LIBERTAS MATHEMATICA,VOL.XI(1991)

Recursive Algorithm for Rules Evaluation in Expert Systems

Ilié Popescu

Keywords: Expert system, algorithm, graph, search space.

1. Introduction

This paper presents an efficient algorithm for rules evaluation into a knowledge base expert system in the case that the head of a rule is concluded if any k-subset of terms in its tail is simultaneously true. We first introduce what logic programming has to offer to expert systems and we show the need to perform efficient algorithms in both the search space and goal route approaches. Logic programming is appealing for stating problems because of its relational form and its nondeterminism. Its relational form makes it appropriate for stating problems. The nondeterminism is not only useful from a theoretical point of view but is also a conceptual device for designing expert systems. Essentially, nondeterminism enables the programmer to be liberated from tree-search programming. Logic programming provides a precise language for deducing consequences from premises, for studying the truth or falsity of statements given the truth or falsity of other statements. Both logic and programming require the explicit expression of one's knowledge and methods in an acceptable formalism. It has been widely accepted that a category of clauses relevant to expert system applications is the class of Horn clauses [3]. According to Kowalski [4], an axiom or rule is presented in the form of implication

(1) $\forall(x_1,...,x_k)\ P_0 \leftarrow P_1 \ \& \ ... \ \& \ P_n$

where $x_1,...,x_k$ are variables whose values we want to establish, $P_1,...,P_n$ are subgoals we are trying to delete, and the backward arrow $\leftarrow$ is used to denote logical implication. The left side P_0 is referred to as the *head* or *consequent* of the rule, while the right side is the *premise* or the *antecedent*. In addition to this formal presentation of a rule, it has a semantic interpretation which can be read as follows:

(2) *if* P_1 and ... and P_n *then* P_0

In a special case, if a rule contains no atomic formula in the right side (n=0), then it is regarded as an *assertion* or *fact*. Assertional knowledge includes factual or synthetic concepts, which may be true or false according to the domain taken into consideration, whereas descriptional knowledge expresses concepts of an analytic nature concerning the definition of the concepts in an existentional context and the logical relationships between such concepts. The set of assertions explicitly represented in the knowledge base of an expert system can be extended by defining new concepts, and formulating general rules which determine how these concepts can be derived from the base assertions and/or from other concepts previously deduced. The fact that general rules referred to the knowledge base of an expert system represent some deductive interpretations of knowledge, highly simplifies the task of goal formulation and transfers the burden of a goal route to the expert system. This approach has been developed in a number of works [4], [5], [8], [10], [11], [12], and according to them a goal is a well-formed formula of the first order language describing the knowledge base. An expert system may be represented by a relation $f: KB \times Q \rightarrow A$ where KB is the knowledge base, Q is the set of queries and A is a set of system responses. The knowledge base contains all the information required in order to obtain the solution: a fact base, which represents some factual interpretations and heuristics of the pertinent abstract knowledge, and a rules base which represents some deductive interpretations of the knowledge.

2. Background

This present section is devoted to the presentation of the well-known problem-reduction method [7] called *backward chaining* in the artificial intelligence world, which is the method used by PROLOG.

Let us consider a problem P_0 (goal) we wish to solve. As we have seen above, the representation of this problem as a logical implication is a convenient technique, thanks to its nondeterminism, relational form and declarative semantics. Basically, it consists of a recursive definition of a subset of trees. Unfortunately, this *if-then* formulation is too simple to make the search procedures efficient. Each set of interpretations of $P_1,\ldots,P_n$, constitutes in a way an enormous database. Of course, we cannot store this database in an explicit form. It is represented by a finite set of rules from which all the interpretations in the database can be deduced. Hence, we consider a goal as a logical implication

(3) $\exists(x_1,\ldots,x_k), (P_1 \wedge \ldots \wedge P_n), n \geq 1$

The variables that appear in the rule are quantified universally. In other words, each interpretation is obtained by giving to the variables in the rule (1) all the possible values and by transforming the terms into well-defined trees. A goal-reduction operator transforms a

goal description in a sequence of terms and a system of constraints. In the evaluation of the goal P_0 three cases are of particular interest.

First, if the sequence of terms is empty, the goal can be restated as follows: what are the assignments of the variables that transform the goal in an assertion?

Second, if there exists a substitution which transforms the terms $P_1,...,P_n$ in assertional concepts and/or in concepts previously deduced, then the variables assignments provide an answer to P_0.

Third, we suppose that a term P_i, $1 \le i \le n$, cannot be asserted by any substitution. Assume that there exists a rule R_i

(4) R_i: $\exists (x_i^1,...,x_i^p), (P_i^1 \ \& \ ... \ \& \ P_i^q), \{C_i\}$

which unifies with P_i and produces a new state of the goal. It is possible to progress to this new state only if each term is transformed into a well-formed tree. If necessary, some variables of the rule (4) are renamed so that none of them occur in the set $\{x_1,...,x_k\}$. This is a nondeterministic process and we can exhibit procedures, e.g. *generate and test*, *standard backtracking*, *forward checking*, *looking ahead* that generate the *solutions space* for P_0. Abstractly, these procedures generate all the sequences $(v_1,...,v_k)$ that satisfy the terms P_i, $1 \le i \le n$, where $v_1,...,v_k$ are assignments for the variables $x_1,...,x_k$, and P_0 holds. This approach consists fundamentally of verifying intermediate properties $P_i^j(v_1^j,...,v_j^j)$ such that $P_i^{j+1}(v_1^{j+1},...,v_j^{j+1},v_{j+1}^{j+1})$ implies $P_i^j(v_1^j,...,v_j^j)$ for $1 \le j \le k$. In other words, if $(v_1^j,...,v_j^j)$ does not satisfy P_i^j, then no extended sequence $(v_1^{j+1},...,v_j^{j+1},v_{j+1}^{j+1})$ can satisfy P_i^{j+1}; hence, by induction no extended sequence $(v_1,...,v_k)$ can satisfy the goal P_0. In their current state of development these procedures are very inefficient for executing the natural formulation of problems and their performances drastically decrease as the problem size grows.

3. The k-problem

In usual logic programming languages, all the terms $P_1,...,P_n$ have to be true simultaneously in order to conclude P_0. Our particular interest here is the case when we can conclude P_0 if any k-subset, $k < n$, of terms $\{P_1,...,P_n\}$ is true. (A simple example of such a case is the common cold, where there are about 12 observations and any 3 of them existing simultaneously establish the diagnosis). We express formally this problem as follows: given a list of terms $L = < P_1,...,P_n >$, find any k elements of L which are true simultaneously. In order to solve this problem we reexpress the problem in the following manner:

There exist exactly k elements true in the list of head h and tail T iff one of two following sentences is true:

i) h is true and it exists exactly k-1 elements true in the tail T;

ii) *h* is false and it exists exactly k elements true in the tail *T*.

The following program solves this problem such that the boolean value of the variable "b" is the logical value of the expression: "the list *L* contains exactly k elements true".

k-true program

```
AllTrue(k,L,b) :- AllFalse(L,b), {k=0}.
AllTrue(k,[ ],False) :- {k≠0}.
AllTrue(k,[h1:T],b) :- AllTrue(k-1,T,h2),{k≠0,b=h1 and h2}.
AllTrue(k,[h1:T],b) :- AllTrue(k,T,h2),{k≠0, b=not(h1) and h2}.

AllFalse([ ], True).
AllFalse([h1:T], not (h1) and h2) :- AllFalse(T,h2).
```

This program solves correctly the problem. The special case in which k=n defines the "normal" rule structure, that is, the case in which all the terms $P_1,...,P_n$ have to be satisfied to derive the conclusion P_0. The program achieves an *a posteriori* pruning in the search space. Given a sequence of terms $P_1,...,P_j$, $j < k$, the problem is to extend it by finding a set of values for the variables appearing in P_{j+1} such that all the constraints involving the variables from $P_1,...,P_j$ and a variable from P_{j+1} are satisfied. If there exist no such values, the sequence $P_1,...,P_j$ cannot be extended with P_{j+1}. The search is thus reduced *a posteriori* after the discovering of a failure. This *a posteriori* pruning can be grossly inefficient for many problems since the program develops a term which possibly has nothing to do with the current state of the problem. In our particular case even if the search space is a subspace of the global *solutions space*, we perceive that the number of trials to derive a conclusion is a combinatorial problem. Our particular concern is to minimize the number of trials and consequently to reduce the search space.

4. The k-algorithm

The *k-true* program makes an exhaustive enumeration of all solutions and the problem comes from the extremely large space that must be searched throughout. Here, we propose an efficient searching algorithm based on an intelligent implicit terms evaluation. This algorithm consists mainly of a *branching process*, which splits the list of terms into several subsets, and a *bound process*, which consists of searching into subsets terms which are true with respect to the problem P_0. We denote this algorithm as *k-algorithm*. The *k-algorithm* objective is to

avoid the explicit enumeration of the search space in the following way: as soon as the evaluations of any j terms, $0 \le j < k$, into a subset are true, the remaining set of terms is reduced to a *(k-j)-problem*. This algorithm can be specified in the following way:.

Step 1. Split the list L into $(n \text{ div } k)$ subsets; set S to the set $\{s_1,...,s_p\}$, $p = n \text{ div } k$, where s_l, $1 \le l \le p$ are subsets of k elements from L;

Step 2. Check a subset s_l; the output of the decision table is j;

Step 3. Remove s_l from S and L; (* *The k-algorithm* is now *(k-j)-algorithm* *);

Step 4. If $j=k$ then the algorithm terminates: solution found; otherwise go to step 5;

Step 5. If $j < k$ and $L <> 0$ then go to step 1 with $n:= n-lk$, and $k:=k-j$; otherwise the algorithm terminates: no solution.

This algorithm terminates when a solution has been found or when all the list of terms has been searched throughout. Contrary to the previous strategy, this approach curtails the search quite drastically and requires little memory space.

We now turn to the evaluation of our approach with respect to combinatorial problems. The strong point of our approach lies in the ability to reduce the *k-problem* to a *(k-j)-problem* since j terms are true. Indeed, if in the evaluation of a subset of terms s_l, $1 \le l \le n \bmod k$, the output of the decision table is j, $j < k$, then the total number of checks is $l * \sum_{i=0}^{k} \binom{k}{i}$. Now instead of restarting the problem, we can define a new *reduced* problem such that we check only the remaining $(n-lk)$ terms in the list L. Thus, the *k-algorithm* either derives the conclusion P_0 or reduces the problem to a smaller problem of the same nature. The new problem is solved recursively by our algorithm. The total number of trials in *k-algorithm* is

$$l*\sum_{i=0}^{k} \binom{k}{i} + [(n-lk) \text{ div } (k-j)] * \sum_{i=0}^{k-j} \binom{k-j}{i} = l*2^k + [(n-lk) \text{ div } (k-j)] * 2^{k-j}$$

Theorem: The total number of trials in *k-algorithm* is bounded above by $(n/k)*2^k$.

Proof. Firstly, we remark that the total number of trials in *k-algorithm* satisfies the inequality

(1) $l * 2^k + [(n-lk) \text{ div } (k-j)] * 2^{k-j} \le l * 2^k + \frac{n-lk}{k-j} * 2^{k-j}$ for any $0 \le j < k$, $1 \le l \le n \bmod k$

Let us consider the inequality

(2) $l * 2^k + \frac{n-lk}{k-j} * 2^{k-j} \leq \frac{n}{k} * 2^k,$

We show that this inequality is true for any j, $0 \leq j < k$.

By successive calculations we obtain

(2') $\frac{(lk-lj)2^j + (n-lk)}{(k-j)2^j} \leq \frac{n}{k}$, $0 \leq j < k$, $l \leq l < \frac{n}{k}$

(2'') $2^j(lk^2-lkj-nk+nj) \leq k(n-lk)$

(2''') $2^j[k(lk-n) -j(lk-n)] \leq -k(n-lk)$

(2'''') $2^j(lk-n)(k-j) \leq k(n-lk)$

Since $l < \frac{n}{k}$, this last inequality becomes

(3) $2^j(k-j) \geq k,$

which is true for any j and k, $0 \leq j < k$.

This theorem proves that the implementation of *k-algorithm* into an expert system reduces very strongly the search space. Let us make this more clear by taking a simple example. Consider a rule $P_0 \leftarrow P_1,...,P_n$, with n=20 and k=4. To solve P_0 using *k-algorithm* we perform the following:

(a) Pick the subset s_l and note by j the results of these 4 terms;

(b) The possible cases are as follows:

i. If j=4, meaning that all 4 terms are true, then P_0 is concluded; the algorithm terminates successfully;

ii. If j=3, then we check 1 more term of remaining 16. The problem P_0 is not solved, but instead of restarting the problem we define a new reduced one such that we check only the remaining terms;

iii. if j=2, then we perform a *2-algorithm* for the remaining 16 terms;

iv. If j=1, then we perform a *3-algorithm* for the remaining 16 terms;

v. If j=0, then we perform a *4-algorithm* for the remaining 16 terms.

We can underline several advantages by utilization of the *k-algorithm* over the well-known *k-true* program, for example: (1) there is guaranteed recursive reduction of the list of terms involved in a rule; (2) the algorithm is very general and can be applied to a great variety of expert systems; (3) the number of terms to be checked is very small compared to traditional

evaluation of a rule where the number of trials grows exponentially with the length of the set of terms (see - Table 1).

Rule	Using Combinations	Using k-algorithm
k=4; n=20	77520	80
k=4; n=36	942480	144
k=4; n=16	29120	64

Table 1. Number of trials in a rule evaluation

Having a good evaluation at our disposal reduces the search space to be explored by the subsequent recursive application. In our algorithm, the evaluation is quite simple but also very efficient. Indeed, as can be seen in the Figure 1 and in the Table 2 the number of trials in *k-algorithm* depends on the length k of the subset that establishes the problem P_0 and on the total number, n, of terms.

	Rules evaluation	
	Using Combinations	Using k-algorithm
k= 2	760	40
k= 4	77520	80
k= 6	2480640	213
k= 8	32248320	640
k=10	189190144	2048
k=12	515973120	6826
k=14	635043840	23405
k=16	317521920	81920
k=18	49807360	291271
k=20	1048576	1048576

Table 2. Number of trials for n=20 and k=2,4,...,20

If we choose to ignore anything else about rules evaluation, e.g. the range of variables, constraints satisfaction, domains specific information etc., we get what is referred in the literature to as "any k elements of a list". In this context, the *k-algorithm* is the most accurate. The price to pay for this accuracy consists of the assumption that:

1. The probability that a term check succeeds is independent of
   (i) the variables and the values given to the variables in a term,
   (ii) any past computation;
2. All the terms involved in a rule have the same probability to succeed.

Although this algorithm is very simple it is largely useful in building effective expert systems due to its great efficiency over the classical approaches.

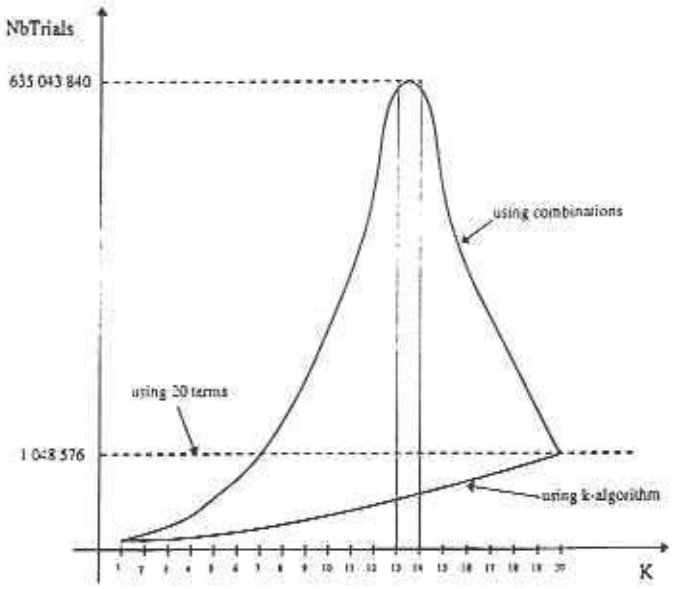

Figure 1. Space trials for n=20 and k=1,2,...,20

5. Conclusion

The *k-algorithm* manages an efficient search in the *solutions space* generated by an expert system. Its goal is to create a frame on a stack of terms which are deleted by a *branch* and *bound process*. The number of rules trials is a very good parameter for this purpose. The approach used by *k-algorithm* reduces recursively the search space, and for this reason produces substantial improvement in efficiency. The main idea behind this technique is an implicit terms evaluation in subsets of terms whose length is reduced recursively. Such handling of terms avoids much redundant work and useless generations: the failures are detected earlier and inference rules avoid bad backtracking choices.

6. Open questions

How can we use this algorithm to derive new algorithms by defining relations that should hold between the variables involved in rules? Initial investigation suggests that the use of constraints and consistency techniques fits perfectly inside this programming style.

How can we introduce in *k-algorithm* domains of variables? In usual logic languages the variables range over the Herbrand universe even when it is obvious that their real range is more restricted. From the user's point of view it allows to introduce the domains of concepts inside the logic programming, information that is hidden in actually programming methodologies. From a practical point of view it allows the addition of domain-specific knowledge and the introduction of new inference rules that embodies idea of *a priori* pruning.

7. BIBLIOGRAPHY

[1] Davis, R. and Lenat, D.B., Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill Inc., 1982.

[2] Farenny, H. and Ghallab, M., Éléments d'Intelligence Artificielle, Hermes-France 1987.

[3] Horn, A., On sentences which are true of direct unions of algebras, J. Symb. Logic 16 (1951), pp. 14-21.

[4] Kowalski, R., Logic for Problem Solving, North-Holland, 1979.

[5] Laurière, J.L., Intelligence Artificielle - résolution de problèmes par l'homme et par la machine - Eyrolles, 1986.

[6] Minker, J., Search strategy and selection function for an inferential relational system- ACM Transaction on Database Systems, Vol. 3, No 1, 1978, pp. 1-31.

[7] Nilson, N.J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, 1971.

[8] Pearl, J., Heuristic intelligent search strategies for computer problem solving, Addison-Wesley, 1984.

[9] Popescu, I., and Roventa, E., Some considerations about a mathematical model for representing data imperfection - Libertas Mathematica, Vol. VII, 1987, Texas, pp. 85-94.

[10] Popescu, I., A Dual Approach for representation and Utilization of Declarative Knowledge - Proceedings IASTED - Expert Systems, 1988, Los Angeles, California, pp. 23-27.

[11] Shapiro, E., Concurrent Prolog, Vol. 1 & 2, MIT Press, 1988.

[12] Warren, D.H.D., An Improved Prolog Implementation which Optimizes Tail Recursion. In Proceedings of Logic Programming Workshop, Debrecen, Hungary, July 1980.