Self-Organizing Search Lists Using Probabilistic Back-Pointers

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

ABSTRACT: A class of algorithms is presented for maintaining self-organizing sequential search lists, where the only permutation applied is to move the accessed record of each search some distance towards the front of the list. During searches, these algorithms retain a back-pointer to a previously probed record in order to determine the destination of the accessed record's eventual move. The back-pointer does not traverse the list, but rather it is advanced occasionally to point to the record just probed by the search algorithm. This avoids the cost of a second traversal through a significant portion of the list, which may be a significant savings when each record access may require a new page to be brought into primary memory. Probabilistic functions for deciding when to advance the pointer are presented and analyzed. These functions demonstrate average case complexities of measures such as asymptotic cost and convergence similar to some of the more common list update algorithms in the literature. In cases where the accessed record is moved forward a distance proportional to the distance to the front of the list, the use of these functions may save up to 50% of the time required for permuting the list.

1. 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 expected search time. One common application in which this situation arises is a list (or lists) of identifiers maintained by a compiler or interpreter. The list cannot be initially ordered since frequencies are unknown, but since most programs tend to access some identifiers much more often than others, the more frequently accessed identifiers should be nearer the front of the search list containing them. In general, sequential searches may be useful any time the number of elements is small, the space is severely limited, or the performance of sequential search is acceptable and there is a desire to keep the code simple [Ben85]. In any of these cases, the extra few lines of code required to make the list self-organizing can improve the expected search time significantly. Interesting questions are what algorithms can be used for this permutation, and how do they perform relative to each other in terms of expected search time.

We propose a new class of algorithms, called *JUMP*, which is based on retaining a back pointer in the list during searches to be used for determining what reordering shall take place. We show that specific members of this class involving probabilistic functions can be made to demonstrate the same permutations, in the average case, as some of the more commonly proposed algorithms in the literature, but the permutations themselves can often be accomplished more efficiently.

2. BACKGROUND

The *accessed record* is the record we are looking for, and the *probed record* is the record we are currently looking at during the search.

For a given initial list configuration and search sequence ρ, the *cost* of a permutation algorithm α is the average number of comparisons made per record searched over all searches in ρ. To determine expected cost in a given case, it is necessary to make some assumptions about the contents of ρ. The most common assumption is that there is a fixed probability of access for each record, and that accesses to records are independent of each other. Under this assumption the *asymptotic cost* of the algorithm is the limit of the average cost per access as $|\rho|$ increases.

It is usually assumed that the initial list is unordered. As α permutes the list after each access, the expected search time for the next record should decrease until a *steady state* is reached where many further permutations by α are not expected to increase or decrease the expected search time significantly. Note that this steady state is not any single ordering of the list, or even a set of orderings, but rather a condition where further changes are not expected to have a significant effect on the average search time. When we say an algorithm *converges* on its steady state, we mean that the effect of further permutations on the average search time decreases as permutations are performed, and the effect should approach zero as the number of permutations approaches infinity. Rate of convergence and asymptotic cost are often tradeoffs in permutation algorithms.

2.1. Algorithms

The following commonly analyzed permutation algorithms will be referenced relative to the results of this paper. The reader is directed to [Bit79, Gon81, Hes85] for more complete lists and analyses.

Move-to-front moves the accessed record, when found, to the front of the list if it is not already there. This algorithm tends to converge quickly to a steady state, but the price of this convergence speed is a large asymptotic cost since a record accessed only once moves all the way to the front, which increases the costs of accesses to many other records. When the search sequence has a large degree of *locality* (the searches to some records are not evenly distributed throughout the sequence), *move-to-front* is quick to adjust to the changing probabilities of access for local sections of the sequence.

Transpose moves the accessed record, if not at the front of the list, up one position by changing places with the record just ahead of it. Thus a record only approaches the front of the list if it is accessed frequently. The slower record movement gives *transpose* slower convergence, but the resultant stability tends to keep the expected cost of its steady state lower than that of *move-to-front* for search sequences having a small degree of locality.

Move-ahead-k, a compromise between the relative extremes of *move-to-front* and *transpose,* moves the record forward k positions where k can be a constant or a function of n and/or the location of the accessed record. Adjusting the value of k allows the system to be tuned to obtain a good tradeoff of asymptotic cost vs. convergence. No formal techniques currently exist for this tuning, but a decent improvement should be possible by empirically observing the effect of small adjustments.

Due to the tradeoff between convergence rate and asymptotic cost, Bitner [Bit79] proposed hybrid algorithms that initially use an algorithm with fast convergence (such as *move-to-front*) until that algorithm approaches its steady state and then switching to an algorithm with a better asymptotic cost (such as *transpose*) for further searches.

2.2. Data Structure

A standard assumption is that the list is linked. This allows moving a single record in constant time by relinking, once the record is found and the destination of the move has been determined. *Move-to-front* and *transpose* determine where to move the record in constant time, since a pointer to the front of the list is available, and the last record probed can easily be remembered. However, algorithms that move the record any non-constant distance forward may spend time proportional to the distance of the move searching for the destination of the move.

We also assume that all records that will be searched for are in the list, and we can therefore ignore failed searches. If this assumption is false, we merely add detection of the end of the list to the search algorithm and append the record to the end of the list. In this case, whatever permutation is normally called for would be applied as usual.

3. THE JUMP FUNCTION

We wish to find record x. Our algorithm initially sets a back-pointer b to the first record in the list, and then begins searching. Each time a record p is probed and is not x, a boolean function β is evaluated. If β is $true$, b is advanced to p. The search then continues. When x is found, x is moved just ahead of b, unless b is x (which is true if and only if x is at the front of the list). Note that β can cause a record to move forward any distance between 1 and the full distance to the front of the list. The evaluation of β after a failed probe at the first record in the list will have no effect because the initial value of b is already pointing to this record. Otherwise, the back-pointer always points at least 1 record behind the probed record. The following simplified search algorithm illustrates the use of the function:

```
**function** search( *searchkey*, *listhead* )
**begin**
    $b \leftarrow listhead$
    $p \leftarrow listhead$
    **while** $KEY[p] \neq searchkey$ **do begin**
        **if** $\beta$ **then** $b \leftarrow p$
        $p \leftarrow NEXT[p]$
    **end**
    remove $p$ from list
    re-insert $p$ in front of $b$
    **return** $p$
**end**
```

The main advantage of this algorithm is that it allows moving a record forward a distance other than one place or all the way to the front of the list without requiring a second search through the list looking for the place to move to. This provides the ability to tune the system without paying the cost of (up to) doubling the traversal time that was required by *move-ahead-k*. If the keys are extremely large or (more likely) there are a large number of small records (as with object lists in LISP, for example), then each access will have a good chance of requiring access to slower secondary memory. The dynamic (linked) nature of the list prevents simple attempts to keep records that are close to each other (in terms of their logical location in the list) on the same physical page of memory as searches and permutations progress.

More efficient search structures are often advisable in these cases, but a simple linear list may be advisable when space is at a premium, or when the efficiency of linear search good enough to desire avoiding complicated additions to the code. Bentley and McGeoch [Ben85] provide a list of other situations in which self-organizing sequential search is useful. *JUMP* can provide for a significant constant speedup in any of these cases without adding much complexity to the code.

β may be any function desired. Thus we have defined a class of algorithms rather than a single one. β may take any parameters desired, such as the location of the probed record, the location of the record pointed to by the back-pointer, the number of accesses previously performed, the length of the list, etc. These potential parameters, however, require space proportional to the log of the number of records or the log of the number of accesses.

We define $JUMP(p, b)$ as a class of β functions that take as parameters the locations of the current record being probed and the current back-pointer. We will give analyses of the use of various $JUMP$ functions.

Note that *move-to-front* can be implemented by having $JUMP$ always evaluate to *false,* and *transpose* can be implemented by having $JUMP$ always evaluate to *true.* By using a non-constant $JUMP$ function, we are able to move x forward by various distances without the need of additional searching to find where to move x. Since $JUMP$ is calculated once for every record probed, the total time spent is of the same order as the time needed to perform a linear search to find where to move x, but calls to a simple $JUMP$ function may have a trivial cost when compared with accesses to secondary memory.

Although b and p are pointers to records, we will occasionally refer to them as integer values corresponding to the logical distance from the front of the list to the record to which they point. Thus, if we say that a record at p is expected to move forward $.5p$, we mean that the record pointed to by the search probe will move halfway to the front of the list from its current location.

3.1. Fixed Jumps

This set of jump functions is based on the idea that the locations at which the backpointer is advanced are predetermined, or fixed. An unfortunate side-effect of these fixed ranges is a difficulty with predicting average distances which records will move once they begin to be somewhat ordered, as this biases the probability of the record being at various locations within a range. This problem will be solved by the probabilistic functions of the next section; the functions of this section are given primarily to demonstrate the difficulties of analyzing simple non-probabilistic jump functions.

3.1.1. Constant Moves

Let $JUMP(p, b) = (p \geq b + 2c)$ for any fixed integer c. Assuming records are in any location with equal probability, advancing the backpointer every $2c$ steps will clearly result in records moving an average of about c logical positions forward in the list.

The assumption that records are equally likely to be in any of the relative locations between jump points may be valid initially, when the list is assumed to be unordered, but the effects of $JUMP$ over many calls will, on the average, cause the elements with higher probabilities to be located closer to the front of the search list. This means that it is not clear what the average p as a function of b will be, since the records which are closer to the front of the list will be more likely to be found than records which are further from the front. Even knowing the average value of p may not be sufficient, since the distribution of weights may affect the average move distance.

It appears that the effect of this could be predicted only by making further assumptions about the values of the probabilities; however, we can predict that the average moves would be less than those predicted by the formula derived above, since the heavier weights would be nearer the backpointer. This might be looked upon as a desirable attribute, since we would like quick convergence when the list is unordered, with a better asymptotic cost as the list becomes more ordered. It would demonstrate a behavior similar to the hybrid algorithms proposed by Bitner [Bit79] for similar results. Proving this and determining the magnitude of the decreasing move, if any, is an open question we choose not to pursue due to the superior analysis permitted by probabilistic backpointers.

3.1.2. Fractional Moves

Let $JUMP(p,b) = (p/b \geq c)$ for any fixed $c > 1$. Intuitively, the backpointer will be advanced every time the probe reaches a power of c, but this turns out to be the case only when c is integer. For non-integer values of c, the points at which a jump takes place are still fixed, but the cumulative effect of the cutoffs in the boolean relation cause the true jump points to diverge from powers of c. For any given backpointer location b (b is any of a predetermined set) the average fraction of p that a record will move, in terms of c (assuming an equal probability that p lies anywhere in the range from $b+1$ to $\lceil bc \rceil$), is

$$\frac{1}{\lceil cb \rceil - b} \sum_{p=b+1}^{\lceil bc \rceil} \frac{p-b}{p} \quad = \quad 1 - \frac{b}{\lceil b(c-1) \rceil} \left[H_{\lceil cb \rceil} - H_b \right]$$

This can be approximated as

$$= \quad 1 - \frac{b}{\lceil b(c-1) \rceil} \left[\ln \lceil cb \rceil - \ln b + \Theta\left(\frac{1}{\lceil cb \rceil}\right) - \Theta\left(\frac{1}{b}\right) \right]$$

For sufficiently large values of b, this is approximated by

$$1 - \frac{\ln c}{c-1}$$

This approximation may be solved for c numerically to obtain average moves of any desired fractional quantity. The following table gives examples of some c values for desired moves, and the lowest value of b after which the true average from $b+1$ to $\lceil bc \rceil$ differs from the approximation by no more than .5%:

desired move	c	lowest b
$.10p$	1.23	150
$.25p$	1.73	102
$.50p$	3.51	40
$.75p$	10.35	14
$.90p$	37.15	4

As in the case of constant moves, this analysis breaks down as the records begin to be ordered by probability of access. We again expect this to reduce the average move distances, but the amount of that reduction is predictable only by assuming a function for the distribution of accesses.

The accuracy of the formula above is also affected by the fact that the size of the list is probably not such that a jump point happens to fall at the end. This means that the records in the last range will contribute a shorter average fractional movement than other ranges. This difference is severe when a larger average move is desired. For example, if we used $c = 3.5$ in order to obtain an average move of $.5p$, then about 7/11 of all of the records are in this last range. These records will contribute low fractional moves to the overall average, which will not be properly offset by the non-existent records in the higher portion of that range. Thus, the true average move may still be shorter than expected, without even taking into account the eventual ordering of records.

It should be possible to reduce this error by assuming that the end of the list occurs midway in a range, or by leaving the size of the list as a variable to be filled in when it is known. Although this would lead to a more accurate result for this model, we do not pursue it since the following algorithms serve the same purpose and may be analyzed more accurately.

3.2. Probabilistic Jumps

The primary difficulty with the methods described above is that the eventual ordering of the records biases the average movement within a given range, making analysis difficult. The following probabilistic functions remove the fixed ranges between jumps, thus allowing calculation of the expected distance from any given record to the backpointer without worrying about where in a given range that record may be.

In the following, the definitions of $JUMP$ are independent of the value of b and thus will be denoted simply as $JUMP(p)$ rather than $JUMP(p, b)$.

3.2.1. Constant Moves

Let the probability that $JUMP(p)$ evaluates to *true* be $1/c$ for any fixed $c \geq 1$. Recall that, unless a record is found in the first location of the list, it will move forward at least one position. It will move further only if $JUMP(p-1)$ evaluated to *false*, which happens with probability $1-1/c$. In this case, the record will move the single space to the previous position in the list plus the expected move distance from that position. This gives the following recurrence for the expected distance a record found at location r will move forward:

$$M_C(r,c) = \begin{cases} 0 & r = 1 \\ 1 + (1 - 1/c)M_C(r-1,c) & r > 1 \end{cases}$$

Although this could be solved as a non-homogeneous first-order finite difference equation, a simple solution can be obtained for $r > 1$ by noting that the equation is equivalent to a finite geometric series and simplifying:

$$\begin{aligned} M_C(r,c) &= \sum_{i=0}^{r-2} \left(1 - \frac{1}{c}\right)^i \\ &= c - c\left(1 - \frac{1}{c}\right)^{r-1} \end{aligned}$$

Note that this result also satisfies the function for $r = 1$.

For $c << r$, c is a good approximation of the expected move distance M_C. For $c \approx r$, $M_C \approx r(1-1/e) \approx .63r$. For $c >> r$, M_C approaches $r-1$ from below. This will only be significant if something more is known about the probabilities of accesses for records such that most of the accesses are expected to be to positions not much larger than c. In cases like this (where c implies desired moves equal to or greater than the expected distance to the front), *move-to-front* is a better choice for an algorithm.

3.2.2. Fractional Moves

Sleator and Tarjan [Sle85] extended amortized results by Bentley and McGeoch [Ben85] to prove that the search time resulting from moving a record forward a fraction of the distance to the front is no worse than a constant times the optimal off-line algorithm. They further showed that the constant is 2 for *move-to-front* and is inversely proportional to the fraction moved. Although *move-to-front* has the best bound by this measure, moving a smaller fraction of the full distance may be profitable if the search sequence has a small degree of locality. The following function allows movement of any desired fraction in the average case.

Let the probability that $JUMP(p)$ evaluates to true be defined as

$$Pr(JUMP(p) \text{ evaluates to } true) = \begin{cases} c/p & p \geq c \\ 1 & p < c \end{cases}$$

for some constant $c > 0$. The expected distance a record located at location r will move forward will be

$$M_F(r,c) = \begin{cases} 0 & r = 1 \\ 1 & r > 1, \quad c \geq r-1 \\ 1 + \left(1 - \dfrac{c}{r-1}\right) M_F(r-1,c) & r > 1, \quad c \leq r-1 \end{cases}$$

We again have a non-homogeneous first-order finite difference equation, but this time the non-constant coefficient complicates matters. Fortunately we can bound the solution fairly easily, obtaining a formula which is acceptably accurate.

First note that, for $c \in N$ and $c \leq r-1$, simple induction shows that $M_F(r,c) = r/(c+1)$. This immediately gives us the ability to set c to obtain simple moves less than or equal to $.5p$. For non-integer values of c greater than 1, the value of M_F can probably be bounded acceptably, but the following step will make this unnecessary.

If we wish to move records forward more than $.5p$, it is necessary to find out what happens when $0 < c < 1$. Unfortunately, simple induction on natural values of c does us no good here. We prove that $M_F(r,c) = r/(c+1)$ for all c in this range, with an error of at most one. Our proof includes, as a bonus, all values of $c > 1$ (non-integer as well as integer).

To prove: for all $c > 0$ and $p \geq \lfloor c+1 \rfloor$,

$$\frac{r}{c+1} - 1 \leq M_F(r,c) \leq \frac{r}{c+1} + 1$$

Proof by induction on r.

Basis: $r = \lfloor c+1 \rfloor$.

Case 1: $0 < c < 1$

Since $r = \lfloor c+1 \rfloor = 1$, $M_F(r,c) = 0$ by definition.

In this case, $1/2 \leq r/(c+1) \leq 1$.

We then know that

$$\frac{r}{c+1} - 1 \leq M_F(r,c) \leq \frac{r}{c+1} - \frac{1}{2}$$

Case 2: $c \geq 1$

Since $r = \lfloor c+1 \rfloor \leq c+1$ and $r \geq 2$, $M_F(r,c) = 1$ by definition.

In this case, $0 \leq r/(c+1) \leq 1$.

We then know that

$$\frac{r}{c+1} \leq M_F(r,c) \leq \frac{r}{c+1} + 1$$

Combining these two cases we see that, for all $c > 0$, when $r = \lfloor c+1 \rfloor$,

$$\frac{r}{c+1} - 1 \leq M_F(r,c) \leq \frac{r}{c+1} + 1$$

Inductive step:

Assume, for all $c > 0$ and for some $r \geq \lfloor c+1 \rfloor$,

$$\frac{r}{c+1} - 1 \leq M_F(r,c) \leq \frac{r}{c+1} + 1$$

It is necessary to prove that

$$\frac{r+1}{c+1} - 1 \leq M_F(r+1,c) \leq \frac{r+1}{c+1} + 1$$

Note that, by definition,

$$M(r+1,c) = 1 + \left(1 - \frac{c}{r}\right) M_F(r,c)$$

Substituting this into the inductive hypothesis, we obtain

$$1 + \left(1 - \frac{c}{r}\right)\left(\frac{r}{c+1} - 1\right) \le M_F(r+1,c) \le 1 + \left(1 - \frac{c}{r}\right)\left(\frac{r}{c+1} + 1\right)$$

$$\frac{r+1}{c+1} - 1 + \frac{c}{r} \le M_F(r+1,c) \le \frac{r+1}{c+1} + 1 - \frac{c}{r}$$

Since r and c are both positive, it is seen that $c/r > 0$ and therefore

$$\frac{r+1}{c+1} - 1 \le M_F(r+1,c) \le \frac{r+1}{c+1} + 1$$

□

Solving for c, it is seen that a move of any fraction f of p may be attained by setting $c = 1/f - 1$. Thus we have shown that $JUMP$ may be used to move records forward by a distance which is within 1 of any desired fraction of the distance to the front of the list, without need of re-reading records to determine the move destination either during or after the search.

Note that this result is similar to the fixed jump, but is independent of the fact that the records will become partially ordered over time.

4. SUMMARY AND OPEN QUESTIONS

We have presented a method of employing probabilistic back-pointers to implement self-organizing lists for sequential search. This method can be used to implement many of the permutation rules that involve moving only the accessed record some distance forward in the list. In the case where each record is large and requires a significant amount of time to read, this method avoids re-reading a large number of records. Examples showed how constant and fractional moves could be achieved on the average.

For initially random lists, the functions for probabilistic jumps achieve the same average behavior as the functions for fixed jumps. As the lists become ordered, the behavior of the fixed jump functions changes in a manner which may be difficult to analyze. The probabilistic functions maintain consistent behavior independent of the permuting of the list.

All of the random $JUMP$ functions presented here have decreasing probabilities as p increases. We have not considered functions for which the probabilities increase with time, or where the difference between p and b is used instead of just p. We conjecture that, in both of these cases, the resultant move-up would be a constant, and therefore would not be of utility since we already have a random function giving constant moves. Nevertheless, it might be worthwhile to pursue these cases and verify their behavior.

There may be useful strategies that move records forward other than a constant amount or a fraction of the distance to the front. It might be interesting to search for these, and determine whether a $JUMP$ function can be made to implement them.

Acknowledgments. Our thanks to the referees, whose comments on earlier drafts led to enhancements in accuracy and completeness.

REFERENCES

1. Bentley, J.L., and McGeoch, C.C. Amortized Analyses of Self-Organizing Sequential Search Heuristics *Commun. ACM 28,* 4 (Apr. 1985), 404–411.
2. Bitner, J.R. Heuristics that Dynamically Organize Data Structures. *SIAM J. Comput. 8,* 1 (Feb. 1979), 82–110.
3. Gonnet, G.H., Munro, J.I., and Suwanda, H. Exegesis of Self-Organizing Linear Search. *SIAM J. Comput. 10,* 3 (Aug. 1981), 613–637.
4. Hester, J.H., and Hirschberg, D.S. Self-Organizing Linear Search. *Computing Surveys, 17,* 3 (Sept. 1985) 295–311.
5. Sleator, D.D., and Tarjan, R.E. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM 28,* 2 (Feb. 1985) 202–208.

CR Categories and Subject Descriptors: 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;* G.3 [**Mathematics of Computing**]: Probability and Statistics—*Probabilistic algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: linear search, self-organizing, move-to-front, paging.

Biographical Sketches

J. H. Hester is a Ph.D. student in the Department of Information and Computer Science at the University of California, Irvine. His research interests include data structures and the design and analysis of algorithms.

D. S. Hirschberg is the Associate Chair of Graduate Studies in the department of Information and Computer Science at the University of California, Irvine. His research interests include the design and analysis of combinatorial algorithms for serial and parallel systems.