Self-Organizing Search Lists Using Probabilistic Back-Pointers

J. H. Hester and D. S. Hirschberg

Technical Report # 85–14

April, 1985

Abstract. A class of algorithms is given 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 occationally 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.

Self-Organizing Search Lists Using Probabilistic Back-Pointers

J. H. Hester

D. S. Hirschberg

University of California, Irvine

ABSTRACT

A class of algorithms is given 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 occationally 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.

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. 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.

Much work has been done on this problem, and there is a wealth of methods now available for permuting records [BIT79, GON81, HES85]. 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 often can be accomplished more efficiently.

1. PREVIOUS WORK

The most general case of the problem places no restriction on the permutation applied to the list or on how much time or space is required for the permutation. However, most analyses in the literature are of permutation algorithms that use only constant extra space (called *memoryless* algorithms), and only move the accessed record some distance forward in the list while leaving the relative ordering of all other records unchanged.

Let ρ be the search sequence. The first record is in location 1 and the last of n records is in location n. 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.

1.1. EVALUATION CRITERIA

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

Since it is assumed that ρ is unknown before the searches are performed, 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 and the *worst case cost* is the greatest average cost per access of any ρ.

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 and the number of future searches considered in the average approaches infinity. There is generally a tradeoff between low asymptotic cost and high convergence rate. In cases where $|\rho|$ is small, the speed with which α converges to its steady state may be a more important measure than its asymptotic cost. This is even more important in cases which demonstrate some degree of locality, i.e. where the accesses in ρ are *not* independent. Cases where those records accessed in the near past are more likely to be accessed again in the near future are common in both English text and programming, where words or variables tend to occur in bursts. In these cases, algorithms with high convergence rates often have lower cost than other algorithms that have lower asymptotic costs.

1.2. *ALGORITHMS*

The following list of permutation algorithms only includes those that will be referenced relative to the results of this paper. Although this is not a complete list of existing algorithms, it does include most of the more commonly addressed algorithms in the literature.

1.2.1. *Move-to-front*

When the accessed record is found, it is moved to the front of the list if it is not already there. This is the most commonly mentioned algorithm in the literature.

The *move-to-front* 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.

1.2.2. *Transpose*

The accessed record, if not at the front of the list, is moved up one position by changing places with the record just ahead of it. This way 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.

1.2.3. *Move-ahead-k*

Move-ahead-k is a compromise between the relative extremes of *move-to-front* and *transpose*. *Move-ahead-k* 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. By this definition, if r is the location of the accessed record, *move-to-front* is *move-ahead-*$(r-1)$ 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 possible 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 proposed by Rivest [RIV76], and addressed later by Gonnet, Munro and Suwanda [GON81]. They showed that, for $j > k$, *move-ahead-j* converges on a steady state faster than *move-ahead-k,* but at the penalty of a higher asymptotic cost.

1.2.4. *k-in-a-row*

k-in-a-row is a meta-algorithm proposed by Gonnet, Munro and Suwanda [GON81]. It can be applied in conjunction with many (if not all) permutation algorithms. An algorithm is applied only if the accessed record has been accessed k times in a row. This does not break the memoryless assumption, since it only needs to remember the last record accessed, with a finite counter for previous consecutive accesses. It was shown that the rate of convergence using *move-to-front* with *2-in-a-row* is the same as that of *transpose* without a *k-in-a-row* meta-algorithm.

1.3. HYBRIDS

Due to the tradeoff between convergence rate and asymptotic cost, Bitner [BIT79] proposed hybrid algorithms that initially use an algorithm with fast convergence until that algorithm approaches its steady state, and then switching to an algorithm with a better asymptotic cost for further searches.

2. ASSUMPTIONS

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. If the keys are extremely large or (more likely) there are a large number of records, 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.

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.

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

3.1. FIXED JUMPS

Assume record x is in location r. Let $JUMP(p, b) = (p/b \geq c)$ for any fixed c. Since p is an integer in the range $[b+1, \ldots, cb]$, the average value of p in terms of b and c, assuming we know nothing about where in this range p lies, is

$$p \quad = \quad \frac{1}{(c-1)b} \sum_{i=b+1}^{cb} i \quad = \quad \frac{c+1}{2} b + \frac{1}{2} \quad \approx \quad \frac{c+1}{2} b$$

Thus the average distance record x (at location r) will move forward is

$$r - b \quad \approx \quad r - \frac{2}{c+1} r \quad = \quad \frac{c-1}{c+1} r$$

Therefore, if we want records on the average to move forward $P\%$ of the total distance to the front, we want

$$\frac{c-1}{c+1} r = \frac{P}{100} r, \quad \text{or solving for } c, \quad c = \frac{100+P}{100-P}$$

For example, if we want to move items forward by an average of 80%, *JUMP* should be *true* when

$$\frac{p}{b} \geq c = \frac{100+80}{100-80} = 9$$

This allows fine tuning of the algorithm for the desired tradeoff between convergence and asymptotic cost.

The above analysis assumes that the list is unordered. This may be true initially, 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. It appears that the effect of this could only be predicted by making further assumptions about the values of the probabilities.

If this *does* have an effect, and the average p as a function of b becomes smaller as the list becomes more ordered, then the average percent of move-ahead distance decreases. 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 chose not to presue due to a similar result we present in a later section.

3.2. PROBABILISTIC JUMPS

The definitions of *JUMP* are independent of the value of b in the following, and thus will be denoted 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:

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

To show that the expected move is about c, we define a *fudge factor* $X_C(r,c)$ such that $M_C(r,c)=c+X_C(r,c)$, and bound the possible values of $X_C(r,c)$.

LEMMA 1:

$$X_C(r,c) = \begin{cases} -c & r=1 \\ (1-1/c)X_C(r-1,c) = (1-c)(1-1/c)^{r-2} & r>1 \end{cases}$$

PROOF:

The value for $r=1$ comes directly from the definitions of M_C and X_C. For $r>1$,

$$M_C(r,c) = 1+(1-1/c)M_C(r-1,c)$$
$$c+X_C(r,c) = 1+(1-1/c)(c+X_C(r-1,c))$$

solving for $X_C(r,c)$ and simplifying gives the recursive form of the result. The closed form follows directly, using the value for $r=2$ as the basis of the recurrence. ∎

Thus, for $r>2$, Lemma 1 and the definition of X_C gives

$$M_C(r,c) = c+(1-c)(1-1/c)^{r-2} = c-c(1-1/c)^{r-1}$$

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.

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

Also note that this method does not have the side-effect of reading the records a second time as would be the case if a pointer were maintained some constant distance behind the accessed record.

3.2.2. FRACTIONAL MOVES

Sleator and Tarjan [SLE85] extended amortized results by Bentley and McGoech [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 will show that the expected distance a record found at location r will move forward is about $r/(c+1)$. As before, define a fudge factor $X_F(r,c)$ such that

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

LEMMA 2: *For $r > 1$ and $c \leq r-1$,*

$$X_F(r,c) = \left(1 - \frac{c}{r-1}\right) X_F(r-1,c)$$

PROOF:

For $r > 1$ and $c \le r-1$,

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

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

Solving for $X_F(r,c)$ and simplifying gives the desired result. ▮

For $c \ge 1$ and $r = \lfloor c \rfloor + 1$, the recurrence gives $M_F(\lfloor c \rfloor + 1, c) = 1$ and our definition of X_F gives

$$M_F(\lfloor c \rfloor + 1, c) = \frac{\lfloor c \rfloor + 1}{c+1} + X_F(\lfloor c \rfloor + 1, c)$$

Combining these two and solving for X_F,

$$X_F(\lfloor c \rfloor + 1, c) = \frac{c - \lfloor c \rfloor}{c+1}$$

Since we assumed $c \ge 1$ we get $0 \le X_F(\lfloor c \rfloor + 1, c) < 1$. Using this as a basis, Lemma 2 then provides the body of an inductive proof that $0 \le X_F(r,c) < 1$ for all $r \ge \lfloor c \rfloor + 1$ and $c \ge 1$.

For $0 < c < 1$ and $r = 1$, the recurrence gives $M_F(1,c) = 0$ and our definition of X_F gives

$$M_F(1,c) = \frac{1}{c+1} + X_F(1,c)$$

Combining these two and solving for X_F,

$$X_F(1,c) = \frac{-1}{c+1}$$

Since we assumed $0 < c < 1$ we get $-1 < X_F(1,c) < -1/2$. Using this as a basis, Lemma 2 then provides the body of an inductive proof that $-1 < X_F(r,c) < 0$ for all $r \ge 1$ and $0 < c < 1$.

Thus, for all $0 < c < r-1$, we see that $-1 < X_F(r,c) \leq 1$, and the expected distance a record will be moved forward is bounded by

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

which shows that *JUMP* may be used to move records up 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.

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 memoryless 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.

All of the random *JUMP* functions presented here have decreasing probabilities as p increases. We have not considered possibilities where the probabilities were increasing over time, or where the difference between p and b was used instead of just p. We conjecture that, in both of these cases, the resultant move-up will 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 L. L. Larmore, for suggesting the use of fudge factors as a simple method of proving bounded solutions to the recurrences.

REFERENCES

[BEN85] Bentley, J.L., and McGeoch, C.C. Amortized Analyses of Self-Organizing Sequential Search Heuristics *Commun. ACM 28,* 4 (Apr. 1985), 404–411.

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

[GON81] Gonnet, G.H., Munro, J.I., and Suwanda, H. Exegesis of Self-Organizing Linear Search. *SIAM J. Comput. 10,* 3 (Aug. 1981), 613–637.

[HES85] Hester, J.H., and Hirschberg, D.S. Self-Organizing Linear Search. To appear in *Computing Surveys.*

[RIV76] Rivest, R. On Self-Organizing Sequential Search Heuristics. *Commun. ACM 19,* 2 (Feb. 1976), 63–67.

[SLE85] Sleator, D.D., and Tarjan, R.E. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM 28,* 2 (Feb. 1985) 202–208.