HIERARCHICAL LIST ORGANIZATION STRATEGIES
FOR NONSTATIONARY ENVIRONMENTS

by

Abdelrahman Amer

A THESIS SUBMITTED TO THE
FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

OTTAWA-CARLETON INSTITUTE FOR COMPUTER SCIENCE
SCHOOL OF COMPUTER SCIENCE
CARLETON UNIVERSITY
OTTAWA, ONTARIO
AUGUST 2004

Copyright © 2004 by Abdelrahman Amer

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97449-9

Our file Notre référence

ISBN: 0-612-97449-9

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

ERREUR DE PAGINATION.

TEXT COMPLETE.

LE TEXTE EST COMPLET.

Abstract

This Thesis examines the problem of self organizing linear search lists, which are lists that react to the queries from an “Environment” by running a heuristic to reorganize the records in order to minimize the search cost. In particular, we are concerned with environments with the locality of reference phenomenon, when the queries exhibit a probabilistic dependence between them.

We introduce a list organization framework that we call *Hierarchical List Organization*, which regards the list as a set of sublists that are manageable in the same way individual records are. A hierarchical list organization involves a reorganization operation on the accessed record level as well as another on the sublist which it belongs to (the record’s *context*). It is beneficial to consider the reorganization of the context together with the accessed record, due to the nature of the Environment, as other records within the context are likely to be accessed in the near future. With the aid of a *partitioning algorithm*, we demonstrate that we can accurately classify the different contexts of the sublist.

Using this framework, we were able to achieve average costs that by far exceeded the best reported in the literature, to our knowledge. We also demonstrate that this framework can be extended to other data structures, by implementing it on doubly linked lists.

Acknowledgements

All praise be to God, the most benevolent, the ever-merciful, without whose grace this work would not have been possible.

I am deeply grateful to Dr. B. John Oommen, not only for his profound scholarship, excellent tutoring, insightful opinions, and continuous help and support, but also for showing me how to conduct proper research. His words "Do not despair, this is research!" will always resonate in my mind.

I would also like to thank my mother, who was the main reason for me to persevere with my research, lest I should disappoint her when she inquires about my progress. I dedicate this work to her and to my late father.

Finally, I would like to thank my dear wife for her support and compassion, and for having to care for two children: our daughter and myself. I would also like to thank my daughter, whose smile can instantly wipe out the darkest of moments.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Thesis Contribution	6
1.2 Thesis Organization	8
2 Survey of Previous Methods	10
2.1 Introduction	10
2.2 Definitions	11
2.2.1 Self-Organizing Data Structures	11
2.2.2 Self-Organizing Lists	11
2.2.3 Cost	12
2.2.4 Convergence	20
2.3 Deterministic Strategies	21
2.3.1 Frequency Count (FC)	21
2.3.2 The Move-To-Front (MTF) Rule	22
2.3.3 The Transposition (TR) Rule	25
2.3.4 Hybrid MTF and TR Algorithms	29

2.3.5	Swap-With-Parent (SWP) and Move-To-Parent (MTP)	30
2.3.6	Batched Algorithms	31
2.3.7	TS(0)	33
2.3.8	MRI(m) and PRI(m)	34
2.3.9	SBR(α) and SBD(k)	35
2.3.10	Move-To-Rear (MTR) Algorithms	36
2.4	Probabilistic Strategies	39
2.4.1	Probabilistic MTF and MTR	39
2.4.2	Competitive Randomized Algorithms	40
2.5	Other Algorithms	44
2.6	Nonstationary Environments	45
2.7	Conclusion	48
3	Managing Lists Through Sublist Manipulation	50
3.1	Introduction	50
3.2	Models of Dependence	51
3.2.1	Markovian Switching Environment (MSE)	52
3.2.2	Periodic Switching Environment (PSE)	53
3.3	Motivation for Sublist Manipulation	54
3.3.1	Alternate Ways to Manage Lists	54
3.3.2	Working Sets	55
3.3.3	Managing Sublists	56
3.4	Sublist Manipulation Algorithms	59
3.4.1	The MTF-MTF Algorithm	59
3.4.2	Discussion	62
4	Capturing Dependence	64
4.1	Clustering	64

4.2	Object Partitioning Problem	66
4.2.1	The Basic Adaptive Method (BAM)	68
4.2.2	FSSA-based solutions for the OPP	69
4.2.3	The Object Migrating Automaton (OMA)	76
4.2.4	The Stochastic Move Automaton (SMA)	82
4.2.5	VSSA-based solution for the OPP	85
4.3	Conclusion	89
5	Hierarchical List Organization Strategies	90
5.1	Query Generators	90
5.2	OMA-Based Algorithms	92
5.2.1	MTF-MTF-OMA and TR-MTF-OMA	93
5.2.2	Periodic Variations of the MTF-MTF-OMA and the TR-MTF-OMA	102
5.3	ML_{RP} -Based Algorithms	105
5.4	Conclusion	106
6	Strategies for Doubly Linked Lists	108
6.1	Doubly Linked Lists	108
6.1.1	Expected Search Cost	109
6.1.2	Research on Self-Organizing DLLs	110
6.2	Models of Dependence	112
6.3	Hierarchical List Organization Algorithms in DLLs	114
6.3.1	Directionally Biased Environment	114
6.3.2	Dually Monotonic Environment	118
6.4	Conclusion	120

7 Summary and Conclusion	121
7.1 Future Work	125
Bibliography	127
A Results for Singly Linked List Algorithms	132
B Results for Doubly Linked List Algorithms	139

List of Tables

2.1	Results by Dong [11] for the asymptotic cost for different algorithms .	31
2.2	Results by Dong [11] for the convergence for different algorithms . . .	32
2.3	Results by Bachrach <i>et al.</i> [6] for the asymptotic cost of different algorithms	44
3.1	Experimental results for the asymptotic cost of the MTF-MTF and TR-MTF	62
A.1	Singly linked list algorithms with varying number of sublists 1	133
A.2	Singly linked list algorithms with varying number of sublists 2	134
A.3	Singly linked list algorithms with varying number of sublists in a periodic environment 1	135
A.4	Singly linked list algorithms with varying number of sublists in a periodic environment 2	136
A.5	Singly linked list algorithms with varying levels of locality of reference	1137
A.6	Singly linked list algorithms with varying levels of locality of reference	2138
B.1	Doubly linked list undirected-mapping algorithms with varying probability of an access from the right end	140
B.2	Doubly linked list directed-mapping algorithms with varying probability of an access from the right end	141

B.3	Doubly linked list directed-mapping algorithms in a dually monotonic environment	142
-----	---	-----

List of Figures

2.1	Markov chain diagram for 3-element list using MTF	15
2.2	Collapsed Markov chain diagram for 3-element list using MTF	17
2.3	Markov chain diagram for 3-element list using TR	18
2.4	Asymptotic cost for algorithms with different locality of reference degrees from Bachrach <i>et al.</i> [6]	47
3.1	Lists viewed as unordered heaps for the SWP and MTP rules	54
3.2	Lists viewed as a series of doubly exponential trees	56
3.3	Illustration of the MTF-MTF and the TR-MTF algorithms	57
4.1	Learning automaton's interaction with the environment	70
4.2	The two-action Tsetlin automaton.	73
4.3	The two-action Krinsky automaton.	73
4.4	Transition rules for two-action OMA	78
5.1	Asymptotic cost for the MTF-MTF-OMA for different numbers of sublists	96
5.2	Asymptotic cost for hierarchical algorithms with different degrees of locality of reference	98
5.3	Rate of change in amortized cost for hierarchical list organization algorithms	99

5.4	Asymptotic cost for the MTF-MTF-OMA for different numbers of sublists in a periodic environment	100
5.5	Asymptotic cost of periodic variations of hierarchical algorithms for different numbers of sublists in a periodic environment	103
6.1	Asymptotic cost for undirected mapping algorithms	116
6.2	Asymptotic cost for directed mapping algorithms	117
6.3	MTF-MTF-OMA (undirected mapping) versus MTE-MTE-OMA (directed mapping)	118
6.4	Directed mapping algorithms with different degrees of locality of reference	119

Chapter 1

Introduction

A fundamental branch of Computer Science is the study of data structures, or how data can be logically organized within a system for storage and retrieval. The core of this branch deals with *lists*, which represent such data merely as a string of consecutive records, strung together through pointers. Because of their simplicity and intuitiveness, the list is perhaps one of the oldest yet most commonly used data structure to date.

For data retrieval from a list of n elements, a *linear* or *sequential search* is performed. Beginning from the list's first record or *head*, each record is probed in order, by following the pointers, until the desired element is found. The time complexity of such search is $O(n)$. If the list's records are uniformly distributed, the average search would cost $n/2$ probes. One way to significantly reduce this cost is by making the list *self-organizing*, such that it dynamically adapts to the query distribution to minimize the search cost, by running a reorganization algorithm. It can be shown that the best ordering of the records would be in terms of the descending order of their respective access probabilities. As such access probabilities are rarely known *a priori*, one has to resort to a heuristic to approximate an optimal behaviour. A *self-organizing linear search list* is therefore a list that runs a reorganization heuristic

with each access query in the hope of optimizing the linear search cost.

Research on self-organizing linear search lists began with McCabe's work in 1965 [23]. He introduced two such heuristics: the move-to-front rule (MTF), which moves the queried record to the front of the list, and the transposition rule (TR), which swaps the queried record with its predecessor. The MTF is characterized by quick convergence rates and the ability to quickly adapt to changes in the Environment. The TR, on the other hand, is much more reserved and is more likely to find a better asymptotic arrangement for the list than the MTF, through its "tiny" conservative changes. However, its convergence rate and ability to respond to changes in the Environment are poor.

In essence, a lot of the research that followed McCabe's work was trying to combine the two algorithms, to benefit from the speed of the MTF and the accuracy of the TR. In this regard we mention Bitner's algorithm [8], that runs MTF initially then TR for fine-tuning, Rivest's move-ahead- k [34], as well as some other variations by Tenenbaum and Nemes [39]. Other "batched" methods are modifiers to be used in conjunction with a reorganization heuristic, such as the move-every- k th-access due to McCabe [23], the k -in-a-row by Kan and Ross [19], and the k -in-a-batch by Gonnet *et al.* [12]. As well, there has been a host of other algorithms suggested by Schulz [36] and Bachrach *et al.* [6]. Again, the goal of such algorithms was to limit the aggressiveness of the MTF or increase the accuracy of the TR.

One interesting such attempt was introduced by Oommen and Dong [27] through the swap-with-parent (SWP) and move-to-parent (MTP) algorithms. These algorithms regard the list as a heap with no ordering constraints among its elements. An accessed record would be moved to (or swapped with) its parent, which is the half way point to the list's head. Thus, the algorithms took into account the relativity of the list's length, a point overlooked by most of the previous literature. SWP and MTP were empirically shown to be significantly faster than the TR in converging to

the asymptotic state, while at the same time providing better asymptotic cost than the MTF.

Another family of list organization algorithms were introduced by Oommen *et al.* [28, 29, 32] that move the access record to the *rear* of the list. Although this seems to run against intuition, Oommen *et al.* showed that this can result in optimal ordering. Consider, for example, the deterministic linear space move-to-rear algorithm [29]. In this algorithm, an accessed record is moved to the rear of the list after it had been accessed k times. Once a record is moved, it is never moved again. Assuming that the access probabilities are time invariant, one can see that this essentially organizes the records in the optimal ordering with a large enough k .

Probabilistic (or randomized) algorithms are another means to reorganize lists that, generally speaking, try to move the accessed record in a more conservative way than the MTF. Some examples include SPLIT by Irani [18], BIT, COUNTER(s, S), and RANDOM-RESET(s, D) by Reingold *et al.* [33] and TS(p) by Albers [1].

In 2002, Bachrach *et al.* [6] performed a rather extensive experimental test on most of the algorithms available in the literature and showed that, in terms of asymptotic cost, no algorithm (deterministic or probabilistic) outperformed the TR except for the Frequency Count (FC), which maintains an access counter for every record and arranges the records in descending order of the frequency of access. Of course, the TR has the advantage of being memoryless (as opposed to requiring linear memory for the FC) as well as having constant reorganization time.

An interesting behaviour of the algorithms manifests when the Environment has the property of *locality of reference*¹, which means that the number of distinct queries in a small part of the access sequence is considerably smaller than the number of distinct queries in the entire access sequence, because of probabilistic dependence

¹In this Thesis we will only deal with *spacial* locality of reference, where a record is more likely to be accessed if another one “near” it was recently accessed.

among the access queries. In other words, during a given time interval, only a subset of the query set is accessed on the list, and then this subset changes with time. This property is also sometimes referred to as dependent accesses. Examples of the locality of reference phenomenon are indeed abundant in computer science, such as in program execution, paging and caching, memory management, file systems, and database systems.

To illustrate, consider this example from Hester and Hirschberg [15]. A list of 26 records (represented by the English alphabet) has a uniformly distributed access probability. Under the access sequence $\langle a, \dots, z, a, \dots, z, \dots, a, \dots, z \rangle$, the MTF heuristic would take 26 probes to reach the requested record, except for the first access. However, under the access sequence $\langle a, \dots, a, b, \dots, b, \dots, z, \dots, z \rangle$, it would take only one probe to reach the requested record, except for the first access. Despite the fact that both distributions have the same “offline” probability distribution (i.e., when considered collectively after the experiment finishes), the locality of reference made the difference in search cost dramatic.

Bachrach *et al.* [6] have empirically shown that the MTF outperforms all other algorithms they tested in such environment, as the dependence factor increases (See Figure 2.4). A lot of other researchers also agree with this conclusion (e.g., [9]). To our knowledge, no algorithm has been reported in the literature that has better performance than the MTF in environments with higher dependence degrees.

One common approach to algorithms dealing with environments possessing such a property is to minimize the access cost for the required resource as well as its surrounding “context,” since the odds that a nearby element will soon be requested are high. In the context of self-organizing lists, this is how we first researched the idea of moving some of the elements around the accessed record as well as the record itself, essentially reorganizing a *sublist* that the accessed record belongs to, within the list. This is especially effective in lists since the reorganization cost for a sublist

is the same as that of a single record (by updating few pointers), without incurring additional cost, provided that the sublist boundaries are known. We call such an approach a *Hierarchical List Organization*.

For example, consider the hierarchical list organization scheme MTF-MTF. In this scheme, we divide the list into a number of sublists k , of length m each. When a requested record is found, it is moved to the front of the sublist it belongs to, and then the entire sublist is moved to the front of the list. If the sublist contains records within the current Environment's context, we may just be lucky enough to have minimized the access costs for a host of forthcoming requests until the Environment's context changes. In the same time, we are bringing records that are more frequently accessed *within this context* to the head of the list, thereby reducing the cost even more.

What remains is to ensure that the sublists actually reflect the Environment's various contexts. It would be of no benefit for us to move a number of records to the beginning of the list if they are not likely to be accessed in the near future. In the literature, this is known as the *Object Partitioning Problem*. To achieve this goal, we have enlisted the help of a *dependence capturing* or *partitioning* algorithm, alongside with the hierarchical reorganization algorithm. This is done by running the current queried key (at time t) with the last queried key (at time $t-1$) against the partitioning algorithm. If the two elements match the algorithm's current partitioning, we are rewarded and the hierarchical reorganization takes place. Otherwise, we are penalized and instead of performing a hierarchical reorganization, we move the elements between sublists to achieve better partitioning, as dictated by the partitioning algorithm. We used two such algorithms, the Object Migration Automaton (OMA), and the Modified Linear Reward-Penalty reinforcement rule (ML_{RP}). Both algorithms are based on the field of learning automata, which we explain in Chapter 4. The resulting hierarchical, dependence capturing algorithm would (for example) be denoted by MTF-MTF-OMA.

The partitioning algorithm does not come at no cost. The space complexity for the OMA, for example, is $O(n)$, and the time complexity is $O(m)$ in the case of a penalty. However, after convergence, the time complexity is almost always constant. Furthermore, a space requirement of $O(n)$ is not unfamiliar in the literature; indeed, the majority of the algorithms discussed in Chapter 2 require an additional memory requirement of $O(n)$, with the obvious exception of the memoryless MTF and TR.

In all brevity, we also add that all these principles can be extended to any other data structure. To justify this, we have also demonstrated their applicability for doubly linked lists.

A note about levels of hierarchy

Throughout the body of this Thesis, we only consider two levels whenever we are designing hierarchical list organization strategies, as described above. The term “hierarchical”, however, implies that the depth of hierarchy is not limited by a constant (say, two), but by the size of the list. In other words, the reorganization step could potentially be performed recursively until there is only a “trivial” list to be reorganized. We believe, however, that such a nested “recursive” reorganization may not be meaningful unless the Environment itself exhibits nested locality of reference. Thus, although we merely advocate the two-level hierarchy, there is no reason why a higher level hierarchy is not utilized for a more general setting.

1.1 Thesis Contribution

In this Thesis, we have introduced a framework for self-organizing data structures that we believe is more powerful than any other reported framework in the context of dependent environments. Denoted in the form X-Y-Z, an algorithm would employ the hierarchical operation X-Y using the partitioning algorithm Z. Although

the only family of data structures studied in this Thesis involve the linked list and the hierarchical operations are chiefly permutations of the MTF and the TR, many other algorithms and data structures can be explored and experimented with, as the framework is independent of the underlying algorithm specifics.

We used two models to design an environment with dependent accesses. In a *Markovian* model, the Environment stays in the current context with probability α . In a *periodic* environment, the Environment stays in a given context for precisely T queries, and then moves on to another context. It does not cycle back to the first context except after precisely $(k - 1)T$ queries.

We have shown that in a singly linked list setting with a Markovian environment with higher values of α , both the MTF-MTF-OMA and TR-MTF-OMA have excelled all other algorithms in the asymptotic and amortized cost. For illustration, see Figure 5.2, which shows a clear advantage for hierarchical algorithms over MTF, the best performing algorithm reported by Bachrach *et al.* [6] (See Figure 2.4.) By using the $MLRP$ as the dependence capturing algorithm, we were able to achieve much faster convergence rates, in addition to the impressive asymptotic cost. As the MTF is widely acknowledged to be the best strategy in dependent environments, we believe that our algorithms therefore yield better performance than all other algorithms reported in the literature.

We also devised a variation of the hierarchical algorithms for periodic environments, both when the period is known *a priori* and when it is not. For known periods, we simply move the first sublist to the rear of the list whenever the period T expires. For unknown periods, we move the first sublist to the rear whenever we detect that two successive queries are not dependent (i.e., whenever we are penalized by the partitioning algorithm), since this is an indication that the period has expired. We have found out that the MTF-MTF-OMA-Periodic is *ideal* in periodic environments, although it performs rather poorly in Markovian environments.

The MTF-MTF-OMA-Unknown_Period, on the other hand, provided excellent performance in both kinds of environments, and is therefore a very good candidate for use when the Environment's attributes are unknown.

We have also tested our algorithms with doubly linked lists. These lists have the property of being traversable from both left and right ends, and therefore a query can come from either end. The move-to-end (MTE) [22] is a generalization of the MTF due to Matthews *et al.*, where the moving operation is done with respect to the direction of the access query, while the SWAP is the similar generalization of the TR, due to Ng and Oommen [25]. We have shown that in environments where records are more likely to be accessed from one end than the other, the MTF-MTF-OMA and TR-MTF-OMA still provided the best results, outperforming all other algorithms. In other environments, however, we obtained better results for the MTE-SWAP-OMA and the SWAP-SWAP-OMA, although for certain distributions the simple SWAP provided better results.

We believe that we are reporting results that outperform all others reported in the literature, to our knowledge. However, we submit that the most salient contribution of the Thesis is that we have viewed list reorganization in a truly novel way that has not been explored before, by considering the list as a pool of sublists that could be manipulated just as the individual records are.

1.2 Thesis Organization

The following Chapter introduces the problem of self-organizing data structures. We define the metrics used for judging which algorithms are "good" and which are not, and describe the basic analytical methods for the algorithms. We proceed with a survey of deterministic versus probabilistic methods, and identify key features of each algorithm discussed, occasionally including the idea behind the algorithm's design.

The survey clearly shows that while the literature is rich with self-organizing list algorithms, they ultimately range between the accuracy and stability of the TR at one end of the spectrum and the speed and responsiveness of the MTF.

After describing the models of dependence to be used in the Thesis, Chapter 3 makes the case for sublist manipulation within lists, by discussing some other data structures which possess a similar notion of focusing on a specific *subset* of the data set at any given time. It proceeds by formally describing the MTF-MTF, and then showing how hierarchical list organization strategies without a dependence capturing algorithm can yield a poor performance. In Chapter 4, we look at various ways to partition objects into subsets, according to their interdependence relationships. We discuss some of the algorithms available to that end, together with the associated time and space complexity, as well as implementation details.

In Chapter 5, we combine the hierarchical algorithms with the partitioning ones to build our list organization framework. We test our algorithms with varying sublist sizes and dependence degrees, under Markovian and periodic environments. We empirically show the results stated above, and discuss some associated implementation details. In the following Chapter, we introduce the doubly linked list problem, and generalize our algorithms so that they can be directionally mapped, that is, applied with respect to the query direction. We make some recommendations on the choice of the algorithms based on the results we obtained. Chapter 7 concludes the Thesis.

Chapter 2

Survey of Previous Methods

2.1 Introduction

The study of self-organizing data structures can be classified in many ways: according to the type of search, whether the underlying data structure is linear (involving sequential lists) as opposed to two-dimensional (involving trees); according to the approach used by the organization strategy, for example, whether it is deterministic as opposed to stochastic; according to the memory required: whether the additional memory is zero or a constant, as opposed to linear; and according to the environment, whether the data accesses are independent as opposed to dependent. This chapter surveys, in some detail, the vast body of literature available on self-organizing data structures. It also provides an introduction to the methods of analysis that are used in studying these data structures.

2.2 Definitions

2.2.1 Self-Organizing Data Structures

A *self-organizing data structure* is a data structure that employs a rule (also referred to as a scheme, strategy, or a permutation algorithm), for changing the arrangement of its elements in order to improve the average or amortized access time. The rule is applied periodically, for example, after every access to the list.

The system manipulating the self-organizing data structure is not aware of the probability of access for the individual records, although it is generally assumed that the access probabilities follow some distribution, albeit an unknown one. When analyzing the performance of the data structure, we also assume that the access sequence will not request a record that is not found in the list, and that each record will be accessed at least once [15].

2.2.2 Self-Organizing Lists

Suppose we have a *list* of records $R_1, R_2, \dots, R_n$ in an arbitrary order π , where R_i is in position $\pi(i)$ for $1 \leq i \leq n$. The record R_i is accessed with unknown probability, s_i . Whenever R_i is accessed, a sequential search is done on the list, the operation costing $\pi(i)$ units of time. The expected search cost for π would then be

$$cost(\pi) = \sum_{1 \leq i \leq n} s_i \pi(i). \quad (2.1)$$

In order to minimize the search cost, it is desirable to arrange the records of the list in an optimal manner, which, in this case, implies that they must be in the descending order of their access probabilities. In the literature, this problem has many names including online list organization, the adaptive list update problem, self-organizing linear search, or more simply, the study of self-organizing lists. A *self-organizing list*

is a list that employs a strategy to alter the position of its records, possibly with each access, so as to approach an optimal ordering with the more frequently accessed records being closer to the front of the list. Strategies which accomplish this can be divided into two groups: deterministic and stochastic, and will be discussed later in detail.

Other extensively studied self-organizing data structure are tree-based data structures. These structures, although more difficult to implement, have the benefit of reducing the search time considerably. List data structures are still, however, very popular both in research and practice, because of their simplicity and ease of implementation. As self-organizing trees are outside the scope of the Thesis, they will not be discussed in this survey.

2.2.3 Cost

Cost is the most obvious metric for judging whether a self-organizing data structure is “good” or not. After all, the whole point of utilizing a self-organizing data structure is to minimize the cost. However, the term “cost” can mean different things, and needs to be formally defined.

The *total time* cost associated with accessing a record in a data structure is the sum of the cost required to find the record, and the cost involved in reordering the list after applying the permutation algorithm. However, the reordering cost is often a constant, or quite small when compared to the search cost, and is therefore negligible or ignored. This is the standard cost model, wherein the organization cost is not considered (called *free exchange*), as opposed to the *paid exchange* cost model (P^d), where the cost of organization itself involves d units [4]. Most of the literature, however, uses the standard cost model, and therefore the paid exchange model is not considered in this Thesis. The analysis of self-organizing data structures, therefore, deals with three aspects of the associated cost: asymptotic cost, amortized cost, and

relative cost.

Asymptotic cost

As mentioned earlier, while the self-organizing data structure has no knowledge of the probabilities of the access sequence, it is expected that the sequence follows a non-uniform distribution with some records having higher access probabilities than others. If the records are accessed uniformly, no amount of reordering can yield a superior access time [15]. With this assumption, however, if the algorithm can be expected to converge over time, the average search cost attains its asymptotic value. Thus, the term “average search cost” usually means the average search cost after convergence, and consequently, the phrases “average”, “asymptotic”, and “expected search cost” are used interchangeably.

The expected search cost for algorithm A is readily given by the expected value law over all list configurations,

$$\begin{aligned} ExpectedCost(A) &= \sum_{1 \leq j \leq n!} (P\{\pi_j\}_A \text{cost}(\pi_j)) \\ &= \sum_{1 \leq j \leq n!} (P\{\pi_j\}_A \sum_{1 \leq i \leq n} s_i \pi_j(i)), \end{aligned} \quad (2.2)$$

where $\text{cost}(\pi_j)$ is given by Equation (2.1). However, it is often difficult to obtain the probabilities of all possible list configurations. Instead, several authors (e.g., [14, 34]) showed another way to compute the expected search cost that depends on the probability that record R_j precedes R_i . Let I_j be defined as

$$I_j = \begin{cases} 1 & \text{if } R_j \text{ precedes } R_i \\ 0 & \text{otherwise.} \end{cases}$$

In other words, the position of R_i is given by the expression $1 + \sum_{j \neq i} I_j$. Notice also

that the expected value for I_j is the probability that R_j precedes R_i . Therefore,

$$\begin{aligned} E[\text{Position of } R_i] &= 1 + \sum_{j \neq i} E[I_j] \\ &= 1 + \sum_{j \neq i} P\{R_j \text{ precedes } R_i\}. \end{aligned}$$

Consequently, the expected search cost would be

$$\begin{aligned} E[\text{Cost}] &= \sum_{1 \leq i \leq n} (s_i E[\text{Position of } R_i]) \\ &= \sum_{1 \leq i \leq n} (s_i (1 + \sum_{j \neq i} P\{R_j \text{ precedes } R_i\})) \\ &= \sum_{1 \leq i \leq n} (s_i + s_i \sum_{j \neq i} P\{R_j \text{ precedes } R_i\}) \\ &= 1 + \sum_{1 \leq i \leq n} \sum_{j \neq i} (s_i P\{R_j \text{ precedes } R_i\}). \end{aligned} \tag{2.3}$$

The literature also showed that in order for any permutation algorithm A to be better than B for a given self-organizing list, it suffices to show that the probability the record R_j precedes R_i using rule A is higher than it is for rule B , whenever $s_j > s_i$.

Again, even though the above equation is fairly simple, it is often difficult to derive a closed form expression for the quantity $P\{R_j \text{ precedes } R_i\}$ for a given algorithm. Still, we can calculate the average search cost for a given list and the associated access probabilities of its records can be calculated, through a series of algebraic steps. It has been noted since McCabe's work [23] in 1965 that self-organizing lists can be analyzed through the theory of Markov chains. A list with n records can assume any of $n!$ configurations by applying the permutation algorithm A . This can be represented by a Markov chain with transition matrix M of $n!$ states, where every state represents a particular configuration $(\pi_j, 1 \leq j \leq n!)$ of the list. Transition from one state to another occurs by applying the permutation algorithm, and is therefore dependent on

Figure 2.1: State diagram for the Markov chain of a three element list using the MTF rule.

the access probabilities s_i . If the chain is ergodic, the stationary probability vector of the possible configurations $\underline{P} = (P\{\pi_1\}P\{\pi_2\} \dots P\{\pi_n\})^T$ is given by the eigenvector of M corresponding to the eigenvalue unity, and is calculated by the formula

$$M^T \cdot \underline{P} = \underline{P} \quad (2.4)$$

Using (2.4), it is then a “simple” matter to calculate the expected cost for algorithm A through Equation (2.2).

The following example (taken from [11]) illustrates this process. Suppose we have a list of three records $\{a, b, c\}$ with corresponding access probabilities $\{s_a, s_b, s_c\}$. The list employs the move-to-front (MTF) rule, and thus, whenever a record is accessed it is moved to the front of the list. There are six possible configurations for the list:

$\pi_1 = \langle abc \rangle$, $\pi_2 = \langle bac \rangle$, $\pi_3 = \langle cba \rangle$, $\pi_4 = \langle acb \rangle$, $\pi_5 = \langle cab \rangle$, and $\pi_6 = \langle bca \rangle$. Figure 2.1 illustrates the state diagram of the underlying Markov chain. If the list is in configuration π_1 and record b is accessed, b is moved to the front of the list and the list configuration changes to π_2 . As this happens with probability s_b , element $m_{1,2}$ of the Markov chain with transition matrix M will be assigned s_b . We can similarly assign all the elements to obtain the following matrix characterizing the Markov chain obeyed by the MTF rule:

$$M = \begin{bmatrix} s_a & s_b & 0 & 0 & s_c & 0 \\ s_a & s_b & s_c & 0 & 0 & 0 \\ 0 & 0 & s_c & s_a & 0 & s_b \\ 0 & s_b & 0 & s_a & s_c & 0 \\ 0 & 0 & 0 & s_a & s_c & s_b \\ s_a & 0 & s_c & 0 & 0 & s_b \end{bmatrix}$$

Solving Equation (2.4) for $\underline{P}$ in terms of s_i is not trivial. However, for particular values of s_a , s_b and s_c , $\underline{P}$ can be easily calculated.

As this technique becomes computationally infeasible with larger lists, since it requires Markov chains with $n!$ states, researchers devised other mechanisms to facilitate the study of the Markov chain describing the system. Consider that, in the last example, we are only concerned with whether or not record a precedes b . This instantly collapses the Markov chain into two states, the first being all the list configurations in which a precedes b and the second representing those in which b precedes a (Figure 2.2). The resulting transition matrix is:

$$M = \begin{bmatrix} 1 - s_b & s_b \\ s_a & 1 - s_a \end{bmatrix}$$

Figure 2.2: State diagram for the Markov chain describing list configurations when record a precedes b and vice-versa, using MTF.

which, when solved using Equation (2.4), yields the static probability vector $\underline{P} = (s_a/(s_a + s_b), s_b/(s_a + s_b))^T$. In other words, the asymptotic probability that the system is in a state where record a precedes record b is $s_a/(s_a + s_b)$ using the move-to-front rule. Obviously, while this method works well with the move-to-front, it may not, and indeed, does not often work for other algorithms.

Time reversibility is another useful property of some Markov chains, that manifests when, starting from a given state, the probability of following a loop back to that state is the same as the probability of the reverse path. That is, if the Markov chain has a transition matrix M with elements $m_{i,j}$, the chain is time reversible if and only if

$$m_{i,i_1}m_{i_1,i_2} \dots m_{i_k,i} = m_{i,i_k}m_{i_k,i_{k-1}} \dots m_{i_1,i}$$

for all states $i, i_1, \dots, i_k$ [35]. Again, consider the above example, but this time using the transposition rule, in which whenever an element is accessed, it is swapped with the element directly preceding it unless it is the first element. The resulting Markov chain is shown in Figure 2.3. Starting from state 1, one can follow a clockwise loop through states $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 1$ with probability $s_b s_c s_c s_a s_a s_b = s_a^2 s_b^2 s_c^2$. Taking the reverse path $1 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$, the probability is $s_c s_c s_b s_b s_a s_a = s_a^2 s_b^2 s_c^2$, i.e., the same probability of the original path. For a thorough analysis of time reversible algorithms we refer the reader to [11].

Figure 2.3: State diagram for the Markov chain of a three record list using the transposition rule.

Time reversibility is a powerful analysis tool because of the following property: For a time-reversible self-organizing list with a transition matrix M ,

$$P\{\pi_i\}m_{i,j} = P\{\pi_j\}m_{j,i} \quad (2.5)$$

for all $i \neq j$. This property was first described by Rivest [34] in his analysis of the transposition rule, and was later generalized by Oommen and Dong [27] in their analysis of the swap-with-parent scheme. It gives a way by which we can find the relative probabilities of two different list configurations in terms of transition probabilities, which can then be used to ultimately derive the expression $P\{R_j \text{ precedes } R_i\}$, as we shall see in Section 2.3.3.

Finally, researchers often opt for comparing algorithms empirically. In such cases, a simulation of the data structure under study is implemented, and the average

asymptotic costs are calculated. By taking ensemble averages over all the experiments performed, as well as the time average “across” an experiment, the final converged statistics can be computed. For ergodic systems, ensemble and time average are equivalent. These averages “smoothen” the curve of the observed cost, and an accurate estimation of the asymptotic cost can be obtained.

Amortized cost

The amortized analysis [7, 37] of a self-organizing data structure is the cost analysis averaged over *all* operations performed, and not just after convergence. Usually the cost is averaged over a worst-case sequence of accesses, but this is not necessarily the case [15].

Some researchers have used amortized analysis to consider the worst case sequence of requests. Bentley and McGeoch [7] have provided empirical results that suggest that the amortized analysis of permutation heuristics better describes the behaviour of the heuristic in real data, than a probabilistic analysis.

The amortized cost analysis is of particular importance in nonstationary environments as there is no fixed optimal action over time. Narendra and Thathachar [24] define the optimality of a system in a nonstationary environment if it minimizes

$$\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{n=1}^T E[\beta(n)]$$

where $E[\beta(n)]$ is the expected cost at time n , which is exactly the definition of the amortized cost. Minimizing the amortized cost will, therefore, be one metric by which we can compare the algorithms described in this Thesis.

Relative measures and competitiveness

Another way to measure the performance of self-organizing lists is by comparing the specific algorithm to another algorithm for the same request distribution. One can achieve this by comparing the algorithm against the *optimal static ordering*, where the self-organizing list is compared against another list whose records are arranged in the descending order of the static access probabilities [2]. Notice, however, that the optimal static ordering may not always be the ordering that would minimize the search cost. Consider sequences that are characterized with *locality of reference*. Subsequences of these sequences can have access frequencies that are different from the overall access frequencies. Obviously, the optimal static ordering may not be “optimal” in this setting.

Competitiveness is another relative measurement of an online algorithm against an offline one. By *online*, we mean that the algorithm does not know the access sequence beforehand, whereas in an *offline* setting, the entire sequence is known *a priori* and the algorithm can serve the requests at a minimum cost. An online algorithm is *c*-competitive if, for any request sequence, its cost is no more than *c* times the cost of the optimum offline algorithm for that sequence [33]. Sleator and Tarjan [37] were the first to use competitive analysis in their analysis of the MTF algorithm.

2.2.4 Convergence

Convergence is the second aspect of analysis of self-organizing data structures, although not nearly as widely studied as the different types of costs. There are several reasons for this. First of all, it is not too reasonable to believe that an algorithm will arrive at an optimal ordering. Secondly, when the algorithm gets there, unless it is an absorbing algorithm, the likelihood that the ordering will change is great. An arbitrary deviation margin around the asymptotic cost should be set in order to

define the “steady state.” Thirdly, as explained earlier, the asymptotic cost is often not known *a priori*.

Bitner [8] suggested a measure of convergence that he called *overwork*. The overwork of an algorithm is defined as the area between the cost curve and its asymptote. A steeper cost curve means that the algorithm is converging fast, and will result in a smaller overwork area. Therefore, the smaller the overwork, the faster the convergence. To compare two algorithms, Bitner suggested measuring the area between their respective costs.

A closed form expression for the overwork is intractable except for specific distributions, and for a limited number of algorithms. Often, researchers opt for other ways to compare the convergence of algorithms under investigation. One way to achieve this is to allow the algorithm to run for a sufficiently long time, and to evaluate this final cost as the asymptotic cost. The time for convergence would then be the amount of time taken for the measured cost to come within an arbitrary percentage of the asymptotic cost. Throughout the literature, though, convergence does not seem to be a big concern in studying self-organizing algorithms.

2.3 Deterministic Strategies

2.3.1 Frequency Count (FC)

The simplest approach to achieve a self-organizing list is by using the *frequency count* (FC) algorithm, which keeps a counter for each record, and maintains the list according to the access frequencies. Whenever an element is accessed, its corresponding counter is incremented, and is then moved forward in front of all the records with lower access counts.

Indeed, FC yields some impressive analysis results: its asymptotic cost is the same

as that of the optimal static ordering, and its amortized cost is at most twice that of the optimal [7]. However, this method has some drawbacks: besides its linear memory requirement that could perhaps be better utilized by some better nonsequential search technique as Knuth remarks [20], FC is relatively insensitive to locality of reference [15, 37], meaning that it would perform poorly if there is dependence among the the access requests. Furthermore, it appears that FC requires an extra non-constant search to find the new position where the accessed record has to be placed. Finally, FC is not c -competitive for any c [37].

Apart from FC, the two most famous memory-free, self-organizing schemes for linear lists are the *move-to-front* (MTF) and *transposition* (TR) rules, introduced by McCabe in 1965 [23]. In MTF, a record is moved to the front of the list whenever it is accessed unless it is the first record. In TR, the record is moved one position closer to the front of the list by interchanging it with its preceding record unless it is at the front of the list. Most of the other deterministic schemes can be seen to be variations of these two.

2.3.2 The Move-To-Front (MTF) Rule

The average search cost of the move-to-front heuristic has been studied by [8, 20, 23, 34]. Recall, from Section 2.2.3, that we need a closed form expression for the probability that record R_j precedes R_i for a given rule, in order to calculate the average search cost. We have already derived this expression in Section 2.2.3 using Markov chains. Here is a simpler derivation taken from [14, 34]: R_j will precede R_i if there has been a request for R_j subsequent to the most recent request of R_i . Let the time elapsed since that request be k . If R_j precedes R_i , we know that k requests ago there had been one request to R_j , and since then, $k - 1$ requests for records other

than R_i or R_j . If we sum the probability of this event occurring over all k , we get,

$$\begin{aligned} P\{R_j \text{ precedes } R_i\} &= s_j * \sum_{1 \leq k \leq \infty} (1 - s_i - s_j)^{(k-1)} \\ &= \frac{s_j}{s_i + s_j}. \end{aligned} \quad (2.6)$$

Applying Equation (2.3), we obtain the average cost of the MTF rule (A_{MTF}) to be:

$$\begin{aligned} A_{MTF} &= 1 + \sum_{1 \leq i \leq n} \sum_{j \neq i} (s_i P\{R_j \text{ precedes } R_i\}) \\ &= 1 + \sum_{1 \leq i < j \leq n} (s_i P\{R_j \text{ precedes } R_i\} + s_j P\{R_i \text{ precedes } R_j\}) \\ &= 1 + \sum_{1 \leq i < j \leq n} \left(\frac{s_i s_j}{s_i + s_j} + \frac{s_j s_i}{s_i + s_j} \right) \\ &= 1 + 2 \sum_{1 \leq i < j \leq n} \frac{s_i s_j}{s_i + s_j}. \end{aligned} \quad (2.7)$$

Compared to the optimal static ordering, which has the form $A_{STAT} = \sum_{1 \leq i \leq n} (s_i * i)$, Rivest [34] shows how the MTF rule does not do more than twice the average optimal work, by taking the ratio of A_{MTF} to A_{STAT} . Chung *et al.* [10] later lowered that bound by proving that $\frac{A_{MTF}}{A_{STAT}} \leq \pi/2 \approx 1.5708$ for any distribution. Bentley and McGeoch [7] proved that the amortized search cost for the MTF is at most twice that of the optimal static ordering if the list initially contains the items ordered by first access.

In terms of competitiveness, Sleator and Tarjan [37] proved that the MTF is 2-competitive. They proved that, for any algorithm A (online or not), and any access sequence S , if reordering costs are not counted,

$$C_{MTF}(S) \leq 2C_A(S) - m,$$

where C is the total cost of all operations, and m is the number of accesses. Their proof utilized the concept of a *potential function*, which maps the configuration of a list to a real number, Φ . If an operation at time t costs $c(t)$, and caused the list to shift from potential $\Phi(t-1)$ to $\Phi(t)$, they defined the *amortized time* of such an operation to be $c(t) + \Phi(t) - \Phi(t-1)$, i.e., the running time plus the change in the potential. They then invoked both algorithms *MTF* and *A* to run in parallel on two lists that were initially identical.

Let us define the potential function to be the number of inversions in the *MTF*'s list with respect to *A*'s list, where an inversion in one list with respect to the other is defined as an unordered pair i and j such that i occurs anywhere before j in one list and anywhere after j in another. Clearly, since the lists are initially identical, the initial potential is zero.

Consider an access on element x . Let k denote the number of elements that precede x in both *MTF*'s and *A*'s lists. Further, let l denote the number of items that precede x in *MTF*'s list but not in *A*'s. In such a case, $c_{MTF}(t) = k + l + 1$ and $c_A \geq k + 1$. When *MTF* serves the request and moves x to the front of the list, l inversions are destroyed, and simultaneously, at most k new inversions are created. Thus, the amortized time $a_{MTF}(t)$ is:

$$\begin{aligned} a_{MTF}(t) &= c_{MTF}(t) + \Phi(t) - \Phi(t-1) \\ &\leq c_{MTF}(t) + k - l = 2k + 1 \\ &\leq 2c_A(t) - 1. \end{aligned}$$

Summing over all t , we get

$$\begin{aligned} \sum_{1 \leq t \leq m} a_{MTF}(t) &= C_{MTF} + \Phi(m) - \Phi(0) \\ &\leq 2C_A - m. \end{aligned}$$

But since the lists are initially equal, the initial potential $\Phi(0)$ is zero and the final potential $\Phi(m)$ is nonnegative. Therefore, $C_{MTF} \leq C_{MTF} + \Phi(m) - \Phi(0) \leq 2C_A - m$.

This result is, indeed, powerful as it implies that, for *any* sequence of requests, the MTF algorithm is bounded by being within a constant factor as efficient as *any* algorithm that moves the accessed element closer to the front of the list. This includes algorithms with lookahead, and indeed, the ones with knowledge of the entire access sequence, because the *a priori* knowledge of access sequences cannot significantly reduce the cost. Karp and Raghavan [33, reported personal communication] later suggested that the competitive ratio of 2 is the best ratio that a deterministic online algorithm, for the list update problem, can achieve.

We believe, however, that this theorem does not include algorithms that do not move accessed elements closer to the front of the list (such as the move-to-rear algorithms, discussed later). This is because the l inversions may not have been destroyed and the k inversions may not have been created. For the same reason, algorithms that move more than one element are also believed to not be included in the theorem.

2.3.3 The Transposition (TR) Rule

Unlike the MTF, the transposition rule is a time-reversible algorithm, since, informally speaking, the original list can be reconstructed by reversing the access sequence. Equation (2.5) gave a relation between two specific configurations of a time-reversible system in terms of transition probabilities. A more specific version of this relation for the transposition rule was given by Rivest [34], which states that the system's stationary probabilities obey the following relation,

$$\frac{P\{\pi_a = \langle R_{i_1} R_{i_2} \dots R_{i_j} R_{i_{j+1}} \dots R_{i_n} \rangle\}}{P\{\pi_b = \langle R_{i_1} R_{i_2} \dots R_{i_{j+1}} R_{i_j} \dots R_{i_n} \rangle\}} = \frac{s_{i_j}}{s_{i_{j+1}}}, \quad (2.8)$$

for $1 \leq j < n$ if $s_k \neq 0$ for $1 \leq k \leq n$. In other words, the ratio of the probability of a list configuration before and after a pair of elements are transposed is the same as the ratio of the respective access probabilities of these elements. Notice that we can find the ratio of the probabilities of any two list configurations π_a and π_b through the following relation,

$$\frac{P\{\pi_a\}}{P\{\pi_b\}} = \frac{P\{\pi_a\}}{P\{\pi_{i_1}\}} \times \frac{P\{\pi_{i_1}\}}{P\{\pi_{i_2}\}} \times \dots \times \frac{P\{\pi_{i_{k-1}}\}}{P\{\pi_{i_k}\}} \times \frac{P\{\pi_{i_k}\}}{P\{\pi_b\}}$$

for some $1 \leq k \leq n! - 2$. Using Equation (2.8), this relation can be rewritten as

$$\frac{P\{\pi_a\}}{P\{\pi_b\}} = \prod_{u,v} \left(\frac{s_u}{s_v} \right) \quad (2.9)$$

for some $1 \leq u, v \leq n$. Let $\delta_i(\pi_a, \pi_b)$ denote the number of places that R_i is displaced from its position during transformation from π_a to π_b . Further, let this quantity be positive if R_i is moved further from the list head (i.e., R_i appeared in π_a in a position prior to its position in π_b) and negative if R_i is moved closer to the list head. For every transposition of R_i , s_i will appear in the numerator of Equation (2.9) when it moves down the list and in the denominator when it moves up the list. Thus, all instances of the term s_i in Equation (2.9) can be grouped as $s_i^{\delta_i(\pi_a, \pi_b)}$, and the equation can be rewritten as:

$$\frac{P\{\pi_a\}}{P\{\pi_b\}} = \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_a, \pi_b)}. \quad (2.10)$$

Now that we can find an expression for the probability of any list configuration π in terms of the probability of the initial configuration π_0 , we can use Equation (2.2) to find the expected search cost under the transposition rule A_{TR} as given by Rivest

to be:

$$A_{TR} = P\{\pi_0\} \sum_{\pi} \left(\left(\prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right)$$

where

$$P\{\pi_0\} = \left(\sum_{\pi} \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \right)^{-1}$$

The details of the derivation are omitted, and can be found in [11].

Rivest [34] proved that the transposition heuristic always has a lower asymptotic cost than that of the MTF, except for lists with two records or those where all nonzero probabilities s_i are equal. Rivest also conjectured that TR minimizes the average search cost for any distribution, but Anderson *et al.* [5] found a counterexample.

Rivest also performed an empirical analysis on the TR, and more specifically concentrated on requests that obeyed Zipf's law. On these sequences, Rivest found that TR performs better than MTF. In contrast, Bentley and McGeoch [7] ran their experiment on requests from text and Pascal files, and found that TR performed worse than both MTF and FC. Yet, they reported that TR outperformed the expected results of their analysis, and that the output may have been different had Pascal reserved words been treated differently than identifiers.

Dong [11] showed empirically that the transposition rule yields an asymptotic search cost that is always better than that of MTF for five different probability distributions. She performed 12 parallel experiments on lists with 100 records for each of the following distributions: Zipf, 80-20, Lotka, exponential and linear. She first generated 250,000 queries on the lists to allow the algorithms to converge. She then averaged the cost over the last 50,000 queries (time average) and over the twelve experiments (ensemble average), to get an estimate of the asymptotic cost. Table 2.1 illustrates the results obtained. We have done similar experiments which confirm her results, except for the exponential distribution where we believe the results are off by

one decimal point.

As a result of the above discussions, compared to the optimal static ordering, we know that the following relation is true, for most sequences that obey a stationary distribution

$$A_{TR} \leq A_{MTF} \leq \frac{\pi}{2} * A_{STAT}$$

By providing a counterexample, Bentley and McGeoch [7] showed that, unlike MTF and FC, TR does not have the property of a constant-bound amortized cost, compared to the amortized cost of the optimal static ordering. Consider the access sequence $ABCDE(ED)^k$. In TR, the last two elements will keep swapping positions and the average search cost will be around 5, whereas it is 1.5 in optimal ordering.

In terms of competitive analysis, again, Sleator and Tarjan [37] showed that TR is not c -competitive for any constant c that is independent of the list size n , citing the same counterexample by Bentley and McGeoch.

It is an accepted fact in the literature that TR is the slowest list organizing strategy. Using his definition of the overwork measure of convergence, Bitner [8] shows that for a particular distribution where $s_1 = 0$ and $s_i = 1/(n-1)$, $2 \leq i \leq n$, the overwork for the transposition rule is $(n^2 - 1)/6$ and for the move-to-front rule is $(n-1)/2$. Thus, there is an *order of magnitude* of difference between the two.

Bitner suggested another measure for comparing the convergence of two algorithms, by counting the *total expected cost*, which is the sum of the costs for r accesses. From his results, it is noticeable that the total expected cost for the transposition rule over Zipf distribution grows quadratically with the number of accesses, indicating poor convergence prospects.

In the experiments cited above, Dong also showed empirically that the transposition rule always converges to a steady state much slower than the MTF does. She observed the approximate number of accesses required for the scheme to converge to

within 1% of its asymptotic search cost to be as reported in Table 2.1. Table 2.2 illustrates these results, which our experiments also confirm.

2.3.4 Hybrid MTF and TR Algorithms

It is clear that the MTF and TR rules have their own strengths and weaknesses. MTF is superior when quick convergence and a quick response to a switching environment is required. It also performs better in environments characterized with locality of reference [9]. TR excels if stability is desired and quick convergence is not essential. Researchers have suggested several ways to combine these algorithms to benefit from their respective advantages.

Bitner's Algorithm

Bitner [8] proposed an algorithm which behaves initially as the move-to-front rule to exploit its quick convergence property, then switch to transposition to benefit from the lower asymptotic cost and higher stability. Determining when to switch, however, is not straightforward and purely assumptive.

Move-ahead- k (MHD(k))

Rivest [34] proposed the move-ahead- k rule as a compromise between MTF and TR. In move-ahead- k , when an element at position j is accessed, it is moved k -positions forward to the front of the list unless it is in the first k positions, in which case it is moved to the front. Observe that move-ahead-1 is the same as the transposition rule, while move-ahead- n is the same as move-to-front. As such, the performance of move-ahead- k is expected to lie between the two. However, there is no formula for the average search cost of move-ahead- k as its analysis is difficult. Tenenbaum [38] carried out simulations on the move-ahead- k rule that showed that the best value of

k is correlated to the list size.

POS(k) and SWITCH(k)

Tenenbaum and Nemes [39] also generalized the move-to-front and transposition rules in their POS(k) rule. In this rule, when an element at position j is accessed, it is moved to the k^{th} position of the list if $j > k$, or it is transposed with the preceding element if it is in positions 2 to k . Again, observe that POS(1) is the same as move-to-front, whereas POS($n - 1$) is the transposition rule. They also suggested the SWITCH(k) rule, which uses the same semantics as POS(K) except that elements are moved to front if they are in positions 2 to k and transposed otherwise.

2.3.5 Swap-With-Parent (SWP) and Move-To-Parent (MTP)

Algorithms that take the list size into consideration during reorganization are few. This is despite the fact that Tenenbaum reported that considering the list size could be crucial in leading to better organization [38]. The exception to this is Hester and Hirschberg JUMP algorithms [16, see Section 2.5]. Motivated by this fact, Oommen and Dong [11, 27] proposed the swap-with-parent (SWP) algorithm. SWP conceptually considers the list as a heap structure with no ordering constraints between the parents and their children. The “parent” record of record i is the record that lies half way between the beginning of the list and i . SWP swaps the accessed record with its parent in the heap. As such, it takes as few as $\lceil \lg n \rceil$ accesses to bring a frequently accessed record to the front of the list. As the case of transposition, the Markov chain associated with SWP is time-reversible, since any configuration can be reconstructed by reversing the access sequence since it was constructed. Oommen and Dong used time-reversibility techniques to derive formulas for the average search cost of the SWP rule.

Scheme	Zipf	80-20	Lotka	Exponential	Linear
MTF	26.439	31.703	4.484	2.7722	41.735
SWP	24.276	29.245	4.235	2.7719	41.379
MTP	22.118	27.008	3.771	2.3235	38.660
TR	19.717	24.237	3.419	2.0357	34.515

Table 2.1: Experimental results for the asymptotic cost of the MTF, SWP, MTP, and TR rules for lists of size 100 [11]. Results for the Exponential distribution have been modified to reflect the results obtained in *this* present study.

SWP avoids the slow convergence of the transposition rule by a logarithmic factor. In the same time, it is more conservative than the MTF, and is less likely to destabilize a list when an element with low access probability is accessed. Although not formally proven, SWP empirically yields better asymptotic results than that of MTF. However, SWP has a drawback: if an element has zero or negligible access probability, it will fall to the bottom of the *heap*, but not be at the end of the list. If such element is not the parent to any other element, it is never further moved, and consequently up to almost half of the lists' elements may have higher access probabilities, and yet be positioned behind that element.

To overcome this problem, Oommen and Dong suggested another rule, the move-to-parent (MTP) rule. Under this rule, the accessed record is moved to its parent's position, and the parent, along with all the records that follow it, are shifted back one position. Notice that this is actually a move-ahead- k where k is not constant, but rather a function of the position of the accessed record.

Dong [11] provided empirical results comparing the asymptotic cost (Table 2.1) and convergence (Table 2.2) for the MTF, SWP, MTP, and TR rules.

2.3.6 Batched Algorithms

A batched algorithm is not a permutation algorithm, but a modifier that is used in conjunction with one, to determine if the reorganization should take place or not.

Scheme	Zipf	80-20	Lotka	Exponential	Linear
MTF	9000	30000	3000	5500	1000
SWP	25000	20000	30000	20000	2500
MTP	20000	30000	20000	60000	20000
TR	170000	160000	250000	270000	130000

Table 2.2: Experimental results for the convergence of the MTF, SWP, MTP, and TR rules for lists of size 100 [11]

Move-every- k th-access

McCabe [23] suggested the move-every- k th-access rule to reduce the time spent re-ordering the list. In this rule, a counter is needed for every record in the list, which is incremented whenever the record is accessed. Whenever the number of accesses for a record reaches k , it is moved to the front of the list (assuming we are using the MTF as the permutation algorithm) and its associated counter is reset to zero. Bitner [8] has shown that, for any set of access probabilities except the uniform distribution or a distribution with a record of probability one, the move-to-front rule, combined with move-every- k th-access, has strictly lower cost than the move-to-front rule by itself.

Bitner suggested a modification to move-every- k th-access, in which all the counters are cleared whenever a list permutation takes place, instead of only the permuted record's counter. Bitner showed that this modification causes the asymptotic search cost to approach the optimum as k approaches infinity, whereas this is not the case for the standard move-every- k th-access. However, it could be expensive to reset all the counters unless they are stored separately from the records in a contiguous area of memory that could be efficiently cleared.

k -in-a-row

Kan and Ross [19] studied the k -in-a-row scheme, in which the list is permuted only after it is accessed k consecutive times. This scheme has a constant memory requirement of two counters: one that stores the index of the last record requested, and another for counting the number of consecutive accesses for that record.

Kan and Ross showed that under this rule, the asymptotic probability that the record with the j^{th} largest access probability is in position j approaches unity as k approaches infinity. Gonnet *et al.* [12] showed the same result by proving that $(k + 1)$ -in-a-row is superior to k -in-a-row.

 k -in-a-batch

Gonnet *et al.* [12] also suggested a modification to the k -in-a-row rule, called k -in-a-batch rule. Under this rule, accesses are grouped into batches of size k . The permutation algorithm is applied only if all the requests within a batch are for the same record. The memory requirement for such scheme is the same as that of k -in-a-row, i.e., constant requirement of two counters.

Gonnet *et al.* proved that the asymptotic search cost of the k -in-a-batch heuristic in combination with either the move-to-front rule or the transposition rule is less than or equal to that of the k -in-a-row with the same permutation algorithms, respectively.

2.3.7 TS(0)

Albers and Mitzenmacher [1, 2] introduced the algorithm TS(0), which is a deterministic member of the TS(p) (time-stamp) family of randomized algorithms which we will discuss presently (see Section 2.4.2).

The TS(0) algorithm is formulated as follows: Insert the requested item, x , in front of the first item in the list that has been requested at most once since the last

request to x . If there is no such item or if x has not been requested so far, then leave the position of x unchanged.

As an example, consider the list $\langle x_3, x_2, x_4, x_6, x_1, x_5 \rangle$. Suppose $TS(0)$ needs to service the second request for x_1 in the sequence $x_1, x_2, x_2, x_4, x_3, x_3, x_1$. Notice that since the last request for x_1 , only x_4 and x_6 have been requested at most once. Since x_4 is the closest to the beginning of the list, x_1 is inserted before x_4 , and the new list configuration becomes $\langle x_3, x_2, x_1, x_4, x_6, x_5 \rangle$.

Albers [1] showed that $TS(0)$ achieves a competitive ratio of 2 on any request sequence, as does the MTF. It was further shown in [2] that $TS(0)$ performs better than MTF, both in terms of the expected search cost, and by comparison against the optimal static ordering and optimal dynamic algorithms, for request sequences that are generated by a probability distribution other than the uniform distribution.

$TS(0)$, however, has a linear memory requirement, as information on past requests needs to be stored in order to determine where the requested item should be moved. Observe that each record should at least keep track of the last two times it was requested. Furthermore, in a simple implementation, $TS(0)$ may need a second pass on the list to locate the target position for the accessed record. Albers and Westbrook [4] note that such a price is often tolerated, as the positions of the records matter more than reorganization time in a target application, such as data compression.

2.3.8 $MRI(m)$ and $PRI(m)$

Bachrach *et al.* [6] introduced a new family of list update algorithms, called the move-to-recent-item (MRI) family. The $MRI(m)$ algorithm is formulated as follows: Move the requested item x just after the last item y on the list that is in front of x and which was requested at least $m + 1$ times since the last request of x . If there is no such item or this is the first request for x , move x to the front.

Bachrach *et al.* have shown that $MRI(1)$ is equivalent to $TS(0)$, except for treating

first requests. It is also obvious that the limiting element in MRI as m approaches infinity is MTF. In this family, $\text{MRI}(i)$ is more conservative than $\text{MRI}(i + 1)$ in the sense that it is more reluctant to move an accessed element to the front of the list. The MRI family, however, suffers from the same memory requirements and processing penalty issues as $\text{TS}(0)$.

Bachrach *et al.* introduced another family of list update algorithms, called the pass-recent-item (PRI) family. $\text{PRI}(m)$ does the following: It moves the accessed item x just in front of the first item y on the list that was requested at most m times since the last request for x . It does nothing if there is no such item, y . If this is the first request for x , x is moved to the front.

Again, $\text{PRI}(1)$ is equivalent to $\text{TS}(0)$, except for handling the first requests, and the limiting member of the PRI family as m approaches infinity is MTF. Bachrach *et al.* conjectured that for all $m \geq 1$, $\text{PRI}(m)$ is 3-competitive.

2.3.9 $\text{SBR}(\alpha)$ and $\text{SBD}(k)$

Schulz [36] introduced two new families of list update algorithms. He defined $w_k(x, t)$ to be the k th last access time to record x at time t . He also defined the rank function $s_k(x, t) = t - w_k(x, t)$, or the time that elapsed since the last k th access on element x at time t . For $\alpha \geq 0$, he defined a new rank function to be the convex combination of $s_1(x, t)$ and $s_2(x, t)$, as follows:

$$r_t(x, \alpha) = (1 - \alpha) \cdot s_1(x, t) + \alpha \cdot s_2(x, t).$$

In other words, the rank is a convex combination of the time since the last access and the time since the second last, with α giving different weights to the two times. Now the sort-by-rank class of algorithms, $\text{SBR}(\alpha)$, is defined as follows: Upon a request for record x at time t , insert x just after the last item y in front of x with

$r_t(y, \alpha) < r_t(x, \alpha)$. Move x to the front of the list if there is no such item, or if x is requested for the first time.

Schulz noted that $\text{SBR}(0)$ is the MTF, and that $\text{SBR}(1)$ is equivalent to $\text{TS}(0)$ except for handling the first requests. He further proved that for $0 \leq \alpha \leq 1$, $\text{SBR}(\alpha)$ is 2-competitive.

The second class of list update algorithms introduced by Schulz is the sort-by-delay class, $\text{SBD}(k)$, which is defined as follows: Upon a request for record x at time t , insert x just after the last item y in front of x such that $s_k(y, t) < s_k(x, t)$. Move x to the front of the list if there is no such item or if x is requested for the first time. In other words, x is inserted in front of the first item y before x that was requested at most $k - 1$ times since the $(k - 1)$ th last access to x . Schulz proved that $\text{SBD}(k)$ is k -competitive for $k \geq 2$. He further showed that $\text{SBD}(k)$ is asymptotically optimal (i.e., as k approaches infinity, the resulting list arrangement approaches the optimal static ordering).

2.3.10 Move-To-Rear (MTR) Algorithms

Section 2.2.3 described how to calculate the asymptotic cost of a self-organizing list that can be represented by an ergodic Markov chain. In fact, not all self-organizing lists are ergodic, some obey an absorbing Markov chain, in which the final positions of the records after convergence never change. Oommen *et al.* [28, 29, 32] were the first to suggest schemes with such a property, the move-to-rear (MTR) schemes, which contrary to intuition, move the accessed records to the end of the list, instead of promoting them directly to the front. The end result, however, is that the lists are asymptotically ordered into an optimal ordering. Moreover, each element is permuted only once in the entire scheme, thereby greatly reducing the reordering time.

Deterministic linear space MTR

This scheme, introduced by Oommen *et al.* in 1990 [29], is essentially a move-every- k th access scheme, in which a record is moved to the end of the list after it has been accessed k times. It is different, however, in that once a record is moved, it is never moved again.

The algorithm requires linear space because each record R_i maintains a counter x_i to track the number of accesses and a boolean variable m_i to mark if the record has already been moved. Initially, all counters are set to zero and all boolean variables are set to “false.” When record R_i is accessed, if m_i is false then x_i is incremented. If $x_i = k$ then m_i is set to “true” and the record is moved to the rear of the list.

Oommen *et al.* showed that under this scheme, the asymptotic probability that record R_j precedes R_i , given that either record is accessed, can be made as close to unity as desired if $s_j > s_i$. Consequently, assuming that the access probabilities are all distinct, the probability of the list converging into the optimal ordering can be made as close to unity as desired.

Deterministic constant space MTR

Oommen *et al.* [29] also proposed another MTR scheme that requires constant additional space. In this scheme, records are moved to the rear of the list if they are accessed k consecutive times. Once a record is moved, it is not to be moved again.

The memory requirements for this scheme is an indicator of the last record accessed and the number of times it has been consecutively accessed. There is no need to keep a boolean variable associated with each record. Instead, the implementation can keep track of the position of the first record that was moved to the rear (called *FrontNew*). Any record that precedes *FrontNew* are then known to have not been moved. In fact, Oommen *et al.* [29] suggested starting the linear search from *FrontNew*, and then

from the beginning of the list if the record had not been found. This is because we know that the record at *FrontNew* is going to be the first record in the converged list, meaning that it will, most likely, have the highest access probability (for an optimal scheme). The implementation details can be applied on any other MTR scheme too.

Oommen *et al.* have proved that this scheme is expedient and conjectured its optimality.

Optimal deterministic constant space MTR

Later, Oommen and Ng [32] improved the deterministic constant space move-to-rear scheme to yield the asymptotically optimal. Records are still moved to the rear of the list if accessed k “consecutive” times, except that it is permitted to access records that have been moved before, in between these “consecutive” requests. The access counter is reset only if a request for a *different* record arrives, namely for one that has not been moved to the end of the list.

Oommen and Ng noted that this was (at its time) the only algorithm in the literature that conditions its reorganization based on the event that the accessed element has been moved before or not, and therefore its analysis is based on time-varying conditional access probabilities, since the set of unmoved records changes with time. They proved that the probability that record R_j precedes R_i , given that $s_j > s_i$, is given by $s_j^k / (s_j^k + s_i^k)$, and therefore increases with k . Hence, they proved that the strategy is asymptotically optimal.

2.4 Probabilistic Strategies

2.4.1 Probabilistic MTF and MTR

Constant space probabilistic MTF

In 1987, Oommen and Hansen [28] introduced a probabilistic version of the move-to-front scheme. In this scheme, records are moved to the front of the list whenever accessed, but with a probability that decreases with time. More formally, a function $f(n)$ is defined to be the probability of any element being moved to the front of the list on being accessed at time n . A single memory location is required to store the value of $f(n)$. For the initial condition, we have $f(0) = a$, where $0 < a < 1$. With every time step, the function is updated with the rule:

$$f(n+1) = af(n)$$

Observe also that $f(n)$ can be computed using the equation $f(n) = a^n$, and therefore all that is needed to be stored is the index, n , of time that has lapsed since the scheme was initiated. With time, the probability of records being moved to the front decreases, and the list is absorbed into one of the $n!$ orderings (i.e., the Markovian representation of the list is absorbing).

Oommen and Hansen showed that this scheme is expedient, i.e., for any two records R_i and R_j with access probabilities $s_j > s_i$, the probability that the list is absorbed into a configuration in which R_j precedes R_i is always greater than 0.5. However, the scheme is also less optimal than the deterministic move-to-front algorithm in all cases.

Linear space probabilistic MTR

Oommen and Hansen [28] also proposed a second probabilistic scheme, called the linear space probabilistic move-to-rear scheme. In this scheme, a record R_i is moved to the rear of the list with probability q_i , which decreases every time R_i is accessed. Initially, all q_i 's are set to unity. They are updated at every time R_i is accessed by the following rule:

$$q_i(n+1) = \begin{cases} a q_i(n) & \text{if } R_i \text{ is accessed,} \\ q_i(n) & \text{otherwise,} \end{cases}$$

where $0 < a < 1$. As the probability of a record being moved to the rear of the list decreases asymptotically to zero, so does the number of list operations performed, and so the list *slowly* enters into its absorption configuration. This has the benefit of reducing the time spent on ordering the list.

Oommen and Hansen have shown that for any i and j ,

$$E[q_j(n)] < E[q_i(n)] \text{ if and only if } s_j > s_i$$

and that if $s_j > s_i$, then the probability that $q_j(n) < q_i(n)$ can be made as close to unity as desired. Consequently, the probability of the list converging to the optimal arrangement can be made as close to unity as desired.

2.4.2 Competitive Randomized Algorithms

Recall that no deterministic online algorithm can achieve a competitive ratio that is better than 2, as reported in [33]. A number of researchers have suggested randomized algorithms to achieve better competitiveness. Essentially, the idea behind all of these algorithms is to move the requested items in a more conservative way than the MTF, which is the classical deterministic algorithm and which achieves 2-competitiveness

[4].

One should be cautious, however, when interpreting the results of competitive analysis. Bachrach *et al.* [6] reported surprising results that are contrary to some theoretical results regarding competitive analysis, while empirically testing the performance of list update algorithms. They stated that in many instances, the best performance was obtained by deterministic algorithms that are not theoretically as competitive as the randomized ones!!

SPLIT

Algorithm SPLIT, introduced by Irani in 1991 [18], was the first algorithm to beat the competitive ratio of 2. Every record x in the list maintains a pointer to another record, $x.split$. Informally speaking, what the algorithm does is that it advances an item to the beginning of the list with probability 0.5, otherwise it places the item before what used to be the beginning of the list when it was last moved. The algorithm (adopted from [4]) is formalized as follows:

Algorithm SPLIT

Initialization:

For all items x in the list, set $x.split \leftarrow x$.

Repeat:

If item x is requested:

For all items y with $y.split = x$, set $y.split \leftarrow$ item behind x in the list

With probability 0.5:

Move x to the front of the list.

With probability 0.5:

Insert x before item $x.split$

If y preceded x and $x.split = y.split$, then set $y.split \leftarrow x$

Set $x.split$ to the first item in the list.

End Algorithm SPLIT

The competitive ratio of SPLIT is 1.875 [4]. Obviously, SPLIT requires linear memory and a second pass on the list to adjust the relevant pointers.

BIT, COUNTER(s, S), and RANDOM-RESET(s, D)

BIT is a simple randomized algorithm introduced by Reingold *et al.* [33]. Every item maintains a bit that is initialized independently and uniformly at random. Whenever an item is accessed, its bit is complemented. If the bit changes to 1, the item is moved to the front of the list, otherwise the list is left unchanged. BIT can be shown to be 1.75-competitive.

Reingold *et al.* also generalized BIT to a class of algorithms called COUNTER(s, S). Let s be a positive integer, and S be a nonempty subset of $\{0, 1, \dots, s-1\}$. Every item maintains a counter that is initialized with some number from the set $\{0, 1, \dots, s-1\}$, chosen independently and uniformly at random. Whenever an element is accessed, its corresponding counter is decremented modulo s . The item is moved to the front of the list if the new counter value belongs to S . The reader should observe that BIT is COUNTER(2, $\{1\}$). Reingold *et al.* showed that COUNTER(7, $\{0, 2, 4\}$) is 1.735-competitive. It should be noted that in the COUNTER family, the randomness comes from the initialization phase, after which the algorithms proceed in a strictly deterministic fashion.

RANDOM-RESET(s, D) is another variation of the COUNTER family by Reingold *et al.* [33]. Let s be a positive integer, and D be a probability distribution on the set $S = \{0, 1, \dots, s-1\}$, such that for $i \in S$, $D(i)$ is the probability of i . Every item maintains a counter that is initialized randomly with a number $i \in S$ using the distribution D . With each access to an item, its counter is decremented. When the counter reaches zero, the element is moved to the front of the list and the counter is re-initialized using D . With $s = 3$ and D such that $D(2) = (\sqrt{3} - 1)/2$ and

$D(1) = (3 - \sqrt{3})/2$, one can achieve a competitive ratio of $\sqrt{3} \approx 1.732$.

TS(p)

The time-stamp (TS) family of algorithms was introduced by Albers in [1]. We have already discussed TS(0) in Section 2.3.7. The generalized TS(p) algorithm is defined as follows:

Algorithm TS(p)

When an item x is requested, with probability p ,

- (a) Move x to the front of the list.

With probability $1 - p$,

- (b) Insert x in front of the first item in the list that precedes x , and
 - (i) that was not requested since the last request to x

or

- (ii) that was requested exactly once since the last request to x , and the corresponding request was served using step (b) of the algorithm

If there is no such item or if x is requested for the first time, leave the position of x unchanged.

End Algorithm TS(p)

Albers has shown that setting $p = (3 - \sqrt{5})/2$, one can achieve a competitive ratio of $(1 + \sqrt{5})/2 \approx 1.62$ which is the Golden Ratio.

Albers *et al.* [3] have proposed a combined BIT and TS(0), called COMB. Initially, they choose one of the two algorithms with probabilities $4/5$ and $1/5$ respectively. Afterwards, they serve the entire sequence using the chosen algorithm. Albers *et al.* showed that this algorithm achieves a competitive ratio of 1.6, which is the best ratio known so far.

FC	1.00000	MRI(0)	1.11243	MHD(16)	1.25063	MTF2	1.29230
TR	1.04362	MHD(8)	1.16194	COMB	1.27125	SPLIT	1.30888
MHD(2)	1.06413	PRI(0)	1.17338	TS(0)	1.27254	RMTF	1.32862
MHD(4)	1.10116	RMHD	1.22564	BIT	1.29185	MTF	1.32908

Table 2.3: Experimental results for the average cost (relative to FC) of different algorithms in a list of size 32 and an access sequence obeying Zipf's law [6].

2.5 Other Algorithms

Hester and Hirschberg [16] introduced a class of algorithms (called JUMP) that advance the requested record towards the front of the list using a variety of deterministic and probabilistic strategies. While searching for record x in the list, the algorithm maintains a *back-pointer* b that points initially to the first record. With every record p probed, a function $\beta(p, b)$ is evaluated, which is a function of the location of the current probed record p and that of the backpointer b . If β returns true, b is advanced to p . When x is finally found, it is advanced in front of b . This means that x can move forward anywhere from a single position to the front of the list. The way β is defined determines if the algorithm is deterministic or probabilistic. They introduced four kinds of such strategies: deterministic constant moves, deterministic fractional moves, probabilistic constant moves, and probabilistic fractional moves.

Bachrach *et al.* [6] used some more deterministic and randomized algorithms in their experimental study of list update algorithms, that are worth mentioning in the interest of completion:

MTF2: Upon access of element x at location i , move x to the front if and only if i is even.

Randomized MTF (RMTF): Upon access of element x , with probability $1/2$ move x to front, and with probability $1/2$ leave it in place.

Randomized move ahead (RMHD): Upon access of element x at the i th position, randomly choose a position from the set $\{1, 2, \dots, i\}$ and move x to that position.

Table (2.3) shows the experimental results by Bachrach *et al.* [6] of several algorithms described in this chapter for a list with 32 elements serving a request sequence obeying Zipf's law, relative to the best performing algorithm (FC), which produced an average access cost of 7.88.

2.6 Nonstationary Environments

When analyzing self-organizing data structure, it is often assumed that the access sequence is independent, that is, the access probabilities of an individual record are stationary and do not vary with respect to time. Little research is available on records being accessed in environments with locality of reference. The performance of these data structures under such environments is the core of this Thesis.

Lam *et al.* [21] laid the foundation for the analysis of heuristics in such environments, by proposing a Markovian model for dependent accesses. We shall show in Section 3.2.1 that the Environment can be described by a Markov chain of n states, with the transition probability matrix $P = (p_{ij})$, where p_{ij} signifies the conditional probability that record R_j is accessed at time $t + 1$ if R_i is accessed at time t . Lam *et al.* then defined an *extended Markov chain* for a given heuristic H , with $n(n!)$ states, and transition probability matrix $Q = (q[(\pi, i), (\sigma, j)])$. Let $H(\pi, i)$ denote the configuration resulting from applying heuristic H on configuration π when record R_i is accessed. The matrix Q would therefore be defined as:

$$q[(\pi, i), (\sigma, j)] = \begin{cases} p_{ij} & \text{if } H(\pi, i) = \sigma \\ 0 & \text{otherwise.} \end{cases}$$

If Q is ergodic, then each state (π, i) of Q has a stationary probability $q(\pi, i)$, where the quantity (π, i) means that record R_i is accessed when the list's configuration is π . Therefore, the expected search cost for heuristic H and access pattern P would

be given by:

$$ExpectedCost(H, P) = \sum q(\pi, i)\pi(i)$$

Using this technique, Lam *et al.* were able to give formulae for the expected search cost of the MTF in a dependent environment. However, such a method is still limited by the infeasibility of analysis of the stationary probability vector for a Markov chain with increasing number of states. We hope that this model would serve as a starting point for analysis of the algorithms presented in this Thesis in a subsequent work.

Chassaing [9] has shown that the expected search time for MTF in environments with locality of reference is not greater than that of any other sequential strategy. Later, in 2002, Bachrach *et al.* [6] backed this result with a comprehensive experiment that tested almost all the algorithms reported in the literature up to the time of the study. To model locality of reference, they used a source with $n = k \cdot m$ distinct requests, where k and m are positive integers. They partitioned n into k disjoint sets S_i of m elements, called *local sets*. Once a request is generated from S_i , the probability that the next request would be from the same set would be α , $0 \leq \alpha \leq 1$. A higher value of α indicates stronger locality. Within a local set, the probability distribution used obeyed Zipf's law. They found out that with stronger locality ($\alpha \geq 0.8$), MTF outperformed each of FC, TR, MRI(0), and TS(0) (Figure 2.4), and indeed all other algorithms they have tested. While FC was the best algorithm for lower values of α , it was the worst when there was strong locality, costing 42% more than the cost of MTF with $\alpha = 0.9$. Notice that the average access cost for FC remained the same for all values of α , indicating very low sensitivity for locality of reference. $\alpha \geq 0.8$ seems to be the threshold of where locality of reference really alters the behaviour of most algorithms. This model of query generation, called the Markovian query generation model, was described by Narendra and Thathacher in [24]. It will be one of the ways by which we model nonstationary environments in this Thesis.

Figure 2.4: Results obtained from Bachrach *et al.* for algorithms tested with different levels of locality of reference.

2.7 Conclusion

In this chapter we have surveyed, in some detail, the available literature for self-organizing linear search lists. We have described the different approaches of analysis to study and compare their performances. Generally speaking, there are two such approaches: the “probability-dependent” approach, which analyzes the average search cost using the access probabilities of the individual records, and the competitive “distribution-free” approach, which analyzes the amortized search cost for any request sequence, irrespective of the access distribution, and in relation to the optimum offline algorithm for that sequence.

The most famous of self-organizing list strategies are the move-to-front and transposition heuristics. We have seen that the probabilistic and competitive approaches give different, and often conflicting, performance evaluation for the different heuristics. While the transposition heuristic produces the best results in probabilistic analysis, the move-to-front heuristic is superior in competitive analysis. For that reason, many algorithms have tried to combine the advantages of these two, by doing more drastic list changes than the transposition rule, while at the same time being more conservative than the move-to-front rule. However, it is generally agreed that the transposition rule is best utilized in a converged and stable environment to achieve excellent average access times, while the move-to-front is best used in more dynamic environments.

Both deterministic and randomized list organizing algorithms have been proposed in the literature. Although most of the suggested probabilistic algorithms achieve better competitive ratios, it is our belief that the associated implementation complexities, as well as additional time and space requirements, do not make these algorithms very practical in real life scenarios. Indeed, the trivially implemented transposition and move-to-front rules have been empirically shown to outperform almost all other

algorithms [6].

There is some analytical research available on the performance of self-organizing linear lists in nonstationary environments, especially using the move-to-front heuristic. It has been shown that the move-to-front rule is the best rule known so far for use in a nonstationary environment context, especially with high levels of locality of reference. There has been no research, to our knowledge, that suggests novel ways to reorganize lists in such environments. The aim of this Thesis is to suggest such ways, thus achieving better access times as well as providing insight into new ways to handle environments with dependent accesses.

Chapter 3

Managing Lists Through Sublist Manipulation

3.1 Introduction

In the previous chapter, we saw how the research on self-organizing lists with dependent accesses have mainly focused on the move-to-front heuristic, where it is generally agreed upon that this is the best heuristic to use in such case. This is not unfounded, since the move-to-front heuristic is well-suited for operation in such environments, because of its ability to quickly adapt to environmental changes, and to rapidly converge after such change.

In this Chapter, we first discuss the different models of dependence in the environment queries. We present two such models, the Markovian and the Periodic dependent query generators. We proceed by introducing the main direction of this Thesis: a new way to approach list organization in a nonstationary environment, by regarding the list as a series of sublists, and manipulating the elements within the sublists *as well as* the sublists themselves. We discuss the motivation behind such idea, and present the results of a simulation run. We conclude by discussing the

results, and highlighting the need to improve the algorithm by introducing clustering algorithms that allow the records to move within the sublists.

3.2 Models of Dependence

Narendra and Thathachar [24] define nonstationary environments as those where the penalty probabilities corresponding to the various actions vary with time. In our context, this means that the expected search cost for servicing a request does not correspond to the stationary (offline) expected search cost, but rather, it varies with time. One reason for this occurring is when the environment is characterized by “dependent accesses” or “locality of reference.” The *dependent accesses* or *locality of reference* phenomenon manifests when the number of distinct queries in a small part of the access sequence is considerably smaller than the number of distinct queries in the entire access sequence; i.e., if there is a probabilistic dependence between consecutive queries [6].

In order to study self-organizing lists in environments with dependent accesses, we need to define our models of dependence first. We will be using two different kinds of dependent query generators: Markovian and Periodic query generators.

To initiate the study, we assume that the set of n distinct query elements can be divided into equal k disjoint subsets with m elements, where $n = k \cdot m$. These subsets are sometimes referred to as *local sets* or *local contexts*. Elements of the same subset are “dependent” on each other, i.e., if an element from set i is requested at time t , the likelihood that the element requested at time $t + 1$ is from the same local context is significant.

The environment can be regarded as having a set of finite states $\{Q_i | 1 \leq i \leq k\}$. The way the environment transitions from one state to another defines the type of dependence model.

3.2.1 Markovian Switching Environment (MSE)

In a Markovian switching environment, the states of the environments are also the states of an (unkown) Markov chain, and the transition between these states happens in a Markovian manner. After generating the request, the environment stays in the same state with probability α , and makes a transition to another state with probability $(1 - \alpha)/(k - 1)$.

Consider, for example, an environment with $n = 128$ distinct records and $k = 4$ subsets with $m = 32$ elements in each subset. Let the set of elements produced in states Q_1, Q_2, Q_3, Q_4 be $\{1 \dots 32\}, \{33 \dots 64\}, \{65 \dots 96\}, \{97 \dots 128\}$ respectively. Further, let us define the dependency factor $\alpha = 0.9$. The transition matrix for such a Markov chain that characterizes the environment is given below.

$$\begin{bmatrix} 0.9 & \frac{0.1}{3} & \frac{0.1}{3} & \frac{0.1}{3} \\ \frac{0.1}{3} & 0.9 & \frac{0.1}{3} & \frac{0.1}{3} \\ \frac{0.1}{3} & \frac{0.1}{3} & 0.9 & \frac{0.1}{3} \\ \frac{0.1}{3} & \frac{0.1}{3} & \frac{0.1}{3} & 0.9 \end{bmatrix}.$$

After a state is obtained, the query generation thus follows a fixed probability distribution to facilitate analysis. Suppose that such distribution is represented by the vector $(p_1, \dots, p_m)$. Let P be the $(m \times m)$ matrix where all the lines are identical to the vector $(p_1, \dots, p_m)$. Bachrach *et al.* [6] showed that a generalized $(k \cdot m \times k \cdot m)$ Matrix that combines the dependency factor and the probability distribution. The result is an n state Markov chain that fully describes the behaviour of the Markovian query generator, and is given by:

$$\begin{bmatrix} \alpha P & \frac{1-\alpha}{k-1}P & \dots & \frac{1-\alpha}{k-1}P \\ \frac{1-\alpha}{k-1}P & \alpha P & \dots & \frac{1-\alpha}{k-1}P \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1-\alpha}{k-1}P & \frac{1-\alpha}{k-1}P & \dots & \alpha P \end{bmatrix}.$$

Bachrach *et al.* [6] observed that the expected number of subsequent requests in a local set is given by $1/(1 - \alpha)$.

3.2.2 Periodic Switching Environment (PSE)

In the periodic switching environment, the environment changes states in a round-robin fashion. After every T queries, the environment changes state from Q_i to $Q_{i+1 \bmod k}$. In other words, within these T queries, all the requests belonged to exactly the same local set.

Although this environment looks simple, it does have real life applications. Consider for example a global database that is accessed from different parts of the world. It may be the case that the database is most often accessed from New York five hours after its peak access from London. Other examples where the accessed data changes periodically may include traffic management (rush hour versus past rush hour), telephone resource management (beginning of the business day versus its end), etc.

There are two variations for this environment. In the first, the algorithm which manipulates the data structures is aware of the time after which the environment changes its state. In the second, the algorithm does not know about this time period. It should be the case that the algorithm's performance is better given this extra information. In any case, we hope that the algorithm responds more quickly to this environment than the Markovian one, since, within the period, all the queries are guaranteed to come from the same group.

Figure 3.1: A list with 10 records viewed as an unordered heap.

3.3 Motivation for Sublist Manipulation

In this section, we outline our motivation to explore an alternate way to manage the list. We then introduce the concept of sublist manipulation, and illustrate the associated problems.

3.3.1 Alternate Ways to Manage Lists

Although the idea of considering a list as a collection of other data structures may sound unreasonable initially, it is, indeed, often beneficial to do so. One obvious example is the swap-with-parent and move-to-parent algorithms, already discussed in the previous chapter. In these algorithms, the list was considered as an unordered heap or a tree, and the list reorganization took place with respect to an element's parent in the “imaginary” tree (Figure 3.1). This leads to improved access times for the asymptotic cost as well as drastically reduces the time needed for convergence. Dong [11] reported that the idea of utilizing a heap was inspired from self-organizing binary search trees, where the accessed record, under certain conditions, is rotated upwards to its parent's level.

Another example is the doubly exponential tree data structure, suggested by Iacono [17], which considers the list as a series of increasing-sized queues and trees, which leads to much reduced access times ($O(\log n)$ at worst). We will be discussing

Iacono's data structure in the following section.

3.3.2 Working Sets

Working sets are another property that is common to a lot of algorithms dealing with environments characterized by locality of reference. Informally, the algorithm's attention is focused on a subset of the data structure (the working set), (for example, caching the memory pages in a virtual memory paging algorithm), with the hope that records that have been requested recently will be requested again shortly, and thus can be retrieved quickly (e.g., without page faults).

One good example for working sets in the context of self-organizing data structures is the data structure suggested by Iacono [17]. The data structure maintains a series of balanced binary search trees $t_1, \dots, t_k$ and a series of queues $q_1, \dots, q_k$. The queues can be viewed as a list if they are strung together. For any $1 \leq i \leq k$, the size of q_i equals the size of t_i . Furthermore, the contents of q_i and t_i are identical. The tree series is doubly-exponential, meaning that the size of $t_i = 2^{(2^i)}$. If there are n total elements in the data structure, then the number of trees and queues k is $O(\lg \lg n)$. Notice that if the queues are strung together they would form a list (See Figure 3.2).

When an element x is requested, it is first searched for in t_1 . The search time is $\lg 2^{(2^1)} = 2$. If it is not located, the element is searched for in the following tree t_2 , involving time 2^2 , and so on, until found in t_i . Since the trees grow in a doubly-exponential manner, the search time grows exponentially. If x is found in t_i , it is dequeued from q_i and inserted in q_1 . To keep the queue and tree size constant, an element is dequeued from every $q_j, j < i$, and enqueued into q_{j+1} . The associated trees are updated accordingly. Notice that this is essentially a move-to-front scheme, where the search operation is done in "batches" of exponentially increasing time. The worst case is when the element is not found, in which case the running time is $O(\lg 2^{(2^k)}) = O(\lg 2^{(2^{\lg \lg n})}) = O(\lg n)$.

Figure 3.2: Iacono's data structure, consisting of a series of doubly-exponential queues and corresponding balanced search trees. If the queues are strung together, they form the original list.

Iacono's data structure is a clear demonstration of the benefits of considering different working sets while searching for elements in a move-to-front list. It requires $2n$ more storage however, for maintaining the lists and the queues. Iacono's data structure is also a good example of managing self organizing lists in a novel way, i.e., as a series of queues with increasing size.

3.3.3 Managing Sublists

We propose the idea of dividing the self-organizing list of size n into a set of k sublists. For the present, we assume that the sublists are all of the same size, but we may consider different sized lists as well. We would then apply self-organization algorithms hierarchically, i.e., on the elements within the sublist and then the sublists *themselves*. For example, we may choose to transpose the record within the sublist

Figure 3.3: Illustration of the MTF-MTF and the TR-MTF. (a) A record with key 5 is accessed in sublist i . (b) In MTF-MTF, the record is moved to the front of the sublist, and then the sublist is moved to the front of the list. (c) In TR-MTF, the record is transposed with the preceding record, and then the sublist is moved to the front of the list. The pointers used in the MTF-MTF algorithm are also shown.

and then move the sublist to the front of the original list. In our nomenclature, this would be called transpose-then-move-to-front, or *TR-MTF*. Another example would be to move the record to the front of the sublist, and then move the sublist to the front of the list, which would be *MTF-MTF* (See Figure 3.3). We will be examining different combinations of algorithms in a later chapter.

The rationale behind such a strategy can be explained as follows. In a non-stationary environment, when an element is accessed, the likelihood that another element within its local set is going to be chosen in the immediate future is high. It is therefore beneficial to make use of this information by promoting the entire set to an advanced position within the list, thereby reducing the search time when (and

if) such element is requested. This is somewhat related to the benefits one would gain from using the MTF rule in a non-stationary environment, since the frequently accessed records are “crammed” at locations near the front of the list. The hierarchical approach has several advantages, however. Consider the MTF-MTF algorithm, for example. When compared to the MTF, it has the following advantages:

- It promotes the records *en masse*, and therefore increases the prospects of the access time being reduced in forthcoming accesses. This is, clearly, an advantage in environments characterized with locality of reference, because it will take at least n/k distinct requests to bring the entire context to the front of the list under the MTF.
- If a record outside of the local context is accessed, it is quite likely that it will be displaced much further from the front in subsequent calls than in the MTF case. This is because, in subsequent calls, the group of all records that obey the context may be promoted to precede it. In the case of the MTF, however, such an “unfortunate” access may cause the record to linger at the front of the list for a much longer duration, i.e., until the entire context is moved to the front of the list, one element at a time.

Notice that the storage requirements for such a list maintenance strategy is at most $O(k)$, to store the locations of the sublist boundaries. In fact, one may need only constant additional storage, if the sublists are equally sized, in which case one can determine the sublist which a particular element belongs to while searching for that element. Additionally, the cost to manipulate the sublist is constant because it only involves pointer assignments.

In its simplest form, however, this idea is not good enough, as we shall soon explain. The benefits outlined above assume that the sublists mirror the dependence relationship between the records. This is not necessarily the case, because as the list

is initially arranged in an arbitrary order, the sublists will most likely not represent the dependence relation *between* the records. Unless elements are rearranged within the sublists such that they eventually mirror such dependence, the concept which we have introduced will, indeed, often produce inferior results. We shall now present the idea algorithmically and discuss the associated results. The partitioning refinement is the subject of subsequent chapters.

3.4 Sublist Manipulation Algorithms

In this section, we will formally define one of the hierarchical sublist manipulation algorithms, the MTF-MTF. We will present the associated implementation issues, and provide experimental results comparing its performance with other well-known algorithms. We will conclude the section by discussing some reasons for the results obtained.

3.4.1 The MTF-MTF Algorithm

Recall that the list of n records would be divided into k consecutive sublists with m elements each, and that the MTF-MTF algorithm involves moving the accessed element to the front of the sublist that it is found in, and then moving the sublist itself to the beginning of the list. This means that we need to determine the beginning and the end of that sublist. There are two ways to achieve this: we either maintain a data structure with pointers to the list boundaries, or we determine the list boundaries during the linear search for the accessed record.

The first solution entails maintaining a data structure of $O(k)$ storage. This data structure can be of a random access sort (e.g., an array) where the index indicates the sublist number and the key points to the list boundaries. Such a data structure would require $O(k)$ additional time for rearrangement after the sublists have been

reordered. Alternatively, the data structure can be of a linear access sort (e.g., a list in itself), and can be updated in constant time, but that will require $O(k)$ time to obtain the sublist boundaries.

The second solution involves constant additional storage, but it will require $O(m)$ additional probes to locate the tail of the sublist. We will present this version of the algorithm as it is easier to present and implement.

The algorithm maintains four pointers (See Figure 3.3):

- **current** points to the currently probed record.
- **prev** points to the record *previous* to the currently probed record.
- **currentTail** points to the tail of the sublist where the requested record is found.
- **prevTail** points to the tail of the sublist *preceding* the sublist where the requested record is found.

Notice that we are maintaining pointers to the *tails* of the sublists because, since we are dealing with a singly linked list, the only way to detach a sublist is to change the pointer at the sublists' tails. In addition to the pointers above, we have a pointer **head** pointing to the head of the list, and a variable **counter**, which is incremented every record probe modulo m . Also, in the interest of simplicity, we will be moving the sublist to the front first, and then moving the element to the front of the list.

Algorithm MTF-MTF

Repeat for every access to record R

```

current = head
prev = Null
prevTail = Null
currTail = Null
counter = 0

```

```

While curr → element ≠ R
    If counter = m - 1
        prevTail = current
        counter = (counter + 1) mod m
        prev = current
        current = current → next
    If prevTail ≠ Null
        /* First, find the tail of the sublist */
        currTail = current
        While counter ≠ m - 1
            currTail = currTail → next
            counter = (counter + 1)
        /* Now the pointers are matching Figure 3.3 */
        /* Move sublist to front */
        newHead = prevTail → next
        prevTail → next = currTail → next
        currTail → next = head
        head = newHead
    If current ≠ head
        /* Move record to the front of the sublist */
        prev → next = current → next
        current → next = head
        head = current
End Algorithm MTF-MTF

```

The algorithm can be easily modified to implement the TR-MTF, by swapping the contents of *current* and *prev*, which are readily available in the algorithm, instead of moving *current* to the front.

Scheme	Zipf	80-20	Lotka	Exponential	Linear
MTF	32.03	39.08	4.57	2.52	53.30
TR	23.80	29.97	3.61	2.26	43.68
MTF-MTF	52.05	57.55	18.00	13.51	62.26
TR-MTF	48.33	55.67	20.54	16.91	62.00
MTF	32.16	39.24	4.62	2.52	53.09
TR	24.32	30.42	3.84	2.33	44.30
MTF-MTF	52.20	57.64	17.98	13.49	62.45
TR-MTF	48.40	55.51	20.46	16.92	61.92

Table 3.1: Experimental results for the asymptotic cost (top portion) and the amortized cost of the MTF, TR, MTF-MTF, TR-MTF rules for a list of size 128 and 8 sublists with size 16 each.

Table 3.1 compares the performance of the MTF-MTF and the TR-MTF against the MTF and the TR, in terms of both the asymptotic and amortized cost. To measure the asymptotic cost, we let the experiment run for 250,000 requests to converge, and then measure the average cost for 50,000 more requests. For the amortized cost, we calculate the average cost for the entire 300,000 requests. The list size was 128, with 8 sublists of 16 records each. The records' initial ordering followed a uniform distribution. The different access distributions are formally defined in 5.1.

3.4.2 Discussion

It is clear that the performance of the MTF-MTF and TR-MTF lag behind that of the MTF and TR substantially. In fact, for the linear distribution which has a mean of 64, the algorithms are hardly even expedient with scores around 62.

The reason for such a poor performance is that we do not allow for elements to move between list boundaries, and therefore benefit from the “working set” property alluded to earlier, and the locality of reference. Indeed, we need to have the elements of the list rearranged, mirroring an underlying partitioning, in which elements that are most frequently accessed together would be grouped in the same sublist.

Consider, for example, a sublist, L_i , with a total access probability that is greater than that of a record R_j . In other words, $\sum_u s_u > s_j$ where $R_u \in L_i$. In the MTF-MTF, this will place all the records in L_i in front of R_j , when in fact many of the *individual* records in L_i may have lower access probabilities than R_j , resulting in a poorly arranged list. As the initial list arrangement is arbitrary, the probability of this scenario occurring is very high.

This leads us to our discussion of the object partitioning problem in the next Chapter, as a means for capturing the underlying dependence of the record queries in the sublists.

Chapter 4

Capturing Dependence

We have seen in the previous Chapter that hierarchical sublist organization is not meaningful unless the sublists being organized reflect the dependence pattern for access queries. We, therefore, need to investigate techniques that not only discover such patterns, but also those which react adaptively to changes in such environments, in order to incorporate them in our list organizing strategy.

We begin the Chapter with a brief overview of clustering techniques. We proceed by discussing the Object Partitioning Problem in detail, and discuss solutions that help us in our strategy. In the process, we shall provide a brief introduction to learning automata.

4.1 Clustering

Clustering analysis is an important tool that is often discussed in the fields of pattern recognition and data mining, which deal with recognizing patterns and extracting knowledge from large amounts of data, respectively. It is used in a variety of applications, including unsupervised pattern classification, data analysis, image processing and market research, to discover the overall distribution patterns and correlations

among data attributes [13].

Clustering is the process of grouping data objects in classes that maximize the intra-class similarity and simultaneously minimize the inter-class similarity. That is, after clustering, objects within a class must have high degree of similarity with one another, whereas those which are dissimilar should fall into other classes, with respect to the classification metric. Unlike classification, cluster analysis does not rely on predefined classes to perform the objects grouping, since the category used for the clustering process is not known to the algorithm. Han and Kamber [13] categorized clustering algorithms into partitioning methods, hierarchical methods, density-based, grid-based, and model-based methods. A clustering technique can fit in more than one category, as we shall see later in the Chapter.

In partitioning methods, the goal is to construct k partitions out of n data objects, where each partition represents a cluster and $k \leq n$. Each group must contain at least one object, and each object must belong to exactly one group, although the latter requirement is sometimes relaxed in some fuzzy partitioning techniques.

To achieve an absolutely optimal partitioning-based clustering would require an exhaustive search of all the possible partitions. As this is prohibitively costly, we need to resort to adaptive heuristics instead. Two such heuristics, that are very frequently used, are the k -means and the k -medoids algorithms. In the k -means algorithm, the objects are initially grouped into arbitrary clusters. Each cluster is represented by the mean value of the objects in it. The objects are then “attracted” to the different clusters’ means, which could be thought of as their “centres of gravity”. Objects migrate from one cluster to another with a closer mean, and the different means change accordingly, until convergence is achieved.

As a k -means method is only useful when the mean of the cluster is defined, the k -medoids heuristic represents each cluster by one of the objects that is located near the center of the cluster (a medoid). Different objects are then attracted to the

medoid instead of the mean. This method is generally more robust than the k -means method as it is less sensitive to *outlier* data points, which are objects that do not comply with the general behavior or model of the data. In both these methods, the necessity to specify k can sometimes be seen as an advantage.

In a hierarchical clustering method, the set of data objects is arranged hierarchically. One way for such grouping is the agglomerative or “bottom-up” approach, where the objects are initially placed in separate groups and then groups that are closer to each other are successively merged into larger groups. In the divisive “top-down” approach, the objects start in the same cluster and are successively split into smaller clusters, until a termination condition holds.

In density-based methods, the objects are clustered based on the distance between them. These methods regard clusters as dense regions of objects in the data space that are separated by regions of low density, and the objective is to recognize and separate the dense regions from one another. In grid-based methods, the object space is quantized into a finite number of cells, forming a grid. The clustering operations are then performed on the grid. As the object space is discrete, such methods are characterized with fast processing times. Finally, in model-based methods, the objective is to find the best fit of clusters in a hypothesized probability model.

4.2 Object Partitioning Problem

Suppose we have a set of W objects. We need to place these objects into R disjoint subsets, such that objects that are accessed more frequently together lie in the same group. The joined access probabilities are assumed to be unknown, to render the problem nontrivial. This problem is known in the literature as the object partitioning problem (OPP). When the subsets are desired to be of equal cardinality, the problem is referred to as the equal partitioning problem (EPP). The OPP is thus a clustering

problem where the criterion for clustering is frequent co-joined accesses of the objects.

There are various applications for the object partitioning problem. One real life setting would be in a library environment, where the set of disciplines (e.g., history, geography, biology, etc.) is desired to be grouped such that books of related disciplines are closer to each other. For example, a library patron who needs a book in computer science is more likely to need another in mathematics than geography. It is therefore preferable to locate books of computer science and mathematics in closer proximity than those of computer science and geography.

Other computer science applications of the OPP have been reported in [30]. For example,

- In a database environment, it may be needed to partition the set of attributes that belong to a relation in the normal form, such that attributes that are accessed simultaneously in a query are located in the same group.
- In file organization, it is required to partition records that are simultaneously accessed to be on the same block or track of disk, to minimize access time.
- In a distributed database, the relations are distributed in various remote sites. It would be desirable if data that is jointly accessed is physically located in the same geographical site, to minimize communication costs.

In our context, it is, indeed, desirable to partition the list such that records that have a high probability of simultaneous access are located together, to benefit from the idea of sublist manipulation, and to thus overcome the otherwise poor results obtained. We shall discuss the best results known for the object partitioning problem. In the process, we will briefly discuss background information relating to learning automata.

4.2.1 The Basic Adaptive Method (BAM)

Until 1988, the best algorithm for the EPP was due to Yu *et al.* [41], called the Basic Adaptive Method (BAM). Associated with each physical object in $\{A_1, \dots, A_W\}$, they defined an abstract object in the set $\{O_1, \dots, O_W\}$. BAM manipulates these abstract objects as the queries are processed. Periodically (i.e., every T queries), the physical objects are repartitioned to conform with the partitioning of the abstract objects dictated by the BAM.

Initially, each abstract object O_i is assigned a real number X_i . Queries for records are conceptually perceived as a pair. When records A_i and A_j are requested, X_i and X_j are moved toward their centroid by a constant Δ_1 . Subsequently, two distinct random abstract objects O_p and O_q are chosen, $1 \leq p, q \leq n$, and X_p and X_q are moved away from *their* centroid by a constant Δ_2 . After a fairly long time, the partitions are created based on the proximity of X_i 's. An algorithmic version of BAM is given below.

Algorithm BAM

Initialization

Initialize arbitrarily X_i for all $1 \leq i \leq n$.

Repeat forever

For a sequence of T queries (A_i, A_j) ,

If $(X_i < X_j)$

$$X_i = X_i + \Delta_1$$

$$X_j = X_j - \Delta_1$$

Else

$$X_i = X_i - \Delta_1$$

$$X_j = X_j + \Delta_1$$

Randomly select distinct indexes p and q , $1 \leq p, q \leq n$

If $(X_p < X_q)$

$$X_p = X_p - \Delta_2$$

$$X_q = X_q + \Delta_2$$

Else

$$X_p = X_p + \Delta_2$$

$$X_q = X_q - \Delta_2$$

Determine the partitions based on the proximity of X'_i s.

End Algorithm BAM

The BAM can be categorized as a partitioning clustering method in which, rather than moving the objects to the clusters' means or medoids, the co-accessed objects are moved towards each other. The BAM has the benefits of being adaptive and not having to compute or store any query statistics. However, it takes quite a long time to converge, rendering it inefficient when quick convergence is desired. BAM also utilizes floating point operations, which are quite costly compared to integer computation. Finally, observe that the last step of the algorithm, which involves determining the partitions based on the proximity of the abstract objects, is not trivial. Indeed, it can be thought of as a density-based clustering problem of its own.

4.2.2 FSSA-based solutions for the OPP

In 1988, Oommen and Ma [30] proposed a truly innovative solution to the equal partitioning problem using learning automata techniques. Learning automata have been developed to model biological learning systems in learning the optimal action which an environment offers. The automaton can be thought of as a machine operating in a stochastic environment. The machine can choose an *action* from a finite set of actions, which is made available to the Environment. The Environment, in turn, responds to the action by producing a *reward* or *penalty*, that is a stochastic function of the input action. The penalty probabilities of different actions are unknown to the

Figure 4.1: Learning automaton's interaction with the environment

automaton. The learning process is done by processing the Environment's responses to the actions chosen by the automaton (See Figure 4.1). A learning automaton is one that learns the optimal action, i.e., produces the action with minimum penalty probability more frequently than other actions.

Applications of learning automata include parameter optimization, pattern recognition, and telephone routing. We refer the reader to Narendra and Thathachar's reference textbook [24] on the subject for further information. Here, we will only provide a brief introduction to one category of learning automata, the family of Fixed Structure Stochastic Automata (FSSA). Later, we shall discuss the other category, the Variable Structure Stochastic Automata (VSSA).

An FSSA is a quintuple $(\underline{\alpha}, \underline{\Phi}, \underline{\beta}, F, G)$, where

- $\underline{\alpha} = \{\alpha_1, \dots, \alpha_R\}$ is a finite set of actions that the automaton chooses from.
- $\underline{\beta} = \{0, 1\}$ is the set of Environment outputs (which are the possible automaton inputs). 0 corresponds to reward, and 1 corresponds to penalty.
- $\underline{\Phi}$ is a finite set of automaton states.
- F is a map from $\underline{\Phi} \times \underline{\beta}$ to $\underline{\Phi}$, called the *transition function*. It is the means by

which the automaton chooses its next state, $\phi(t+1)$, based on its current state $\phi(t)$ and the input. F may be either deterministic or stochastic.

- G is a map from $\underline{\Phi}$ to $\underline{\alpha}$. It is the means by which the automaton chooses the action given its current state. G may be either deterministic or stochastic, although it is easy to show that for any FSSA with a stochastic G , an equivalent FSSA with a deterministic G can be obtained [24].

The Environment has a set of penalty probabilities $\{c_i\}$, with which it penalizes the automaton. In other words,

$$c_i = Pr\{\beta(t) = 1 | \alpha(t) = \alpha_i\},$$

where $\beta(t)$ is the penalty value at time t , $\alpha(t)$ is the chosen action at time t , and $1 \leq i \leq R$. Naturally, the c_i 's are not known to the automaton.

The finite set of states $\underline{\Phi}$ and the fixed state transition probabilities give the automata their “fixed structure” nature. The nature of F determines whether the automaton is stochastic or deterministic. The transition function F can be represented either in the form of matrices or graphs. The state sequence of fixed structure stochastic automata with a stationary random binary input can be represented by a Markov chain, where the states of the Markov chain correspond to the states of the automaton, and the state transitions can be represented by F . As the transition probabilities in F are constants, the state sequence forms a homogeneous Markov chain that can be studied through the theory of Markov chains.

An automaton is said to be *expedient* if the expected penalty at time t , $E[M(t)]$, is less than the initial expected penalty, M_0 , as t tends towards infinity. The automaton is *optimal* if $E[M(t)]$ equals the minimum penalty probability in the limit as time goes towards infinity. Finally, the automaton is ϵ -*optimal* if, in the limit, $E[M(t)] < c_L + \epsilon$, where c_L is the minimum penalty probability, for any arbitrary $\epsilon > 0$. That is, the

expected penalty can be pushed as close to the minimum probability as desired.

Solving the EPP using the Tsetlin and Krinsky automata

It is, perhaps, best to illustrate FSSA with an example, in our context of discussing the equal partitioning problem. Oommen and Ma [30] proposed two solutions for the EPP using the classic Tsetlin and Krinsky automata (See [24] for details of these automata). The $L_{KN,K}$ Tsetlin automaton has K actions $\{\alpha_1, \dots, \alpha_K\}$ and KN states $\{\phi_1, \dots, \phi_{KN}\}$. N is called the “depth” of the automaton.

The G map is deterministic and is defined as follows:

$$G(\phi_j) = \alpha_i \quad \text{if } (i-1)N + 1 \leq j \leq iN.$$

Effectively, the automaton is divided into K groups, each of length N . The first group contains states 1 through to N , the last one contains states $(N-1)K + 1$ through to KN , and the i^{th} one contains states $(i-1)N + 1$ through to iN . α_i is chosen if the current state lies in group i .

Graphically, the automaton states are drawn as a collection of rays with a common centre. On the edges of the rays lie the *extreme* or *innermost* states $1, \dots, (i-1)N + 1, \dots, (N-1)K + 1$, and at the centre lie the *boundary* states $N, \dots, iN, \dots, KN$ (See Figure 4.2 for a two state Tsetlin automaton). Notice that the *innermost* states are innermost with respect to their action, and not with respect to the graphical schematic.

What we have to finally define is the transition function, F . Upon reward, the current state of the automaton moves one outward step to the edge of the ray. If the current state is an extreme state, it does not change. On penalty, the state moves inward towards the centre. If the current state is a boundary state, it “jumps” to the

Figure 4.2: The two-action Tsetlin automaton.

Figure 4.3: The two-action Krinsky automaton.

boundary state of the following group, modulo K . Thus, for all $1 \leq i \leq K$,

$$\begin{aligned} F_{\text{Tsetlin}}(\phi_j, 0) &= \phi_{j-1} && \text{if } (i-1)N + 1 < j \leq iN \\ &= \phi_j && \text{if } j = (i-1)N + 1, \end{aligned}$$

and,

$$\begin{aligned} F_{\text{Tsetlin}}(\phi_j, 1) &= \phi_{j+1} && \text{if } (i-1)N + 1 \leq j < iN \\ &= \phi_{((i+1)N) \bmod KN} && \text{if } j = iN. \end{aligned}$$

The Krinsky automaton is identical to the Tsetlin automaton, except that when the automaton is being rewarded, it jumps directly to the extreme state of the current group (Figure 4.3). The corresponding transition rule for reward inputs is therefore:

$$F_{\text{Krinsky}}(\phi_j, 0) = \phi_{(i-1)N+1} \quad \text{if } (i-1)N + 1 \leq j \leq iN.$$

Tsetlin's automaton is known to be ϵ -optimal in all random environments in which the minimum penalty probability is less than 0.5. Krinsky's automaton is known to be ϵ -optimal in all random environments [24]. In general, it is observed that the asymmetric treatment of rewarding and penalizing environment outputs yields better results than treating them symmetrically.

We can now proceed to describe how these automata can be used to solve the EPP, as shown in [30]. We choose the automaton's actions to correspond to the various partitions of the abstract objects. Assume that at time t , the automaton chooses action α_k . Let the query at time $t+1$ be for elements (A_i, A_j) . This query either supports or contradicts the partitioning represented by α_k . If α_k represents a configuration whereby A_i and A_j are grouped together, the automaton is rewarded and the learning automaton moves outward according to the relative transition function

(i.e., the state of the machine changes to be one that is outward from the center), and consequently the same action is chosen again. If not, however, then the automaton is penalized and the state is changed towards the center. If the current state had been an boundary one, the automaton jumps to another state group, and chooses another action.

In theory, such solutions can produce good results that can be as close to optimality as desired. Indeed, Oommen and Ma reported impressive results for the Tsetlin automaton, and even better results for the Krinsky automaton as opposed to the BAM. Queries were randomly generated based on an underlying distribution unknown to the partitioning algorithm. If π_i was the partition in which A_i was *to be* located, then these queries obeyed the following distribution:

$$\sum_{A_j \in \pi_i} \Pr\{A_i, A_j \text{ accessed together}\} = p.$$

p is called the *joint access probability*. In the case where the number of objects was 4 and the number of groups was 2, it took BAM 80 iterations to converge to the optimal solution, compared to 14 for Tsetlin and 7 for Krinsky, with the joint access probability $p = 0.9$. When the number of objects was 6 and the number of groups was 2, the convergence times for BAM, Tsetlin and Krinsky were 240, 20 and 12, respectively. There is a clear order of magnitude of difference between BAM and FSSA-based solutions.

A serious practical issue arises from this solution, however. Oommen and Ma have shown that if W is the number of objects to be partitioned, and R is the number of desired partitions, then the number of actions for a Tsetlin or Krinsky automaton to solve the EPP is $\frac{W!}{((W/R)!)^R R!}$. This puts a clear limit on the practicality of this solution in systems where the number of objects or partitions is anywhere larger than the small numbers used in the simulations.

4.2.3 The Object Migrating Automaton (OMA)

To overcome this limitation, Oommen and Ma [30] suggested a new creative use for FSSA to solve the EPP. The automaton defines actions only as much as the number of classes needed for partitioning the objects. For each action, there is a fixed depth N . Instead of having the automaton move towards an action (class partitioning) itself, they have the abstract objects *themselves* move around the automaton. If the abstract object O_i is in action α_k , then O_i should be placed in the k^{th} grouping. Consequently, if O_i and O_j are in the same action, then they belong to the same grouping.

The automaton defines its requirements in a manner that is very similar to that of Tsetlin's. For each action α_k , there is a set of states $\{\phi_{k1}, \dots, \phi_{kN}\}$, where N is the depth of the automaton. We consider ϕ_{k1} to be the extreme state of that action and ϕ_{kN} to be its boundary state. The automaton can be thought of as having multiple "current states," one corresponding to each abstract object, as opposed to the standard FSSA which have only one such state. When a query requests objects (A_i, A_j) , if the corresponding abstract objects O_i and O_j lie in the same class, say, α_k , both objects are rewarded by being moved one step towards the extreme state ϕ_{k1} (Figure 4.4(a)). If the objects lie in different classes, say α_k and α_m respectively, the objects are updated in the following manner:

- If both objects are not in boundary states, we move both objects one step towards their boundary states (Figure 4.4(b)).
- If only one object (say O_j) is at a boundary state, we move the other object O_i towards its boundary state ϕ_{kN} (Figure 4.4(c)).
- If both objects are in boundary states, then randomly choose an object (say O_i) to be moved to the boundary state of the other objects class, ϕ_{mN} . To keep

the objects equally partitioned, we have to move back an object from α_m to the original class α_k . We choose to move one that is closest to the boundary state, ϕ_{mN} (Figure 4.4(d)).

We now provide an algorithmic version of OMA. We shall use the expression ξ_i to indicate the state number of the abstract object O_i . ξ_i can take values from 1 through RN . If $(k-1)N+1 \leq \xi_i \leq kN$, then object O_i is assigned to class k (i.e., action α_k)

Algorithm OMA

Initialization

Initialize arbitrarily ξ_i for $1 \leq i \leq W$ evenly among R classes, each class having W/R objects.

Repeat forever

For a sequence of T queries (A_i, A_j) ,

If $(\xi_i \text{ div } N) = (\xi_j \text{ div } N)$ /* The partitioning is rewarded */

If $(\xi_i \text{ mod } N) \neq 1$ /* Move O_i towards extreme state */

$\xi_i = \xi_i - 1$

If $(\xi_j \text{ mod } N) \neq 1$ /* Move O_j towards extreme state */

$\xi_j = \xi_j - 1$

Else /* The partitioning is penalized */

If $(\xi_i \text{ mod } N) \neq 0$ and $(\xi_j \text{ mod } N) \neq 0$

/* Both are non-boundary states */

$\xi_i = \xi_i + 1$

$\xi_j = \xi_j + 1$

Else If $(\xi_i \text{ mod } N) \neq 0$ /* Only O_i is in non-boundary state */

$\xi_i = \xi_i + 1$

Else If $(\xi_j \text{ mod } N) \neq 0$ /* Only O_j is in non-boundary state */

$\xi_j = \xi_j + 1$

Else /* Both are in boundary states */

Figure 4.4: Transition rules for two-action OMA. On reward, we move the abstract objects towards the extreme states (a). On penalty, move the objects towards their boundary states (b), (c), or move the object to the boundary state of the other action (d).

```

temp =  $\xi_i$  /* store the state of  $O_i$  */
 $\xi_i = \xi_j$  /* move  $O_i$  to  $O_j$ 's group at the boundary state */
t = index of an object in group of  $O_j$ 
where  $O_t$  is closest to  $\xi_j$ 
 $\xi_t = \text{temp}$ 

```

Present partitions based on the groupings of ξ_i 's

End Algorithm OMA

Analysis and Results

Although the OMA is a partitioning clustering method, it is also a grid-based one, since the partitioned objects can assume a finite number of states. Grid-based methods are famous for quick convergence time, and the OMA is no exception. Oommen and Ma [30] reported experimental results for the OMA that were clearly better than the BAM. For example, when the number of objects to be partitioned (W) is 18 and the number of groups (R) is two, the BAM required 3352 iterations to converge, whereas the OMA needed only 277 ($p = 0.9$). Oommen and Ma also showed that the number of iterations needed for convergence for the BAM grew at a much faster rate than the OMA, with increasing numbers of objects. Although the OMA figures are not as good as those for the Tsetlin and Krinsky automata, the OMA has the huge advantage of having quite feasible memory requirements.

But the benefits of the OMA extend beyond that. For example, it is easily implemented, as shall be explained later. Its space requirements are comparable to those of the BAM. Also, since it only uses integer arithmetic, where only three integers are updated in the worst case (when both objects are at boundary states), the amount of computation required by the OMA per iteration is less than the BAM, which uses four floating point operations per iteration, thereby reducing the overall running time. In addition, the BAM invokes a random number generator twice in every iteration.

Finally, the OMA does not need to do any computation to determine the partitioning, as it is readily available through the indexes of the automaton's states.

Oommen and Ma did not provide formal analysis for the OMA except for the special case when $W = 4$ and $R = 2$, in which case they proved the automaton's expediency. Although a general analysis of the automaton's performance is hard to obtain, the simulation results seem to suggest the OMA is ϵ -optimal.

Implementation Details

The OMA can be implemented effectively in few lines of code. At creation time, the OMA is configured with the number of objects to be partitioned W and the number of groups R . The main data structure is a simple integer array $\mathcal{A}$ of length W . The array holds the state number for every object, referred to above as ξ_i .

Initially, the array is initializing by distributing the abstract objects evenly on the boundary states of the various actions. This is done through the following rule:

$$\mathcal{A}[i] = (((i - 1) \bmod R) + 1) \cdot N \quad (4.1)$$

for all $1 \leq i \leq W$. The result is that each action has exactly W/R objects, assuming that $W \bmod R = 0$. Notice that we can also determine the group G_{A_i} to which an element A_i belongs through the equation:

$$G_{A_i} = ((\mathcal{A}[i] - 1) \operatorname{div} N) + 1. \quad (4.2)$$

The operation of the automaton follows exactly the algorithm outlined above. The OMA expects queries to come in pairs (A_i, A_j) . To model this, a wrapper layer stores the last query that was received at time $t - 1$ and couples it with the current query at time t to form a query pair, which is then processed by the OMA. The current

query is then stored to be paired with the next query, and so on. The processing by the OMA is invoked after the second query.

There is no need to wait for T queries to report the partitioning. Instead, the automaton reports changes in the partitioning instantly in its return value, through the following code:

- 0: There has been no change in the object partitioning. The queried objects reside in the same partition and have been moved towards their extreme state, unless they were already at that state.
- 1: The elements were in boundary states of different actions. O_i has been moved to O_j 's group. The object that has been moved back to the original group is also reported.
- 2: The elements were in boundary states of different actions. O_j has been moved to O_i 's group. The object that has been moved back to the original group is also reported.
- 3: The elements were in different groups, but there had been no re-arrangement in the groupings because they were not both at boundary states. The elements have been moved towards the centre of the automaton.

Notice that, since we use a random access array, the running time for the OMA is often constant, because, given an element, we can instantly know the group it belongs to by examining its current state. However, the step in the algorithm which involves locating an element to be moved *outside* a group can be $O(W)$. Some implementation tweaks can be used to reduce such penalty to $O(W/R)$, as we shall see in Section 5.2.1.

The automaton has been implemented as a C++ object because it maintains a state. The query initialization and query routines can be coded with minimal effort.

4.2.4 The Stochastic Move Automaton (SMA)

The OMA has its own drawbacks that have been outlined by Oommen and Ma [31]. First of all, it is designed to only solve the equal partitioning problem, and not the generalized partitioning problem. It is imperative that the automaton is initialized properly such that each action has an equal amount of objects. Secondly, the OMA is sometimes prone to permanently anchor a group of objects in the same class, when a different arrangement might have been better. Consider for example that O_i and O_j are in the same class, referred to by action α_k . Further, suppose that the query pairs (A_i, A_m) and (A_j, A_n) are more frequent than the pairs (A_i, A_j) , (A_i, A_n) and (A_j, A_m) . The OMA may decide to migrate O_m and O_n to join O_i and O_j in the same action α_k . However, the different access frequencies suggest that the objects should not be grouped together; rather, that O_i and O_m should have been in one group, and that O_j and O_n should have been in another group.

A similar situation may arise in the case when a query requests objects that belong to different groups, which is a non-zero probability (i.e., an outlier query). Suppose (A_i, A_j) are accessed frequently, where O_i and O_j are in actions α_p and α_q . Both objects are in their boundary states, and the object migration is about to happen from one action (say α_p) to the other (α_q). Now suppose that the outlier queries (A_i, A_k) and (A_j, A_k) happened before (A_i, A_j) , where O_k is in a different action α_r . If O_k is at a boundary state, it could be the case that O_i and O_j are migrated to the “wrong” class α_r , and remain there for a long period of time.

Oommen and Ma [31] proposed a generalization to the OMA that avoids or minimizes the risk of these problems, called the Stochastic Move Automaton (SMA). In addition to being a partitioning clustering scheme, it also arranges the groups in a hierarchical (bottom-up) manner. The SMA is similar in structure and behaviour to the OMA, except for the following:

- In structure, the SMA defines W distinct actions for the W objects to be partitioned. Every abstract object is initially placed in its own class.
- The SMA does not mandate a fixed number of objects to be in the same class. When an object is migrated, there is no need to migrate another object back to the original class.
- An object can be migrated to another class if it is at a boundary state. There is no need that the other object in the query be in a boundary state as well.
- The migration of an object in a boundary state, out of its current action, is probabilistic. That is, on receiving a penalty, an object can move out of its current action with probability δ and stay in that action with probability $1 - \delta$.

The following is an algorithmic version of the SMA, which uses the same notation as in the case of the OMA. Notice that the expression $\lceil \xi_i / N \rceil$ gives the class for O_i , (and hence the corresponding action) for the automaton, where $1 \leq i \leq W$.

Algorithm SMA

Initialization

Initialize arbitrarily ξ_i for $1 \leq i \leq W$ evenly among R classes, each class having W/R objects.

Repeat forever

For a sequence of T queries (A_i, A_j) ,

If $(\xi_i \text{ div } N) = (\xi_j \text{ div } N)$ /* The partitioning is rewarded */

If $(\xi_i \text{ mod } N) \neq 1$ /* Move O_i towards extreme state */

$\xi_i = \xi_i - 1$

If $(\xi_j \text{ mod } N) \neq 1$ /* Move O_j towards extreme state */

$\xi_j = \xi_j - 1$

Else /* The partitioning is penalized */

```

If  $(\xi_i \bmod N) \neq 0$  and  $(\xi_j \bmod N) \neq 0$ 
    /* Both are non-boundary states */
     $\xi_i = \xi_i + 1$ 
     $\xi_j = \xi_j + 1$ 
Else If  $(\xi_i \bmod N) \neq 0$  /* Only  $O_i$  is in non-boundary state */
     $\xi_i = \xi_i + 1$ 
    /* Move  $O_j$  to  $O_i$ 's class with prob  $\delta$  */
    If  $(RAND < \delta)$   $\xi_j = \lceil \xi_i / N \rceil * N$ 
Else If  $(\xi_j \bmod N) \neq 0$  /* Only  $O_j$  is in non-boundary state */
     $\xi_j = \xi_j + 1$ 
    /* Move  $O_i$  to  $O_j$ 's class with prob  $\delta$  */
    If  $(RAND < \delta)$   $\xi_i = \lceil \xi_j / N \rceil * N$ 
Else /* Both are in boundary states */
    temp1 =  $\xi_i$ 
    temp2 =  $\xi_j$ 
    If  $(RAND < \delta)$   $\xi_i = temp2$ 
    If  $(RAND < \delta)$   $\xi_j = temp1$ 

```

Present partitions based on the groupings of ξ_i 's

End Algorithm SMA

Analysis and Results

The SMA also yields better results than BAM. For 18 objects to be partitioned into two classes, the number of iterations for SMA and BAM were 368 and 3352 respectively.

However, the accuracy of the SMA is not as good as that of BAM. With the joint access probability $p = 0.9$, the accuracy of BAM was 100%, and only 82% for the SMA ($\delta = 0.25$). When p is decreased to 0.6, the accuracy of BAM remained steady

at 100%, while it decreased to only 14% for SMA. The situation was even worse when the migration probability δ is increased to 0.5, at which case the accuracy of SMA dropped to 5% with $p = 0.6$. And although it is designed to be more flexible than the OMA, the SMA yields slightly worse results than the OMA in terms of convergence time.

4.2.5 VSSA-based solution for the OPP

Oommen and Ma observed that the behaviour of the SMA still suffers because of outlier queries that do not follow the underlying optimal partitioning, which happens with probability $1 - p$. Suppose O_i was moved to the “wrong” class because of the outlier query (A_i, A_k) . If (A_i, A_j) is a more frequent query and O_j resides in a different class, the automaton is likely to be penalized, and it would be desirable for the objects to leave their actions. If O_j happens to move into the same group as O_i and O_k , rather than the other way around, they would be considered to be in an unfavourable group, and it would be very hard for them to leave it. This inspired Oommen and Ma to look for an OPP solution using Variable Structure Stochastic Automata (VSSA), which can choose two different actions at two consecutive time instants.

Unlike FSSA, VSSA are automata whose structure evolve with time. Essentially, a VSSA (also called a *reinforcement scheme*) involves a probability vector $\underline{P} = [p_1, \dots, p_R]^T$ of the possible actions for the automaton. The action is chosen based on the probability vector. When the Environment responds to the action, an update rule is invoked to change the probability vector, and thus,

$$p_i(n+1) = f[p_i(n), \alpha(n), \beta(n)],$$

where $p_i(n)$ is the probability of choosing action α_i at time n .

Narendra and Thathachar [24] report that the linear reward-penalty (L_{RP}) is perhaps the earliest scheme considered in mathematical psychology. It involves updating the probability vector with a linear function. If α_i is chosen and the automaton is penalized, p_i is decreased by a factor $b < 1$. The decrease is equally divided among the other actions. If the environment is rewarded, all the other probabilities $p_j, j \neq i$ are decreased by a factor $a < 1$, to the benefit of p_i .

Recall that treating rewards and penalties asymmetrically has been shown to produce better results than, otherwise, treating them symmetrically. This is why the constants a and b should not be equal. Other famous reinforcement schemes with explicit asymmetric treatment of rewards and penalties are the linear reward-inaction (L_{RI}) and the linear inaction-penalty (L_{IP}), which operate in the same way as the L_{RP} except that they completely ignore the penalties and rewards, respectively.

Reinforcement schemes can exhibit either ergodic or absorbing behaviour. In ergodic schemes, the goal of the automaton is to converge to the static state probability vector for the system. In absorbing schemes, the goal is to converge upon one of the R unit vectors, thereby choosing only one action eventually. In either case, the pace by which the automaton approaches its goal becomes increasingly smaller. To overcome this problem, the concept of discretizing the probability space was introduced. Oommen [26] proposed discretized versions of L_{RI} and L_{IP} which he called DL_{RI} and DL_{IP} . He showed the ϵ -optimality of the former and its superiority to its linear counterpart, as well as the expedience of the latter. He also proposed the notion of artificially absorbing barriers for automata that are originally ergodic (such as DL_{IP}), which made the automaton ϵ -optimal.

Solving the OPP using Modified L_{RP}

Oommen and Ma [31] proposed a modified linear reward-penalty scheme (ML_{RP}) for solving the OPP. The ML_{RP} is a model-based clustering scheme that provides a

probability distribution for the partitioning of the different objects, according to the linear reward-penalty reinforcement scheme. The automaton defines W actions for the W objects to be partitioned. It also maintains a set of abstract objects $\{O_1, \dots, O_W\}$ which can “choose” their action. The automaton defines a set of probability vectors $\{\underline{P}_1, \dots, \underline{P}_W\}$ rather than only one such vector. Each vector $\underline{P}_i = [p_{i1}, \dots, p_{iW}]^T$ is associated with an abstract object, O_i . The element p_{ij} represents the probability that action α_j is chosen by O_i .

When a pair of objects (A_i, A_j) are accessed, if O_i and O_j choose the same action α_k , they are rewarded by increasing the probability of them choosing the same action, and by simultaneously decreasing the probabilities of them choosing all other actions. If, however, the objects choose different actions α_p and α_q , they are encouraged to “get closer” by increasing the probability of O_i choosing α_q and O_j choosing α_p . This is done (in the case of O_i) for example by decreasing p_{ip} and increasing p_{iq} . Notice that the probabilities of choosing other actions have not been changed.

The following is an algorithmic version of the ML_{RP} . α_p and α_q are the actions chosen by O_i and O_j , respectively.

Algorithm ML_{RP}

Initialization

Initialize p_{ii} to 1.0 and p_{ij} to 0.0

Repeat forever

For a sequence of T queries (A_i, A_j) ,

If $(\alpha_p = m)$ and $(\alpha_q = m)$ /* Reward: Both objects chose same class */

$p_{im} = 1 - \lambda_1(1 - p_{im})$ /* increase chance to choose α_p */

$p_{jm} = 1 - \lambda_1(1 - p_{jm})$ /* increase chance to choose α_q */

For $(k = 1 \text{ to } W, k \neq m)$

$p_{ik} = \lambda_1 p_{ik}$ /* decrease chance to choose */

$p_{jk} = \lambda_1 p_{jk}$ /* other actions */

Else If $(\alpha_p = m)$ and $(\alpha_q = n)$ /* Penalty: Different actions chosen */

$p_{in} = p_{in} + (1 - \lambda_2)p_{im}$ /* Encourage O_i to choose α_q */

$p_{im} = \lambda_2 p_{im}$ /* and not α_p */

$p_{jm} = p_{jm} + (1 - \lambda_2)p_{jn}$ /* Encourage O_j to choose α_p */

$p_{jn} = \lambda_2 p_{jn}$ /* and not α_q */

Present partitions based on the the probabilities p_{ij} 's

End Algorithm ML_{RP}

The ML_{RP} also shows superior performance to that of the BAM. For 18 objects to be partitioned into 2 groups, the ML_{RP} took 233 steps to converge, as opposed to 3352 for the BAM. In terms of accuracy, the ML_{RP} was found to be only marginally less accurate than the BAM. Oommen and Ma showed that ML_{RP} is expedient and conjectured its ϵ -optimality.

Implementation Details

The minimum space complexity for the ML_{RP} is $O(W^2)$, to construct the probability vector set $\{\underline{P}_1, \dots, \underline{P}_W\}$. In fact, to cut down on processing time, one needs another array ($O(W)$) to store the current action of a given abstract object O_i . Also, upon reward, the time complexity is $O(W)$ as the fields of the probability vectors get updated. This, however, can be considerably reduced, by not updating a vector that is within a certain limit of a unit vector, and thus considering it having converged at this state. Thus, after convergence, the time complexity of the ML_{RP} could be considered constant.

Oommen and Ma [31] suggest the values of λ_1 and λ_2 to be 0.5 and 0.975, respectively. Our experiments, however, suggest that $\lambda_2 = 0.9$ gives much quicker convergence without sacrificing accuracy. Changing λ_1 to be within the range of 0.5 to 0.9 did not seem to significantly affect the performance or the accuracy of the automaton.

4.3 Conclusion

In this Chapter, we defined the clustering problem and provided an overview of the different clustering techniques as given by [13]. We then defined the Object Partitioning Problem (OPP), which is a clustering problem concerned with classifying objects based on their co-joined access probabilities. Solutions to this problem, therefore, are directly applicable to be used to capture the dependence relation among access queries for a self-organizing data structure scheme.

We presented a classical object partitioning scheme (the BAM) that is based on moving co-jointly accessed objects closer to their centroid. We proceeded by defining one family of learning automata, the Fixed Structure Stochastic Automata (FSSA), and introducing two FSSA-based schemes for the OPP. The OMA seems to be the best solution, both in terms of accuracy and convergence time, but is unfortunately only suited to the equal partitioning problem, i.e., when the partitioning classes are desired to be of equal size. The SMA does not have the same limitation, but yields a performance that is not as good as the OMA. We also defined another learning automata family, the Variable Structure Stochastic Automata (VSSA), and introduced a VSSA-based algorithm, the modified linear reward-penalty scheme (ML_{RP}), which solves the general OPP with acceptable accuracy, but has a greater time and space complexity than the OMA.

Chapter 5

Hierarchical List Organization Strategies

In Chapter 3, we suggested a novel way to manage self-organizing lists, by viewing a list as a collection of sublists and manipulating elements within these sublists as well as the sublists themselves in their entirety. We discussed the rationale behind such an approach and concluded that for it to work, the lists should mirror the dependence relationships between the access queries. To motivate this, in Chapter 4, we defined solutions for the Object Partitioning Problem as a means to capture such dependence.

We are now in a position to apply the dependence capturing algorithms to the suggested sublist manipulation concept, in the hope that we produce a better self-organizing scheme in environments characterized by dependent access sequences.

5.1 Query Generators

Throughout our experiments, we used a data source with $n = 128$ distinct queries, labelled 1 through 128. To produce dependent access sequences, we divided the ordered query space into k subsets $(Q_1 \dots Q_k)$, each of size m . We then applied the

Markovian and periodic models of dependence discussed in Section 3.2. For example, if $k = 8$ and $m = 16$, Q_0 would include the elements $1 \dots 16$, Q_1 would include $17 \dots 32$, etc.

Within the query space subsets, the access probabilities followed a well-defined probability distribution. This means that query labelled (1) always has the same probability as query (17) (if m was 16), and query labelled (i) always has the same probability as query labelled $((i + m) \bmod n)$.

The following probability distributions were used to describe the access probabilities $\{s_i\}$, of the queries in the first query space subset, where $1 \leq i \leq m$. The probabilities for other query subsets follow according to the rules discussed above.

1. *Zipf's distribution*: The access probability is given by:

$$s_i = \frac{1}{iH_m} \quad \text{for } 1 \leq i \leq m,$$

where H_m is the m^{th} harmonic number, and is given by $\sum_{k=1}^m (1/k)$. Zipf's distribution is the most commonly used rule for modeling real-life access probabilities, as we have seen in [6, 7, 8, 34]. The most common use for Zipf's law is for modeling the frequency of English words.

2. *80-20 distribution*: The access probability is given by:

$$s_i = \frac{1}{i^{(1-\theta)} H_m^{(1-\theta)}} \quad \text{for } 1 \leq i \leq m \text{ and } \theta = \frac{\log 0.80}{\log 0.20} \approx 0.1386,$$

where $H_m^{(1-\theta)}$ is the m^{th} harmonic number of order $(1 - \theta)$, and is given by $\sum_{k=1}^m (1/k^{(1-\theta)})$.

3. *Lotka's distribution*: The access probability is given by:

$$s_i = \frac{1}{i^2 H_m^2} \quad \text{for } 1 \leq i \leq m,$$

where H_m^2 is the m^{th} harmonic number of order 2, and is given by $\sum_{k=1}^m (1/k^2)$.

4. *Exponential distribution*: The access probability is given by:

$$s_i = \frac{1}{2^i K} \quad \text{for } 1 \leq i \leq m,$$

where $K = \sum_{k=1}^m (1/2^k)$.

5. *Linear distribution*: The access probability is given by:

$$s_i = K(m - i + 1) \quad \text{for } 1 \leq i \leq m,$$

where $K = \sum_{k=1}^m k$.

Some of the distributions (in particular, the exponential, Lotka, and to a lesser degree, the Zipf distributions) produce logarithmic-type (L-shaped) graphs when plotted on linear axes, yielding very high probabilities (and thereby giving a highly preferential treatment) for a very small subset of the query space. This makes our comparisons fair against the MTF algorithm, the *de facto* standard for self-organizing schemes in non-stationary environments, which performs very well under such conditions. Indeed, beating the MTF in such distributions proved to be a rather hard job.

5.2 OMA-Based Algorithms

In this section, we use the OMA as our dependence capturing engine, and combine it with different hierarchical sublist organization strategies, in the hope of getting

better results in both the Markovian and periodic switching environments. For all our experiments, the automaton's depth was set to 20.

5.2.1 MTF-MTF-OMA and TR-MTF-OMA

The MTF-MTF algorithm (Section 3.4.1) is modified to include an OMA (Section 4.2.3) object, where the subclasses of the OMA reflect the sublist division of the MTF-MTF *at all times*. Whenever a query pair is requested, the MTF-MTF locates the requested records in the list and the sublists they reside in. The query pair is run against the OMA to determine if and how they should be relocated from their current subclasses. The relocations are then reflected in the list. Finally, the elements are moved to the front of their sublist, and if they reside in the same sublist, this sublist is moved to the front of the list.

We now present a high-level algorithmic version of the MTF-MTF-OMA. Notice that we couple incoming queries into pairs as discussed in Section 4.2.3.

Algorithm MTF-MTF-OMA

Initialization

Consider the list of n elements to be composed of k sublists ($L_1 \dots L_k$) of m elements each. Initialize the OMA such that its subclasses reflect the list's decomposition into sublists.

For every query pair (A_i, A_j) ,

Locate A_i and A_j in the list, and determine the boundaries of their respective sublists, L_u and L_v .

Run A_i and A_j against the OMA, and examine the output.

If OMA output = 0

/* L_u is the same as L_v */

Move A_i to the front of L_u

Move A_j to the front of L_u

Move L_u to the front of list

Else If OMA output = 1

/* A_i has been moved to A_j 's subclass, and */

/* an element has been moved back to A_i 's subclass */

Swap A_i with the element that was moved from L_v

Move A_i to the front of L_v

Move A_j to the front of L_v

Move L_v to the front of the list

Else If OMA output = 2

/* A_j has been moved to A_i 's subclass, and */

/* an element has been moved back to A_j 's subclass */

Swap A_j with the element that was moved from L_u

Move A_j to the front of L_u

Move A_i to the front of L_u

Move L_u to the front of the list

Else If OMA output = 3

/* A_i and A_j are in different subclasses */

Move A_i to the front of L_u

Move A_j to the front of L_v

End Algorithm MTF-MTF-OMA

Notice that we only move the sublist to the front of the list if it contains both elements, at the end of the loop. This is because we are only interested in moving the sublist to the front of the list when the locality of reference phenomenon manifests itself, which translates to both elements being in the same subclass (and consequently, the same sublist) according to the OMA partitioning.

The TR-MTF-OMA algorithm is similar to the MTF-MTF-OMA, except that records within the sublist are *transposed* with their preceding neighbours, instead of

being moved to the front. This is to take advantage of the more conservative approach of the transposition algorithms, which tend to give better performance in the long run. Notice, however, that in both algorithms, we always move the sublist of the accessed records to the front of the list, rather than transpose it with other sublists. This makes sense in our environment, where there is a high probability a record from the same sublist will be accessed in the short term because of the locality of reference phenomenon.

Results in Markovian Environments

Tables A.1 and A.2 tabulate¹ the asymptotic and amortized cost for the OMA-based list organizing algorithms. The experiments involved lists with 128 records, and were done on varying number of sublists ($k = 2, 4, 8, 10, 16, 32, 64$), in both a Markovian environment ($\alpha = 0.9$) and a periodic one (period = 10).

In the Markovian environment, even when there are only two sublists ($k = 2$), the MTF-MTF-OMA yielded better asymptotic and amortized cost performance (29.31, 29.52) than the MTF (34.42, 34.97) in the Zipf distribution. For the 80-20 and linear distributions, the MTF-MTF-OMA also produced better costs than the MTF, (33.00, 32.72) versus (41.11, 41.11), and (39.11, 38.42) versus (52.13, 51.58), respectively. However, for the Lotka and exponential distributions (which produce the L-shaped curves, discussed above), the performance is not as good, (10.38, 10.29) versus (6.74, 6.97) and (10.95, 11.00) versus (3.77, 3.81), respectively. However, with increasing number of sublists (and decreasing number of elements within the sublist), the performance of the algorithm improves. With $k = 16$ and $k = 32$, the algorithm achieves better performance than the MTF in *all* distributions.

Figure 5.1 plots the ratio of the asymptotic cost of the MTF-MTF-OMA algorithm

¹The results obtained are catalogued in tables which are presented in Appendix A. We have chosen to present them in the Appendix in the interest of readability and continuity.

Figure 5.1: The ratio of the asymptotic cost of the MTF-MTF-OMA to that of the MTF, for different numbers of sublists k and different probability distributions, in a Markovian environment with $\alpha = 0.9$.

to that of the MTF, for different values of k . A value greater than unity indicates a performance worse than MTF, while a value less than unity indicates a superior performance. Observe that for the Zipf, 80-20, and linear distributions, the MTF-MTF-OMA produced results that were consistently better than the MTF, whereas for the Lotka and exponential distributions a small number of sublists yielded a poor performance. This is because with smaller list sizes, the number of elements with very low probability is smaller, and therefore the likelihood of a non-optimal partitioning by the OMA appears to be also lower. Also observe that the curves are U-shaped, implying that the optimum number of sublists (and records within a sublist) is in the center, when k and m are closer to each other in value. The amortized cost results were almost identical to the asymptotic costs.

Tables A.1 and A.2 also show that the TR-MTF-OMA consistently produced

better results than the MTF-MTF-OMA, albeit with a much smaller margin than the differences between TR and MTF.

We have also studied the performance of the MTF-MTF-OMA with respect to varying degrees of locality of reference (α). In this experiment, the number of sublists k was fixed at 16. Recall, from the experiments by Bachrach *et al.* (Figure 2.4), that the algorithms which did well at higher degrees of dependence were worse with weaker dependence (and vice versa), and that the crossover from “bad” to “good” performance occurred when α was greater than 0.7 to 0.8. The algorithms studied in this Chapter follow the same pattern. For example, at the lowest dependence degree ($\alpha = 0.1$), the MTF-MTF-OMA had an asymptotic cost of 60.17, compared to 51.16 for the MTF. At the highest dependence degree, however, the score were 16.06 as opposed to 26.79, respectively. Tables A.5 and A.6 show the results obtained for different probability distributions, and Figure 5.2 plots the results for the Zipf distribution.

Observe that the curve for the MTF-MTF-OMA (as well as the TR-MTF-OMA) dip well below that of the MTF, which was thought to be the best performing algorithm for high degrees of locality of reference. This, we believe, is a significant result, since, to our knowledge, it indicates that the algorithms perform better than any other algorithm reported in the literature, within the same space and time requirements. Also, notice that the TR-MTF-OMA produced better results than the MTF-MTF-OMA, for almost all values of α , but significantly so with $\alpha = 0.7$, at which point the TR-MTF-OMA outperformed the TR while the MTF did not.

To get a feeling for the rate of convergence of the algorithms, we plotted the amortized cost as the number of queries increased, for the first 100,000 queries, in a Markovian environment with $\alpha = 0.9$, $k = 16$, for the Zipf distribution (Figure 5.3). The reader should recall that Narendra and Thathatchar [24] define the optimality of a system in a non-stationary environment by its ability to minimize the amortized

Figure 5.2: The asymptotic cost of various algorithms, in a Markovian environment with different degrees of dependence α , for $k = 16$ sublists and the Zipf probability distribution.

cost.

From the beginning, the MTF-MTF-OMA curve is way below the TR curve, and quite close to the MTF, suggesting that even early on in the experiment, the cost incurred by the MTF-MTF-OMA is not much larger than that of the MTF, and is therefore quite tolerable. The curve then intersects the MTF curve at around 10,000 queries, and continues improving in performance, though with an ever decreasing slope. We observed that the learning trend of the MTF-MTF-OMA does not seem to stop even at 5,000,000 queries. On the other hand, the MTF curve is steady almost from the beginning. The curve for the TR-MTF-OMA follows that of the MTF-MTF-OMA almost exactly, with a slightly slower convergence rate for the first 20,000 queries.

Figure 5.3: The amortized cost of various algorithms, for 100,000 queries, in a Markovian environment with $\alpha = 0.9$, for $k = 16$ sublists and the Zipf probability distribution.

Results in Periodic Environments

We tested the MTF-MTF-OMA in a periodic environment with a period of 10 accesses. This produces a comparable environment to the Markovian one (with $\alpha = 0.9$), since the average number of accesses from the same subclass in the Markovian environment is $1/(1 - \alpha) = 10$. In a periodic environment, the MTF-MTF-OMA delivers performance that is just as good as in a Markovian one with strong locality of reference (See Tables A.3 and A.4). This is because periodic environments can be thought of as Markovian ones with very strong locality of reference ($\alpha = 1$) for a specified number of queries. When compared to MTF (Figure 5.4), the curves produced are of a similar nature to the ones produced in a Markovian environment, and all the arguments can be carried over. However, since we know that the environment is in

Figure 5.4: The ratio of the asymptotic cost of the MTF-MTF-OMA to that of the MTF, for different number of sublists and different probability distributions, in a periodic environment with period = 10.

fact periodic, rather than Markovian, we can make use of this information in devising a better strategy, as we shall discuss later in this Chapter. Again, Tables A.3 and A.4 show that the TR-MTF-OMA produces consistently better results than the MTF-MTF-OMA, with few exceptions.

Implementation Details

We now present some implementation details that are crucial to improve the overall running time for the MTF-MTF-OMA and TR-MTF-OMA algorithms.

- Recall that a wrapper layer provides the queries in pairs, such that if the access sequence $\langle \dots A_i A_j A_k \dots \rangle$ is requested, it will be presented to the algorithm as $\langle \dots (A_i, A_j)(A_j, A_k) \dots \rangle$. Therefore, if the algorithm retains a reference to A_j , it does not need to search for it again as it is guaranteed to be the first

element of the query pair of the next query pair. Also notice that in many cases (for OMA output 0 and 3), only A_j needs to be moved to the beginning of the sublist, as A_i has already been moved at time $t - 1$.

- Also recall that the running time of the OMA is usually constant, but it can be $O(n)$ in order to locate an item to be migrated outside a group. To illustrate, suppose that A_j needs to be moved to A_i 's group. This means that an element, say A_k , needs to be moved *out* from A_i 's group back to A_j 's group. In the original OMA implementation, this involved $O(n)$ steps to locate such an element, say A_k . However, since the sublists always reflect the OMA classification, we can search for A_k in A_j 's sublist, to which we have reference already. This reduces the worst running time for the OMA from $O(n)$ to $O(m)$.
- Finally, notice that we can determine the class which an element A_i belongs to in constant time, by applying Equation (4.2). Observe that, again, since the sublists always reflect the OMA classification, we can also obtain the sublist of A_i in constant time. If the implementation retains pointers to sublist boundaries, instead of determining such boundaries “on the fly” as discussed in Section 3.4.1, the algorithm can start searching for the requested element from the beginning of the corresponding *sublist*, rather than from the beginning of the list. This means that we can put to better use the extra information that comes with the random access storage in the OMA implementation, to reduce the search time for the accessed element to $O(m)$.

All in all, with few implementation modifications, the running time of the MTF-MTF-OMA can actually be reduced to $O(m)$ instead of $O(n)$ as a normal sequential search would require. This means that the MTF-MTF-OMA can be more appealing than a lot of other algorithms, without *even* considering the gains obtained in the asymptotic and amortized search costs! This, of course, is only valid for environments

following the Markovian or periodic query generation models.

5.2.2 Periodic Variations of the MTF-MTF-OMA and the TR-MTF-OMA

Given that we know that the environment is periodic, we felt that the MTF-MTF-OMA and TR-MTF-OMA can be made to perform even better in such environments. If we have *a priori* knowledge of the period T , the trick is that whenever the period expires (i.e., after T queries), we simply move the first sublist to the *end* of the list. If the OMA has correctly converged, no records from this sublist are expected to be accessed except after $(k - 1)T$ queries. We call these algorithms the MTF-MTF-OMA-Periodic and the TR-MTF-OMA-Periodic, respectively.

If we do not know the period *a priori*, however, we choose to move the first sublist to the rear if two successive queries are not dependent. That is, if the OMA does not find two successive queries in the same group, we interpret that as the end of the period and move the sublist to the rear accordingly. We call such algorithms the MTF-MTF-OMA-Unknown_Period and TR-MTF-OMA-Unknown_Period, respectively.

Results in a Periodic Environment

Tables A.3 and A.4 show that the MTF-MTF-OMA-Periodic and the TR-MTF-OMA-Periodic are *ideal* for the periodic environment. For smaller numbers of sublists ($k = 2, 4$), the MTF-MTF-OMA-Periodic “defects” the asymptotic and amortized cost of the MTF-MTF-OMA, already shown to be better than the MTF, by more than a factor of two. For larger numbers of sublists ($k = 32$, Zipf distribution), the difference is dramatic: 40.55 for the asymptotic cost of the MTF, 27.05 for the MTF-MTF-OMA, and 2.26 for the MTF-MTF-OMA-Periodic! (Figure 5.5).

The reason for such a good performance is that for very small sublist sizes ($k =$

Figure 5.5: The asymptotic cost of different algorithms plotted against the number of sublists for the Zipf distribution, in a periodic environment with period = 10.

32, 64), the probability that the OMA approaches an optimal partitioning becomes very high. This is because all the queries within the query generator's sets have substantial probability of occurrence, and therefore the OMA has less chance of being locked up in an erroneous partitioning. For example, when there are only two elements per groupset ($k = 64$), the probabilities of the first and second elements are 0.66 and 0.33, respectively, and there are no elements with negligible probability. With the OMA performing very well, the search cost can be made as low as 1.43. This is very close to the expected cost of a static optimal *offline* algorithm, for a list of two records, which would be 1.33^2 .

Figure 5.5 also shows that the performance of the MTF-MTF-OMA-Unknown_Period algorithm is often comparable to that of the MTF-MTF-OMA-Periodic, and always better than the MTF-MTF-OMA. This behaviour is consistent across all probability

²The expected search cost of a static optimal offline algorithm would be $0.66 \times 1 + 0.33 \times 2 = 1.33$.

distributions and list sizes in Tables A.3 and A.4. The reason that the MTF-MTF-OMA-Unknown_Period does not outperform the MTF-MTF-OMA-Periodic, is that, if the OMA wrongly indicates that two records do not belong in the same group, the group will be moved to the rear of the list before the period expires. This means that a penalty of at least $n - m$ probes will be added to the next requested record.

For the same reason, the amortized cost of the periodic variants of the algorithms is sometimes higher than the asymptotic one. This is because sending the first sublist to the rear before the OMA converges (as in the case of the MTF-MTF-OMA-Periodic) or before the period expires (as in the case of the MTF-MTF-OMA-Unknown_Period) can cause penalties that weigh into the amortized cost. In all cases, however, both costs remained significantly lower than those of the other algorithms.

Results in Markovian Environments

As expected, both MTF-MTF-OMA-Periodic and TR-MTF-OMA-Periodic do not perform as well as MTF/TR-MTF-OMA. Needless penalties are incurred since the algorithms only consider the period T , rather than the query dependence, for moving the first sublist to the rear.

However, the performance of the MTF/TR-MTF-OMA-Unknown_Period was often on a par to that of the MTF/TR-MTF-OMA. This is because when the local set changes in a Markovian environment, all the other local sets are equally likely to become the current local set, and therefore the position where the first sublist is moved to is not very relevant. In fact, the MTF/TR-MTF-OMA-Unknown_Period seemed to perform slightly better than the MTF/TR-MTF-OMA in all but the Lotka and Exponential distributions, and for lower values of k (See Tables A.1 and A.2). Indeed, among the OMA-based algorithms, the MTF/TR-MTF-OMA-Unknown_Period algorithms seem to be the best choice to suit all kinds of environments characterized by dependent access queries.

5.3 ML_{RP} -Based Algorithms

We have seen that the OMA-based family of list organization strategies produced results that are better than any other reported algorithm in the literature, in environments with strong locality of reference. However, they still suffer the same setbacks as the OMA, in that they function best when the sizes of the environment's local contexts (and consequently, the sublists) are equal. But the hierarchical list organization strategy and the dependence capturing mechanism need not be tightly coupled. Indeed, as long as the interface is well-defined, any dependence capturing mechanism can be used with the algorithms described above, instead of the OMA.

As a proof of concept, we implemented the MTF-MTF- ML_{RP} , where the ML_{RP} was used for the partitioning of the the environment inputs and determining the sublists. The ML_{RP} (Section 4.2.5) can partition objects into different sized groups, thereby offering more flexibility than the OMA, with comparable accuracy. However, we have seen that the time and space requirements are higher than that of the OMA.

The MTF-MTF- ML_{RP} algorithm is very similar to the MTF-MTF-OMA, except that in initialization, every record in the list sits in a sublist of its own. The sublists are then gradually merged according to the access sequences to form bigger sublists, that eventually match the dependence model of the environment. Notice that, when an element is moved from one sublist to another, no other element is moved back to the original sublist.

Tables A.1 to A.4 show the asymptotic and amortized costs for the MTF-MTF- ML_{RP} , under the same conditions in which OMA-based algorithms were tested. Notice that the costs for the MTF-MTF- ML_{RP} and the MTF-MTF-OMA are very close to each other, and quite often within each other's variance. The one striking difference is the exceptionally better performance of the MTF-MTF- ML_{RP} under the Lotka and exponential distributions for lower values of k , which almost matches that of the MTF

(Recall that the MTF performed much better than the other schemes for lower values of k). Although we cannot explain the reason for this, it seems that the ML_{RP} is especially more suited for classification under such conditions than the OMA. The fact that both algorithms perform similarly suggests that their convergence accuracy is comparable, and therefore, we expect that all other algorithms can be adopted to use the ML_{RP} and yield the same results.

The most interesting result about the MTF-MTF- ML_{RP} is the rate of change in its amortized cost, illustrated in Figure 5.3. Although the cost starts higher than its OMA-based counterpart, it catches up at 10,000 queries, and proceeds with an impressive convergence speed to quickly reach the final value well before the MTF-MTF-OMA. Although the literature [30, 31] did not provide comparisons between the ML_{RP} and the OMA, we believe that the former yields much faster convergence speed, which in turn could be well worth the additional time and space penalties incurred.

5.4 Conclusion

In this chapter, we have introduced hierarchical list organization strategies that try to arrange sublists within a list such that dependent records lie closer to each other, by employing object partitioning mechanisms. We introduced the MTF-MTF-OMA and the TR-MTF-OMA, which are two algorithms that produce better results than any other reported algorithm in the literature, in environments with heavily dependent accesses. We also introduced several variations of the algorithms, among which is the MTF-MTF-OMA-Unknown_Period, which not only performs extremely well in periodic environments without *a priori* knowledge of the period, but which also performs almost as well as the MTF-MTF-OMA in Markovian environments too.

We also showed that the hierarchical list organization strategies can be coupled

with any other partitioning algorithm, such as the ML_{RP} . The MTF-MTF- ML_{RP} produced results that are as good, and sometimes better than the MTF-MTF-OMA, while delivering much faster convergence rates. However, the MTF-MTF- ML_{RP} comes also with the inherent disadvantages of the ML_{RP} , namely, the higher space and time requirements.

Combining partitioning algorithms with self-organizing lists produced very exciting and promising results. We believe they will also lead to beneficial results for on other types of self-organizing data structures.

Chapter 6

Hierarchical List Organization Strategies for Doubly Linked Lists

The results obtained in Chapter 5 are not just conceptual. In this Chapter we shall show that they can also be used for other data structures. To demonstrate this we extended our hierarchical list organization strategies to another linear search data structure, the Doubly Linked List (DLL).

Not nearly as much research is available on self-organizing DLLs as their singly linked cousins, and certainly not in non-stationary environments. In this Chapter, we first define the self-organizing doubly linked list data structure, and discuss the available literature on that problem. We then introduce the new dependence models that apply to this data structure, and discuss the results of applying our algorithms to them.

6.1 Doubly Linked Lists

Unlike singly linked lists, a node in a doubly linked list has two pointers: one to the predecessor node and another to a successor node [20]. This means that a cursor

pointer traversing the list can move in both the “left” and “right” directions. In the context of linear search, this also means that an access query can come from either the left or right end of the list, which we shall assume to be well-defined nodes. We shall also assume that the search begins from the direction that the query came from.

To illustrate this, consider a doubly linked list that is being accessed by two requestors from either ends, with potentially different access probabilities for the list nodes. The self-organizing DLL problem would be to maintain the list arrangement so as to optimize the search cost for both requestors simultaneously [40].

A record R_i is accessed from the left end and the right end of the list with the probabilities s_{iL} and s_{iR} , respectively. The Theory of Probability tells us that the sum of probabilities of all events in the access probability set is certainty. In order to satisfy this axiom, the environment has to fulfill the constraint:

$$\sum_{i=1}^n s_{iL} + \sum_{i=1}^n s_{iR} = 1,$$

where n is the number of records in the list.

6.1.1 Expected Search Cost

To calculate the expected cost for searching the doubly linked list, we modify Equation (2.1) to include the cost of access from the right. Recall that the list is in arbitrary order π where $\pi(i)$ is the cost of accessing record R_i from the left end, which makes the cost for accessing record R_i from the right to be $n - \pi(i) + 1$. Therefore,

$$\begin{aligned} E[Cost] &= \sum_{i=1}^n s_{iL} \pi(i) + \sum_{i=1}^n s_{iR} (n - \pi(i) + 1) \\ &= \sum_{i=1}^n (s_{iL} - s_{iR}) \pi(i) + \sum_{i=1}^n s_{iR} (n + 1). \end{aligned} \quad (6.1)$$

We notice that the second term in Equation (6.1) is independent of the ordering, π , and therefore the list ordering is optimized when arranged in the decreasing order of $(s_{iL} - s_{iR})$, as has been noted in [22, 25, 40]. This confirms the informal intuition that records that are most frequently accessed from the left end should be placed closer to that end and vice versa.

6.1.2 Research on Self-Organizing DLLs

The earliest work, to our knowledge, in the field of self-organizing DLLs is due to Matthews *et al.* [22]. In their paper, they described the move-to-end (MTE) heuristic, which is an extension to the MTF heuristic in singly linked list, whereby a queried element is moved to the respective end from which it was accessed. For example, given a list in the order $\langle R_1 R_2 \dots R_i \dots R_n \rangle$ and element R_i is queried from the left end, the list would be reorganized to $\langle R_i R_1 R_2 \dots R_n \rangle$. Had R_i been accessed from the right, the heuristic would have organized the list to be $\langle R_1 R_2 \dots R_n R_i \rangle$.

Matthews *et al.* [22] showed the asymptotic search cost for the MTE heuristic by first showing the asymptotic probability of record R_j to be to the left of record R_i :

$$P\{R_j \text{ left of } R_i\} = \frac{s_{jL} + s_{iR}}{s_{iL} + s_{iR} + s_{jL} + s_{jR}},$$

and hence obtaining the average cost for the MTE (A_{MTE}) as:

$$A_{MTE} = 1 + 2 \sum_{i=1}^n \left\{ s_{iL} \sum_{j < i} \frac{s_{iR} + s_{jL}}{s_{iL} + s_{iR} + s_{jL} + s_{jR}} + s_{iR} \sum_{j > i} \frac{s_{jR} + s_{iL}}{s_{iL} + s_{iR} + s_{jL} + s_{jR}} \right\}.$$

Matthews *et al.* also proved two Lemmas that are important for our discussion of the Environment later in this Chapter:

- If every element in the DLL is equally likely to be accessed from either end (i.e., $s_{iL} = s_{iR} \forall i$), then the average search cost cannot be made lower than $(n + 1)/2$. In other words, there is no need to run a heuristic to maintain the list in optimal order. Algorithms for maintaining a DLL are, therefore, only significant when the access probabilities from either ends are not equal.
- If the search is more likely to be accessed from one end of a DLL L_D than the other (e.g., $s_{iL} > s_{iR} \forall i$), and if A is an optimal heuristic for the singly linked list L_A corresponding to L_D , then A is also optimal for L_D . In other words, it suffices to regard the DLL as a singly linked one and maintain it as such. Such an environment is said to be “directionally biased”, as we shall explain later.

While the MTE heuristic is a generalization of the MTF, Ng and Oommen [25] introduced the SWAP heuristic as a generalization of the TR. Under the SWAP heuristic, an accessed record is transposed with the neighbouring record from the direction of the access query. That is, if record R_i is accessed in a list configuration $\langle R_1 \dots R_{i-1} R_i R_{i+1} \dots R_n \rangle$ from the left end, the configuration becomes $\langle R_1 \dots R_i R_{i-1} R_{i+1} \dots R_n \rangle$. If the query comes from the right end, the configuration becomes $\langle R_1 \dots R_{i-1} R_{i+1} R_i \dots R_n \rangle$. Ng and Oommen experimentally showed that the SWAP heuristic is more efficient than the MTE heuristic, extending Rivest’s work [34] in comparing the TR to the MTF.

Such an application of algorithms, i.e., with respect to the direction of the access request, is called a *directed mapping*. In contrast, an *undirected mapping* is when the list reorganization is done irrespective of the direction of the access request. Using these definitions, Ng and Oommen [25] generalized the second lemma of Matthews *et al.* that, if heuristic τ is more efficient than heuristic ρ in singly linked lists, then in a directionally biased environment, τ is also more efficient than ρ , in their undirected mapping form. Ng and Oommen also conjectured that the directed mapping of τ is

more efficient than that of ρ , regardless of whether the Environment is directionally biased or not.

Valiveti and Oommen [40] later proved that the SWAP is better than the MTE for *exclusive* distributions. A distribution is said to be *exclusive* if the following condition holds whenever $i < j$:

$$s_{kL}(s_{iL} - s_{jL}) + s_{kR}(s_{jR} - s_{iR}) + (s_{iL}s_{jR} - s_{jL}s_{iR}) > 0 \quad \forall k \neq i, j.$$

Valiveti and Oommen identified two families of exclusive distributions:

- *Dually monotonic distributions* are distributions in which $s_{iL} > s_{jL}$ and $s_{jR} > s_{iR}$ for all $i < j$. In other words, if $i < j$, R_i is more likely to be accessed from the left end and R_j is more likely to be accessed from the right end.
- *Directionally biased distributions*, introduced above, are those in which the conditional probability of accessing a record from a given direction is the same for all records (i.e., independent of the record index), and is greater than 0.5.

Valiveti and Oommen also conjectured that the SWAP heuristic is more efficient than the MTE, for all access distributions.

6.2 Models of Dependence

We shall now consider how the models of dependence introduced in Chapter 3 can be expanded to DLLs. There are two ways to model dependent access queries in doubly linked lists. In the first, the query dependence is itself dependent on the query direction. For example, for queries at the left end, records R_i and R_j may be dependent (i.e., belonging to the same local context Q_u , see Section 3.2.1), but not R_k . However, for queries at the right end, records R_j and R_k may be dependent (belonging to a different local context, $Q_v, v \neq u$), but not R_i .

Considering the work already done, we do not have a tool to capture dependence that takes into account the two query directions for DLLs. The only way to apply our algorithms in such environments, therefore, would be to duplicate the entire list and maintain two sets of dependence capture tools. This would transform the problem into that of a two singly-linked lists, and we would get identical results to the ones in Chapter 5, albeit at the added memory cost. We will not be discussing this model further in this Chapter.

In the second model, the query dependence is independent of the query direction. This means that the probability the next query will be from Q_u given the current query is from Q_u is α , irrespective of the query direction for either the current or the next query.

A simple way to implement this second model is to associate a parameter, $0 \leq \beta_i \leq 1, 1 \leq i \leq k$, with every context Q_i in the Environment, to indicate the total probability of accessing it within that working group to be from the right end of the list. If $\beta_i = 0$, then all access queries for the given context come from the left end, and vice versa. If β_i is the same for all i , we simply call it β . In this case, although different records may have different left and right access probabilities, the distribution becomes directionally biased, as the access probability from one direction is always greater than from the other.

We can also set up the Environment to be dually monotonic *on the level of the local context*, rather than the individual record level as defined by Valiveti and Oommen [40], by using an equation such as this:

$$\beta_i = i/(k + 1). \quad (6.2)$$

For example, an environment with two local contexts would be assigned $\beta_1 = 0.33$ and $\beta_2 = 0.66$. Therefore, the total probability for Q_1 to be accessed from the left

is more than Q_2 , and the opposite is true for queries from the right end, therefore fulfilling the conditions for dually monotonic environment, on the level of the local context.

6.3 Hierarchical List Organization Algorithms in DLLs

We have already seen that a superior algorithm for a singly linked list is generally believed to deliver better performance as well in doubly linked ones, be it in directed or undirected mapping form. Since we have devised algorithms that are superior for singly linked list environments (see Chapter 5), we can speculate that they can provide a similar performance in doubly linked ones.

We use the notation described earlier, that algorithm X-Y-Z signifies that operation X was done within the sublist, then operation Y was done among the sublists, using the dependence capturing algorithm Z. We tested our algorithms in the undirected mapping form in a directionally biased environment, against the well studied undirected mapping algorithms MTF and TR. The four permutations were: MTF-MTF-OMA, MTF-TR-OMA, TR-MTF-OMA and TR-TR-OMA.

We also implemented a directionally mapped version of our hierarchical list organization algorithms to compare them to the MTE and SWAP. Again, we tested all four permutations: MTE-MTE-OMA, MTE-SWAP-OMA, SWAP-MTE-OMA, and SWAP-SWAP-OMA. The results are given in Appendix B.

6.3.1 Directionally Biased Environment

Table B.1 tabulates the results obtained for undirected mapping algorithms tested in a directionally biased environment for varying values of β ranging from 0 to 0.5, with

the number of sublists $k = 16$, and the dependence factor α set to 0.9. Other than their new property β , the query generators followed the same characteristics as those described in Chapter 5.

At $\beta = 0.5$, all the records have an equal probability for access from either end of the list (i.e., $s_{iL} = s_{iR} \forall i$). Therefore, Matthews *et al.* [22] Lemma applies (see Section 6.1.2) and the search cost is around 64.5 for all algorithms and distributions. At $\beta = 0$, the Environment is identical to that tested in Chapter 5, and we get identical results. Both asymptotic and amortized costs for the MTF-MTF-OMA and TR-MTF-OMA are within the variance of those reported in Table A.2, outperforming all other algorithms. The same is true for all other values of β , where hierarchical algorithms that perform a move-to-front operation on the sublist level outperformed all other algorithms at all distributions. For example, for the Zipf distribution with $\beta = 0.1$, the asymptotic and amortized costs for the MTF-MTF-OMA were 25.65 and 26.16, respectively, as opposed to (35.29, 35.42) for the MTF and (43.49, 43.96) for the TR. The MTF-MTF-OMA and the TR-MTF-OMA continues to deliver a significantly better cost than the MTF and TR, although this gap wanes with increasing values of β . Figure 6.1 illustrates how the algorithms' asymptotic cost are constantly below those for the for the Zipf distribution and different values of β .

For the directed mapping algorithms, the results are tabulated in Table B.2. Again, at $\beta = 0.5$, none of the algorithms is expedient since the cost is near the mean, and at $\beta = 0$ we again see results similar to Table A.2, since the Environment is effectively similar to that of a singly linked list. At lower values of β (i.e., 0.1 and 0.2), both the MTE-MTE-OMA and SWAP-MTE-OMA outperformed all other algorithms for most of the distributions, except for the Lotka and Exponential distributions. For example, for the Zipf distribution at $\beta = 0.1$, the MTE-MTE-OMA yielded an asymptotic cost of 34.26 and amortized cost of 33.73, while the values for the MTE and the SWAP were (41.29, 41.23) and (43.79, 44.13), respectively. At

Figure 6.1: The asymptotic cost of undirected mapping algorithms, in a Markovian environment with the Zipf distribution ($\alpha = 0.9, k = 16$), for different values of β .

$\beta = 0.2$, the MTE-MTE-OMA cost was (46.22, 47.21), while that for the MTE and the SWAP were (51.43, 51.38) and (49.21, 49.42).

With increasing values of β , the dependence-capturing algorithms begin losing ground to the memoryless ones. At $\beta = 0.3$, the MTE-MTE-OMA cost was (57.08, 56.87), and the cost for the MTE and the SWAP were (58.69, 58.68) and (54.38, 54.63). Figure 6.2 shows the change in asymptotic cost of different algorithms with varying values of β for the Zipf distribution. This degradation actually happens much earlier in the Lotka and Exponential distributions, for $\beta \geq 0.1$. We have discussed the properties of these two distributions that make them unsuitable for a dependence-capturing hierarchical list organizing algorithm in Chapter 5. In other distributions, however, the performance was comparable to that of the MTE and the SWAP.

Although the SWAP heuristic always outperforms the MTE, the dominance of algorithms performing a move-to-end operation on the sublist level is well established

Figure 6.2: The asymptotic cost of directed mapping algorithms, in a Markovian environment with the Zipf distribution ($\alpha = 0.9, k = 16$), for different values of β .

over those performing a swap instead. For example, at $\beta = 0.1$ in the Lotka distribution, the asymptotic costs for the MTE, SWAP, MTE-MTE-OMA and SWAP-SWAP-OMA were 32.01, 28.11, 33.74 and 43.15 respectively. This suggests that due to the directional bias of the Environment (say, to the left), it is better to move all records within the local context to the left end of the list. Although this risks an access from the list's right end that could traverse the whole list and add to the cost, the fact that it is more probable to get an access from the left end makes such a risk worthwhile, and yields a superior cost.

It's worth noting that the performance delivered by undirected mapping algorithms was much better than directed mapping ones. For example, at $\beta = 0.1$, the asymptotic cost for the MTF-MTF-OMA was 25.65, as opposed to 34.26 for the MTE-MTE-OMA (See Figure 6.3). However, as we shall see later, undirected mapping algorithms were not even expedient in dually monotonic environments. This

Figure 6.3: Comparing the asymptotic cost of MTF-MTF-OMA to that of MTE-MTE-OMA in a directionally biased Markovian environment with the Zipf distribution, $k = 16$ and different values of β .

means that prior knowledge of the Environment is a helpful factor in choosing which algorithm to deploy.

6.3.2 Dually Monotonic Environment

Using Equation (6.2), we can construct a set of β_i 's that would make the Environment dually monotonic on the local context level, namely, whenever $i < j$, $\beta_i < \beta_j$. Table B.3 shows the results for our tests in this Environment.

In a dually monotonic environment, all undirected hierarchical algorithms were non-expedient. However, for the directed mapping algorithms, the ones that do a SWAP operation on the sublist level (i.e., SWAP-SWAP-OMA and MTE-SWAP-OMA) generally performed better than the other algorithms, and certainly provided better performance than when tested in a directionally biased environment. For

Figure 6.4: The asymptotic cost of various algorithms, in a Markovian environment with the Zipf distribution, $k = 16$ and two values for α . The Environment is dually monotonic on the level of the local contexts.

example, at $\alpha = 0.9$ and the Zipf distribution, the asymptotic cost for the MTE, SWAP, MTE-SWAP-OMA and SWAP-SWAP-OMA were 49.25, 42.89, 40.84 and 40.37, respectively.

The SWAP algorithm, however, outperformed all the others for the Lotka and the Exponential algorithms. Only when α was increased by an order of magnitude to 0.99 did the MTE-SWAP-OMA and SWAP-SWAP-OMA come out as the best algorithms (Figure 6.4). This could be due to the fact that the high value of α causes the OMA to be less prone to errors, thereby yielding a more optimal arrangement of the list.

Notice that unlike in directionally biased environments, algorithms doing a SWAP operation on the sublist level excelled those doing a move-to-end. This is because a dually monotonic environment dictates a ranking of the local contexts (according to β_i) that the corresponding sublists should ideally follow. A SWAP operation is the best way to achieve this ranking, and moving the sublists to the end therefore violates this ranking and significantly increases the cost.

6.4 Conclusion

This Chapter introduced the concept of the doubly linked lists being reorganized even though they are accessed from both ends. The MTE and SWAP algorithms came as natural extensions of the MTF and TR algorithms of the singly linked list, and are applied with respect to the direction of access (directed mapping).

While it is generally conjectured that if an algorithm performs better in singly linked lists its directed mapping would also perform better in doubly linked ones, the SWAP has been formally shown to outperform the MTE in two families of distributions: The dually monotonic and the directionally biased distributions. We tested directed mappings versions of the algorithms that we introduced in Chapter 5, in environments that are either directionally biased or dually monotonic on the level of the local contexts. For the latter, algorithms that did a MTE on the sublist level performed better than all others, while for the former, those that did a SWAP on the sublist level were better. We also tested undirected mapping algorithms in both environments. In a directionally biased environment, the results for the MTF-MTF-OMA and SWAP-MTF-OMA excelled all others. However, in dually monotonic ones, all undirected hierarchical algorithms were not expedient, for all values of β .

In general, the hierarchical algorithms provided better performance for both the asymptotic and amortized costs. However, prior knowledge of the Environment can help in choosing the optimal algorithm for it. Since such knowledge is rarely available *a priori*, a safe choice could be MTE-MTE-OMA, although even this algorithm does not always provide the best performance, notably in the Lotka and Exponential distributions in dually monotonic Environments. We therefore conclude that the hierarchical list organizing algorithms should be applied to the doubly linked list in spite of marginal additional associated costs, if we are willing to reap occasional non-optimal performance for *some* distributions.

Chapter 7

Summary and Conclusion

In this Thesis we have studied the problem of self organizing linear search lists, and in particular in environments exhibiting the locality of reference phenomenon. This phenomenon manifests itself when subsequences of the access sequence have different probability distributions than the overall off-line distribution, because of probabilistic dependence between the access queries.

We have introduced a strategy for reorganizing linear search lists under such environments that we call Hierarchical List Organization. Under such schemes, the list can be viewed as a set of sublists that could be manipulated and reorganized, just as the individual records are. Since the list's structure allows sublist reorganization in constant time by means of pointer manipulation (provided that the sublist boundaries are known), this scheme is very effective for linked lists. A hierarchical list organization operation, denoted in the form X-Y, would involve two operations: operation X carried on within the sublist which the queried element belongs to, and operation Y carried on between the sublists. For example, the MTF-MTF would move the accessed record to the beginning of the sublist it is found in, and then the sublist *itself* would be moved to the front of the list.

Chapter 3 built the case for hierarchical list organization by discussing how the

notion of “working sets” (i.e., how the algorithm’s attention is focused on a subset of the data structure) has already been found successful in other data structures. The benefits of this approach are evident: at no extra reorganization cost, the queried record has been brought to the front of the list alongside with its surrounding context. If this context contains records that are likely to be accessed in the near future, then we have saved the cost of bringing each record of this context to the front. If not, however, then the algorithm’s behaviour is distinctively poor. The question, then, is one of inferring how to arrange the sublists such that they would map the Environment’s different contexts. This problem is known in the literature as the Object Partitioning Problem (OPP).

Chapter 4 explored the different approaches for the OPP. We discussed various algorithms based on the field of learning automata, which were introduced in the beginning of the Chapter. In particular, two algorithms are studied in detail: the Object Migration Automaton (OMA), which is based on fixed structure stochastic automata, and the Modified Linear Reward-Penalty reinforcement scheme ($MLRP$), based on the variable structure stochastic automata.

With the help of these algorithms, we introduced the final version of our hierarchical algorithms strategy in Chapter 5, which would employ an OPP algorithm in conjunction with the hierarchical strategy. Denoted in the form X-Y-Z, the scheme would carry out an extra reorganization step by rearranging records within sublists with the help of the OPP algorithm Z. The access query at time t , paired with that at time $t - 1$, is run by algorithm Z. Upon reward, the hierarchical reorganization takes place, since this pairing matches the algorithm’s partitioning. Upon penalty, the records are rearranged, as dictated by algorithm Z, to better reflect the Environment’s partitioning.

We tested permutations of the Move-To-Front (MTF) and Transposition (TR) heuristics, the most common list organization heuristics, together withn the OMA

and the ML_{RP} . Through algorithms that did a MTF operation on the sublist level, we were able to achieve results that, *by far*, excelled the best results reported in the literature for the MTF rule, widely acknowledged as the best heuristic for use in dependent environments. This was consistent for most of the probability distributions and different sublist sizes tested. Additionally, by using the ML_{RP} , we were able to achieve very fast convergence rates.

We also implemented periodic variations of our hierarchical algorithms. In a periodic setting, the environment switches context after exactly T access queries, after which other contexts replace it in turn. When the period expires, we know that the current context would not be in focus again for a long time, so we send it to the back of the list. Using this techniques we were able to achieve asymptotic costs that are very close to the optimal costs. When the algorithm does not know the period, we simply consider that a penalty from the partitioning algorithm means the period has expired, and accordingly send the context to the end of the list. This technique not only delivered excellent results in periodic environments, but also performed very well in Markovian ones as well. We therefore recommend using this technique whenever the Environment's properties are unknown.

For doubly linked lists, which have the property of being accessible from both left and right ends, we explored how our algorithms could be extended to account for such accesses. We implemented hierarchical schemes for the Move-To-End (MTE) and the SWAP algorithms, which are extensions to the MTF and the TR, applied with respect to the direction of the access query. Our algorithms still yielded better performance, although not as impressive as those for the singly linked list. Again, as with the case of singly linked lists, prior knowledge of the Environment could help in selecting the most appropriate algorithm.

In summary, we list here the experimental results that we obtained in this Thesis:

- In singly linked lists, the MTF-MTF-OMA and TR-MTF-OMA delivered performance that by far excelled the MTF in environments with high locality of reference.
- The MTF-MTF- ML_{RP} matched the performance of the MTF-MTF-OMA, while, at the same time, it delivered a much quicker convergence rate.
- The MTF-MTF-OMA-Periodic delivered performance that approached that of the ideal off-line algorithm in periodic environments, if the period is known to the algorithm.
- The MTF-MTF-OMA-Unknown_Period delivered performance that matched that of the MTF-MTF-OMA, as well as the MTF-MTF-OMA-Periodic, making it suitable for both kinds of switching environments discussed in this Thesis.
- For the case of doubly linked lists, where the probability of access from the left is higher than the right for all records (the so-called “directionally biased” distributions), the MTF-MTF-OMA and TR-MTF-OMA were still the best choices for list organization.
- In other environments (such as the so-called “dually monotonic” environments on the level of the local context), the best algorithms for most distributions were the MTE-SWAP-OMA and SWAP-SWAP-OMA.

In a nutshell, we submit that the most significant contribution of the Thesis is the novel way of viewing the list as a set of sublists, that are manageable using the same strategy used to manage individual records. To our knowledge, such an approach has not been explored before.

7.1 Future Work

It is important to stress that the general structure of hierarchical list organization techniques is independent of the underlying organization heuristics and partitioning algorithms. This opens the door to experimentation with permutations of virtually all of the organization techniques and partitioning algorithms available in the literature. We can, for example, experiment with MTP-MTP-SMA or SWP-SWP-BAM, etc. However, we think that the permutations of the MTF and TR should cover most of the interesting cases. This is because the idea behind most other algorithms is to compromise between the speed of the MTF and accuracy of the TR. We, for our part, get this compromise “for free” through our hierarchical reorganization strategies.

One other field of potential future work is to relax the condition that the different local contexts in the dependence models are of equal sizes (and consequently, the sublist sizes). The $MLRP$ already provides a means for capturing dependence into partitions of various sizes. We only need to define dependence models which work with different sizes of local contexts. We could also look into other possibilities for environments, such as overlapping contexts, where each context does not have to be disjoint from the others. As well, other models for the doubly linked lists could be researched and investigated, since the dually monotonic and directionally biased distributions are not capable of covering all the possible environments for DLLs.

As we have seen in DLLs, the philosophy behind hierarchical reorganization is easily extensible to other types of data structures. One possible area for research would involve self organizing trees, which has been extensively studied in the literature. It is worth noting, however, that such a research endeavour would not be a straightforward application of our techniques to trees. This is because tree reorganization operations do not exhibit the extremes that the MTF and the TR show in lists. For example, moving a record to the root of the tree and adjusting the tree

accordingly would not result in as big a “penalty” as the MTF presents, if the query is an outlier one. As well, moving a record to the its parent level is not as conservative as the TR, because of the logarithmic nature of searching in the tree. Collectively, this leads us to believe that our techniques, though still useful, may not exhibit the same performance enhancements as in the case of lists.

Additionally, it would be interesting to see how the algorithms would perform in a real life setting, by using real life access sequences, instead of environment models.

Finally, and most importantly, formal analysis remains open for these methods. Since the theoretical work on such environments is scarce, we believe that analysis will be far from trivial.

Bibliography

- [1] S. Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27:670–681, 1998.
- [2] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21:312–329, 1998.
- [3] S. Albers, B. von Stengel, and R. Werchner. A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.
- [4] S. Albers and J. Westbrook. Self-organizing data structures. In Amos Fiat and Gerhard Woeginger, editors, *Online Algorithms: The State of the Art*, pages 13–51. Springer LNCS 1442, 1998.
- [5] E. J. Anderson, P. Nash, and R. R. Weber. A counterexample to a conjecture on optimal list ordering. *Journal of Applied Probability*, 19(3):730–732, 1982.
- [6] R. Bachrach, R. El-Yaniv, and M. Reinstadtler. On the competitive theory and practice of online list accessing algorithms. *Algorithmica*, 32(2):201–245, 2002.
- [7] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, 1985.

- [8] J. R. Bitner. Heuristics that dynamically organize data structures. *SIAM Journal on Computing*, 8(1):82–110, 1979.
- [9] P. Chassaing. Optimality of move-to-front for self-organizing data structures. *Annals of Applied Probability*, 3(4):1219–1240, 1993.
- [10] F. R. K. Chung, D. J. Hajela, and P. D. Seymour. Self-organizing sequential search and hilbert’s inequality. In *Proceedings of the 17th Annual Symposium on the Theory of Computing*, pages 217–223, 1985.
- [11] J. Dong. Time reversible self-organizing sequential search algorithms. Master’s thesis, Carleton University, 1997.
- [12] G. H. Gonnet, J. I. Munro, and H. Suwanda. Exegesis of self-organizing linear search. *SIAM Journal on Computing*, 10(3):613–637, 1981.
- [13] J. Han and M. Kamber. *Data mining: Concepts and techniques*. Morgan Kaufmann, 2001.
- [14] W. J. Hendricks. An account of self-organizing systems. *SIAM Journal on Computing*, 5(4):715–723, 1976.
- [15] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17(3):295–311, 1985.
- [16] J. H. Hester and D. S. Hirschberg. Self-organizing search lists using probabilistic back-pointers. *Communications of the ACM*, 30(12):1074–1079, 1987.
- [17] J. Iacono. Alternatives to splay trees with $o(\log n)$ worst-case access times. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 516–522, 2001.

- [18] S. Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.
- [19] Y. C. Kan and S. M. Ross. Optimal list order under partial memory constraints. *Journal of Applied Probability*, 17:1004–1015, 1980.
- [20] D. E. Knuth. *The art of computer programming*. Addison-Wesley, 1973.
- [21] K. Lam, M. Y. Leung, and M. K. Siu. Self-organizing files with dependent accesses. *Journal of Applied Probability*, 21:343–359, 1984.
- [22] D. Matthews, D. Rotem, and E. Bretholz. Self-organizing doubly linked lists. *International Journal of Computer Mathematics*, 8:99–106, 1980.
- [23] J. McCabe. On serial files with relocatable records. *Operations Research*, 12:609–618, 1965.
- [24] K. S. Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction*. Prentice Hall, 1989.
- [25] D. T. H. Ng and B. J. Oommen. Generalizing singly-linked list reorganizing heuristics for doubly-linked lists. In *Proceedings of the 1989 Conference on the Mathematical Foundations of Computer Science*, pages 380–389, 1989.
- [26] B. J. Oommen. Absorbing and ergodic discretized two-action learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(2):282–293, 1986.
- [27] B. J. Oommen and J. Dong. Generalized swap-with-parent schemes for self-organizing sequential linear lists. In *Proceedings of ISAAC'97, the 1997 International Symposium on Algorithms and Computation*, pages 414–423, 1997.

- [28] B. J. Oommen and E. R. Hansen. List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations. *SIAM Journal on Computing*, 16(4):705–716, 1987.
- [29] B. J. Oommen, E. R. Hansen, and J. I. Munro. Deterministic optimal and expedient move-to-rear list organizing strategies. *Theoretical Computer Science*, 74:183–197, 1990.
- [30] B. J. Oommen and D. C. Y. Ma. Deterministic learning automata solutions to the equipartitioning problem. *IEEE Transactions on Computers*, 37(1), 1988.
- [31] B. J. Oommen and D. C. Y. Ma. Stochastic automata solutions to the object partitioning problem. *The Computer Journal*, 35:A105–A120, 1992.
- [32] B. J. Oommen and D. T. H. Ng. An optimal absorbing list organization strategy with constant memory requirements. *Theoretical Computer Science*, 119:355–361, 1993.
- [33] N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.
- [34] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.
- [35] S. M. Ross. *Introducton to Probability Models*. Academic Press, London, 2 edition, 1980.
- [36] F. Schulz. Two new families of list update algorithms. In *Proceedings of the 9th International Symposium on Algorithms and Computation (ISAAC)*, pages 99–108. LNCS 1533, 1998.

- [37] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
- [38] A. M. Tenenbaum. Simulations of dynamic sequential search algorithms. *Communications of the ACM*, 21(9):790–791, 1978.
- [39] A. M. Tenenbaum and R. M. Nemes. Two spectra of self-organizing sequential search algorithms. *SIAM Journal on Computing*, 11:557–566, 1982.
- [40] R. S. Valiveti and B. J. Oommen. Self-organizing doubly-linked lists. *Journal of Algorithms*, 14:88–114, 1993.
- [41] C. T. Yu, M. K. Siu, K. Lam, and F. Tai. Adaptive clustering schemes: General framework. In *Proceedings of the IEEE COMPSAC*, pages 81–89, 1981.

Appendix A

Results for Singly Linked List Algorithms

This Appendix tabulates the results of experiments done on the singly linked list algorithms discussed in Chapter 5. The experiments were done on lists of 128 records for 5,000,000 queries. The records of the list are initially uniformly distributed. The amortized cost is calculated as the averaged sum of all access costs, while the asymptotic cost is calculated as the averaged sum of the last 5,000 queries only. The experiments included varying numbers of sublists as well varying degrees of locality of reference.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$k = 2$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	34.42	34.97	41.11	41.11	6.74	6.97	3.77	3.81	52.13	51.58
TR	26.32	26.80	32.28	32.26	5.50	5.30	3.42	3.48	43.89	43.91
MTF-MTF-OMA	29.31	29.52	33.00	32.72	10.38	10.29	10.95	11.00	39.11	38.42
MTF-MTF-MLRP	29.71	29.30	31.54	32.46	6.55	7.36	3.82	3.94	37.90	37.94
TR-MTF-OMA	24.94	25.35	28.60	28.47	4.13	14.55	8.58	9.08	32.72	33.85
MTF-MTF-OMA-P.	32.81	33.29	37.46	36.54	15.99	15.66	15.21	15.47	41.58	41.96
TR-MTF-OMA-P.	29.19	29.19	31.71	31.95	15.80	15.70	17.41	17.53	36.95	37.58
MTF-MTF-OMA-U.P.	25.12	25.09	28.23	28.32	10.74	11.05	10.30	12.41	32.50	33.65
TR-MTF-OMA-U.P.	21.11	37.34	24.10	23.77	9.76	10.20	15.09	14.83	29.00	29.58
$k = 4$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	35.13	35.71	40.26	40.43	9.86	9.94	5.98	5.80	46.89	46.59
TR	29.88	30.16	34.48	35.10	8.64	8.07	5.97	6.04	44.44	44.15
MTF-MTF-OMA	23.16	22.74	23.46	23.70	12.20	13.03	11.47	11.56	26.57	26.00
MTF-MTF-MLRP	23.39	22.27	26.84	25.67	15.60	15.13	7.43	7.65	26.41	25.58
TR-MTF-OMA	20.17	21.00	20.86	21.83	13.19	12.60	10.59	10.79	23.75	24.09
MTF-MTF-OMA-P.	29.20	29.64	30.90	31.03	21.30	21.46	17.63	18.16	32.48	32.85
TR-MTF-OMA-P.	27.95	27.54	28.65	28.90	20.76	20.77	19.70	20.12	31.09	31.10
MTF-MTF-OMA-U.P.	21.27	20.70	21.99	22.38	16.46	16.38	12.32	15.62	23.26	24.19
TR-MTF-OMA-U.P.	18.27	18.35	19.26	20.17	11.48	15.82	16.57	18.51	21.71	22.16
$k = 8$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	32.63	33.61	37.03	36.67	14.10	13.36	9.43	9.72	37.03	38.77
TR	34.20	34.16	38.82	38.43	12.09	12.41	11.51	11.55	44.66	44.57
MTF-MTF-OMA	18.29	18.70	18.46	19.05	14.48	14.57	13.84	13.78	19.96	20.57
MTF-MTF-MLRP	17.54	18.12	17.59	18.66	14.57	14.78	14.79	12.90	19.02	19.25
TR-MTF-OMA	17.78	18.20	17.03	18.70	14.21	14.16	11.01	12.34	17.82	19.99
MTF-MTF-OMA-P.	26.18	27.58	27.95	28.60	24.45	24.34	22.10	22.85	27.85	29.29
TR-MTF-OMA-P.	26.06	26.59	26.77	27.27	23.88	23.47	22.86	23.24	27.31	29.33
MTF-MTF-OMA-U.P.	17.66	18.66	18.96	19.92	16.59	17.93	19.38	19.45	18.67	21.98
TR-MTF-OMA-U.P.	16.41	18.93	16.70	17.57	14.31	17.98	18.28	18.64	17.64	19.21

Table A.1: Empirical results for singly linked list algorithms with varying number of sublists ($k = 2, 4, 8$), in a Markovian environment with $\alpha = 0.9$.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$k = 16$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	28.38	28.15	28.40	29.57	15.95	16.05	16.09	16.46	30.09	29.56
TR	39.45	38.81	42.81	42.36	19.10	18.82	22.68	22.50	44.73	45.39
MTF-MTF-OMA	16.07	16.35	15.39	16.59	14.42	15.37	15.20	15.58	15.47	16.59
MTF-MTF-MLRP	15.87	15.75	15.04	15.93	14.41	14.33	13.95	14.53	17.01	16.25
TR-MTF-OMA	15.36	16.12	17.10	16.53	14.22	15.12	15.50	15.77	15.97	17.05
MTF-MTF-OMA-P.	25.27	26.16	25.57	25.94	24.66	25.10	24.20	25.71	26.73	26.47
TR-MTF-OMA-P.	25.12	25.92	25.69	26.22	24.10	25.09	25.11	25.79	24.80	26.54
MTF-MTF-OMA-U.P.	15.65	17.53	16.76	17.66	15.75	17.43	19.45	22.57	17.37	18.25
TR-MTF-OMA-U.P.	15.89	17.38	16.29	17.71	16.14	16.97	19.19	20.85	16.52	17.62
$k = 32$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	20.45	20.35	20.16	20.75	16.28	15.80	18.69	18.68	21.38	20.50
TR	43.22	44.01	45.81	46.70	28.13	27.87	37.07	37.72	47.95	46.97
MTF-MTF-OMA	13.26	14.40	14.84	14.53	14.20	13.94	14.49	14.24	14.39	14.75
MTF-MTF-MLRP	14.17	14.37	15.41	14.72	13.48	13.85	13.01	14.20	14.87	14.43
TR-MTF-OMA	14.39	14.38	15.07	14.43	13.78	13.87	14.10	14.26	13.95	14.49
MTF-MTF-OMA-P.	24.61	24.40	23.48	24.44	23.89	23.91	23.84	24.25	23.53	24.58
TR-MTF-OMA-P.	24.84	24.30	24.91	24.45	23.41	23.89	23.67	24.24	24.78	24.44
MTF-MTF-OMA-U.P.	13.91	14.12	14.16	14.30	13.96	13.71	14.47	14.04	14.12	14.24
TR-MTF-OMA-U.P.	12.92	14.09	15.79	14.57	13.92	13.69	13.24	14.38	13.60	14.48
$k = 64$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	12.85	12.81	13.25	12.90	11.60	11.97	12.98	12.83	13.40	12.80
TR	48.78	49.87	48.46	51.35	40.27	40.11	48.95	49.81	50.81	49.81
MTF-MTF-OMA	13.82	13.57	13.92	13.60	14.29	13.47	13.33	13.60	13.52	13.61
MTF-MTF-MLRP	14.15	13.61	13.66	13.58	13.80	13.46	13.84	13.61	13.75	13.57
TR-MTF-OMA	13.75	13.56	13.73	13.61	13.00	13.50	13.05	13.58	13.49	13.59
MTF-MTF-OMA-P.	23.25	23.71	22.94	23.75	24.37	23.64	23.48	23.75	23.85	23.79
TR-MTF-OMA-P.	23.39	23.71	24.30	23.77	23.74	23.60	23.77	23.75	24.38	23.71
MTF-MTF-OMA-U.P.	13.54	13.44	14.64	13.43	13.46	13.32	13.12	13.41	13.05	13.40
TR-MTF-OMA-U.P.	14.49	13.44	13.03	13.43	13.63	13.31	12.78	13.44	12.42	13.45

Table A.2: Empirical results for singly linked list algorithms with varying number of sublists ($k = 16, 32, 64$), in a Markovian environment with $\alpha = 0.9$.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$k = 2$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	36.29	35.98	42.16	42.31	6.97	7.21	4.07	4.11	53.58	53.43
TR	26.30	26.90	31.99	32.31	5.42	5.38	3.64	3.67	43.77	43.96
MTF-MTF-OMA	30.89	31.30	34.38	40.53	17.11	17.11	11.96	11.76	39.58	40.05
MTF-MTF-MLRP	31.52	31.03	33.68	34.19	16.83	16.77	5.33	5.31	43.61	43.97
TR-MTF-OMA	27.13	27.07	29.52	29.94	16.25	16.32	9.88	9.34	35.52	35.66
MTF-MTF-OMA-P.	18.12	18.62	21.62	21.85	15.96	15.56	2.73	2.83	26.92	27.16
TR-MTF-OMA-P.	14.28	14.41	17.40	17.23	3.45	3.70	3.42	3.15	22.81	22.91
MTF-MTF-OMA-U.P.	25.69	26.14	28.53	28.34	10.51	10.96	9.2	9.23	33.19	33.59
TR-MTF-OMA-U.P.	38.67	38.43	23.48	23.72	10.73	10.66	8.97	8.97	29.37	29.28
$k = 4$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	39.92	40.18	46.11	45.72	10.91	11.04	7.02	7.27	53.94	53.85
TR	30.29	30.41	35.58	35.28	8.52	8.50	6.57	6.68	43.93	44.27
MTF-MTF-OMA	30.18	30.37	31.47	31.76	10.92	10.99	13.23	13.23	33.06	33.30
MTF-MTF-MLRP	29.78	29.68	31.07	31.03	22.58	22.56	10.96	10.89	36.79	36.84
TR-MTF-OMA	29.94	30.61	29.10	29.49	10.27	10.86	13.05	13.66	31.15	31.72
MTF-MTF-OMA-P.	10.85	10.99	12.11	17.94	12.19	12.44	17.86	17.89	13.77	14.27
TR-MTF-OMA-P.	8.86	13.37	10.81	10.66	13.87	14.09	11.20	10.95	11.92	12.28
MTF-MTF-OMA-U.P.	18.34	18.64	21.72	21.80	15.00	14.88	11.30	11.58	16.94	18.70
TR-MTF-OMA-U.P.	14.83	14.65	13.77	14.96	23.05	24.49	12.80	13.30	15.90	16.74
$k = 8$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	43.25	44.15	47.99	48.75	16.90	16.40	13.91	13.60	52.98	53.75
TR	34.87	34.73	38.31	38.91	13.69	13.77	12.72	12.72	44.83	44.92
MTF-MTF-OMA	29.88	30.45	30.06	30.62	22.07	19.46	21.50	21.67	31.76	31.75
MTF-MTF-MLRP	28.66	28.75	29.10	29.07	25.12	25.18	22.12	21.95	31.24	31.39
TR-MTF-OMA	30.65	30.73	29.93	30.98	21.42	21.39	16.50	17.24	30.26	30.97
MTF-MTF-OMA-P.	7.68	8.55	8.91	9.51	9.20	9.78	14.19	15.07	7.52	9.17
TR-MTF-OMA-P.	8.88	10.98	6.67	9.64	8.90	10.74	12.43	13.18	7.04	7.62
MTF-MTF-OMA-U.P.	9.90	26.39	20.33	20.96	14.09	14.77	13.97	16.63	12.50	16.96
TR-MTF-OMA-U.P.	12.50	14.06	13.41	16.32	10.32	12.45	22.56	22.25	10.09	12.51

Table A.3: Empirical results for singly linked list algorithms with varying number of sublists ($k = 2, 4, 8$), in a periodic environment with period = 10.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$k = 16$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	45.53	45.67	49.41	49.08	22.79	22.56	24.94	25.28	50.15	50.65
TR	40.22	39.88	42.73	42.29	21.38	21.17	24.32	24.21	45.85	46.22
MTF-MTF-OMA	30.67	31.07	31.04	31.29	23.65	23.41	22.38	23.09	29.95	30.72
MTF-MTF-MLRP	27.65	27.65	28.48	28.46	26.21	26.23	26.50	26.45	27.97	27.96
TR-MTF-OMA	28.79	29.50	29.31	30.69	27.82	27.22	24.22	22.79	32.05	33.17
MTF-MTF-OMA-P.	5.86	7.29	6.15	8.74	6.00	7.18	8.63	10.04	8.29	9.25
TR-MTF-OMA-P.	7.03	7.50	4.84	6.08	6.32	7.22	8.52	7.02	6.13	6.53
MTF-MTF-OMA-U.P.	19.23	19.64	18.59	21.90	16.89	16.65	15.33	18.67	25.82	24.88
TR-MTF-OMA-U.P.	23.61	25.59	23.53	26.63	16.42	17.87	12.74	21.55	26.96	27.82
$k = 32$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	40.55	40.33	42.11	41.88	26.09	27.01	34.94	34.92	40.49	40.86
TR	46.14	46.18	49.66	48.66	31.93	31.56	40.17	40.17	48.61	48.72
MTF-MTF-OMA	27.05	27.06	27.10	27.12	26.55	26.56	27.64	27.45	27.08	27.14
MTF-MTF-MLRP	27.33	27.39	27.09	27.11	26.53	26.54	26.86	26.86	27.11	27.10
TR-MTF-OMA	27.02	27.17	27.09	27.10	26.49	26.53	26.81	26.90	27.08	27.10
MTF-MTF-OMA-P.	2.26	2.28	2.31	2.44	1.75	1.86	2.05	2.15	3.71	3.74
TR-MTF-OMA-P.	2.21	2.27	2.29	2.49	2.25	2.33	2.01	2.20	2.53	2.80
MTF-MTF-OMA-U.P.	2.62	4.33	2.68	2.99	3.77	5.30	6.66	6.50	2.69	3.02
TR-MTF-OMA-U.P.	5.91	6.13	7.32	8.18	3.06	4.85	2.41	2.88	2.65	2.87
$k = 64$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	26.35	26.20	26.29	26.34	24.17	23.94	26.15	26.20	26.37	26.20
TR	53.18	54.13	55.52	55.47	45.72	45.63	54.74	54.09	54.29	54.10
MTF-MTF-OMA	26.64	26.64	26.65	26.65	26.51	26.51	26.64	26.64	26.65	26.64
MTF-MTF-MLRP	26.64	26.65	26.65	26.66	26.51	26.52	26.63	26.65	26.64	26.65
TR-MTF-OMA	26.64	26.64	26.65	26.65	26.51	26.51	26.65	26.64	26.64	26.64
MTF-MTF-OMA-P.	1.43	1.44	1.46	1.45	1.33	1.32	1.45	1.44	1.43	1.44
TR-MTF-OMA-P.	1.44	1.44	1.45	1.45	1.30	1.32	1.44	1.44	1.44	1.44
MTF-MTF-OMA-U.P.	1.64	1.64	1.65	1.65	1.52	1.52	1.65	1.64	1.64	1.64
TR-MTF-OMA-U.P.	1.65	1.64	1.65	1.65	1.51	1.52	1.64	1.64	1.64	1.64

Table A.4: Empirical results for singly linked list algorithms with varying number of sublists ($k = 16, 32, 64$), in a periodic environment with period = 10.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$\alpha = 0.1$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	51.16	50.59	53.14	53.75	28.74	28.11	31.16	31.76	55.70	55.43
TR	40.04	39.81	43.89	43.22	21.28	21.02	24.29	24.11	46.21	46.15
MTF-MTF-OMA	60.17	59.85	61.28	61.10	45.26	46.13	50.57	51.13	62.04	62.23
TR-MTF-OMA	59.81	59.90	60.58	61.11	47.23	46.19	51.22	51.15	62.70	62.29
MTF-MTF-OMA-P.	60.32	60.93	61.30	61.88	50.72	50.88	53.89	54.45	62.33	62.71
TR-MTF-OMA-P.	59.95	60.97	62.32	61.89	51.49	50.94	54.19	54.45	61.84	62.74
MTF-MTF-OMA-U.P.	62.42	62.24	62.88	62.76	58.64	58.29	58.23	59.07	63.82	63.25
TR-MTF-OMA-U.P.	62.97	62.30	61.72	62.78	58.34	58.32	57.95	59.08	64.00	63.24
$\alpha = 0.2$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	50.34	49.82	52.49	52.97	27.70	27.44	31.66	31.05	54.91	54.57
TR	38.99	39.81	42.65	43.21	20.78	20.99	24.36	24.08	46.56	46.13
MTF-MTF-OMA	59.30	58.73	60.31	60.01	44.58	44.86	50.13	49.79	60.14	61.18
TR-MTF-OMA	58.01	58.79	59.20	60.07	44.67	44.93	49.37	49.88	61.12	61.30
MTF-MTF-OMA-P.	60.28	60.01	60.78	61.00	49.92	49.70	53.75	53.32	61.84	61.88
TR-MTF-OMA-P.	59.79	60.13	60.94	61.06	49.39	49.76	53.62	53.40	62.01	61.94
MTF-MTF-OMA-U.P.	60.69	61.08	60.97	61.69	56.39	56.02	57.73	57.50	61.77	62.21
TR-MTF-OMA-U.P.	61.06	61.15	61.38	61.72	55.66	56.09	57.82	57.53	62.55	62.29
$\alpha = 0.3$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	48.87	48.87	52.64	51.94	25.99	26.73	30.65	30.23	53.56	53.49
TR	40.31	39.77	43.24	43.20	20.19	20.90	23.58	24.01	46.41	46.13
MTF-MTF-OMA	58.14	57.36	60.03	58.69	42.62	43.34	47.73	48.16	59.81	59.96
TR-MTF-OMA	57.53	57.46	60.07	58.80	44.36	43.44	47.72	48.26	60.12	60.05
MTF-MTF-OMA-P.	59.03	58.90	59.59	59.97	48.93	48.40	52.57	52.03	60.65	60.86
TR-MTF-OMA-P.	59.15	59.04	60.28	60.05	48.75	48.50	51.49	52.06	60.77	60.97
MTF-MTF-OMA-U.P.	59.01	59.82	60.47	60.43	53.16	53.94	55.70	56.09	60.64	61.07
TR-MTF-OMA-U.P.	58.44	59.95	60.46	60.55	52.60	54.08	57.22	56.21	61.04	61.16
$\alpha = 0.4$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	48.75	47.73	51.11	50.78	26.23	25.89	28.75	29.20	52.31	52.16
TR	39.63	39.72	43.69	43.17	20.84	20.83	23.79	23.97	46.03	46.10
MTF-MTF-OMA	55.44	55.67	57.61	57.05	41.58	41.50	46.02	46.29	58.45	58.32
TR-MTF-OMA	53.90	55.78	57.40	57.23	41.91	41.70	45.85	46.37	59.30	58.43
MTF-MTF-OMA-P.	58.74	57.61	59.32	58.63	46.33	46.93	49.33	50.54	59.68	59.62
TR-MTF-OMA-P.	58.19	57.73	59.22	58.78	47.50	47.09	50.67	50.69	60.99	59.74
MTF-MTF-OMA-U.P.	57.90	58.45	59.26	59.18	51.65	52.02	54.49	54.89	60.01	59.76
TR-MTF-OMA-U.P.	58.81	58.57	59.70	59.24	51.62	52.17	57.14	54.98	60.04	59.89
$\alpha = 0.5$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	46.73	46.21	49.12	49.12	24.46	24.87	28.10	27.96	51.51	50.41
TR	39.86	39.68	42.72	43.12	20.64	20.70	23.86	23.90	45.84	46.09
MTF-MTF-OMA	52.69	53.52	54.89	54.92	39.08	39.40	44.16	43.95	55.83	56.17
TR-MTF-OMA	54.08	53.68	54.15	55.11	39.48	39.65	43.90	44.11	56.36	56.35
MTF-MTF-OMA-P.	56.58	55.94	57.36	57.02	44.80	45.26	48.86	48.86	57.80	57.98
TR-MTF-OMA-P.	55.79	56.11	58.31	57.22	46.04	45.48	48.68	49.03	58.00	58.10
MTF-MTF-OMA-U.P.	58.25	56.99	58.95	57.66	49.59	50.18	53.24	53.69	58.00	58.38
TR-MTF-OMA-U.P.	57.39	57.18	57.54	57.84	51.14	50.32	54.63	53.85	59.19	58.54

Table A.5: Empirical results for singly linked list algorithms with varying levels of locality of reference ($\alpha = 0.1 - 0.5$), where the number of sublists $k = 16$.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$\alpha = 0.6$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	45.23	44.17	46.23	46.98	24.24	23.66	26.40	26.35	47.63	48.03
TR	39.91	39.61	42.47	43.07	20.85	20.54	23.37	23.77	45.81	46.03
MTF-MTF-OMA	51.53	50.72	51.39	52.14	36.93	36.87	40.79	41.15	52.22	53.30
TR-MTF-OMA	49.70	50.92	51.28	52.31	36.46	37.05	41.22	41.30	53.18	53.52
MTF-MTF-OMA-P.	53.75	53.86	54.47	54.94	42.71	43.35	46.93	46.91	55.82	55.83
TR-MTF-OMA-P.	54.93	54.04	55.40	55.14	43.09	43.59	48.00	47.04	56.57	56.01
MTF-MTF-OMA-U.P.	55.10	55.49	57.23	56.18	48.61	48.22	52.10	52.21	56.66	56.96
TR-MTF-OMA-U.P.	55.55	55.69	57.19	56.40	48.79	48.39	51.83	52.33	57.76	57.13
$\alpha = 0.7$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	41.42	41.26	43.91	43.77	22.52	22.09	23.56	24.28	46.04	44.62
TR	39.37	39.52	41.74	42.95	20.22	20.24	23.59	23.77	45.35	45.95
MTF-MTF-OMA	46.04	46.06	35.36	37.73	34.09	33.69	37.07	36.11	36.11	38.08
TR-MTF-OMA	35.21	43.59	34.19	37.50	33.80	33.88	36.58	36.40	36.09	38.41
MTF-MTF-OMA-P.	40.52	46.65	42.75	44.38	40.56	41.12	44.91	43.66	40.63	43.17
TR-MTF-OMA-P.	40.87	43.93	41.36	43.17	40.10	41.28	42.82	43.68	41.97	43.02
MTF-MTF-OMA-U.P.	54.36	52.60	36.13	35.62	48.54	45.79	45.78	47.26	34.78	38.05
TR-MTF-OMA-U.P.	34.37	46.91	34.44	37.87	45.98	45.96	47.32	47.34	34.46	36.50
$\alpha = 0.8$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	36.75	36.66	40.51	38.82	19.39	19.84	21.53	21.30	38.03	39.29
TR	38.73	39.30	43.11	42.77	19.35	19.78	23.01	23.28	46.11	45.81
MTF-MTF-OMA	27.17	26.92	27.52	27.38	25.19	26.01	26.21	25.89	27.31	27.32
TR-MTF-OMA	26.12	26.74	26.72	26.96	28.01	28.29	25.84	25.48	26.02	27.13
MTF-MTF-OMA-P.	34.37	34.10	33.89	34.75	36.97	36.79	32.36	33.15	33.58	34.68
TR-MTF-OMA-P.	33.66	34.00	33.40	34.14	37.50	37.03	34.21	33.31	34.22	34.80
MTF-MTF-OMA-U.P.	26.31	27.10	26.74	27.49	42.01	39.81	26.02	26.89	26.02	27.92
TR-MTF-OMA-U.P.	26.61	27.18	26.00	26.93	40.03	39.24	26.94	27.01	25.83	27.20
$\alpha = 0.9$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	26.79	28.14	27.53	29.55	16.65	16.05	16.06	16.47	29.56	29.56
TR	37.58	38.83	42.34	42.36	19.24	18.86	22.46	22.55	45.52	45.42
MTF-MTF-OMA	16.06	16.44	15.93	16.66	14.89	15.27	14.33	15.63	16.24	16.76
TR-MTF-OMA	15.36	16.12	16.16	16.54	15.64	15.40	14.98	15.42	16.16	16.58
MTF-MTF-OMA-P.	25.44	26.15	26.41	26.31	24.33	24.94	25.39	25.96	25.31	26.78
TR-MTF-OMA-P.	25.71	25.81	25.01	25.98	24.09	25.03	25.53	26.02	26.62	26.06
MTF-MTF-OMA-U.P.	15.12	17.28	18.00	17.83	14.77	16.64	19.49	22.73	16.60	17.55
TR-MTF-OMA-U.P.	16.47	17.09	15.03	17.08	14.68	15.96	20.68	20.78	17.44	17.73

Table A.6: Empirical results for singly linked list algorithms with varying levels of locality of reference ($\alpha = 0.6 - 0.9$), where the number of sublists $k = 16$.

Appendix B

Results for Doubly Linked List Algorithms

This Appendix tabulates the results of experiments done on the doubly linked list algorithms discussed in Chapter 6. The experiment setup was the same as in Appendix A, except for the probability of an access from the right end of the list β which is indicated in the table caption.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$\beta = 0.0$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	27.35	28.13	29.99	29.56	16.43	16.05	16.21	16.50	29.09	29.62
TR	38.57	38.81	42.25	42.34	18.83	18.82	22.05	22.55	45.73	45.44
MTF-MTF-OMA	15.69	16.43	16.56	16.66	14.62	15.40	14.40	15.01	15.75	16.77
MTF-TR-OMA	33.71	33.72	34.13	33.95	31.79	32.37	28.01	27.41	34.74	34.12
TR-MTF-OMA	16.62	16.04	14.33	15.82	14.77	15.42	14.25	15.06	15.82	16.68
TR-TR-OMA	31.87	33.49	33.44	33.21	31.50	32.23	32.13	32.26	33.04	33.88
$\beta = 0.1$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	35.29	35.42	37.11	36.57	24.83	25.76	26.31	26.08	37.43	36.56
TR	43.49	43.96	47.36	46.77	28.84	27.95	31.00	30.92	50.41	49.23
MTF-MTF-OMA	25.65	26.16	25.87	26.35	23.87	25.12	24.44	24.85	25.08	26.40
MTF-TR-OMA	40.67	39.80	41.03	39.90	38.50	38.91	31.32	31.86	38.86	39.84
TR-MTF-OMA	25.03	25.60	25.77	26.00	24.79	25.16	24.50	24.90	25.51	26.03
TR-TR-OMA	37.60	39.67	39.63	39.56	37.57	38.84	32.76	33.31	39.63	39.97
$\beta = 0.2$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	42.35	42.68	43.29	43.52	34.89	35.41	35.02	35.67	43.50	43.54
TR	48.75	49.07	51.68	51.22	37.77	37.09	38.92	39.30	53.24	53.05
MTF-MTF-OMA	35.80	35.56	35.68	35.62	33.49	35.13	33.17	34.88	36.64	35.74
MTF-TR-OMA	46.34	45.85	44.88	46.10	46.92	45.31	43.74	42.80	47.63	46.34
TR-MTF-OMA	33.72	35.30	34.85	35.62	33.58	34.91	35.75	35.00	34.34	35.92
TR-TR-OMA	45.03	45.79	46.35	45.97	42.62	45.20	41.29	41.43	46.91	46.14
$\beta = 0.3$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	49.57	49.94	50.96	50.53	44.78	45.14	44.79	45.30	50.47	50.51
TR	53.09	54.23	55.83	55.66	46.82	46.24	48.60	47.73	56.22	56.85
MTF-MTF-OMA	44.57	45.29	45.68	45.34	44.26	44.99	43.16	44.99	45.21	45.45
MTF-TR-OMA	52.14	52.15	51.20	52.33	50.26	51.64	52.04	49.86	51.63	52.29
TR-MTF-OMA	44.93	45.17	46.32	45.29	44.03	44.76	44.38	44.98	44.06	45.23
TR-TR-OMA	52.30	52.02	53.70	51.91	52.68	51.44	49.88	50.33	53.81	52.28
$\beta = 0.4$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	57.45	57.25	55.70	57.51	55.22	54.78	53.69	54.92	56.95	57.51
TR	59.16	59.36	60.47	60.04	56.30	55.37	56.15	56.13	60.49	60.67
MTF-MTF-OMA	55.60	54.84	55.22	54.89	54.62	54.72	54.63	54.67	55.67	54.96
MTF-TR-OMA	58.72	58.32	58.64	58.40	58.41	58.03	57.60	57.06	58.21	58.45
TR-MTF-OMA	54.49	54.81	54.96	54.85	55.03	54.71	54.82	54.63	56.82	54.91
TR-TR-OMA	57.24	58.31	56.83	58.31	57.99	57.97	57.80	57.08	57.58	58.30
$\beta = 0.5$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTF	63.92	64.53	65.22	64.53	65.56	64.46	64.10	64.51	64.50	64.48
TR	64.08	64.52	64.60	64.48	65.65	64.49	63.21	64.48	64.97	64.48
MTF-MTF-OMA	65.94	64.51	64.14	64.51	65.04	64.46	65.41	64.51	64.18	64.49
MTF-TR-OMA	64.47	64.51	64.20	64.53	64.86	64.46	66.02	64.49	64.13	64.53
TR-MTF-OMA	65.74	64.51	65.37	64.46	64.43	64.53	67.06	64.51	65.53	64.52
TR-TR-OMA	64.07	64.48	65.45	64.51	63.51	64.51	63.46	64.54	63.35	64.51

Table B.1: Empirical results for DLL undirected-mapping algorithms with varying probability of an access from the right end (β), with the number of sublists $k = 16$ in a Markovian environment with $\alpha = 0.9$.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$\beta = 0.0$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	28.70	28.15	29.19	29.57	16.00	16.06	16.55	16.42	29.42	29.60
SWAP	38.45	38.81	42.44	42.36	18.92	18.81	23.54	22.53	44.16	45.39
MTE-MTE-OMA	15.51	16.49	16.48	16.87	15.07	15.38	14.94	15.54	16.97	16.78
MTE-SWAP-OMA	31.11	33.29	32.68	33.89	34.82	31.57	28.16	27.72	32.89	33.85
SWAP-MTE-OMA	15.31	16.00	15.92	15.99	14.85	15.29	15.10	15.30	16.17	16.82
SWAP-SWAP-OMA	32.85	33.21	32.78	33.55	32.04	32.19	25.98	26.55	35.51	33.60
$\beta = 0.1$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	41.29	41.23	41.45	42.14	32.01	33.48	32.99	33.76	42.46	42.12
SWAP	43.79	44.13	48.09	46.98	28.11	28.23	30.76	31.18	49.65	49.35
MTE-MTE-OMA	34.26	33.73	33.58	33.69	33.74	32.90	34.14	32.99	33.74	33.94
MTE-SWAP-OMA	43.35	42.93	45.15	43.05	40.62	42.33	34.45	35.81	45.19	43.06
SWAP-MTE-OMA	33.62	33.48	32.39	33.18	33.61	32.71	31.78	32.99	33.94	33.52
SWAP-SWAP-OMA	44.49	42.49	41.71	42.80	43.15	41.39	36.61	37.56	42.11	43.00
$\beta = 0.2$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	51.43	51.38	51.76	51.84	44.61	47.03	47.56	47.20	52.41	51.93
SWAP	49.21	49.42	51.35	51.56	37.85	37.57	39.28	39.77	52.95	53.36
MTE-MTE-OMA	46.22	47.21	48.45	47.24	47.51	46.84	46.55	46.94	46.15	47.33
MTE-SWAP-OMA	51.23	51.14	51.68	51.33	49.29	50.60	46.99	47.25	51.47	51.45
SWAP-MTE-OMA	46.57	46.88	46.30	47.05	48.01	46.39	47.09	46.39	45.76	47.09
SWAP-SWAP-OMA	52.31	50.93	48.24	51.20	49.59	50.11	46.06	46.78	50.67	51.12
$\beta = 0.3$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	58.70	58.69	58.50	58.89	56.10	56.71	57.64	56.82	58.44	58.88
SWAP	54.39	54.63	55.79	56.00	47.63	46.84	47.33	48.25	57.03	57.24
MTE-MTE-OMA	57.08	56.88	55.53	56.88	56.01	56.68	56.15	56.61	56.87	56.84
MTE-SWAP-OMA	57.66	58.10	57.10	58.14	57.49	57.84	54.61	54.44	57.72	58.05
SWAP-MTE-OMA	56.16	56.56	55.56	56.64	55.87	56.32	56.11	56.36	56.41	56.73
SWAP-SWAP-OMA	58.41	57.83	56.64	57.96	57.84	57.41	57.67	55.67	58.72	58.01
$\beta = 0.4$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	63.50	63.08	61.89	63.05	62.30	62.57	61.64	62.59	62.33	63.10
SWAP	59.58	59.78	60.11	60.47	54.10	55.88	56.42	56.58	61.25	61.03
MTE-MTE-OMA	62.95	62.58	61.57	62.63	60.77	62.53	63.68	62.56	62.73	62.58
MTE-SWAP-OMA	64.09	62.79	62.65	62.85	62.07	62.71	61.45	61.03	62.46	62.83
SWAP-MTE-OMA	63.11	62.53	62.24	62.48	61.22	62.40	61.40	62.43	62.70	62.57
SWAP-SWAP-OMA	61.91	62.73	63.09	62.74	62.22	62.41	61.54	61.06	61.98	62.77
$\beta = 0.5$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	66.25	64.52	65.66	64.48	66.02	64.50	64.61	64.53	63.34	64.51
SWAP	64.59	64.49	64.73	64.51	65.05	64.49	63.63	64.48	64.50	64.52
MTE-MTE-OMA	63.95	64.50	65.08	64.52	64.04	64.54	65.29	64.49	64.73	64.52
MTE-SWAP-OMA	64.43	64.51	63.69	64.51	64.02	64.53	64.35	64.50	64.51	64.50
SWAP-MTE-OMA	64.22	64.54	64.83	64.50	65.21	64.48	65.86	64.51	64.35	64.52
SWAP-SWAP-OMA	65.23	64.51	65.46	64.52	64.61	64.52	64.35	64.49	64.69	64.49

Table B.2: Empirical results for DLL directed-mapping algorithms with varying probability of an access from the right end (β), with the number of sublists $k = 16$ in a Markovian environment with $\alpha = 0.9$.

Strategy	Zipf		80-20		Lotka		Exp		Linear	
$\alpha = 0.9$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	49.26	49.95	49.05	50.15	47.62	48.04	48.47	48.20	50.37	50.19
SWAP	42.89	41.98	42.42	42.59	37.40	38.27	37.58	38.95	43.22	42.90
MTE-MTE-OMA	46.12	48.64	46.46	48.50	46.58	48.10	49.27	48.26	47.60	48.53
MTE-SWAP-OMA	40.85	42.59	41.99	42.58	42.25	42.45	41.80	41.85	43.55	42.57
SWAP-MTE-OMA	49.80	48.12	48.68	48.52	47.41	47.90	48.42	48.08	49.02	48.65
SWAP-SWAP-OMA	40.38	42.48	42.21	42.33	42.39	41.94	42.71	41.90	44.20	42.60
$\alpha = 0.99$	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor	Asym	Amor
MTE	48.87	47.11	48.32	47.06	45.87	46.67	48.75	46.87	52.97	47.24
SWAP	44.38	40.96	44.91	41.79	40.07	37.53	35.61	37.98	43.08	41.87
MTE-MTE-OMA	52.38	47.28	46.87	47.12	44.64	46.49	43.47	46.77	47.63	47.14
MTE-SWAP-OMA	31.13	39.36	35.41	39.20	39.65	39.65	34.85	40.57	36.69	39.34
SWAP-MTE-OMA	49.11	46.79	48.67	46.93	41.92	46.60	43.28	46.46	48.39	46.92
SWAP-SWAP-OMA	40.90	39.71	34.91	39.58	38.87	39.95	37.68	39.26	37.78	39.70

Table B.3: Empirical results for doubly linked list algorithms in a dually monotonic environment (on the working set level), for two different levels of locality of reference α . The number of sublists k is 16.