A New Technique for Self-Organising Linear Searches

P. M. FENWICK
Department of Computer Science, University of Auckland, Auckland, New Zealand

Searches of sequential lists can be improved by moving active elements closer to the start of the list. Several existing techniques are investigated on a selection of text and program files, and compared with some new techniques. One of these new methods, which involves a simple transposition of the element just found with a randomly selected element about halfway towards the front of the list, is found to be an efficient method if the data are held in an array.

Received April 1988, revised June 1988

1. INTRODUCTION

It is well known that the simple linear search of an unordered table can be improved by reordering the table entries according to knowledge gained from searching the table. Two recent papers (Bentley and McGeogh,[1] and Hester and Hirschberg[2]) contain excellent descriptions and analyses of the techniques which are involved, together with extensive references. Bentley and McGeogh in particular mention the following three heuristics which are representative of the methods used. (The descriptions are quoted directly from their paper.)

1. *Transpose* When the key is found, move it one closer to the front of the list by transposing with the key directly in front of it.
2. *Move-to-Front* When the key is found, move it to the front of the list (all other keys retain their relative order).
3. *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.

In each case the idea is to try to keep the 'active' elements as close as possible to the start of the list, so that the average search length is minimised. *Transpose* works with the minimum possible movement of just one position at each reference. It provides a very gradual reordering of the table, in fact a reordering which is so gradual that in many cases the table does not have time to stabilise during its lifetime. *Move-to-Front* is a complete contrast in that each key, when found, is moved to the very front. It certainly optimises access to the most recently accessed element, but at the cost of slower access to all of the other currently active elements. As a compromise, *Count* attempts to maintain an optimum ordering based on the entire history of past accesses, but can pay little regard to current locality effects, except in that these affect the overall history. While it will certainly respond to entries becoming active, it has no real mechanism for an entry to become forgotten after a period of intense activity. Yet another technique is *Move-ahead-k* in which an element, when found, is moved k positions toward the start of the list (Rivest[4]). It may be regarded as a modification of *Transpose* and has been investigated by Tenenbaum.[5]

Both *Move-to-Front* and *Count* may involve extensive data movement if implemented in the obvious way using data in an array. This is a very real problem, especially if the cost of moving an entry is comparable to the cost of a test. It may be eased by using a linked list as the data structure and 'moving' data with appropriate manipulation of the links, but at the cost of space for the links.

In a recent paper Hester and Hirschberg[4] propose a new heuristic *JUMP*, in which a backpointer is moved to the current position if, and only if, the search is still unsuccessful and some function of the search history is TRUE. When the search succeeds the key is then transposed with the element located by the backpointer. *JUMP* is particularly applicable to a linked list implementation (although it can be use with arrays), and with suitable choice of function can have an average behaviour similar to that for *Move-To-Front*, *Transpose*, or *Move-ahead-k*.

2. THE NEW TECHNIQUES

The present paper describes some other techniques for implementing a self-organising linear search. As an initial observation, we may note that neither *Transpose* nor *Move-to-Front* is necessarily ideal, *Transpose* because it is 'weak', and *Move-to-Front* because it is 'strong' in its data movement. What then is the performance of a heuristic which moves data by some intermediate distance? Tenenbaum has investigated some *Move-by-k* heuristics for small values of k. His results may be summarised by the formula $k = N^{1.25}/115$, where N is the table size and k is the optimum distance by which an element should be moved towards the front of the table. By contrast, the new methods of this paper are based on moving data a variable distance. Hester and Herschberg[1] mention a generalisation of *Move-ahead-k* in which data are moved a percentage of the distance to the front, rather than moving a fixed distance.

As an initial attempt (and paralleling the *move-ahead-k* generalisation of Hester and Hirschberg[2]), if *Transpose* (i.e. Move by 1) is too weak, and *Move-to-Front* is too strong, what is the effect of moving an element half-way towards the front? In other words, if an element is found at position i in the table, it is moved down to position $i/2$, with intervening elements moved up. This heuristic will be known as *MoveAM* (Move to Arithmetic Mean). Another possibility, with stronger movement, is *MoveGM* (Move to Geometric Mean). These methods are most immediately applicable to data held in linear arrays as it is then easy to find the insertion point, but they suffer from the cost of moving the data (a cost which has been ignored in the initial measurements, but will be discussed later). Alternative techniques which minimise data movement are to modify the simple transpose method by transposing the found element with one closer to the front of the list: the two methods tried are *TransposeAM* and *TransposeGM*, with obvious meanings. Tests were also run with a simple Linear Search (*Linear*), a linear search of the table sorted into decreasing frequency (*Optimum Static Ordering*), *Move-to-Front*, standard *Transpose* and *Count*, these being established methods which may serve as references to evaluate the new techniques.

The data used were seven Text files, with sizes from 781 to 6241 words and vocabularies of 308 to 1051 words, and eight program source files, from 112 to 2290 words and vocabularies from 40 to 510 words. The programs were mostly Pascal, with two PL/I and one B6700 Algol. Only 'identifiers' were recognised and text punctuation and all program special characters were ignored in the analysis. In all cases the file was read from the start and the table built up as the search statistics were gathered (except of course for the *Optimum Static Order* case where the table had to be constructed first). The situation therefore corresponded to a simple lexical analyser of a text processor or compiler. The tables also include three results from interpreter traces; these will be discussed later. No attempt has been made to use artificial data, constructed according to some assumed distributions. The measured performance is so sensitive to the data that it seems most unlikely that any theoretical model could be a satisfactory representation of reality.

3. RESULTS

Table 1 shows the raw results for the initial tests, with the values being the average search distance as the data were read and the table searched and built, all expressed as a percentage of the final table size. (For *Optimum Static Order* the table had to be constructed and ordered first, and the times given are for only the search phase.) In Table 2 selected methods are then compared with the simple Linear Search for each file. (The more important of these results are presented graphically later in the paper.) All of the tests used arrays for data storage; none used linked lists.

There are two immediate observations from these results. The first is that *MoveAM* and *MoveGM* are often better than the standard techniques. The second is that there is an obvious difference between text and program files. Programs contain many 'special characters' which are clearly identifiable and do not appear as identifiers or words. The corresponding elements of text files are the articles, prepositions, conjunctions and the like; they are all frequent words and all appear in the statistics. Again, the English language has a rich and varied vocabulary and some words may appear few times within a document if synonyms are used extensively. Most program languages have a relatively limited repertoire of words with limited scope for synonyms and the vocabulary is correspondingly restricted. There is a tendency for *MoveAM* to be better for text and *MoveGM* to be better for program source, indicating that programs need the stronger data movement.

To investigate the differences between text and programs, the *MoveAM* algorithm was modified to move the element to 10%, 20%, 30%,...,90% of the distance from the front of the table towards its previous position. While the performance is relatively insensitive to movement distance and there is considerable variation between files, it does appear that for text files the element should be moved to about 33% of its present displacement from the front of the table, while for program source the element should be moved to about 25% of its present displacement. The detailed results are not given here as *MoveAM* is a relatively costly method (as explained in the next section), even though its search performance is quite good.

4. COSTS OF DATA MOVEMENT

None of the preceding tests with the 'Move' algorithms allowed for the time to move the intervening data to create a hole to receive the moved item, if the data is held in a simple array. If it takes as long to move an element one position as it takes to search through one position, the times for *Move-to-Front* will double, while *MoveAM*

Table 1. Average search length, as % of table size

	Words	Vocab	Linear	Optimum order	Move front	Move AM	Move GM	Trans.	Trans. AM	Trans. GM	Count
Text 1	781	308	33.21	25.27	29.94	28.17	29.07	32.58	29.67	31.55	28.76
Text 2	1481	554	32.92	23.80	27.54	26.44	26.92	32.16	28.18	29.81	27.25
Text 3	1744	519	32.58	24.21	28.76	27.52	27.63	31.83	28.80	31.81	28.68
Text 4	1950	557	28.50	22.19	26.51	24.60	25.53	27.84	26.01	27.99	25.24
Text 5	1971	583	33.20	24.39	28.21	27.25	27.41	32.45	28.97	31.36	28.42
Text 6	2379	609	26.42	19.22	21.95	20.96	21.47	25.79	23.13	26.20	22.15
Text 7	6241	1051	24.74	14.44	15.74	15.26	15.36	23.78	17.15	27.89	16.69
Prog 1	112	40	37.06	30.07	36.18	38.08	34.67	36.24	34.80	33.80	33.99
Prog 2	324	66	39.20	26.75	25.17	27.60	24.53	36.54	31.01	32.75	30.47
Prog 3	375	89	43.70	28.94	28.54	32.79	28.36	42.49	34.99	37.94	35.10
Prog 4	499	110	40.85	20.66	22.44	23.07	21.91	38.71	25.76	33.53	23.88
Prog 5	573	97	40.55	22.62	22.26	24.14	21.87	36.17	28.21	29.76	26.03
Prog 6	759	173	51.58	19.03	18.85	20.42	18.40	48.30	29.48	34.63	22.96
Prog 7	1114	104	43.58	18.40	22.37	20.19	21.50	34.42	23.28	32.83	19.60
Prog 8	2290	510	30.91	18.27	19.48	19.73	19.17	29.94	21.46	26.03	21.33
Intp 1	865	23	36.45	26.57	20.66	19.69	19.63	19.83	19.75	25.71	21.46
Intp 2	1697	60	31.50	21.58	9.99	11.23	9.91	20.93	13.85	18.73	19.04
Intp 3	1988	58	39.13	18.33	17.60	15.57	16.80	21.21	16.95	31.97	15.95

Table 2. Average search length, relative to Linear Search

	Opt. order	Move GM	Move AM	Move front	Count	Trans	Trans AM	Trans GM
Text 1	0.761	0.875	0.848	0.902	0.866	0.981	0.893	0.950
Text 2	0.723	0.818	0.803	0.837	0.828	0.977	0.856	0.906
Text 3	0.743	0.848	0.845	0.883	0.880	0.977	0.884	0.976
Text 4	0.779	0.896	0.863	0.930	0.886	0.977	0.913	0.982
Text 5	0.735	0.826	0.821	0.850	0.856	0.977	0.873	0.945
Text 6	0.727	0.813	0.793	0.831	0.838	0.976	0.875	0.992
Text 7	0.584	0.621	0.617	0.636	0.675	0.961	0.693	1.127
Prog 1	0.811	0.936	1.028	0.976	0.917	0.978	0.939	0.912
Prog 2	0.682	0.626	0.704	0.642	0.777	0.932	0.791	0.835
Prog 3	0.662	0.649	0.750	0.653	0.803	0.972	0.801	0.868
Prog 4	0.506	0.536	0.565	0.549	0.585	0.948	0.631	0.821
Prog 5	0.558	0.539	0.595	0.549	0.642	0.892	0.696	0.734
Prog 6	0.369	0.357	0.396	0.365	0.445	0.936	0.572	0.671
Prog 7	0.422	0.493	0.463	0.513	0.450	0.790	0.534	0.753
Prog 8	0.591	0.620	0.638	0.630	0.690	0.969	0.694	0.842
Intp 1	0.729	0.539	0.540	0.567	0.589	0.544	0.542	0.705
Intp 2	0.685	0.315	0.357	0.317	0.604	0.664	0.440	0.595
Intp 3	0.468	0.429	0.398	0.450	0.408	0.542	0.433	0.817

Table 3. Overall Search costs, relative to linear search

	Opt. St. order	move front	Count	Count overall	Trans	Random trans
Text 1	0.761	0.902	0.866	0.979	0.981	0.876
Text 2	0.723	0.837	0.828	0.936	0.977	0.851
Text 3	0.743	0.883	0.880	0.984	0.977	0.874
Text 4	0.779	0.930	0.886	1.001	0.977	0.905
Text 5	0.735	0.850	0.856	0.982	0.977	0.869
Text 6	0.727	0.831	0.838	0.954	0.976	0.844
Text 7	0.584	0.636	0.675	0.770	0.961	0.681
Prog 1	0.811	0.976	0.917	1.072	0.978	0.937
Prog 2	0.682	0.642	0.777	0.913	0.932	0.742
Prog 3	0.662	0.653	0.803	0.963	0.972	0.793
Prog 4	0.506	0.549	0.585	0.675	0.948	0.581
Prog 5	0.558	0.549	0.642	0.745	0.892	0.625
Prog 6	0.369	0.365	0.445	0.527	0.936	0.446
Prog 7	0.422	0.513	0.450	0.496	0.790	0.494
Prog 8	0.591	0.630	0.690	0.794	0.969	0.682
Intp 1	0.729	0.567	0.589	0.614	0.544	0.466
Intp 2	0.685	0.317	0.604	0.634	0.664	0.374
Intp 3	0.468	0.450	0.408	0.432	0.542	0.389

will increase by 50% and *Transpose* will be little affected. The recommended implementation for *Move-to-Front* is to hold the data in a linked list and move data by adjusting the links.

If a linked list is used for *MoveAM* it would need a subsidiary vector to trace the history of each search and locate the 50% (or 33% etc.) distance in each case. The overheads of maintaining this vector are likely to be a considerable fraction of the actual search cost, and the *MoveAM* and *MoveGM* methods are therefore expensive in terms of movement, even though relatively efficient in terms of search distance. As an alternative to a history vector with *MoveAM* and *MoveGM* it is possible to maintain a subsidiary pointer to indicate the destination element. If the subsidiary pointer is advanced along the list by one on alternate steps of the main data pointer we obtain the *MoveAM* method (or every 3rd or 4th step may be better in the light of the above discussion). If the subsidiary, or destination, pointer is advanced by one after first step of the data pointer, then after 3 more steps, and then after further intervals of 5, 7, 9,...,steps of the data pointer we obtain *MoveGM*. However, the cost is probably similar to that of building the history vector and, unless we have an expensive comparison operation (such as programmed string comparison), any of this subsidiary book-keeping is nearly as expensive as a search and must be included in the real cost of the search. The *JUMP* heuristic of Hester and Herschberg avoids the overheads of maintaining and traversing the history vector, but (as they point out) one must be careful of the cost of the test function.

To summarise the costs of data movement (as distinct from the cost of the search itself), *Move-to-Front* is inexpensive with a list and very expensive with an array, while *Transpose* is inexpensive with both. *MoveAM* and *MoveGM* are relatively expensive with both implementations.

Count may be implemented with an array of values or with a linked list. In both cases it will be necessary to search back along the structure to find the correct new position for the target. While a list is probably the more efficient structure, the cost of a reverse scan along a list (examining the count) is still similar to the cost of the forward search (examining the data). We must therefore include the cost of reorganising the list if we want a true evaluation of *Count*. The data movement distance with *Count* is found to be about 4% of the full table, or about 12–15% of the average search length for program and text files. *Count* is therefore rather more expensive than a simple search distance would indicate; the actual values are given later in Table 3 as part of the final results.

5. METHODS WITH LIMITED DATA MOVEMENT

Given that an array searched linearly from the beginning is a simple and convenient form of data storage, it is desirable to consider search techniques which reorganise the table without extensive data movement. The *Transpose* methods form a convenient basis. While the simple *Transpose* gives a quite modest improvement over the simple search, the table shows that *TransposeAM* is relatively efficient. *TransposeAM* was therefore investigated more fully, transposing an element at position i with the element at position $f * i$, where $0 < f < 1$. The results (not given) show generally broad minima around $f = 0.5$. (The cost for favouring one element is an equal penalty in some other arbitrary element; it seems reasonable that the effects should balance at 50% movement for each.) While most of the curves were reasonably smooth, some had quite abrupt discontinuities. On the assumption that these were due to contention between search targets, the algorithm was modified so that the exchange position was not $i/2$, but some randomly chosen element near $i/2$. This change not only reduced the discontinuities, but also improved the performance on most files, usually by 3–5%, but in one case by 27%. In general the performance is similar to that for *Move-to-Front*, and is better than that for *Count*, even without the data movement overheads. There is no systematic effect from varying the selection among the surrounding 3, 5 or 7 elements, provided only that there is a random selection.

RandomTranspose is therefore a reasonable low-cost heuristic for simple applications. With data stored in an array it is little more complex than even the simple *Transpose*, while giving a considerably better performance. Its essence is that the data order adapts reasonably quickly to a newly-active element. For example, the largest of the text files used has a vocabulary of 1051 words and on average it takes only 6 references for an element to move to within 8 positions of the start of the list; in this it is much better than *Transpose*. In comparison with both *Move-to-Front* and *Count* it is much better at 'forgetting' elements which become inactive.

Table 3 summarises the results for the standard methods of *Optimum Static Order*, *Move-to-Front*, *Count*, *Transpose*, and *Random Transpose*, in all cases with respect to Linear Search for that file. All of the values reflect the cost for the 'best' implementation; *Move-to-Front* assumes a linked list, the *Transpose* methods assume an array, and *Count* assumes a list, but with an extra cost for moving the element. *MoveAM*, *MoveGM* and *TransposeGM* are omitted because of their poorer performance or high cost. These results are also shown in a graphical form in Figure 1. For text files there is little to choose between the traditional *Move-to-Front* and the new *Random Transpose*, provided of course that data movement is minimised by implementing *Move-to-Front* as a list and *RandomTranspose* as an array. *Move-to-Front* is somewhat better for program files. In all cases

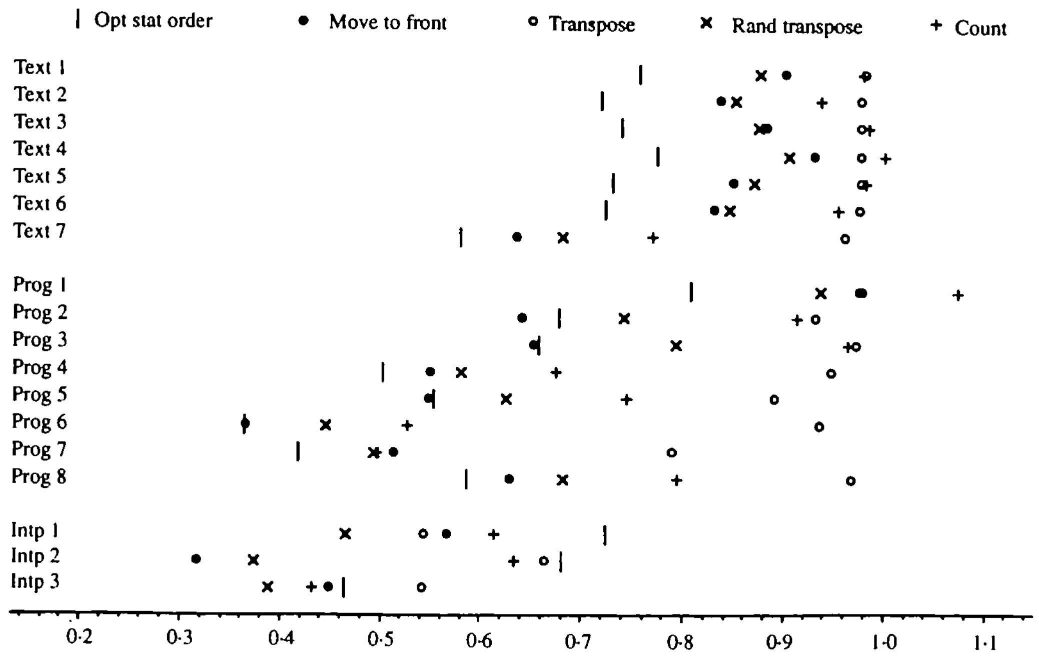

Figure 1. Search cost relative to linear search.

these two methods are better than *Count* when its unavoidable data movement cost is included.

6. DYNAMIC EFFECTS

A trace of the names in an interpreter should show a quite different behaviour from the text and source files used in these investigations. The files used so far have a modest degree of locality within procedures (or perhaps paragraphs), but an actual interpreter should show much more locality as a result of loop execution.

Two interpreter files have been analysed. The first smooths data from a spectrometer and locates peaks within the spectrum, and has a relatively simple loop structure. Two 'traces' are given – a partial one which includes only the initial filtering and differentiation of the raw data, and the full one which includes also the final peak location and line fitting. The two traces are sufficiently different in behaviour to warrant the inclusion of both. They are shown as *Intp 1* and *Intp 2* in Table 3. These, and especially *Intp 1*, are representative of numeric programs which are dominated by loops with little logical decision involved. The second case, shown as *Intp 3*, is a program which generates a Huffman code, given the symbol probabilities. It is mostly numerical, but involves much more testing and conditional execution.

The first observation is that almost any method of reorganising the list gives a useful improvement in performance. Even the poorest cases with the interpreters give more improvement than is achieved with the best methods for many text files. *Optimal Static Order* is now one of the poorer methods, whereas it was one of the best ones for text files, demonstrating the effects of locality in the data. An important observation from all of these results is that the performance of all of the heuristics is very data sensitive. Variations of a few percent between methods are insignificant compared with the variations between data.

Once again, the two best reorganisation techniques are the traditional *Move-to-Front* and the new *Random Transpose*. There is some evidence from the detailed results for the Interpreter traces that *Random Transpose* may be improved slightly if the transposed element is about 60–70% up from the start towards the target element, rather than the 50% which has been used.

7. CONCLUSIONS

Some established methods of self-organising linear searches have been investigated on several text and program source files and compared with some new methods. The best of the new methods involves a simple transposition of the accessed element with a randomly chosen element approximately halfway towards the start of the table. It is found to give a similar search performance to the established *Move-to-Front* method, is often better than the other methods, and is appropriate where data must be held in an array. The new *RandomTranspose* heuristic complements the *JUMP* heuristic which is designed more especially for linked-list data storage. It is interesting to note that both *Random Transpose* and *JUMP* improve the performance by introducing a random aspect to the choice of transposition target.

Acknowledgements

Thanks are due to the referee for especially helpful comments and criticisms. He also pointed out the most recent paper on the *JUMP* heuristic (which appeared after the manuscript was prepared), and emphasised the complementary nature of *JUMP* and the new *Random Transpose*.

REFERENCES

1. J. L. Bentley and G. C. McGeogh, Amortized Analyses of self-organizing sequential search heuristics. *Commun. ACM 28* 4, (1985).
2. J. H. Hester and D. S. Hirschberg, Self-Organising Linear Search. *Computing Surveys 17* 3, 295–311 (1985).
3. J. H. Hester and D. S. Hirschberg, Self-Organizing Search Lists Using Probabilistic Back-Pointers. *Commun. ACM 30* **12**, 1074–1079 (1987).
4. R. Rivest, On Self-Organizing Sequential Search Heuristics. *Commun. ACM 19* 2, 63–67 (1976).
5. A. Tenenbaum, Simulations of dynamic sequential search algorithms. *Commun. ACM 21* 9, 790–791 (1978).

Announcement

23–27 March 1992

International conference on Extending Database Technology, Palais Auersperg, Vienna, Austria

The Conference

EDBT 92 will be a forum for presentation of new results in research, development and applications of database technology. The conference will favour the sharing of information between the researchers and practitioners and outline future developments of database systems and applications.

Tutorials will be offered in the first two days of the Conference, and keynote speakers will be invited. The Conference and Tutorials will be held at the Palais Auersperg in Vienna.

Topics of interest

EDBT 92 will accept scientific and technical papers on all areas related to database technology. Major topics of interest include, but are not limited to the following.

- Deductive databases
- Knowledge bases
- Multimedia databases, Hypermedia systems
- Object-oriented database systems, Object managers
- Environment and technology to support database design and programming
- New applications of databases
- Active databases and real-time databases
- Performance issues and implementation techniques
- Extensible systems
- Database machines
- User interfaces
- Theory of data and knowledge bases

Conference organisation

Conference chairperson: C. Delobel, University of Paris-Sud and GIP Altaïr

Vienna is a very busy and popular tourist centre, in order to facilitate arrangements and to ensure hotel reservation you are kindly requested to contact the organisers as soon as possible.

For further information on EDBT 92 please contact **Brigitte Haberstroh**, Conference Secretariat, EDBT 92, Technical University Vienna, Paniglgasse 16, A-1040 Vienna, Austria. Tel.: +43 (1) 58801/6122. Fax: +43 (1) 5055304. E-mail: haberstroh@vexpert.dbai.tuwien.ac.at.