Chapter 4

The searching algorithm

[Note: This chapter has been written as a self-contained paper]

4.1. Introduction

Much current work in IR is concerned with the reconciliation of Boolean and associative retrieval methods. One problem within this general area is the design of front-end systems which will organise a weighted search into a series of Boolean search statements, which can then be transmitted to a traditional Boolean search system. The front-end system should then process the output of the Boolean searches, to present documents to the user in rank order.

This paper presents an algorithm for generating an appropriate series of Boolean search statements, and storing the results in an appropriate form, for such a front-end. The algorithm presented has some advantages over the two previously proposed.

4.2. Previous suggestions

The central problem is that for a request of any size, a very large number of Boolean statements may theoretically be generated - in fact at least 2**n - 1 for an n-term request. Many of these statements may retrieve no documents. Jamieson's (1979) approach is to select out a number of the possibilities by doing initial term-pair searches: any term-pair which retrieves no documents eliminates a whole series of more complex statements.

Morrissey's (1981) approach, following earlier work by Harper (1980), is to send only single-term requests, and to bring back to the front-end large sets of document identifiers. The front-end then performs the necessary comparisons between the sets to arrive at the ranked list.

4.3. Constraints on the present approach

The present algorithm was devised in the context of an experimental front-end system, connected into the public packet-switched network and interrogating a commercial on-line retrieval host.

In initial experimentation, it was found that requesting large sets of document identifiers was out of the question for this particular host, because the transmission of these sets over the network would take an inordinate amount of time. (Clearly the host's software is designed on the assumption that there is a human being at the other end, and no serious attempt has therefore been made to speed up such processes beyond a certain point.) This effectively eliminated Morrissey's method.

It was decided, therefore, to adopt something closer to Jamieson's approach. However, the present algorithm differs somewhat from Jamieson's. Two particular considerations led to the present alternative. It was thought desirable to have a more systematic way of eliminating some of the possible search statements, perhaps using other criteria. There would also be some advantage in building up complex Boolean searches stage by stage, rather than sending a complex statement in one operation.

The result of these considerations is a backtracking algorithm implemented as a recursive function, which explores and builds up a tree-structure of Boolean searches, from the bottom up (Jamieson's might be described as top-down). It bears some similarity to Weiss's (1981) algorithm (but of course it uses an existing database and Boolean search system, rather than requiring the database to be organised in a special way).

4.4. Description of the algorithm

The root of the tree is the set of documents defined by ORing all the terms. The basic procedure is to AND in each term in sequence (decreasing weight sequence is appropriate, but not vital) and then to backtrack, replacing each ANDed term by the same term NOTed, and exploring the corresponding branches. Thus the sequence for a three-term request would be:-

A	AB	ABC
		AB¢
	A¢	A¢C
		A¢¢
¢A	¢AB	¢ABC
		¢AB¢
	¢A¢	¢A¢C
		¢A¢¢

(where ¢ is "not C").

However, some branches may not be fully explored. Criteria for stopping exploration of a particular branch are:

(a) that the current search has retrieved no documents;

(b) that no document on the branch can exceed in matching-value the x best documents found so far.

A formal description of the algorithm is given in section 4.8.

4.5. Use of the algorithm

The algorithm as described can be used with any simple (sum of weights) matching function. The terms would normally be sorted in decreasing weight order (to maximise the extent to which large branches can be excluded), and an initial search with all the terms ORed must be performed. The results of the application of the algorithm, in the form of a tree-structure as described, can form the basis for an algorithm for presenting documents to the user in ranked order (clearly, this

involves additional commands to the host, to bring back details of the documents).

Subsequent modifications to the request, in the form of (a) modified weights, (b) additional terms, can be accommodated very easily. Modified weights have the effect of modifying the matching values associated with each node; additional terms are added to the end of the list, generating (potentially) two new nodes at the end of each branch. In either case, the algorithm needs to be called up again, but many of the results previously obtained can be used again without repeating searches.

4.6. Experiences

The algorithm has been implemented as part of a relevance feedback system (weights are initially based on collection frequency and subsequently on relevance data - Robertson and Sparck Jones, 1976; Croft and Harper, 1979). It is written in C on an LSI 11/23 connected directly to PSS, the British Telecom packet-switching network. It is designed to interact with the Data-Star service of Radio-Suisse, using the Medline database.

Clearly, a crucial aspect of such an algorithm is the length of time it takes, which itself depends on the number of Boolean searches which must be sent to the host. This is likely to be exponentially related to the number of terms in the query.

On the assumption that it would be unreasonable to expect a user to wait more than about five minutes for one search, our system is in effect limited to queries of about 7 or 8 terms. Some sample search times are as follows:

4 terms: 50 seconds; 1 minute
6 terms: 2 minutes 20 seconds;2-40
8 terms: 5 minutes 30 seconds;6-20

(for these searches, the system was looking for the 15 best matching documents). The second example in each case involved 12, 38 and 110 Boolean requests respectively.

4.7. Discussion

The algorithm presented exhibits some valuable positive features. It is relatively simple (provided only that the programming language allows recursive function definitions). Each search performed is useful, in the sense that its results are stored and may subsequently be used. In particular, it accommodates modifications to the request efficiently.

There are, however, some limitations. The size of the query (number of terms) is clearly limited, if only by the time required to do all the necessary searches (the implementation discussed is in fact limited to eight terms). Further, the algorithm as described does not allow term deletion; these two limitations taken together are fairly severe.

It would clearly be possible to implement a term-deletion algorithm; it would involve mapping one branch of the tree onto another, node for node. But with the data-structure as presently defined, this would involve some extra searching. A modification of the data-structure which might obviate this necessity would be to replace the search-set number by a string, which could be a search statement. This change might also permit a speeding up of the main search algorithm, in that the NOT nodes need not then be searched explicitly.

A further modification which might improve the efficiency of the algorithm would be to include additional criteria for stopping the exploration of a branch. In particular, a probablistic analysis might be devised which would exclude large chunks, on the basis that they were not likely to contain better documents than those found already.

It should be noted that we have not implemented Jamieson's original algorithm, and have therefore not been able to make any direct comparisons as to efficiency. Some such comparison should be made in the future.

4.8. Appendix

This appendix presents the algorithm in the form of a recursive function **Rsearch**.

The data structures required are a linked list of query terms and a tree of search nodes. Each query term has associated with it:

- **Weight:** the term weight
- **Num:** the search set number of the single-term search on the host
- **Next:** a pointer to the next term

Each search node has associated with it:

- **Num:** the search set number of the corresponding search
- **Count:** the number of documents retrieved by this search
- **Wght:** the matching value at this node (ie the total of the corresponding term weights)
- **With:** a pointer to the node which is derived from the present node ANDed with the next query term
- **Wout:** a pointer to the node which is derived from the present node NOTed with the next query term.

Branches in the search tree are terminated by tags **FREE** or **NULL** in the pointer position - FREE means "not yet searched"; NULL means "no documents in this branch".

At each recursion of the Rsearch function, it starts from a node that has already been searched (**Prev**), and performs the searches for the corresponding With and Wout nodes. The call to the function passes a pointer to the next query term Q and a pointer to Prev.

A second recursive function (**Rmwt**) is provided which finds the

matching values of the x top ranking document in the entire tree as presently known. The matching value of the x'th ranked document is Mx. Rsearch will not explore any branch of the tree which cannot yield any documents with matching values better than Mx.

After a search is sent to the host, **Result** is the returned number of documents retrieved; **Num** is the search set number. **Depth** is the current depth in the search tree.

```
Rsearch(Q,Prev,Depth)

if:     Q is NULL    (ie no more query terms)
then:   return
if:     Depth is one
then:   call Rmwt
if:     With is FREE
then:   send search request "Prev-Num AND Q-Num"
        if:     Result not zero
        then:   set With-Num = Num
                set With-Count = Result
                subtract Result from Prev-Count
                set With-Wght = Prev-Wght + Q-Weight
                set With-With, With-Wout = FREE
                if:     With-Wght > Mx
                then:   call Rmwt
        else:   set With = NULL
if:     With not NULL
then:   call Rsearch(Q-Next,With,Depth+1)

if:     Wout is FREE
then:   if:     Prev-Count is zero
        then:   set Wout = NULL
        else:   set Total = Prev-Wght + weights of remaining terms
                if:     Total > Mx
                then:   set Wout-Count = Prev-Count
                        set Wout-Wght = Prev-Wght
                        set Prev-Count = zero
                        set Wout-With, Wout-Wout = FREE
                        if:     With not NULL
                        then:   send search request "Prev-Num not Q-Num"
                                set Wout-Num = Num
                        else:   set Wout-Num = Prev-Num
if:     Wout not NULL or FREE
then:   call Rsearch(Q-Next,Wout,Depth+1)
return
```

The algorithm as presented works for repeated searches on the same request, with new weights or new terms added, doing only those additional searches that are needed (using the results of previous searches where it can). For new terms, some preliminary work needs to be done, in particular the base set has to be expanded (the additional documents have to go in at the end of the Wout-Wout-Wout... chain).

The implemented version includes some additional data and operations:

The node data includes **Bmap** (a bit-map of the search pattern - terms present and absent) and **Print** (the number of documents already seen by the user). Both of these have to be adjusted by Rsearch. If a node that has already been searched is split by the addition of a new term, it must be assumed that none of the documents at the new nodes have been seen, since the program does not know to which of the new nodes the seen documents belong. Print is used by Rmwt, which tries to find the matching values of the x top ranking unseen documents.

Rsearch displays a representation of the tree as it goes along.

Query term data includes **Searched** (whether or not this term has yet been reached by Rsearch), which is used to decide how to introduce new terms into the list.

In order to conserve space, the implemented version allocates space for the data associated with each node as it creates that node.