Programming Techniques and Data Structures

Ellis Horowitz Editor

Amortized Analyses of Self-Organizing Sequential Search Heuristics

JON L. BENTLEY and CATHERINE C. McGEOCH

ABSTRACT: The performance of sequential search can be enhanced by the use of heuristics that move elements closer to the front of the list as they are found. Previous analyses have characterized the performance of such heuristics probabilistically. In this article, we use amortization to analyze the heuristics in a worst-case sense; the relative merit of the heuristics in this analysis is different in the probabilistic analyses. Experiments show that the behavior of the heuristics on real data is more closely described by the amortized analyses than by the probabilistic analyses.

1. INTRODUCTION

The performance of sequential search in an unsorted list can be enhanced by the use of *self-organizing* heuristics that attempt to ensure that frequently accessed keys are near the front of the list.[1] The following three heuristics are representative of a larger class.

- *Transpose.* When the key is found, move it one closer to the front of the list by transposing it with the key immediately in front of it.
- *Move-to-front.* When the key is found, move it to the front of the list (all other keys retain their relative order).
- *Count.* When the key is found, increment its count field (an integer that is initially zero) and move it forward as little as needed to keep the list sorted in decreasing order by count.

Note that the first two heuristics require no memory other than that for representing the lists, and the third heuristic requires an additional count field. Previous work, described in the next section, has shown that under various probabilistic assumptions, these heuristics can significantly reduce the time required by sequential search.

In this article, we will investigate the heuristics in a worst-case sense. Such an analysis is trivial if we consider the worst-case cost of a single search: the entire list must be searched. We will therefore count the worst-case number of comparisons made by a heuristic for any particular *sequence* of requests; because the cost is distributed over a series of requests, this is called an *amortized* analysis. We show that the amortized cost of the move-to-front and count heuristics is at most twice that of the optimal static ordering (defined in the next section) for any sequence of search keys. We also present a counterexample to show that the transpose heuristic can have a very poor amortized performance.

These analyses are of both theoretical and practical interest. Section 3 emphasizes the theoretical tools used in the amortized analysis. Our results provide a simpler proof of the strongest general theorem known for the average performance of the move-to-front heuristic. Furthermore, the analyses use a simple but elegant bookkeeping technique of general interest. The practi-

[1] The term list refers only to its sequential nature; our results apply to lists implemented with arrays or with records and pointers.

This research was supported in part by the Office of Naval Research under contract N00014-76-0370 and in part by the National Science Foundation under an NSF Graduate Fellowship.

cal contribution of this article is not so much prescriptive as descriptive: practitioners have long used the move-to-front heuristic even though theoretical analyses indicated that the transpose heuristic was superior. Our analysis provides a metric under which move-to-front is superior to transpose and thereby explains their actions.

This article is organized in six sections. An overview of previous work can be found in Section 2. The new results are presented in Section 3. Section 4 describes the results of empirical studies, and advice to practitioners is given in Section 5. Conclusions are in Section 6.

2. PREVIOUS WORK

In this section, we will briefly survey previous results concerning self-organizing heuristics. Only the more important results are presented here; for further study, consult the references.

The heuristics (or rules) that we study deal with a list of N keys. A query is answered by performing a linear search for the requested key and then reordering the list according to some search rule. A string of requests forms a *request sequence.* An important kind of request sequence independently chooses the ith key with probability p_i according to the probability distribution $P = (p_1, p_2, \ldots, p_N)$. The *cost* of a search rule for such a distribution is defined as the asymptotic expected search cost for a single key (measured as the number of comparisons made) when the list is being reordered according to the rule; we will denote the cost for rule R by $A_R(P)$. The *optimum static ordering* arranges the keys in decreasing order of request probabilities and never reorders them. Although this is not necessarily optimal over all rules (because it is static rather than dynamic), it has been used as a basis for comparing the performance of the heuristics; its cost will be denoted by $A_O(P)$. The heuristics have been studied under this asymptotic model since 1965; we sketch the significant results below.

The asymptotic expected search cost $A_M(P)$ of the move-to-front rule for probability distribution P has been derived by McCabe [17], Burville and Kingman [7], Knuth [15], Hendricks [13], Rivest [19], and Bitner [5]. The formula can be used to show that $A_M(P)$ is at most twice the cost of the optimal static ordering, $A_O(P)$. Rivest conjectured that this bound is not tight; Gonnet, Munro, and Suwanda [8] presented a distribution that gives a ratio of $\pi/2$, which is the highest yet found. The asymptotic expected cost for transpose, $A_T(P)$, was shown by Rivest to be less than or equal to $A_M(P)$; the bound is strict for all P except where all the nonzero probabilities are equal or when $N = 2$.

The count heuristic uses a frequency count f_i of requests for the ith key. Because of the additional storage required, count has been considered in a different class from move-to-front and transpose, and has received less attention. By the law of large numbers, if $p_i > p_j$, then the frequency f_i may be less than f_j for only a finite number of requests. The search cost under count, therefore, asymptotically approaches that of the optimal static ordering. Bitner [4] showed that if P is not known beforehand, then count produces the ordering with the lowest expected cost for each request.

A recent line of study has centered on the discovery of optimal heuristics over classes of probability distributions and heuristics. Rivest considered the class of *permutation rules* that apply some fixed permutation τ_j to the list after finding a key in position j. He defined the *optimal permutation rule* to be the one with least cost for all probability distributions and any initial list order, and he conjectured that transpose is optimal over all permutation rules. Yao (reported in Bitner [4]) and Bitner [6] have given distributions where transpose indeed is optimal over all such rules, but Anderson, Nash, and Weber [2] presented a counterexample to the conjecture in the form of a rule with less cost than transpose for a specific distribution. For work on other distributions and on modifications of the heuristics, see Gonnet, Munro and Suwanda [8], Hendricks [12], Bitner [4, 6], Kan and Ross [14], and Tenenbaum and Nemes [24]. One natural and interesting distribution is Zipf's Law: Knuth [15] showed that the cost of move-to-front under this distribution is bounded by 2 ln 2 (about 1.386) times the cost of optimal static ordering.

Measurements other than the asymptotic cost have been considered. Bitner [4, 5] showed that while transpose is asymptotically more efficient, move-to-front converges more quickly. Move-to-front may therefore be preferred when the number of requests is not large. Bitner proposed a hybrid rule that changes from move-to-front to transpose when the number of requests falls in a certain range: For Zipf's Law he suggests a change point in the range of $\Theta(N)$ to $\Theta(N^2)$ requests. Bitner also discussed the *overwork* (the area between the cost curve and its asymptote) for the two rules, and presented distributions for which move-to-front performs much better than transpose under this measure. Under Zipf's Law, for example, the overwork is $O(N^2)$ for move-to-front and is $\Omega(N^3)$ for transpose.

Rivest [19] introduced the family of *move-ahead-k* heuristics, where a requested key is moved ahead k positions ($k = 1$ corresponds to transpose and $k = N - 1$ corresponds to move-to-front). He simulated the asymptotic behavior of these heuristics for k ranging from 1 to 6 and N from 3 to 12 on request sequences of length 5000 constructed by Zipf's Law. Bitner [4] conjectured (and Gonnet, Munro, and Suwanda [8] later proved) that for any two heuristics in this range, one will converge faster and the other will have lower asymptotic cost. Tenenbaum [23] performed similar tests for k ranging from 1 to 7 and N from 3 to 230 on sequences of length 12,000. He empirically determined the best choice of k for a given N on sequences of this length.

Just as modifications of move-to-front and transpose have been proposed, various schemes have been considered to reduce the space needed to maintain the frequency counts for the count heuristic. Bitner [4, 5]

suggested that it is better to maintain the differences between frequencies of adjacent keys rather than their actual counts and also proposed a *limited-difference* rule, where the counts are left unchanged after some upper limit is reached. Other modifications have been suggested for use in combination with move-to-front or transpose. Gonnet, Munro and Suwanda [8], and Kan and Ross [14] have examined *k-in-a-row* heuristics, where a key is moved only after it has been requested k times in a row, and Bitner [4] has analyzed rules of the form *wait-c-and-move*. Lam, Sui, and Yu [16] presented a scheme that was shown to be optimal over all heuristics that use frequency information.

The amortized analysis of this article has precedents in other areas. For example, Aho, Hopcroft, and Ullman [1 (Sections 4.6 and 4.7)] use this model to study union-find algorithms: Although any particular operation may be costly, the amortized cost for a sequence of operations is small. Sleator and Tarjan [21] use amortization to study heuristics for self-organizing binary search trees.

3. AMORTIZED ANALYSES

The primary results of the previous section can be summarized as follows: For any probability distribution P,

$$A_M(P) \leq 2 \cdot A_O(P),$$

$$A_T(P) \leq A_M(P), \quad \text{and}$$

$$A_C(P) = A_O(P).$$

All of these results deal with the asymptotic expected cost of a single search when the queries are from a distribution P. In this section we will consider the amortized cost of performing all searches in a given sequence S of queries. When the list is being reordered by rule R, we will denote the total number of comparisons required for the sequence S by $C_R(S)$ (signifying the concrete cost as opposed to the asymptotic cost). The next two subsections will show that for any sequence S,

$$C_M(S) \leq 2 \cdot C_O(S) \quad \text{and} \quad C_C(S) \leq 2 \cdot C_O(S).$$

In Section 3.3 we will show that such a result does not hold for the transpose heuristic by exhibiting a particular sequence with very poor performance under that rule.

Before describing the results we must define our cost functions precisely. For the move-to-front, count, and transpose rules we define $C_R(S)$ (the cost of rule R applied to request sequence S) by considering the effect of S on an initially empty search list. For each element s of S we search the current list of size m at a cost of i comparisons if s is in position i, or m comparison if s is not present (in which case we insert s at the end of the list). In either case we reorder the list by rule R. The cost $C_O(S)$ of the optimal static ordering is fundamentally different: rather than starting with an initially empty list, each search uses the (unchanging) list in which the keys are arranged by decreasing frequency of their appearance in S. This assumes that all keys are known in advance and implies that each search will be successful.

3.1 Move-to-Front Heuristic

In this section, we will show that for any sequence of requests the number of comparisons made by the move-to-front heuristic is never more than twice that under optimal static ordering. To do this, we will reduce the problem to the case in which the request list contains just two distinct keys, analyze that simple case, and finally combine several facts to complete the proof.

Two types of comparisons are made for a given request sequence: *intraword* comparisons (successfully) compare equal keys, and *interword* comparisons (unsuccessfully) compare unequal keys. For any sequence, the number of intraword comparisons is invariant under all heuristics. For the move-to-front heuristic, the total number of interword comparisons is the sum over all pairs of keys of the number of interword comparisons made between each pair. Furthermore, for any sequence S and all pairs of keys A and B, the number of interword comparisons of A and B is exactly the number made for the subsequence of S consisting solely of A's and B's. We call this the *pairwise independence property* of the move-to-front heuristic: the number of (A, B) comparisons is dependent only on the relative ordering of the A's and B's in the request sequence and is independent of other keys. The proof that move-to-front has this property is easy: requesting A will cause an (A, B) interword comparison if B is in front of A in the search list, which is true if and only if B was requested more recently than A. The other keys in the request sequence do not affect this relationship.

We will now demonstrate the following fact.

FACT 1.
The number of interword comparisons made by the move-to-front heuristic on a sequence S of A's and B's is at most twice the number of interword comparisons made by the optimal static ordering applied to S.

To prove this fact we assume that S consists of m A's and n B's where (without loss of generality), $m \leq n$. Under the optimal static ordering a total of m interword comparisons will be made (because the search list is always in the order $B\ A$, and so only requests for A will cause an interword comparison). Under the move-to-front rule, an interword comparison will be made whenever the request sequence changes from A to B or from B to A. The total number of such changes possible is twice the number of A's (because each change involves an A, and each A can be involved in at most two changes). Therefore, the total number of comparisons made by move-to-front is at most $2m$. Fact 1 follows immediately.

We are now ready to prove the key fact of this subsection.

FACT 2.
For any sequence S, $C_M(S) \leq 2 \cdot C_O(S)$.

We will prove this by simple algebra on the relations

$$C_M(S) = \text{Intra}(S) + \text{Inter}_M(S) \quad \text{and}$$

$$C_O(S) = \text{Intra}(S) + \text{Inter}_O(S),$$

where Intra and Inter_R refer to the total number of comparisons of each type made by rule R. By Fact 1 we know that each pair of keys satisfies the factor of 2 inequality; by the pairwise independence property we can sum over all pairs to get

$$\text{Inter}_M(S) \leq 2 \cdot \text{Inter}_O(S),$$

for any sequence S. Combining this inequality with the above definition gives Fact 2.

The factor of 2 in Fact 2 cannot be tightened. The request sequence

$$(A\ B\ C\ D)^m$$

has $C_O(S) \sim 2.5m$ but $C_m(S) \sim 4m$. Increasing from 4 to k keys gives a ratio of $2k/(k+1)$, which approaches 2 as k increases.

Fact 2 allows a simple proof[2] of the known result that $A_M(P) \leq 2 \cdot A_O(P)$ for any distribution P. Let $C_{AOP}(S)$ denote the cost of applying the (asymptotically) optimal ordering for distribution P to the sequence S. Then,

$$A_M(P) = \lim_{m \to \infty} \Sigma_{S \text{ s.t.} |S|=m} Q_P(S) \cdot C_M(S),$$

and

$$A_O(P) = \lim_{m \to \infty} \Sigma_{S \text{ s.t.} |S|=m} Q_P(S) \cdot C_{AOP}(S).$$

where $Q_P(S)$ is the probability that sequence S will appear under probability distribution P. By the definition of $C_O(S)$ we know that

$$C_O(S) \leq C_{AOP}(S).$$

Combining this with Fact 2 yields

$$C_M(S) \leq 2 \cdot C_{AOP}(S).$$

Substituting the inequality into the above summations term by term completes the proof.

3.2 Count Heuristic

In this section, we will show that the cost of the count heuristic on any particular sequence is at most twice the cost of the optimal static ordering. The argument is essentially the same as for move-to-front, so we will proceed at a faster pace.

First, we establish that the count heuristic has the pairwise independence property: for any sequence S, the number of interword comparisons of A and B is exactly the number made for the subsequence of S consisting solely of A's and B's. This is easily proved: the relative position of A and B in the search list depends entirely on the value of their counts (or in the case of equal counts, on which had the greater count most recently), and the counts are not affected by other keys. As with the move-to-front heuristic, this pairwise independence allows us to focus on two-element sequences. We therefore prove the following fact.

FACT 3.
The total number of interword comparisons made by the count heuristic on a sequence S of A's and B's is at most twice the number of interword comparisons made by the optimal static ordering applied to S.

To prove this fact, we again assume that S consists solely of m A's and n B's, with $m \leq n$. An interword comparison is made every time the key in the rear of the search list is requested; at that time, its count field is incremented. The count field of A can be incremented while it is in the rear at most m times (because it is requested m times). Furthermore, the count field of B can be incremented while it is in the rear at most m times (because after its count field is greater than m it can no longer lie in the rear). The number of interword comparisons is therefore at most m (requests for A's) plus m (for B's), or $2m$. Fact 3 follows immediately.

The key fact of this subsection follows from the same kind of reasoning used to establish Fact 2.

FACT 4.
For any sequence S, $C_C(S) \leq 2 \cdot C_O(S)$.

Again, the proof involves summing over the factor of 2 inequality. By an example similar to that in the previous section, the factor of 2 cannot be tightened.

3.3 Transpose Heuristic

In this section we will demonstrate that the ratio of the amortized performance of transpose to that of the optimal static ordering cannot be bounded by any constant. This is easily observed if we consider the request sequence

$$A\ B\ C\ D\ E(E\ D)^k$$

After the first five elements are stored by transpose, the sequence of $(E\ D)$ request pairs will cause those two elements to swap position at the back of the list and neither will advance. The average cost of a search in this sequence will therefore approach 5, whereas under the optimal static ordering 1.5 comparisons would suf-

[2] Knuth [14, Exercise 6.1-11] rates the problem as M30, implying that it is mathematically oriented and may require over two hours work to solve.

TABLE I. Average Search Costs

	Distinct words	Total words	Zipf's law	Optimal static ordering	Move-to front	Count	Transpose
Pascal files							
P1	100	480	18.28	27.52	24.49	33.16	40.43
P2	107	431	19.36	26.23	25.62	31.50	38.92
P3	117	1,176	20.90	18.04	18.21	20.63	30.53
P4	181	1,456	30.32	30.78	31.40	35.71	47.41
Text files							
T1	471	1,888	68.95	93.03	104.46	111.21	147.41
T2	498	1,515	72.36	112.86	119.31	135.63	160.69
T3	564	3,296	80.53	96.29	98.90	112.41	155.17
T4	999	5,443	132.48	149.34	168.79	175.42	258.20
T5	1,147	7,482	149.47	143.72	174.50	166.10	204.74
T6	1,590	7,654	199.02	232.53	280.83	267.64	349.94

fice. For large k this example gives

$$C_T(S) > 3.33C_O(S).$$

The constant 3.33 can be increased to $\sim 2k/3$ by increasing the length of the "filler" sequence preceding the "active" pair to $k - 2$. Note that this counterexample exploits the fact that the transpose heuristic does not have the pairwise independence property: the relative order of any two keys depends not only on the request sequence but also on whether the keys are adjacent in the search list.

4. EMPIRICAL RESULTS

The theoretical analyses in Sections 2 and 3 are by no means unanimous in their evaluation of the heuristics. To gain further insight into their behavior, we used each heuristic to perform "word counts" on several files. The words in each file served as a request sequence. As each word (defined to be an alphanumeric string delimited by spaces or punctuation marks) was requested, a linear search of the key list was performed, the count field for the key was incremented, and the list was reordered according to the appropriate rule. They key lists were initially empty; at the first request for a word, the list was searched to the end to determine its absence and the reordering occurred as though the element had been found in the (new) last position. Although this particular problem clearly suggests the count heuristic (since the frequencies are stored anyway), this input is one natural indicator of the behavior of the heuristics.

The average search cost required by each heuristic for each file is reported in Table I (the cost is defined as the total number of interword comparisons made for the file divided by the number of words in the file). The input files included four Pascal files and six text files.[3] Under a uniform distribution of requests the average search cost is $D/2$, where D is the number of distinct words in the file. We might expect better performance for these files, however, because request frequencies in many natural contexts obey Zipf's Law (in which the average search cost is roughly $D/\ln D$: See Knuth [15 (Section 6)]. The third column in Table I gives the cost of the optimal static ordering for requests from Zipf's distribution with the same number of total and distinct words; comparing that column to the cost of the optimal static ordering shows that the data is closer to Zipf's distribution than to a uniform distribution.

Figure 1 shows the percent of the list searched (on average) by the three heuristics and the optimal static ordering for each of the ten files. On file P4, for example, move-to-front made an average of 31.4 comparisons in a list of 181 distinct words, which corresponds to searching about 17 percent of the list; the dashed line connecting optimal static ordering with transpose associates the measurements of a single file.

Figure 1 illustrates several points.

- The heuristics are effective: in each file, all heuristics search less than half the list (which is the expected amount searched in a randomly ordered list).
- Move-to-front sometimes does better than the optimal static ordering (files P1 and P2). This is possible only because move-to-front is not a member of the (static) class in which the optimal static ordering is best.[4] "Words" in program text exhibit strong temporal locality, which is handled very well by move-to-front (consider, for instance the several occurrences of "**integer**" at the start of a program, assignments of the form "**x[i] = x[i] + 1**", and "**end; end; end**" sequences).

[3] The text files include the Constitution of the United States (T6), the script to *The Rocky Horror Picture Show* (T5), a preliminary version of this article (T4), excerpts from an on-line documentation system, and text files augmented with commands to the Scribe document production system.

[4] Motivated by this observation, Sleator and Tarjan [22] compared the performance of several heuristics to the optimal *dynamic* algorithm over a large class of dynamic heuristics. They showed that move-to-front is within a factor of 2 of optimal, even in this larger class.

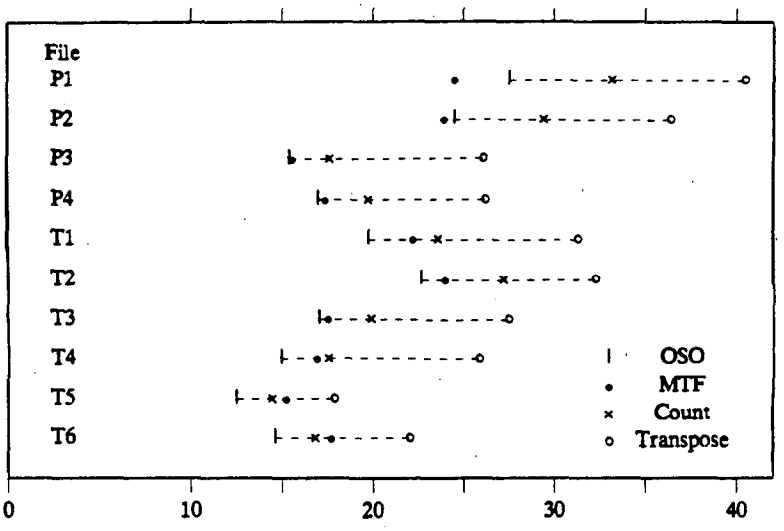

FIGURE 1. Percent of List Searched

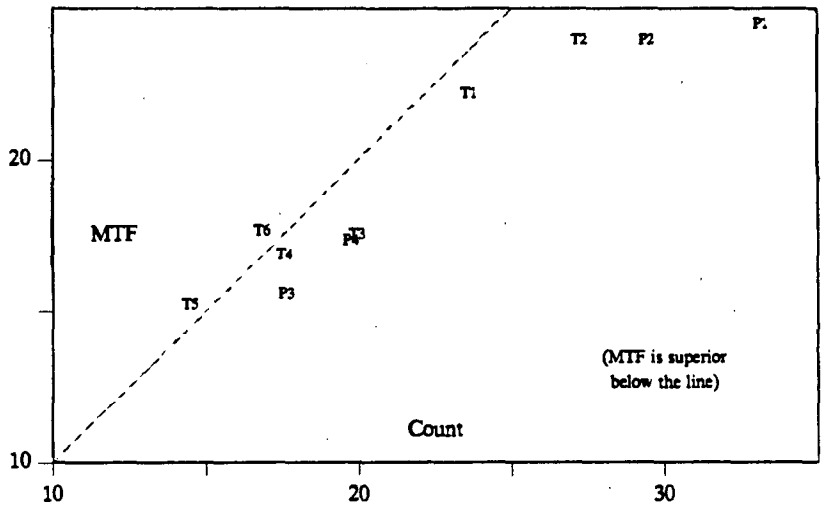

FIGURE 2. Percent of List Searched by MTF and Count

- In most cases move-to-front and count are quite close to one another (and typically to the optimal static ordering), while transpose lags far behind. Figure 2 plots the performance of move-to-front against the performance of count for each file; although move-to-front appears to be marginally better for this small sample, more data will have to be gathered before a definitive statement can be made.

The empirical results indicate that neither the amortized nor the probabilistic analyses by themselves completely describe the behavior of the heuristics under natural conditions: transpose is clearly not the best heuristic for this application, yet it never performed as badly as our results showed it might. The distribution and size of the request sequence seem less significant than the ordering of the requests. Empirical results for the Pascal files would be less dramatic if Pascal reserved words were treated differently from identifiers, as might happen within a compiler. If sequential search were used by an interpreter for identifier lookup at run time, however, the presence of *dynamic locality* (for example, in requests for loop variables) would argue strongly for the use of move-to-front.[5] This phenomenon is not restricted to program text; for example, the

[5] The widely-used Microsoft BASIC interpreter uses the heuristic of storing symbols in its run-time (linear) symbol table in the order in which they were first seen. One of the authors (Bentley) once reduced the run time of a BASIC program under such an interpreter from 14 hours to 7 hours simply by referring to each of five "hot" variables once in a dummy statement at the front of the program. The use of the move-to-front heuristic in such an interpreter would probably substantially decrease the run time of many BASIC programs.

word "president" appears with high locality in the U.S. Constitution. In most written prose, locality of subject (and therefore of certain words) determines paragraph construction. Move-to-front is clearly able to take advantage of this characteristic. Indeed, such a phenomenon as "pairwise-locality" (the appearance of word pairs or two-word phrases such as "vice-president" or *A*[1]) might hamper the performance of transpose; if two such words have the misfortune to be adjacent in the key list, they will contend with each other rather than drifting to the front as they should.

5. ADVICE TO PRACTITIONERS

The previous sections have evaluated the heuristics from various viewpoints, using theoretical tools as well as observed results for several data sets. We now consider the heuristics from a different perspective: how should they be used by practicing programmers?

The purpose of the heuristics is to enhance the performance of linear search. This raises the most important point of this section: if a programmer faces an efficiency problem in a search procedure, then linear search is probably not the method of choice. Knuth [15 (Chapter 6)] describes a number of other search methods that are usually significantly more efficient. There are, however, contexts in which self-organizing linear search may be appropriate.

- When N is small (say, at most several dozen), the greater constant factors in the run times of other strategies may make linear search competitive. 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 be too space-expensive to use.
- If the performance of linear search is almost (but not quite) good enough, a self-organizing heuristic may make it effective for the application at hand without adding more than a few lines of code.

McCreight [18] found himself in the last situation when improving the performance of a VLSI circuit simulator that had two primary phases: the first phase read the description of the circuit and the second phase simulated the circuit. On typical runs the first phase took five minutes while the second phase took several hours. Although the five minutes of the first phase was not crucial, it was irritating for users to have to wait that long to see the simulation begin (especially when they knew that most of the time was going to sequential searches in the simulator's symbol table). Following McCreight's suggestion, the implementer of the program augmented the straightforward sequential search with the move-to-front heuristic. Those additional half-dozen lines of code decreased the run time of the first phase from five minutes to half a minute (most of which was *not* going to symbol table routines).

Knowing when to use self-organizing sequential search heuristics still leaves the implementer with the decision of which one to choose for a given application. Some authors have interpreted the results in Section 2 in a way we feel is unwarranted; for instance, Gotlieb and Gotlieb [10 (p. 118)] assert in their excellent data structures text that "[move-to-front] is not the best [strategy] for a self-organizing list. It is better to promote the referenced entry only one place by transposing it with its predecessor." The following observations may be relevant to most situations in which self-organizing schemes are applicable.

- *Move-to-front.* The linked list implementation of this heuristic is the method of choice for most applications. The heuristic 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. Unfortunately, move-to-front is expensive if the sequence is implemented as an array.
- *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.

In the above discussion, we have intentionally kept several potentially quantifiable measures vague, such as lines of code and run times. Rather, we appeal to an intuition that asymptotically efficient algorithms (such as balanced binary trees) tend to require more code and to have larger constant factors than simpler structures (such as sequential search). We have avoided hard data because it is extremely sensitive to coding style and to details of the compiler and machine architecture. Readers who insist on such detail should consult Knuth [15 (Chapter 6)], but we warn that data on his MIX implementations may be misleading for other computing environments.

6. CONCLUSIONS

The conclusions of this article are clear: when a self-organizing sequential search is appropriate in an application, the count and (especially) the move-to-front heuristics should be considered for implementation. Although previous probabilistic analyses showed that transpose is superior to move-to-front under some measures, both our amortized analyses and our empirical results show contexts in which the opposite is true.

The results in this article could be extended in a number of ways. An implementer of these algorithms may wish to consider measurements other than num-

ber of comparisons, such as number of moves or total distance moved. To predict the behavior of the heuristics on input such as that described in the previous section more accurately, it would be helpful to have theoretical tools for decribing the locality in the input data: Bellows [3] has studied models for describing locality and the performance of the heuristics under these models. The amortized analysis of other algorithms previously analyzed only for their expected performance is an interesting area. The proof techniques that we presented may be useful in studying other self-modifying structures in an amortized sense.

Acknowledgments. The helpful comments of Jim Saxe, Bruce Wiede, and an anonymous referee are gratefully acknowledged.

REFERENCES

Note: References 9, 11, 20, and 25 are not mentioned in text.

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. Anderson, E.J., Nash, P., and Weber, R.R. A counterexample to a conjecture on optimal list ordering. *J. Appl. Prob. 19*, 3 (1982), 730–732.
3. Bellows, M.U. Performance of self-organizing sequential search heuristics under stochastic reference models. Ph.D. dissertation, Dept. of Statistics, Carnegie-Mellon Univ., Pittsburgh, Pa., 1983.
4. Bitner, J.R. Heuristics that dynamically alter data structures to reduce their access time. Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, 1976.
5. Bitner, J.R. Heuristics that dynamically organize data structures. *SIAM J. Comput. 8*, 1 (Feb. 1979), 82–110.
6. Bitner, J.R. Two results on self-organizing data structures. Tech. Rep. TR-189, Dept. of Computer Science, Univ. of Texas at Austin, Austin, Tex., Jan. 1982.
7. Burville, P.J., and Kingman, J.F.C. On a model for storage and search. *J. Appl. Prob. 10*, 3 (Sept. 1973), 697–701.
8. Gonnet, G., Munro, J.I., and Suwanda, H. Toward self-organizing sequential search heuristics. In *Proceedings of 20th IEEE Symposium Foundations Computer Science*, (San Juan, Puerto Rico, 1979), 169–174.
9. Gonnet, G., Munro, J.I., and Suwanda, H. Exegisis of self-organizing linear search. *SIAM J. Comput. 10*, 3 (Aug. 1981), 613–637.
10. Gotlieb, C.C., and Gotlieb, L.R. *Data Types and Structures.* Prentice Hall, Englewood Cliffs, N.J. 1978, p. 118.
11. Hendricks, W.J. The stationary distribution of an interesting Markov chain. *J. Appl. Prob. 9*, 1 (Mar. 1972), 231–233.
12. Hendricks, W.J. An extension of a theorem concerning an interesting Markov chain. *J. Appl. Prob. 10*, 4 (Dec. 1973), 886–890.
13. Hendricks, W.J. An account of self-organizing systems. *SIAM J. Comput. 5*, 4 (Dec. 1976), 715–723.
14. Kan, Y.C., and Ross, S.M. Optimal list order under partial memory constraints. *J. Appl. Prob. 17*, 4 (Dec. 1980), 1004–1015.
15. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
16. Lam, K., Sui, M.K., and Yu, C.T. A generalized counter scheme. *Theoretical Comput. Sci. 16*, 3 (Dec. 1981), 271–278.
17. McCabe, J. On serial files with relocatable records. *Oper. Res. 13*, (July 1965), 609–618.
18. McCreight, E. Personal communication, Xerox Palo Alto Research Center, Palo Alto, CA, Feb. 1983.
19. Rivest, R. On self-organizing sequential search heuristics. *Commun. ACM 19*, 2 (Feb. 1976), 63–67.
20. Schay, G., Jr., and Dauer, F.W. A probabilistic model of a self-organizing file system. *SIAM J. Appl. Math. 15*, 4 (Feb. 1967), 874–888.
21. Sleator, D., and Tarjan, R. Self-adjusting binary search trees. In *Proceedings of 15th Symposium on Theory of Computing*, (Boston, Mass., 1983), 235–245.
22. Sleator, D., and Tarjan, R. Amortized efficiency of list update and paging rules. *Commun. ACM 28*, 2 (Feb. 1985), 202–208.
23. Tenenbaum, A. Simulations of dynamic sequential search algorithms. *Commun. ACM 21*, 9 (Sept. 1978), 790–791.
24. Tenenbaum, A., and Nemes, R.M. Two spectra of self-organizing sequential search algorithms. *SIAM J. Comput. 11*, 3 (Aug. 1982), 557–566.
25. Veinott, A.F. Optimal policy in a dynamic, single product, nonstationary inventory mode with several demand classes. *Oper. Res. 13*, (1965), 761–778.

CR Categories and Subject Descriptors: E.1 **[Data]**: Data Structures—*lists*; *tables*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*sorting and searching*
General Terms: Algorithms, Theory
Additional Key Words and Phrases: self-organizing, move-to-front.

Received 4/83; revised 7/84; accepted 12/84

Authors' Present Addresses: Jon L. Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974. Catherine C. McGeoch. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.