불리언 질의 구성 알고리즘의 시간복잡도 분석

김 남 호[†] · Donald E. Brown[††] · James C. French[†††]

요 약

알고리즘의 성능은 여러 측면에서 측정할 수 있다. 하나의 질의 구성(Query Formulation) 알고리즘이 개발되었다고 가정할 때, 이 알고리즘 검색 성능의 효과성(재현율과 정확율)이 다른 알고리즘에 비해 우수하다 하여도 질의 작성 시간적인 측면, 즉 효율성 에서 다른 알고리즘에 뒤진다면 모든면에서 우수하다고 평가하기는 어렵다. 본 연구에서는 질의나무(Query Tree)라고 불리는 자동 질의 재구성 알고리즘과 다른 2개의 알고리즘(DNF method, Dillon's method)을 이론적 측면과 실시간 측정을 Sun SparcStation 2를 이용하여 비교하여 보았다. 3가지 test set인 CACM, CISI, 그리고 Medlars를 이용하여 실험한 결과 질의나무 알고리즘이 이론적, 실시간 측면 모두에서 가장 빠른 알고리즘이라는 결과가 나왔다.

Time Complexity Analysis of Boolean Query Formulation Algorithms

Nam Ho Kim[†] · Donald E. Brown[††] · James C. French[†††]

ABSTRACT

Performance of an algorithm can be measured from several aspects. Suppose there is a query formulation algorithm. Even though this algorithm shows high retrieval performance, i.e., high recall and precision, retrieving items can take a long time. In this study, we analyze the time complexity of automatic query reformulation algorithms, named the Query Tree, DNF method, and Dillon's method, and compare them in theoretical and practical aspects using a real-time performance(the absolute times for each algorithm to formulate a query) in a Sun SparcStation 2. In experiments using three test sets, CACM, CISI, and Medlars, the Query Tree algorithm was the fastest among the three algorithms tested.

1. Introduction

With the proliferation of computers, it becomes easier to produce and access information than ever before. In addition, the technology of digital network enables us to connect computers in remote areas and to retrieve a large amount and variety of information on-line regardless of their location. A major problem is that it gets harder to find right information as we have more information available. For example, information on the Internet which connects the millions of computers and grows everyday is so overloaded that a user in general has a difficult time to find information he desires. On the Internet there are special types of computers, which guide a user and search for requested information. These computer servers are Internet search engines that allow a user either to go through a menu system(categorized by subjects) or to formulate a query. Good examples of such systems

† 정 회 원:선문대학교 산업공학과
†† 정 회 원:Systems Engr. Dept., U. of Virginia
††† 정 회 원:Computer Science Dept., U. of Virginia
논문접수:1996년 7월 4일, 심사완료:1996년 12월 3일

are the Yahoo, InfoSeek, Lycos, and Altavista. These systems are a special type of Information Retrieval (IR) Systems.

The most fundamental IR model is the Boolean retrieval model and most of commercial IR systems are based on Boolean query formulation. The Boolean query is a list of keywords connected by the Boolean operators(AND, OR, and NOT), and the search is based on exact matching. Information is generally stored in inverted files. The advantages of this model are simple structure, easy implementation, and fast retrieval. Its inverted file structure allows set operations to be carried out easily. From this on, our main focus in this paper is a Boolean-based IR system.

A main purpose of information retrieval systems is to deliver user-requested information quickly, accurately, and easily. Current IR systems are often criticized for delivering only a portion of what is requested along with nonrelevant information. Improvements in retrieval performance(i.e., ability to retrieve only the relevant information) can be achieved from more effective query formulation. The quality of a query fed to an IR system has a direct impact on the success of the search outcomes, i.e., garbage in garbage out. In fact one of the most important but frustrating tasks in information retrieval is query formulation. Most IR system users, especially novices, experience some degree of difficulty in transforming their information needs into the required query language(e.g., Boolean expression) of an IR system. In addition, there is very little help provided by current IR systems. Users usually employ a trial-and-error search and often have to rely on an intermediary(e.g., librarian) in searching for right information. As more computers and networks become available, experts would be less likely to be present in every site and expensive to employ. There is a definite need for an IR system which cannot only help users formulate an effective query but also increase retrieval performance.

In the research of Kim, Brown and French[5, 6], a Boolean query formulation algorithm(*called query tree*) which automates query formulation from the user's relevance feedback(i.e., a set of information that a user assigns if it is relevant to his search or not) is developed. The performance of an IR system can be measured in large from two different perspectives, i.e., effectiveness and efficiency [13]. In short, the effectiveness measures the quality of the retrieved information, (e.g., among retrieved items, how many of them are relevant?) On the other hand, the efficiency concerns with issues such as accessibility of IR systems, ease of use, cost of retrieval, and response time. Even though an IR system achieves high effectiveness, the system can't be regarded as good unless its response time is tolerable to a user.

In this study, based on the research of Kim [5], the efficiency of query tree and other query formulation algorithms developed by Salton, et. al. [14] (named DNF method) and by Dillon, et. al. [3, 4] (named Dillon's method) is compared and analyzed from the point of query formulation time, i.e., time taken to formulate a Boolean query. (Analyses of retrieval effectiveness for these algorithms are presented in article [6]). The next section describes how analysis of algorithm's efficiency is performed.

2. Methods of Time Complexity Analysis

In producing a complete complexity analysis of the query formulation time of an algorithm, two phases are necessary: a priori analysis and a posteriori testing. In a priori analysis a function is obtained, which bounds the algorithm's computing time(the frequency of execution of the statements in the algorithm). On the other hand, in a posteriori testing actual statistics about the algorithm's consumption of time is collected while it is executing [2]. In our priori analysis we use the asymptotic notation called O-notation(e.g., $O(n)$, $\Theta(n)$, etc., for definition, see [2]), which are customary in the analysis of time complexity. For example, $O(n)$ is defined as

$f(n) = O(g(n))$ if and only if there exist two positive constants c and n_0
such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

Two cases, the best and worst cases of an algorithm are considered in our priori analysis. The best-case priori analysis considers the shortest time which an algorithm can take while the worst-case computes the longest time. For a posteriori testing, actual query formulation time of an algorithm is collected using SparcStation 2. From three test sets, CACM, CISI, Medlars, an average query formulation time of each algorithm is computed by keeping the total time and dividing it by the total number of queries in a test set. The results of these analyses including a brief description of algorithms' effectiveness, recall and precision are discussed in the final section.

The following section describes how the query tree algorithm works and what its query formulation procedure is, including a priori analysis of its query formulation time. Section 3 and 4 explain DNF and Dillon's methods, respectively, along with their priori analyses. The last section summarizes the results of priori analysis and posteriori testing, and reports experimental results run on three benchmark test sets (CACM, CISI, and Medlars).

3. Query Tree

Query formulation in information retrieval is in fact a classification problem. That is, a Boolean query is a classifier which classifies information in a database into two classes, i.e., relevant and nonrelevant information. The retrieved information belongs to the relevant class and the rest to the nonrelevant. Based on the CART(Classification and Regression Tree [1]), which is a tree-structured classification algorithm, a tree classifier, named *query tree* capable of assigning information to either relevant or nonrelevant categories was developed in [5, 6]. To complete a query tree, two main phases need to be processed: query tree growing

```
(Initialization)
1.1  Let $R = \{ d \mid d \in D_I \}$   (*$D_I$ = initial input doc. and $R$ = root node *)
     $N\_stack$ = Push($R$)   (* Push the root node into the stack $N\_stack$. *)
1.2  $T_I$ = $\{t \mid t = 1$ and $t \in D_I\}$   (*collect term t assigned in $D_I$ *)
(Tree Growing)
WHILE ($N\_stack \neq \varnothing$) DO
2.1  $C\_node$  Top($N\_stack$)
2.2  $\Psi(C\_node) = \min_i [\Pr(i \mid C\_node)]$
       where $\Pr(i \mid x)]$ is the proportion of data belonging to class $i$ at node $x$.
2.3  IF ($\Psi(C\_node) < \delta$) and ($C\_node \neq$ root node)
     THEN
     2.3.1  IF $\Pr(i \mid C\_node) > \Pr(j \mid C\_node)$
            THEN  $C\_node$ = class $i$   (* Set $C\_node$ as terminal node
            ELSE  $C\_node$ = class $j$      and assign a class to $C\_node$*)
     ELSE
     2.3.2  $t^* = \min_t M_a(C\_node, t)$  and $t \in T_I$
     2.3.3  Create the left and right child nodes for $C\_node$ and
            FOR each $d \in C\_node$
                IF $t^*$ is present in $d$
                      THEN assign $d$ to the left child node
                      ELSE assign $d$ to the right child node.
            END OF FOR
     2.3.4  Put these child nodes in $N\_stack$.
     END OF-IF
END OF WHILE
```

(Fig. 1) Procedure for the query tree growing

and query tree pruning. Then, from a pruned query tree a Boolean query is generated to be finally submitted to the system.

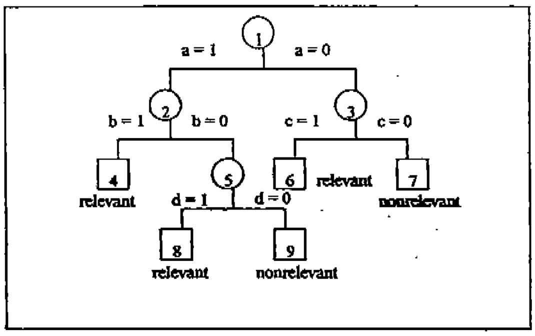
A brief pseudo algorithm for query tree growing is listed in (Fig. 1). Since the main interest of this study is in time complexity analysis and comparisons of the different query formulation algorithms, description of the query tree algorithm will be short here. In addition, the query tree pruning procedure is omitted in this paper because the performance of a query tree with pruning didn't show any improved performance (i.e., recall and precision) compared to that of the one without pruning. For interested readers, please refer to [5, 6].

The main idea of the query tree growing is to identify and capture characteristics of the input data and to group them into a same class(e.g., what makes this input data belong to a certain class?). Growing a query tree consists of the three operations[1]: (1) determination of terminal nodes(2.3 in Fig. 1), (2) selection of splitting criteria for nonterminal nodes(2.3.3 in Fig. 1), and (3) assignment of the terminal nodes to a class (2.3.1 in Fig. 1).

In the initialization phase, the input documents (D_I)[1] are prepared and the root node is pushed into the *N_stack* (Step 1.1). Then, the index terms assigned in the input documents are saved in T_I (Step 1.2). A node is popped from the *N_stack* (Step 2.1) and its impurity level (Ψ) is computed (Step 2.2). If the current node of the Ψ is under the threshold value, δ and not a root node, the class of that node is determined in Step 2.3.1; otherwise, go to Step 2.3.2 and find an index term t^* which provides the lowest misclassification cost, M_α where

$$M_\alpha(t, x) = \alpha \cdot N_{r/n}(t, x) + (1-\alpha) \cdot N_{n/r}(t, x)$$

$$\text{where } \alpha = \frac{N_r(x)}{N_r(x) + N_n(x)}$$



(Fig. 2) An example of query tree

$N_r(x)$ and $N_n(x)$ be a number of relevant and nonrelevant documents at node x, respectively, and $N_{r/n}(t, x)$ and $N_{n/r}(t, x)$ be the number of documents assigned as relevant given that they are nonrelevant, and the number of documents assigned as nonrelevant given that they are relevant at the node x by the term t, respectively. In Step 2.3.3, the right and left child nodes of the current node are created and the documents assigned to the current node are distributed to the child nodes according to the presence of the term t^*.

```
(Query Generation)
1. FOR (each node x in query tree) DO
      IF ( $x \in X_{Tr}$ ) THEN
         Set C_node = x
         WHILE (C_node ≠ root node) DO
            Collect a term t in parent(C_node)
            IF t = 1 THEN collect it in And_Term
            IF t = 0 THEN collect 'not' + t in And_Term
            Set C_node = parent(C_node)
         END (WHILE)
         Connect terms in And_Term with 'and' and store it in Or_Clause
      END (IF)
   END OF FOR
2. Formulate a query in DNF by connecting clauses in Or_Clause with 'or'
   operator.
```

(Fig. 3) Query generation from a query tree

Finally the created child nodes are pushed to the *N_stack*. Steps from 2.1 to 2.3.4 are iterated until the *N_stack* is empty. An example of the query tree constructed is illustrated in (Fig. 2). The alphabets (e.g., a, b, c, etc.) in (Fig. 2) represent t^* at each node.

After the query tree is completed, the documents in the database D can be classified by sending them through the root node and see where they end. Examining every document in the database would be very inefficient, especially for a large database. The query tree can be easily transformed into a Boolean query in disjunctive normal form (DNF). Call a query generated from the query tree the *tree-derived query*. The transformation procedure is described in (Fig. 3). First, collect the T-nodes (i.e., terminal nodes) classified as relevant and denote them as X_{Tr}. For every x in X_{Tr}, starting from the node x, read and collect its parents' splitting index terms recursively up the root node; connect each term in the collected set with the Boolean operator *and*. If the term is equal to zero, add the Boolean operator *not* in front of the term. After finishing this process with every relevant terminal nodes, connect the *anded* clauses (i.e., conjuncts) with the Boolean operator *or*. The Boolean query generation procedure is described in (Fig. 2). If a Boolean query is generated using (Fig. 2), it would be

[1] For experimental purpose, the input documents were selected randomly from the database D.

(b AND a) OR (d AND (NOT b) AND a) OR (C AND (NOT a)).

3.1 Priori Analysis of the Query Tree

In our priori analysis, the best-case and worst-case for the query tree algorithm were investigated and we are only interested in a single query formulation, not including the retrieval of documents from the database. As described in the previous section, query formulation in the query tree algorithm consists of three procedures: tree growing, tree pruning[2], and tree transformation to a query in DNF.

Let D_I be a set of input documents from the database D and T be a set of index terms assigned to the documents in the database D. Define T_I as a set of index terms $(T_I \subseteq T)$ such as

$$T_I = (t \mid A(t, d) = 1 \text{ for } \forall t \in T \text{ and } \forall d \in D_I)$$

where $A(t, d)$ is equal to 1 if index term t is assigned to the document d, otherwise 0. Assume that, $|T_I| = k$, $|T| = m$, $|D_I| = n$, and $|D| = h$.

The best-case (denoted by T_{best}(query tree)) for the algorithm occurs when there is a term t^* in T_I which assigns all the relevant documents at the root node to its left child node and the nonrelevant to its right child node. First, at the initialization procedure, Step 1.1 (in Fig. 1) takes n steps for assigning input documents to the root node and 1 step for putting the root node into a stack; Step 1.2 takes $n \cdot m$ steps to collect the index terms assigned to the input documents; the while loop takes 1 step to check if the stack is empty; Step 2.1 takes 1 step to pop a node from the stack and Step 2.2 takes $2(n+1)$ steps since the number of the classes is 2 and $n+1$ steps to compute a probability; Step 2.3 takes 2 step; since the node is the root node the process moves to the Step 2.3.2 and it takes $4n+6$ steps to compute the misclassification cost and k steps to find t^*, thus, total $(4n+6)k$; Step 2.3.3 takes $2n$ step to assign input documents to the child nodes; Phase 2.3.4 takes 2 steps to put them in the stack. Returning to the while statement, Steps 2.1, 2.2, and 2.3 are executed and with zero impurity level, the process moves to Step 2.3.1 which takes 2 steps. The same process goes to the rest child node. With an empty stack (it takes 1 step to check the stack's status), the algorithm stops. Thus, the total number of executed statements (T_{total}) is

$$\begin{aligned} T_{total}(\text{growing}) &= n+1+n \cdot m+(1+1+2n+2) \\ &\quad +(4n+6)k+2n+2+2+1 \\ &= n(m+4k+5)+6k+10 \end{aligned}$$

Without loss of generality, we may assume $k < m$.

$$T_{total}(\text{growing}) \le n(m+4m+5)+6m+10$$

Finally the tree-driven query will consist of a single term, and take a single execution time to transform a tree query into DNF. Therefore, for some constant c_1,

$$T_{total}(\text{query tree}) \le c_1 \cdot n \cdot m$$

Hence,

$$T_{best}(\text{query tree}) = O(m \cdot n).$$

The worst-case (denoted by T_{worst}(query tree)) is more complicated than the best-case. The worst-case will occur when the tree is grown to a full size, i.e., every terminal node contains a single document. With n number of input documents, there will be n number of terminal nodes and $n-1$ number of nonterminal nodes. Since every nonterminal node is split, this implies that there will be $n-1$ node splits during the tree growing procedure. The total execution time is (n

[2] As described, the tree pruning procedure is not implemented in this research, and the time complexity analysis of the algorithm is analyzed without the tree pruning procedure.

-1) multiplied by the execution time taken during Step 2.1 through 2.3.4 in (Fig. 3-6). That is,

$$\begin{aligned} T_{total}(\text{growing}) &= (n-1)(1+1+2n+2+(4n+6)\cdot k+2n+5)) \\ &= (n-1)(4n+4n\cdot k+6k+9) \\ &\leq c_2\cdot m\cdot n^2 \quad \text{for some constant } c_2 \end{aligned}$$

In the query tree, there are n number of terminal nodes and $n-1$ nonterminal nodes. For the nonterminal nodes, they do not satisfy the condition of the IF statement in (Fig. 3) that the process returns to the FOR loop without any computation. On the other hand, the terminal nodes runs at most n WHILE loops. Therefore, the total execution time for the tree transformation procedure in (Fig. 3) is bounded by

$T_{total}(\text{transforming}) \leq c_3\cdot n^2$ for some constant c_3

Hence,

$$T_{worst}(\text{query tree}) = c_2\cdot m\cdot n^2 + c_3\cdot m\cdot n^2 \leq c_3\cdot n^2$$

Assuming that $n \ll m$ (the number of input documents is much less than that of index terms in the database),

$$T_{worst}(\text{query tree}) = O(m\cdot n^2)$$

4. DNF Method

Salton and his colleagues[12, 14] developed an automatic Boolean query reformulation technique called the disjunctive normal form(DNF) method. A distinguishing feature in this algorithm is that the user can specify the number of documents he wants to retrieve. A summarized procedure of the DNF algorithm is shown in (Fig. 4). The procedure is for a single query formulation and can be extended by returning a subset of the retrieved documents with its relevance.

The DNF algorithm computes the term importance of index terms in the set T_I (previously defined) using a function called relevance weight (*rel_wt*), which is

```
0. Prepare input documents $D_I$ and collect
       $T_I = \{t \mid t = 1 \text{ and } t \in D_I\}$
1. FOR (each $t$ in $T_I$)
       Compute relevance weight, rel_wt($t$)
   END OF FOR
2. Select the best 70 terms by rel_wt and keep them in Heap 1
       along with their freq.
3. FOR $i$ =1 to $_{70}C_2$ (every possible combination of $t_1$, $t_2$ in Heap 1)
       Compute rel_wt_double($t_1$, $t_2$) and freq_double($t_1$, $t_2$)
   END OF FOR
4. Select pairs by rel_wt_double and keep the best 70 pairs in
       Heap 2 with their freq_double($t_1$, $t_2$).
5. FOR $i$ = 1 to 70
       FOR $j$ =1 to 70
           compute  freq_triple($t_1$, $t_2$, $t_3$). and rel_wt_triple($t_1$, $t_2$, $t_3$).
       END OF FOR
   END OF FOR
6. Select triples by rel_wt_triple and keep the best 70 triples in Heap 3
       with their freq_triple.
7. Set an initial query, $q_o$ as disjuncts of terms in Heap 1 and estimate
       the size of retrieval with query $q$ (called estret($q$)).
8. WHILE (estret($q$) > U_num) DO
    8.1 Adjust the number of clauses in $q$ using terms in Heap 1, 2, and 3.
    8.2 Update estret with the adjusted $q$.
   END (WHILE)
```

(Fig. 4) Summarized procedure for the DNF method

defined as

$$rel\_wt(t) = [rel(t)/(R+qcount) - (freq(t)/N)] \times \ln\,[N/(freq(t)+10)] \qquad \text{(E-1)}$$

where	$rel(t)$	is the number of relevant documents containing the term t in D_I,
	R	is the total number of relevant documents D_I;
	$qcount$	is a controllable parameter which adjusts the occurrence characteristics of the query terms;
	$freq(t)$	is the frequency of the term t in D ;f
	N	is the total number of documents.

(Note: the definition of *rel_wt* is a simple version defined on [14]. But it has no effects on overall time complexity analysis.) The first part of the equation represents the proportion of relevant documents, in which term t occurs minus the proportion of all documents in the collection in which the term occurs. The right-hand side of the expression is an inverse document factor which implies the less term frequency in the database, the higher the term importance is.

There are two more functions similarly defined for term pairs and triples. The function *rel_wt_double*(t_1,

t_2) is the term importance for both term t_1 and t_2 together (i.e., t_1 and t_2); *rel_wt_triple*(t_1, t_2, t_3) is defined similarly defined for t_1 *and* t_2 *and* t_3. These functions are defined same as (E-1) except that *rel*(t) and *freq*(t) are replaced by *rel_double*(t_1, t_2) and *freq_double*(t_1, t_2) for *rel_wt_double* and by *rel_triple*(t_1, t_2, t_3), and *freq_triple*(t_1, t_2, t_3) for *rel_wt_triple*(t_1, t_2, t_3). The function $r(t_1, t_2)$ is the number of relevant documents retrieved with the terms t_1 and t_2 (i.e., t_1 *and* t_2), and *freq_double*(t_1, t_2) is the expected number of documents retrieved by using both terms t_1 and t_2 under the assumption of term independence. The triple functions *rel_triple*(t_1, t_2, t_3), and *freq_triple*(t_1, t_2, t_3) are similarly defined.

Step 0 (Fig. 4) prepares the input and collect index terms assigned in the input documents. In Step 1 and 2, the best 70 terms are selected from T_I by *rel_wt* and their frequencies (computed by *freqs*) are stored in *Heap* 1. Pairs and triples are collected from *Heap* 1 and kept in *Heap* 2 and *Heap* 3, respectively (Step 3 thru 6). An initial Boolean query q_o is formulated by connecting terms in *Heap* 1 with the operator or (i.e., t_1 or t_2 or t_3 or ... or t_{70}). The expected retrieval size by query q_o (denoted as *estret*(q)) is estimated using *freq*, that is,

$$estret(q_o) = freq(t_1) + freq(t_2) + \ldots + freq(t_{70}).$$

In Step 8, *estret* is compared with the predefined *U_num* (i.e., the user specified desired number of documents). If *estret*(q_o) > *U_num* (it will be always the case with a broad initial query), the term with the lowest *rel_wt*(t) in q_o will be dropped or substituted by the pairs and triples in *Heap* 2 and *Heap* 3. Then another *estret* is estimated for the updated query. This iteration continues until (*estret* < *U_num*) is satisfied. The Salton's algorithm presented here is a summarization. For detailed descriptions, refer to [14].

The distinguishing feature of the DNF method is the capability of controlling the number of documents retrieved. Once the user specifies the desired number of documents (*U_num* in Fig. 4), the algorithm can adjust the number of terms in the query to collect the right size. It is a desirable feature, since it is hard to control the number of document retrieved by a Boolean query.

4.1 Priori Analysis of the DNF method

The best case of the Salton's method occurs when the WHILE loop in Step 9 of (Fig. 4) is not executed, i.e., when the *estret*(q) is under the threshold value of *U_num*. In Step 0, the input preparation takes same as the query tree case. That is $n + mn$ steps. In Step 1, it takes $(3n+8)k$ steps to compute relevance weight for each index term in T_I. In Step 2, it takes k steps to select the single best term. Since we need to find the 70 best terms, the total number of steps is $70k$. To find *freq* for each 70 term, the total number of steps is $70h$ (h is the total number of documents in the database); In Step 3 it takes ${}_{70}C_2$ steps to prepare every possible pair terms and $(3n+8)$ steps to compute each *rel_wt_double*. For *freq_double* computations, it requires 2(70) steps. Therefore, the total is ${}_{70}C_2\,(3n+8) + 2(70)$ steps. As for the case in Step 2, Step 4 takes $(70)\,{}_{70}C_2$ steps to collect the best 70 pairs. In Step 5, it takes (70)(70) steps to prepare triple terms from *Heap* 1 and *Heap* 2 and $(3n+8)$ steps to compute each *rel_wt_triple*. For the *freq_triple* computations, it requires 3(70) steps; thus, the total number of steps is $(70)^2(3n+8) + 3(70)$; In Step 6, there are $(70)^2$ number of triples and to select the best 70 triples, it takes $(70)^3$ steps; Step 7 takes 1 step to estimate the size of the retrieved documents. As described above, Step 8 is skipped for the best case. Thus, the total number of executed statements (T_{total}) is

$$\begin{aligned} T_{total}(\text{DNF method}) &= n + n\cdot m + (3n+8)k + 70k + 70h \\ &\quad + {}_{70}C_2(3n+8) + 2(70) + (70)\,{}_{70}C_2 \\ &\quad + (70)^2(3n+8) + 3(70) + (70)^3 + 1 \\ &= n + n\cdot m + (3n+8)k + 70k + 70h \\ &\quad + {}_{70}C_2(3n) + (70)^2(3n) \\ &\quad + \text{large constant} \end{aligned}$$

Without loss of generality, we may assume $k<m$. From the analysis of the test sets (i.e., CACM, CISI, and Medlars), it revealed that the number of documents is less than the number of index terms in the database. Therefore, it is assumed that $h<m$.

$$T_{total}(\text{DNF method}) \leq n + n \cdot m + (3n+8)m + 70m$$
$$+70m + {}_{70}C_2(3n) + (70)^2(3n) + \text{large constant}$$

For some constant c_5,

$$T_{total}(\text{DNF method}) \leq c_5 n \cdot m$$
$$T_{best}(\text{DNF method}) = O(n \cdot m).$$

For the worst case (i.e., When Step 8 is executed), it is more difficult to count the number of times the statements in the WHILE loop are executed because the threshold value, *U_num* is dependent on the user. The worst case happens when the final query consists of all the triple terms. It would take k steps to find the lowest *rel_wt* and $70k$ steps to replace all the singles to pair terms. With the same method, it would take another $70k$ steps to replace all the pairs to the triples. Therefore, The worst case of the Salton's method is

$$T_{worst}(\text{DNF method}) = O(n \cdot m) + 70k + 70k$$
$$= O(n \cdot m)$$

5. Dillon's Method

Another automatic technique for query reformulation (call it Dillon's method) was developed by Dillon and his colleagues[3, 4]. Its procedure is very similar to the DNF method in that it also has the function similar to *rel_wt*. The differences are the ways which the terms are selected to formulate a query in disjunctive normal form. A summarized procedure is described in (Fig. 5).

```
0. Prepare input documents $D_I$ and collect
       $T_I = \{t \mid t = 1 \text{ and } t \in D_I\}$
1. Set floors for single (floor1) and pairs (floor2).
2. FOR (each $t$ in $T_I$)
       Compute prev(t).
   END OF FOR
3. FOR (each $t$ in $T_I$)
       Normalize prev(t)
   END OF FOR
4. FOR (each $t$ in $T_I$) DO
    4.1 If  (prev(t) > floor1) then keep t in Heap 1.
    4.2 If  (floor1 < prev(t) < floor2) then keep Temp_Heap
   END OF FOR
5. Keep every possible pairs in Heap 2 using terms in Temp_Heap
6. Formulate a query in DNF with singles and pairs in Heap 1 and Heap 2.
```

(Fig. 5) The summarized procedure for the dillon's method

While in the DNF method the numbers of singles, pairs, and triples of terms in a Boolean query are controlled by the user's requested number of documents (*U_num*), Dillon's method has instead pre-determined settings, called *floors*. The floor settings divide the search terms into groups and each group is determined to form a single, double or triple. As described in the Salton's algorithm, Step 0 prepares the inputs and collects their index terms. In Step 1, two floors are set : *floor*1 and *floor*2. In Step 2, the function called prevalence (*prev*) which measures the term importance is then computed for each term in the input documents. The function *prev* is based on two other measures: positive prevalence (*pos_prev*) and negative prevalence (*neg_prev*). They are defined as

pos_prev(t) = (# of relevant documents term t appears in) /(total # of documents judged as relevant)
neg_prev(t) = (# of nonrelevant documents term t appears in) /(total # of documents judged as nonrelevant)
prev(t) = *pos_prev*(t)-*neg_prev*(t)/log (*freq*(t)).

where *freq*(t) is the frequency of occurrence of t in the database. In Step 3, the function *prev*(t) for each term t is normalized by subtracting it from the mean and dividing it by the standard deviation (for details, see [4, 5]). The search terms are divided into two groups by the floor settings initialized in Step 1 : the terms with greater than *floor1* (call it Group 1) and those between floor1 and *floor2* (call it Group 2). If there are more than 50 terms in Group1, only the 50 best terms are kept in Heap 1 (Step 4.1). For the terms falling into Group 2 are then paired by the Boolean operator "and" in every possible combination and kept in *Heap* 2. The final query is formed by connecting the terms in *Heap* 1 and *Heap* 2 with the Boolean operator "or".

The major disadvantage of Dillon's method is its determination of the floor settings. There is no known procedure for determining the best cutting point for grouping single and double.

5.1 Priori Analysis of the Dillon's Method

The Dillon's method is rather straight forward than the query tree and the DNF method. In the query tree and the DNF case, there are two cases: the best and worst. In the Dillon's method, the steps in the algorithm are not dependent on the input documents that both of the cases are the same and a single time complexity is analyzed. In Step 0 (initialization), it takes n steps to prepare D_I and $m \cdot n$ step to collect T_I. To compute $prev(t)$ in Step 2, the steps required is $k(4n+5)$. The normalization of Step 3 takes $(k+1)$ steps to compute the average and $2k$ steps for the computation of its standard deviation. To assign terms in the proper groups, Step 4 takes k steps; In Step 5, to keep every possible pairs in *Heap* 2, it takes maximum ${}_{50}C_2$ steps. Finally, it takes 1 step to formulate a Boolean in Step 6.

Thus, the total number of executed statements (T_{total}) is

$$\begin{aligned} T_{total}(\text{Dillon's method}) &= n + n \cdot m + k(4n+5) + (k+1) \\ &\quad + 2k + k + {}_{50}C_2 + 1 \\ &= n + m \cdot n + 4k \cdot n + 9k + {}_{50}C_2 + 2 \end{aligned}$$

Without loss of gnerality, we may assume $k < m$.

$$T_{total}(\text{Dillon's method}) \leq n + m \cdot n + 4m \cdot n + 9m + {}_{50}C_2 + 2$$

For some constant c_6,

$$T_{total}(\text{Dillon's method}) \leq c_6 \cdot m \cdot n$$

$$T_{best}(\text{Dillon's method}) = O(n \cdot m).$$

6. Posteriori Testing and Conclusions

To investigate a real-time performance of an algorithm, the absolute times for each algorithm to formulate queries were measured in a Sun SparcStation 2 (with 64 MBytes of main memory and a 64 KBytes of cache), and the results are listed in ⟨Table 1⟩. Only the query formulation times (i.e., no document retrieval) were measured in seconds using the three test sets, CACM, CISI, and Medlars. In these experiments, the number of input documents are 3 relevant and 2 nonrelevant documents.

⟨Table 1⟩ Average query formulation time in seconds (the numbers in parentheses represents the number of queries used in a test set)

	Input documents: 3 relevant and 2 nonrelevant.			
	CACM (41)	CISI (73)	Medlars (30)	Average
DNF method	0.95	0.83	1.10	0.96
Dillon's method	0.46	0.44	0.57	0.49
Query Tree	0.24	0.23	0.27	0.25

As seen in ⟨Table 1⟩, every algorithm took more or less than 1 second in average to formulate a query. Without the query tree pruning procedure, the query tree algorithm takes much less time than the other methods do. The query tree is almost as four times faster than the DNF method and twice faster than the Dillon's method. The speed of the algorithms is in order of the query tree, the Dillon's method, and the DNF method.

A priori analysis summary of the three algorithms is listed in ⟨Table 2⟩. By just looking at the theoretical analysis, we may assume that the best case of the three algorithms are the same, and for the worst case, the query tree is the lowest. One of the pitfalls in theoretical analysis (i.e., priori analysis) is that it gives little weight to the constants. In the DNF case, there was a very large number of constant (i.e., $2(70) + {}_{70}C_2$

$70 + (70)^2(3n + 8) + 3(70) + (70)^3 + 1)$, but it is replaced by some large constant c_5. Although the constant c_6 in the Dillon's case is smaller than that of the DNF case, the situation is very similar. The real-time analysis in ⟨Table 1⟩ shows this situation very clearly. The number of input documents (i.e., n) is in general a small number. In these experiments, 5 documents are used; therefore, the best and worst cases of the query tree may be assumed to be the same.

⟨Table 2⟩ Posteriori analysis of the three algorithms (n: the number of input documents; m: the number of index terms in the database)

	Query Tree	DNF Method	Dillon's Method
Best-case	$O(m \cdot n)$	$O(m \cdot n)$	$O(m \cdot n)$
Worst-case	$O(m \cdot n^2)$	$O(m \cdot n)$	$O(m \cdot n)$

In conclusion, the posteriori testing revealed that the query tree is faster than any other algorithms while there is not much difference among algorithms in the priori analysis. A lesson to be learned here is that a theoretical analysis (i.e., the priori analysis) alone is a dangerous approach to analyze the performance of algorithms and a real-time analysis should accompany a theoretical analysis. In this paper we have discussed the query formulation time of an algorithm which is a part of an algorithm's efficiency. Another important aspect of an IR system's performance is retrieval effectiveness, which is usually measured by recall and precision. Since it is difficult to compare overall system's effectiveness using recall and precision because of an inverse relationship between them [13], E-measure [15], which is a composite measure of recall and precision was computed in these experiments. The best performance was also achieved by the query tree followed by DNF and Dillon's methods (for detailed analysis, please see [5, 6]). From our research, it is concluded that the performance (both efficiency and effectiveness) of query tree is superior than those of DNF and Dillon's methods.

References

[1] Breiman, L. et al, 'Classification And Regression Tree', Wadsworth Inc., 1984.

[2] Horowitz E., and Sahni S., 'Fundamentals of Computer Algorithms', Computer Science Press, 1978.

[3] Dillon, M. and Desper, J., "The Use of Automatic Relevance Feedback in Boolean Retrieval Systems," J. of Documentation, vol.36, no.3, pp. 197-208, 1980.

[4] Dillon, M., Ulmshneider, J. and Desper, J., "A Prevalence Formula for Automatic Relevance Feedback in Boolean Systems," Information Processing & Management, vol.19, no.1, pp. 27-36, 1983.

[5] Kim, N., 'A Classification Approach to the Automatic Reformulation of the Boolean Queries in Information Retrieval', Ph. D. Dissertation, University of Virginia, 1993.

[6] Kim, N., Brown, D. E., and French, J. C., "Boolean Query Reformulation with the Query Tree Classifier," the 4th ASIS SIG/IR, Columbus, Ohio, 1993.

[7] Salton, G., "A Simple Blueprint for Automatic Boolean Query Processing", Information Processing & Management, vol.24, No.3, pp. 269-280, 1988.

[8] Salton, G., 'Automatic Text Processing: the transformation, analysis, retrieval of computer', Addison-Wesley Co., 1988.

[9] Salton, G., "Development in Automatic Text Retrieval," Science, vol.253, pp. 974-980, 1991.

[10] Salton, G. and Buckley, C., "Improving Retrieval Performance by Relevance Feedback," J. of Amer. Soc. Info. Sci., vol.41, no.4, pp. 288-297, 1990.

[11] Salton, G., Buckley, C. and Fox, E. A., "Automatic Query Formulations in Information Retrieval," J. of Amer. Soc. Info. Sci., vol.34, no.4, pp. 262-280, 1983.

[12] Salton, G., Fox, E. A. and Voorhees E., "Advanced Feedback Methods in Information Retrieval," J. of Amer. Soc. Info. Sci., vol.36, no.3, pp. 200-210, 1985.

[13] Salton, G. and McGill, M. J., 'Introduction to Modern Information Retrieval', McGraw-Hill, Inc., 1983.

[14] Salton, G., Voorhees, E. and Fox, E. A. (1984), "A Comparison of Two Methods for Boolean Query Relevancy Feedback," Information Processing & Management, vol.20, no.5/6, pp. 637-651, 1984.

[15] Van Rijsbergen, 'Information Retrieval', 2nd Ed., Butterworths, London., 1979.

[16] Vernimb, C., "Automatic Query Adjustment in Document Retrieval," Information Processing & Management, vol.13, pp. 339-353, 1977.

김 남 호

1985년 University of Pittsburgh 전산학과(학사)
1986년 Rensselaer Polytechnic Institute 대학원 수학과(이학석사)
1989년 University of Virginia 대학원 시스템공학과(공학석사)
1993년 University of Virginia 대학원 시스템공학과(공학박사)
1993년~1994년 한국전산원 표준본부 선임연구원
1995년~현재 선문대학교 산업공학과 교수
관심분야: 시스템최적화, 정보검색, 기계학습

Donald E. Brown

1973 United States Military Academy, West Point, B.S.
1979 University of California, Berkeley, M.Eng. and M.S. in Operations Research
1985 University of Michigan, Ann Arbor, Ph.D. in Operations Research
1973~1982 Officer, U.S. Army, Honorable Discharge at the rank of Captain
1982~1984 Systems Analyst, Vector Research, Inc., Ann Arbor, Mi.
1985~1990 Assistant Professor of Systems Engineering, University of Virginia
1987~present Associate Director, Institute for Parallel Computation, University of Virginia
1990~1996 Associate Professor of Systems Engineering, University of Virginia
1996~present Professor and Chairman of Systems Engineering, University of Virginia.
Research: Data fusion, Forecasting and Prediction Theory, Data Analysis and Inductive Modeling.

James C. French

1973 University of Virginia, B.A., Mathematics.
1979 University of Virginia, M.S., Computer Science.
1982 University of Virginia. Ph.D., Computer Science.
1984~1987 Vice-President, Software Development, Information Research Corporation.
1987~1990 Senior Scientist, Institute for Parallel Computation University of Virginia.
1990~present Research Assistant Professor, Dept. of Computer Science University of Virginia.
Research: Information retrieval, distributed systems, scientific databases.