Self-Organizing Linear Search

J. H. Hester and D. S. Hirschberg

Technical Report #240

November, 1984

Abstract. 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 paper 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.

Self-Organizing Linear Search

J. H. Hester

D. S. Hirschberg

University of California, Irvine

ABSTRACT

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 paper 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.

INTRODUCTION

Sequential searches are performed on a list of initially unordered records. After a record is found, the list is permuted by some algorithm in an effort to place the more frequently accessed records closer to the front of the list, thus reducing the expected search time. An interesting question is what algorithms can be used for this permutation, and how do they perform relative to each other in terms of expected search time.

One common application in which this situation arises is a list of identifiers maintained by a compiler or interpreter. An example of this is the scatter table used by the UCI LISP system. Identifiers are hashed into a list of buckets, each of which is an unordered sequential list of identifier descriptions. The lists cannot be initially ordered since the frequencies of references to the identifiers are not known.

Since most programs tend to access some identifiers more often than others, those identifiers should be nearer the front of the list.

1. RESTRICTIONS

We will refer to algorithms that maintain self-organizing lists by permuting the records as *permutation algorithms.* 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 will be applied to the problem being considered. The primary assumption is that no knowledge exists concerning probabilities of accesses prior to the accesses themselves. It is however acceptable to know or assume that the access probabilities follow some distribution or general rule without knowing how that applies to individual records.

Only algorithms concerning linear lists will be considered. Although there is much literature concerning arranging the records in trees which dramatically reduces search time, sequential lists are still interesting due to savings in space.

In the general case, the permutation algorithm could completely reorder the list at any time. We will only consider algorithms that apply permutations after each access, and that the permutations only 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 fitting this restriction will leave the list unchanged whenever the first record is accessed. To avoid dwelling on details of insertion, it will be assumed that the list initially contains all records that will be accessed, and no others.

Finally, it will be assumed that the permutation algorithm requires no more than time proportional to the time taken to find the record. Most algorithms to be considered actually perform the permutation in constant time.

2. 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 $\rho[1], \rho[2], \ldots$ be the search sequence, such that $\rho[k]$ is the label of the record to be searched for on the k^{th} access. Let λ be the initial configuration of the list such that λ_k is the *location* of the record labeled k. The first record is in location 1 and the last record in a list of n records is in location n. Extending these definitions, let $\Lambda(\rho, k)$ be the configuration of the list after the first k accesses from sequence ρ, assuming initial configuration λ and application of algorithm α. Note that for all $1 \le i \le n$, $\Lambda(\rho, 0)_i = \lambda_i$.

2.1. Cost

The *cost* of a permutation algorithm α for a given λ and ρ is the average cost per access, in terms of the time (or work) to find the accessed record and the time to permute the records afterwards. Formally:

$$C_\alpha(\lambda, \rho) = \frac{\sum_{k=1}^{|\rho|} \Lambda(\rho, k-1)_{\rho[k]}}{|\rho|}$$

Recall that this only reflects the count of probes needed to find the record, not any extra cost to apply the permutation.

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 will be used.

2.1.1. 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 that at any given time some records can be expected to be accessed with a higher probability than others. Without this expectation, it must be assumed that records are accessed purely 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.

2.1.2. 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*.

2.1.3. 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.

2.2. Convergence

Since all algorithms start from an initially unordered list, they must spend some time with poor performance to begin ordering the list before the average costs begin to approach the expected asymptotic costs. The amount of time required for this is the *convergence* of the algorithm, and is particularly important if the accesses demonstrate *locality*.

2.2.1. Locality

As stated earlier, one common assumption made about ρ is that the probability of access for each record is fixed, so that the only access sequences included are those (of any length) that access each record the same percentage of the time relative to the other sequences in the set. It is usually assumed that accesses are independent of each other. Thus average case analyses tend to assume that *all* sequences are equally likely, which is not always true for strings of accesses.

It is fairly easy to see that simple permutation algorithms will tend to order the list, but it is unreasonable to expect the list to be in the perfect order (by the probabilities of access on the records) for any length of time. In the asymptotic case, there must be a ***steady state*** which consists of a number of list configurations all of which have expected access times close to that of the asymptotic cost.

This assumption fails to model a common feature of access sequences called *locality*, where subsequences of ρ may have relative frequencies of access 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 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

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

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

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

Note that the worst case of the second example is still 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 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 nonuniformly 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 [DEN72].

2.2.2. Measures of Convergence

The main idea of permutation algorithms is to order the list so that the records that are likely to be accessed tend to be near the front of the list. The list is never expected to become perfectly ordered, but should eventually reach one of many steady states where the expected time for an access remains within some small constant of the asymptotic cost for that algorithm. Once a steady state is reached, permutations based on further accesses are expected to continue to result in steady states.

Bitner proposes a measure of convergence he calls *overwork* as the area between two curves [BIT79]. The horizontal (independent) axis is the number of record accesses made (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 few accesses have yet 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 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.

Another measure of convergence could be the time (in terms of number of record accesses) needed for a permutation algorithm to a reach a steady state from a random initial state. As an alternate to his overwork measure, Bitner suggests a modification of this time measure for which the time required for the expected cost of *transpose* is less than that of *move-to-front*. (This measure could be generalized to compare any two algorithms, but Bitner was primarily interested in these two and thus only mentioned the measure relative to them.)

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

2.3. Relative Measurements

It is often desirable to compare permutation algorithms directly to each other. This is useful both when no results can be obtained for a given algorithm, and when the correlation between results for two or more algorithms is not clear. When comparing algorithms, any of the previous measures may be used.

Algorithms are often compared to the use 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 an algorithm by our definitions, since it uses knowledge about probabilities that are assumed to be unavailable at the beginning of the algorithm. 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 increases.

It should be noted that the optimal static ordering is not optimal overall. As soon as locality of accesses is allowed, it is possible for self-adjusting algorithms to have lower cost than the static order. Recall the examples given earlier with ten calls to each of 26 keys. An optimal static order would require an average of 13.5 probes for any ρ of this sort. 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.

3. KNOWN ALGORITHMS AND ANALYSES

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

3.1. Move-to-front

This is the algorithm used in the preceding example of the effects of locality. 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.

Gonnet, Munro, and Suwanda give analyses of the cost of *move-to-front* for a number of specific distributions of ρ [GON79]. Let P_i be the probability that record i will be accessed at any given time, assuming that the probabilities are fixed over all accesses. Thus $P_i =$ (the number of occurrences of record i in $\rho)/|\rho|$. The asymptotic cost for a general ρ has been shown by several analyses [MCC65, BUR73, KNU73, HEN76, RIV76, BIT79] to be

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

This counts only the search time; it would approximately double if the time to permute the records were included.

Hendrics applies Markov chains to the same problem, obtaining the same solution with possibly less work [HEN72]. He extends the application to *move-to-k* strategies, which cause the records ahead of location k to move backwards on the list [HEN73].

Gonnet, Munro and Suwanda give analysis for *move-to-front* assuming the probabilities follow known distributions including Zipf's Law, Lotka's Law, exponential distribution, and Wedge distribution [GON79].

3.2. 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.

The slower record movement gives the *transpose* algorithm slower convergence, but its stability tends to keep its steady state closer to optimal static ordering, and results in a lower asymptotic cost. This is best if there is little locality in the accesses.

3.3. Count

An extra field of memory is kept with each record, in which a count of accesses to that record is maintained. The records are ordered by decreasing values of this field. This may require linear time to reorder the records after an access if many of the counts are equal, but maintaining a backpointer during the search can minimize this cost.

3.4. 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.

The *move-ahead-k* algorithm was initially proposed by Rivest [1976]. Gonnet, Munro, and Suwanda show that, for $j > k$, *move-ahead-j* converges faster than *move-ahead-k*, but at the penalty of a higher asymptotic cost [GON79].

3.5. k-in-a-row

This might be called a meta-algorithm, since it can be applied to any permutation algorithm α. Algorithm α is applied only if the accessed record has been accessed k times in a row. The purpose here is to slow the convergence of the algorithm by not moving records on a single access, which may be a one-time occurrence. 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 global counter for number of accesses.

Gonnet, Munro and Suwanda propose this scheme and show that, for $k = 2$, the convergence using *move-to-front* is the same as that of *transpose* [GON79].

Kan and Ross use semi-Markov chains to analyze *k-in-a-row* policies [KAN80]. They consider rules that move elements closer to the front of the list only after the elements have been accessed k times in a row. 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 ∞.

3.6. 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 best if 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 sequential traversal at the end of the search, the pointer must occasionally be jumped forward to the record which was just probed. The pointer will be 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 both constant *k* and *k* being a percentage of the distance to the front).

4. 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. The following sections describe these methods of comparison.

4.1. Optimal Algorithm?

A major question being asked is whether an optimal memoryless strategy exists. Rivest defines an ***optimal permutation rule*** to be one with least cost for all probability distributions and any initial ordering of the keys, and conjectures that ***transpose*** is optimal under this definition [RIV76]. Kan and Ross provide evidence to support this, and show that ***transpose*** is optimal among the rules that move a single element some number of places closer to the front of the list [KAN80]. This result holds only under the assumption of no locality. Yao proves that if an optimal algorithm exists, than it must be ***transpose*** [RIV76]. Anderson, Nash and Weber provide a complex counterexample to Rivest's conjecture by showing a permutation algorithm that outperforms ***transpose*** in the average case [AND82].

Therefore, no algorithm can be optimal over all ρ, and the following analyses for different algorithms cannot be expected to demonstrate the superiority of any given algorithm.

4.2. Asymptotic Cost

The asymptotic cost of ***move-to-front*** has been shown by many to be at most twice the asymptotic cost of the optimal static ordering [MCC65, KNU73, HEN76, RIV76, BIT79]. Rivest shows that the asymptotic cost of ***transpose*** is less than or equal to that of ***move-to-front*** for every probability distribution [RIV76]. The result by Gonnet, Munro and Suwanda concerning ***move-ahead-k*** algorithms extends Rivest's work to show that ***transpose*** has lower asymptotic cost than ***move-ahead-k*** for any $k > 1$ [GON79].

Bitner 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 [BIT79].

4.3. Worst Case

The worst case cost of ***move-to-front*** has been shown by many to be no more than twice that of the optimal static ordering [BUR73, KNU73, HEN76, RIV76, BIT79, BEN82, SLE84]. This might be intuitively seen by again considering the examples given earlier. 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 about n, and the cost using the optimal static ordering in the same situation is about $n/2$. This does not show that ***move-to-front*** is ***never*** worse than twice the optimal static ordering, but suggests this might be true.

The relative analyses for the worst case of ***move-to-front*** are by techniques which are not generally applicable, with the exception of the analysis of Sleator and Tarjan [SLE84]. 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.

Bently and Cole additionally point out that while 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 [BEN82].

4.4. Convergence

Bitner shows that, while *transpose* is asymptotically more efficient, *move-to-front* often converges more quickly [BIT79]. Bitner 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*.

Bitner suggests the use of a hybrid that changes from *move-to-front* to *transpose* when the number of requests is between $\Theta(n)$ and $\Theta(n^2)$. He mentions that, in his thesis, 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 observation.

5. OPEN PROBLEMS

Since it has been shown that no single optimal permutation algorithm exists, it becomes necessary to try to characterize under what circumstances different algorithms are preferable. There are several conditions to consider, which give rise to the following open questions concerning algorithms.

5.1. Locality/Convergence

Hendricks lists relaxing the assumption of independence for analysis of permutation algorithms as an interesting open problem [HEN76]. 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-adjusting sequential search. This is unfortunate since taking advantage of locality is one of the main reasons for using these techniques in the first place, and it restricts measures of convergence to very general cases.

5.2. 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 present in a access sequence, development of algorithms that optimize convergence for the given level of locality is an obvious step. A few hybrids were suggested that quickly converge to a steady state and also have a good asymptotic cost such as using ***move-to-front*** initially and then switching to ***transpose*** at some later time. As was pointed out, the time to switch to ***transpose*** in the above example, and similar problems in other algorithms, is currently difficult under the assumption that nothing at all is known about the access sequence. Allowing only knowledge of the approximate degree of locality of the sequence may make it possible to tailor these algorithms well.

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 just a fixed constant.

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 a 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.

References

[AND82] Anderson, E. J., Nash, P., and Weber, R. R. A Counterexample to a Conjecture on Optimal List Ordering. *J. Appl. Prob. 19*, 3 (September, 1982), 730–732.

[BEN82] Bentley, J. L., and Cole, C. Worst-case Analyses of Self-Organizing Sequential Search Heuristics. Appearing in *Proc.* 20th *Allerton Conference on Communication, Control, and Computing*, 1982, pp. 452–461.

[BIT79] Bitner, James R. Heuristics that Dynamically Organize Data Structures. *SIAM J. Comput. 8*, 1 (February, 1979), 82–110.

[BUR73] Burville, P. J. and Kingman, J. F. C. On a Model for Storage and Search. *J. Appl. Prob. 10*, 3 (September, 1973), 697–701.

[DEN72] Denning, Peter J. and Schwartz, Stuart C. Properties of the Working-Set Model. *CACM 15*, 3 (March, 1972), 191–198.

[GON79] Gonnet, Gaston H., Munro, J. Ian, and Suwanda, Hendra Toward Self-Organizing Sequential Search. *Proc.* 20th *IEEE Symp. on Foundations of Computer Science* (October, 1979), 169–174.

[HEN72] Hendricks, W. J. The Stationary Distribution of an Interesting Markov Chain. *J. Appl. Prob. 9*, 1 (March, 1972), 231–233.

[HEN73] Hendricks, W. J. An Extension of a Theorem Concerning an Interesting Markov Chain. *J. Appl. Prob. 10*, 4 (December, 1973), 886–890.

[HEN76] Hendrics, W. J. An account of self-organizing systems. *SIAM J. Comput. 5*, 4 (December, 1976), 715–723.

[HES84] Hester, J. H., and Hirschberg, D. S. Self-Organizing Lists Using Backpointers. Tech. Rept. ICS Dept, University of California, Irvine (1984).

[KAN80] Kan, Y. C. and Ross, S. M. Optimal List Order Under Partial Memory Constraints. *J. Appl. Prob. 17*, 4 (December, 1980), 1004–1015.

[KNU73] Knuth, D. E. A "Self-Organizing" File. Appearing in *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp. 398–399.

[McC65] McCabe, John On Serial Files with Relocatable Records. *Operations Research* (July/August, 1965), 609–618.

[RIV76] Rivest, Ronald On Self-Organizing Sequential Search Heuristics. *CACM 19*, 2 (February, 1976), 63–67.

[SLE84] Sleator, Daniel Dominic, and Tarjan, Robert Endre Amortized Efficiency of List Update Rules. *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing* (April, 1984), 488–492.

[TEN82] Tenenbaum, Aaron and Nemes, Richard M. Two Spectra of Self-Organizing Sequential Search Algorithms. *SIAM J. Comput. 11*, 3 (August, 1982), 557–566.