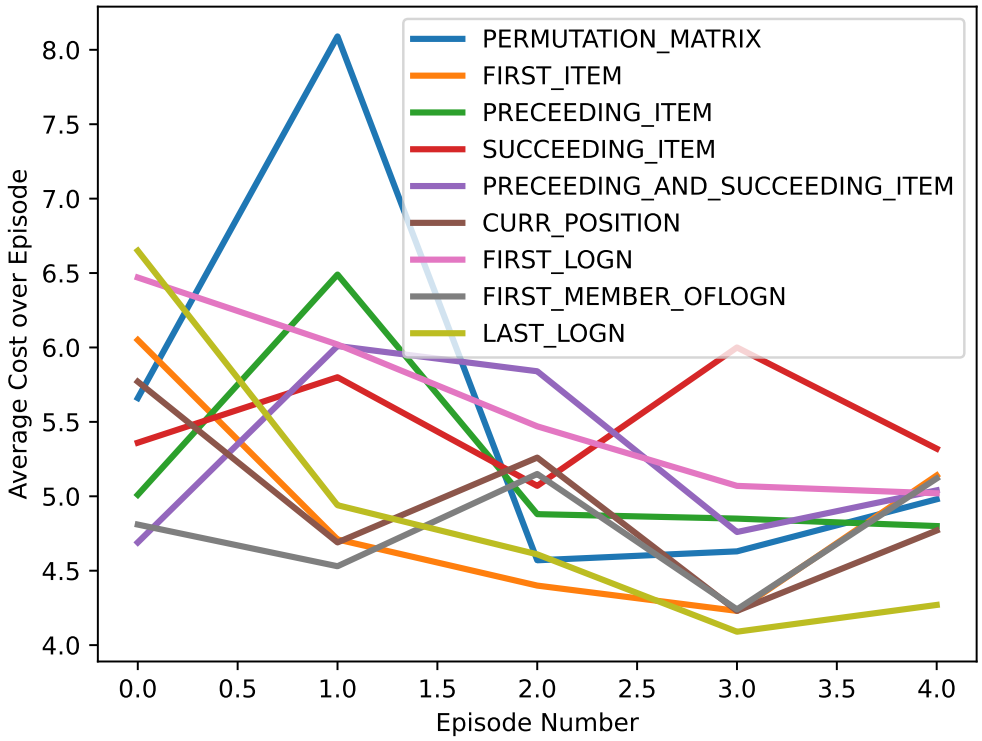

Figure 7-17: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50 for different state representations. 10% heavy items with heavy items making up 90% of the query sequence. List remains unchanged and Distribution remains unchanged from episode to episode

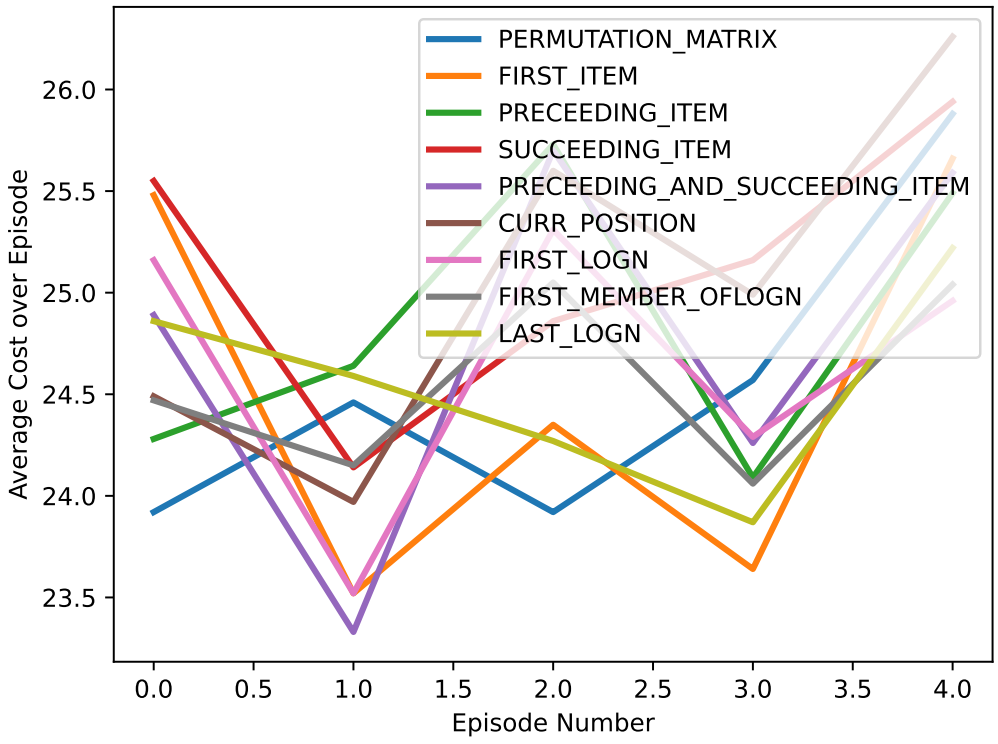

Figure 7-18: Average cost of learned algorithm on Uniform query sequence on a list of size 50 for different state representations. List remains unchanged and Distribution remains unchanged from episode to episode

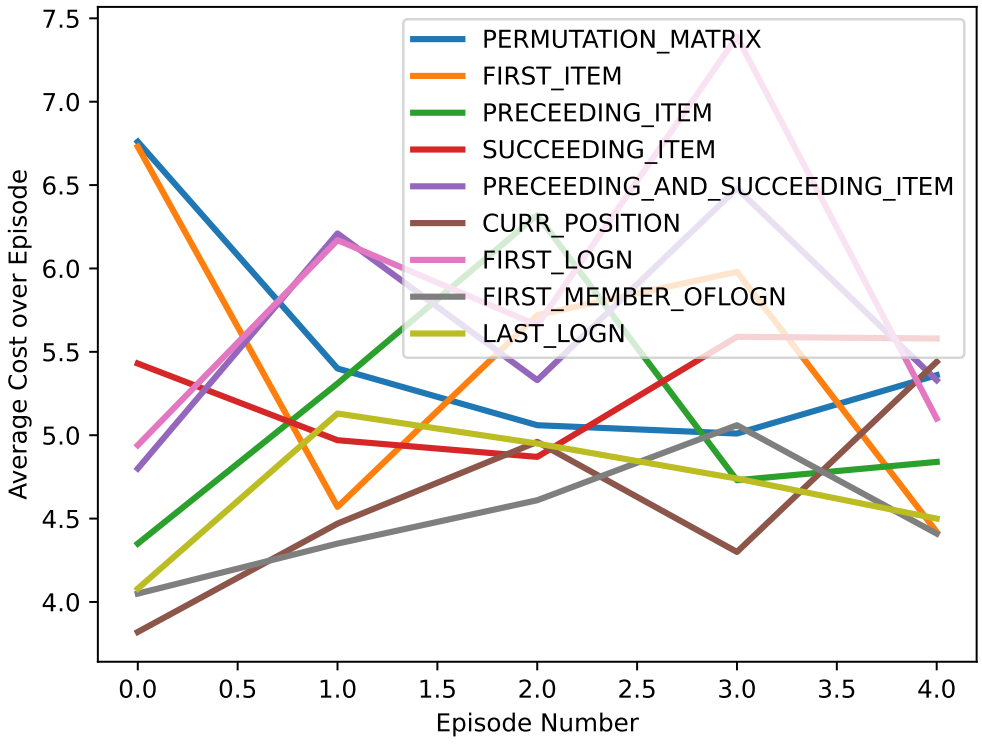

Figure 7-19: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50 for different state representations. List remains unchanged but distribution changes from episode to episode.

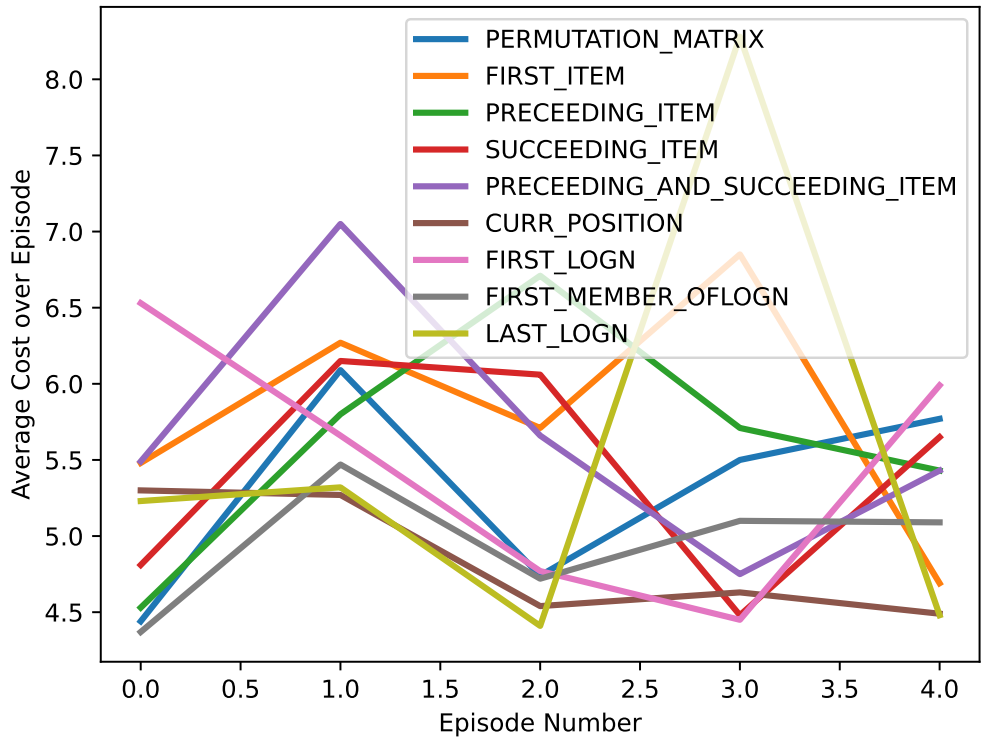

Figure 7-20: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50 for different state representations. List changes but distribution remains unchanged from episode to episode

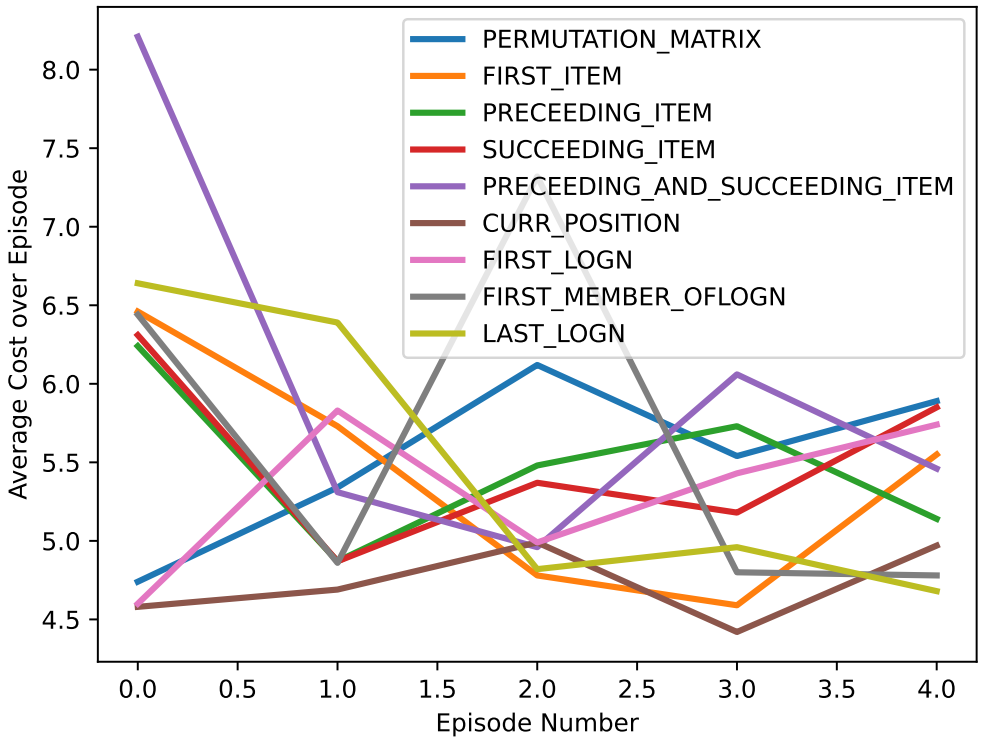

Figure 7-21: Average cost of learned algorithm on Heavy/Light query sequence on a list of size 50 for different state representations. Both the list and distribution change from episode to episode

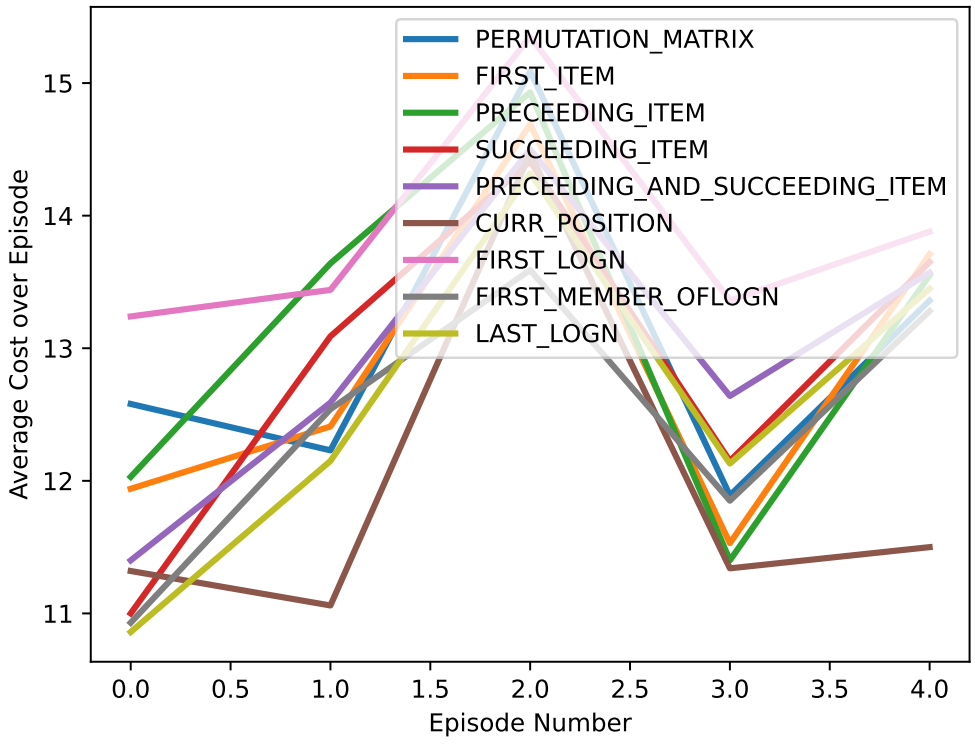

Figure 7-22: Average cost of learned algorithm on Zipfian query sequence on a list of size 50 for different state representations. List remains unchanged but distribution changes from episode to episode.

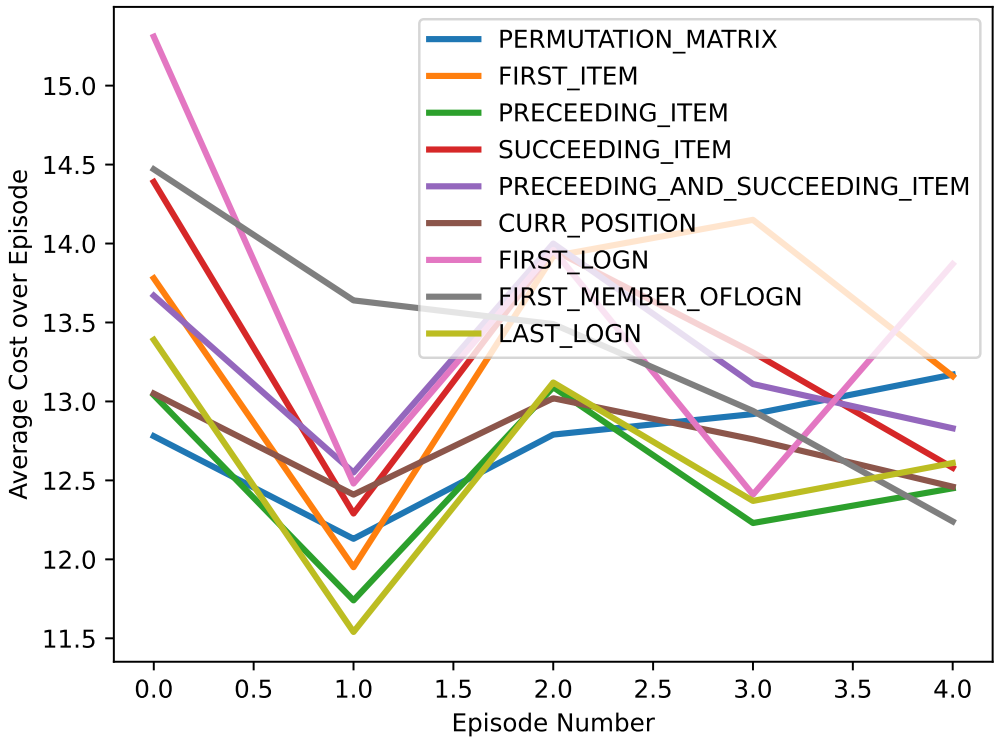

Figure 7-23: Average cost of learned algorithm on Zipfian query sequence on a list of size 50 for different state representations. List changes but distribution remains unchanged from episode to episode

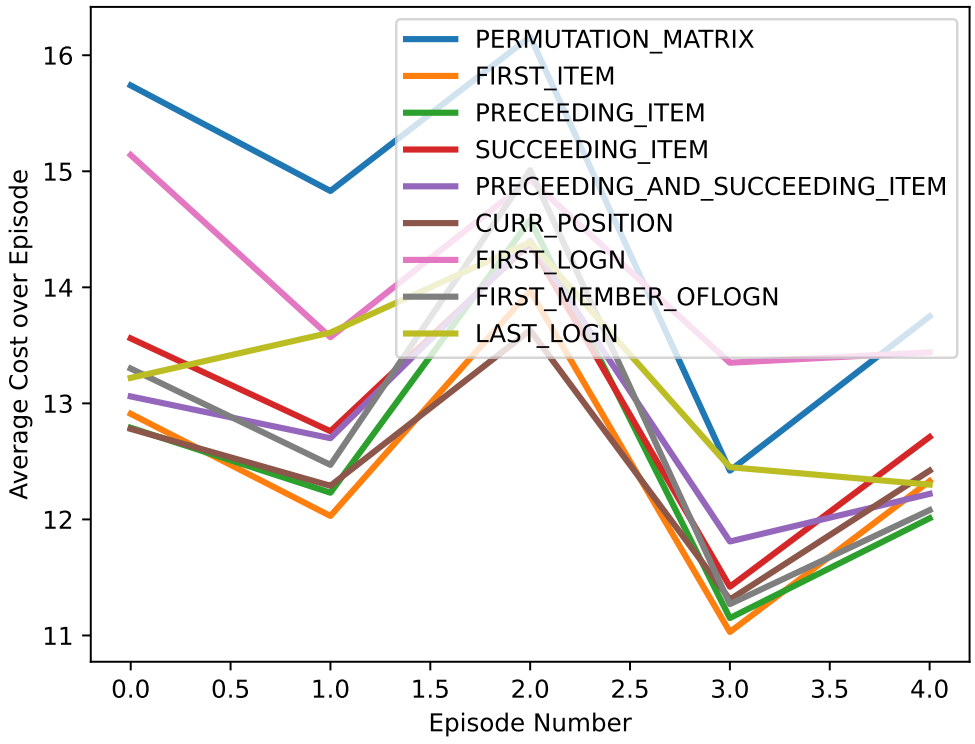

Figure 7-24: Average cost of learned algorithm on Zipfian query sequence on a list of size 50 for different state representations. Both the list and distribution change from episode to episode

7.4 Reward functions and their Impact on the Learned Algorithm

Alongside the state representation, the reward function/signal also has a huge influence on the agent's learned policy. In our experiments, we unsurprisingly observed that when the reward function changes, the reinforcement learning agent's policy changes too. We looked at three different reward functions. In the descriptions below, we consider the reward at time step t where record x is queried, it is found in position i and the agent elects to move it to position j to be:

1. Reward Function One:
$$\begin{cases} -i & j \leq i \\ -j & j > i \end{cases}$$

2. Reward Function Two:
$$\begin{cases} -i & j \leq i \\ -1000 & j > i \end{cases}$$

3. Reward Function Three:
$$\begin{cases} -i & j < i \\ -1000 & j \geq i \end{cases}$$

Denote π_{R_1} as the learned policy of reward function one, π_{R_2} as the learned policy of reward function two and π_{R_3} as the learned policy of reward function three. Since we chose to use reward function one in our design, π_{R_1} has already been analyzed in Chapter 5. We therefore start by discussing the policy map for reward function two across different query sequences: From the policy maps of reward

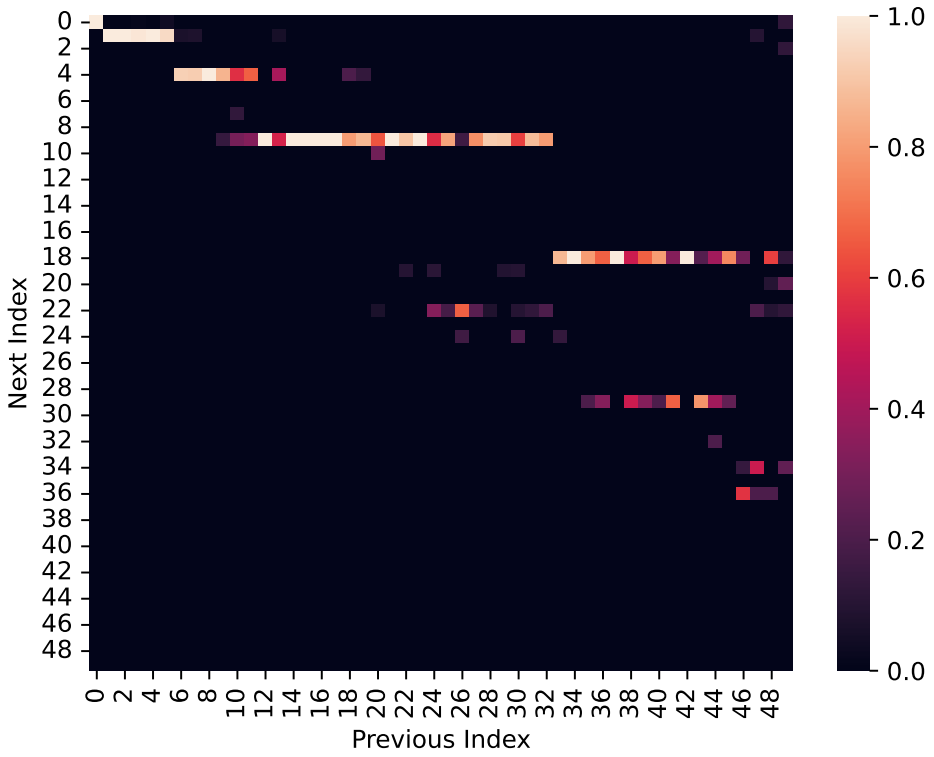

Figure 7-25: Normalized Policy Map of learned algorithm from reward function two for last 20% of Zipfian query sequence on a list of size 50. Both the list and distribution remain unchanged from episode to episode

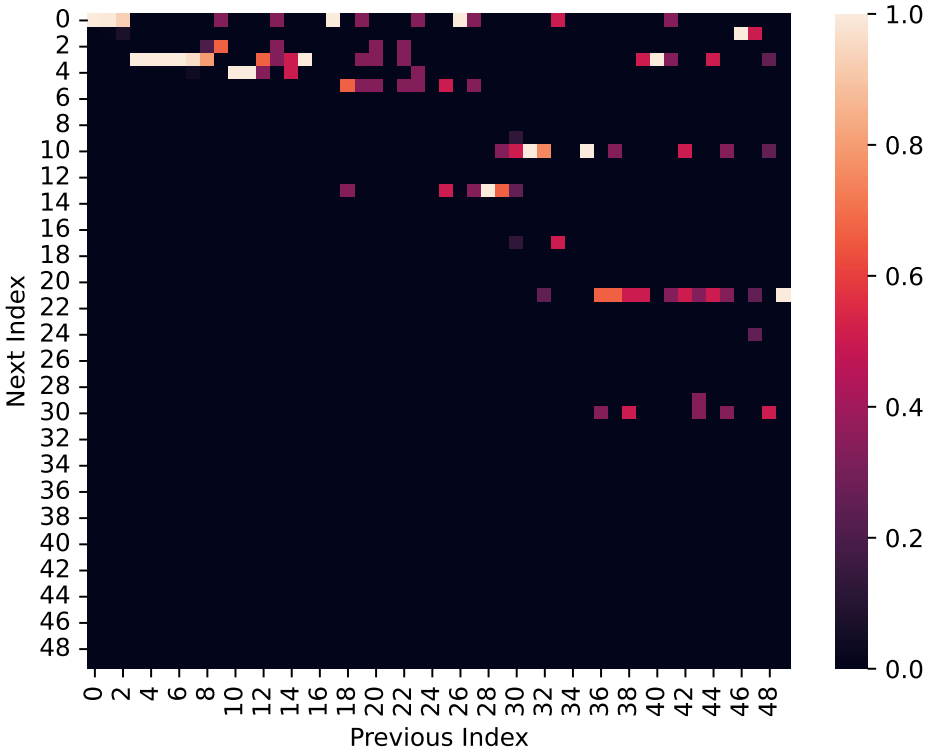

Figure 7-26: Normalized Policy Map of learned algorithm from reward function two for last 20% of Heavy/Light query sequence on a list of size 50. Both the list and distribution remain unchanged from episode to episode

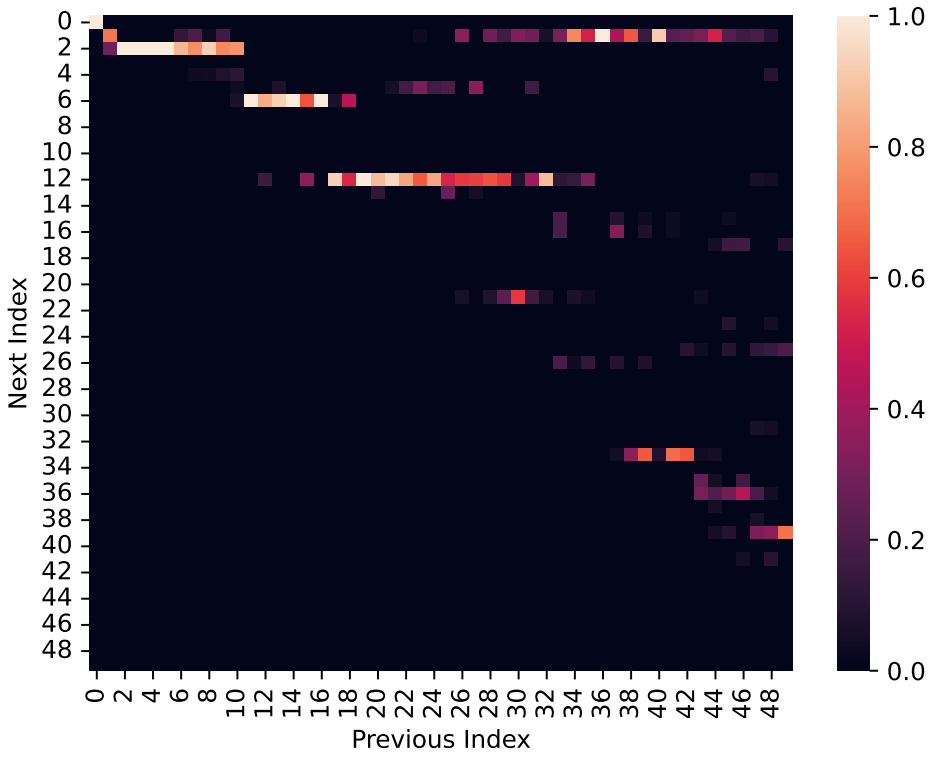

Figure 7-27: Normalized Policy Map of learned algorithm from reward function two for last 20% of Uniform query sequence on a list of size 50. Both the list and distribution remain unchanged from episode to episode

function two shown in Figures 7-25, 7-26 and 7-27 we see that π_{R_2} also displays a similar banded policy as described in Chapter 5. The main difference, is that π_{R_2} does not employ randomized move-to-front for buckets located further from the front of the list as π_{R_1} does. Instead, π_{R_2} follows a move-to-front policy within each of its buckets. This is interesting, because, by changing the cost of moving a record backward, it appears we forced the agent to maintain move-to-front policy in all buckets. However, upon careful inspection of the Q-function values estimated by the neural network, we observed that by choosing a large cost for moving a record backward, the Q-values computed over time tend to approach ∞. For this reason, taking $\max_{a \in A} q(s, a)$ returns the action of moving the item to the front of the list always.

We now turn our attention to the learned algorithm resulting from reward function three(π_{R_3}) shown in Figures 7-28, 7-29 and 7-30. The policy is also banded like in π_{R_2}, except in the policy map, we see that π_{R_3} prefers to move items closer to the front and seldom keeps records in the same position. Again, this is because in reward function three, there is a large penalty associated with keeping a record in the same position once it has been accessed. As in Section 7.3, the experiment for comparing each reward function is identical to the process described in section 7.2. The difference is that rather than a different buffer size for each agent, the reward function is different.

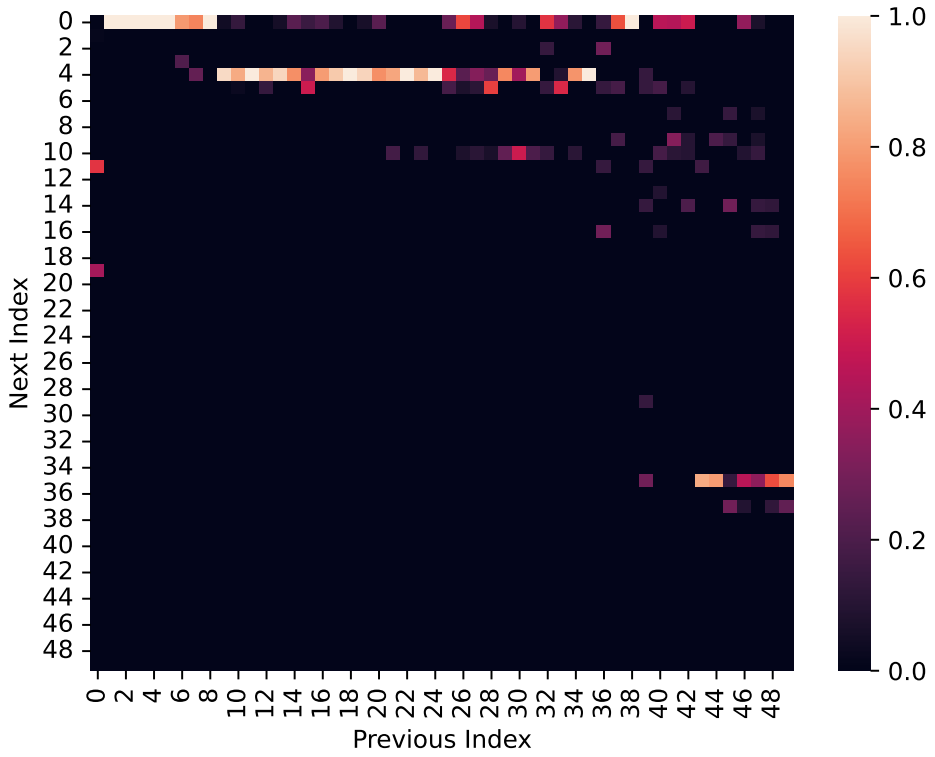

Figure 7-28: Normalized Policy Map of learned algorithm from reward function three for Zipfian query sequence on a list of size 50. Both the list and distribution remain unchanged from episode to episode

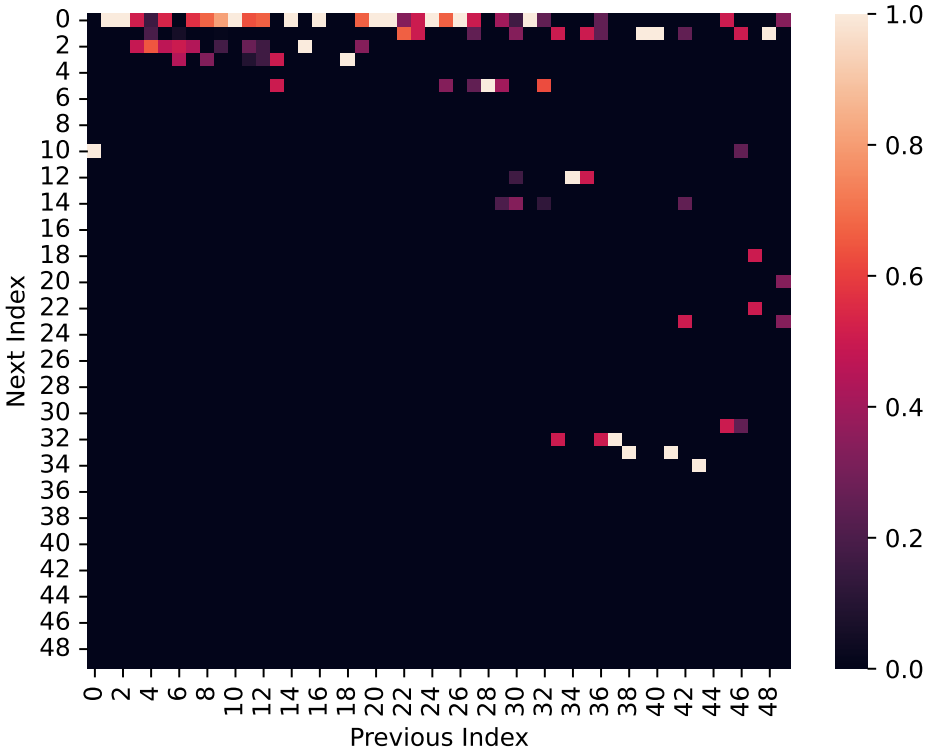

Figure 7-29: Normalized Policy Map of learned algorithm from reward function three for Heavy/Light query sequence on a list of size 50. Both the list and distribution remain unchanged from episode to episode

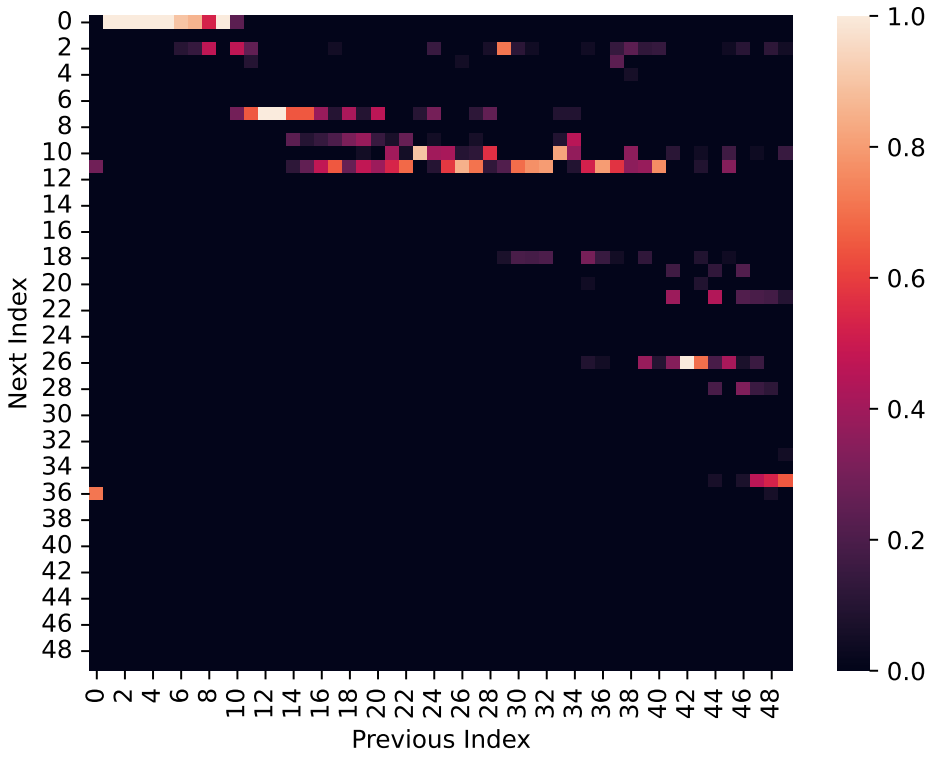

Figure 7-30: Normalized Policy Map of learned algorithm from reward function three for Uniform query sequence on a list of size 50. Both the list and distribution remain unchanged from episode to episode

Chapter 8

Future Work & Conclusion

The learned algorithm, as we saw in the evaluation presented in Chapter 4 holds great promise. We now present potential extensions for the ideas presented in this thesis.

8.1 Extension to other Data Structures

While we presented a learned algorithm for the list update problem, the idea can be extended to other data structures such as the binary search tree. The binary search tree is a good candidate because there are a finite number of actions that can be performed in the binary search tree model(i.e. rotations) and the cost/reward function is easy to define. The challenge, however, lies in developing a good state representation. One can draw inspiration from some of the state representations suggested in Chapter 7. For example, choosing to represent the state of the tree using the root or the first $\log n$ items in the tree in level-order. A learned binary search tree algorithm, like the learned algorithm for list update, could potentially

give us insights into designing novel self-adjusting binary search trees.

8.2 Analysing Novel Algorithms

In Chapter 5, we discussed the banded behaviour of the learned algorithm. We now present two algorithms inspired by this:

- $k-$Banded Move-to-Front: This is a parameterized algorithm where k represents the number of buckets/bands. Each bucket behaves as its own list as described in Chapter 5. Each bucket is maintained with a move-to-front policy. Records may be promoted from higher bucket indices to lower bucket indices and records can be demoted from lower bucket indices to higher bucket indices. When the first record in a bucket i (where $i > 1$) is accessed, it is promoted to the next bucket $i - 1$ by swapping it with the last record in bucket $i - 1$.
- $k, \beta-$Banded Move-to-Front: This algorithm is identical to the first one except when the first record in bucket $i > 1$ is accessed, it is not immediately promoted to the bucket $i - 1$. It is only promoted when that same record has been accessed β times while it was in position one of bucket i.

An area of future work is to compute the competitive ratio of these algorithms.

As was our desired goal, we presented a learned algorithm for the list update problem using reinforcement learning which we evaluated and analyzed. The observations from our analysis led us to prove a competitive ratio for the transposition heuristic for Zipfian distributions. Beyond the insights gleaned from the

learned algorithm, we also have a neural network capable of distinguishing heavy and light records in a stream of query sequences. The contributions of this thesis are only just the beginnings of interesting research directions.

Appendix A

Code

All code for this project can be found at `https://github.com/isabellequaye/MEng` including code for generating the plots and graphs here.

Bibliography

[1] Anders Aamand, Justin Y. Chen, and Piotr Indyk. (Optimal) Online Bipartite Matching with Degree Information. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[2] Anders Aamand, Justin Y Chen, Huy Lê Nguyen, Sandeep Silwal, and Ali Vakilian. Improved frequency estimation algorithms with and without predictions. *arXiv preprint arXiv:2312.07535*, 2023.

[3] M Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*, 55(7):1–38, 2022.

[4] Susanne Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27(3):682–693, 1998.

[5] Susanne Albers, Bernhard Von Stengel, and Ralph Werchner. A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56(3):135–139, 1995.

[6] Keerti Anand, Rong Ge, and Debmalya Panigrahi. Customizing ml predictions for online algorithms. In *International Conference on Machine Learning*, pages 303–313, 2020.

[7] Edward James Anderson, P Nash, and Richard R Weber. A counterexample to a conjecture on optimal list ordering. *Journal of Applied Probability*, 19(3):730–732, 1982.

[8] Spyros Angelopoulos, Christoph Dürr, Shendan Jin, Shahin Kamali, and Marc Renault. Online computation with untrusted advice. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[9] Antonios Antoniadis, Christian Coester, Marek Eliáš, Adam Polak, and Bertrand Simon. Online metric algorithms with untrusted predictions. *ACM Transactions on Algorithms*, 19(2):1–34, 2023.

[10] Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2):740–759, 2020.

[11] Yossi Azar, Stefano Leonardi, and Noam Touitou. Flow time scheduling with uncertain processing time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1070–1080, 2021.

[12] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In *International Conference on Machine Learning*, pages 353–362, 2018.

[13] PJ Burville and JFC Kingman. On a model for storage and search. *Journal of Applied Probability*, 10(3):697–701, 1973.

[14] Justin Y. Chen, Sandeep Silwal, Ali Vakilian, and Fred Zhang. Faster fundamental graph algorithms via learned predictions. In *International Conference on Machine Learning, ICML*, volume 162 of *Proceedings of Machine Learning Research*, pages 3583–3602, 2022.

[15] Ilias Diakonikolas, Vasilis Kontonis, Christos Tzamos, Ali Vakilian, and Nikos Zarifis. Learning online algorithms with distributional advice. In *International Conference on Machine Learning*, pages 2687–2696, 2021.

[16] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. *Advances in neural information processing systems*, 34:10393–10406, 2021.

[17] Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. Learning sublinear-time indexing for nearest neighbor search. *arXiv preprint arXiv:1901.08544*, 2019.

[18] Jon C. Ergun, Zhili Feng, Sandeep Silwal, David P. Woodruff, and Samson Zhou. Learning-augmented k-means clustering. In *10th International Conference on Learning Representations, ICLR*, 2022.

[19] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[20] Sreenivas Gollapudi and Debmalya Panigrahi. Online algorithms for rent-or-buy with expert advice. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2319–2327, 2019.

[21] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[22] Anupam Gupta, Debmalya Panigrahi, Bernardo Subercaseaux, and Kevin Sun. Augmenting online algorithms with ε-accurate predictions. *Advances in Neural Information Processing Systems*, 35:2115–2127, 2022.

[23] WJ Hendricks. The stationary distribution of an interesting markov chain. *Journal of Applied Probability*, 9(1):231–233, 1972.

[24] WJ Hendricks. An extension of a theorem concerning an interesting markov chain. *Journal of Applied Probability*, 10(4):886–890, 1973.

[25] Sungjin Im, Ravi Kumar, Aditya Petety, and Manish Purohit. Parsimonious learning-augmented caching. In *International Conference on Machine Learning*, pages 9588–9601. PMLR, 2022.

[26] Sandy Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.

[27] Sandy Irani, Nick Reingold, Jeffery Westbrook, and Daniel D Sleator. Randomized competitive algorithms for the list update problem. In *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 251–260, 1991.

[28] Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 251–266. Springer, 2013.

[29] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[30] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[31] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504, 2018.

[32] Kin Lam, Ming-Ying Leung, and Man-Keung Siu. Self-organizing files with dependent accesses. *Journal of Applied Probability*, 21(2):343–359, 1984.

[33] Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Online scheduling via learned weights. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1859–1877. SIAM, 2020.

[34] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. In *International Conference on Machine Learning*, pages 3302–3311, 2018.

[35] Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[36] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

[37] John McCabe. On serial files with relocatable records. *Operations Research*, 13(4):609–618, 1965.

[38] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*, pages 464–473, 2018.

[39] Michael Mitzenmacher. Scheduling with predictions and the price of misprediction. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[40] Thy Nguyen, Anamay Chaturvedi, and Huy Le Nguyen. Improved learning-augmented algorithms for k-means and k-medians clustering. 2023.

[41] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ml predictions. In *Advances in Neural Information Processing Systems*, pages 9661–9670, 2018.

[42] Nick Reingold, Jeffery Westbrook, and Daniel D Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.

[43] Ronald Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

[44] Tim Roughgarden. Beyond worst-case analysis. *Communications of the ACM*, 62(3):88–96, 2019.

[45] Geza Schay, Jr and Francis W Dauer. A probabilistic model of a self-organizing file system. *SIAM Journal on Applied Mathematics*, 15(4):874–888, 1967.

[46] Sandeep Silwal, Sara Ahmadian, Andrew Nystrom, Andrew McCallum, Deepak Ramachandran, and Seyed Mehran Kazemi. Kwikbucks: Correlation clustering with cheap-weak and expensive-strong signals. In *The Eleventh International Conference on Learning Representations*, 2023.

[47] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[48] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. Learning to hash for indexing big data - a survey. *Proceedings of the IEEE*, 104(1):34–57, 2016.

[49] Alexander Wei and Fred Zhang. Optimal robustness-consistency trade-offs for learning-augmented online algorithms. *Advances in Neural Information Processing Systems*, 33:8042–8053, 2020.