INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

TIME REVERSIBLE SELF-ORGANIZING SEQUENTIAL SEARCH ALGORITHMS

by

Juan Dong, M.Sc.

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of Master of Science
Information and System Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

April 1997

© Copyright

1997, Juan Dong

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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

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 a accordé une licence non exclusive permettant à la Bibliothèque nationale du 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.

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.

0-612-22138-5

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
**TIME REVERSIBLE SELF-ORGANIZING SEQUENTIAL SEARCH
ALGORITHMS**

submitted by
Juan Dong, M.Sc.
in partial fulfillment of the requirements
for the degree of Master of Science
Information and System Science

Director, School of Computer Science

Thesis Supervisor

Carleton University
April 29, 1997

Abstract

This thesis gives an extensive literature survey on the known self-organizing sequential search algorithms, and introduces two new schemes, the swap-with-parent scheme and the move-to-parent scheme. Both of the new schemes overcome the common drawback of all the previous algorithms which fail to consider the size of the list when reorganizing it.

The Markov chain representing the swap-with-parent scheme has been shown to be time reversible. This property makes the analysis of the swap-with-parent scheme greatly simplified, and permits a rigorous analysis of the convergence properties of the scheme.

This thesis also introduces a new class of time reversible Markovian tree transformation processes. This initiates a new research area in the use and the analysis of “implicit” tree structures in adaptive algorithms. Using this, two classes of self-organizing sequential search schemes, the swap-with-parent-in-an-ss_tree scheme and the move-to-parent-in-an-ss_tree scheme, have been introduced. The properties of the former have been formally proven and the properties of the latter remain uninvestigated.

Various possible extensions which could strengthen and generalize the results presented in this thesis have been addressed.

Acknowledgments

I would like to express my deep appreciation to my supervisor, Prof. John Oommen, for not only suggesting the topic for this thesis but also for his motivation, advice and support. I would also like to thank him for his many constructive comments and suggestions on the results I have submitted to him. His work ethic and constant questioning of published results gave me a realistic portrait of what a good researcher should be.

I am also grateful to my committee, Prof. J. N. K. Rao, Prof. F. Dehne and Prof. F. Fiala for their comments, suggestions, and especially, for their recommending me for the Senate Medal.

Thanks to Jason Morrison for taking the time to discuss the problem with me which provided me with very helpful and motivational comments.

Special thanks must go to Prof. John Oommen again for arranging the necessary financial support during my study.

I would also like to thank my husband, Bing, and my daughter, Andrea, for their support and understanding.

Finally, thanks to Marlene Elliott, Marlene Wilson and Rosemary Carter for all of their help during my study in the school.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Historical Overview	4
1.2 Salient Contributions of the Thesis	8
1.3 Thesis Outline	10
2 Previous Algorithms Review and Analysis	11
2.1 Definitions and Analysis Techniques	11
2.1.1 Cost	12
2.1.2 Convergence	19
2.2 The Move-to-Front Heuristic	22
2.2.1 The Average Search Cost	22
2.2.2 Comparisons with the Cost of the Optimal Ordering	24
2.2.3 Rate of Convergence	27
2.3 The Transposition Heuristic	29

2.3.1	The Average Search Cost	30
2.3.2	Comparisons with the Cost of the Optimal Ordering	43
2.3.3	Rate of Convergence	43
2.4	Hybrid Algorithms of MTF and TR	45
2.4.1	The Move-Ahead- k Heuristic	45
2.4.2	The POS(k) Heuristic and the SWITCH(k) Heuristic	46
2.4.3	Bitner's Hybrid Algorithm	47
2.5	Batched Algorithms	47
2.5.1	The Move-Every- k th-Access Heuristic	47
2.5.2	The k -in-a-Row Heuristic	49
2.5.3	The k -in-a-Batch Heuristic	51
2.6	Probabilistic Algorithms	52
2.6.1	The Stochastic Move-to-Front Scheme	53
2.6.2	The Stochastic Move-to-Rear Scheme	54
2.6.3	The Deterministic Move-to-Rear Scheme	56
2.7	Conclusion	63
3	The Swap-with-Parent Heuristic	65
3.1	Introduction	66
3.2	The Performance of the SWP Heuristic	70
3.3	Implementation Details	84
3.4	Drawbacks of the SWP Heuristic	85
3.5	A Modification to the SWP Heuristic - the Move-to-parent Heuristic	86
3.6	Empirical Results	87
3.7	Conclusion	90

4	General Time Reversible Markovian Lists	92
4.1	Preliminaries on Time Reversible	
	Markov Chains	92
4.2	A Class of Time Reversible Markov Chains	94
4.2.1	Sequential Search Trees	96
4.2.2	The Swap-with-Parent-in-an-SS_Tree Heuristic	99
4.2.3	Drawbacks of the SWPSST Rule and its Modification - the Move-to-Parent-in-an-SS_Tree Rule	107
4.3	Conclusion	108
5	Discussions and Conclusions	109
5.1	Amortized Analysis	109
5.2	Criteria for Choosing an Algorithm	110
5.3	Locality	111
5.4	Conclusions	113
A	Proof of Theorem 3.5	115
B	Proof of Corollary 3.3	118
	References	119

List of Tables

2.1	The Overwork of MTF for Zipf's Law (Adapted from Bitner [79]). . .	29
2.2	The Overwork, r , for Zipf's Distribution (Adapted from Bitner [79]). .	44
2.3	Best k for Different Sized Lists for Move-ahead- k (From Tenenbaum[78]).	46
3.1	Simulation results (cost) for MTF, SWP, MTP and TR rules.	90
3.2	Simulation results (convergence) for MTF, SWP, MTP and TR rules.	90

List of Figures

2.1	State diagram with 3 elements per state using the MTF rule.	16
2.2	Bitner's overwork measure for convergence (Adapted from Bitner[79]).	21
2.3	State diagram with three elements per state using the TR rule.	31
2.4	State diagram with four elements per state using the TR rule.	35
2.5	Simplified state diagram with five elements per state using the TR rule.	37
2.6	Simplified state diagram with $(k + 1)$ elements per state using the TR rule.	38
2.7	Implementation of the MTR rule (Adapted from Oommen et al.[90]).	62
3.1	A list viewed as a "heap".	67
3.2	A s_heap of height 3.	68
3.3	Operations of swapping with parent.	69
3.4	State diagram for a Markov chain using the SWP rule.	70
3.5	Transformations from state π_s to π_t through $\pi_1, \dots, \pi_6$	75
3.6	Transformations from state π_s to π_t through $\pi_1, \dots, \pi_7$	78
3.7	An s_heap of 15 elements.	85
4.1	Tree orderings and transformations on repeated accesses of record "9".	95
4.2	Various ss_trees	97
4.3	Simplified state diagram with four elements per state using the SW- PSST rule with an arbitrary ss_tree.	101

Chapter 1

Introduction

A linear search list is a list of initially unordered records that will be sequentially searched on the basis of a key value associated with each record. It is arranged in such a way that searching for a record in the list can only progress linearly from the first record until the desired record is found or the end of the list is encountered. The list is generally implemented as a linked list.

Consider storage and retrieval of information in a linear search list. We are concerned with the process of retaining and organizing the given list in such a way that the retrieval of information in the list can be made as fast as possible.

If we assume that every element will be accessed as often as every other, then the average cost of accessing a record will be the same regardless of the order of the records in the list. A more realistic assumption would be that some records are accessed more frequently than others. Then the retrieval cost will be minimized by arranging the list such that the more frequently accessed records are near the front of the list.

Suppose we are given a set of records $\{R_1, R_2, \dots, R_n\}$ which are in an arbitrary order π , so that R_i is in position $\pi(i)$ for $1 \leq i \leq n$. At every instant of time one of these records R_i is accessed. We do so, as stated earlier, by examining each record of the list starting from the first record until R_i is found. This search costs $\pi(i)$

units of time to perform.

Assuming that each record R_i is accessed with a possibly unknown probability s_i and that the accesses are made independently, the expected search cost for an ordering π of the list, $\text{ES_Cost}(\pi)$, is given by

$$\text{ES_Cost}(\pi) = \sum_{1 \leq i \leq n} s_i \pi(i) \quad (1.1)$$

To minimize the access cost, we can see that it is desirable that the records are ordered in the descending order of their access probabilities. We shall refer to this kind of list ordering as a perfect ordering or optimal ordering and refer to a file ordered in this way as a completely organized file. However the access probabilities are seldom known *a priori* in practice, and so the list can not be arranged in the *optimal ordering* in advance. In this situation we need an algorithm which dynamically rearranges the list and gradually transforms it to a less costly ordering.

A *self-organizing sequential search list* is a linear search list in which the order of the records may be altered each time a sequential search occurs so that after sufficiently many accesses, it tends to be in the optimal ordering with high probability where the most frequently accessed record is at the front of the list and the rest of the list is recursively ordered in the same manner. The most useful and common type of alteration is to move the accessed record forward one or more positions in the list. In this way more frequently accessed records move towards the front of the list so that fewer comparisons are needed on subsequent accesses.

Before studying the problem in any greater detail, let us look at some possible applications of self-organizing linear search algorithms. Bentley and McGeoch [5] describe three situations where the use of self-organizing linear search may be a better choice when space and efficiency are concerned.

The first situation arises when the list is small (say, has at most several dozen records). The simplicity of sequential search methods can make it faster than more complex algorithms. For example, to resolve the overflow problem in hashing, one of the strategies called 'chaining' uses linked lists which are rooted in appropriate home buckets to store the overflowed records. Comparing to the computational approach,

the data structure approach using a linked list performs better, in general, especially when the file is almost full [22].

The second case occurs when space is severely limited. Sophisticated data structures usually require more space than simpler ones. For example, to maintain a binary search tree structure we need to keep three pointers (one for the parent and one for each child) whereas a singly linked list only requires one pointer.

The third situation is when the performance of linear search is almost (but not quite) good enough, then self-organizing linear search list may give acceptable performance without adding more than a few lines of code. As an example, Bentley and McGeoch describe a case in which a very large-scale integration (VLSI) circuit-simulation program spent 5 minutes in a setup phase, most of which was taken up by linear searches through a symbol table. Since this simulator was run on line, the 5 minutes were annoying to those who were waiting for the simulation to begin. After incorporating a simple self-organizing search by adding about half a dozen lines of code, the setup time was reduced to about 30 seconds.

Another potential application described by Rivest [31] is the construction of list-processing systems such as LISP. Typically there is a large number of names (atoms), each associated with a set of attribute-value pairs. A separate "property list" is usually maintained for each name listing these attributes and associated values. These property lists are repeatedly searched for the various variable values and function definitions during execution, and there is often not enough storage available to consider using a complex algorithm. A self-organizing linear search heuristic could easily be inserted into such a system to reduce the running time of an average program by a significant percentage at no extra cost in terms of space.

Many other similar applications are imaginable such as the one mentioned by Hester and Hirschberg [16] where a list of identifiers is maintained by a compiler or interpreter. Identifiers are hashed into a list of buckets, each of which is an unordered linear list of identifier descriptions. Virtually every command interpreted by the system involves one or more accesses to the identifiers. Since most programs tend to access some identifiers more often than others, it would be preferable that

the more frequently accessed identifiers are nearer the front of the list in order to obtain a lower average search cost. However since the number of references to each identifier is unknown beforehand, self-organizing linear search algorithms can be used to make searching more efficient.

Self-organizing linear search algorithms have been in the literature for almost 30 years, and many schemes have been proposed during that time. Here we will give a brief history of them. (As in the literature concerning self-organizing data structures and adaptive algorithms, we shall use the terms *algorithms*, *heuristics*, *schemes*, and *rules* interchangeably).

1.1 Historical Overview

An intuitive scheme for reordering a list to a, hopefully, less costly ordering is to keep a counter of accesses for each record, and maintain the records in the descending order of their access frequencies. However, as Knuth remarks ([22] p398), this counter scheme is undesirable as it requires extra memory space which could perhaps be better used by employing nonsequential search techniques.

The first memory-free self-organizing scheme proposed by McCabe in 1965 [23] is the *move-to-front* rule. In this scheme, each time a record is accessed, it is moved to the front of the list, and all the records before the accessed record are shifted back one position. McCabe derived its close form expression for the expected search cost per access as a function of the access probabilities $s_1, s_2, \dots, s_n$. Rivest [31] proved that the move-to-front scheme never does more than twice the work done with the optimal ordering. Many other researchers have also studied the rule and various properties of it are available in the literature [13, 22, 35].

McCabe [23] also introduced a scheme called the *transposition* rule in the same paper where he first described the move-to-front rule. In this rule, the accessed 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. This was later proved in 1976 by Rivest [31] to have lower expected search cost per access than the move-to-front

rule, and he conjectured that the transposition rule is optimal. This conjecture was further strengthened by the result by Bitner [7] that for some special distributions the transposition rule is optimal over all rules. However Anderson *et al.* [1] found a counterexample to this conjecture by deriving a rule that is better than the transposition rule for a specific distribution. Bitner [6, 7] later showed that while the transposition rule is asymptotically more efficient, the move-to-front rule converges more quickly and proposed a hybrid of these two rules that attempts to incorporate the best features of both. This simple hybrid rule initially uses the move-to-front algorithm until its steady state is approached, and then switches to the transposition rule. The difficulty of deciding when to switch is a major problem. It is still unclear if the best time to switch can be determined for a fixed number of accesses.

Rivest [31] proposed a compromise between the relative extremes of move-to-front and transposition, namely the *move-ahead- k* heuristic where the accessed record is moved k -positions forward towards the front of the list unless it is in the first k positions, in which case it is moved to the front. This is a generalization of the transposition rule and the move-to-front rule as transposition is *move-ahead-1* and move-to-front is *move-ahead- n* . Rivest conjectured that the move-ahead- k rule is superior to the move-ahead- $(k + 1)$ rule; but this is unproven.

Tenenbaum and Nemes [35] suggested two generalizations of the move-to-front rule and the transposition rule, the *POS(k)* rule and the *SWITCH(k)* rule. The *POS(k)* rule moves the accessed record to position k of the list if it is in positions $k + 1$ to N , or it transposes it with its preceding record if it is in positions 2 to k . If it is the first in the list, it is left unchanged. Note that *POS(1)* is the move-to-front, whereas *POS($n - 1$)* is the transposition strategy. The *SWITCH(k)* has the same rule as *POS(k)* except that the uses of move-to-front and transposition are reversed, an accessed record found in positions 2 to k is moved to the front and others are transposed.

Notice that all the algorithms described above alter the list on a single access basis. (We shall call such an algorithm a *reordering algorithm* or *permutation algorithm* as the reorganization of the list is done by permuting the records.) McCabe

[23] considered reorganizing the list only once every k accesses to reduce the time spent reordering the list. We will refer it as the *move-every- k th-access* rule. It is to be used in conjunction with permutation algorithms. In this scheme, a counter is needed for each record to store the value of k . A record is moved forward (according to the permutation algorithm chosen) only if it has been accessed k times, not necessarily in a row, and then the counter for *that record* is reset. Bitner [7] also studied this rule (he called it *wait c and move* rule), and analyzed the performance of the rule when used with the move-to-front rule. Bitner also suggested a modification to it, the *wait c , move and clear* rule. After a record has been accessed c times, not necessarily in a row, it is moved forward and the counter for *every record* is reset. Bitner proved that as $c \rightarrow \infty$ the asymptotic search cost of this rule approaches the optimum.

Kan and Ross [17] and Gonnet *et al.* [12] proposed the *k -in-a-row* heuristics, where a record is moved forward only after it is accessed k times in a row. If the record is accessed even twice in a row, the chances are greater that it will have additional accesses in the near future. Gonnet *et al.* [12] proved that *$(k+1)$ -in-a-row* is superior to *k -in-a-row* and suggested a minor modification called the *k -in-a-batch* heuristic, where accesses are grouped into batches of size k , and a record will be moved if it is accessed k times in a batch. They proved that the batched- k rule is better than the *k -in-a-row* rule when in combination with either the move-to-front rule or the transposition rule. Note that these rules all require extra space for storing the values of k .

Almost all the schemes discussed in the literature are represented by Markov chains with $n!$ states, each being one of the possible list orderings. Consider any rule which moves an accessed record forward one or more positions (unless it is the first record of the list). Every access transforms the list from one ordering to another and there are total of $n!$ list orderings. Let the orderings of the list be the states of a system. Since it is possible to reach any state from any other by a sequence of one or more transformations, the transformations form an irreducible Markov chain with the state at any time being the list ordering at that time. Notice that the

Markov chains for all the schemes described above are ergodic. By virtue of this fact, the list can be in any of its $n!$ configurations. Taking the move-to-front rule as an example: a completely organized list can be thoroughly disorganized by a single access to the most infrequently accessed record. After this, it will take a long time for the list to become organized again.

Since we are assuming, so far, that the access probabilities $\{s_i\}$ are time invariant, the ergodic Markovian representation of the list organizing scheme is undesirable. Based only on the access probabilities, it is unreasonable to expect the list to converge to the perfect ordering if it was modified by any of the schemes listed above. In addition, even if it ever were to converge to the perfect ordering, it is unlikely that the list will remain at the perfect ordering; all the effort taken for an element to learn its right position can easily be destroyed by one bad access such as the access of the least frequently accessed record when applying the move-to-front rule.

As opposed to ergodic representations, Markovian behavior can also be absorbing [21, 32]. In an absorbing Markov chain, the chain converges to one of a (finite) set of absorbing barriers. Oommen [24] proposed an absorbing scheme called *stochastic move-to-front*, where each record has a probability attached with it to determine whether it is to be moved to the front of the list or not. Each time a record is accessed, its probability of moving is decreased by a multiplicative constant. As the probabilities of moving the records decrease to zero, the list will be less likely to change. Although the scheme may cause the list to converge to any of the $n!$ list orderings, Oommen showed that the multiplicative constant for changing the moving probabilities might be chosen to make the probability of convergence to the optimal static ordering as near unity as desired.

Oommen and Hansen [25] modified the *stochastic move-to-front* scheme and introduced two absorbing list organizing schemes, the *bounded memory stochastic move-to-front* scheme and the *stochastic move-to-rear* scheme. In both algorithms, the move operation is performed stochastically in such a way that ultimately no more move operations are performed. When this has occurred, they say that the scheme has converged. The first algorithm is essentially a move-to-front algorithm

with the exception that on the n th access, the accessed record is moved to the front of the list with a probability $f(n)$. This probability is systematically *decreased* every time a record is accessed. After the operation is executed for a sufficiently long period, each record tends to stay in place - instead of being moved to the front. This makes the Markovian representation of the scheme to be absorbing - the organization of the list gets "absorbed" to one of the $n!$ orderings. Oommen *et al.* show that this scheme is expedient, but is always worse than the deterministic move-to-front algorithm. The second scheme presented is far more powerful. Upon being accessed, the record is moved to the rear of the list with a probability q_i . This quantity q_i is progressively decremented every time the record is accessed. Again ultimately no more move operations are performed on the list. So the Markovian representation of the scheme is absorbing, and could converge to any one of its $n!$ orderings. However, Oommen *et al.* showed that the probability of convergence to the optimal ordering could be made as close to unity as desired.

Oommen *et al.* [26] later proposed two deterministic absorbing schemes, both of which perform move-to-rear operation. One is the *deterministic linear-space move-to-rear* scheme, the other is the *deterministic constant-space move-to-rear* scheme. The former moves the accessed record to the *rear* of the list if it has been accessed k times. The scheme is asymptotically optimal, the probability of being absorbed into the optimal ordering can be made as close to unity as desired. The second scheme moves the accessed record to the *rear* of the list if it has been accessed k consecutive times. It is proven to be expedient. Oommen and Ng [27] modified the latter scheme, and showed that the scheme could be viewed as an *move-to-front* strategy in which the "front" of the list is dynamically updated. The modified scheme is also asymptotically optimal.

1.2 Salient Contributions of the Thesis

The central property that motivated this thesis is the time reversibility of adaptive data structures. The literature describes only one - the transposition rule, and the

proof of it being time reversible is a non-constructive one. Indeed, as a complete constructive proof of this was not available in the literature, our first endeavour was to give a formal constructive proof of this property. This is the first original contribution of this thesis, and is currently being compiled as a potential publication [28].

Notice that there is a common drawback to all the algorithms in the literature, that is, they all fail to consider the *size* of the list when reorganizing it. All the permutation algorithms either move the accessed record to the front of the list, move it one step closer to the front, or move it a fixed number of positions forward regardless of the size of the list. The two extremes are the move-to-front rule and the transposition rule. This thesis presents two new memory-free self-organizing sequential search algorithms, the *swap-with-parent* heuristic and the *move-to-parent* heuristic. Both take into account the size of the list when reorganizing it. The swap-with-parent heuristic exchanges the accessed record with its parent (considering the list as a heap structure with no ordering constraints between parent nodes and their children), and leaves the rest of the records untouched. Alternatively, instead of swapping the accessed record with its parent, the move-to-parent heuristic moves the accessed record to its parent's position and shifts the parent and all the records between the accessed record and its parent back one position. The interesting feature of these schemes is that it only takes $O(\lg n)$ accesses for the most frequently accessed element to reach the front of the list, (compared to $O(n)$ accesses for the transposition rule.) No reported scheme possesses such a property. This is the second contribution of this thesis, and is also currently being compiled as a potential publication [29].

It is shown that under the swap-with-parent rule, the Markov chain representing the scheme is *time reversible*. This property provides a useful tool for us to investigate the algorithm and greatly simplifies its analysis. Using the concept of tree transforming processes, we also demonstrate the existence of a class of time reversible Markov chains, which opens a new research area in the use and the analysis of "implicit" tree structures in adaptive sequential search algorithms. Using this, we

have introduced two generalized classes of self-organizing sequential search schemes, the *time reversible* swap-with-parent-in-an-ss_tree heuristic and the move-to-parent-in-an-ss_tree heuristic. This is the third contribution of this thesis, and it, too, is presently being compiled as a potential publication [30].

1.3 Thesis Outline

This thesis summarizes the previous work done on self-organizing sequential search algorithms and presents two new algorithms and their generalizations. Chapter 2 gives detailed analysis and comparisons of the known algorithms. Chapter 3 contains the description and the analysis of the two new schemes, the *swap-with-parent* scheme and the *move-to-parent* scheme. Chapter 4 discusses a condition for an arbitrary Markov chain representing a list reorganizing scheme to be time reversible and introduces two generalized self-organizing sequential search algorithms. In Chapter 5, the concluding chapter, we give some indication of the further possible work which could extend the results contained in this thesis.

The main prerequisite for the understanding of this thesis is a knowledge of basic probability theory, finite Markov chains and elementary data structures. These can be found in [3, 10, 18, 20, 32].

Chapter 2

Previous Algorithms Review and Analysis

This chapter provides a literature survey of the common algorithms and their analysis. Our goal is to find a good list reordering algorithm. It is necessary to define what is meant by “good” and what our measurements are. Therefore Section 1 of this chapter gives definitions and analysis techniques that apply to self-organizing sequential search algorithms.

2.1 Definitions and Analysis Techniques

In self-organizing linear search algorithms, the *accessed record* is the record currently being searched for, and the *probed record* is the record currently being considered during the search.

Several assumptions apply to the problem of self-organizing linear search algorithms as studied in this thesis.

1. The probability of access to any record is unknown prior to its access.
2. The access probabilities follow some unknown distribution or general rule.
3. Each access is made independently from previous accesses (no locality), and the access probabilities do not vary with respect to time, and are hence stationary.

4. All the records searched for are in the list. That is, the search will not access records that are not found in the list.
5. Each record will be accessed at least once. (The access probability of the record may still be close to zero when the total number of accesses is large.)

When choosing an algorithm, it is natural to ask questions such as: What is the *cost* of applying the algorithm? How *efficient* is the algorithm? In the literature concerning self-organizing sequential search algorithms, we measure efficiency by how fast a list *converges* to its optimal static ordering after applying the reordering algorithm, and measure cost by the *average* number of probes required to search for a record after the list has converged, i.e., it has approached a *steady state*. Subsequent to this further reordering on the list is not expected to increase or decrease the average search time significantly. Another measure of cost is by *amortization* - averaging the running time of an algorithm over a worst-case sequence of operations [5, 33]. However this will not be the focus of this thesis. We shall give some discussion on the issue in the last chapter. A little research is available where assumptions are relaxed - accesses are made dependently, i.e., the access probabilities are non-stationary. But it is not discussed in details in this thesis.

2.1.1 Cost

Let $\{R_1, R_2, \dots, R_n\}$ be a set of records that are in an arbitrary order π , so that R_i is in position $\pi(i)$ for $1 \leq i \leq n$. At each unit of time a request is made for one of these records, R_i , independent of the previous accesses, with probability s_i . After being accessed the record is then moved forward by a certain distance depending on the algorithm applied. We call such an algorithm a permutation algorithm or a reordering algorithm. We are interested in determining the *expected position* of the record R_i accessed *after this process has been in operation for a long time*.

To compute the expected position of the record accessed we start by conditioning

on which record is accessed. This yields

$$\begin{aligned} E[\text{Position of record accessed}] &= \sum_{1 \leq i \leq n} (s_i E[\text{Position} \mid R_i \text{ is accessed}]) \\ &= \sum_{1 \leq i \leq n} (s_i E[\text{Position of } R_i]). \end{aligned} \quad (2.1)$$

Note that,

$$\text{Position of } R_i = 1 + \sum_{j \neq i} I_j$$

where

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

and so the expected position of R_i ,

$$\begin{aligned} E[\text{Position of } R_i] &= 1 + \sum_{j \neq i} E[I_j] \\ &= 1 + \sum_{j \neq i} P\{R_j \text{ precedes } R_i\}. \end{aligned} \quad (2.2)$$

Therefore from Equation (2.1) and (2.2) we have

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

Equation (2.3) gives us the *expected search cost* per access in terms of the number of probes required to find a record after this process has been in operation for a long time. As we shall see in the next section, the expected search cost approaches the asymptotic cost as the number of accesses increases, and eventually reaches the asymptotic cost. In the general case, the asymptotic search cost of an algorithm

is the average search cost over all possible list orderings and search sequences (access distributions). Therefore, from here on, we shall use the terms *average search cost*, *average search length*, *expected search cost* and *asymptotic search cost* interchangeably. Note that the cost given by Equation (2.3) does not include the cost of reordering the list after each access is made. Thus for a given rule A , the total cost of applying A is the sum of the search cost and the reordering cost.

$$\text{Total_Cost}(A) = \text{ES_Cost}(A) + \text{Reordering_Cost}(A). \quad (2.4)$$

However, as the list is generally implemented as a linked list, the reorganization of the list is simple and can be done quite inexpensively - normally in a constant time. The cost of reordering is thus small compared to the time spent accessing the list. Therefore the major concern when analyzing the cost of self-organizing sequential search algorithms is the *expected search cost* (short for ES_Cost). Given two self-organizing algorithms, A and B , $\text{ES_Cost}(A) < \text{ES_Cost}(B)$ implies that rule A is eventually “better” than rule B ; rule A requires fewer comparisons over a large number of retrievals. Using Equation (2.3), we have the following theorem:

Theorem 2.1 *Suppose that we are processing a list of records $\{R_1, R_2, \dots, R_n\}$ with access probabilities $\{s_1, s_2, \dots, s_n\}$ respectively, and investigating two self-organizing linear search algorithms A and B . To show that rule A is better than rule B , it is sufficient to show that the asymptotic probability that R_j precedes R_i under rule A is greater than that under rule B given that $s_j > s_i$ for all $j \neq i$,*

Proof:

We are actually required to prove that

$$P\{R_j \text{ precedes } R_i\}_{\text{rule}_A} > P\{R_j \text{ precedes } R_i\}_{\text{rule}_B}$$

given that $s_j > s_i$ for $1 \leq i, j \leq n$ and $j \neq i$.

By Equation (2.3), we have

$$\begin{aligned}
 & \text{Expected search cost} \\
 &= 1 + \sum_{1 \leq i \leq n} \sum_{j \neq i} (s_i P\{R_j \text{ precedes } R_i\}) \\
 &= 1 + \sum_{1 \leq i < j \leq n} (s_i P\{R_j \text{ precedes } R_i\} + s_j P\{R_i \text{ precedes } R_j\}) \\
 &= 1 + \sum_{1 \leq i < j \leq n} (s_i P\{R_j \text{ precedes } R_i\} + s_j (1 - P\{R_j \text{ precedes } R_i\})) \\
 &= 1 + \sum_{1 \leq i < j \leq n} ((s_i - s_j) P\{R_j \text{ precedes } R_i\}) + \sum_{1 \leq i < j \leq n} s_j. \tag{2.5}
 \end{aligned}$$

Thus, to minimize the expected cost, we would want to make $P\{R_j \text{ precedes } R_i\}$ as large as possible when $s_j > s_i$ and as small as possible when $s_i > s_j$. Therefore, to show that rule A is better than rule B , it suffices to show that

$$P\{R_j \text{ precedes } R_i\}_{\text{rule}_A} > P\{R_j \text{ precedes } R_i\}_{\text{rule}_B} \quad \text{when } s_j > s_i.$$

This completes the proof. $\square$

It is usually difficult to derive a closed form expression for $P\{R_j \text{ precedes } R_i\}$ for a given algorithm. However it is still possible, and actually quite straightforward, to calculate the average search cost of an arbitrary set of list orderings π for any given set of probabilities s_i by carrying out a series of tedious steps. Recall that in the previous chapter, we mentioned that a self-organizing scheme can be represented by a Markov chain with each state as one of the $n!$ possible orderings of the list. The transition probabilities M_{st} for the Markov chain are determined by the access probabilities s_i of the records and the reordering rule used; each M_{st} will either be zero or one of the s_i 's. The stationary probabilities $P\{\pi\}$ of each state π are the elements of the eigenvector of the matrix $M = \{M_{st} \mid 1 \leq s \leq n! \text{ and } 1 \leq t \leq n!\}$ corresponding to the eigenvalue unity [20]. The cost of a self-organizing rule A is then easily calculated from the formula

Figure 2.1: State diagram with 3 elements per state using the MTF rule.

$$\begin{aligned}
 \text{ES_Cost}(A) &= \sum_{\text{all } \pi} (P\{\pi\}_A \text{ES_Cost}(\pi)) \\
 &= \sum_{\text{all } \pi} (P\{\pi\}_A \sum_{1 \leq i \leq n} s_i \pi(i)), \quad (2.6)
 \end{aligned}$$

where $\text{ES_Cost}(\pi) = \sum_{i=1}^n s_i \pi(i)$ is given by Equation (1.1).

As an example, consider a list of three records $\{a, b, c\}$ with respective access probabilities of $\{s_a, s_b, s_c\}$ operated on by the *move-to-front* (MTF) rule. There are total of 6 possible list orderings. $\pi_1 = (a, b, c)$, $\pi_2 = (b, a, c)$, $\pi_3 = (c, b, a)$, $\pi_4 = (a, c, b)$, $\pi_5 = (c, a, b)$ and $\pi_6 = (b, c, a)$. If the system is in state $\pi_1 = (a, b, c)$ and record b is accessed, it is moved to the front of the list. Thus the system is transformed into state $\pi_2 = (b, a, c)$. Figure 2.1 shows the state diagram for the underlying Markov chain.

Since record b is accessed with probability s_b , and its access transforms the system from state $\pi_1 = (a, b, c)$ to state $\pi_2 = (b, a, c)$, the transition probability M_{12} of the Markov chain is s_b . Similarly, we can get the values of M_{st} for all s and t .

The transition matrix M for the system is then

$$M = \begin{array}{c} \begin{matrix} (abc) & (bac) & (cba) & (acb) & (cab) & (bca) \end{matrix} \text{ state} \\ \begin{bmatrix} s_a & s_b & 0 & 0 & s_c & 0 \\ s_a & s_b & s_c & 0 & 0 & 0 \\ 0 & 0 & s_c & s_a & 0 & s_b \\ 0 & s_b & 0 & s_a & s_c & 0 \\ 0 & 0 & 0 & s_a & s_c & s_b \\ s_a & 0 & s_c & 0 & 0 & s_b \end{bmatrix} \end{array} \begin{matrix} (abc) \\ (bac) \\ (cba) \\ (acb) \\ (cab) \\ (bca) \end{matrix}$$

The eigenvector of M corresponding to the eigenvalue unity gives the stationary probabilities $P\{\pi_s\}$ of the states π_s ($1 \leq s \leq n!$) of the system.

Let $\Pi = (P\{\pi_1\} P\{\pi_2\} P\{\pi_3\} P\{\pi_4\} P\{\pi_5\} P\{\pi_6\})^T$. Then we have

$$M^T * \Pi = \Pi \quad (2.7)$$

that is,

$$\begin{bmatrix} s_a & s_b & 0 & 0 & s_c & 0 \\ s_a & s_b & s_c & 0 & 0 & 0 \\ 0 & 0 & s_c & s_a & 0 & s_b \\ 0 & s_b & 0 & s_a & s_c & 0 \\ 0 & 0 & 0 & s_a & s_c & s_b \\ s_a & 0 & s_c & 0 & 0 & s_b \end{bmatrix}^T * \begin{bmatrix} P\{\pi_1\} \\ P\{\pi_2\} \\ P\{\pi_3\} \\ P\{\pi_4\} \\ P\{\pi_5\} \\ P\{\pi_6\} \end{bmatrix} = \begin{bmatrix} P\{\pi_1\} \\ P\{\pi_2\} \\ P\{\pi_3\} \\ P\{\pi_4\} \\ P\{\pi_5\} \\ P\{\pi_6\} \end{bmatrix}.$$

Obviously, the stationary probabilities $P\{\pi_s\}$ are difficult to compute symbolically as explicit functions of the variables s_a , s_b and s_c . However, the eigenvector is easily calculated for any particular values of s_a , s_b and s_c . For example, if we take $s_a = 0.6$, $s_b = 0.3$, $s_c = 0.1$, we obtain

$$\begin{bmatrix} P\{\pi_1\} \\ P\{\pi_2\} \\ P\{\pi_3\} \\ P\{\pi_4\} \\ P\{\pi_5\} \\ P\{\pi_6\} \end{bmatrix} = \begin{bmatrix} P\{abc\} \\ P\{bac\} \\ P\{cba\} \\ P\{acb\} \\ P\{cab\} \\ P\{bca\} \end{bmatrix} = \begin{bmatrix} 0.54 \\ 0.16 \\ 0.02 \\ 0.16 \\ 0.08 \\ 0.04 \end{bmatrix}.$$

By Equation (2.6), we see that the expected (average) search cost for this system is 1.72, i.e., the average number of probes required to find a record in the list $\{a, b, c\}$ by using the move-to-front rule is 1.72.

Theoretically, we can calculate the average search cost for any given set of access probabilities, permutation rule and access sequence. However, it is not always possible to do so. Imagine the case where there are 100 records in the list. In this case we are going to have a Markovian system which has $100!$ states. Solving a set of linear equations with $100!$ variables given by Equation (2.7) is computationally impractical. Therefore, finding the absolute measurement for a self-organizing algorithm can be very difficult. Comparisons of costs between two algorithms according to Theorem 2.1 and Equation (2.3) may also not be possible. In this case relative measurements are needed.

One of the measurements available is to compare the cost of the permutation algorithm with the cost of the *optimal static ordering* of the list, in which the records are initially ordered by their static access probabilities and left in that order throughout the access sequence. Therefore under the optimal ordering, we have $s_i \geq s_{i+1}$ for $1 \leq i \leq n$, and thus the average search cost on the optimal ordering (OPT) is

$$\text{Cost (OPT)} = \sum_{1 \leq i \leq n} s_i * i. \quad (2.8)$$

The worse-case for this ordering occurs when the access probabilities are equal, i.e., $s_i = 1/n$ for all i , in which case the average search cost is

$$\sum_{1 \leq i \leq n} s_i * i = \sum_{1 \leq i \leq n} \left(\frac{1}{n} * i \right) = \frac{1}{n} \sum_{1 \leq i \leq n} i = \frac{n+1}{2}.$$

That is, under the optimal ordering, it takes no more than $(n+1)/2$ probes to locate a record (A best-case cost is 1 probe [16]).

When the access probabilities follow Zipf's distribution, i.e., $s_i = 1/(i H_n)$ where $H_n = \sum_{k=1}^n (1/k)$ is the n th harmonic number, the average search cost for the optimal ordering is

$$\sum_{1 \leq i \leq n} \left(\frac{1}{i H_n} i \right) = \frac{1}{H_n} \sum_{1 \leq i \leq n} 1 = n H_n^{-1},$$

which, as Knuth shows [22], is $O(\ln n)$ times faster than with a random ordering.

Notice that using the optimal static ordering is not considered self-organizing by our definitions as it uses knowledge about the access probabilities which are assumed to be unavailable before the searches begin. However it provides a relative measurement for self-organizing linear search algorithms.

Another thing should be noticed is that the optimal ordering is only optimal under the assumption that the accesses are made independently (See Section 2.1). If *locality* is allowed for the accesses, self-organizing algorithms may have lower search cost than the optimal static ordering. (The *locality* of access sequences means that the subsequences may have relative access frequencies that are drastically different from the overall access frequencies. (See Chapter 5 for details.) We will address this problem again in the next section when we discuss convergence issues.

2.1.2 Convergence

Convergence is one of the two major concerns when analyzing a self-organizing sequential search algorithm. Till now we have discussed the *asymptotic cost* of the algorithms, the expected search cost per access in terms of the number of probes required to find a record after the process has been in operation for a long time (i.e., after large enough number of accesses). A natural question arises at this point: how many accesses can be considered to be “large enough”? Or how quickly do the reordering rules approach their asymptotic cost?

Ideally the result of applying a self-organizing sequential search algorithm is to cause the list to converge to the perfect ordering where the most frequently accessed record is at the front of the list and the rest of the list is recursively ordered in the same manner. Based only on the access probabilities, however, it is unreasonable to expect this to happen. It is more unlikely that the list will remain in the perfect ordering even after it was converged to it. For example, under the move-to-front rule, a completely organized list can be thoroughly disorganized by a *single* access to the record with the least access probability, as we have pointed out in Chapter 1. Therefore the algorithms are expected to approach a *steady state*, where many

further organizing on the list are not expected to increase or decrease the search cost significantly. The *steady state* is not a particular list ordering but a set of list orderings that all have expected search costs close to the asymptotic cost of the algorithm. *Convergence* thus refers to the amount of time or the number of accesses required to approach the *steady state* [16].

It is hard to measure the convergence of an algorithm, as it is not well defined as to how close the expected cost of a list ordering must be to the asymptotic cost of the algorithm in order to be considered part of the steady state. This is an important problem as a rule may have very low asymptotic cost, and yet converge so slowly that it is not practically useful. Taking the move-to-front rule and the transposition rule as examples, the move-to-front rule has higher search cost than that of the transposition rule [31] but converges much more quickly than the transposition rule. Thus the move-to-front rule will, generally speaking, demonstrate superior performance after a relatively small number of accesses have been made. The reason for this is intuitive and clear. In the initial random ordering, many records with high access probability may be far down in the list. Using the move-to-front rule, these records are shifted quickly to the front of the list. In contrast, the transposition rule only moves one position forward for each access and so the cost decreases slowly.

Bitner [7] proposed a measure of convergence that he called *overwork*, which is defined as the area between the cost curve and its asymptote (see figure 2.2). The horizontal axis is the number of accesses performed on the list with range from 0 to ∞ . The vertical axis is the cost of the permutation algorithm. The expected search cost (average search cost) is a nonlinear function of the number of accesses (see the curve on figure 2.2). The asymptotic search cost is a constant function. When the number of accesses is small, the expected cost curve is expected to be high as the list is still close to the random ordering so that it is unlikely that the records with higher access probabilities are already near the front of the list. The expected cost is expected to approach the asymptotic cost as the number of accesses increases. The *overwork* (*OV*) is defined as the area between the expected cost curve and the

Figure 2.2: Bitner's overwork measure for convergence (Adapted from Bitner[79]).

asymptotic cost line. Note that the "steeper" the expected cost curve is, the smaller the overwork will be. An algorithm is said to converge faster than another algorithm if its overwork area is smaller.

Bitner [7] also suggested a modification of his overwork measure that allows us to compare the convergence of any two algorithms by using their expected search cost as the upper and lower curves of the graph respectively, and examining the area between them. Hester and Hirschberg pointed out that this can be further generalized to compare the convergence of several algorithms to one another by choosing an algorithm (not necessarily one of those to be compared) as a metric and comparing the areas resulting from using its cost curve as a lower (or upper) bound with the cost curves of each of the other algorithms. They suggest that in the general case of using a single algorithm as a ruler for comparing several algorithms, the *transposition* rule is a good choice to be chosen as the lower bound of the cost curves because of its low asymptotic cost and stability.

In the next following sections, we shall present the major self-organizing sequential search algorithms that are known in the literature with the absolute analysis

when the analysis techniques introduced in this section are applicable.

2.2 The Move-to-Front Heuristic

The *move-to-front* (MTP) heuristic is by far the most extensively analyzed memory-free self-organizing sequential search algorithm in the literature. It has been studied by McCabe [23], Hendricks [13, 14], Knuth [22], Rivest [31], Bitner [7] and Gonnet *et al.* [11, 12]. Using this heuristic, whenever a record R_i is found in position $\pi(i)$, the list is rearranged by moving R_i to the front of the list and moving the records in positions $1, \dots, \pi(i) - 1$ each down one position. Thus the records which are accessed frequently will tend to stay near the front of the list, while records infrequently accessed will drift towards the end of the list. It is unlikely that we can achieve a stable ordering using the move-to-front rule because a completely organized list can be totally disorganized by a single request to the most infrequently accessed record. However we can expect that near-optimal orderings will occur with high probability.

It is intuitive that the move-to-front algorithm will converge quickly, but has a large asymptotic search cost. Every time a record with low access probability is accessed, it is moved all the way to the front, which increases the costs of future accesses to many other records. We shall look at the performance of the move-to-front heuristic from the three aspects discussed in the previous section of this chapter: the average (asymptotic) search cost per access, comparisons with the cost of the optimal ordering and the rate of convergence.

2.2.1 The Average Search Cost

The average search cost per access for the move-to-front heuristic can be easily calculated using Equation (2.3). The following theorem gives the asymptotic probability that record R_j is before record R_i in the list.

Theorem 2.2 *The asymptotic probability that record R_j is before record R_i in the list under the move-to-front heuristic is*

$$P \{R_j \text{ precedes } R_i\}_{MTF} = \frac{s_j}{s_i + s_j}. \quad (2.9)$$

Proof:

Note that it will be true at any time that record R_j precedes record R_i if the most recent request for R_j has occurred since the most recent request for R_i . In this case, there exists a unique k such that the preceding k requests have been *one* request for R_j followed by $(k - 1)$ requests for records other than R_j or R_i . Thus

$$P \{R_j \text{ precedes } R_i\}_{MTF} = s_j * \sum_{1 \leq k < \infty} (1 - s_i - s_j)^{(k-1)} = \frac{s_j}{s_i + s_j}.$$

The proof completes. $\square$

Corollary 2.1 *The move-to-front heuristic is expedient. That is,*

$$P \{R_j \text{ precedes } R_i\} > \frac{1}{2} \quad \text{if } s_j > s_i.$$

The proof of the above corollary is obvious from Theorem 2.2.

Theorem 2.3 *The asymptotic search cost (average search cost) for the move-to-front heuristic is*

$$1 + 2 \sum_{1 \leq i < j \leq n} \sum_{1 \leq k < \infty} \frac{s_i s_j}{s_i + s_j}. \quad (2.10)$$

Proof:

Applying Equation (2.3) and (2.9) we have

Average search cost for MTF

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

which is the result shown in Equation (2.10). $\square$

This result has been obtained by several analysis [7, 14, 22, 23, 31].

2.2.2 Comparisons with the Cost of the Optimal Ordering

Let us consider how the move-to-front scheme compares with the optimal ordering. Under the optimal ordering, the records are ordered in descending order of their access probabilities and left in that order throughout the access sequence. Rivest [31] proves that the worst case performance of the move-to-front scheme is no more than twice that of the optimal static ordering. Gonnet *et al.* [12] show that for a particular distribution the average case ratio of the cost of move-to-front to the cost of the optimal static ordering is $\pi/2$. Chung *et al.* [9] show that this is an upper bound on the average case for all distributions and it is the best possible.

Theorem 2.4 *The average search cost of the move-to-front rule is at most twice the cost of the optimal ordering.*

Proof:

We have seen from Equation (2.8) that the average search cost under the optimal ordering (OPT) is $\sum_{i=1}^n s_i * i$. The ratio of (2.10) and (2.8) is then

$$\begin{aligned} \frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} &= \frac{1 + 2 \sum \sum_{1 \leq i < j \leq n} ((s_i s_j) / (s_i + s_j))}{\sum_{1 \leq i \leq n} (s_i * i)} \\ &= \frac{1 + 2 \sum_{1 \leq i \leq n} (s_i * \sum_{1 \leq j < i} (s_j / (s_i + s_j)))}{1 + \sum_{1 \leq i \leq n} (s_i * (i - 1))}. \end{aligned}$$

Let

$$x = \sum_{1 \leq i \leq n} (s_i * (i - 1)).$$

Now since $s_i \geq 0$, we have

$$\sum_{1 \leq j < i} \frac{s_j}{s_i + s_j} \leq \sum_{1 \leq j < i} \frac{s_j}{s_j} = \sum_{1 \leq j < i} 1 = i - 1 \quad \text{for all } 1 \leq i \leq n.$$

Hence

$$\sum_{1 \leq i \leq n} s_i * \sum_{1 \leq j < i} (s_j / (s_i + s_j)) \leq \sum_{1 \leq i \leq n} (s_i * (i - 1)),$$

that is

$$\sum_{1 \leq i \leq n} s_i * \sum_{1 \leq j < i} (s_j / (s_i + s_j)) \leq x.$$

Therefore we have

$$\frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} \leq \frac{1+2x}{1+x} \leq \frac{2x}{1+x} = 2 \left(1 - \frac{1}{1+x}\right) \leq 2. \quad (2.11)$$

This completes the proof. $\square$

Thus the move-to-front rule does at most twice the work done for the optimal ordering. Rivest [31] pointed out an interesting question which was to determine the least constant c , an upper bound to (2.11); the value 2 we have in (2.11) may not be the best possible bound.

Knuth [22] showed that if the access probabilities of the records obey Zipf's law, then the asymptotic cost of the move-to-front heuristic is approximately 1.386 times the cost of the optimal ordering. The reasoning for this is as follows.

Under Zipf's law:

$$\begin{aligned} s_i &= 1 / (i H_n) && \text{for } 1 \leq i \leq n \\ \text{Cost (OPT)} &= n / H_n && (\text{see section 2.1}) \\ \text{Cost (MTF)} &= 2 \ln 2 \frac{n}{H_n} - \frac{1}{2} + o(1) \end{aligned}$$

The ratio between the two is then

$$\frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} \leq 2 \ln 2 = 1.38629 \dots$$

Gonnet *et al.* [12] gave the results of such an analysis for several other interesting distributions shown below.

Under Lotka's law:

$$\begin{aligned} s_i &= 1 / (i^2 H_n^{(2)}) && \text{for } 1 \leq i \leq n \\ \text{Cost (OPT)} &= H_n / H_n^{(2)} \\ \text{Cost (MTF)} &= \frac{3}{\pi} \ln n - 0.00206339 \dots + O((\ln n) / n) \end{aligned}$$

The ratio between the two is then

$$\frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} \leq \frac{\pi}{2} = 1.57080 \dots$$

Under the Exponential distribution:

$$\begin{aligned} s_i &= (1-a) a^{i-1} \quad \text{for } 1 \leq i \leq n, \quad \text{where } 0 < a \leq 1 \\ \text{Cost (OPT)} &= 1/(1-a) \\ \text{Cost (MTF)} &= 1 + \sum_{j=1}^{\infty} \frac{a^j}{1+a^j} = O(\ln^3 a) - \frac{2 \ln 2}{\ln a} - \frac{1}{2} - \frac{\ln a}{24} \end{aligned}$$

The ratio between the two is then

$$\frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} \leq 2 \ln 2 = 1.38629 \dots$$

Under Wedge distribution:

$$\begin{aligned} s_i &= \frac{2(n+1-i)}{n^2+n} \quad \text{for } 1 \leq i \leq n \\ \text{Cost (OPT)} &= (n+2)/3 \\ \text{Cost (MTF)} &= \frac{4(1-\ln 2)}{3} * n - H_n + \frac{5(1-\ln 2)}{3} + O(1) \end{aligned}$$

The ratio between the two is then

$$\frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} \leq 4(1-\ln 2) = 1.22741 \dots$$

Chung *et al.* [9] proved that the average search cost required for the move-to-front heuristic is no more than $\pi/2$ times the cost of the optimal ordering by using Hilbert's inequality and showed that this is a tight bound. We state the result in the following theorem without the proof as it involves complex analysis and the use of Hilbert's inequalities.

Theorem 2.5 (Adapted from Chung et al. [9]) *For any probability distribution, we have*

$$\frac{\text{Cost (MTF)}}{\text{Cost (OPT)}} \leq \frac{\pi}{2}. \quad (2.12)$$

□

2.2.3 Rate of Convergence

We shall give the analysis of the *overwork* measure of convergence for the move-to-front rule done by Bitner [7]. To begin the analysis of the overwork, we need to determine the expected search cost of the move-to-front rule as a function of the number of accesses as given by the following theorem adapted from Bitner [7] - the details of the proof are omitted in the interest of brevity.

Theorem 2.6 *The expected search cost of the move-to-front rule after t requests is*

$$1 + 2 \sum_{1 \leq i < j \leq n} \sum_{s_i + s_j} \frac{s_i s_j}{s_i + s_j} + \sum_{1 \leq i < j \leq n} \sum_{2(s_i + s_j)} \frac{(s_i - s_j)^2}{2(s_i + s_j)} (1 - s_i - s_j)^t.$$

□

Theorem 2.7 *The overwork for the move-to-front rule, OV_{MTF} , is*

$$OV_{MTF} = \sum_{1 \leq i < j \leq n} \sum_{2(s_i + s_j)^2} \frac{(s_i - s_j)^2}{2(s_i + s_j)^2}.$$

Proof:

Recall that the asymptotic search cost of the move-to-front rule given by Theorem 2.3 is

$$1 + 2 \sum_{1 \leq i < j \leq n} \sum_{s_i + s_j} \frac{s_i s_j}{s_i + s_j}.$$

The difference between the cost after t requests and the asymptotic cost is given by the last term in Theorem 2.6. Summing this over $0 \leq t \leq \infty$ yields

$$\sum_{t=0}^{\infty} (1 - s_i - s_j)^t = \frac{1}{s_i + s_j}$$

which, in turn, gives the overwork. □

Bitner then showed that this is less than $n(n-1)/4$ for $n > 2$.

Corollary 2.2 $OV_{MTF} < \frac{1}{4} n(n-1)$ for every probability distribution with $n > 2$.

Proof:

Since obviously

$$\frac{(s_i - s_j)^2}{2(s_i + s_j)^2} \leq \frac{1}{2}$$

and the strict equality does not hold for every distinct pair i and j , we have

$$OV_{MTF} < \sum_{1 \leq i < j \leq n} \frac{1}{2} = \frac{1}{2} \sum_{1 \leq i \leq n} \sum_{i < j \leq n} 1 = \frac{1}{2} \sum_{1 \leq i \leq n} (n-i) = \frac{1}{4} n(n-1).$$

Hence the result. $\square$

Bitner also gave the overwork for the Zipf's distribution.

Theorem 2.8 If the access probabilities obey Zipf's law, i.e., $s_i = 1/(iH_n)$, $1 \leq i \leq n$ and $H_n = \sum_{i=1}^n (1/i)$, then

$$OV_{MTF} = \frac{5n^2}{12} - (n^2 + n + \frac{1}{6})(H_{2n} - H_n) + \frac{n(n+1)(2n+1)}{3}(H_{2n}^{(2)} - H_n^{(2)})$$

where $H_n^{(2)} = \sum_{i=1}^n 1/i^2$. For large n ,

$$OV_{MTF} \approx (\frac{3}{4} - \ln 2)n^2 \approx 0.57n^2.$$

$\square$

See [6] for the proof of the above theorem.

The overwork of a rule tells us how fast the rule converges to its asymptotic cost. The smaller the overwork is, the faster the rate of convergence is. Table 2.1 (adapted from Bitner [7]) shows the overwork of the move-to-front rule under Zipf's distribution for the number of records in the list varying from 3 to 20.

More analysis of the move-to-front rule in terms of comparisons with other algorithms will be given in the following sections when applicable.

# of records	Move-to-front rule	# of records	Move-to-front rule
3	0.2006	12	6.3473
4	0.4463	13	7.5882
5	0.7978	14	8.9420
6	1.2567	15	10.4087
7	1.8272	16	11.9884
8	2.5076	17	13.6812
9	3.2994	18	15.4871
10	4.2031	19	17.4063
11	5.2189	20	19.4387

Table 2.1: The Overwork of MTF for Zipf's Law (Adapted from Bitner [79]).

2.3 The Transposition Heuristic

One obvious disadvantage of the move-to-front rule is its instability. One “bad” access can considerably increase the expected search cost for many subsequent accesses. The *transposition* (TR) heuristic avoids this potential problem by being far more conservative in its record moves. Under the *transposition* heuristic, the accessed record, if not at the front of the list, is moved only one closer to the front by exchanging it with the record just ahead of it, and thus, a record can only be advanced to the front of the list if it is accessed frequently.

The slower record movement gives *transposition* slower convergence, but its stability tends to keep its steady state closer to the optimal static ordering. Another interesting property it has is that the Markov chain representing the scheme is *time reversible*. Rivest [31] proved that the transposition rule has lower (or equal) asymptotic search cost than the move-to-front rule for any probability distribution and conjectured that the transposition rule is optimal among all the permutation rules for every distribution. However Anderson *et al.* [1] found a counterexample

to this conjecture by deriving a rule that is better than the transposition rule for a specific distribution. Rivest [31] also gave an expression for the average search cost of the transposition heuristic.

2.3.1 The Average Search Cost

There is no simple formula for the asymptotic search cost of the transposition rule due to the complexity of computing the asymptotic probability that record R_j precedes record R_i , ($P\{R_j \text{ precedes } R_i\}$), when applying Formula (2.3). However we can still calculate the average search cost for a given set of probabilities s_i by Equation (2.6) from Section 2.1. Recall that a self-organizing scheme can be represented by a finite Markov chain where each state is one of the $n!$ possible orderings of the list. The stationary probabilities $P\{\pi\}$ of each state π are the elements of the eigenvector of the matrix $M = \{M_{st} \mid 1 \leq s \leq n! \text{ and } 1 \leq t \leq n!\}$ corresponding to the eigenvalue unity.

Let us look at the example given in Section 2.1 again where we have a list of three records $\{a, b, c\}$ with respective access probabilities of $\{s_a = 0.6, s_b = 0.3, s_c = 0.1\}$. This time we will apply the transposition rule. Figure 2.3 shows the state diagram for the Markov chain resulting from the transposition rule.

The transition matrix M for this system is then

$$\begin{array}{c}
 (abc)(bac)(bca)(cba)(cab)(acb) \quad \text{state} \\
 \\
 M = \begin{array}{c} \left[\begin{array}{cccccc} s_a & s_a & 0 & 0 & s_c & s_b \\ s_b & s_b & s_a & 0 & 0 & 0 \\ 0 & s_c & s_b & s_b & 0 & 0 \\ 0 & 0 & s_c & s_c & s_b & 0 \\ 0 & 0 & 0 & s_a & s_c & s_c \\ s_c & 0 & 0 & 0 & s_a & s_a \end{array} \right] \end{array} \begin{array}{l} (abc) \\ (bac) \\ (bca) \\ (cba) \\ (cab) \\ (acb) \end{array}
 \end{array}$$

The eigenvector of M corresponding to the eigenvalue unity gives the stationary

Figure 2.3: State diagram with three elements per state using the TR rule.

probabilities $P\{\pi_s\}$ of the states π_s , ($1 \leq s \leq n!$) of the system. That is,

$$\begin{bmatrix} s_a & s_a & 0 & 0 & s_c & s_b \\ s_b & s_b & s_a & 0 & 0 & 0 \\ 0 & s_c & s_b & s_b & 0 & 0 \\ 0 & 0 & s_c & s_c & s_b & 0 \\ 0 & 0 & 0 & s_a & s_c & s_c \\ s_c & 0 & 0 & 0 & s_a & s_a \end{bmatrix}^T * \begin{bmatrix} P\{abc\} \\ P\{bac\} \\ P\{bca\} \\ P\{cba\} \\ P\{cab\} \\ P\{acb\} \end{bmatrix} = \begin{bmatrix} P\{abc\} \\ P\{bac\} \\ P\{bca\} \\ P\{cba\} \\ P\{cab\} \\ P\{acb\} \end{bmatrix}.$$

Solving the above equation, we have

$$\begin{bmatrix} P\{abc\} \\ P\{bac\} \\ P\{bca\} \\ P\{cba\} \\ P\{cab\} \\ P\{acb\} \end{bmatrix} = \begin{bmatrix} 0.5000 \\ 0.0417 \\ 0.1670 \\ 0.0278 \\ 0.2500 \\ 0.0139 \end{bmatrix}.$$

Notice that there is a remarkable relationship holding between the values above, that is,

$$\begin{aligned} \frac{P\{abc\}}{P\{acb\}} &= 3 = \frac{s_b}{s_c}, & \frac{P\{abc\}}{P\{bac\}} &= 2 = \frac{s_a}{s_b}, & \frac{P\{acb\}}{P\{cab\}} &= 3 = \frac{s_a}{s_c}, \\ \frac{P\{bac\}}{P\{bca\}} &= 2 = \frac{s_a}{s_c}, & \frac{P\{bca\}}{P\{cba\}} &= 3 = \frac{s_b}{s_c}, & \frac{P\{cab\}}{P\{cba\}} &= 2 = \frac{s_a}{s_b}. \end{aligned}$$

This is not coincidental. We shall see that the Markov chain for the transposition heuristic has a special property, namely *time reversibility*, which makes the above relationship hold. It actually provides a strong tool allowing us to compare the cost of the transposition rule to the cost of other rules, and further to derive a (rather complicated) expression for its average search cost.

Consider a stationary ergodic Markov chain (that is, a Markov chain that has been in operation for a long time) having transition probabilities M_{st} and stationary probabilities $P\{\pi_s\}$. Suppose that starting at some time we trace the sequence of states going backwards in time. That is, starting at time t , consider the sequence of states $X_t, X_{t-1}, X_{t-2}, \dots, X_0$. It turns out that this sequence of states is itself a Markov chain with transition probabilities $Q_{st} = (P\{\pi_t\}/P\{\pi_s\}) * M_{ts}$. If $Q_{st} = M_{st}$ for all s, t , then the Markov chain is said to be *time reversible*. Note that the condition for time reversibility, namely $Q_{st} = M_{st}$, can also be expressed as

$$P\{\pi_s\} M_{st} = P\{\pi_t\} M_{ts} \quad \text{for all } s \neq t. \quad (2.13)$$

The condition in the above equation can be stated as, for all states s and t , the rate at which the process goes from s to t (namely $P\{\pi_s\} M_{st}$) is equal to the rate at which the process goes from t to s (namely $P\{\pi_t\} M_{ts}$). It is worth noting that this is an obvious necessary condition for time reversibility since a transition from s to t going backward in time is equivalent to a transition from t to s going forward in time; i.e., if $\pi_m = s$ and $\pi_{m-1} = t$, then a transition from s to t is observed if we are looking backward, and one from t to s if we are looking forward in time.

The following theorem adapted from Ross ([32]) gives the necessary and sufficient condition for a finite ergodic Markov chain to be time reversible. The proof of the theorem can be found in [32] p.143.

Theorem 2.9 *A finite ergodic Markov chain for which $M_{st} = 0$ whenever $M_{ts} = 0$ is time reversible if and only if starting in state s , any path back to s has the same probability as the reversed path. That is, if*

$$M_{s,s_1} M_{s_1,s_2} \dots M_{s_k,s} = M_{s,s_k} M_{s_k,s_{k-1}} \dots M_{s_1,s}$$

for all states $s, s_1, \dots, s_k$. □

Now let us look at Figure 2.3 - the state diagram for the Markov chain resulting from the transposition rule for a list of three elements $\{a, b, c\}$. Observing the paths from state 1 to itself, one path is found if we look forward (clockwise), which is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 1$, and the reversed path can be seen if we look backward (counter-clockwise), which is $1 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$. Taking the product of the transition probabilities of each path, we can see that they are equal. This is actually true for all Markov chains resulting from the transposition heuristic as shown in the following theorem.

Theorem 2.10 *The Markov chain which results from using the transposition heuristic is time reversible. That is, under the transposition heuristic the stationary probabilities obey:*

$$\frac{P \{ R_{i_1} R_{i_2} \dots R_{i_j} R_{i_{j+1}} \dots R_{i_n} \}}{P \{ R_{i_1} R_{i_2} \dots R_{i_{j+1}} R_{i_j} \dots R_{i_n} \}} = \frac{s_{i_j}}{s_{i_{j+1}}} \quad (2.14)$$

for $1 \leq j < n$ if $s_k \neq 0$ for $1 \leq k \leq n$.

Proof:

Using Theorem 2.9, to prove that the Markov chain resulting from the transposition rule is time reversible, it is sufficient to prove that starting in any state π_s , any path back to π_s has the same probability as the reversed path. We will prove it by induction on the number of records, n , in the list.

1. *Base case $n = 3$.*

For any state π_s , say, $\pi_s = \{a, b, c\}$, considering the path from state π_s to itself as shown below (see the state diagram shown in Figure 2.3),

$$\{a, b, c\} \Rightarrow \{b, a, c\} \Rightarrow \{b, c, a\} \Rightarrow \{c, b, a\} \Rightarrow \{c, a, b\} \Rightarrow \{a, c, b\} \Rightarrow \{a, b, c\}.$$

The product of the transition probabilities in the forward direction is

$$s_b s_c s_c s_a s_a s_b = s_a^2 s_b^2 s_c^2, \text{ whereas in the reverse direction, it is } s_c s_c s_b s_b s_a s_a = s_a^2 s_b^2 s_c^2.$$

Therefore, the Markov chain is time reversible.¹

2. Case $n = 4$.

Before going to the case for $n = k$, we shall see how we can construct the state diagram for the case of $n = 4$ from that of the case for $n = 3$. The construction is best illustrated through the diagram shown in Figure 2.4.

In the diagram, each small box represents a subset of the chain of the states. Thus, for example, $\boxed{a \sim b \sim c \mid d}$, contains 6 states. Inside each box (sub-chain), the last element remains in the same position in all 6 states. For example, in sub-chain $\boxed{a \sim b \sim c \mid d}$, element d is the last element in every state inside the sub-chain; only the other three elements a, b, c permute among one another. Thus, from any states inside the box, accesses to a, b, c always transform the state to another state inside *this* sub-chain; to reach any state in another sub-chain, element d must be accessed. Note that since we are modifying the list using the transposition rule, we can only return to the sub-chain $\boxed{a \sim b \sim c \mid d}$ from the states where d is the second last element in the list. Consequently, the sub-chain of the Markov chain which is constrained to be entirely within sub-chain $\boxed{a \sim b \sim c \mid d}$ is actually equivalent to the Markov chain of three elements $\{a, b, c\}$ per state as shown in Figure 2.3, and thus it is time reversible. Note also that all four constituent boxes which represent the sub-chains are symmetric, and totally constitute the entire chain.

Next we will show that the Markov chain consisting of four elements, $\{a, b, c, d\}$, per state is time reversible. Thus, we shall prove that starting in any state π_s , any path back to π_s , has the same probability as the reversed path. Since all four small sub-chains are symmetric, it will be sufficient to prove that if we start in any one of the sub-chains, say, $\boxed{a \sim b \sim c \mid d}$, starting in any state π_s from that sub-chain ($\boxed{a \sim b \sim c \mid d}$), any path back to π_s , has the same probability as the reversed path.

¹Note that this is the only case considered by Ross in [32]. Indeed, it proves the time reversibility of only the base case

Figure 2.4: State diagram with four elements per state using the TR rule when partitioned into four sub-chains. Note that the transition from a state to itself is not shown in the diagram for simplification purposes.

Without loss of generality, let $\pi_s = \{a, b, c, d\}$, then any path back to π_s from inside the sub-chain will have the same probability as the reversed path since we have shown that the Markov chain consisting of states entirely within this sub-chain is time reversible. So it remains to show that any path back to π_s , which goes *outside* the box has the same probability as the reversed path. We know that to reach any states outside the sub-chain, element d must be accessed. After d is accessed, we are in state $\{a, b, d, c\}$ in sub-chain $[a \sim b \sim d \ c]$, and so to get back to state π_s , we have to get back to sub-chain $[a \sim b \sim c \ d]$. Since all sub-chains are symmetrical, we only need to consider the path back to $[a \sim b \sim c \ d]$ directly from $[a \sim b \sim d \ c]$, which, in turn, means that the last state we will visit inside $[a \sim b \sim d \ c]$ right before we get back to $[a \sim b \sim c \ d]$ must be a state where d is the last second element because of the nature of the transposition rule. Since $\{a, b, d, c\}$ has already been visited when we first entered the sub-chain $[a \sim b \sim d \ c]$, we will not allowed to use it again. There is thus only one choice - we must get back from state $\{b, a, d, c\}$. Obviously, the shortest path is thus

$$\{a, b, c, d\} \Rightarrow \{a, b, d, c\} \Rightarrow \{b, a, d, c\} \Rightarrow \{b, a, c, d\} \Rightarrow \{a, b, c, d\}.$$

The product of the transition probabilities in the forward direction is $s_d s_b s_c s_a$, and in the reverse direction is $s_b s_d s_a s_c$. The equality in Theorem 2.9 holds.

The argument holds for a larger path which goes around visiting more states inside $[a \sim b \sim d \ c]$. This is because the large path inside $[a \sim b \sim d \ c]$ also starts from state $\{a, b, d, c\}$ and ends at state $\{b, a, d, c\}$, and it is already proven that sub-chain $[a \sim b \sim d \ c]$ is time reversible.

A similar argument can be given for every state in $[a \sim b \sim c \ d]$. Therefore, the Markov chain for $n = 4$ is time reversible.

Similarly, we can construct the state diagram for the Markov chain with five elements. It can be seen from Figure 2.5 that it consists of five symmetric sub-chains which are equivalent to the Markov chains with three elements.

Figure 2.5: Simplified state diagram with five elements per state using the TR rule.

3. Induction hypothesis.

Assuming that the Markov chain for k records $\{R_1, \dots, R_k\}$ is time reversible, we shall prove that the Markov chain for $(k+1)$ records $\{R_1, \dots, R_k, R_{k+1}\}$ is time reversible. The state diagram with $n = k+1$ is shown in Figure 2.6

As before, considering any state π_s , say $\pi_s = \{R_1, \dots, R_k, R_{k+1}\}$, from one of the sub-chains $\boxed{R_1 \sim \dots \sim R_k \ R_{k+1}}$. We need to prove that starting in π_s , any path back to π_s has the same probability as the reversed path. Again, by our hypothesis, we don't need to consider paths that are inside the sub-chain, but only paths that go outside the sub-chain. To reach any state outside the sub-chain, element R_{k+1} has to be accessed, this makes the transition from π_s to π_s to $\{R_1, \dots, R_{k-1}, R_{k+1}, R_k\}$ in sub-chain $\boxed{R_1 \sim \dots \sim R_{k-1} \sim R_{k+1} \ R_k}$. Since all sub-chains are symmetrical, to get back to π_s , we only need to consider the path that is directly from sub-chain $\boxed{R_1 \sim \dots \sim R_{k-1} \sim R_{k+1} \ R_k}$. A shortest path which visits only two states inside $\boxed{R_1 \sim \dots \sim R_{k-1} \sim R_{k+1} \ R_k}$ is

$$\begin{aligned} \{R_1, R_2, \dots, R_k, R_{k+1}\} &\rightleftharpoons \{R_1, R_2, \dots, R_{k+1}, R_k\} \rightleftharpoons \{R_2, R_1, \dots, R_{k+1}, R_k\} \\ &\rightleftharpoons \{R_2, R_1, \dots, R_k, R_{k+1}\} \rightleftharpoons \{R_1, R_2, \dots, R_k, R_{k+1}\}. \end{aligned}$$

Figure 2.6: Simplified state diagram with $(k + 1)$ elements per state using the TR rule.

The product of the transition probabilities in the forward direction is $s_{k+1}s_2s_ks_1$, whereas in the reverse direction, it is $s_2s_{k+1}s_1s_k$. The argument holds for any large path for the same reason stated for the case where $n = 4$. Therefore, the Markov chain is time reversible, and the inductive step is complete. The result follows. $\square$

It is worth mentioning that Rivest [31] also proved Theorem 2.10, but didn't use a constructive scheme to prove time reversibility. He alluded to the final result since it is the *unique* eigenvalue of the Markov chain, where he derived the stationary probability of the chain. Time reversibility was thus not a condition he sought to prove. The constructive proof we have presented above gives us a further insight into the algorithm, and is actually the only known proof of the time reversibility of the transposition rule. Ross [32] also gave a proof of the time reversibility of the transposition rule, but his proof is far too sketchy and proven only for the base case of the induction shown above.

Corollary 2.3 *Under the transposition heuristic, the stationary probabilities between any two arbitrary states π_s and π_t obey:*

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_t, \pi_s)} \quad (2.15)$$

where $\delta_i(\pi_t, \pi_s) = \pi_t(i) - \pi_s(i)$ for $1 \leq i \leq n$ is the number of places that record R_i is displaced in π_s from its position in π_t .

Proof:

Since the Markov chain representing the scheme is time reversible, so we can reach any state from any other by a sequence of one or more transformations. Assuming that the sequence of intermediate states which transforms state π_s to state π_t are $\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_k}$ for $1 \leq k \leq n! - 2$, i.e., we can go from state π_{i_j} to state $\pi_{i_{j+1}}$ by one transformation for $1 \leq j \leq k - 1$. Then,

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \frac{P\{\pi_s\}}{P\{\pi_{i_1}\}} \times \frac{P\{\pi_{i_1}\}}{P\{\pi_{i_2}\}} \times \dots \times \frac{P\{\pi_{i_{k-1}}\}}{P\{\pi_{i_k}\}} \times \frac{P\{\pi_{i_k}\}}{P\{\pi_t\}}.$$

Theorem 2.10 gives the expression for each of the fractions on the right hand side of the above equation. Thus

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{u,v} \left(\frac{s_u}{s_v} \right) \quad \text{for some } u \text{ and } v \text{ where } 1 \leq u, v \leq n. \quad (2.16)$$

Note that each transformation from state π_{i_j} to state $\pi_{i_{j+1}}$ transposes record R_u and record R_v . Note also that if a record appears in π_s in the position prior to its position in π_t , then its access probability will appear as a numerator in the expression shown by Equation (2.16) above. Similarly, if a record appears in π_s in the position that is after its position in π_t , its access probability will appear as a denominator in Equation (2.16).

The number of times that record R_i will be transposed during the transformations from π_s to π_t is the number of places that R_i is displaced from its position in π_t , i.e., $\delta_i(\pi_t, \pi_s)$. If record R_i appears in π_s in the position prior to its position in π_t , then the value of $\delta_i(\pi_t, \pi_s)$ will be positive, whereas if it appears in π_s in the position that is after its position in π_t , the value of $\delta_i(\pi_t, \pi_s)$ will be negative. If

R_i appears in the same position in π_s as it is in π_t , then the value of $\delta_i(\pi_t, \pi_s)$ is 0. Therefore,

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_t, \pi_s)},$$

and the result is proved. $\square$

Let us look at an example. Suppose that $\pi_s = \{3, 1, 4, 2\}$ and $\pi_t = \{1, 2, 3, 4\}$. To transform from π_s to π_t , one path is $\pi_s \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \pi_t$ where $\pi_1 = \{1, 3, 4, 2\}$ and $\pi_2 = \{1, 3, 2, 4\}$ as shown in the following figure.

Using time reversibility, we have

$$\begin{aligned} \frac{P\{\pi_s\}}{P\{\pi_t\}} &= \frac{P\{\pi_s\}}{P\{\pi_1\}} \times \frac{P\{\pi_1\}}{P\{\pi_2\}} \times \frac{P\{\pi_2\}}{P\{\pi_t\}} \\ &= \frac{s_3}{s_1} \times \frac{s_4}{s_2} \times \frac{s_3}{s_2} \\ &= \frac{s_3^2 s_4}{s_2^2 s_1} = s_1^{-1} s_2^{-2} s_3^2 s_4^1. \end{aligned}$$

It can be easily verified that Corollary 2.3 gives the same result.

Rivest [31] reported a complicated expression for the average search cost of the transposition heuristic using time reversibility. It is given in the following theorem along with its proof which we have not been able to locate in the literature.

Theorem 2.11 *Let π_0 denote the identity permutation on n records where $\pi_0(i) = i$ for $1 \leq i \leq n$; which is the optimal ordering under our assumption that $s_i \geq s_{i+1}$ for $1 \leq i \leq n$. Let $\delta_i(\pi_0, \pi)$ denote the quantity $i - \pi(i)$ for any permutation π and $1 \leq i \leq n$, which is the number of places that R_i is displaced from its optimal position in π . Then the asymptotic search cost using the transposition heuristic is*

$$P\{\pi_0\} \sum_{\text{all } \pi} \left(\left(\prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right) \quad (2.17)$$

where

$$P\{\pi_0\} = \left(\sum_{\text{all } \pi} \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \right)^{-1}. \quad (2.18)$$

Proof:

From Equation (2.6) we know that the average search cost of an algorithm can be calculated by the following formula

$$\sum_{\text{all } \pi} \left(P\{\pi\} \sum_{1 \leq j \leq n} s_j \pi(j) \right).$$

Under the transposition rule, using Corollary 2.3, we have,

$$P\{\pi\} = P\{\pi_0\} \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \quad \text{for all } \pi. \quad (2.19)$$

Consequently, we see that

Average search cost for Transposition

$$\begin{aligned} &= \sum_{\text{all } \pi} \left(\left(P\{\pi_0\} \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right) \\ &= P\{\pi_0\} \sum_{\text{all } \pi} \left(\left(\prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right), \end{aligned}$$

thus Equation (2.17) is proved.

Since

$$\sum_{\text{all } \pi} P\{\pi\} = 1,$$

applying Equation (2.19), we have

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

which proves Equation (2.18). $\square$

Theorem 2.11 gives us the average search cost of the transposition rule. The following results prove that the transposition heuristic costs less than or equal to the move-to-front heuristic does.

Theorem 2.12 *The asymptotic probability that record R_j is before record R_i in the list given that $s_j > s_i$ is always larger under the transposition heuristic than under the move-to-front heuristic. That is*

$$P \{R_j \text{ precedes } R_i\}_{TR} > P \{R_j \text{ precedes } R_i\}_{MTF} \quad \text{if } s_j > s_i$$

for $1 \leq i, j \leq n$.

Proof:

Consider any state where R_j precedes R_i , say $(\dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots)$. By successive transposition using Theorem 2.10, we can show that

$$P \{\dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots\} = \left(\frac{s_i}{s_j}\right)^{k+1} P \{\dots, R_j, R_{i_1}, \dots, R_{i_k}, R_i, \dots\}.$$

Since $s_j > s_i$, $\left(\frac{s_i}{s_j}\right)^{k+1} < \left(\frac{s_i}{s_j}\right)$. The above equation implies that

$$P \{\dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots\} < \left(\frac{s_i}{s_j}\right) P \{\dots, R_j, R_{i_1}, \dots, R_{i_k}, R_i, \dots\}.$$

Summing over all states for which R_j precedes R_i using the above, we have

$$\sum_{\text{state } \pi} P \{\pi \mid R_i \text{ precedes } R_j \text{ in } \pi\} < \left(\frac{s_i}{s_j}\right) \sum_{\text{state } \pi} P \{\pi \mid R_j \text{ precedes } R_i \text{ in } \pi\},$$

which implies that

$$P \{R_i \text{ precedes } R_j\} < \left(\frac{s_i}{s_j}\right) P \{R_j \text{ precedes } R_i\}.$$

Since $P \{R_i \text{ precedes } R_j\} = 1 - P \{R_j \text{ precedes } R_i\}$, it yields

$$P \{R_j \text{ precedes } R_i\}_{TR} > \frac{1}{1 + \left(\frac{s_i}{s_j}\right)} = \frac{s_j}{s_i + s_j}.$$

By Equation (2.9), the proof completes. $\square$

From the above theorem and Theorem 2.1, we easily get the following result.

Theorem 2.13 *The asymptotic search cost of the transposition heuristic is less than or equal to that of the move-to-front heuristic for every probability distribution with equality holding when $n = 2$ or the distribution where all nonzero probabilities $\{s_i\}$ are equal. $\square$*

2.3.2 Comparisons with the Cost of the Optimal Ordering

The worst-case cost of the move-to-front rule has been shown by Chung *et al.* [9] to be no more than half a π times that of the optimal static ordering. However the worst-case performance of the transposition rule could be much worse, as Bentley and McGeoch [5] point out. This can easily be seen by considering a case in which two records are alternately accessed many times, they will continue to exchange places without advancing toward the front of the list.

Since we assume that there is no locality in the query sequence, according to Theorem 2.4 and Theorem 2.12, we conclude that

$$\text{Cost (TR)} < \text{Cost (MTF)} < \frac{\pi}{2} * \text{Cost (OPT)}.$$

2.3.3 Rate of Convergence

No general formula has been derived for the *overwork* measure of convergence for the transposition heuristic. Bitner showed the overwork of the transposition rule for a particular distribution given in the following theorem. The proof of the theorem can be found in [7].

Theorem 2.14 *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 and the move-to-front rule are:*

$$OV_{TR} = \frac{n^2 - 1}{6} \quad \text{and} \quad OV_{MTF} = \frac{n - 1}{2}.$$

□

Note that the overwork for the transposition rule is $O(n)$ greater than the overwork for the move-to-front rule. Bitner also gave another simpler distribution, with the same results. He also empirically calculated the values of the overwork for the transposition rule under Zipf's law for various values of n . His values appear to be $O(n^3)$, which is again $O(n)$ greater than his overwork measure for the move-to-front rule. It indicates that the transposition rule converges much slower than the move-to-front rule.

# of accesses n	overwork r
3	6
4	10
5	14
6	20
7	27
10	50
20	212

Table 2.2: The Overwork, r , for Zipf's Distribution (Adapted from Bitner [79]).

Bitner suggested another measure of convergence which is the number of accesses, r , required for the *total expected cost* of the transposition rule to be smaller than that of the move-to-front rule. This *total expected cost* is the sum of the costs for the first r accesses, and gives us an indication as to which rule will be cheaper when retrieving r requests. Table 2.2 shows some values for r under Zipf's law. Note that r increases quadratically with n , showing again that the move-to-front rule converges much faster than the transposition rule.

Gonnet *et al.* [12] gave a measure for how many accesses it takes for the transposition rule to reach its steady state assuming that $\text{Cost (TR)} < (\pi/2) * \text{Cost (OPT)}$. Again we state the theorem without any proofs.

Theorem 2.15 *The transposition heuristic can take as many as $\Omega(n^2)$ accesses to reach within a factor of $(1 + \epsilon)$ of the steady state behavior.* $\square$

The conclusion is clear: although the transposition rule converges much slower than the move-to-front rule, its lower asymptotic cost makes it a better choice when the number of accesses is large.

2.4 Hybrid Algorithms of MTF and TR

The move-to-front heuristic and the transposition heuristic clearly have trade-offs between convergence and asymptotic search cost. If the number of accesses is known beforehand to be small, move-to-front is probably the better algorithm, whereas transposition may be better if the number of accesses is expected to be large. In this section we shall present algorithms that attempt to incorporate the best of both measures.

2.4.1 The Move-Ahead- k Heuristic

The *move-ahead- k* heuristic is a compromise between the two extremes - move-to-front and transposition. In this scheme, the accessed record 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. Notice that it is a generalization of the transposition rule and the move-to-front rule as transposition is *move-ahead-1* and move-to-front is *move-ahead- n* .

The scheme was initially proposed by Rivest [31]. He conjectured that the move-ahead- k rule is superior to the move-ahead- $(k + 1)$ rule; but this is unproven. Absolute analysis of the scheme in term of deriving the formula for the average search cost etc. is difficult and is not available in the literature. Intuition tells us that its performance (cost and convergence) should lie between the transposition rule and the move-to-front rule.

Empirical evidence from Tenenbaum [34] gives the best values of k for lists of various sizes. Simulations were done on lists with 12,000 accesses distributed by Zipf's law (see Table 2.3). The size of the list is shown to have a direct correlation to the best value of k .

The cost of the move-ahead- k heuristic may also be affected by the way it is implemented. A straightforward implementation can be done by keeping an extra pointer to the record k positions prior to the accessed record. Searching forward is

size of the list, n	best k
3-64	1
65-94	2
95-124	3
125-150	4
151-183	5
184-204	6
205-230	7

Table 2.3: Best k for Different Sized Lists for Move-ahead- k (From Tenenbaum[78]).

undesirable because it would double the search cost by requiring a second traversal of the list.

2.4.2 The $POS(k)$ Heuristic and the $SWITCH(k)$ Heuristic

Tenenbaum and Nemes [35] suggested two classes of hybrids. The first is the $POS(k)$ rule. It moves the accessed record to position k of the list if it is in positions $k + 1$ to N , or it transposes it with its preceding record if it is in positions 2 to k . If it is the first in the list, it is left unchanged. Note that $POS(1)$ is the move-to-front, whereas $POS(n-1)$ is the transposition strategy.

The second class is the $SWITCH(k)$ rule. It has the same rule as $POS(k)$ except that the uses of move-to-front and transposition are reversed, an accessed record found in positions 2 to k is moved to the front and others are transposed.

Tenenbaum and Nemes proved various results about the $POS(k)$ rule and the $SWITCH(k)$ rule, primarily involving a distribution in which $p_2 = p_3 = \dots = p_N = (1 - p_1)/(N - 1)$.

2.4.3 Bitner's Hybrid Algorithm

Bitner [7] also proposed a simple hybrid rule which has the fast convergence of the move-to-front rule and the low asymptotic cost of the transposition rule. It initially uses the move-to-front rule until its steady state is approached, then switches to the transposition rule. Therefore initially it behaves like the move-to-front rule and has rapid convergence, asymptotically it behaves like the transposition rule and has low asymptotic cost. Determining when to switch is difficult and is the major disadvantage of this rule.

Bitner suggested switching when the number of requests is between $\Theta(n)$ and $\Theta(n^2)$ (see [6]). He referred to his Ph.D thesis, where he addressed the issue of finding good bounds for when to change from one to the other but found it difficult and was forced to make a guess on the basis of specific observations derived by assuming Zipf's law.

2.5 Batched Algorithms

The following algorithms are used in conjunction with permutation algorithms. When a record is accessed, it is up to the *batched algorithm* to decide whether to reorganize the list or not (i.e., whether to invoke the permutation algorithm or not). The purpose of using batched algorithms is to slow down the convergence of permutation algorithms by not moving records on the basis of single accesses only. Analysis available in the literature are primarily on those combined with either the move-to-front rule or the transposition rule.

2.5.1 The Move-Every- k th-Access Heuristic

McCabe [23] considered another scheme where the record is moved forward only after it has been accessed k times. It reduces the time spent reordering the list, but requires n extra space for counters - one for each record in the list to keep track of the number of accesses made on that record. Initially all counters are set to zero.

When a record is accessed, the corresponding counter for it is incremented. If the counter is equal to k , the record is moved forward according to the permutation rule used, and the counter for that record is reset to zero.

McCabe showed that, independent of the value of k , the asymptotic search cost of move-every- k th-access would be the same as the corresponding permutation algorithm, and the number of accesses required for the move-every- k th-access rule to reach its asymptote or steady state is k times the number of accesses required for the permutation rule by itself to approach the same asymptote.

Bitner [7] also studied this rule (he called it *wait c and move* rule) and analyzed its performance with the corresponding permutation algorithm being the move-to-front rule. He proved that we could actually do better - the cost of the wait c and move with move-to-front is less than the cost of the move-to-front rule by itself.

Theorem 2.16 (See [7], p92 for the proof.) *The asymptotic search cost of the move-every- k th-access when used with the move-to-front rule is*

$$1 + \sum_{i=1}^n \sum_{j \neq i} (s_i P \{R_j \text{ precedes } R_i\})$$

where

$$P \{R_j \text{ precedes } R_i\} = \frac{s_j}{(s_i + s_j)(k+1)^2} \sum_{l=0}^k (k-l+1) \left(\frac{s_i}{s_i + s_j} \right)^k \sum_{m=0}^k \binom{m+l}{l} \left(\frac{s_j}{s_i + s_j} \right)^m.$$

□

Theorem 2.17 (See [7], p94 for the proof.) *For $k \geq 1$ and any set of access probabilities except the uniform distribution or a distribution with a record of probability one, the move-every- k th-access with move-to-front has strictly lower cost than the move-to-front rule by itself.*

□

Bitner suggested a modification to his *wait c and move* rule. It is called the *wait c , move and clear* rule. In this scheme, all of the counters are reset whenever one

reaches the limit (instead of resetting only the one that reaches the limit). The cost of resetting all the counters may be very significant. However, as Bitner remarks, if all counters are stored in the same area (instead of being directly associated with the record), they can be efficiently reset by zeroing a contiguous area of core. Bitner proved that as $c \rightarrow \infty$ the asymptotic cost of the wait c , move and clear rule approaches the optimum.

Theorem 2.18 *The asymptotic search cost of the wait c , move and clear rule is exactly the same as the asymptotic search cost of the corresponding permutation rule with modified access probabilities $\hat{s}_1(c), \hat{s}_2(c), \dots, \hat{s}_n(c)$ where*

$$\hat{s}_i(c) = \sum_{a_1=0}^c \dots \sum_{a_{i-1}=0}^c \sum_{a_{i+1}=0}^c \dots \sum_{a_n=0}^c \left(\frac{(c + a_1 + \dots + a_{i-1} + a_{i+1} + \dots + a_n)!}{c! a_1! \dots a_{i-1}! a_{i+1}! \dots a_n!} (s_i^{c+1} s_1^{a_1} \dots s_{i-1}^{a_{i-1}} s_{i+1}^{a_{i+1}} \dots s_n^{a_n}) \right).$$

□

Theorem 2.19 *For $c \geq 1$ the asymptotic search cost of the wait c , move and clear is less than that of the move-to-front rule.*

□

Theorem 2.20 *As $c \rightarrow \infty$ the asymptotic search cost of the wait c , move and clear rule approaches the optimum.*

□

Bitner proved that the asymptotic search cost of the wait c and move rule does not approach the optimum as $c \rightarrow \infty$. However it has faster convergence than the wait c , move and clear rule, but is still much slower than the corresponding permutation algorithm. In addition, both rules require n extra memory on the counters.

2.5.2 The k -in-a-Row Heuristic

The k -in-a-row heuristic only reorganizes the list (by applying the permutation algorithm) if the accessed record has been required k times in a row. Such rules require

two additional counters of memory space - one for keeping track of the last record requested and the other for keeping track of the number of accesses in a row it had been requested. Once a record has been accessed k times in a row we reorder the list according to the permutation algorithm chosen and then start over again as far as waiting for another run of k identical requests.

Kan and Ross [17] proposed this scheme. They proved that as k approaches infinity we can do as well as if we knew the access probabilities of the records in the list, and in addition they showed that the convergence is monotone if the move-to-front rule is used as the permutation algorithm. The related results are presented in the following theorems. The corresponding proofs can be found in [17].

Theorem 2.21 *The asymptotic search cost of the k -in-a-row rule is exactly the same as the asymptotic search cost of the corresponding permutation rule (by itself) with modified access probabilities $s_1^{(k)}, s_2^{(k)}, \dots, s_n^{(k)}$ where*

$$s_i^{(k)} = \frac{s_i^k (1 - s_i) / (1 - s_i^k)}{\sum_{j=1}^n (s_j^k (1 - s_j) / (1 - s_j^k))}.$$

□

Theorem 2.22 *If the move-to-front rule is applied to the k -in-a-row rule, then the average search cost of the k -in-a-row rule, namely*

$$\sum_{i=1}^n \sum_{j \neq i} \frac{s_i^{(k)} s_j^{(k)}}{s_i^{(k)} + s_j^{(k)}}$$

is a decreasing function of k .

□

Theorem 2.23 *Under the k -in-a-row rule, the asymptotic probability that the record with the j th-largest access probability is in position j approaches 1 as k goes to ∞ .*

□

Gonnet *et al.* [12] also studied this heuristic and showed that the cost of the k -in-a-row rule with move-to-front as the permutation algorithm is no more than $(1 + O(\ln k / k))$ times the cost of the optimal ordering. Further they proved that

the $(k + 1)$ -in-a-row rule is superior to the k -in-a-row which strengthens Kan and Ross's result in Theorem (2.2)3. These results are summarized in the following theorems. Proofs can be found in [12].

Theorem 2.24 *The ratio of the expected search cost of the k -in-a-row rule with move-to-front as the permutation algorithm to the search cost of the optimal ordering is bounded by*

$$\frac{\text{Cost}(k\text{-in-a-row with MTF})}{\text{Cost}(OPT)} \leq 1 + O\left(\frac{\ln k}{k}\right).$$

□

Theorem 2.25 *For $k > 1$ $\text{Cost}((k + 1)\text{-in-a-row}) \leq \text{Cost}(k\text{-in-a-row})$. Furthermore, the inequality is strict for non-trivial distributions. (A distribution is trivial if $s_i = s_j$ for all non zero probabilities or if $s_i = 0$ for $i \geq 3$.)*

□

These rules have the advantage of not requiring as much extra memory as do Bitner's batched algorithms introduced in the previous section.

2.5.3 The k -in-a-Batch Heuristic

Gonnet *et al.* [12] suggested a modification to the k -in-a-row rule called the k -in-a-batch rule. In this scheme, accesses are grouped into batches of k consecutive requests, a permutation algorithm is applied only if all k accesses in a batch are for the same record. This scheme may seem to be equivalent to the k -in-a-row scheme. However, if a record is accessed and subsequently not moved forward because of the access of some other (unknown) record, the original record has a lower probability of being moved forward than in the case of the k -in-a-row scheme. Intuitively, these heuristics are better than the k -in-a-row heuristics because these heuristics perform fewer changes.

Gonnet *et al.* give the asymptotic probability that record R_j is before record R_i in the list when applying the k -in-a-batch heuristic with the move-to-front rule as the permutation algorithm. It is

$$P\{R_j \text{ precedes } R_i\} = \frac{s_j^k}{s_i^k + s_j^k}.$$

By Theorem 2.1, we have the following result:

Theorem 2.26 *For $k > 1$ the asymptotic search cost of the k -in-a-batch heuristic in combination with the move-to-front heuristic is less than or equal to that of the move-to-front heuristic.* $\square$

Chung *et al.* [9] derived the worst-case bounds for the k -in-a-batch with the move-to-front rule using some combinatorial inequalities.

Theorem 2.27 (See [9], p156 for the proof) *The ratio of the expected search cost of the k -in-a-row rule with move-to-front as the permutation algorithm to the search cost of the optimal ordering is bounded by*

$$\frac{\text{Cost (}k\text{-in-a-batch with MTF)}}{\text{Cost (OPT)}} \leq \frac{\pi}{2k} \csc \frac{\pi}{2k}.$$

$\square$

Gonnet *et al.* proved that the k -in-a-batch rule is better than the k -in-a-row rule when in combination with either the move-to-front rule or the transposition rule.

Theorem 2.28 (See [12] for the proof) *For $k > 1$ the asymptotic search cost of the k -in-a-batch heuristic in combination with either the move-to-front heuristic or the transposition heuristic is less than or equal to that of k -in-a-row heuristic with the same permutation algorithms respectively.* $\square$

These rules require two additional memory space for the counters as the k -in-a-row rules do.

2.6 Probabilistic Algorithms

Almost all of the schemes discussed in the literature are represented by Markov chains which are ergodic. This means that the list can be in any one of its $n!$ list orderings. Taking the move-to-front rule as an example, a fairly organized list can be quite disorganized by a single request for the record which is accessed relatively

infrequently. Notice that after this unfortunate occurrence, it will take a long time for the list to be organized again, i.e., for this record to shift back to the tail of the list.

As Oommen *et al.* [26] remark, if the access probabilities are time invariant, clearly, the ergodic Markovian representation of a list organizing scheme is undesirable. As opposed to ergodic representation, Markovian behavior can also be absorbing [21]. In an absorbing Markov chain, the chain converges to one of the absorbing barriers. The problem of computing the probability of converging to any particular barrier involves evaluating what is called the first passage probability which is well known in the theory of stochastic processes [21].

The literature reports only two *stochastic absorbing* list organizing schemes [25] and two *deterministic absorbing* schemes [26]. They will be presented in the sections that follows.

2.6.1 The Stochastic Move-to-Front Scheme

Oommen and Hansen [25] proposed a list organizing strategy using stochastic move-to-front rule. In this scheme, the move operation is performed stochastically in such a way that ultimately no more move operations are performed. When this has occurred, they say that the scheme has converged.

The scheme is essentially a move-to-front algorithm with the exception that on the n th access, the accessed record is moved to the front of the list with a probability $f(n)$. This probability is systematically *decreased* every time a record is accessed. After the operation is executed for a sufficiently long period, each record tends to stay in place - instead of being moved to the front. This makes the Markovian representation of the scheme to be absorbing - the organization of the list gets "absorbed" to one of the $n!$ orderings.

If $f(n)$ is the probability of a record being moved to the front of the list on being accessed at time n , then the probability that the record will stay in place (instead

of being moved) is $(1 - f(n))$. They define the initial condition to be

$$f(0) = a \quad \text{where } 0 < a < 1$$

and the updating rule to be

$$f(n+1) = a f(n) \quad \text{where } 0 < a < 1.$$

Note that $f(n)$ can actually be computed by the formula below

$$f(n) = a^n.$$

This tells us that $f(n)$ is decreased by a multiplicative constant after each access. This makes the scheme less optimal than the deterministic move-to-front algorithm in all cases. However Oommen *et al.* show that this scheme is expedient - i.e., if $s_j > s_i$ the probability of absorption into an arrangement in which record R_j precedes record R_i is always greater than 0.5.

2.6.2 The Stochastic Move-to-Rear Scheme

Oommen *et al.* [25] proposed another stochastic scheme called *stochastic Move-to-rear*, which moves the accessed record R_j to the *rear* of the list with a probability q_j which is progressively decremented every time record R_j is accessed. Again ultimately no more move operations are performed on the list. So the Markovian representation of the scheme is absorbing, and could converge to any one of its $n!$ orderings. However, Oommen *et al.* showed that the probability of convergence to the optimal ordering could be made as close to unity as desired.

This scheme is different from the stochastic move-to-front scheme introduced in the previous section in two places. First it performs *move-to-rear* operations instead of move-to-front. Secondly q_j (the probability of moving the accessed record R_j to the rear of the list) is associated with record R_j and it is defined differently from $f(n)$.

Initially all the q_j 's are set to 1, i.e., $q_j(0) = 1$ for all $1 \leq j \leq n$. Then it is decremented every time R_j is accessed by the following rule

$$q_j = \begin{cases} a q_j(n) & \text{if } R_j \text{ is accessed} \\ q_j(n) & \text{otherwise.} \end{cases}$$

Oommen *et al.* prove that given two records R_j and R_k , the probabilities q_j and q_k converge to zero with probability 1. Further, at any instant, they prove that

$$s_j > s_k \quad \text{if and only if} \quad E[q_j(n)] < E[q_k(n)] \quad \text{for all } 0 < a < 1.$$

By appropriately choosing the value of " a ", they show the following results (see [25] for the proofs).

Theorem 2.29 *For all j and k where $j \neq k$, if $s_j > s_k$, then the quantity $P\{q_j(n) < q_k(n)\}$ can be made as close to unity as desired.* $\square$

Theorem 2.30 *The asymptotic probability of record R_j preceding record R_i is unity if and only if $s_j > s_i$.* $\square$

Corollary 2.4 *The probability of the list converging to the optimal arrangement can be made as close to unity as desired.* $\square$

This scheme requires linear space. However, apart from it being much more accurate than all the previous algorithms, it is also computationally more efficient. Since a list operation is essentially stochastic, it is not necessarily performed on every access. Further, since the Markovian representation of the scheme is absorbing, the number of list operations performed *asymptotically decreases to zero*. This greatly reduces the time spent reordering the list.

Another interesting property of this scheme is that it is a *move-to-rear* scheme. This is indeed quite counter-intuitive. But the fact is that by *stochastically* moving an accessed record to the *rear* of the list, as optimal list arrangement could result.

2.6.3 The Deterministic Move-to-Rear Scheme

Oommen *et al.* [26] suggested another two absorbing list organizing schemes. They are again of a move-to-rear style. Different from the stochastic move-to-rear scheme, they are deterministic. In addition, the number of move operations performed in both schemes are exactly equal to the number of records in the list as a move operation is performed on a record *exactly* once.

The first scheme moves the accessed record to the *rear* of the list if it has been accessed T times. The scheme is asymptotically optimal - the probability of being absorbed into the optimal ordering can be made as close to unity as desired. The second scheme moves the accessed record to the *rear* of the list if it has been accessed T times in a row. It is proven to be expedient. They conjectured its optimality.

Note that in both schemes, a record will not be moved again after it has been moved once. The first scheme requires linear space whereas the second requires constant space.

Later, Oommen and Ng [27] modified the latter scheme to yield an optimal algorithm.

1. The Deterministic Linear Space Move-to-rear Scheme

This scheme requires linear space (space proportional to the number of records in the list). We shall describe the algorithm in the pseudo-code form and then present the properties of the scheme [26].

Memory Requirements:

- (1) An integer memory location x_i (access count) associated with each record R_i .
- (2) A boolean variable M_i for each record R_i indicating if it has already been moved.
(The actual implementation can be done without this. See part 4 for details.)
- (3) An integer constant T .

Deterministic-MTR-LSpace:

```

    for  $i = 1$  to  $n$  do
        set  $x_i = 0$  and  $M_i = \text{False}$ 
    endfor
    repeat
        ReadInput ( $R_i$ )
        if Not( $M_i$ ) then
            let  $x_i = x_i + 1$ 
            if ( $x_i = T$ ) then
                Move-to-Rear ( $R_i$ )
                set  $M_i = \text{True}$ 
            endif
        endif
    endif
    forever
end Algorithm Deterministic-MTR-LSpace

```

The above algorithm possesses the following properties proven in [26].

Theorem 2.31 *The deterministic linear space move-to-rear scheme is absorbing. Furthermore, the total number of list reorganizing operations done is exactly n . $\square$*

Theorem 2.32 *For any two records R_u and R_v , let $P\{R_j \text{ precedes } R_i\}$ be the asymptotic probability that R_u precedes R_v given that either R_u or R_v is accessed. Then under the deterministic linear space move-to-rear scheme we have,*

- (i) $P\{R_j \text{ precedes } R_i\}$ can be made as close to unity as desired if $s_u > s_v$.
- (ii) $P\{R_j \text{ precedes } R_i\}$ is exactly 0.5 if $s_u = s_v$. $\square$

Theorem 2.33 *Assuming that the access probabilities $\{s_1, s_2, \dots, s_n\}$ are all distinct, then the probability of the list converging into the optimal ordering under the deterministic linear space move-to-rear scheme can be made as close to unity as desired. $\square$*

2. The Deterministic Constant Space Move-to-rear Scheme

This scheme requires constant space (space independent of the number of records in the list). The algorithm and the properties of the scheme [26] are shown below.

Memory Requirements:

- (1) Two integers Z_1 and Z_2 . Z_1 stores the index of the last accessed record and Z_2 counts the number of times it has been consecutively accessed.
- (2) A boolean variable M_i for each record R_i indicating if it has already been moved.
(The actual implementation can be done without this. See part 4 for details.)
- (3) An integer constant T .

Deterministic-MTR-CSpace:

```

    for  $i = 1$  to  $n$  do
         $M_i = \text{False}$ 
    endfor
    set  $Z_1 = -\infty$  and  $Z_2 = 0$ 
    repeat
        ReadInput ( $R_i$ )
        if Not ( $M_i$ ) then
            if ( $Z_1 = i$ ) then
                let  $Z_2 = Z_2 + 1$ 
            else
                set  $Z_2 = 1$  and  $Z_1 = i$ 
            endif
            if ( $Z_2 = T$ ) then
                Move-to-Rear ( $R_i$ )
                set  $M_i = \text{True}$ 
            endif
        endif
    forever
end Algorithm Deterministic-MTR-CSpace

```

Theorem 2.34 *The deterministic constant space move-to-rear strategy is expedient.*

□

The pseudo-code above indicates that a linear space is required for the algorithm as it has a boolean variable M_i associated with each record. However as we shall see later in part 4 of this section that by introducing an extra pointer, the implementation can be done with only constant space.

3. The Optimal Constant Space Move-to-rear Scheme

The *optimal constant space move-to-rear* scheme [27] is an improved version of the deterministic constant space move-to-rear scheme. It is asymptotically optimal.

Let $\mathcal{R}$ be the original list of records $\{R_1, \dots, R_n\}$, and let $\mathcal{B}$ be the set of records that have not been moved from their initial positions. $\mathcal{B}$ is initially set to be $\mathcal{R}$ and is repeatedly shrunk until it is the null set.

The list is reorganized if the accessed record is in set $\mathcal{B}$ and has been accessed T times in a row. The pseudo-code of the scheme is given below.

Memory Requirements:

- (1) Two integers Z_1 and Z_2 . Z_1 stores the index of the last accessed record and Z_2 counts the number of times the record with index Z_1 has been consecutively accessed.
- (2) A set of maximum n records, $\mathcal{B}$.
(The actual implementation can be done without this. See part 4 for details.)
- (3) An integer constant T .

Optimal-MTR-CSpace:

```

    set  $\mathcal{B} = \mathcal{R}$  and  $Z_1 = 0, Z_2 = 0$ 
    repeat
        ReadInput ( $R_i$ )
        if  $R_i \in \mathcal{B}$  then
            if ( $Z_1 = 0$ ) then
                set  $Z_1 = i$  and  $Z_2 = 1$ 
            elseif ( $Z_1 = i$ ) then
                let  $Z_2 = Z_2 + 1$ 
            else
                set  $Z_1 = 0$  and  $Z_2 = 0$ 
            endif
            if ( $Z_2 = T$ ) then
                Move-to-Rear ( $R_i$ )
                 $\mathcal{B} = \mathcal{B} - R_i$ 
            endif
        endif
    forever
end Algorithm Optimal-MTR-CSpace

```

The fundamental difference between this algorithm and all the other algorithms reported in the literature is that the list reorganization is done based not only on the accesses but also on some conditional events - the reorganization is conditioned on the event that the accessed record is in set $\mathcal{B}$. Therefore the analysis is based on the *time-varying conditional* access probabilities. Although records in set $(\mathcal{R} - \mathcal{B})$ will also be accessed, these accesses will not have any impact on the reorganization of the list or the convergence of the list. Only records that are never moved are involved in the list reordering. This is the *only* scheme known in the literature that reorganizes a list on the base of such conditional events.

The major results from [27] are presented below. Proofs can be found in [27].

Theorem 2.35 *For any two records R_u and R_v , if $s_u > s_v$, then the asymptotic probability that R_u precedes R_v under the optimal constant space move-to-rear scheme increases monotonically with M and it is actually*

$$P\{R_j \text{ precedes } R_i\} = \frac{s_u^M}{s_u^M + s_v^M}.$$

□

Theorem 2.36 *The constant space move-to-rear strategy is asymptotically optimal.*

□

This scheme is stochastically absorbing and performs exactly n list reorganizing operations. Note that it can also be viewed as an MTF strategy in which the front of the list is dynamically changed (see next part for details).

4. Implementation Details

Note that in a pure *move-to-rear* scheme, the records which are accessed very infrequently would linger at the front of the list until they are themselves at the rear of the list. These infrequently accessed records could thus potentially prove to be a drag on the entire access strategy. However this potential drawback can easily disappear by a specific implementation introduced by Oommen *et al.* [26].

Considering the deterministic linear space scheme described in part 1 of this section. Notice that initially the scheme actually moves the most frequently accessed records to the rear of the list. Thus the records which are accessed very infrequently would linger at the front of the list until they are accessed T times. However this can be easily overcome by maintaining two list pointers, *FrontOld* and *FrontNew*. *FrontOld* always points at the front of the list and *FrontNew* always points at the record in the list which is first moved to the rear.

Searching through the list always starts from the *FrontNew*. If the record is found, *neither* the counter (x_i in the pseudo-code) gets updated *nor* the list gets reorganized. If the record searched for is not found in the list pointed to by the

Figure 2.7: Implementation of the MTR rule (Adapted from Oommen et al.[90]).

FrontNew, a search continues from the **FrontOld**, terminates at **FrontNew** and *may* result in a list reordering.

The implementation of the scheme is illustrated in Figure 2.7 for a list of five records where $s_1 > s_2 > s_3 > s_4 > s_5$. **FrontOld** points to the original Head of the list and **FrontNew** points to the element which was first moved to the rear. The list is traversed from **FrontNew**, and if the element sought for is not obtained, it is then traversed from **FrontOld**.

The principles are analogous for the other schemes described in this section.

2.7 Conclusion

In this chapter, we have given an extensive survey of the previous self-organizing linear search algorithms, along with analytic results concerning them. To compare the performance of different algorithms, we have given the criteria and the analysis techniques applied to self-organizing linear search algorithms. We have focused on the average search cost of an algorithm in terms of the average number of probes required to find a record in the given list after the list has approached a steady state, and we have measured convergence by the amount of time or the number of accesses required to approach the steady state.

In particular, we have studied the move-to-front heuristic, the transposition heuristic, the move-ahead- k heuristic, hybrid heuristics (the POS(k) heuristic and Bitner's hybrid algorithm), batched heuristics (the move-every- k th-access Heuristic, the k -in-a-row heuristic, the k -in-a-batch heuristic) and probabilistic heuristics (the stochastic move-to-front scheme, the stochastic move-to-rear scheme and the deterministic move-to-rear scheme).

We have also provided a formal proof for the time reversibility of the transposition heuristic. To our knowledge, it is the only known complete proof of it and is currently being submitted for publication [28].

Absolute analyses are available only for the move-to-front rule and the transposition rule. It is proven that the asymptotic search cost of the transposition rule is less than or equal to that of the move-to-front rule for every probability distribution. However the move-to-front rule converges much faster than the transposition rule, and has been shown to be better in some real applications than the transposition rule. The move-ahead- k rule is conjectured to be in between the move-to-front rule and the transposition rule, yet there is no theoretical results for the best value of k . Hybrid algorithms tend to incorporate the best features of both move-to-front rule and transposition rule, but no theoretical evidence is available to support this. Batched algorithms are used in conjunction with permutation algorithms for the purpose of slowing down the convergence of permutation algorithms and reducing the

time spent reordering the list at the cost of extra memory space. Analysis available is mainly on those combined with either the move-to-front heuristic and the transposition heuristic. Probabilistic algorithms reported in the literature are proven to be computationally more efficient and more accurate in terms of the Markov chain representing the scheme being absorbing instead of ergodic. However the code is more complex than the code for pure permutation algorithms, and they all require extra memory space. A general absolute analyses (direct bounds) for all the other ergodic schemes must be done before realistic comparisons can be made.

Almost all average-case analyses are based on the assumption that accesses are made independently from one another, (or no locality in the access sequence,) which is not practical in many applications. Relaxing the assumption of independence for analysis of permutation algorithms is still much of an open question. We will discuss this problem again in Chapter 5.

In the next chapter, we shall present and analyze two new memory-free self-organizing linear search algorithms, the swap-with-parent heuristic and the move-to-parent heuristic.

Chapter 3

The Swap-with-Parent Heuristic

Self organizing sequential search list has been in the literature for thirty years and as we have seen in Chapter 2, numerous algorithms have been proposed during that time. We notice that all of the reordering rules used in self-organizing linear search algorithms involve moving the accessed record forward (or conceptually backward) by various distances, either constant or based on the location of the record or past events. Notice that there is a common drawback to all the algorithms in the literature, that is, they all fail to consider the size of the list when reorganizing it. The two extremes are the well-known and most commonly analyzed algorithms, the move-to-front rule and the transposition rule. The move-to-front rule simply moves the accessed record to the front of the list and all the records before the accessed record back one position; as opposed to this, the transposition rule moves the accessed record one position ahead by changing places with the record just ahead of it, regardless of the size of the list. Move-ahead- k heuristic attempts to incorporate the best features of the move-to-front rule and the transposition rule by moving the accessed record forward a fixed number of positions (constant distances). Although Hester and Hirschberg [16] point out that the move-ahead- k rule can be generalized to move a percentage of the distance, no known solutions or analysis have been published to our knowledge. Other hybrid algorithms and all batched algorithms are used in conjunction with permutation algorithms, and therefore cannot avoid the drawbacks which permutation algorithms possess. Probabilistic algorithms move

the accessed record, upon certain conditions, either to the front of the list or to the rear of the list, again ignoring the size of the list.

In this chapter we shall present two new memory-free self-organizing sequential search algorithms, both of which take into account the size of the list when reorganizing it. The first algorithm is called *swap-with-parent* (SWP) and the second is called *move-to-parent* (MTP). Under the *swap-with-parent* heuristic, the accessed record gets exchanged with its “parent” (considering the list as a heap structure with no ordering constraints between parents and their children), and all the other records stay unchanged. Instead of swapping the accessed record with its “parent”, the *move-to-parent* heuristic moves the accessed record to its parent’s position and shifts the parent and all the records between the accessed record and its parent back one position.

In this chapter, we shall show that under the swap-with-parent heuristic, the Markov chain representing the scheme is *time reversible*. Using this property, we further derive an expression for the average search cost of the swap-with-parent heuristic. Also because of this property, we can derive expressions for the scheme which generalize the transposition rule. However, the swap-with-parent rule is no better than the transposition rule. Although we can not formally prove that the swap-with-parent rule is superior than the move-to-front rule, we conjecture that it costs no more than the move-to-front rule and its convergence is intermediate to the move-to-front rule and the transposition rule.

For the performance of the move-to-parent rule, empirical comparison shows that it lies between the move-to-front heuristic and the transposition heuristic, and that it is better than the swap-with-parent rule.

3.1 Introduction

Unlike all the other schemes in the literature, the *swap-with-parent* heuristic actually takes into account the size of the list when reorganizing it. Like the *move-to-front*

Figure 3.1: A list viewed as a “heap”.

rule and the *transposition* rule, it is memoryless. The scheme is actually a compromise between the tradeoffs of the move-to-front rule and the transposition rule.

The idea of the scheme comes from self organizing binary search tree operations [8] where the accessed record, under certain conditions, is rotated upwards to its parent level. Since we don't utilize lexicographic ordering, we will not perform rotation operations in the swap-with-parent scheme, but simply exchange the accessed record with its parent. For any given list of records $\{R_1, R_2, \dots, R_n\}$, we can conceptually construct a *heap structure* with *no* ordering constraints between parents and their children. For example, given a list of elements $\{7, 3, 10, 16, 14, 8, 9, 2, 1, 4\}$, the corresponding “heap” structure with *no ordering constraints* is shown in Figure 3.1.

Note that in the context of data structures, the term “heap” is defined as an array object that can be viewed as a complete binary tree with the exception that the leaf level may not be completely filled (only filled from the left up to a point). Heaps also satisfy the heap property that the value of every parent is greater than or equal to the values of its children [10]. However because we ignore lexicographic ordering, our “heap” data structure *does not* have the heap property, and so we shall refer to this structure as a *sequential_heap* or *s_heap*. Throughout this thesis, whenever we refer to heaps, we shall mean the structure defined here.

Figure 3.2: A s_heap of height 3.

The root of the s_heap has index 1, and given the index i of a node, the index of its parent, $\text{Parent}(i)$, can be computed simply:

$$\text{Parent}(i) = \lfloor i/2 \rfloor. \quad (3.1)$$

We define the *depth* of node i , $\text{depth}(i)$, in an s_heap to be the length of the path from the root to the node i as shown in Figure 3.2, and we define the *height* of the s_heap to be the depth of its leaf nodes. Since an s_heap of n elements can be viewed as a complete binary tree, its height is $\lfloor \lg n \rfloor$, and the depth of node i , $\text{depth}(i)$, is,

$$\text{depth}(i) = \lfloor \lg i \rfloor. \quad (3.2)$$

Suppose we have a set of n records $\{R_1, R_2, \dots, R_n\}$ which are in an arbitrary order π , so that R_i is in position $\pi(i)$ for $1 \leq i \leq n$. Using the *swap-with-parent* heuristic, whenever a record R_i is found in position $\pi(i)$, the list is rearranged by exchanging the positions of R_i and its parent which is in position $\lfloor \pi(i)/2 \rfloor$ and leaving all the other records untouched; if R_i heads the list nothing is done. Observe, first of all, a powerful property that unlike the transposition rule, it takes as few as $\lfloor \lg n \rfloor$ steps for a frequently accessed record to be moved from the back of the list to the front (root) of the list as opposed to n accesses needed for the transposition rule.

Figure 3.3: Operations of swapping with parent.

Figure 3.4: State diagram for a Markov chain using the SWP rule.

By way of example, consider a list of 10 elements $\{7, 3, 10, 16, 14, 8, 9, 2, 1, 4\}$, using the swap-with-parent rule, Figure 3.3 shows the changes on the orders of the list after elements 16 and 1 are accessed consecutively.

The scheme avoids the slow convergence problem which the transposition rule has. However it may still tend to make bigger mistakes (like the move-to-front rule) by moving a record to the front about half way of its current position on the basis of a single access comparing to the transposition rule. But the “mistakes” too are far more conservative. We shall look at its performance in the next section.

3.2 The Performance of the SWP Heuristic

Let us consider a list of three records $\{a, b, c\}$ with respective access probabilities of $\{s_a, s_b, s_c\}$. Figure 3.4 shows the state diagram for the underlying Markov chain resulted from using the swap-with-parent rule. The reader will profit by comparing this diagram with the analogous diagram for the transposition rule shown in Figure 2.3.

Observing the paths from state 1 to itself, one path is found if we are looking forward (clockwise), which is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 1$, and the reversed path

can be seen if we are looking backward (counter-clockwise), which is $1 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$. This gives us an indication that the Markov chain shown in the diagram is time reversible. This is actually true for all Markov chains derived from the swap-with-parent heuristic as shown in the following theorem.

Theorem 3.1 *For a given set of records $\{R_1, R_2, \dots, R_n\}$ with respective access probabilities $\{s_1, s_2, \dots, s_n\}$, the Markov chain which results from using the swap-with-parent heuristic is time reversible. That is, under the swap-with-parent heuristic the stationary probabilities obey:*

$$\frac{P\{R_{i_1} \dots R_{i_j} \dots R_{i_{2j}} \dots R_{i_n}\}}{P\{R_{i_1} \dots R_{i_{2j}} \dots R_{i_j} \dots R_{i_n}\}} = \frac{s_{i_j}}{s_{i_{2j}}} \quad (3.3)$$

and

$$\frac{P\{R_{i_1} \dots R_{i_j} \dots R_{i_{2j+1}} \dots R_{i_n}\}}{P\{R_{i_1} \dots R_{i_{2j+1}} \dots R_{i_j} \dots R_{i_n}\}} = \frac{s_{i_j}}{s_{i_{2j+1}}} \quad (3.4)$$

for $1 \leq j < n$ if $s_k \neq 0$ for $1 \leq k \leq n$. $\square$

Like Theorem 2.10, the analogous theorem for the transposition rule, there are two ways to prove this theorem, the constructive way and the non-constructive way. We have presented the constructive proof for Theorem 2.10. The idea of the constructive proof of the above theorem is essentially the same as the one for Theorem 2.10, therefore we will not give the constructive way of proof to the above theorem. Instead, we will present the non-constructive way of proof which is similar to the one Rivest had for the transposition rule.

Proof of Theorem 3.1:

Since the access probabilities are stationary, and from Equation 2.7, we know that the stationary probabilities of the system satisfy

$$M^T * \Pi = \Pi$$

where M is the transition matrix for the corresponding Markov chain and $\Pi = (P\{\pi_1\} \dots P\{\pi_n\})^T$ is the stationary probability vector. Therefore,

$$\begin{aligned}
P\{R_{i_1} \dots R_{i_n}\} &= s_{i_1} P\{R_{i_1} \dots R_{i_n}\} \\
&+ \sum_{1 \leq j \leq n_1} s_{i_j} P\{R_{i_1} \dots R_{i_{2j}} \dots R_{i_j} \dots R_{i_n}\} \\
&+ \sum_{1 \leq j \leq n_2} s_{i_j} P\{R_{i_1} \dots R_{i_{2j+1}} \dots R_{i_j} \dots R_{i_n}\} \quad (3.5)
\end{aligned}$$

where $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lfloor (n-1)/2 \rfloor$.

Since any sequence of swapping from some ordering π to another specific ordering π' will always yield the same ratio $P\{\pi\}/P\{\pi'\}$, therefore the stationary distribution is unique. Applying Equation 3.3 and Equation 3.4 to Equation 3.5, we have

$$\begin{aligned}
P\{R_{i_1} \dots R_{i_n}\} &= s_{i_1} P\{R_{i_1} \dots R_{i_n}\} + \sum_{1 \leq j \leq n_1} s_{i_j} \left(\frac{s_{i_{2j}}}{s_{i_j}} \right) P\{R_{i_1} \dots R_{i_n}\} \\
&+ \sum_{1 \leq j \leq n_2} s_{i_j} \left(\frac{s_{i_{2j+1}}}{s_{i_j}} \right) P\{R_{i_1} \dots R_{i_n}\} \\
&= s_{i_1} P\{R_{i_1} \dots R_{i_n}\} + \sum_{1 \leq j \leq n_1} s_{i_{2j}} P\{R_{i_1} \dots R_{i_n}\} \\
&+ \sum_{1 \leq j \leq n_2} s_{i_{2j+1}} P\{R_{i_1} \dots R_{i_n}\} \\
&= P\{R_{i_1} \dots R_{i_n}\} \left(s_{i_1} + \sum_{1 \leq j \leq n_1} s_{i_{2j}} + \sum_{1 \leq j \leq n_2} s_{i_{2j+1}} \right) \\
&= P\{R_{i_1} \dots R_{i_n}\} \sum_{1 \leq j \leq n} s_{i_j} \\
&= P\{R_{i_1} \dots R_{i_n}\}
\end{aligned}$$

which shows that the stationary probabilities must satisfy Equation 3.3 and 3.4. Hence the result. $\square$

The reader should observe that the time reversibility of the swap-with-parent follows as a *consequence* of the above proof. But if one looks at the problem by studying the state transitions instead of the asymptotic probabilities, we can (as in Theorem 2.10) again show the time reversibility of the swap-with-parent rule.

Let us look at the path from state (a, b, c) to itself shown in Figure 3.4 again. It is illustrated using the `s_heap` below.

The product of the transition probabilities in the forward direction is

$$s_b s_c s_a s_b s_c s_a = s_a^2 s_b^2 s_c^2,$$

whereas in the reverse direction, it is

$$s_c s_b s_a s_c s_b s_a = s_a^2 s_b^2 s_c^2.$$

The general result of the time reversibility follows in much the same manner as the proof of Theorem 2.10. The main loop of six states for three records follows transitions as above, and the inductive step would consider how the fourth (k th) element would move to position 2 ($\lfloor k/2 \rfloor$). By following the proof it can be seen that the Markov chain resulting from the swap-with-parent is indeed time reversible. The details of this proof will be clear in Chapter 4 when we derive the time reversibility of generalized swap-with-parent schemes.

Theorem 3.1 shows a relationship between the stationary probabilities of two states where the list orderings differ by two records which have a parent/child relationship. Corollary 3.1 below shows the relationship between the stationary probabilities of two states where the two swapped records do not have parent/child relationship. In a more general case, for a given set of records $\{R_1, \dots, R_n\}$ with respective access probabilities $\{s_1, \dots, s_n\}$, we are interested in the relationship between the stationary probabilities of two *arbitrary* states. This is given by Corollary 3.2.

Corollary 3.1 *Under the swap-with-parent rule the stationary probabilities obey:*

$$\frac{P\{\dots, \mathbf{R}_i, R_{i_1}, \dots, R_{i_k}, \mathbf{R}_j, \dots\}}{P\{\dots, \mathbf{R}_j, R_{i_1}, \dots, R_{i_k}, \mathbf{R}_i, \dots\}} = \left(\frac{s_i}{s_j}\right)^{k_{ij}} \quad (3.6)$$

where $k_{ij} = |\lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor|$, for $1 \leq i, j \leq n$ if $s_j \neq 0$.

Proof:

This follows easily by successive swapping using Equation (3.3) and (3.4). $\square$

It is advantageous to see the results shown above through an example. Considering a list of 10 elements $\{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9, R_{10}\}$. Figure 3.5 illustrates the sequence of transformations from state π_s to state π_t , where $\pi_s = \{R_1, R_2, \mathbf{R}_3, R_4, \dots, R_8, \mathbf{R}_9, R_{10}\}$ and $\pi_t = \{R_1, R_2, \mathbf{R}_9, R_4, \dots, R_8, \mathbf{R}_3, R_{10}\}$.

To go from state π_s to state π_t , swapping element R_9 with its parent until it reaches the root and then swapping R_3 with it, now R_9 is in the position of R_3 . Then swapping R_1 with R_3 , then R_2 with R_3 , then R_4 with R_3 . We have reached state π_t . Let π_i denote all the states during the transformation, then

$$\begin{aligned}
 \pi_s &= \{R_1, R_2, \mathbf{R}_3, R_4, R_5, R_6, R_7, R_8, \mathbf{R}_9, R_{10}\}, \\
 \pi_1 &= \{R_1, R_2, R_3, \mathbf{R}_9, R_5, R_6, R_7, R_8, \mathbf{R}_4, R_{10}\}, \\
 \pi_2 &= \{R_1, \mathbf{R}_9, R_3, \mathbf{R}_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
 \pi_3 &= \{\mathbf{R}_9, \mathbf{R}_1, R_3, R_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
 \pi_4 &= \{\mathbf{R}_3, R_1, \mathbf{R}_9, R_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
 \pi_5 &= \{\mathbf{R}_1, \mathbf{R}_3, R_9, R_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
 \pi_6 &= \{R_1, \mathbf{R}_2, R_9, \mathbf{R}_3, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
 \pi_t &= \{R_1, R_2, \mathbf{R}_9, R_4, R_5, R_6, R_7, R_8, \mathbf{R}_3, R_{10}\}.
 \end{aligned}$$

From Equation (3.3), we have

$$\begin{aligned}
 P\{\pi_s\} &= (s_4 / s_9) P\{\pi_1\}, \\
 P\{\pi_1\} &= (s_2 / s_9) P\{\pi_2\}, \\
 P\{\pi_2\} &= (s_1 / s_9) P\{\pi_3\}, \\
 P\{\pi_3\} &= (s_9 / s_3) P\{\pi_4\}, \\
 P\{\pi_4\} &= (s_3 / s_1) P\{\pi_5\}, \\
 P\{\pi_5\} &= (s_3 / s_2) P\{\pi_6\}, \\
 P\{\pi_6\} &= (s_3 / s_4) P\{\pi_t\}.
 \end{aligned}$$

Figure 3.5: Transformations from state π_s to π_t through $\pi_1, \dots, \pi_6$.

Therefore,

$$P\{\pi_s\} = \frac{s_4}{s_9} \times \frac{s_2}{s_9} \times \frac{s_1}{s_9} \times \frac{s_9}{s_3} \times \frac{s_3}{s_1} \times \frac{s_3}{s_2} \times \frac{s_3}{s_4} \times P\{\pi_t\} = \left(\frac{s_3}{s_9}\right)^2 P\{\pi_t\}. \quad (3.7)$$

Since $\text{depth}(R_3) = 1$ and $\text{depth}(R_9) = 3$, the difference between the depths of R_3 and R_9 is 2. Therefore Equation (3.7) above verifies the result shown in Corollary 3.1.

Recall Corollary 2.3 in chapter 2 about the relationship between the stationary probabilities of two arbitrary states under the transposition heuristic, we can now derive a similar result for the swap-with-parent heuristic.

Corollary 3.2 *Under the swap-with-parent heuristic, the stationary probabilities between any two arbitrary states π_s and π_t obey:*

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_t, \pi_s)} \quad (3.8)$$

where $\xi_i(\pi_t, \pi_s) = \lfloor \lg(\pi_t(i)) \rfloor - \lfloor \lg(\pi_s(i)) \rfloor$ for $1 \leq i \leq n$ is the difference between the depths of R_i in ordering π_s and π_t .

Proof:

Since the Markov chain representing the scheme is time reversible, so we can reach any state from any other by a sequence of one or more transformations. Assuming that the sequence of intermediate states which transforms state π_s to state π_t are $\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_k}$ for $1 \leq k \leq (n! - 2)$, we can write

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \frac{P\{\pi_s\}}{P\{\pi_{i_1}\}} \times \frac{P\{\pi_{i_1}\}}{P\{\pi_{i_2}\}} \times \dots \times \frac{P\{\pi_{i_{k-1}}\}}{P\{\pi_{i_k}\}} \times \frac{P\{\pi_{i_k}\}}{P\{\pi_t\}}.$$

Corollary 3.1 gives the expression for each of the fractions on the right hand side of the above equation. Thus

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{u,v} \left(\frac{s_u}{s_v}\right)^k \quad (3.9)$$

for some u and v where $1 \leq u, v \leq n$ and $0 \leq k \leq \lfloor \lg n \rfloor$.

Note that each transformation from state π_i to state $\pi_{i,j+1}$ swaps record R_u and record R_v . Note also that if the *depth* of a record in π_s is shallower than its *depth* in π_t , then its access probability will appear as a numerator in the expression shown

by Equation (3.9) above. Similarly, if the *depth* of a record in π_s is deeper than its *depth* in π_t , its access probability will appear as a denominator in Equation (3.9).

The number of times that record R_i will be *swapped* during the transformations from π_s to π_t is the difference in depths from when R_i is in π_s to it being in π_t , i.e., $\xi_i(\pi_t, \pi_s) = \lfloor \lg(\pi_t(i)) \rfloor - \lfloor \lg(\pi_s(i)) \rfloor$. If the depth of record R_i in π_s is shallower than its depth in π_t , then the value of $\xi_i(\pi_t, \pi_s)$ will be positive, whereas if its depth in π_s is deeper than its depth in π_t , the value of $\xi_i(\pi_t, \pi_s)$ will be negative. If the depth of R_i in π_s is the same as its depth in π_t , then the value of $\xi_i(\pi_t, \pi_s)$ is 0. Therefore,

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_t, \pi_s)},$$

and the result is proved. $\square$

The procedure of the proof is now clarified by means of an example, which illustrates how we actually transform an arbitrary state to another through a sequence of swaps. Suppose that $\pi_s = \{3, 4, 9, 8, 7, 5, 2, 1, 6\}$ and $\pi_t = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. To transform from π_s to π_t , one path is $\pi_s \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \pi_3 \rightarrow \pi_4 \rightarrow \pi_5 \rightarrow \pi_6 \rightarrow \pi_7 \rightarrow \pi_t$ as shown in Figure 3.6.

Using the stationary probabilities, we see that

$$\begin{aligned} \frac{P\{\pi_s\}}{P\{\pi_t\}} &= \frac{P\{\pi_s\}}{P\{\pi_1\}} \times \frac{P\{\pi_1\}}{P\{\pi_2\}} \times \frac{P\{\pi_3\}}{P\{\pi_4\}} \times \frac{P\{\pi_5\}}{P\{\pi_6\}} \times \frac{P\{\pi_7\}}{P\{\pi_t\}} \\ &= \left(\frac{s_8}{s_1}\right) \left(\frac{s_9}{s_6}\right)^2 \left(\frac{s_4}{s_1}\right) \left(\frac{s_6}{s_5}\right) \left(\frac{s_3}{s_5}\right) \left(\frac{s_5}{s_1}\right) \left(\frac{s_7}{s_2}\right)^0 \left(\frac{s_5}{s_2}\right) \\ &= (s_3 s_4 s_8 s_9^2) / (s_2 s_6 s_1^3) \\ &= s_1^{-3} s_2^{-1} s_3^1 s_4^1 s_5^0 s_6^{-1} s_7^0 s_8^1 s_9^2. \end{aligned}$$

It can be easily verified that Corollary 3.2 produces the same result.

Rivest gave an expression for the asymptotic search cost of the transposition rule shown in Theorem 2.11. We can now give an analogous result for the average search cost of the swap-with-parent rule.

Figure 3.6: Transformations from state π_s to π_t through $\pi_1, \dots, \pi_7$.

Theorem 3.2 Let π_0 denote the identity permutation on n records $\pi_0(i) = i$ for $1 \leq i \leq n$, which is the optimal ordering under our assumption that $s_i \geq s_{i+1}$ for $1 \leq i \leq n$. Let $\xi_i(\pi_0, \pi)$ denote the quantity $(\lfloor \lg(i) \rfloor - \lfloor \lg(\pi(i)) \rfloor)$ for any permutation π and $1 \leq i \leq n$, which is the difference between the depths of R_i in the optimal ordering and in ordering π . Then the asymptotic search cost for the swap-with-parent heuristic is

$$P\{\pi_0\} \sum_{\text{all } \pi} \left(\prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \sum_{1 \leq j \leq n} s_j \pi(j) \right) \quad (3.10)$$

where

$$P\{\pi_0\} = \left(\sum_{\text{all } \pi} \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \right)^{-1}. \quad (3.11)$$

Proof:

From Equation (2.6), the average search cost of an algorithm can be calculated by

$$\sum_{\text{all } \pi} \left(P\{\pi\} \sum_{1 \leq j \leq n} s_j \pi(j) \right).$$

Using Corollary 3.2, we have,

$$P\{\pi\} = P\{\pi_0\} \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \quad \text{for all } \pi. \quad (3.12)$$

Consequently

Average search cost for swap-with-parent

$$\begin{aligned} &= \sum_{\text{all } \pi} \left(\left(P\{\pi_0\} \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right) \\ &= P\{\pi_0\} \sum_{\text{all } \pi} \left(\left(\prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right), \end{aligned}$$

thus Equation (3.10) is proved.

Since

$$\sum_{\text{all } \pi} P\{\pi\} = 1,$$

applying Equation (3.12), we have

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

which proves Equation (3.11). $\square$

Using time reversibility, we can now prove the following results.

Theorem 3.3 *The swap-with-parent heuristic is expedient since*

$$P\{R_j \text{ precedes } R_i\}_{SWP} > \frac{1}{2} \quad \text{if } s_j > s_i.$$

Proof:

By Equation (3.6) we have

$$P\{\dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots\} = \left(\frac{s_i}{s_j}\right)^{k_{ij}} P\{\dots, R_j, R_{i_1}, \dots, R_{i_k}, R_i, \dots\}$$

where $k_{ij} = |\text{depth}(i) - \text{depth}(j)| \geq 0$.

Since $s_j > s_i$, therefore $\left(\frac{s_i}{s_j}\right)^{k_{ij}} < \left(\frac{s_i}{s_j}\right)^0 = 1$. The above equation implies that

$$P\{\dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots\} < P\{\dots, R_j, R_{i_1}, \dots, R_{i_k}, R_i, \dots\},$$

summing over all states for which R_j precedes R_i using the above, we have

$$\sum_{\text{state } \pi} P\{\pi \mid R_i \text{ precedes } R_j \text{ in } \pi\} < \sum_{\text{state } \pi} P\{\pi \mid R_j \text{ precedes } R_i \text{ in } \pi\},$$

which implies that

$$P\{R_i \text{ precedes } R_j\} < P\{R_j \text{ precedes } R_i\}.$$

Since $P\{R_i \text{ precedes } R_j\} = 1 - P\{R_j \text{ precedes } R_i\}$, it yields

$$P\{R_j \text{ precedes } R_i\} > \frac{1}{2},$$

which completes the proof. $\square$

Theorem 3.4 *For any two records R_i and R_j , let $k_{ij} = | \lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor |$, for $1 \leq i, j \leq n$. Then under the swap-with-parent heuristic we have:*

$$P \{ R_j \text{ precedes } R_i \mid k_{ij} = d \} = \frac{1}{1 + \left(\frac{s_i}{s_j} \right)^d} = \frac{s_j^d}{s_i^d + s_j^d}. \quad (3.13)$$

Proof:

By Equation (3.6) we have

$$P \{ \dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots \} = \left(\frac{s_i}{s_j} \right)^{k_{ij}} P \{ \dots, R_j, R_{i_1}, \dots, R_{i_k}, R_i, \dots \},$$

summing over all states for which R_j precedes R_i and $k_{ij} = d$ using the above, we have

$$\sum_{\text{state } \pi} P \{ \pi \mid R_i \text{ precedes } R_j \text{ in } \pi \text{ and } k_{ij} = d \} = \left(\frac{s_i}{s_j} \right)^d \sum_{\text{state } \pi} P \{ \pi \mid R_j \text{ precedes } R_i \text{ in } \pi \text{ and } k_{ij} = d \},$$

which implies that

$$P \{ R_i \text{ precedes } R_j \mid k_{ij} = d \} = \left(\frac{s_i}{s_j} \right)^d P \{ R_j \text{ precedes } R_i \mid k_{ij} = d \}.$$

Since $P \{ R_i \text{ precedes } R_j \} = 1 - P \{ R_j \text{ precedes } R_i \}$, it yields

$$P \{ R_j \text{ precedes } R_i \mid k_{ij} = d \} = \frac{1}{1 + \left(\frac{s_i}{s_j} \right)^d} = \frac{s_j^d}{s_i^d + s_j^d}.$$

Hence the result. $\square$

Using Theorem 3.1, it is very easy to show that the swap-with-parent rule is not as good as the transposition rule in terms of the asymptotic cost. However we conjecture that it is better than the move-to-front rule when $n > 3$. Experimental results shown in Table 3.1 support our conjecture. (See section 3.5 for the details of the experiments.)

We shall now present various enumeration arguments which lead us to believe that the swap-with-parent rule is better than the move-to-front rule. We know from

Theorem 2.1 that to prove rule A is better than rule B , it is sufficient to prove that the asymptotic probability that record R_j precedes record R_i under rule A is greater than that under rule B given that $s_j > s_i$ for all $j \neq i$. Therefore, to prove that the swap-with-parent rule is superior to the move-to-front rule, we need to prove that

$$P\{R_j \text{ precedes } R_i\}_{SWP} > P\{R_j \text{ precedes } R_i\}_{MTF} = \frac{s_j}{s_i + s_j}$$

given that $s_j > s_i$ for $1 \leq i, j \leq n$ and $j \neq i$.

By Theorem 3.4, we know that

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d\}_{SWP} = \frac{s_j^d}{s_i^d + s_j^d} \quad \text{for } 0 \leq d \leq \lfloor \lg n \rfloor.$$

Notice that if $s_j > s_i$, then

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d = 0\}_{SWP} = \frac{1}{1+1} < \frac{1}{1 + \left(\frac{s_i}{s_j}\right)} = \frac{s_j}{s_i + s_j}. \quad (3.14)$$

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d = 1\}_{SWP} = \frac{s_j}{s_i + s_j}. \quad (3.15)$$

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d > 1\}_{SWP} = \frac{1}{1 + \left(\frac{s_i}{s_j}\right)^d} > \frac{1}{1 + \left(\frac{s_i}{s_j}\right)} = \frac{s_j}{s_i + s_j}. \quad (3.16)$$

Invoking the law of total probability, we have that in the average case

$$P\{R_j \text{ precedes } R_i\}_{SWP} = \sum_{d=0}^{\lg n} (P\{R_j \text{ precedes } R_i \mid k_{ij} = d\} * P\{k_{ij} = d\}). \quad (3.17)$$

Unfortunately we are unable to find a bound or an expression for the value of $P\{k_{ij} = d\}$, because it is related in a fairly complex way to both the access probabilities of the records R_i and R_j , and the required swapping operations. Consequently we have not, as yet, formally proven that the swap-with-parent rule is better than the move-to-front rule. However, alluding to Equation (3.14), (3.15) and (3.16) above, we know that to make $P\{R_j \text{ precedes } R_i\}_{SWP} > (s_j/(s_i + s_j))$ given that $s_j > s_i$, we would want to make $P\{k_{ij} = 0\}$ as small as possible and $P\{k_{ij} = d > 0\}$ as large as possible. That is to say that given two records R_i and R_j with $s_j > s_i$, under the swap-with-parent heuristic, we hope that the *asymptotic*

probability of these two records being in the same level in the corresponding s.heap is much (we don't know how much) smaller than that of them not being in the same level.

Theorem 3.5 below seems to indicate that this is true, even though the results shown in the theorem ignore the access probabilities and the required reordering rule.

Theorem 3.5 (See Appendix A for the proof.) *Consider a complete binary tree with n records $\{R_1, \dots, R_n\}$ which are in a random order, (note that $n = 2^q - 1$ for some $q > 1$.) For any two records R_i and R_j , let $k_{ij} = |\lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor|$ for $1 \leq i, j \leq n$. Then the following equations hold:*

$$\begin{aligned} P\{k_{ij} = d = 0\} &= \frac{1}{3}, \\ P\{k_{ij} = d = 1\} &= \frac{1}{3} + \frac{1}{n}, \\ P\{k_{ij} = d > 1\} &= \frac{1}{3} - \frac{1}{n}. \end{aligned}$$

□

Theorem 3.5 tells us that when $n > 3$,

$$P\{k_{ij} = 0\} = \frac{1}{3} \quad \text{and} \quad P\{k_{ij} > 0\} = \frac{2}{3}. \quad (3.18)$$

Notice that the results shown in the above equations are based on the condition that the tree is complete, i.e., $n = 2^q - 1$ for some $q > 1$. If this condition is not true, then we have the following corollary.

Corollary 3.3 (See Appendix B for the proof.) *Consider a incomplete binary tree (the leaf level is not full) with n records $\{R_1, \dots, R_n\}$ which are in a random order, (i.e., $n \neq 2^q - 1$ for some $q > 1$.) Then for any two records R_i and R_j , if $k_{ij} = |\lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor|$ for $1 \leq i, j \leq n$, the following inequalities hold.*

$$P\{k_{ij} = 0\} < \frac{1}{3} \quad \text{and} \quad P\{k_{ij} > 0\} > \frac{2}{3}. \quad (3.19)$$

□

Therefore, 3.18 and 3.19 together informally seem to imply that the number of cases in which the swap-with-parent rule performs better than the move-to-front rule exceeds the number of cases in which it does not. Thus we conjecture that

$$P\{R_j \text{ precedes } R_i\}_{SWP} > P\{R_j \text{ precedes } R_i\}_{MTF} \quad \text{if } s_j > s_i.$$

The formal conjecture is given below.

Conjecture 3.1 *The asymptotic search cost of the swap-with-parent heuristic is less than or equal to that of the move-to-front heuristic when $n > 3$.* $\square$

The worst case cost of the swap-with-parent rule is unknown. If locality is allowed in the access sequence, it may be much worse than the move-to-front rule. For example, if we encounter the same situation that happens in the transposition rule, where two records are alternately accessed many times, they will continue to exchange places without advancing toward the front of the list. This same phenomenon will also happen with the swap-with-parent rule.

The rate of convergence of the swap-with-parent rule is open, no general formula is currently known for the *overwork* measure of convergence. Intuition tells us that it should lie between the move-to-front rule and the transposition rule.

3.3 Implementation Details

Implementation of the swap-with-parent heuristic is very simple. It can be done with no additional pointers. The parent of the accessed record can be easily computed by using Equation 3.1, i.e.,

$$\text{Parent}(i) = \lfloor i/2 \rfloor.$$

Thus the array implementation is straightforward. Since each record in the list can be referred to by its index directly, whenever record R_i is found in position $\pi(i)$, its parent must be in position $\lfloor \pi(i)/2 \rfloor$. Simply switching the indices of these two records yields the actual swap-with-parent operation.

Figure 3.7: An s_heap of 15 elements.

The linked list implementation can be done by having a temporary variable 'parent' which is initially pointing at the front of the list, and then moved to the next position whenever the current pointer is moved twice. Whenever record R_i is found, 'swap-with-parent' simply *exchanges* the contents of the current pointer (R_i) and the parent pointer (parent of R_i).

3.4 Drawbacks of the SWP Heuristic

If some of the records have access probabilities zero or a negligible small value, then a potential drawback would be present when using the swap-with-parent heuristic to organize a list.

For example, if we have a list of 15 elements which are initially ordered as shown in Figure 3.7. Assuming element 2 has access probability 0, then eventually it will fall to the position 8, 9, 10 or 11, and it will *never* be moved again. Therefore the cost of future accesses to the elements behind element 2 in the leaf level will be increased, and the situation will never get better. The worst case happens when one of the elements with 0 access probabilities falls to the leftmost position on the leaf level since about half of the elements reside on the leaf level, and so the cost of access to these elements will always be sub-optimal. Therefore the elements with zero (or

negligible) access probabilities will prove to be a drag to the entire access strategy. Indeed, in a more general setting, the elements with lower access probabilities may still cause a similar problem once they are at the leaf level.

One solution to this problem is to shift all the records between the accessed record and its parent (including the parent) back one position, instead of swapping the accessed record and its parent and leaving all the records in between unchanged. We call this scheme *move-to-parent* scheme. It is presented in the next section.

3.5 A Modification to the SWP Heuristic - the Move-to-parent Heuristic

Under the *move-to-parent* (MTP) heuristic, the accessed record in the `s_heap` is moved to its parent's position and all the records after the parent record (including the parent record) and before the accessed record are shifted back one position. This is actually a *move-head-k* heuristic where k is not a constant but a function of the current position of the accessed record, or more explicitly its depth in the `s_heap`.

The move-to-parent rule avoids the drawbacks of many other algorithms. As we have seen in the previous section, it avoids the drawback caused by accessing records with zero or small access probability in the swap-with-parent heuristic. It also avoids the large asymptotic search cost of the move-to-front heuristic as every time a record with low access probability is accessed, it is moved only half way to the front of the list instead of all the way to the front of the list, which increases the costs of future accesses to only half of the records that the move-to-front rule does. It obviously avoids the slow convergence problem that the transposition rule has.

The most important of all is that when *locality* is considered, the move-to-parent heuristic avoids all the drawbacks related to locality that the move-to-front, transposition and swap-with-parent rules may have. Bitner's hybrid algorithm tends to control the locality problem, however, as we pointed out, it is difficult to know when to switch from move-to-front to transposition. Since the distance that a record is moved forward is dynamically changing under the move-to-parent heuristic, and all

the records in between are moved back one position, it responds much better to the effects of locality. We shall address this problem again in much more detail in Chapter 5.

Absolute analysis is currently not available for this scheme. We conjecture that its average search cost is in between that of the transposition rule and the move-to-front rule when $n > 3$. Note that when $n \leq 3$, the move-to-parent rule is exactly the same as the move-to-front rule. Empirical evidence shown in Table 3.1 supports our conjecture.

Calculating the average search cost from Equation 2.6 is still possible. For the simple case when $n \leq 3$, the scheme is the same as the move-to-front rule. However for the case where $n = 4$, the transition matrix will be 24×24 , and a lot of tedious work is required to solve a set of 24 equations. For lists of size greater than or equal to five, such computations are prohibitive.

Implementation of the move-to-parent scheme is similar to that of the swap-with-parent scheme. It is achieved by keeping a temporary variable which points at the parent of the current accessed record.

3.6 Empirical Results

Simulating the behaviors of the algorithms is another way to compare them. In the absence of analytic results, many researchers have used simulations to test their algorithms, where in most cases, access sequences follow some known probability distributions.

A set of experiments was run to compare the asymptotic search cost of the swap-with-parent heuristic, the move-to-parent heuristic, the move-to-front heuristic and the transposition heuristic. The measure of the cost used is the number of probes required to find a record in the given list. Five different types of distributions were used to describe the access sequences:

1. *Zipf's distribution.* The individual probabilities obey:

$$s_i = \frac{1}{i H_n} \quad \text{for } 1 \leq i \leq n$$

where $H_n = \sum_{k=1}^n (1/k)$ is the n th harmonic number.

2. *"80-20" distribution.* The individual probabilities obey:

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

where $H_n^{(1-\theta)} = \sum_{k=1}^n (1/k^{1-\theta})$ is the n th harmonic number of order $(1 - \theta)$.

3. *Lotka's distribution.* The individual probabilities obey:

$$s_i = \frac{1}{i^2 H_n^{(2)}} \quad \text{for } 1 \leq i \leq n$$

where $H_n^{(2)} = \sum_{k=1}^n (1/k^2)$ is the n th harmonic number of order 2.

4. *Exponential distribution.* The individual probabilities obey:

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

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

5. *Linear distribution.* The individual probabilities obey:

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

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

Zipf's distribution is a typical distribution used by many simulations. It was formulated by G. K. Zipf, who observed that the n th most common word in natural language text seems to occur with a frequency inversely proportional to n . He observed the same phenomenon in census tables, when metropolitan areas are ranked in order of decreasing population. Data obeying Zipf's distribution (also called Zipf's law) is generally considered to be a good approximation to real-life data.

“80-20” law is another approximation to realistic distribution. It is the “80-20” rule of thumb that has been commonly observed in commercial applications. This rule states that 80 percent of the transactions deal with the most active 20 percent of a file. The same principle applies to the latter 20 percent, and so 64 percent of the transactions deal with the most active 4 percent, etc. The actual distribution we give above is a simpler distribution which approximately satisfies the 80-20 rule. Note that this probability distribution is very similar to that of Zipf’s law; as θ varies from 1 to 0, the probabilities vary from a uniform distribution to a Zipfian one. Indeed, Zipf found that $\theta \approx \frac{1}{2}$ in the distribution of personal income.

Other probability distributions listed above are also commonly used in many simulations. Numerous experiments were conducted to compare the performance of the various algorithms and for different list sizes. In the interest of brevity, we report only the results for the case when the number of records is 100.

There were 12 parallel experiments conducted for lists with 100 records for each type of probability distributions. In each experiment, 300,000 accesses were performed on the list. Table 3.1 shows the average asymptotic search cost in terms of the number of probes required to locate a record under the move-to-front, swap-with-parent, move-to-parent and the transposition heuristics for each type of probability distributions listed above. In each case, the results shown are the average result of the last 50,000 queries. The first 250,000 queries were utilized to permit the scheme to converge. Table 3.2 shows the *approximate* number of accesses required for the scheme to converge to its steady state (when the cost is within a certain percentage of the asymptotic cost as shown in the table) based on two experiments.

From the simulation results shown in Table 3.1, we can see that the average search cost of the swap-with-parent scheme is no worse than the move-to-front heuristic, and the average search cost of the move-to-parent heuristic is approximately intermediate to the move-to-front heuristic and the transposition heuristic. Table 3.2 shows that the transposition rule is the slowest rule in terms of the number of accesses needed to converge to its steady state whereas the move-to-front rule is the fastest rule. The convergence of the swap-with-parent rule and the move-to-parent

Scheme	Zipf's Distribution	80-20 Distribution	Lotka's Distribution	Exponential Distribution	Linear Distribution
MTF	26.439	31.703	4.484	27.722	41.735
SWP	24.276	29.245	4.235	27.719	41.379
MTP	22.118	27.008	3.771	23.235	38.660
TR	19.717	24.237	3.419	20.357	34.515

Table 3.1: Simulation results (cost) for MTF, SWP, MTP and TR rules.

Scheme	Zipf's Distribution (within 1%)	80-20 Distribution (within 1%)	Lotka's Distribution (within 5%)	Exponential Distribution (within 2%)	Linear Distribution (within 1%)
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 3.2: Simulation results (convergence) for MTF, SWP, MTP and TR rules.

rule is between the move-to-front rule and the transposition rule.

3.7 Conclusion

In this chapter, we have presented two new memory-free self-organizing sequential search algorithms - the swap-with-parent heuristic and the move-to-parent heuristic.

We have shown that under the swap-with-parent heuristic, the Markov chain representing the scheme is *time reversible*. We then derived an expression for the average search cost of the swap-with-parent heuristic using this property. The result

is similar to that of the transposition rule because of the time reversibility. However the swap-with-parent rule is no better than the transposition rule. We conjecture that it is better than the move-to-front rule and that its convergence is far superior to the transposition rule. Absolute analysis for the move-to-parent rule is not available. We conjecture that its performance is intermediate to the move-to-front rule and the transposition rule, and is better than the swap-with-parent rule. Empirical comparison supports our conjectures.

The difference between these two algorithms and all the other algorithms presented in this thesis is that unlike all the other algorithms, these two algorithms consider the size of the list when they achieve reorganization. Although our analysis assumes no locality, we will show, in Chapter 5, that both these rules are better in some cases when locality is considered.

By closely studying the analysis of the swap-with-parent heuristic, we can show the existence of an entire class of time reversible Markov chains which provides us with a mechanism with which we can design and analyze lists with conceptual tree-like transitions. This is the focus of the next chapter.

Chapter 4

General Time Reversible Markovian Lists

As we have seen in Section 3 of Chapter 2 and Chapter 3 where we analyzed the transposition heuristic and the swap-with-parent heuristic, the fundamental property of the corresponding Markov chain being time reversible greatly simplifies the analysis of the algorithm. The analysis of the swap-with-parent heuristic shows the existence of a class of time reversible Markov chains resulting from performing swaps on “implicit” tree structures which generalize and extend the results concerning the transposition heuristic and the swap-with-parent heuristic.

The first section of this chapter reviews some aspects of time reversibility. Section 2 and 3 present the class of time reversible Markov chains and a generalized swap-with-parent heuristic. Section 4 concludes the chapter.

4.1 Preliminaries on Time Reversible Markov Chains

A brief introduction to *time reversibility* was presented in Section 2.3.1. As we have seen there, some Markov chains have the property that the process behaves in just the same way regardless of whether time is measured forwards or backwards. Kelly [19] made an analogy saying that, “if we take a film of such a process and then run the film backwards the resulting process will be statistically indistinguishable from the original process.” This property is described formally in the following definition.

Definition 4.1 A stochastic process $X(t)$ is time reversible if a sequence of states $(X(t_1), X(t_2), \dots, X(t_n))$ has the same distribution as the reversed sequence $(X(t_n), X(t_{n-1}), \dots, X(t_1))$ for all $t_1, t_2, \dots, t_n$. $\square$

Consider a stationary ergodic Markov chain having transition probabilities M_{ij} and stationary probabilities $P\{\pi_i\}$. The condition for which it is time reversible can be stated that, for all states i and j , the rate at which the process goes from i to j (namely $P\{\pi_i\} M_{ij}$) is equal to the rate at which the process goes from j to i (namely $P\{\pi_j\} M_{ji}$). Explicitly this means that

$$P\{\pi_i\} M_{ij} = P\{\pi_j\} M_{ji} \quad \text{for all } i \neq j. \quad (4.1)$$

This is a necessary condition for a Markov process to be time reversible.

The corresponding sufficient condition can be stated as follows. For a Markov chain $\{M_{ij}\}$, suppose that there exists nonnegative numbers x_i , summing to unity, which satisfy Equation 4.1. Then it follows that the Markov chain is time reversible and that the numbers represent the stationary probabilities. This is because if

$$x_i M_{ij} = x_j M_{ji} \quad \text{for all } i, j \text{ and } \sum_i x_i = 1. \quad (4.2)$$

Then summing over i yields

$$\sum_i x_i M_{ij} = x_j \sum_i M_{ji} = x_j, \quad \sum_i x_i = 1. \quad (4.3)$$

Since the stationary probabilities $P\{\pi_i\}$ are the unique solution of the above, it follows that $x_i = P\{\pi_i\}$ for all i .

Therefore the above argument provides a necessary and sufficient condition for a stationary Markov chain to be time reversible. It is formally stated in the following theorem adapted from Asmussen ([3], p43).

Theorem 4.1 A stationary ergodic Markov chain is time reversible if and only if the transition probabilities M_{ij} and the stationary probabilities $P\{\pi_i\}$ satisfy the following condition

$$P\{\pi_i\} M_{ij} = P\{\pi_j\} M_{ji} \quad (4.4)$$

for all $i \neq j$. $\square$

Recall that we presented another criteria for time reversibility in Theorem 2.9 from Ross [32]. It is restated below for the purpose of completeness.

Theorem 4.2 *A finite ergodic Markov chain for which $M_{st} = 0$ whenever $M_{ts} = 0$ is time reversible if and only if starting in state s , any path back to s has the same probability as the reversed path. That is, if*

$$M_{s,s_1} M_{s_1,s_2} \dots M_{s_k,s} = M_{s,s_k} M_{s_k,s_{k-1}} \dots M_{s_1,s}$$

for all states $s, s_1, \dots, s_k$. □

Using the above theorem, we shall show, in the next section, that any tree structure associated with finite stationary Markov process is time reversible.

4.2 A Class of Time Reversible Markov Chains

Inspired from the result that the Markov chain representing the swap-with-parent heuristic is time reversible, we now follow the avenue of thought that a Markov chain resulting from the “swapping” operation on any tree structure is time reversible. In fact, this result is not totally new. Kelly ([19], p9) proved the following lemma.

Lemma (Adapted from Kelly [19].) *If the graph G associated with a stationary Markov process is a tree, then the process is time reversible.* □

Although Kelly reported this result, he did not demonstrate how to associate a tree with a stationary Markov chain. In this section, we shall give a formal definition for such a tree structure in the self-organizing list application domain and prove the corresponding theorem regarding its time reversibility.

Let T be an arbitrary tree structure with n nodes. Let the ordering of the tree be the sequential order of the nodes in the tree (see Figure 4.1 (a)). At every time instant one of the node in the tree is accessed, with probability s_i , and it is swapped with its parent. Therefore every swap transforms the tree from one ordering to another, and there are total of $n!$ tree orderings. Let the ordering of

Figure 4.1: Tree orderings and transformations on repeated accesses of record "9".

the tree be the states of a system. Since it is possible to reach any state from any other by a sequence of one or more transformations (see Figure 4.1 (b)), the transformations form an irreducible Markov chain with the state at any time being the tree ordering at that time. The transition probabilities M_{st} for the Markov chain are determined by the access probabilities s_i of the nodes and the swapping operation used. Each M_{st} will either be zero or one of the s_i 's. The eigenvector of $M = \{ M_{st} \mid 1 \leq s \leq n! \text{ and } 1 \leq t \leq n! \}$ corresponding to the eigenvalue unity gives the stationary probabilities $P \{ \pi_s \}$ of the states π_s ($1 \leq s \leq n!$) of the system (see also Equation 2.7). We shall call such a process a *tree transformation process*.

Using this tree transformation process we now introduce a generalization of the

transposition and the swap-with-parent heuristic. We shall call it the *swap-with-parent-in-an-ss-tree* (SWPSST) heuristic invoked by what we call a *sequential search trees* (*ss_trees*). Before we go into any details, we shall first define the tree structure.

4.2.1 Sequential Search Trees

In the previous chapter, we introduced the concept of sequential heap or *s_heap* where we view a list as a heap without the heap property. Here we shall define a similar tree structure, the *sequential search tree* or *ss_tree*.

Definition 4.2 *A sequential search tree is an unordered list viewed as a tree. The ordering of the tree is the ordering of the list, and the position of the parent of each node is fully defined in terms of the position of the node itself. Trivially, the parent of the root is the root itself.*

The structure of an *ss_tree* is not a tree, but an unordered list. It is only a tree on a conceptual level (“implicit”) like the *s_heap* we defined in Chapter 3. Therefore there is no lexicographic ordering among the nodes. We cannot perform any tree operations such as *rotation* etc.. Figure 4.2 shows various *ss_trees* given in the following examples.

Example 1.

A *unary ss_tree* is a list viewed as a tree with only one branch. The parent of each node is its previous node. The list operated by the transposition rule is a unary *ss_tree*, and the operation performed on the list is the generalized “swap with parent” of the *ss_tree*. Figure 4.2 (a) shows a list with 5 elements viewed as a unary *ss_tree*.

Example 2.

A *binary ss_tree* is the *s_heap* introduced in Chapter 3. In this case, if node R_i has index $\pi(i)$ in the list, then the parent of R_i has index $\lfloor (\pi(i)/2) \rfloor$ in the list, and the

Figure 4.2: Various *ss_trees*. The number beside each node represents the index of the node in the list.

children of node R_i have index $2\pi(i)$ and $(2\pi(i) + 1)$. A list with 9 elements viewed as a binary *ss_tree* is shown in Figure 4.2 (b).

Example 3.

We now define a *k-branch ss_tree* where $k < n$. The root node has k children and every other node has one child. Therefore if node R_i has index $\pi(i) > k$ in the list, then its parent has index $\pi(i) - k$. Conversely if $\pi(i) < k$, the parent of node R_i is the root. Similarly the children of the root are the nodes with index $\{2, \dots, k + 1\}$, and the only child of the node with index $j > 1$ has index $(j + k)$. A list with 13 elements viewed as a 3-branch *ss_tree* is shown in Figure 4.2 (c).

Example 4.

A *leftist* binary *ss_tree* is a tree in which the root of the tree has two children and every left child of a node has two children (except at the leaf level) but every right child of a node has no children. Therefore if node R_i has index $\pi(i) > 3$ and $\pi(i)$ is an even number, then the parent of node R_i has index $(\pi(i) - 2)$. However, if $\pi(i)$ is an odd number, the parent of node R_i has index $(\pi(i) - 3)$. If $\pi(i) \leq 3$, the parent of node R_i is the root node. Figure 4.2 (d1) shows a leftist binary *ss_tree* with 7 elements and $k = 2$.

Similarly we can define the *leftist k-ary ss_trees* Figure 4.2 (d2) and (d3) show the *ss_trees* with $k = 3$ and $k = 4$ respectively.

Example 5.

A *rightist* binary *ss_tree* is symmetric to the leftist tree. The root of the tree has two children and every right child of a node has two children (except at the leaf level) but every left child of a node has no children. Therefore if node R_i has index $\pi(i) > 3$ and $\pi(i)$ is an even number, then the parent of node R_i has index $(\pi(i) - 1)$. However, if $\pi(i)$ is an odd number, the parent of node R_i has index $(\pi(i) - 2)$. If $\pi(i) \leq 3$, then the parent of node R_i is the root node. Figure 4.2 (e) shows a rightist binary *ss_tree* with 8 elements.

Similarly we can define the *rightist k -ary ss.trees*.

Example 6.

Arbitrary ss.trees are trees where there is no pre-defined ordering but only the parent/child ordering for specific positions. Examples of these are shown in Figure 4.2 (f) and (g). In these case, the parent/child relationship must be tabulated.

We shall show that if we apply the tree transformation process described in the previous section to an ss.tree associated with a list, then the resulting Markov chain is time reversible. That is to say, if we have an ss.tree, and the only operation allowed on the ss.tree is that of swapping a node with its parent, the resulting Markov chain is time reversible. This leads us to a new class of heuristics for self-organizing sequential search.

4.2.2 The Swap-with-Parent-in-an-SS_Tree Heuristic

Suppose we are given a list of records $\{R_1, R_2, \dots, R_n\}$ and an ss.tree associated with it. At every time instant one of these records R_i is accessed with probability s_i . Whenever a record is accessed, it is swapped with its parent in the corresponding ss.tree. We call this list reordering rule the *swap-with-parent-in-an-ss-tree* (SWPSST) rule.

First of all, note that the SWPSST rule is memory-free. Since, by the definition of the ss.trees, the index of the parent of each record is fully defined by the index of the child record, in the actual implementation, the only thing we need is to have a temporary variable to store the position of the parent of the accessed record during the search. The scheme is a generalization of the transposition heuristic and the swap-with-parent heuristic as the transposition heuristic is swap-with-parent-in-an-*unary-ss-tree* and the swap-with-parent heuristic of Chapter 3 is swap-with-parent-in-a-*binary-ss-tree*. As we shall see later, many results that hold for the transposition rule and the swap-with-parent rule can be generalized for the swap-with-parent-in-an-ss-tree rule. The most straightforward one is the time reversibility shown in the theorem below.

Theorem 4.3 *The Markov chain which results from using the swap-with-parent-in-an-ss-tree heuristic is time reversible.*

Proof:

We will carry the proof by induction on the number of elements, n , in the tree, and we will use the ‘box’ (sub-chain) notation which we developed for the proof of the time reversibility of the transposition rule (Theorem 2.10).

1. *Base case $n = 3$.*

For any tree with three elements, the tree can be in only one of the two structures shown below:

If the tree structure is $1(a)$, then the scenario is the same as the swap-with-parent rule on an s_heap of three elements. Similarly, if the tree structure is $1(b)$, then the scenario is the same as the transposition rule. Since in both cases, the Markov chain is time reversible, therefore, the Markov chain with three elements is time reversible in all cases.

2. *Case $n = 4$.*

Before we start the induction hypothesis, we shall look at the case of $n = 4$.

Using Theorem 2.9, to prove that a Markov chain is time reversible, it is sufficient to prove that starting in any state π , any path back to π , has the same probability as the reversed path.

The Markov chain for $n = 3$, has six states resulting from permuting the three elements a, b, c . Regardless of the tree structure, using the ‘box’ notation, the Markov chain can always be denoted by $\boxed{a \sim b \sim c}$. After adding one more element ‘d’ to the list, the simplified state diagram for the ‘four element’ Markov chain can be represented by Figure 4.3.

Figure 4.3: Simplified state diagram with four elements per state using the SW-PSST rule with an arbitrary *ss_tree*.

Since each sub-chain represented by a box in Figure 4.3 is exactly the same as the Markov chain with three elements, transitions within each ‘box’ preserve time reversibility. Also, since all four sub-chains are symmetric, it will be sufficient to prove that if we start in any one of the sub-chains, say, $[a \sim b \sim c \sim d]$, and in any state π , from *that* sub-chain ($[a \sim b \sim c \sim d]$), any path back to π , has the same probability as the reversed path.

Without loss of generality, let $\pi_s = \{a, b, c, d\}$, and assume that the parent of ‘d’ is ‘a’, then any path back to π_s from inside the sub-chain will have the same probability as the reversed path since we have shown that the Markov chain consisting of states entirely within this sub-chain is time reversible. So it remains to show that any path back to π_s which goes *outside* the box has the same probability as the reversed path. We know that to reach any states outside the sub-chain, element d must be accessed. After d is accessed, we are in state $\{d, b, c, a\}$ (since ‘a’ is the parent of ‘d’) in sub-chain $[b \sim c \sim d \sim a]$, and so to get back to state π_s , we have to get back to sub-chain $[a \sim b \sim c \sim d]$. Since all sub-chains are symmetrical, we only need to consider the path back to $[a \sim b \sim c \sim d]$ directly from $[b \sim c \sim d \sim a]$, which, in turn, means that the last state we will visit inside $[b \sim c \sim d \sim a]$ right before we get back to $[a \sim b \sim c \sim d]$ must be a state where ‘d’ is the parent of ‘a’. Since $\{d, b, c, a\}$ has already been visited when we first entered the sub-chain $[b \sim c \sim d \sim a]$, we will not be allowed to use it again. There is thus only one choice - we must get back from state

$\{d, c, b, a\}$. Obviously, the shortest path is then

$$\{a, b, c, d\} \Rightarrow \{d, b, c, a\} \Rightarrow \{d, c, b, a\} \Rightarrow \{a, c, b, d\} \Rightarrow \{a, b, c, d\}.$$

The product of the transition probabilities in the forward direction is $s_d s_c s_a s_b$, and in the reverse direction is $s_c s_d s_b s_a$. The equality in Theorem 2.9 holds.

The argument holds for a larger path which goes around visiting more states inside $\boxed{b \sim c \sim d \ a}$. This is because the large path inside $\boxed{b \sim c \sim d \ a}$ also starts from state $\{d, b, c, a\}$ and ends at state $\{d, c, b, a\}$, and it is already proven that sub-chain $\boxed{b \sim c \sim d \ a}$ is time reversible.

A similar argument can be given for every state in $\boxed{a \sim b \sim c \ d}$. Therefore, the Markov chain for $n = 4$ is time reversible.

3. Induction hypothesis.

Assuming that the Markov chain for k elements $\{R_1, \dots, R_k\}$ is time reversible, we shall prove that the Markov chain for $(k+1)$ elements $\{R_1, \dots, R_k, R_{k+1}\}$ is time reversible.

Note that the Markov chain with $(k+1)$ elements will have $(k+1)$ sub-chains which are all symmetric and are all actually the same as the Markov chain for k elements.

As before, take any state π_s , say $\pi_s = \{R_1, \dots, R_{j-1}, R_j, R_{j+1}, \dots, R_k, R_{k+1}\}$, from one of the sub-chains $\boxed{R_1 \sim \dots \sim R_k \ R_{k+1}}$, and assume that the parent of R_{k+1} is R_j . We need to prove that starting in π_s , any path back to π_s has the same probability as the reversed path. Again, by our hypothesis, we don't need to consider paths that are entirely inside the sub-chain, but only paths that go *outside* the sub-chain. To reach any state outside the sub-chain, element R_{k+1} has to be accessed. This makes the transition of the chain from π_s to $\{R_1, \dots, R_{j-1}, R_{k+1}, R_{j+1}, \dots, R_k, R_j\}$ in sub-chain $\boxed{R_1 \sim \dots \sim R_{j-1} \sim R_{j+1} \dots \sim R_{k+1} \ R_j}$. Since all sub-chains are symmetric, to get back to π_s , we only need to consider the path that is directly from sub-chain $\boxed{R_1 \sim \dots \sim R_{j-1} \sim R_{j+1} \dots \sim R_{k+1} \ R_j}$. Assuming that records R_i and R_i , are with parent/child relationship, a shortest path which visits only two states

inside $\boxed{R_1 \sim \dots \sim R_{j-1} \sim R_{j+1} \dots \sim R_{k+1} R_j}$ is

$$\begin{aligned}
 & \{R_1, \dots, R_{i_p}, \dots, R_i, \dots, R_j, \dots, R_{k+1}\} \\
 \Rightarrow & \{R_1, \dots, R_{i_p}, \dots, R_i, \dots, R_{k+1}, \dots, R_j\} \\
 \Rightarrow & \{R_1, \dots, R_i, \dots, R_{i_p}, \dots, R_{k+1}, \dots, R_j\} \\
 \Rightarrow & \{R_1, \dots, R_i, \dots, R_{i_p}, \dots, R_j, \dots, R_{k+1}\} \\
 \Rightarrow & \{R_1, \dots, R_{i_p}, \dots, R_i, \dots, R_j, \dots, R_{k+1}\}.
 \end{aligned}$$

The product of the transition probabilities in the forward direction which is $s_{k+1}s_i s_j s_{i_p}$ equals to that in the reverse direction which is $s_i s_{k+1} s_{i_p} s_j$. The argument holds for any large path for the same reason stated for the case where $n = 4$. Therefore, the Markov chain is time reversible, and the inductive step is complete. The result follows. $\square$

It directly leads to the equation shown in Corollary 4.1.

Corollary 4.1 *Under the swap-with-parent-in-an-ss_tree heuristic, the stationary probabilities obey*

$$\frac{P\{R_{i_1} \dots R_{\text{parent}(i_j)} \dots R_{i_j} \dots R_{i_n}\}}{P\{R_{i_1} \dots R_{i_j} \dots R_{\text{parent}(i_j)} \dots R_{i_n}\}} = \frac{s_{\text{parent}(i_j)}}{s_{i_j}} \quad (4.5)$$

where $\text{parent}(i_j)$ is the index of the parent of R_{i_j} in the corresponding ss_tree. $\square$

Corollary 4.2 *Under the swap-with-parent-in-an-ss_tree heuristic the stationary probabilities obey:*

$$\frac{P\{\dots, R_i, R_{i_1}, \dots, R_{i_k}, R_j, \dots\}}{P\{\dots, R_j, R_{i_1}, \dots, R_{i_k}, R_i, \dots\}} = \left(\frac{s_i}{s_j}\right)^{k_{ij}} \quad (4.6)$$

where $k_{ij} = |\text{depth}(\pi(i)) - \text{depth}(\pi(j))|$ (in the corresponding ss_tree) and $1 \leq i, j \leq n$ if $s_j \neq 0$.

Proof:

This follows easily by successive swapping on the corresponding ss_tree using Equation (4.5). $\square$

Corollary 4.3 *Under the swap-with-parent-in-an-ss_tree heuristic, the stationary probabilities between any two arbitrary states π_s and π_t obey:*

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_t, \pi_s)} \quad (4.7)$$

where $\xi_i(\pi_t, \pi_s) = \text{depth}(\pi_t(i)) - \text{depth}(\pi_s(i))$ for $1 \leq i \leq n$ is the difference between the depths of R_i in ordering π_s and π_t . $\square$

The proof of the above corollary is the same as the proof for Corollary 3.2.

Using the above we can present the expression for the asymptotic search cost of the SWPSST heuristic. It is a generalization of the one for the swap-with-parent heuristic shown in Theorem 3.2.

Theorem 4.4 *Let π_0 denote the identity permutation on n records $\pi_0(i) = i$ for $1 \leq i \leq n$, which is the optimal ordering under our assumption that $s_i \geq s_{i+1}$ for $1 \leq i \leq n$. Let $\xi_i(\pi_0, \pi)$ denote the quantity $(\text{depth}(i) - \text{depth}(\pi(i)))$ for any permutation π and $1 \leq i \leq n$, which is the difference between the depths of R_i in the optimal ordering and in ordering π of the corresponding ss_tree. Then the asymptotic search cost for the swap-with-parent-in-an-ss_tree heuristic is*

$$P\{\pi_0\} \sum_{\text{all } \pi} \left(\prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \sum_{1 \leq j \leq n} s_j \pi(j) \right) \quad (4.8)$$

where

$$P\{\pi_0\} = \left(\sum_{\text{all } \pi} \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \right)^{-1}. \quad (4.9)$$

Proof:

From Equation (2.6), the average search cost of an algorithm can be calculated by

$$\sum_{\text{all } \pi} \left(P\{\pi\} \sum_{1 \leq j \leq n} s_j \pi(j) \right).$$

Using Corollary 4.3, we have,

$$P\{\pi\} = P\{\pi_0\} \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \quad \text{for all } \pi. \quad (4.10)$$

Consequently

Average search cost for SWPSST

$$\begin{aligned} &= \sum_{\text{all } \pi} \left(\left(P\{\pi_0\} \prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right) \\ &= P\{\pi_0\} \sum_{\text{all } \pi} \left(\left(\prod_{1 \leq i \leq n} s_i^{\xi_i(\pi_0, \pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right), \end{aligned}$$

thus Equation (4.8) is proved.

Since

$$\sum_{\text{all } \pi} P\{\pi\} = 1,$$

applying Equation (4.10), we have

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

which proves Equation (4.9). $\square$

Therefore, to find the cost of the SWPSST rule with a specific *ss_tree*, the only thing we need to do is to find the value of $\xi_i(\pi_0, \pi)$ and replace it in Equation (4.8) and (4.9). For example, the value of $\xi_i(\pi_0, \pi)$ for the 3-branch tree shown in Figure 4.2 (d2) is

$$\lceil \frac{\pi_0(i) - 1}{3} \rceil - \lceil \frac{\pi(i) - 1}{3} \rceil.$$

Similar expressions can be written for each of the *ss_tree* structures given in Figure 4.2 except (f) and (g). They are omitted in the interest of brevity.

Using time reversibility, we can prove the following results. The proof of the results is very close to the proof of their analogous theorems (Theorem 3.3 and Theorem 3.4) in the previous chapter. Therefore, in the interest of brevity, we will not show the proofs here.

Theorem 4.5 *The swap-with-parent-in-an-ss_tree heuristic is expedient since*

$$P\{R_j \text{ precedes } R_i\}_{\text{SWPSST}} > \frac{1}{2} \quad \text{if } s_j > s_i.$$

$\square$

Theorem 4.6 *For any two records R_i and R_j , let $k_{ij} = | \text{depth}(\pi(i)) - \text{depth}(\pi(j)) |$ in the corresponding ss_tree . Then under the swap-with-parent-in-an- ss_tree heuristic we have:*

$$P \{ R_j \text{ precedes } R_i \mid k_{ij} = d \} = \frac{1}{1 + \left(\frac{s_i}{s_j} \right)^d} = \frac{s_j^d}{s_i^d + s_j^d}. \quad (4.11)$$

□

The performance of the SWPSST heuristic (both asymptotic search cost and convergence) depends heavily on the type of the underlying ss_tree . It is not difficult to see that the rule will not cost less than the transposition rule when the corresponding ss_tree has more than one branch. However we conjecture that it costs less than the move-to-front rule. Furthermore, the smaller the number of records in the leaf level in the corresponding ss_tree , the better the algorithm will be. Equivalently, the bushier (more branches) the tree is, the worse the algorithm will be. This is because to make the algorithm better than the move-to-front rule, we would like that

$$P \{ R_j \text{ precedes } R_i \}_{SWPSST} > P \{ R_j \text{ precedes } R_i \}_{MTF} = \frac{s_j}{s_i + s_j}$$

given that $s_j > s_i$ for $1 \leq i, j \leq n$ and $j \neq i$.

From Theorem 4.6 above, we know that if $s_j < s_i$, then

$$P \{ R_j \text{ precedes } R_i \mid k_{ij} = d = 0 \}_{SWPSST} < \frac{s_j}{s_i + s_j}. \quad (4.12)$$

$$P \{ R_j \text{ precedes } R_i \mid k_{ij} = d \geq 1 \}_{SWPSST} \geq \frac{s_j}{s_i + s_j}. \quad (4.13)$$

Therefore we would like the *asymptotic* probability of two records being in the same level in the corresponding ss_tree to be smaller than that of them *not* being in the same level. In other words, the smaller the number of branches the tree has, the less the algorithm costs.

The best-case performance of the algorithm happens when the underlying ss_tree is an *unary* tree, in which case the algorithm is equivalent to the transposition rule. The worst-case occurs when the underlying ss_tree has $(n - 1)$ branches, i.e., the

first record is the parent of all other records in the list. However, the speed of convergence of the algorithm will be more than that of the transposition.

Let us revisit some of the examples of the various *ss_trees* are shown from Section 4.2.1 (see also Figure 4.4). The *k-branch ss_tree* shown in Example 3 (Figure 4.4 (c)) when *k* is chosen properly, and the *ss_tree* shown in Example 6 (Figure 4.4 (f)) will be better choices for the underlying *ss_tree* when using the swap-with-parent-in-an-*ss_tree* heuristic. The *ss_trees* shown in Example 4 and 5 (Figure 4.4 (d1), (d2), (d3) and (e)) are the bad choices for the reasons listed above.

4.2.3 Drawbacks of the SWPSST Rule and its Modification - the Move-to-Parent-in-an-SS_Tree Rule

The drawback of the swap-with-parent heuristic persists in the SWPSST heuristic. The scenario is exactly the same - the elements with zero (or negligible) access probabilities will prove to be a drag to the entire access strategy. After those elements eventually fall to the leaf level, they will not be moved again, therefore the cost of future accesses to the elements *behind* them in the list will be increased, and the situation will not get better. This is again the reason why we prefer the *ss_tree* not to be 'bushy'.

The solution to this problem is the same as the solution to the swap-with-parent heuristic. Instead of swapping the accessed record with its parent in the associated *ss_tree* and leaving all the records in between unchanged, we will move the accessed record to its parent's position and shift its parent and all the records in between back one position. We shall call this scheme the *move-to-parent-in-an-ss_tree* (MTPSST) heuristic.

Notice that if the underlying *ss_tree* is an *k-branch ss_tree* (see Example 3 in Section 4.2.1 and Figure 4.4 (c)), the corresponding MTPSST is the *move-ahead-k* scheme. If the *ss_tree* is an $(n - 1)$ -branch tree, the scheme is the *move-to-front*

scheme, and if the `ss_tree` is a unary `ss_tree` (one-branch tree), the scheme is the transposition scheme.

Like the move-to-parent heuristic, this scheme avoids the drawbacks of many other algorithms including the drawbacks related to ‘locality’ mentioned in Section 3.5. We conjecture that its average search cost is lower than the move-to-front rule and higher than the transposition rule. Absolute analysis of it is needed to allow realistic comparisons.

4.3 Conclusion

In this chapter, we have formally defined the *tree transformation process* and introduced a generalization of the transposition rule and the swap-with-parent rule - the *swap-with-parent-in-an-ss_tree* heuristic and its modification - the *move-to-parent-in-an-ss_tree* heuristic. Both of these are defined on the underlying implicit tree structure defined on the list - a structure we have called the sequential search tree or `ss_tree`.

The performance of both of the schemes depends heavily on the type of the underlying `ss_tree` used. In both of the schemes, we believe that in terms of accuracy, the best-case scenario happens if the underlying `ss_tree` is the unary tree, i.e., the tree with one branch, in which case the scheme is actually the transposition rule. This heuristic is the most sluggish scheme in terms of convergence. The worst-case scenario occurs if the underlying `ss_tree` is an $(n - 1)$ -branch tree. Generally speaking, the ‘bushier’ the `ss_tree` is, the more the algorithms costs, and the faster the convergence rate is.

We conjecture that SWPSST heuristic costs less than the move-to-front heuristic and that the MTPSST heuristic is even better. Further analyses for both of the schemes are needed. This indeed gives rise to numerous open problems.

Chapter 5

Discussions and Conclusions

In this thesis, we have extensively studied self-organizing sequential search algorithms. In particular, we have analyzed *time reversible* self-organizing sequential search algorithms, including previously known algorithms and various new algorithms which we introduced. We shall now present various issues which warrant further research in the area.

5.1 Amortized Analysis

The majority of the analyses on self-organizing sequential search algorithms focus on the average search cost in terms of the number of probes required to locate a record in the list. A little research is available on the amortized analysis of the move-to-front rule and the transposition rule [5, 33]. We shall give a brief discussion on the issue in this section.

Amortization implies averaging the running time of an algorithm over a worst-case sequence of operations. That is, in an *amortized analysis*, the time required to perform a sequence of data-structure operations is averaged over all the operations performed. An amortized bound on the running time of an algorithm can generally be more indicative than an average-case bound of similar magnitude. This is because an amortized bound gives the cost of a worst-case sequence of operations. It guarantees the average performance of each operation in the worst case, as opposed

to the average-case bound which only tells what will *probably* happen, and only applies under probabilistic assumptions. Another reason is that in many situations an amortized bound is more useful than a worst-case average bound, since a worst-case average bound cannot take into account the correlations among different operations. Therefore the amortized complexity of the algorithm often yields additional insight into the behavior of the algorithms.

Amortized analysis is only available for the move-to-front rule and the transposition rule [33]. Sleator and Tarjan proved that the amortized complexity of the move-to-front rule is within a constant factor of minimum among all possible self-organizing list updating rules. This result is much stronger than the previous average-case results on self-organizing list updating heuristics which favor the transposition rule over the move-to-front rule. It also provides a theoretical support to Bentley and McGoech's experiments showing that the move-to-front rule is generally the best rule in practice [4]. This implies that the amortized complexity provides a more realistic measure for self-organizing list updating rules than the asymptotic average complexity.

Unfortunately, the rigorous amortized analysis is not available for most of the self-organizing linear search heuristics. Further research is needed to allow realistic comparisons among the algorithms including the time reversible algorithms discussed in this thesis.

5.2 Criteria for Choosing an Algorithm

In Chapter 2, we have presented various criteria for comparing different algorithms. However due to the complexity of the move rules, absolute analysis is not available for various algorithms that move the accessed record some distance between the two extremes (the move-to-front and the transposition). There are no guidelines for choosing among these algorithms.

Bentley and McGeoch [5] gave some suggestions about choosing between the move-to-front rule and the transposition rule which may be generalized to the other

algorithms that move records forward either a fraction of the distance or a fixed distance to the front of the list.

For the move-to-front rule, since the linked-list implementation is preferable for most applications, when the environment supports dynamic storage allocation, move-to-front can yield an efficient reordering strategy. They also pointed out that the move-to-front rule exploits *locality* of reference present in the input sequence.

For the transposition rule, if storage is extremely limited and pointers for lists cannot be used, then the array implementation of the transposition rule gives very efficient reorganization of the list. Its worst-case performance is poor, but its average performance is commendable.

It is necessary to have criteria for choosing among the various algorithms to characterize the circumstances when any particular algorithm is advantageous. This is still an open question.

5.3 Locality

As stated in Section 2.1, in self-organizing sequential search algorithms, it is commonly assumed that the accesses to the record in the list are made independent of previous accesses and that the access probabilities do not vary with respect to time, (stationary). Consequently, the majority of the analyses on self-organizing sequential search algorithms assume that all access sequences which have the same set of access probabilities are equally likely.

This assumption does not take into account a common attribute of access sequences called *locality* which makes the assumption unreasonable in many applications. Locality means that some “subsequences of the entire access sequence may have relative access frequencies that are drastically different from the overall relative access frequencies” [16].

Hester and Hirschberg [16] showed how the *locality* of the access sequence affects the average search cost of a scheme through the following example. Consider a list of 26 records which are the 26 English letters $\{a, b, \dots, z\}$ in a random order. Each

record is accessed exactly 10 times, that is, $s_a = s_b = \dots = s_z = 1/26$. Let us assume that the reordering rule is the *move-to-front*.

If the access sequence is as below,

$$\underbrace{a, \dots, z, a, \dots, z, \dots, a, \dots, z}_{10 \text{ times}}$$

then under the move-to-front rule, accesses to each record (except the first access of each record) will always take 26 probes. Therefore the total number of probes required to access the records in the input sequence (except the first access to each record) is $9 \times 26 \times 26$, and the first access takes between 26 and the position of the record in the list. The best-case scenario happens when the list is initially in alphabetical order. Then the cost of the move-to-front rule will be:

$$\frac{9 \times 26 \times 26 + \sum_{i=1}^{26} i}{260} = 24.75$$

Now consider another access sequence which has the same probability distribution as the previous, but are in a different order shown below,

$$\underbrace{a, \dots, a}_{10 \text{ times}}, \underbrace{b, \dots, b}_{10 \text{ times}}, \dots, \underbrace{z, \dots, z}_{10 \text{ times}}$$

in this case, all accesses (except the first) to each record will only take one probe. The worst-case scenario happens when the list is initially in the reversed alphabetical order, and the cost of the move-to-front rule is:

$$\frac{9 \times 26 \times 1 + \sum_{i=1}^{26} i}{260} = 3.5$$

Note that the worst-case cost of the second scenario is far less than the best-case cost of the first scenario. This indicates that the cost of a scheme can differ greatly for the same fixed probability distribution when the order of the actual input sequences are different.

Since the move-to-parent rule is not as drastic as the move-to-front rule when moving a record forward, it can be easily verified that the the cost of the first input sequence in the above example under the move-to-parent rule is less than that of

the move-to-front rule. Simultaneously, the cost of the second sequence in the above example under the move-to-parent rule is more than that of the move-to-front rule. Therefore for the same fixed probability distribution, when the order of the actual input sequences are different, the cost of the move-to-parent rule does not differ as much as the move-to-front rule. This demonstrates that the move-to-parent heuristic responds in a superior fashion when it encounters this kind of ‘locality’ problems.

Another case that can happen when using the transposition rule and the swap-with-parent rule is when two records are alternately accessed many times. Under all the *swap* related schemes, these two records will continue exchanging places without advancing toward the front of the list. It is easily seen that the move-to-front rule, the move-to-parent rule and all the generalized move-to-parent rules do not have this problem.

Hendricks [15] mentioned the open problem of relaxing the assumption of independence for analysis of self-organizing algorithms. However, no good definition for locality of accesses has been applied to the problem of quantifying the efficiency of self-organizing linear search algorithms. This problem is still open.

A few algorithms have been suggested to control the ‘locality’ problem. Hybrid algorithms, such as using the move-to-front rule first and then switching to the transposition rule, are one of them. However, as we have pointed out earlier, it is difficult to know when the system should switch between the rules. For reasons explained earlier, we conjecture that the move-to-parent heuristic introduced in Chapter 3 will have better control over ‘locality’ problems, as will the MTPSST heuristics introduced in Chapter 4. More research needs to be done to provide theoretical support to these conjectures.

5.4 Conclusions

In this thesis, we have examined heuristics that dynamically organize linear search lists. An extensive literature survey on the known self-organizing sequential search algorithms has been given. The transposition heuristic generally has been considered

to be the best algorithm in terms of the asymptotic search cost per access, whereas the move-to-front heuristic seem to have the fastest convergence rate.

We have introduced two new memory-free self-organizing sequential search algorithms, the swap-with-parent scheme and the move-to-parent scheme. We have shown that the Markov chain representing the swap-with-parent scheme is time reversible. This property makes the analysis of the swap-with-parent scheme greatly simplified. It further leads us to a class of time reversible Markov chains which opens a new research area in the use and the analysis of “implicit” tree structures in adaptive algorithms. Using these we then introduced two classes of self-organizing sequential search schemes, the swap-with-parent-in-an-ss_tree heuristic and the move-to-parent-in-an-ss_tree heuristic. The properties of the former have been formally proven and the properties of the latter remain uninvestigated. We conjectured that their performance lies between that of the move-to-front rule and the transposition rule.

We have addressed various possible extensions which could strength and generalize the results presented in this thesis. They include the amortized analysis of the algorithms, criteria for choosing an algorithm and taking advantage of locality to allow more realistic applications of these algorithms.

Appendix A

Proof of Theorem 3.5

Theorem 3.5: *Consider a complete binary tree with n records $\{R_1, \dots, R_n\}$ which are in a random order, (note that $n = 2^q - 1$ for some $q > 1$.) For any two records R_i and R_j , let $k_{ij} = | \lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor |$ for $1 \leq i, j \leq n$. Then the following equations hold:*

$$\begin{aligned} P \{ k_{ij} = d = 0 \} &= \frac{1}{3}, \\ P \{ k_{ij} = d = 1 \} &= \frac{1}{3} + \frac{1}{n}, \\ P \{ k_{ij} = d > 1 \} &= \frac{1}{3} - \frac{1}{n}. \end{aligned}$$

Proof:

The proof will be done by induction on the number of records, n , in the tree. (We shall use the term 'depth' of the tree and the 'level' of the tree interchangeably.)

1. *Base case $n = 3$.*

If $n = 3$, then the total number of pairs in the tree is $\binom{3}{2} = 3$. The number of pairs that are in the same level is 1, the number of pairs that are one level apart is 2, and there are no two records that are two or more levels apart. Obviously, the equations in the theorem hold.

2. *Induction hypothesis.*

Assuming that the equations in the theorem hold for $n = 2^r - 1$ where $r < q$, we shall prove that the equations in the theorem hold for $n = 2^{r+1} - 1$ (i.e., the tree has one more level than the tree with $n = 2^r - 1$.) That is, suppose $n = 2^r - 1$ and

$$\begin{aligned} P\{k_{ij} = d = 0\} &= \frac{1}{3}, \\ P\{k_{ij} = d = 1\} &= \frac{1}{3} + \frac{1}{2^r - 1}, \\ P\{k_{ij} = d > 1\} &= \frac{1}{3} - \frac{1}{2^r - 1}. \end{aligned}$$

We need to prove that for $n = 2^{r+1} - 1$, the following equations hold:

$$\begin{aligned} (1). P\{k_{ij} = d = 0\} &= \frac{1}{3}, \\ (2). P\{k_{ij} = d = 1\} &= \frac{1}{3} + \frac{1}{2^{r+1} - 1}, \\ (3). P\{k_{ij} = d > 1\} &= \frac{1}{3} - \frac{1}{2^{r+1} - 1}. \end{aligned}$$

Note that if $n = 2^r - 1$, then the total number of pairs in the tree are

$$\binom{2^r - 1}{2} = \frac{(2^r - 1)!}{(2^r - 3)! 2!} = \frac{(2^r - 1)(2^r - 2)}{2}.$$

If $n = 2^{r+1} - 1$, the total number of pairs in the tree are

$$\binom{2^{r+1} - 1}{2} = \frac{(2^{r+1} - 1)!}{(2^{r+1} - 3)! 2!} = \frac{(2^{r+1} - 1)(2^{r+1} - 2)}{2} = (2^{r+1} - 1)(2^r - 1).$$

Since the number of pairs that are in the same level when $n = 2^{r+1} - 1$ is

$$\binom{2^r - 1}{2} \times \frac{1}{3} + \binom{2^r}{2},$$

we get

$$P\{k_{ij} = d = 0\} = \frac{\binom{2^r - 1}{2} \times \frac{1}{3} + \binom{2^r}{2}}{\binom{2^{r+1} - 1}{2}}$$

$$\begin{aligned}
 &= \frac{(2^r - 1)(2^r - 2)/6 + 2^r(2^r - 1)/2}{(2^{r+1} - 1)(2^r - 1)} \\
 &= \frac{4 \times (2^r - 2)}{6(2^{r+1} - 1)} \\
 &= \frac{1}{3}.
 \end{aligned}$$

Thus Equation (1) is proved.

Similarly, since the number of pairs that are one level apart when $n = 2^{r+1} - 1$ is

$$\binom{2^r - 1}{2} \times \left(\frac{1}{3} + \frac{1}{2^r - 1} \right) + (2^r \times 2^{r-1}),$$

we see that

$$\begin{aligned}
 P\{k_{ij} = d = 1\} &= \frac{\binom{2^r - 1}{2} \times \left(\frac{1}{3} + \frac{1}{2^r - 1} \right) + (2^r \times 2^{r-1})}{\binom{2^{r+1} - 1}{2}} \\
 &= \frac{\left(\frac{(2^r - 1)(2^r - 2)}{2} \right) \left(\frac{1}{3} + \frac{1}{2^r - 1} \right) + 2^{2r-1}}{(2^{r+1} - 1)(2^r - 1)} \\
 &= \frac{(2^r - 1)(2^r - 2)/6 + (2^r - 2)/2 + 2^{2r-1}}{(2^{r+1} - 1)(2^r - 1)} \\
 &= \dots \\
 &= \frac{2^r + 1}{6(2^{r+1} - 1)} + \frac{2^r + 1}{2(2^{r+1} - 1)} \\
 &= \dots \\
 &= \frac{2^{r+1} + 2}{3(2^{r+1} - 1)} \\
 &= \frac{1}{3} + \frac{1}{2^{r+1} - 1}.
 \end{aligned}$$

Thus Equation (2) is proved.

Since

$$\begin{aligned}
 P\{k_{ij} = d > 1\} &= 1 - P\{k_{ij} = d = 0\} - P\{k_{ij} = d = 1\} \\
 &= 1 - \left(\frac{1}{3} \right) - \left(\frac{1}{3} + \frac{1}{2^{r+1} - 1} \right) \\
 &= \frac{1}{3} - \frac{1}{2^{r+1} - 1},
 \end{aligned}$$

which proves Equation (3).

The inductive step is complete. The result follows. $\square$

Appendix B

Proof of Corollary 3.3

Corollary 3.3: *Consider an incomplete binary tree (the leaf level is not full) with n records $\{R_1, \dots, R_n\}$ which are in a random order, (i.e., $n \neq 2^q - 1$ for some $q > 1$.) Then for any two records R_i and R_j , if $k_{ij} = |\lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor|$ for $1 \leq i, j \leq n$, the following inequalities hold.*

$$P\{k_{ij} = 0\} < \frac{1}{3} \quad \text{and} \quad P\{k_{ij} > 0\} > \frac{2}{3}.$$

Proof:

Consider the results of Theorem 3.5. If the tree is not complete, then the number of records in the leaf level will be less than that of the complete tree. Therefore the number of records that are in the same level will be less than that of the complete tree, that is

$$P\{k_{ij} = 0\} < \frac{1}{3}$$

Since

$$P\{k_{ij} > 0\} = 1 - P\{k_{ij} = 0\},$$

it follows that

$$P\{k_{ij} > 0\} > \frac{2}{3}.$$

The proof is complete. □

References

- [1] E. J. Anderson, P. Nash & R. R. Weber (1982). A counterexample to a conjecture on optimal list ordering. *J. Appl. Prob.* 19, 3, pp.730-732.
- [2] D. M. Arnow & A. M. Tenenbaum (1982). An investigation of the move-ahead-k rules. *Congressus Numerantium, Proc. Thirteenth Southeastern Conference on Combinatorics, Graph Theory and Computing, Florida*, pp47-65.
- [3] S. Asmussen (1987). *Applied Probability and Queues*. John Wiley & Sons, Chichester.
- [4] J. L. Bentley & C. C. McGeoch (1982). Worst-case analysis of self-organizing sequential search heuristics. *Comm. ACM* 28, 4, pp401-411.
- [5] J. L. Bentley & C. C. McGeoch (1985). Amortized analyses of self-organizing sequential search heuristics. *Proc. 20th Allerton Conference on Comm. Control, and Computing*.
- [6] J. R. Bitner (1976). Heuristics that dynamically alter data structures to reduce their access time. *University of Illinois Report UIUCDCS-R-76-818*, July, 1976 (Ph.D. thesis).
- [7] J. R. Bitner (1979). Heuristics that dynamically organize data structures. *SIAM J. Computing*, 8, 1, pp. 82-110.
- [8] R. P. Cheetham, B. J. Oommen & D. T. H. Ng (1993). Adaptive structuring of binary search trees using conditional rotations. *IEEE Trans. Knowledge and Data Eng.* 5, pp. 695-704.

- [9] F. R. K. Chung, D. J. Hajela & P. D. Seymour (1988). Self-organizing sequential search and Hilbert's inequalities. *J. Comp. and Sys. Sci.* 36, pp148-157.
- [10] T. H. Cormen, C. E. Leiserson & R. L. Rivest (1993). *Introduction to Algorithms*. The MIT Press, Cambridge, MA.
- [11] G. H. Gonnet, J. I. Munro & H. Suwanda (1979). Toward self-organizing linear search. *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science (San Juan, Puerto Rico, Oct.)*, IEEE, New York, pp. 169-174.
- [12] G. H. Gonnet, J. I. Munro & H. Suwanda (1981). Exegesis of self-organizing linear search. *SIAM J. Computing*, 10, 3, pp. 613-637.
- [13] W. J. Hendricks (1972). The stationary distribution of an interesting Markov chain. *J. Appl. Prob.* 9, pp231-233.
- [14] W. J. Hendricks (1973). An extension of a theorem concerning an interesting Markov chain. *J. Appl. Prob.* 10, pp886-890.
- [15] W. J. Hendricks (1976). An account of self-organizing systems. *SIAM J. Computing*, 5, 4, pp. 715-723.
- [16] J. H. Hester & D. S. Hirschberg (1985). Self-organizing linear search. *ACM Computing Surveys*, pp. 295-311.
- [17] Y. C. Kan & S. M. Ross (1980). Optimal list order under partial memory constraints. *J. Appl. Prob.* 17, pp1004-1015.
- [18] J. Keilson (1979). *Markov Chain Models - Rarity and Exponentiality*. Springer-Verlag, New York.
- [19] F. P. Kelly (1987). *Reversibility and Stochastic Networks*. John Wiley & Sons, Chichester.
- [20] J. G. Kemeny & J. L. Snell (1960). *Finite Markov Chains*. Van Nostrand, Princeton, N. J.

- [21] S. Karlin & H. M. Taylor (1975). *A First Course in Stochastic Processes*. Academic Press, New York.
- [22] D. E. Knuth (1973). *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, Ma..
- [23] J. McCabe (1965). On serial files with relocatable records. *Operations Research*, 12, pp. 609-618.
- [24] B. J. Oommen (1984). On the use of smoothsort and stochastic move-to-front operations for optimal list organization. *Proceedings of the 22nd Annual Allerton Conference on Communication, Control, and Computing* (Monticello, Ill., Oct). Univ. of Illinois, Urbana-Champaign, pp243-252.
- [25] B. J. Oommen & E. R. Hansen (1987). List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations. *SIAM J. Computing*, 16, pp. 705-716.
- [26] B. J. Oommen, E. R. Hansen & J. I. Munro (1990). Deterministic optimal and expedient move-to-rear list organizing strategies. *Theoretical Comp. Sci.*, 74, pp. 183-197.
- [27] B. J. Oommen & D. T. H. Ng (1993). An optimal absorbing list organization strategy with constant memory requirements. *Theoretical Comp. Sci.*, 119, pp. 355-361.
- [28] B. J. Oommen & J. Dong (1997). A formal proof for the time reversibility of a well known self-organizing sequential search algorithm. *In preparation*.
- [29] B. J. Oommen & J. Dong (1997). The swap-with-parent scheme: a self-organizing sequential search algorithm which uses non-lexicographic heaps. *In preparation*.

- [30] B. J. Oommen & J. Dong (1997). Time reversibility: a mathematical tool for creating arbitrary generalized swap-with-parent self-organizing lists. *In preparation*.
- [31] R. L. Rivest (1976). On self-organizing sequential search heuristics. *Comm. ACM*, 19, pp. 63-67.
- [32] S. M. Ross (1980). *Introduction to Probability Models*, 2nd edition. Academic Press, London.
- [33] D. D. Sleator & R. E. Tarjan (1985). Amortized efficiency of list update and paging rules. *Comm. ACM* 28 (Feb.), pp202-208.
- [34] A. Tenenbaum (1978). Simulations of dynamic sequential search algorithms. *Comm. ACM* 21, 9 (Sept.), pp790-791.
- [35] A. M. Tenenbaum & R. M. Nemes (1982). Two spectra of self-organizing sequential search algorithms. *SIAM J. Computing*, 11, pp. 557-566.