Self-Organizing Linear Search

J. H. Hester and D. S. Hirschberg
University of California, Irvine. Irvine, California 92717

Algorithms that modify the order of linear search lists are surveyed. First the problem, including assumptions and restrictions, is defined. Next a summary of analysis techniques and measurements that apply to these algorithms is given. The main portion of the survey presents algorithms in the literature with absolute analyses where available. The following section gives relative measures which are applied between two or more algorithms. The final section presents open questions.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introduction and Survey; D.4.2 [**Operating Systems**]: Storage Management—*swapping*; E.1 [**Data**]: Data Structures—*lists*; *tables*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*; F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs among Complexity Measures

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Linear Search, Self organizing lists, Relative Measures, Convergence, Locality

INTRODUCTION

A *linear search list* is a list of initially unordered records that will be sequentially searched through on the basis of a key value associated with each record. The goal of the search may be merely to see if the record exists in the list, to look up data associated with the key, or to modify data within the record. A linear search list is ordered in the sense that searches may 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 sequentially allocated array (either containing the records or pointers to them) or as a linked list. Linear searches on such a list are required in cases where the list is linked, or the elements are not ordered in any way that would facilitate faster search techniques.

It is assumed that some records are accessed more frequently than others. A *self-organizing linear search list* may permute the order of the records in some fashion after a record is found, attempting to place the more frequently-accessed records closer to the front of the list to reduce future search times. What algorithms can be used for this permutation, and how they perform relative to each other in terms of expected search time, is the question we address in this article.

Two examples of simple permutation algorithms are *move-to-front,* which moves the accessed record to the front of the list, shifting all records previously ahead of it back one position; and *transpose,* which merely exchanges the accessed record with the one immediately ahead of it in the list. These will be described in more detail later.

Knuth [1973] describes several search methods that are usually more efficient than linear search. Bentley and McGeoch [1985] justify the use of self-organizing linear search in the following three contexts:

- When n is small (say, at most several dozen), the simplicity of the code can make it faster than more complex algorithms. This occurs, for example, when linked lists are used to resolve collisions in a hash table.
- When space is severely limited, sophisticated data structures may require too much space.
- If the performance of linear search is almost (but not quite) good enough, a self-organizing linear search list may give acceptable performance without adding more than a few lines of code.

As an example of the third context, they describe a case in which a VLSI circuit-simulation program spent five 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 five minutes were annoying to those who were waiting for the simulation to begin. After incorporating a simple self-organizing search (about a half dozen additional lines of code), the setup time was reduced to about thirty seconds, most of which was not involved with searching the symbol table.

This situation can also arise in a list of identifiers maintained by a compiler or interpreter; for example, in the scatter table used by the University of California at Irvine's (UCI) LISP system. 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 elements in the scatter table. Since most programs tend to access some identifiers more often than others, we would like the more frequently-accessed identifiers to be be nearer the front of the list in order to obtain a lower average search cost. However, the lists cannot be ordered initially, since the number of references to each identifier is not known.

0. RESTRICTIONS

Permutation algorithms are algorithms that maintain self-organizing lists by permuting the records. The *accessed record* is the record currently being searched for, and the *probed record* is the record currently being looking at during a search.

Several limiting assumptions apply to the problem of self-organizing linear search. The primary assumption is that the probability of access to any record is unknown prior to its access. However, we can assume that the access probabilities follow some distribution or general rule, without knowing how that applies to individual records.

Only algorithms concerning linear lists are considered. Although some work has been done with arranging the records in self-organizing trees, which dramatically reduces search time, linear lists are still widely used due to savings in space and code complexity. Among the references given in this survey, Bitner [1979] gives an extensive discussion on trees and Gonnet et al. [1981] mention trees briefly.

In the general case, a permutation algorithm could completely reorder the list at any time. We consider only algorithms that apply permutations after each access, and whose permutations involve moving the accessed record some distance forward in the list, leaving the rest of the records unchanged relative to each other. Note that *any* permutation algorithm that fits this restriction will leave the list unchanged whenever the first record is accessed.

We also assume that the search will not access records that are not found in the list, and that each record in the list will be accessed at least once.

Finally, we assume that the time required by any execution of the permutation algorithm is never more than a constant times the time required for the search immediately prior to that execution, although many of the surveyed algorithms perform the permutation in constant time.

1. MEASURES

Before trying to find a good permutation algorithm, it is necessary to define what is meant by "good." Arbitrarily label the records 1 to n. Let ρ be the search sequence, such that $\rho[k]$ is the label of the record to be searched for on the k^{th} access. Let $\Lambda(\rho, a)$ represent the configuration of the list (the ordering of the records) such that $\Lambda(\rho, a)_r$ is the *location* in the list of the record labeled r after the first a accesses from ρ (and the permutation following each access) have taken place. The first record is defined to be in location 1 and the last record in a list of n records is in location n. $\Lambda(\rho, 0)$ represents the initial configuration of the list, and will be denoted by λ for simplicity.

1.1. Cost

The *cost* of a permutation algorithm α for a given λ and ρ is the average cost per access, in terms of the number of probes required to find the accessed record and the amount of work required to permute the records afterwards. We denote $\Lambda(\rho, k-1)_{\rho[k]}$ to be the location of the k^{th} record to be accessed (which is equivalent to the number of probes required to find it). Thus $C_\alpha(\lambda, \rho)$ (the cost for searching and reordering using permutation algorithm α) is the sum of this value for all accesses in ρ divided by the total number of accesses in ρ:

$$C_\alpha(\lambda, \rho) \quad = \quad \frac{\sum_{k=1}^{|\rho|} \Lambda(\rho, k-1)_{\rho[k]}}{|\rho|} + \text{permutation cost of } \alpha$$

An example incorporating an application of this formula is given in the section on *locality*.

Unfortunately, it is assumed that ρ is unknown at the start of the searches. Therefore a permutation algorithm can only be measured by making some assumptions about the search sequence. The following measures and assumptions are used.

1.1.1. Asymptotic Cost

In the general case, the asymptotic cost of a permutation algorithm is the average cost over all λ and ρ. But the use of permutation algorithms implies the assumption that at any given time some records are expected to be accessed with a higher probability than others. Without this expectation, it must be assumed that records are accessed entirely at random, and no amount of ordering would increase the chance of the next-accessed record being closer to the front of the list. Therefore analyses of asymptotic cost usually assume some constraints on the access strings that ρ can contain. A common assumption is that each record k has a fixed probability of access P_k throughout ρ. It is often further assumed that the fixed probabilities of record accesses follow a known probability distribution.

1.1.2. Worst Case Cost

The worst case cost of a permutation algorithm α is the maximum value of $C_\alpha(\lambda, \rho)$ over all λ and ρ. Note that, by the given definition of cost, the worst case is bounded above by n since cost is measured *per access*.

1.1.3. Amortized Cost

Worst case analyses often take the worst case of any step in a process and multiply it by the number of steps. In many processes, it is impossible for that worst case to occur at every step. Amortized analysis takes this into account and gives (usually worst case) analyses of algorithms on a multiple-instruction basis, which can yield a tighter bound than straight worst case analyses.

1.2. Convergence to Steady State

It is expected that the results of applying a permutation algorithm will cause records in the list that are more frequently accessed to be moved closer to the front of the list. However, it is unreasonable to expect the list to ever converge to and remain at a perfect ordering based on access probabilities. Thus the algorithms are expected to approach a *steady state*, where many further permutations are not expected to increase or decrease the expected search time significantly. Note that this steady state is not any particular list configuration, but rather a set of configurations that all have expected search costs close to the same value, which will be the value of the cost function $C_\alpha(\lambda, \rho)$ for large $|\rho|$ (the asymptotic cost of the algorithm). The amount of time, or number of accesses, required to approach the steady state is the *convergence* of the algorithm.

It is not well defined how close the expected cost of a configuration must be to the algorithm's asymptotic cost in order to be considered part of the steady state. Algorithms that cause extreme permutations will cause larger changes to the expected cost than do algorithms that make minor changes. For example, *move-to-front* makes a much larger change than *transpose*; thus, even when *move-to-front* approaches its steady state, the cost of a search is expected to have a larger deviation from *move-to-front's* asymptotic cost than the deviation expected by *transpose.* The steady state can thus be said to include all possible configurations, with each configuration given a probabilistic weight as to how frequently it is expected to occur.

1.2.1. Locality

As stated earlier, it is commonly assumed that the probability of access for each record is fixed for any given ρ. Since it is also usually assumed that accesses are independent of each other, the average case analyses tend to assume that all access sequences (which have the same set of access probabilities) are equally likely.

This assumption fails to model a common attribute of access sequences called *locality*, where subsequences of ρ may have relative frequencies of access that are drastically different from the overall relative frequencies. For example, consider a list of 26 records with keys "a" through "z". Each record is accessed exactly ten times, so that the fixed probability of each record is $\frac{1}{26}$. Whenever a record is found, it is moved to the front of the list if it is not already there.

First, let ρ consist of 10 repetitions of a string of "a" through "z" (alphabetically ordered), so that accesses to the same record are always spaced 26 apart. In this case, all accesses (except the first) to each record will take 26 probes. Multiplying the number of records (26) by the number of accesses to each record (9—not counting the first) by the number of probes for each access (26) gives the total number probes for all but the first access to each record. The first access will take between 26 and the key's location in the alphabet. For example, the first access to a can take between 1 and 26 probes, since a could be anywhere in the list. But the first access to d cannot be less than 4 since a, b, and c have been accessed and therefore placed ahead of d. Assuming the best case (when λ is initially alphabetically ordered), the total number of probes for first accesses is just the sum from 1 to 26. Thus the *best case cost* of this algorithm given ρ will be

$$\frac{26(9)(26) + \sum_{i=1}^{26} i}{260} \quad = \quad 24.75$$

Now consider the *worst case cost* of a different sequence ρ which accesses “a” ten times in a row, followed by ten accesses to “b”, continuing until it ends with ten accesses to “z”. As in the previous case, the number of probes for the first access to each record is between 26 and the key's location in the alphabet. All other accesses take only one probe. The *worst case cost* in this example will be

$$\frac{26(9)(1) + \sum_{i=1}^{26} 26}{260} \quad = \quad 3.5$$

Note that the worst case of the second example is far better than the best case of the first example. This demonstrates the fact that the cost of this permutation algorithm can differ greatly for the same fixed probabilities if accesses in ρ to the same record tend to cluster together instead of being dispersed throughout ρ.

These two examples highlight the change in behavior based on the *locality* of the search sequence. Permutation algorithms are designed not necessarily just to try to order the list of records by their total frequencies of access. The algorithms can also try to put records near the front that have been accessed more frequently in the recent past. This is usually desirable since many access sequences (such as variable accesses in a program or words in a document) tend to demonstrate locality.

Denning and Schwartz [1972] define the following *principle of locality* for page accesses in operating systems, but it applies equally well in cases of record or variable accesses for most programming contexts: (1) during any interval of time, a program distributes its accesses non-uniformly over its pages; (2) taken as a function of time, the frequency with which a given page is accessed tends to change slowly, i.e. it is quasi-stationary; and (3) correlation between immediate past and immediate future access patterns tends to be high, whereas the correlation between disjoint access patterns tends to zero as the distance between them tends to infinity. This principle, which defines locality in programs' accesses to pages of memory, can be directly translated into a principle describing locality in access sequences' accesses to records in a list.

1.2.2. Measures of Convergence

Bitner [1979] proposes a measure of convergence (called *overwork*) as the area between two curves on a graph:

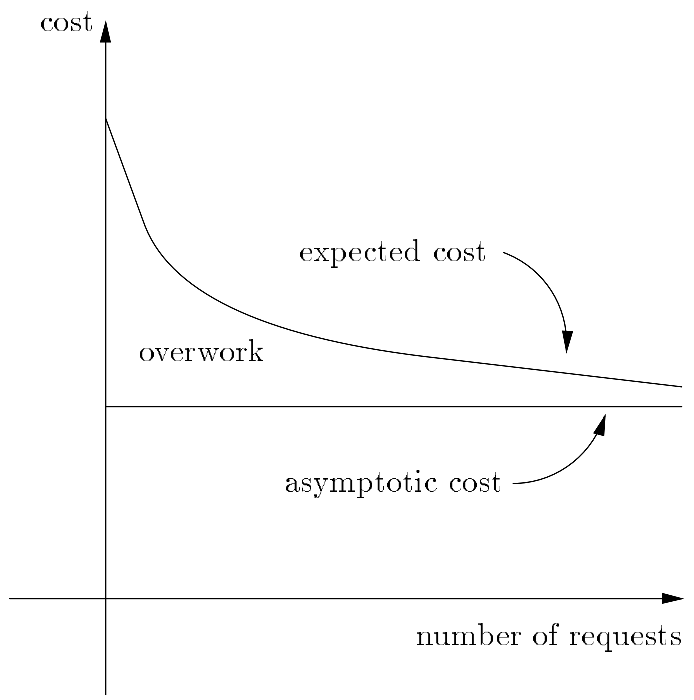

Figure 2.1 Bitner's Overwork Measure of Convergence

The horizontal (independent) axis is the number of record accesses performed (a range from 0 to ∞). The first (and usually higher) curve is the average cost of an algorithm as a function of the number of accesses. When only a few accesses have been made, this curve can be expected to be high since the list still has close to a random ordering, making it unlikely that records with high access probabilities will be near the front. This curve approaches the asymptotic cost of the algorithm as the number of accesses increases. The asymptotic cost of the algorithm is the second curve (a straight horizontal line), and the overwork is defined as the area between the expected cost curve and its asymptote. An algorithm is said to converge faster than another algorithm if its overwork area is less.

One way of looking at the overwork measure is to consider the amortized cost in terms of overwork. For example, if the overwork of a permutation algorithm is $2N$, then after N accesses at most 2 "extra" probes per access occurred. After N^2 accesses the amortized (average) "extra" cost per access is $O(N^{-1})$.

When convergence is expressed as a measure of time, an interesting question is how quickly an algorithm adjusts to changes in locality during search sequences. Tradeoffs between swift convergence and low asymptotic cost will be shown for several classes of permutation algorithms, but absolute measures of convergence for algorithms are rare.

1.3. Relative Measurements

The difficulty in finding absolute measures for permutation algorithms makes relative measurements desirable. Measuring algorithms relative to each other is useful both when no direct measure can be obtained for a given algorithm, and when the comparison between measures for two or more algorithms is not possible. Relative comparisons may be based on any of the measures mentioned in the previous sections.

Costs of algorithms are often compared to the cost of the *optimal static ordering,* in which the records are initially ordered by their static probabilities of access and left in that order throughout the access sequence. The optimal static ordering is not a permutation algorithm by our definitions, since it uses knowledge about probabilities that are assumed to be unavailable before the searches begin. This ordering provides a worst case cost of no more than $n/2$ (when the probabilities are equal), and a best case approaching 1 (as the differences in probabilities increase).

It should be noted that the optimal static ordering is not optimal overall. If locality of accesses is allowed, it is possible for self-adjusting algorithms to have lower cost than the optimal static ordering. Recall the examples given in Section 2.2.1, where ten calls are made to each of 26 keys. An optimal static ordering would require an average of 13.5 probes for any ρ of the type used in those examples. This is less than the best case with no locality (24.75), but more than the worst case with complete locality (3.5). Thus the relative effectiveness of permutation algorithms as compared to the optimal static ordering is heavily dependent on the degree of locality in the access string ρ.

In order to compare the relative convergence of *move-to-front* and *transpose,* Bitner [1979] suggests a modification of his overwork measure that uses the expected cost of *move-to-front* for the higher curve of his graph, and the expected cost of *transpose* (rather than the asymptotic cost of *move-to-front*) for the lower curve. A generalization of this measure allows us to compare the convergence of any two permutation algorithms by using their respective cost curves as the upper and lower curves of the graph, and examining the resultant area between them. This could be further generalized to compare the convergences of several permutation algorithms to each other, by choosing a permutation algorithm (not necessarily one of those in the set to be compared) as a metric and comparing the areas resulting from using its cost curve as a lower (or upper) bound to the cost curves of each of the other algorithms. Bitner chose the cost curve of *transpose* as a lower bound because he was only interested in the relative convergence of *move-to-front* and *transpose.* In the general case of using a single algorithm as a ruler to compare several algorithms, *transpose* still seems a good choice due to its low asymptotic cost and stability.

2. KNOWN ALGORITHMS AND ANALYSES

There is a wealth of information about permutation algorithms in the literature. Bitner [1979] and Gonnet et al. [1981] provide surveys of the most common algorithms and their analyses. However, much work has followed since these surveys appeared, consisting of a few new algorithms, but primarily of more enlightening relative measures *between* existing algorithms.

Many analyses involve unrealistically simple distributions, such as all (or all but one) of the records having the same probabilities of access; or show that the asymptotic cost of algorithms that depend on some constant parameter improve monotonically as the parameter increases (or decreases, in a very few cases). Although the proofs are complex, the results are generally intuitive, and will be treated as such in this survey. A better feel for the effects of these parameters may be gained through the tables provided in the section on empirical comparisons.

2.1. Move-to-front

The *move-to-front* algorithm was used in our preceding example of the effects of locality, and is by far the most commonly analyzed of the algorithms presented here. When the accessed record is found, it is moved to the front of the list, if it is not already there. All records that the accessed record passes are moved back one to make room.

The *move-to-front* algorithm tends to converge quickly, but has a large asymptotic 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.

The asymptotic cost for a general ρ has been shown by several analyses [McCabe 1965; Burville and Kingman 1973; Knuth 1973; Hendricks 1976; Rivest 1976; Bitner 1979] to be:

$$1 + 2 \sum_{1 \le i \le j \le n} \frac{P_i P_j}{P_i + P_j}$$

This cost includes only the search time; it could be as much as twice this value if the time to permute the records were included.

Hendricks [1972] applies Markov chains to the same problem, obtaining the same solution with possibly less work. He extends the application [1973] to *move-to-k* strategies, which cause the records ahead of location k to move backwards on the list.

Rivest [1976] shows that the expected cost of *move-to-front* for Zipf's distribution $\big(p_i = 1/(iH_n)\big)$ is $O(\ln n)$ times faster than a random ordering of the records.

Gonnet et al. [1981] give expected cost formulas for *move-to-front* making several assumptions about the distributions of access probabilities, and compare them to the expected cost of the optimal static ordering:

$$\text{Zipf's law:} \quad p_i = 1/(iH_n), \quad OPT = n/H_n$$
$$cost = 2 \ln 2 \frac{n}{H_n} - \frac{1}{2} + o(1)$$
$$\frac{cost}{OPT} \le 2 \ln 2 = 1.38629\ldots$$

$$\text{Lotka's Law:} \quad p_i = 1/(i^2 H_n^{(2)}), \quad OPT = H_n / H_n^{(2)}$$
$$cost = \frac{3}{\pi} \ln n - 0.00206339\ldots + O\left(\frac{\ln n}{n}\right)$$
$$\frac{cost}{OPT} \le \frac{\pi}{2} = 1.57080\ldots$$

$$\text{Exponential dist.:}\quad p_i = (1-a)a^{i-1}, \quad OPT = 1/(1-a)$$

$$\begin{aligned} cost &= 1 + 2\sum_{j=1}^{\infty} \frac{a^j}{1+a^j} \\ &= O(\ln^3 a) - \frac{2\ln 2}{\ln a} - \frac{1}{2} - \frac{\ln a}{24} \\ \frac{cost}{OPT} &\le 2\ln 2 = 1.38629\ldots \end{aligned}$$

$$\text{Wedge dist.:}\quad p_i = \frac{2(n+1-i)}{n^2+n}, \quad OPT = \frac{n+2}{3}$$

$$\begin{aligned} cost &= \frac{4(1-\ln 2)}{3}n - H_n + \frac{5(1-\ln 2)}{3} + O(1) \\ \frac{cost}{OPT} &\le 4(1-\ln 2) = 1.22741\ldots \end{aligned}$$

Bitner [1979] gives the overwork of *move-to-front* as

$$OV_{MTF} = \frac{1}{2}\sum_{1\le i<j\le n}\left(\frac{p_i - p_j}{p_i + p_j}\right)^2$$

He then shows that this is less than $n(n-1)/4$ for every probability distribution with $n > 2$. If $p_1 = 0$ and $p_i = 1/(n-1), \quad 2 \le i \le n$, then

$$OV_{MTF} = \frac{n-1}{2}$$

(Although this distribution is a bit simple, it will be used later for comparison with a similar analysis for *transpose*.) For Zipf's law he gives a complex formula, which for large n simplifies to $OV_{MTF} \approx .057n^2$.

2.2. Transpose

The accessed record, if not at the front of the list, is moved up one position by changing places with the record just ahead of it. Under this method, a record only approaches the front of the list if it is accessed frequently. This algorithm has a minor advantage in that it is also efficient to implement in sequential arrays.

The slower record movement gives *transpose* slower convergence, but its stability tends to keep its steady state closer to optimal static ordering. If there is little locality in the accesses, this results in a lower asymptotic cost. Intuitively, *move-to-front* has a high asymptotic cost due to its potential to make big mistakes by moving a record all the way to the front on the basis of a single access. Unless that record is accessed again in the near future, the only effect of this move is to increase the expected search time for many other records. *Transpose* avoids this potential error by being far more conservative in its record moves. A record can only be advanced to the front if it is accessed frequently. In cases where there is much locality in the accesses, *transpose* may be too slow in adjusting to the changes of locality, and a more radical algorithm such as *move-to-front* may be justified.

Rivest [1976] gives the average search time of *transpose* as:

$$\frac{\sum_{\pi}\left[\left(\prod_{i=1}^{n} p_i^{\delta(i,\pi)}\right)\sum_{j=1}^{n} p_j \cdot \pi(j)\right]}{\sum_{\pi}\prod_{i=1}^{n} p_i^{\delta(i,\pi)}}$$

where π denotes a possible ordering of the records, $\pi(j)$ denotes the j^{th} record in that ordering, and $\delta(i,\pi)$ denotes the quantity $i - \pi(i)$, or the distance that record i will be displaced from its original location in π.

Bitner could not find a general overwork formula for *transpose,* but in the case of $p_1 = 0$ and $p_i = 1/(n-1), \quad 2 \le i \le n$, he showed it to be:

$$OV_{TR} = \frac{n^2 - 1}{6}$$

It should be noted that this is $O(n)$ greater than his bound for *move-to-front.* Bitner gives analysis for both algorithms under another (even more simple) distribution, with the same results. He also manually calculates the values of overwork for *transpose* 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 *move-to-front.*

Gonnet et al. [1981] show that *transpose* may take as many as $\Omega(n^2)$ accesses to reach within a factor of $1 + \epsilon$ of the steady state behavior. For this result, they assume that the expected cost of the steady state of *transpose* is less than twice the expected cost of the optimal static ordering. This is generally true, but not always (see the section on comparisons between algorithms). Thus, since they only wanted to show that *transpose may* take this long, their proof is valid.

2.3. Count

A count is kept of the number of accesses to each record. After each access, the accessed record's count is incremented. That record is then moved forward to the first position in front of all records with lower counts. Thus, records in the list are always ordered by increasing value of their counts. At the end of the access sequence the list will be perfectly ordered by the access probabilities of the records, which is an optimal static order for that list. Note that during the access sequence the list will not be in the optimal static order, but will approach it as the sequence progresses. This method has obvious disadvantages: it uses extra memory on the order of the size of the list, and is relatively insensitive to locality in accesses.

Bitner [1979] suggests a variation on counts called the *limited difference rule* to limit the potential size of the count fields. The count field of a record i stores the difference of the access counts between record i and record $i-1$ (recall that records are ordered by access counts). Thus an access to a given record will cause the count of record i to be incremented, and the count of record $i+1$ to be decremented. These difference counts may not be increased above a limiting value. He shows that the performance approaches the optimal static ordering as the limit on the difference increases, and is close even for small limits (see Section 4.5, Table 4.3).

2.4. Move-ahead-k

Move-ahead-k is a compromise between the relative extremes of *move-to-front* and *transpose*. In the simple case, *move-ahead-k* simply moves the record forward k

positions. By this definition, if f is the distance to the front, *move-to-front* is *move-ahead-f* and *transpose* is *move-ahead-*1.

This can be generalized to move a percentage of the distance to the front, or some other function based on the distance. Other parameters to the function may also be of interest, such as how many accesses have taken place so far. As usual, if the distance to be moved exceeds the distance to the front, then the record is only moved to (or left at) the front.

Implementation of *move-ahead-k* is straightforward for constant values of k, by keeping a queue of pointers to the last k records probed. Searching back k positions may also be possible, but only if the list provides backward links. Searching forward is undesirable due to the fact that this would effectively double the search time by requiring a second traversal of the list. Allowing k to be a function of the distance to the front implies similar problems. It might seem feasible to maintain a back pointer, advancing it to maintain the correct distance from the currently probed record, but this has the same cost as performing a second traversal at the end of the search.

The *move-ahead-k* algorithm was initially proposed by Rivest [1976]. Gonnet et al. [1979] theorize that, for $j > k$, *move-ahead-j* converges faster than *move-ahead-k,* but at the penalty of a higher asymptotic cost. Section 4.5, Table 4.2 gives empirical evidence to support this.

2.5. JUMP

Hester and Hirschberg [1985] propose a randomized method of leaving a back pointer behind during the search, to be used later as the destination for moving a record forward. This avoids the costly second traversal of the list described in relation to *move-ahead-k* algorithms, where each access might require reading pages from slower secondary memory, such as when the records are large or when there are many records. Since moving the pointer forward slowly as the probes progress would be the same as performing a linear traversal at the end of the search, the pointer must occasionally be jumped forward to the record that was just probed. The pointer is advanced to the probed record if and only if the probed record is not the accessed record, and a boolean function $JUMP$ is true. $JUMP$ can be a function of variables such as the current position of the back pointer, the position of the probed record, and/or the number of accesses preceding the current one. They present definitions of $JUMP$ functions that demonstrate (on the average case) the same behavior as *move-to-front, transpose,* and *move-ahead-k* (for k being any desired constant or fraction of the distance to the front of the list).

2.6. Meta Algorithms

The following meta algorithms may be applied in conjunction with permutation algorithms. At the time when a permutation algorithm would normally be invoked, the meta algorithm decides whether to call the permutation algorithm or leave things as they are. The purpose is to slow the convergence of the permutation algorithm by not moving records only on the basis of single accesses, to reduce the effects of a one-time access to a record.

2.6.1. Move Every k^{th} Access

McCabe [1965] considers applying the permutation algorithm only once every k accesses to reduce the time spent reordering the list. He shows that, independent of the value of k, the asymptotic cost of searches would be exactly the same as if the meta algorithm were not used. However, the meta algorithm requires k times the number of accesses required by the permutation algorithm (by itself) to approach this asymptote.

2.6.2. *k-in-a-row*

The permutation algorithm is applied only if the accessed record has been 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. This has the advantage of not requiring as much memory as do *count* rules, since it only needs to remember the last record accessed, and a single counter for the number of recent consecutive accesses to that record.

Kan and Ross [1980] propose this scheme and use semi-Markov chains to analyze *k-in-a-row* policies. They show that the proportion of time that the element with the i^{th} largest probability of access will spend in the i^{th} location approaches 1 as k approaches ∞.

Gonnet et al. [1979] show that, for $k = 2$, the convergence using *move-to-front* is the same as that of *transpose.* They also suggest a minor modification called the *batched k* heuristic, which groups accesses into batches of size k, and applies the permutation algorithm only when all k accesses in a batch are to the same record. This tends to slow down convergence a bit more than *k-in-a-row,* and provides a lower asymptotic cost.

Bitner [1979] suggests a modification to the *k-in-a-row* strategy called *wait c and move,* which incorporates finite counters for each record. After a record has been accessed c times (not necessarily in a row) the permutation algorithm is applied and the counter for that record is reset. He also suggests a modification where *all* of the counters are reset whenever one reaches the limit. These algorithms are compared in Section 4.5, Table 4.3.

2.6.3. Probabilistic Application of Move Rules

Kan and Ross [1980] suggest an alternative to *k-in-a-row* that slows down convergence by applying a permutation algorithm to the accessed record i with probability α_i, $1 \leq i \leq n$. They show that if the α_i are equal for all i, then the asymptotic cost will be the same as the original algorithm without the probabilities. They also show that the results for *transpose* are independent of the values of α_i.

Oommen [1984] proposes a method he calls *stochastic move-to-front* which combines the properties of Kan and Ross's method with those of *count.* Instead of using a single fixed probability, each record r_i has its own probability p_i which is used to determine whether that record, after being accessed, is to be moved. Oommen only considers *move-to-front* for the permutation algorithm. Initially all probabilities are 1. After each access, the access probability of the accessed record is decreased by a multiplicative constant, and all other probabilities remain unchanged. Note that the probabilities provide information equivalent to that of a count for each record: smaller probabilities correspond to larger frequency counts. After every T accesses, the list is sorted on the values of the probabilities. As the probabilities decrease, the list will be less likely to change during T accesses. Thus by using a sort that is $O(n \log n)$ in the worst case and $O(n)$ if the list is nearly sorted, the cost of applying this sort is minimal after some initial settling down. Although this scheme may cause the record order to converge to any of the $n!$ possibilities, Oommen shows that the multiplicative constant for changing probabilities may be chosen to make the probability of convergence on the optimal static ordering as near unity as desired.

2.6.4. Implicit Data Structures

Frederickson [1984] considers the case where there are also fixed probabilities of searching for values other than the keys. Although this normally leads to tree structures, he proposes algorithms that use linear lists of binary search lists. Records are stored in groups, each group being maintained in lexicographic order to allow binary search within the group. The list of groups is searched linearly until the desired record is found. When a record is found in group i, it is moved into the first group, and records are chosen at random from groups 1 to $i-1$ to be moved back one group. Thus this algorithm applies an analogue of the *move-to-front* algorithm over a different set of objects. A similar algorithm is suggested that uses *transpose* for the group moves. He shows that these algorithms have average expected search costs for the i^{th} record of $O(\log \min\{1/p_i, n\})$. He extends these algorithms to include a similar bound for unsuccessful searches (substitute q_i for p_i in the above formula, where q_i is the probability of accessing a range of missing keys). The extension involves applying the permutation only every n accesses and performing the movement during the intervening accesses to minimize extra traversals.

2.7. Hybrids

Move-to-front and *transpose* clearly have tradeoffs concerning convergence and asymptotic cost. If it is known ahead of time that the number of accesses will be small, *move-to-front* is probably the better algorithm, whereas *transpose* is better if the number of accesses is expected to be large. A hybrid of these or other algorithms that try to get the best of both measures might be feasible.

Bitner [1979] proposes a simple hybrid that uses the *move-to-front* algorithm until its steady state is approached, and then switches to *transpose.* The difficulty of deciding when to switch is a major problem. Since *move-to-front* performs best when the list has a high degree of locality, it is even unclear if the best time to switch to *transpose* can be determined for a fixed number of accesses. Bitner suggests switching when the number of requests is between $\Theta(n)$ and $\Theta(n^2)$. He refers to his 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 based on specific observations derived by assuming Zipf's law.

Lam et al. [1981] propose a combination of *transpose* and *count,* where transpose is used until a steady state is reached, and counts are maintained thereafter. Since *count* is likely to make serious mistakes during the early accesses (similar to those made by *move-to-front*), the intent is that *transpose* will be used during this period, allowing count to take over after things settle down. They gain a minor savings in time by avoiding maintenance of the counts while transpose is running.

To give elements reasonable weights when *count* is started, the elements are ordered on the value of $f_i + n - i$, where f_i is the frequency count of the i^{th} record in the list, and $n - i$ is the distance from that record's location to the end of the list. Thus the record's location in the list is used as a weight, as is its frequency count. They show that their generalized *count* algorithm performs better than other methods of re-arrangement based on counts. They were unable to define a suitable weighting scheme for a similar hybrid using *move-to-front* in place of *transpose*.

In this study, Lam et al. fail to give a definition of how to detect a steady state. One possibility is to maintain a finite queue of size k of the costs associated with the most recent accesses, and define a steady state as one in which the average cost of the last $k/2$ accesses is not more than some ϵ different from the average cost of the $k/2$ accesses preceeding the last $k/2$ accesses. The values of k and ϵ may be tuned to reflect the particular permutation algorithm in question. For example, *move-to-front* might require a larger value of ϵ then *transpose,* since it makes more extensive changes during a small number of accesses.

Tenenbaum and Nemes [1982] suggest two classes of hybrids. The first is $POS(k)$ for $1 \leq k \leq n$. If the accessed record is found in a position $\leq k$, it is transposed with its predecessor, otherwise it is moved to the k^{th} position, shifting all intervening records back one. Note that $POS(1)$ is *move-to-front,* while $POS(n-1)$ is *transpose.*

The second class proposed by Tenenbaum and Nemes is $SWITCH(k)$, which is the same as POS except the uses of *move-to-front* and *transpose* are reversed: an accessed record found in a location $\leq k$ is moved to the front, and others are transposed.

3. COMPARISONS BETWEEN ALGORITHMS

Since it is often difficult to obtain a standard measure for permutation algorithms, it is a common practice to simply measure the algorithms directly against each other, either theoretically or through simulation. The following sections describe these methods of comparison.

3.1. Optimal Algorithm?

Before we can ask whether an optimal memoryless strategy exists, we must define what is meant by "*optimal.*" Rivest [1976] defines a permutation rule to be *optimal* over a set of rules if it has the least cost of those rules for all probability distributions and all initial orderings of the keys. He conjectures that *transpose* is an optimal memoryless algorithm under this definition. Yao (in a personal communication reported by Rivest) supports this conjecture by proving that, if an optimal memoryless algorithm exists, it must be *transpose.* His proof gives a distribution for each n for which *transpose* has least cost over all rules. Kan and Ross [1980] also provide evidence to support Rivest's conjecture by showing that *transpose* is optimal over the rules that move a single element a number of places closer to the front of the list, when all records except one have an equal probability of access. This result also assumes no locality of accesses.

Anderson et al. [1982] provide a complex counterexample to Rivest's conjecture by presenting a single permutation algorithm that outperforms *transpose* in the average case for a single access probability distribution. It should be noted that their algorithm requires an arbitrary definition of the permutations to be performed for each of the possible record locations in which the accessed record might be found, and thus the rule itself is dependent on the number of records rather than being a general rule that applies to any list. The algorithm also radically reorganizes the list, so it is not in the class of algorithms that only move the accessed record some distance towards the front of the list. Thus the algorithm is memoryless in terms of information concerning accesses on the records, but the algorithm itself requires $\Theta(n^2)$ space to define the permutations.

Therefore, no memoryless algorithm can be optimal over all ρ, and the following analyses for different algorithms cannot be expected to demonstrate a complete superiority for any given algorithm. Whether this holds true for algorithms which only move a single record forward in the list is still unknown.

3.2. Asymptotic Cost

The asymptotic cost of *move-to-front* has been shown by many [McCabe 1965; Knuth 1973; Hendricks 1976; Rivest 1976; Bitner 1979] to be at most twice the asymptotic cost of the optimal static ordering. Rivest [1976] shows that the asymptotic cost of *transpose* is less than or equal to that of *move-to-front* for every probability distribution.

Bitner [1979] shows that *count* is asymptotically equal to the optimal static ordering, and that the difference in cost between the two decreases exponentially with time, so that *count* is not much worse than the optimal static ordering even for fairly small numbers of accesses. He further shows that for all distributions except (1) the normal distribution, and (2) the distribution in which a single key has probability one, the *wait c and move* algorithms will not approach the optimal ordering as c approaches infinity. He finally shows that the *limited difference rule* is superior to the *wait c and move* rules in both asymptotic cost and convergence. This is supported in Section 4.5, Table 4.3.

3.3. Worst Case

The worst case cost of *move-to-front* has been shown by many [Burville and Kingman 1973; Knuth 1973; Hendricks 1976; Rivest 1976; Bitner 1979; Bentley and McGeoch 1985; Sleator and Tarjan 1985] to be no more than twice that of the optimal static ordering. The worst case cost of *move-to-front*, when there is no locality and the accessed record is always at the end of the list, is approximately n. The cost using the optimal static ordering in the same situation is approximately $n/2$. This does not show, but does suggest, that *move-to-front* is *never* worse than twice the optimal static ordering.

Bentley and McGeoch [1985] point out with an amortized argument that although the worst-case performances of *move-to-front* and *count* are no more than twice that of the optimal static ordering, the worst-case performance of *transpose* could be far worse. This can easily be seen by considering a case where two records are alternately accessed many times. They will continue to exchange places without advancing towards the front of the list.

Gonnet et al. [1981] 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. [1985] show that this is an upper bound on the average case for all distributions. The distribution provided by Gonnet et al. then serves to make this bound tight.

Most relative analyses for the worst case of *move-to-front* are by techniques that are not generally applicable. Sleator and Tarjan [1985] present a more generalized technique. They give an amortized analysis, which they extend to show that moving a record any fraction of the distance to the front of the list will be no more than a constant times the optimal off-line algorithm. The constant is inversely proportional to the fraction of the total distance moved. This result is valuable in justifying algorithms such as *move-ahead-k* (for k a fraction of the distance to the front) and *JUMP*.

3.4. Convergence

Bitner [1979] shows that, while *transpose* is asymptotically more efficient, *move-to-front* often converges more quickly. As pointed out when presenting the algorithms, he demonstrates that his overwork measure for *move-to-front* is a factor of n less than *transpose* for a variety of probability distributions. He also shows that *move-to-front* converges faster than *transpose* by using his second measure of convergence (the number of accesses, A, required for the expected cost of *transpose* to be less than that of *move-to-front*) by pointing out that A increases quadratically with n, which indicates that *move-to-front* converges faster than *transpose.*

3.5. Empirical Comparisons

Most of the literature deals with the behavior of the permutation algorithms in a strictly theoretical manner. In general, the proofs verify intuitive expectations that varying a parameter of the algorithm will cause the expected cost of the algorithm to approach some asymptote. The following tables were provided by some authors as support to their theorems.

The first table is a compilation of two tables provided by Bentley and McGeoch [1985], comparing the results of running several common algorithms applied to files containing programs and text.

File	Distinct words	Total words
P1	100	480
P2	107	431
P3	117	1,176
P4	181	1,456
T1	471	1,888
T2	498	1,515
T3	564	3,296
T4	999	5,443
T5	1,147	7,482
T6	1,590	7,654

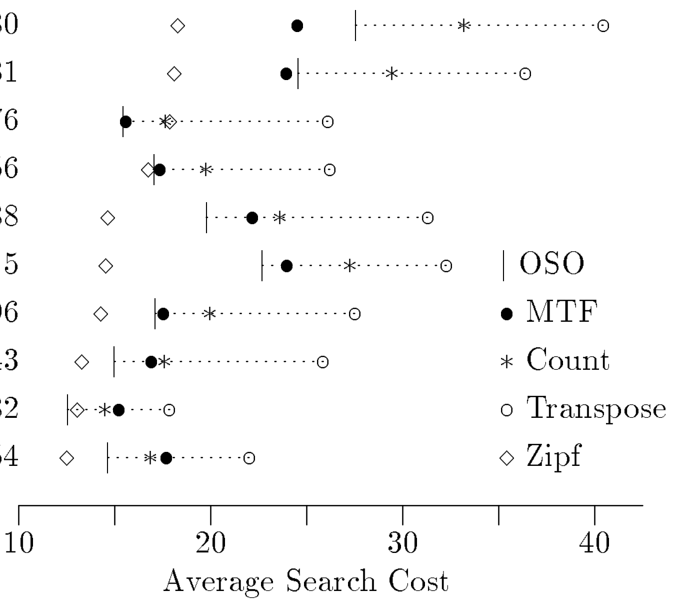

Table 4.1 Measured Average Search Costs of Several Algorithms

The 'P' and 'T' in the file names indicate whether the file is Pascal source or English text. OSO stands for the expected cost if the records are left in the optimal static ordering, and MTF is the *move-to-front* algorithm. The mark for Zipf indicates the expected cost of an optimal static ordering, assuming that the probabilities follow Zipf's distribution. Bentley and McGeoch compare this to the expected cost of the optimal static ordering to suggest that the probabilities of the words are closer to the Zipf distribution than a normal distribution (where the expected cost would be about half the number of distinct words in the file).

The following table submitted by Tenenbaum [1978] contains a condensation of a more extensive table of simulations on *move-ahead-k* algorithms. Files of various sizes (N) containing data which was distributed by Zipf's law were accessed 12,000 times, for $1 \leq k \leq 7$. The best value of k (among those tested) is given for the sizes tested.

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 4.2 Best k for Different Sized Files

Note that in this table there is no optimal value of k. Since the number of accesses remains constant, the number of records in the list is shown to have a direct correlation to the optimal value of k.

Bitner [1979] compared his *limited difference rule* to his two meta-algorithms, using *move-to-front* and *transpose* for the permutation algorithms. He gives the average of 200 trials of 1,000 accesses on a list of nine elements whose probabilities are given by Zipf's law.

$c =$	0	1	2	3	4	5
Limited Difference rule (maximum difference = c)	3.9739	3.4162	3.3026	3.2545	3.2288	3.2113
Wait c, *move-to-front* and clear	3.9739	3.6230	3.4668	3.3811	3.3285	—
Wait c, *transpose* and clear	3.4646	3.3399	3.2929	3.2670	3.2501	—
Wait c and *move-to-front*	3.9739	3.8996	3.8591	3.8338	3.8165	3.8040
Wait c and *transpose*	3.4646	3.3824	3.3576	3.3473	3.3312	3.3272

Table 4.3 Costs of Count Algorithms Relative to c

Comparing these costs to 3.1814 (the expected optimal cost under Zipf's law) gives an indication of how fast they converge as c varies.

Oommen [1984] gives some simulations on his *stochastic move-to-front* method that indicate fast convergence on the optimal ordering. Each test consisted of the average of ten trials consisting of 4,000 accesses on a list of words distributed one of three ways (s_i is the probability of access):

DIST1: $s_i = k_1 \cdot (N - I + 1)$ where $k_1 = 1/\sum_{i=1}^{N} i$

DIST2: $s_i = k_2/i$ where $k_2 = 1/\sum_{i=1}^{N} i^{-1}$

DIST3: $s_i = k_3/2^i$ where $k_3 = 1/\sum_{i=1}^{N} 2^{-i}$

Note that DIST2 is Zipf's law. Sorting cycles occurred every 100 accesses. The table shows the accuracy, $AC\hat{C}(x)$, and the average number of moves performed each cycle, $\hat{K}_T(x)$, after x cycles have taken place. The accuracy is defined as the fraction of distinct pairs of records that are correctly ordered (by the optimal static ordering) relative to each other. Thus an accuracy of 1 is the optimal static ordering, and an accuracy of 0.5 is what is expected for an initial random ordering of records.

N	Distribution	$AC\hat{C}(0)$	$AC\hat{C}(10)$	$\hat{K}_T(0)$	$\hat{K}_T(10)$
	DIST1	0.48	0.92	35.10	0.30
10	DIST2	0.55	0.94	29.50	0.40
	DIST3	0.41	0.90	18.20	0.30
	DIST1	0.50	0.89	43.00	0.70
15	DIST2	0.50	0.90	33.80	1.00
	DIST3	0.51	0.83	21.50	0.20
	DIST1	0.52	0.87	52.00	1.00
25	DIST2	0.50	0.88	36.50	1.10
	DIST3	0.49	0.83	18.60	0.41
	DIST1	0.52	0.87	60.80	0.90
40	DIST2	0.48	0.87	37.00	1.40
	DIST3	0.46	0.87	19.70	0.40

Table 4.4 Simulations of *Stochastic Move-to-front*

The table shows that, after just ten cycles, the records are very near the optimal static ordering, and this ordering will tend to persist since the frequency of further record moves is low.

4. OPEN PROBLEMS

Since it has been shown that no single optimal permutation algorithm exists, it becomes necessary to try to characterize the circumstances that indicate an advantage in using a particular algorithm. There are several conditions to consider, which give rise to the following open questions concerning algorithms.

4.1. Direct Analyses

Move-to-front is the only permutation algorithm that has been extensively analyzed. *Transpose* is widely mentioned in the literature but authors merely state an inability to obtain theoretical bounds on its performance, and most of the work on other algorithms merely shows how their behavior changes as parameters vary. Direct bounds on the behaviors of these algorithms are needed to allow realistic comparisons. Bentley and McGeoch [1985] showed in their table of simulations of common permutation algorithms that Zipf's law seems to be a good approximation of the distribution of common files of words, so direct analyses under the assumption of accesses following Zipf's law would be valuable. Other distributions would also be interesting.

4.2. Locality/Convergence

Hendricks [1976] mentions the open problem of relaxing the assumption of independence for analysis of permutation algorithms. He suggests a model by which, when a record is accessed, its probability of access in the future is increased (or decreased). He does not suggest how other probabilities should be altered to account for this.

As yet, no good definition for locality of accesses has been applied to the problem of measuring self-organizing linear search. This is unfortunate, since taking advantage of locality is one of the main reasons for using these techniques in the first place, and the lack of definition restricts measures of convergence to very general cases.

4.3. Optimize Algorithms for a Given Level of Locality

Once an algorithm can be analyzed in terms of its performance for a given function of locality in an access sequence, developing algorithms that optimize convergence for that function of locality is an obvious step. A few hybrids were suggested, such as using *move-to-front* initially and then switching to *transpose*, which quickly converge to a steady state and also have a good asymptotic cost. As was pointed out, the best time to switch to *transpose* (and similar problems in other algorithms) is difficult to know when we assume no knowledge about the access sequence. However, if we allow knowledge of the approximate degree of locality of the sequence, we may be able to tailor these algorithms effectively.

Hybrids are not the only method of controlling response to locality. The *move-ahead-k* algorithm could also be considered, especially the extended definition that allows moving a function of the distance to the front rather than a fixed constant.

4.4. Choosing an Algorithm

Bentley and McGeoch [1985] give the following advice about choosing between the three permutation algorithms that are widely mentioned in the literature:

- *Move-to-front.* A linked-list implementation is preferable for most applications when using *move-to-front*. The search makes few comparisons, both in the amortized sense and when observed on real data; furthermore, it exploits locality of reference present in the input. The linked list implementation is natural for an environment supporting dynamic storage allocation and yields an efficient reorganization strategy.
- *Transpose.* If storage is extremely limited and pointers for lists cannot be used, then the array implementation of *transpose* gives very efficient reorganization. Its worst-case performance is poor, but it performs well on the average.
- *Count.* Although this heuristic does make a small number of comparisons, its higher move costs and extra storage requirements could be a hindrance for some applications. It should probably be considered only for applications in which the counts are already needed for other purposes.

The above comments concerning *move-to-front* can be generalized to include the various algorithms that move records forward a fraction of the distance to the front of the list. Similarly, the comments concerning *transpose* can be generalized to include algorithms that move records forward by a fixed distance. Meta algorithms may be applied to the above as well. It might be possible to derive rules for finding good values of the parameters associated with these algorithms for common distributions of access sequences. The algorithms' performances could likely benefit from tuning the values of such parameters for a particular application. Such tuning may be sufficient in itself to find good parameter values in reasonable time without the application of formal rules.

5. SUMMARY

Self-organizing lists have been in the literature for almost twenty years, and in that time many permutation algorithms have been proposed that move the accessed record forward by various distances, either constant or based on the location of the record or past events. We have presented these algorithms along with the analyses for them.

The majority of the literature deals with only a few of the permutation algorithms. Many analyses on the same algorithms are given, either demonstrating a new method of analysis or providing results based on different criteria. Almost all average case analyses assume no locality, which is unreasonable in many applications. While *move-to-front* has been shown to be better in some real applications than *transpose,* there are no guidelines for choosing between the algorithms that move the accessed record some distance between the two extremes.

ACKNOWLEDGMENTS

Our thanks to the referees, whose comments on the original draft led to considerable enhancements and other improvements. Also to V. Zielinski, the technical editor, whose comments and suggestions corrected several cases of unclear presentation.

REFERENCES

ANDERSON, E. J., NASH, P., AND WEBER, R. R. 1982. A Counterexample to a Conjecture on Optimal List Ordering. *J. Appl. Prob. 19,* 3 (Sept.), 730–732.

BENTLEY, J. L., AND MCGEOCH, C. C. 1985. Amortized Analyses of Self-Organizing Sequential Search Heuristics. *Commun. ACM 28,* 4 (Apr.), 404–411.

BITNER, J. R. 1979. Heuristics that Dynamically Organize Data Structures. *SIAM J. Comput. 8,* 1 (Feb.), 82–110.

BURVILLE, P. J. AND KINGMAN, J. F. C. 1973. On a Model for Storage and Search. *J. Appl. Prob. 10,* 3 (Sept.), 697–701.

CHUNG, F. R. K., HAJELA, D. J. AND SEYMOUR, P. D 1985. Self-Organizing Sequential Search and Hilbert's Inequalities. In *Proceedings of the* 17^{th} *annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, May 6-8), ACM (SIGACT), New York, NY, 217–223.

DENNING, P. J. AND SCHWARTZ, S. C. 1972. Properties of the Working-Set Model. *Commun. ACM 15,* 3 (Mar.), 191–198.

FREDERICKSON, G. N. 1984. Self-Organizing Heuristics for Implicit Data Structures. *SIAM J. Comput. 13,* 2 (May), 277–291.

GONNET, G. H., MUNRO, J. I., AND SUWANDA, H. 1979. Toward Self-Organizing Linear Search. In *Proceedings of 20th IEEE Symposium on Foundations of Computer Science* (San Juan, Puerto Rico, Oct.), 169–174.

GONNET, G. H., MUNRO, J. I., AND SUWANDA, H. 1981. Exegesis of Self-Organizing Linear Search. *SIAM J. Comput. 10,* 3 (Aug.), 613–637.

HENDRICKS, W. J. 1972. The Stationary Distribution of an Interesting Markov Chain. *J. Appl. Prob. 9,* 1 (Mar.), 231–233.

HENDRICKS, W. J. 1973. An Extension of a Theorem Concerning an Interesting Markov Chain. *J. Appl. Prob. 10,* 4 (Dec.), 886–890.

HENDRICKS, W. J. 1976. An Account of Self-Organizing Systems. *SIAM J. Comput. 5,* 4 (Dec.), 715–723.

Hester, J. H., and Hirschberg, D. S. 1985. Self-Organizing Search Lists Using Probabilistic Back-Pointers. Technical Report # 85–14, ICS Department, University of California, Irvine.

Kan, Y. C. and Ross, S. M. 1980. Optimal List Order Under Partial Memory Constraints. *J. Appl. Prob. 17,* 4 (Dec.), 1004–1015.

Knuth, D. E. 1973. A "Self-Organizing" File. *The Art of Computer Programming, Vol 3: Sorting and Searching,* Addison-Wesley, Reading, MA, 398–399.

Lam, K., Siu, M. K. and Yu, C. T. 1981. A Generalized Counter Scheme. *Theoretical Computer Science 16,* 3 (Dec.), 271-278.

McCabe, J. 1965. On Serial Files with Relocatable Records. *Operations Research* (July/Aug.), 609–618.

Oommen, B. J. 1984. On the Use of Smoothsort and Stochastic Move-to-front Operations for Optimal List Organization. In *Proceedings of the* 22^{nd} *annual Allerton Conference on Communication, Control, and Computing* (Monticello, IL., Oct.), University of Illinois, Urbana-Champaign, 243–252.

Rivest, R. 1976. On Self-Organizing Sequential Search Heuristics. *Commun. ACM 19,* 2 (Feb.), 63–67.

Sleator, D. D., and Tarjan, R. E. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM 28,* 2 (Feb.), 202–208.

Tenenbaum, A. 1978. Simulations of Dynamic Sequential Search Algorithms. *Commun. ACM 21,* 9 (Sept.), 790–791.

Tenenbaum, A. and Nemes, R. M. 1982. Two Spectra of Self-Organizing Sequential Search Algorithms. *SIAM J. Comput. 11,* 3 (Aug.), 557–566.