A LINEAR CONTROL ALGORITHM
FOR A CLASS OF RULE-BASED SYSTEMS

MALIK GHALLAB AND GONZALO ESCALADA-IMAZ

This paper presents a new and efficient control algorithm for the class of
rule-based systems that rely on monotonic inference and on the (object
attribute value) or similar propositional formalisms. Production rules
considered are of the Horn-clause type or of some extended non-Horn
form. The proposed algorithm uses a top-down (backward chaining) strat-
egy and is able to solve the indirect-recursivity problem. Its worst-case
complexity is proved to be linearly bounded by the total number of
propositions. In each case the algorithm uses a minimal number of
propositions. The control algorithm is formally described, illustrated with
examples, proved, and analysed. Extensions to rules in non-Horn form and
to problems seeking "best" solutions are finally considered.

1. INTRODUCTION

Rule-based systems (RBS) are now commonly used, either on their own or as a
part of a frame-oriented or object-oriented environment. Control algorithms in
RBSs remain however quite crude: usually a priori bounded depth-first or back-
track search strategies relying on the assumption of a loop-free search space.
Because of the indirect-recursivity problem (e.g. as in the two rules $Q \land R \rightarrow P$,
$P \land S \rightarrow Q$), such an assumption is rarely met.

This paper presents an efficient control algorithm for RBSs that rely on
monotonic inference and on the (object attribute value) or a similar propositional
formalism. We will first consider standard RBSs where rules of the production
memory (PM), facts of the working memory (WM), and the given goal $Q$ (a
disjunction-conjunction formula) are propositional Horn clauses. The inference
problem of finding whether $Q$ is a logical consequence of PM and WM is of
polynomial complexity [9, 10].

Address correspondence to M. Ghallab, Laboratoire d'Automatique et d'Analyse des Systèmes,
C.N.R.S. 7, Av. du Colonel Roche, 31077 Toulouse, France.
Received June 1988; accepted March 1989.
Some linear algorithms for solving that problem have been proposed [2, 3, 6, 8, 11] ([3] describes also some extensions for non-Horn RBSs). All of them rely on a bottom-up (forward chaining) control strategy; hence they develop a larger search space than would be required by a top-down search focused on the explicitly given goal.

The only top-down algorithm published, Algorithm 3 of [2], is in fact erroneous [3, 13].

Other known algorithms [5, 12] either are unable to deal with the main difficulty of the problem, (that is, the indirect recursivity) or are inefficient. For example, popular strategies for the resolution principle are quadratic [1, 7].

The proposed control algorithm uses a top-down strategy and has linear complexity: it runs in time \( O(n) \), where \( n \) is the total number of proposition occurrences in \( WM, PM, \) and \( Q \). Due to the top-down strategy, a goal \( Q \) is proved or disproved with a reduced search space. This algorithm is asymptotically and practically efficient. Its implementation is straightforward. Many RBS applications, such as diagnosis expert systems, can take advantage of it. It should be of significant benefit to RBSs seeking not any proof of a goal \( Q \), but a best proof of \( Q \) (e.g. problems dealing with approximate reasoning [4]).

A simplified version of the algorithm is presented informally. Then a formal description of the complete algorithm is given. Its correctness and linearity are proved. Interesting generalizations are considered in the last section: to rules in non-Horn form and to best-proof searches.

2. GENERAL PRESENTATION

The proof of a goal can be represented as a search problem in an And/Or graph \( G \). As a running example, let us consider the set of rules:

\[
\begin{align*}
P_2 \land P_8 & \rightarrow P_1, \\
P_2 \land P_9 & \rightarrow P_3, \\
P_3 \land P_{14} & \rightarrow P_{13}, \\
P_{11} \land P_{12} & \rightarrow P_{10}. \\
P_9 \land P_{10} & \rightarrow P_{1}, \\
P_9 \land P_{10} & \rightarrow P_{1}, \\
P_5 \land P_3 & \land P_6 & \rightarrow P_2, \\
P_2 & \rightarrow P_4, \\
P_4 & \land P_{10} & \rightarrow P_{13}, \\
P_{15} & \land P_{10} & \rightarrow P_{17}, \\
P_7 & \land P_1 & \rightarrow P_4, \\
P_1 & \land P_3 & \rightarrow P_{13}, \\
P_7 & \land P_1 & \rightarrow P_{13}. 
\end{align*}
\]

The proof of \( Q = (P_1 \land P_{13}) \lor P_{17} \), with \( WM = \{P_5, P_7, P_9, P_{11}, P_{12}, P_{13}\} \), is a search in the graph shown in Figure 1. To \( Q \) and to each proposition \( P_i \) corresponds a single node in the And/Or graph \( G \). An element of WM is a leaf of \( G \) (underlined in the figure). A rule \( P_1 \land \cdots \land P_k \rightarrow P \) is a connector \( (P, (P_1, \ldots, P_k)) \) of \( G \) from node \( P \) to successor nodes \( P_{1}, \ldots, P_{k} \). A conjunction in the formula \( Q \) is also a connector of \( G \).

The following data structures will be used to search an And/Or graph:

1. To a node \( P \) corresponds:
   \( c(P) \), the set of connectors issued from \( P \).

2. To a connector \( C \) correspond:
   \( r(C) \), the node from which \( C \) is issued, and \( s(C) \), the set of successor nodes of \( r(C) \) along connector \( C \).
Definition. A subgraph $G(P)$ of an And/Or graph $G$ is said to be valid if either:

(a) node $P$ of $G$ belongs to WM: $G(P)$ contains solely the node $P$; or

(b) there exists in $G$ a connector $C \in e(P)$ such that for each $P_k \in s(C)$ there is a valid subgraph $G(P_k)$; in that case $G(P)$ contains only the node $P$, arcs of the connector $C$, and the nodes and arcs of the subgraphs $G(P_k)$ [$P$ has no other successors in $G(P)$ than nodes of $s(C)$].

Notice that $G(P)$ is a DAG rooted at $P$. There may be several valid subgraphs for a given node.

Proposition. $Q$ is a logical consequence from WM and PM (i.e., $Q$ can be proved from WM and PM) iff there exists a valid subgraph $G(Q)$ in $G$.

The proof of this proposition is very simple and is outlined in the appendix.

The proposed algorithm is based on recursive depth-first search of the And/Or graph. For each node $P$ visited, the set $e(P)$ of connectors outgoing from $P$ is sequentially examined (function Or-Expand) until one leads to a solution. Visiting a connector $C$ involves the expansion of all nonleaf successor nodes in $s(C)$ (function And-Expand) until one fails to lead to a solution.

Initially nodes in WM are marked solved, and all other nodes are marked open. A node $P$ being expanded is marked expanding until either:

- $P$ has a solved connector: one along which every successor node is solved; in that case $P$ is marked solved; or
- $P$ is shown to have no such connector: $P$ is marked failed.
Let us illustrate informally a simplified version of the algorithm. For loop-free And/Or graphs the algorithm is straightforward. There is a loop if during the expansion of a node $P$, one of its successors $P_i$ is found to be marked expanding. The sequence of nodes in the loop $[P_1, ..., P_k, P_{k+1}, ..., P_i]$ is a last-in first-out stack; $P_i$ will be called its source node.

A node $P_k$ is said to be confined on the loop if its connector on this loop is solved except for the successor node $P_{k+1}$ in the loop (marked expanding), and if $P_k$ has no other solved connector. If all nodes in the loop are confined, then all will be marked failed. If $P_k$ is confined and later on $P_i$ is found solved, then $P_k$ will also be marked solved. This marking of confined nodes as solved is propagated backward up to the source node.

In general a node $P$ can have more than one expanding successor: it may be on several overlapping loops. Then the algorithm backtracks, leaving $P$ expanding. The final marking of $P$ will depend on that of its expanding successors. Two data structures will be needed for that:

$\text{Followers}(P)$: nodes on which the marking of a confined node $P$ depends. arranged as a formula in disjunctive form, e.g. $\text{Followers}(P_4) = P_2 \lor P_1$; $P_4$ is solved iff either $P_2$ or $P_1$ is solved (Figure 1).

$\text{Leaders}(P')$: list of predecessor nodes of $P'$ that are confined on loops. e.g. $\text{Leaders}(P_2) = (P_3, P_4)$.

If $P'$ appears in $\text{Followers}(P)$ then $P$ belongs to $\text{Leaders}(P')$.

Let us illustrate the use of these structures on the running example (Figure 1). Node $Q$ and then successively $P_1, P_2, P_3$ are expanded. At this step $P_2$ is found expanding. Since $P_3$ is not yet proved to be confined on the loop, we proceed to the next recursion. Two successors of $P_4$ are expanding. $P_7$ is solved, and thus $P_4$ is confined. We set $\text{Followers}(P_2) = P_2 \lor P_1$. $\text{Leaders}(P_1) = \text{Leaders}(P_2) = (P_4)$, and backtrack. Node $P_3$ is now found to be confined: $\text{Followers}(P_3) = P_2 \lor P_7$. $\text{Leaders}(P_2) = P_3, P_4$, $\text{leaders}(P_4) = (P_3)$. Back to $P_2$, we find $P_6$ failed; thus $P_2$ is also marked failed. Leaders of $P_2$ are not examined, and nothing will happen to pending nodes.

We backtrack to $P_1$, skip $P_8$, and visit successively $P_9, P_{10}, P_{11}$, and $P_{12}$. $P_{10}$ and then $P_1$ are marked solved. Before backtracking to $Q$, node $P_4$ in $\text{Leaders}(P_4)$ is examined: $\text{Followers}(P_4) = (P_2 \lor P_7)$ leads us to mark $P_4$ solved. Thus $P_3$ in $\text{Leaders}(P_3)$ is also examined: $\text{Followers}(P_3) = P_2 \lor P_4$, so $P_3$ cannot be marked solved; it remains expanding. This ends the expansion of $P_3$. Since it is the last source node, all pending nodes that remain expanding (only $P_3$) are marked failed.

Finally node $P_{13}$ is expanded, its successor $P_3$ is failed. $P_{14}$ is skipped, and $P_4$ and $P_{10}$ are solved: $P_{13}$ is marked solved. A solution is found, and the rest of the graph is not examined.

To summarize, the algorithm proceeds as follows: if the source node $P_i$ of a loop is marked solved, then nodes in $\text{Leaders}(P_i)$ are sequentially visited. For every node $P_j$ in this list, the formula $\text{Followers}(P_j)$ is evaluated. If it leads to $P_j$ solved, then the list $\text{Leaders}(P_j)$ is in turn examined. Nodes that remain expanding after the end of the expansion of the last source node must be marked failed. For that, all nodes confined in a loop are put in a list called Pending.
3. FORMAL PRESENTATION

In the informal presentation given above two points need to be clarified:

1. how the last source node is computed, and
2. how \text{Followers}(P) is updated and evaluated.

The first question is easily solved if we use the natural order provided by the depth-first search of $G$: $\text{Order}(P) < \text{Order}(P')$ if $P$ was met by the search before $P'$. The only source node found in a loop that is of interest to us will be the smallest one according to this order.

Updating and evaluating a disjunctive formula are costly operations. Furthermore, keeping Followers($P$) as such a formula would lead to a nonlinear algorithm. That is because of the quadratic behavior on examples such as the one shown in Figure 2. Instead of using Followers($P$), we associate to each connector $C$ a counter \text{Counter}(C), which is initially set to the total number of arcs in $C$ found on a loop with the corresponding successors being confined. It is decremented for each successor node $P' \in s(C)$ that is marked \text{SOLVED}. Node $P$ will be marked \text{SOLVED} if one of its connectors in $c(P)$ is reset to zero. In this case, the same operations are propagated from $P$ backward to the connectors and nodes which were visited before $P$. For this purpose, \text{Leaders}(P') is now the list of connectors $C$ containing an arc $(P, P')$ on a loop. Hence, instead of evaluating a disjunctive formula when a node is \text{SOLVED}, we only need to decrement and test for zero a counter. This requires a constant time for each arc on loops.

For instance, in the running example (Figure 1) we have

\text{Counter}(C_{3-2}) = 2, \quad \text{Counter}(C_{4-2}) = 1, \quad \text{with} \quad r(C_{3-2}) = P_3,
\text{Counter}(C_{4-2}) = P_4,
\text{Leaders}(P_2) = (C_{3-2}, C_{4-2}).

In addition to $c(P)$, \text{Order}(P), \text{Leaders}(P), s(C),$ and \text{Counter}(C), the algorithm uses the following data structures:

\text{m}(P): the marker of a node $\in \{\text{OPEN, SOLVED, FAILED, EXPANDING}\}$;

\text{Pending}: a list of pending nodes confined on loops;
Source: the current last (smallest) source node;

N: the highest order of a node expanded.

Initially c(P) and s(C) are set according to the input data. The backward pointer r(C) is not always necessary: it is initialized to nil. Counter(C) and Leaders(P) are also set to nil; m(P), to open except for nodes in WM initialized solved. Pending and Source are set to nil, and N to 0.

The dot is used as a concatenation operator for lists:

**Or-Expand** (P: node)
Let Confined ← nil
m(P) ← EXPANDING
Order(P) ← N ← N + 1
For each C ∈ c(P) and while m(P) = EXPANDING do:
Let And-Result ← And-Expand(C)
If And-Result = SOLVED then m(P) ← SOLVED
Else If And-Result ≠ FAILED then
Confined ← Confined . (C, And-Result)
Endfor
If m(P) = SOLVED then If Leaders(P) ≠ nil then Propagate(P)
Else If Confined = nil then m(P) ← FAILED
Else do:
Pending ← P. Pending
For each (C, And-Result) ∈ Confined do:
Counter(C) ← |And-Result|
r(C) ← P
For each P′ ∈ And-Result do:
Leaders(P′) ← C . Leaders(P′)
If [Source = nil or Order(P′) < Order(Source)]
then Source ← P′
Endfor
Endfor
If Source = P then Expunge
End

**And-Expand** (C: connector): (list of nodes) ∪ {solved, failed}
Let Inloop ← nil
For each node P′ ∈ s(C) such that m(P′) ≠ solved do:
If m(P′) = open then If c(P′) = nil then Return(FAILED)
Else Or-Expand(P′)
If m(P′) = EXPANDING then Inloop ← P′ . Inloop
If m(P′) = FAILED then Return(FAILED)
Endor
If Inloop = nil then Return(SOLVED) else Return(Inloop)
End
If the local variable Confined ≠ nil and $m(P)$ remains EXPANDING after the For loop, then the corresponding node is confined according to our definition. In that case Confined is a list of pairs (connector, list of successor nodes in loops). This latter list is built in the local variable Inloop, and eventually returned by And-Expand. The two procedures Propagate and Expunge are straightforward:

**Propagate** ($P$: node)

  For each $C \in \text{Leaders}(P)$ do:
    Let $P' \leftarrow r(C)$
    If $m(P') = \text{EXPANDING}$ then do:
      decrement $\text{Counter}(C)$
      If $\text{Counter}(C) = 0$ then do:
        $m(P') \leftarrow \text{SOLVED}$
        If $\text{Leaders}(P') \neq \text{nil}$ then Propagate($P'$)
  End

**Expunge**

  For each $P \in \text{Pending}$ such that $m(P) = \text{EXPANDING}$ do $m(P) \leftarrow \text{FAILED}$
  Source $\leftarrow \text{nil}$
  Pending $\leftarrow \text{nil}$
  End

*Theorem (Correctness of the algorithm).* An And / Or graph $G$ has a valid subgraph $G(Q)$ iff Or-Expand($Q$) returns $m(Q) = \text{SOLVED}$.

The proof is given in the appendix.

4. ANALYSIS OF THE ALGORITHM

The algorithm take as input WM, PM, and $Q$, each proposition being uniquely indexed. Let $n$ be the total number of proposition occurrences in PM, WM, and $Q$. The definition of the And / Or graph $G$ is a simple initialization phase for indexing the various rules and propositions. The size (number of arcs and nodes) of $G$ being proportional to $n$, initialization is done in $O(n)$. Most of the preprocessing may be carried out only once for a given PM.

Procedure Or-Expand is executed at most once for each proposition $P$: a call to Or-Expand($P$) may happen only in And-Expand when $m(P) = \text{OPEN}$; $P$ is then marked EXPANDING at the beginning of Or-Expand, and it remains marked as such until its final marking SOLVED or FAILED.

The number of iterations of the first For loop in Or-Expand($P$) is bounded by the number of connectors in $c(P)$. Taking into account And-Expand($C$), which requires a constant time to process each node in $s(C)$, the total complexity of this For loop is thus proportional to the number of arcs issuing from $P$. This number of arcs is also an upper bound to the total number of nodes in Confined (since it contains only some successors of $P$), and hence to the number of steps in the two
following For loops. In total, the work performed in \textbf{Or-Expand} and \textbf{And-Expand} is linearly bounded by \(n\).

The function \textbf{Propagate}(\(P\)) is called at most once for a given node: when \(m(P)\) changes from \texttt{EXPANDING} to \texttt{SOLVED}. There, only arcs leading to \(P\) that are on a loop may be traversed, each at most once. Consequently, the global number of iterations of \textbf{Propagate} is also linearly bounded by \(n\).

Finally, notice that a node \(P\) may be added, during its expansion, at most once in the list \texttt{Pending}; procedure \textbf{Expunge} performs in total a linear number of steps. This achieves the proof that the proposed algorithm runs in \(O(n)\).

5. EXTENSIONS

As mentioned earlier, the proposed algorithm supports several extensions of practical interest. Let us consider some of them briefly. They will be presented informally on the basis of an example, specifying only the essential modifications with respect to the previous algorithm.

5.1. Closed-World Assumption

Negation is often dealt with using the so-called \textit{closed-world assumption}. Rules are not Horn clauses any more: their antecedent parts are conjunctions of propositions \(P\) and negated propositions \(\neg P'\); their consequents are still single positive propositions. A rule is valid if each proposition \(P (\neg P')\) in its antecedent is (is not) a logical consequence of WM and PM.

Arcs in the corresponding And/Or graph will be labeled: "+" for a non-negated proposition, "−" for a negated proposition. Under this assumption it is easily proved that a connector is solved if each "+" arc leads to a \texttt{SOLVED} node and each "−" arc to a \texttt{FAILED} node. Hence, before marking a node \(P_k\) we will test each successor node \(P_{k+1}\) along a connector \(C\) together with the sign labeling the arc \((P_k, P_{k+1})\).

Loop processing must be slightly modified. Instead of Leaders, two equivalent and separate lists \(+\text{Leaders}(P)\) and \(−\text{Leaders}(P)\), for "+" and "−" arcs respectively, are needed. Nodes in \texttt{Pending} are ordered now as a FIFO stack: the first node found confined must be the first one considered in the final marking.

The backward propagation through "+" arcs is done as before by \textbf{Propagate}, using only \(+\text{Leaders}\). The structure \(−\text{Leaders}(P)\) is not processed until the final marking, after the expansion of the smallest source node. It is used by \textbf{Expunge} for marking nodes in \texttt{Pending} \texttt{FAILED}. If a node \(P'\) is still \texttt{EXPANDING}, then it is marked \texttt{FAILED} (see remark in the end of this section), and the counters in \(−\text{Leaders}(P')\), i.e. the counters associated to each connector \(C\) containing a "−" arc \((P', P)\), are decremented. When such a counter is reset to zero, \(P\) will be marked \texttt{SOLVED} and this change will be propagated through \(+\text{Leaders}(P)\) by \textbf{Propagate}.

The functions \textbf{Or-Expand}, \textbf{And-Expand}, and \textbf{Propagate} are the same up to the small variations indicated; the function \textbf{Expunge} is the following:
FIGURE 3.

Expunge

While \textbf{Pending} \neq \text{nil} do:

\[ P \leftarrow \text{Pop(Pending)} \]

If \( m(P) = \text{EXPANDING} \) then do:

\[ m(P) \leftarrow \text{FAILED} \]

For each \( C \in - \text{Leaders}(P) \) do:

Let \( \text{Father} \leftarrow r(C) \)

If \( m(\text{Father}) = \text{EXPANDING} \) do:

Decrement \( \text{Counter}(C) \)

If \( \text{Counter}(C) = 0 \) do:

\[ m(\text{Father}) \leftarrow \text{SOLVED} \]

If \( + \text{Leaders}(\text{Father}) \neq \text{nil} \) then \( \text{Propagate(\text{Father})} \)

Endwhile

End

Let us illustrate the algorithm on the example of Figure 3. Node \( Q \) and then \( P_1 \) are expanded. The first connector outgoing from \( P_1 \) is \text{FAILED}. \( P_4 \) is expanded and then marked \text{FAILED} because \((P_4, P_6)\) is a "+" arc and \( P_6 \) is \text{FAILED}. Therefore \( P_1 \) will be \text{SOLVED}, since \((P_1, P_4)\) and \((P_1, P_7)\) are labeled with "-" and \( P_4 \) and \( P_7 \) are \text{FAILED}.

After \( P_1 \), nodes \( P_8 \), \( P_{10} \), \( P_{12} \), and \( P_{13} \) are expanded [arcs \((P_8, P_9),(P_{10}, P_{11}),(P_{13}, P_{14})\) were solved] and a loop is detected. Hence nodes \( P_{13} \), \( P_{12} \), \( P_{10} \), and \( P_8 \) are successively put in the FIFO stack \text{Pending}, and the search continues on the other connector outgoing from \( P_8 \). Because this connector is \text{SOLVED}, \text{Propagate}(P_8) \) is executed. This leads to \( P_{13} \) \text{SOLVED}. The backward propagation is stopped because \( + \text{Leaders}(P_{13}) = \text{nil} \).

The last source node is \( P_8 \); we proceed with the final marking of \text{Pending} = \{\( P_{13}, P_{12}, P_{10}, P_8 \)\}. The first node \( P_{13} \) is \text{SOLVED}, so "-" arc \((P_{12}, P_{13})\) is not processed. Next, node \( P_{12} \) is \text{EXPANDING}; after marking it \text{FAILED}, \(- \text{Leaders}(P_{12}) = \{C_{10-12}\} \text{ and } \text{Counter}(C_{10-12}) = 1 \) are processed. This leads to \( P_{10} \) \text{SOLVED}.
backward propagation Propagate($P_{10}$) passes through "+" arc ($P_8, P_{10}$), but $P_8$ is not visited because its marking is already solved.

Finally, returning to $Q$, its last arc ($Q, P_{12}$) is solved because it is labeled with "−" and $P_{12}$ is failed. Therefore $Q$ is solved.

The complexity of the algorithm remains linear. Only the final marking (Expunge) has been significantly modified. Procedure Propagate processes a node at most once: when its marking changes from expanding to solved. In this case, "+" arcs incoming to the new node solved will be processed. Similarly, in the final marking of Pending nodes by Expunge, only the "−" arcs incoming to these nodes are computed. Therefore the number of operations performed by Propagate (by Expunge) is limited by the number of "+" ("−") arcs on loops.

Remark. It is known that the completion of a theory under closed-world assumption is not necessarily consistent. A proposition that does not follow from PM and WM, assumed false under this assumption, may later on be deduced true. The algorithm can be adapted to detect such inconsistencies and to solve them by some particular decision criterion.

5.2. Factorized Rules

The proposed algorithm can be extended to rules with conjunctive normal-form antecedent and conjunctive consequent, e.g. ($P_1 \lor P_4) \land (P_5 \lor P_6) \land P_7 \land P_2$. Each such a rule is equivalent to a set of rules in Horn form (eight in the preceding case).

Notice that, due to the distributivity of $\lor$/$\land$, the transformation of a PM with factorized rules to a PM with simple Horn rules may augment exponentially the size of PM. Therefore, an algorithm able to compute in linear time a RBS with factorized rules may perform a total number of operations exponentially smaller than that of an algorithm treating only simple rules.

Here two types of nodes must be considered: proposition nodes, and rule nodes to which connectors are attached. An arc is a disjunction of propositions (for example $P_3 \lor P_4$ in the preceding case). An arc is solved if one of the propositions in the corresponding disjunction is marked solved. A connector is solved if every arc in it is solved.

To search the graph a data structure for rule nodes, similar to that used for propositional nodes, must be employed:

- $m(C)$ contains the marking of the rule node $C$, preventing more than one expansion of the same connector; and
- $s(C)$ contains the set of arcs (disjunctions) forming the antecedent of a particular rule [$e(P)$ is now a list of pointers to every rule node in the conclusion of which $P$ appears].

The data structures and the functions used for processing loops are slightly modified:

Order and Source are used indifferently for rule and proposition nodes. Cycles may be detected with $m(P) = \text{Expanding}$ as well as with $m(C) = \text{Expanding}$. Confined propositions and rules can be put in the same list Pending and processed as in the standard case.
\( r(C) \) is now the list of propositions in the consequent of \( C \) that are on loops.

Note that \( r(C) \) is for connectors \( C \) equivalent to \( \text{Leaders}(P) \) for propositions \( P \).

The backward propagation is based on the following principle: when a source node \( P \) is marked \text{SOLVED}, the set of counters \( \text{Counter}(C) \) such that \( C \) is in \( r(P) \) is decremented. If one of them is reset to zero, \( m(C) \) and the nodes in \( r(C) \) are marked \text{SOLVED} (if they are \text{EXPANDING}), and the backward propagation is carried on. The principle is similar when the source node is a rule node \( C \).

However, a problem arises if the standard function \( \text{Propagate} \) is used. Several backward propagations might be carried out on the same arc: one per node in the disjunction of the arc whose marking has just changed from \text{EXPANDING} to \text{SOLVED}. In order to avoid redundant backward propagations on the same arc, a flag, denoted \( \text{First}(arc) \), is put on each arc. If no backward propagation has been performed through an arc, its flag \( \text{First}(arc) \) points to the connector \( C \); it is set to nil after the first backward propagation. Thus \( \text{Leaders}(P) \) is a list of pointers to arcs on loops, instead of pointers to the connectors \( C \) that contains the arcs.

The functions \( \text{And-Expand}, \text{Or-Expand}, \) and \( \text{Expunge} \) are very similar to the standard case; the function \( \text{Propagate} \) is as follows:

\[
\text{Propagate}(CP)
\]

If \( CP \) is a proposition node then do:

For each arc \( \in \text{Leaders}(CP) \) do:

If \( \text{First}(arc) \neq \text{nil} \) do:

Let \( C \leftarrow \text{First}(arc) \)

\( \text{First}(arc) \leftarrow \text{nil} \)

Decrement \( \text{Counter}(C) \)

If \( \text{Counter}(C) = 0 \) then do:

\( m(C) \leftarrow \text{SOLVED} \)

If \( r(C) \neq \text{nil} \) then \( \text{Propagate}(C) \)

Else do:

;;; \( CP \) is a connector node

For each \( P \in r(CP) \) do:

If \( m(P) = \text{EXPANDING} \) then do:

\( m(P) \leftarrow \text{SOLVED} \)

If \( \text{Leaders}(P) \neq \text{nil} \) then \( \text{Propagate}(P) \)

End

Let us illustrate the algorithm on the example of Figure 4 (where few rule nodes are labeled). Node \( Q \) and then \( P_2 \) are expanded. Since the first rule node outgoing from \( P_2 \) is \text{FAILED}, the second rule node is expanded. Its first arc is a disjunction of literals \( P_5 \) and \( P_6 \). \( P_5 \) is \text{FAILED}, so \( P_6 \) will be expanded. The only connector outgoing from \( P_6 \) is solved, and so is \( P_6 \). Hence, the first arc in connector \( C_o \) is \text{SOLVED}. Its expansion continues; it is found \text{SOLVED}, since \( P_7 \) is \text{SOLVED}. Equally \( P_2 \) is marked \text{SOLVED}. Thus the next node, \( P_8 \), is immediately marked \text{SOLVED} because \( m(C_o) \) is \text{SOLVED}.

Node \( P_9 \) and successively \( (P_{24}, P_{10}, C_1, P_{12}, C_2, P_{15}, C_4 \) are expanded \( (P_{11} \) and \( P_{14} \) are \text{SOLVED}), and a loop is detected. Hence \( C_4, P_{15}, C_2, \) and \( P_{12} \) will be confined. The processing goes on for \( P_{13} \), and then \( C_3 \) and \( P_{13} \) are confined and thus put in \text{Pending}. Similarly for the nodes \( C_1, P_{10}, \) and \( P_{24} \). Back to \( P_9 \); its second successor rule node is \text{SOLVED}, and so \( P_9 \) is marked \text{SOLVED}.
The backward propagation from $P_9$ decrements and sets to zero the counters $\text{Counter}(C_4)$ and $\text{Counter}(C_2)$. Hence $C_4$, $P_{15}$, $C_2$, and $P_{12}$ are marked $\text{SOLVED}$. Continuing the backward propagation, $\text{Leaders}(P_{12})$ contains now the pointer to flag $\text{First}(P_{12}-P_{13})$, which in turn points to a counter $\text{Counter}(C_1)$; this is because it is the first backward propagation through this arc. This counter is decremented, and that is the only action, since the connector outgoing from $C_{12}$ had two arcs in loops. Back to $P_{15}$, the backward propagation is continued for $C_3$ and $P_{13}$, which will be $\text{SOLVED}$ too. But in this case the backward propagation is stopped because $\text{First}(P_{12}-P_{13})$ has been set to nil. Thus, the backward propagation is finished and $\text{Expunge}$ marks $C_4$, $P_{10}$, and $P_{24}$ $\text{FAILED}$.

Finally $P_{25}$ is expanded, $P_{10}$ is $\text{FAILED}$, and because the second rule node outgoing from $P_{24}$ is $\text{SOLVED}$, this node and $Q$ will be $\text{SOLVED}$ too.

While taking advantage of the factorized form of rules, the total number of operations remains linear. Predecessor proposition nodes of a rule node can be marked $\text{SOLVED}$ by the expansion of their common connector. In contrast, each connector $C$ is expanded at most once: $m(C)$ prevents further expansion of a connector $C$. On the other hand, nodes of the disjunction of an arc will be visited only while a node is not marked $\text{SOLVED}$.

As in the standard case, loop processing is also bounded by the number of nodes and arcs in loops: a node is visited only when it is marked $\text{SOLVED}$, and then arcs incoming to it will be traversed. Each arc is processed at most once, even if its associated disjunction contains more than one literal: $\text{First}(\text{arc})$ enables only one backward propagation through the same arc.
5.3. Uncertainty Factors

Here the idea is to find a "best" solution graph according to some associative criterion that relies on some parameters attached to rules in PM and to propositions in WM. Approximate reasoning in RBS provides an interesting example: uncertainty factors are attached to propositions and rules; a MYCIN-like propagation of uncertainty or some more theoretically sound propagation mechanism [4] can be used. In order to find the least uncertain proof for \( P \), the expansion of node \( P \) should proceed, even after a first connector is found solved, to the other connectors. The best current uncertainty factor for \( P \) is taken as a pruning value for the expansion of the remaining connectors. This almost exhaustive search should be improved by further pruning based on any available heuristic information. Its complexity remains in any case linearly bounded by the dimension of the And/Or graph.

6. CONCLUSION

A large class of rule-based systems rely on the propositional formalism considered in this paper. Known control algorithms for such knowledge representation use either

- a bottom-up strategy (forward chaining): they do not focus the search on the explicitly given goal, and hence they tend to be inefficient; or
- a top-down strategy (backward chaining): they are able to deal with loop-free search spaces.

The control algorithm proposed here has some interesting properties. It takes advantage of a focused top-down strategy with efficient loop processing and therefore reduces the total number of propositions processed. Its worst-case complexity is strictly linear. It supports some important extensions which also run in linear time. Furthermore it uses simple data and control structure and enables a straightforward implementation. This algorithm should be of significant benefit to a large class of RBS applications.

Another class of applications may be considered if one is able to lift the principles of the proposed algorithm from the propositional level to the first-order level in order to build, for instance, an efficient PROLOG interpreter.

APPENDIX

*Proposition.* \( Q \) is a logical consequence of WM and PM iff \( G \) contains a valid subgraph \( G(Q) \).

*Proof.* Let us define \( \text{Rank}(P) \) recursively as follows:

- If \( P \in \text{WM} \) then \( \text{Rank}(P) = 0 \).
- If there exists a connector \((P, (P_1, \ldots, P_k))\) such that for \( 1 \leq j \leq k \), \( \text{Rank}(P_j) \) is defined, then \( \text{Rank}(P) = \max\{\text{Rank}(P_j): 1 \leq j \leq k\} + 1 \).
The proof of the proposition is straightforward by recurrence over \( \text{Rank}(P) \).

**Theorem (Correctness of the algorithm).** An And/Or graph \( G \) has a valid subgraph \( G(Q) \) iff \( \text{Or-Expand}(Q) \) marks \( m(Q) = \text{SOLVED} \).

This theorem is decomposed into three lemmas.

**Lemma 1.** At the end of the expansion of a node \( P \):

1. If \( m(P) = \text{SOLVED} \), then there exists a valid subgraph \( G(P) \).
2. If \( m(P) = \text{FAILED} \), then there does not exist a valid subgraph \( G(P) \).
3. If \( m(P) = \text{EXPANDING} \), then there exist \( (C_1, \ldots, C_n) \) such that \( r(C_i) = P \) and \( \text{Counter}(C_i) = n_i \), where \( n_i \) is the number of successors \( P_{ij} \in s(C_i) \) that are EXPANDING and for each \( P_{ij}, 1 \leq i \leq n, 1 \leq j \leq n_i, C_i \in \text{Leaders}(P_{ij}) \).

**Proof.** Let \( \text{height}(P) \) be the maximal length of a path \([P, P_1, \ldots, P_r, \ldots, P_k, P_{k+1}]\) in \( G \) issuing from \( P \), such that \( P_1, P_2, \ldots, P_k \) are distinct nodes and \( P_{k+1} \) is either a leaf of \( G \) or the first node that loops back in the path (i.e., there is a node \( P_i \) such that \( P_{i+1} = P_i \)). The proof is by recurrence on \( \text{height}(P) = k \).

The lemma is obviously true for \( k = 0 \): \( P \) is marked FAILED if \( \epsilon(P) = \emptyset \); otherwise \( P \) remains SOLVED independently of \( \text{height}(P) \) if \( P \in \text{WM} \), by initialization.

Let us take a node \( P \) whose successors \( P' \) are of \( \text{height}(P) \leq k \) and for which the lemma is assumed to be true. To each path issuing from \( P' \) of length \( k \) and containing \( P \), i.e., \([P', P_1, \ldots, P_r, \ldots, P_{k-1}]\), corresponds a path \([P, P_{i+1}, \ldots, P_k]\) issuing from \( P \) of length \( r < k \). To each path issuing from \( P' \) of length \( k \) and not containing \( P \), i.e., \([P', P_1, \ldots, P_{k-1}]\), corresponds a path \([P, P', \ldots, P_k] \) issuing from \( P \) of length \( k + 1 \). Hence, either \( \text{height}(P) < k \) or \( \text{height}(P) = k + 1 \).

If \( \text{height}(P) < k \) then the lemma is verified by induction. Let us consider \( \text{height}(P) = k + 1 \).

If \( \text{Or-Expand}(P) \) marks \( m(P) = \text{SOLVED} \), then there is necessarily \( C \in \epsilon(P) \) such that \( \text{And-Expand}(P) \) returns SOLVED. This happens only if each successor \( P' \in s(C) \) has been found SOLVED. Since \( \text{height}(P') \leq k \), by induction there is a valid subgraph \( G(P) \).

Point (2) of the lemma, which is the dual case of point (1), is verified in a similar way.

Whenever \( m(P) \) is EXPANDING after the For loop of \( \text{Or-Expand} \), the block of lines following “Else-do” is executed. This and the induction hypothesis prove point (3) of the lemma.

**Lemma 2.** If \( \text{Or-Expand}(P) \) marks \( m(P) = \text{EXPANDING} \) and later on \( m(P) \) is changed to SOLVED, then there exists a valid subgraph \( G(P) \).

**Proof.** The proof is based on the following proposition: At any moment of the processing and for each \( C \) such that \( \text{Counter}(C) = n, C \) has exactly \( n \) successors \( P' \) EXPANDING (for which the algorithm has not found a valid subgraph). If this proposition holds and if \( \text{Counter}(C) \) becomes 0, then there exists a valid subgraph for \( r(C) \).
The proposition holds after the expansion of \( P \) by the preceding lemma. Later on, the counters are only decremented: Propagate\((P)\) decrements the counters in Leaders\((P)\) and marks \( r(C) \) SOLVED only if Counter\((C)\) is set to 0 and \( r(C) \) is EXPANDING. Thus the proposition follows from the fact that Propagate is called only:

(i) at the end of Or-Expand\((P')\), if \( P' \) is marked SOLVED and so \( G(P') \) does exist; or

(ii) during a backward propagation by changing \( m(P') \) from EXPANDING to SOLVED. Here again a recurrence on the calls to propagate [noticing that on the first call only case (i) applies] shows that there is a valid subgraph \( G(P') \).

\[\square\]

**Lemma 3.** If Or-Expand\((P)\) marks \( m(P) = \text{EXPANDING} \) and later on \( m(P) \) is changed to FAILED, then there does not exist a valid subgraph \( G(P) \).

**Proof.** When the Source variable is the node \( P \) whose expansion has just been completed, then nodes \( P' \) in Pending that are marked EXPANDING are changed to FAILED.

Let us first prove, by recurrence over Order\((P)\), the following proposition: Source is the last descendant of a node in Pending that will be completely expanded and whose marking (to SOLVED) could modify (to SOLVED) those of nodes in Pending. The proposition is trivially verified at the beginning of the processing because Pending = \( \emptyset \).

Let us assume the proposition to be true after the development of nodes of higher order than \( k \), and let us consider a node \( P \) with Order\((P) = k \).

**Case 1.** If \( P \) is marked SOLVED or FAILED, Pending and Source are not modified, and so the proposition remains true.

**Case 2.** If \( P \) is EXPANDING, then it is included in Pending, and so we must prove that the new Source obtained verifies the proposition for the set Pending \( \cup \{P\} \). By Lemma 1 it is clear that each connector \( C \) issuing from \( P \) with a successor \( P' \) marked FAILED cannot form a valid subgraph \( G(P) \), i.e., successor nodes along \( C \) cannot lead to \( P \) SOLVED. So the search of the new Source can be limited to nodes EXPANDING along a connector \( C \) that contains only nodes EXPANDING and SOLVED. But we remark that in this case a pair \((C, \text{Inloop})\), where Inloop contains successor nodes EXPANDING along \( C \), is put into Confined. Thus, by induction and taking into account that the first node developed is the last completely expanded, the new node Source \( P' \) is such that

\[\text{Order}(P') = \min \{\text{Order}(P''): P'' = \text{Source} \text{ or } P'' \in \text{Inloop} \text{ for } (C, \text{Inloop}) \in \text{Confined}\}.\]

From this proposition it is easy to prove that every node descendant of a node \( P \) in Pending has been already completely expanded, and thus, nodes in Pending that remain EXPANDING do have not a valid subgraph \( G(P) \) and so can be marked FAILED. This assertion can be proved by refutation. Assume that there is a valid \( G(P) \), with \( C \in \epsilon(P) \). Some nodes \( P' \in s(C) \) are not marked SOLVED, and thus they
must be marked EXPANDING: if they are marked FAILED, we are in contradiction with Lemma 1. Thus, by recurrence of this hypothesis we obtain: There exists a loop in the DAG $G(P)$ or a node in $PM$ has been marked EXPANDING. Both of these possibilities obviously lead to a contradiction. □

The proof of the theorem is achieved by noticing that:

$Q$ is the first node visited;

every visited node $P$ is marked SOLVED or FAILED (cannot remain EXPANDING).

$P$ is marked SOLVED (FAILED) iff there is (there is not) a valid subgraph $G(P)$.

REFERENCES