The Achievement of Knowledge Bases by Cycle Search

Olivier Roussel and Philippe Mathieu

LIFL, CNRS URA 369, USTL bâtiment M3, 59655 Villeneuve d’Ascq Cedex, France
E-mail: roussel@lifl.fr, mathieu@lifl.fr

Forward chaining is an algorithm that is particularly simple and therefore used in many inference systems. It computes the facts that are implied by a set of facts and rules. Unfortunately, this algorithm is not complete with respect to negation. To solve this problem, it is possible, in the context of propositional calculus, to automatically add the rules needed to make forward chaining complete. This transformation is a logical compilation of knowledge bases. This article presents a new method, based on a cycle search in a graph associated to the set of rules to compile, which allows a precise identification of what is needed for completeness.

Key Words: knowledge compilation; expert systems; resolution.

1. INTRODUCTION

Among the various inference algorithms, forward chaining appears to be particularly simple and efficient. Based on modus ponens, it is commonly used in expert systems. Beyond its simplicity and its speed, it has the advantage of being able to produce the set of facts that are implied by the knowledge, which is not the case of theorem provers such as backward chaining.

Unfortunately, forward chaining is not complete when disjunctions or negations are used. An interesting solution to this problem is called achievement. It is a kind of knowledge compilation which was first proposed in [MD90].

This paper presents a new method of achievement based on a cycle search in a graph. First, the definition of achievement is recalled. Then, we explain the incompleteness of forward chaining while giving the basis of the new method. To formalize this explanation, linear input resolution is recalled to introduce a necessary and sufficient condition of completeness. We show then that this condition yields an elegant achievement algorithm based on the search of cycles in a graph. We also prove that two kinds of cycles, namely ambiguous cycles and nonsimple cycles, need not be considered. Then, we stress the strengths and weaknesses of the method.

1 This work is supported by the Conseil Régional Nord-Pas-de-Calais and the CNRS as part of the GANYMEDE II project.
and give an example of its use for knowledge elicitation. Last, we compare the use of forward chaining on an achieved base to the use of a complete production method. This comparison shows that the cost of achievement is amortized after several queries and proves that achievement is a real compilation method.

2. ACHIEVEMENT

We deal with propositional calculus and the following definitions.

**Definition 1.** An *atom* is a Boolean variable. A *literal* is an atom or its negation ($a$ or $\neg a$). A *clause* is a disjunction of literals. A clause $C_1$ *subsumes* a clause $C_2$ iff $C_1 \subseteq C_2$. A *rule* is a formula of the form $\text{if condition then conclusion}$, where the condition is a conjunction of literals and the conclusion a single literal. Rules are also noted $\text{condition} \rightarrow \text{conclusion}$ (“$\rightarrow$” meaning *implies*). A rule $a_1 \land \cdots \land a_n \rightarrow b$ is semantically equivalent\(^2\) to the clause $\neg a_1 \lor \cdots \lor \neg a_n \lor b$.

A *fact* is a literal assumed to be true. A *base* of clauses (resp. rules, facts) is a set of clauses (resp. rules, facts). An *implicate* of a base is a clause which is implied by the base. A *prime implicate* is an implicate which is not subsumed by another. A *variant* of a clause $C$ is a rule whose clausal form is $C$.

Forward chaining works on a set of rules and a base of facts. It consists in going through the set of rules, and, for those whose condition is satisfied, adding their conclusion to the base of facts. For example, with the rules $[a \rightarrow b, a \rightarrow c, b \land c \rightarrow d]$ and the known fact $a$, the first and second rules let us deduce that $b$ and $c$ are both true. These literals are therefore added to the set of facts. Then, one can deduce with the last rule that $d$ is true.

Unfortunately, this algorithm is not complete (i.e., it fails to produce some implied literals) when disjunctions or negations are used. On the example above, if it is known that $d$ is false, forward chaining cannot derive $\neg a$ since no rule has $\neg d$ as condition.

There are several ways to avoid this incompleteness:

- writing only knowledge bases for which forward chaining is complete. This method is certainly the most used. Yet, it has severe disadvantages. First, it makes knowledge elicitation harder, since one has to take into account not only the semantic point of view, but also the syntactic one. But also, it becomes quickly intractable for an expert to guarantee the completeness of forward chaining. Therefore, there are usually no proofs that every deduction will be made, which is clearly unacceptable in critical applications.

- using a complete inference algorithm instead of forward chaining. For example, a satisfiability test could be used to build a production algorithm similar to forward chaining by testing for each atom $a$ whether or not $a$ and $\neg a$ are consistent with the base and current facts. But if $n$ is the number of atoms, this requires $O(n)$ satisfiability tests for each new set of facts and each test may take exponential

\(^2\) Throughout this paper, we consider the usual two-valued semantic. A formula is equivalent to another one if both have the same set of two-valued models.
time. Furthermore, this method is not compatible with most expert systems and makes procedural attachments difficult.

- adding rules so that forward chaining becomes complete for any set of facts.

This method is called achievement [MD90, MD94b] and has none of the previous disadvantages. First, there is no restriction on the way the expert writes the knowledge base. On the contrary, instead of sticking to rules, knowledge can be encoded by any formulae (such as disjunctions, but also at least and at most constructs) since achievement will translate them into rules understood by forward chaining. Second, completeness is guaranteed for any set of facts. Third, achievement is done once and only once. It is a compilation of the knowledge which remains valid whatever set of facts is later used. Therefore, its cost is amortized after several queries as will be shown later. Achieving a given base may be difficult since in the worst case, an exponential number of rules must be added. Nevertheless, such cases rarely occur in practice and even when they do, using forward chaining on the compiled base is often more efficient than $O(n)$ satisfiability tests on the initial base.

Last achieved bases are also complete for some other inference methods such as unit refutation [dV94]. The difference between unit refutation and forward chaining is that forward chaining is a production algorithm which computes all the unit implicates in a single run, whereas unit refutation can only check that one literal is implied.

This paper is devoted to the third solution, the use of achievement, which is defined below:

**Definition 2.** An *achievement* of a base $B$ is a base $Achvt(B)$ semantically equivalent to $B$ such that $\forall F$ set of facts, $\forall l$ literal, $(B \cup F \models l) \iff (l$ is produced by forward chaining on $Achvt(B) \cup F)$.

Several achievement algorithms are discussed in [MD90, MD94b]. Each basically computes the set of prime implicates. Unfortunately, none explain exactly why a rule must be added for completeness. The goal of this paper is to give a method that allows a precise identification of what is needed for completeness, allowing finer analyses of previous and future achievement methods.

### 3. BASIC IDEA OF THE NEW METHOD

Let us first see on our previous example why forward chaining is incomplete. With the rules \{$a \rightarrow b, a \rightarrow c, b \land c \rightarrow d$\}, it appears that forward chaining is unable to make use of the first rule to deduce $\neg a$ from the fact $\neg b$. This is caused by the orientation of rules and can easily be solved by the use of variants which generalize the notion of reciprocal or by an extension of forward chaining to clauses [Mat91]. This latter method, also called the Boolean constraint propagation in [FdK93], is just a way to avoid storing all the variants of a clause and is therefore the preferred one. Both methods are, however, equivalent. Thus, throughout this paper, we will use rules for clarity (and compatibility with existing expert systems) and clauses for efficiency and generality.
However, this is not the only reason of incompleteness. On our example, it is also impossible to infer \( \neg a \) from the fact \( \neg d \), and for that, variants are not sufficient. To get this implicate, it is necessary to infer first that if \( \neg d \) is true, then, either \( \neg b \) is true or \( \neg c \) is true. Such a derivation is impossible by forward chaining and is not really interesting since we only care about unit implicates (facts). However, it is essential to note that, whichever from \( \neg b \) and \( \neg c \) is true, it is always possible to infer \( \neg a \). So, deriving that a disjunction of facts is true appears useful only when each of these facts lets us prove a common consequence. In that case, the completeness of the method requires this inference. When we use arrows on a graph to represent the “implications” used in the above inference, it appears that disjunctions correspond to a fork and, deducing a common consequence, to a junction of arrows. These forks-junctions, illustrated below, are characteristic of the clauses to add.

![Diagram](image)

The remarks, together with the results on achievement by parts \(^3\) [RM95], lead to the idea of achieving a base by searching cycles in a graph.

The formalization of these intuitions is based on the representation of a set of clauses by a graph and on linear input resolution [Lov78]. It is shown that searching for one kind of cycle is a way to easily identify linear resolutions producing the clauses needed for completeness.

4. LINEAR INPUT RESOLUTION

The definition of linear input resolution as well as some of its properties are now recalled.

**Definition 3.** A linear input resolution L from a set B of clauses is a pair of sequence of clauses \( ((C_0, C_1, \ldots, C_{n-1}), (R_0, R_1, \ldots, R_n)) \) such that (1) \( \forall i, C_i \in B \), (2) \( R_0 = C_0 \) and, (3) \( \forall i > 0, R_i \) is a binary resolvent of \( R_{i-1} \) with \( C_{i-1}. \) The clauses \( C_i \) are called side clauses, while \( R_i \) are called center clauses. The clause \( C_0 \) is the root of the linear input resolution. \( R_n \) is the result of the linear input resolution, also called the resolvent of \( L \).

The only difference with the general linear resolution is that it is forbidden to use a center clause as a side clause. Therefore, linear input resolution is not complete for refutation or the computation of prime implicates. For example, it is impossible to derive the empty clause by input resolution from the set of clauses \( \{a \lor b, a \lor \neg b, \neg a \lor b, \neg a \lor \neg b\} \).

\(^3\) Achievement by parts is an efficient achievement method which splits the base in a certain way, achieves each part independently, and merges the resulting bases. Splitting is done by taking the biconnected components of a graph different from the one considered here.
This incompleteness is avoided by doing a saturation. Whenever a resolvent is produced, every linear input resolution using this resolvent is built, the same way saturation by binary resolution is done [CL73].

**Definition 4.** Let $B$ be a base of clauses. Let $S(B) = B \cup \{C, C$ is a clause obtained by linear input resolution from $B\}$. The saturation by linear input resolution of a base $B$ is the first set $S^n(B)$ such that $S^{n+1}(B) = S^n(B)$.

**Proposition 5.** Each prime implicate of a base $B$ is in the saturation by linear input resolution of $B$.

Each prime implicate can be obtained by a linear resolution [Mat91] and each linear resolution can be turned into a sequence of linear input resolutions which first generate the center clauses also used as side clauses and then reuse these clauses to produce the desired implicate.

## 5. NECESSARY AND SUFFICIENT CONDITION OF ACHIEVEMENT

It is known [MD90, MD94b] that the prime implicates of a base are an achievement of it and that only some of them are needed. A characterization of the required implicates based on the concept of merge resolvent is now given.

**Definition 6.** A binary resolvent between two clauses $A$ and $B$ is a merge resolvent if $A \land B \neq \emptyset$. The literals in $A \land B$ are merge literals.

Generally, a merge literal is one that forward chaining cannot produce. For example, it cannot infer $b$ from $\{\neg a, p \lor h \lor a, \neg p \lor b\}$ because $b$ is a merge literal in the implicate $a \lor b$. However, when a literal is merged in an implicate, but not in the same implicate obtained by another way, forward chaining may be able to infer it.

**Example 7.** The following base is achieved.

\[
\begin{align*}
& p \lor a \lor b \\
& \neg p \lor a \lor c \\
& q \lor b \lor a \\
& \neg q \lor b \lor c
\end{align*}
\]

Yet, the prime implicate $a \lor b \lor c$ cannot be obtained without a merge. However, because the merge literal is never the same, it is unnecessary to add this clause.

The theorem below generalizes this example.

**Theorem 8.** A base $B$ is achieve if and only if $\forall C$ prime implicate of $B$, $\forall l \in C$, $C$ can be obtained by at least a linear input resolution from $B$ which does not contain a merge of $l$ and such that no center clause is resolved upon a merged literal.

**Proof.** $(\Leftarrow)$ is proved in [RM96a] and $(\Rightarrow)$ is easily proved by contradiction.

**Example 9.** The base $\{a \lor b, a \lor \neg b, \neg a \lor b\}$ is not achieved since $a$ is a prime implicate which we cannot get by a linear input resolution where $a$ would not
be merged. This is also true for \( b \). Adding \( a \) to the base achieves it, since now every prime implicate satisfies the conditions of the theorem. For example, \( b \) can be obtained with no merge from \( a \) and \( \neg a \lor b \).

So, to achieve a given base, it is sufficient to add the implicate that are obtained by different linear input resolutions with the same literal being merged in each of these resolutions. This can of course be done by saturation by linear input resolution. It can easily be proved that it is unnecessary to keep at each stage of the saturation clauses obtained by linear input resolutions without a merge. So each step consists in producing clauses that are obtained by a linear input resolution ending with a merge.

6. TRANSLATION ON A GRAPH

We now consider how the notions of linear input resolution and merge resolution translate on a graph and obtain an elegant characterization of the clauses needed for achievement.

**Definition 10.** A graph of literals associated to a base \( B \) is the graph such that

- its set of nodes is
  - the set of atoms of \( B \) and their negation (literal nodes),
  - the set of the identifiers of the clauses of \( B \) (clause nodes).
- its set of edges is
  - the set of \( i \land j \) such that \( i \) is a literal of the clause whose identifier is \( j \) (clause edges)
  - the set of \( a \land \neg a \) for every atom \( a \) (link edges).

We use numbers to identify clauses and two different colors for edges: one for clause edges and another one for link edges. Also, when it is not ambiguous, the identifier of a clause node may be omitted on the graph.

**Example 11.** The graph of literals of the base \( \{1 \ a \rightarrow b, \ 2 \ b \rightarrow a, \ 3 \ a \} \) is the following:

![Graph of literals](image)

We chose to draw a link edge with a dashed line and to mark clause nodes by a dot.

The link edges represent literals that may be resolved upon, exactly as links in connection graphs [Kow75]. The main difference between our graph and a connection graph is that this latter does not allow an easy detection of merge resolvents.
Definition 12. A resolution chain is a chain (i.e., a sequence of edges) of the graph of literals of the form

$$l_{L1} - C_1 - l_{R1} \longrightarrow l_{L2} - C_2 - l_{R2} \longrightarrow \ldots \longrightarrow l_{Ln} - C_n - l_{Rn},$$

where each $l_{Li}$, $l_{Ri}$ is a literal of the base and $C_i$ is an identifier of a clause of the base such that

$$\forall i, \begin{cases} l_{Li} \in C_i, \\ l_{Ri} \in C_i, \\ l_{Le} = -l_{R(i-1)}, \\ l_{Le} \neq l_{Ri}. \end{cases}$$

A simple resolution chain is a resolution chain which does not contain the same node more than once, except the ends of the chain excepted.

Note 13. The condition $l_{Le} \neq l_{Ri}$ just forbids resolving twice upon the same literal in two successive resolutions. Since the literal resolved upon disappears from the resolvent, it cannot be resolved upon again.

To each resolution chain can be associated a linear input resolution, obtained by reading the clauses of the chain from one end to the other, keeping the same order in the linear input resolution and resolving upon literals designated by link edges.

But in fact, a resolution chain represents several linear input resolutions. The root clause of the resolution can be any of the clauses in the chain. The next side clause in the resolution can be any of the clauses adjacent to the clauses already put in the resolution. So, in the most general case, it is possible to choose for the next side clause between a clause to the left and one to the right of the clauses already put in the resolution.

Example 14. The figure below shows the four linear input resolutions associated to the following chain.

![Diagram](image)

Definition 15. A linear input resolution $L$ is associated to a resolution chain $C$ and reciprocally if and only if the conditions below are satisfied

- The root clause of $L$ is a clause of $C$.
- The side clause $C_i$ of $L$ is one of the clauses of $C$ linked to the subchain composed by the preceding side and root clauses by a link edge which represents the literal to be resolved upon in the resolution with $C_i$. 

Note 16. It is clear that to each resolution chain is associated at last one linear input resolution. However, the converse is false. There are some linear input resolutions which are not associated to a single resolution chain, such as the one having \( \{ a \lor b \lor c, \neg c \lor d, \neg b \lor e, \neg a \lor f \} \) for side clauses. However, each resolvent needed for achievement does correspond to a single resolution chain as will be proved later.

Definition 17. A **Merge cycle** is a resolution chain such that the first literal of the chain and the last are the same.

It is the same concept as that of the tied chain developed in [Esh93]. However, we prefer the term merge cycle which is more representative of the main property of this object, in our opinion. [Esh93] used tied chains to build a sufficient test of completeness for unit refutability. Since forward chaining is a kind of unit resolution, it is not surprising to find the same concept in both contexts.

The key property of a merge cycle is that any merge resolvent translates into a merge cycle.

Proposition 18 [RM96a]. Each linear input resolution ending with a merge can be associated to a merge cycle.

7. AMBIGUOUS CYCLES ARE USELESS

Several different linear input resolvents may correspond to a cycle or resolution chain. If this is the case, the resolution chain is told to be ambiguous. Such chains are characterized by the occurrence of bridges.

Definition 19. A **bridge** in a resolution chain is an edge which links a clause of the chain (the bridge’s clause) to a literal of the chain (the bridge’s literal) belonging to another clause in the chain, and which is not an end of the chain.

The bridge’s literal is not allowed to be one end of the chain because, for the chain to be ambiguous, the bridge’s literal must be resolved upon. Depending on the order in which the bridge’s clause and the resolution upon the bridge’s literal appear in the linear input resolution, the bridge’s literal will appear or not in the final resolvent, which is the way we can get different resolvents from a single chain.

Definition 20. An **ambiguous chain** is a chain with at least one bridge. An **ambiguous merge cycle** is an ambiguous chain with the same literal at both ends.

Example 21. The chain below \( \{ a, b, \neg b, c, \neg c, d, \neg d, e \} \) is not ambiguous

\[
\begin{array}{cccccccc}
 a & \bullet & b & \longrightarrow & \neg b & \bullet & c & \longrightarrow & \neg c & \bullet & d & \longrightarrow & \neg d & \bullet & e
\end{array}
\]

whereas the following one is, due to the bridge drawn in bold.
Two different linear input resolvents can be associated to this last chain:

\[
\begin{align*}
\neg a \lor b & \lor f \lor d \lor c \\
\neg b & \lor f \lor d \lor e \\
\neg c & \lor g \lor h \lor d \\
\neg c & \lor e \lor g \lor h \\
\neg b & \lor f \lor d \lor e \\
\neg d & \lor e \\
\neg c & \lor g \lor h \\
\neg b & \lor e \lor f \lor g \lor h \\
\neg a & \lor f \lor g \lor h \\
a & \lor e \lor f \lor g \lor h
\end{align*}
\]

PROPOSITION 22 [RM06a]. Let \( C \) be a resolution chain of the form:

\[
l_{L_1} - C_1 - l_{R_1} \cdots l_{L_2} - C_2 - l_{R_2} \cdots l_{L_n} - C_n - l_{R_n}.
\]

If \( C \) is unambiguous, it represents a unique linear input resolvent which can be written \( \{l_{L_1}\} \cup \{l_{R_1}\} \cup \{C_i - \{l_{L_i}, l_{R_i}\}\} \).

Informally, an unambiguous (and simple) merge cycle looks like a sea urchin. The proposition above states that the associated resolvent is obtained by keeping its spines and its mouth (the ends of the chain).

Fortunately, the proposition below shows that ambiguous chains can be ignored in a saturation since the associated resolvents will be either useless or obtainable by unambiguous chains. This result frees us from considering every linear input resolvents associated to a chain or cycle and saves many computations.

PROPOSITION 23 [RM96a]. Let \( \% \) be an ambiguous resolution chain and \( L \) one of its associated linear input resolution. The resolvent of \( L \) is

- a tautology, or
- a clause subsumed by another implicate of clauses of \( \% \), or
- a clause obtainable from an unambiguous resolution chain built from \( \% \) by replacing merge cycles created by bridges with the linear input resolvent they represent.
8. NONSIMPLE CYCLES ARE USELESS

It is also proved that considering nonsimple cycles is useless. This is a crucial result for the method to be applicable (there are infinitely many nonsimple cycles as soon as there is a cycle).

PROPOSITION 24 [RM96a]. Any resolvent neither tautological nor subsumed, and which is associated to a resolution chain neither ambiguous nor simple, can be obtained by saturation using only resolvents obtained by simple and unambiguous resolution chains.

Unfortunately, we cannot restrict further the search to only fundamental cycles, or avoid a saturation, as the example below proves.

Example 25. It is necessary to add four clauses to achieve this base.

\[ a_1 \lor a_2 \lor a_2 \\
\neg a_1 \lor b_1 \lor c_i \quad i \in \{1, 2, 3\} \\
\neg c_i \lor d \]

If no saturation is done, it is impossible to add the clause \( b_1 \lor b_2 \lor b_3 \lor d \) which is required for completeness. If we consider only fundamental cycles, one of the three clauses \( b_i \lor b_j \lor a_k \lor d \) (i, j, k all different) will be missing, depending on the chosen cycle base. This missing clause cannot be produced later by saturation.

9. ALGORITHM

The preceding results prove that the following algorithm computes an achievement of a base \( B \). In this algorithm, \( B_i \) holds the base after \( i \) saturation steps and \( C_i \) the clauses added by step number \( i \).

1: \( B_0 \leftarrow B; \ C_0 \leftarrow \emptyset; \ i \leftarrow 0 \)
2: while \( C_i \neq \emptyset \) do
3: \/* find merge cycles */
4: Compute \( S \) the set of clauses associated to simple and unambiguous merge cycles of \( B_i \) in which at least one clause of \( C_i \) occurs
5: \/* cancel tautologies, subsumed clauses and implicates obtainable by different ways with different merges */
6: \( C_{i+1} \leftarrow \emptyset \)
7: for all clause \( C \) of \( S \) do
8: \/* find merge cycles */
9: if \( C \) is neither a tautology, nor subsumed by a clause of \( B_i \cup C_{i+1} \) then

---

4 A simple cycle is a cycle which does not contain the same vertex twice, except at both ends.
5 A fundamental cycle is a cycle, a member of a basis from which every cycle can be built.
10: \quad \text{if there is at least one variant of } C \text{ which represents an inference that cannot be done by forward chaining from } B_i \cup C_{i+1} \text{ then}
11: \quad \quad C_{i+1} \leftarrow C_{i+1} \cup \{C\}
12: \quad \quad \text{end if}
13: \quad \text{end if}
14: \quad \text{end for}
15: \quad B_{i+1} \leftarrow B_i \cup C_{i+1}; \; i \leftarrow i + 1
16: \quad \text{end while}

On the last described base, the algorithm will work as follows. First, \(B_0 = C_0 = B\). \(B_0\) contains three cycles which all have a clause of \(C_0\). These cycles are all simple, unambiguous merge cycles. The resolvents they represent are \(a_1 \lor b_2 \lor b_3 \lor d\), \(b_1 \lor a_2 \lor b_3 \lor d\), and \(b_1 \lor b_2 \lor a_3 \lor d\). As these clauses are neither tautological nor subsumed, and not even obtainable without merging \(d\), they all appear in \(C_1\) and are added to the graph. Thus \(B_1\) contains three simple, unambiguous merge cycles using clauses of \(C_1\). All these cycles produce the same clause \(b_1 \lor b_2 \lor b_3 \lor d\) which is not a tautology, is not subsumed, and cannot be obtained without merging \(d\). Hence, \(C_2 = \{b_1 \lor b_2 \lor b_3 \lor d\}\) and this clause is added to \(B_1\) to get \(B_2\). This addition does not generate any new merge cycle and the algorithm stops with \(C_3 = \emptyset\). The base \(B\) is achieved by the addition (to the variants of the initial base) of the variants of the four clauses above having \(d\) as conclusion (since it is the merge literal).

10. STRENGTHS AND WEAKNESS

The previous algorithm is not a precise description of an implementation, but rather a simple and concise description of the method. We now give some details about the refinements which make the interest of the method and also stress some of its weaknesses.

The first point is that the cycle search starts from the merged literal, that is from both ends of the cycle toward the center. This brings substantial savings compared to a filtering algorithm such as \(\text{FPI}_0\) \[^{[dV94]}\]. Briefly stated, \(\text{FPI}_0\)\(^6\) uses any prime implicate algorithm to produce the prime implicates and filters among them those obtained by a merge resolution. Our point of view is to first identify possible merged literals and then check if we can build a merge resolution from them. Therefore, no resolution step is performed if no possible merged literal is found, which is not the case in the filtering approach.

Next, the cycle approach forbids resolutions between clauses belonging to different biconnected components. A biconnected component is a maximal subgraph which contains no cut node. A cut node is a node whose removal cancels every path between two nodes that were previously connected. Every path between two biconnected components goes through the cut node. Therefore, no simple cycle can cross the boundary of a biconnected component. This restriction dramatically improves the compilation time as \[^{[RM95]}\] proved.

\(^6\)\(\text{FPI}_0\) (Filtered Prime Implicate 0) as well as \(\text{FPI}_2\) produce achieved bases, but with redundant clauses. Bases produced by \(\text{FPI}_1\) are not achieved in the general case.
Also, before adding a clause generated by a cycle to the base, our algorithm checks if this addition really allows a deduction that forward chaining was unable to perform. This is not the case when the clause can be obtained by another linear input resolution with no merging of the literal at both ends of the cycle. Of course, enumerating every other linear input resolution to do this test would be intractable. In fact, the most efficient way is to use forward chaining. Let \( C \) be the clause produced by the cycle and \( l \) the literal at both ends. We just have to check if forward chaining can produce \( l \) from the set of literals \( \neg(C - \{l\}) \). If it cannot, \( C \) must be added. If it can, \( C \) need not be added (unless another cycle yields the same clause with another merged literal that forward chaining cannot produce).

For example, the first four clauses of \( \{a \lor b \lor c, \neg c \lor d \lor e, \neg e \lor f, \neg f \lor a, b \lor d \lor e\} \) form a merge cycle.

This cycle generates the clause \( b \lor d \lor a \), \( a \) being the merged literal. Yet, it need not be added to the base (the resolution of the last three clauses of the base give the same clause but without merging \( a \)). Here is how it is detected by forward chaining.

We put in the base of facts the negation of the literals of the produced clause—merged literal excepted—that is, \( \neg b, \neg d \). From there, forward chaining proves \( e \), then \( f \), and at last \( a \), the merged literal. This means that the considered merge cycle represents a deduction that forward chaining was already able to do. Therefore, we do not add the corresponding clause.

This is, of course, an efficient test but the most interesting point is that it can be included in the search for cycles and used to further prune the search space. On the previous example, we could start the cycle search from \( a \) which may be a merged literal since it belongs to two clauses. If we start the cycle by the two clauses \( a \lor b \lor c \) and \( \neg c \lor d \lor e \), we already know that \( b \) and \( d \) will appear in the clause corresponding to the cycle. So, we can add their negation to the base of facts and run forward chaining. This produces \( e \). It also proves that whatever way we complete the cycle, the corresponding clause will be redundant since it will be obtainable by a linear input resolution without merging \( a \). So we know we can avoid the resolutions steps with \( \neg e \lor f \) and \( \neg f \lor a \). More generally, each time we add a clause to the current path in the cycle search, we also add to the base of facts the negation of the literals which do not belong to the path. Then forward chaining is run. The pruning criterium is to forbid the extension of the current path by edges containing one literal proved by this forward chaining because this would give a redundant cycle.

Last, when trying to extend a path with a clause, it is interesting to check if the resulting path can effectively be extended to close the cycle, especially when performing incremental achievement. This can be done by maintaining information on the
transitive closure of the relation $\mathcal{R}$ defined by $C \mathcal{R} C'$ iff there is a chain $C \vdash \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \n
6: if $l \in F$ then
7:     continue; /* $l$ is obviously implied */
8: end if
9: if $\neg l \in F$ then
10:    continue; /* $l$ is not implied since the base is satisfiable */
11: end if
12: if $B \land F \cup \{\neg l\}$ unsatisfiable then
13:    $F \leftarrow F \cup \{l\}$; /* $l$ is implies */
14: end if
15: end for
16: return $F$

We measured $t_{\text{sat}}$, the global time needed for this algorithm to perform the inferences on a set of $n$ random bases of facts, with the time $t_{\text{fetch}}$ needed by forward chaining to perform the same inferences on the achieved base. From these measures, we define the mean speedup by inference as $S = t_{\text{sat}} / t_{\text{fetch}}$ and the mean break-even point as $B = n \cdot t_{\text{comp}} / (t_{\text{sat}} - t_{\text{fetch}})$ where $t_{\text{comp}}$ is the time required for the achievement of the base. This number represents how many inferences should be performed before compilation is amortized. The experimental results are given in the table (all times are in seconds and measured on a Sparc-5/85).

<table>
<thead>
<tr>
<th>Name</th>
<th>$t_{\text{comp}}$</th>
<th>$t_{\text{sat}}$</th>
<th>$t_{\text{fetch}}$</th>
<th>$n$</th>
<th>$S$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deputes</td>
<td>0.085</td>
<td>0.558</td>
<td>0.252</td>
<td>422</td>
<td>2.2</td>
<td>117.2</td>
</tr>
<tr>
<td>Logiciens</td>
<td>0.035</td>
<td>0.307</td>
<td>0.154</td>
<td>300</td>
<td>1.9</td>
<td>68.7</td>
</tr>
<tr>
<td>Pannes</td>
<td>0.085</td>
<td>0.608</td>
<td>0.253</td>
<td>450</td>
<td>2.3</td>
<td>107.8</td>
</tr>
<tr>
<td>Type1-76</td>
<td>0.175</td>
<td>2.44</td>
<td>0.867</td>
<td>526</td>
<td>2.8</td>
<td>58.1</td>
</tr>
<tr>
<td>Type3-50</td>
<td>0.125</td>
<td>1.818</td>
<td>0.193</td>
<td>102</td>
<td>9.4</td>
<td>7.8</td>
</tr>
<tr>
<td>Type4-9</td>
<td>0.005</td>
<td>0.504</td>
<td>0.185</td>
<td>252</td>
<td>2.7</td>
<td>79484.2</td>
</tr>
<tr>
<td>Type5-5</td>
<td>0.005</td>
<td>0.587</td>
<td>0.252</td>
<td>420</td>
<td>2.3</td>
<td>6.2</td>
</tr>
<tr>
<td>Type6-11</td>
<td>0.055</td>
<td>1.821</td>
<td>0.583</td>
<td>529</td>
<td>3.1</td>
<td>23.4</td>
</tr>
<tr>
<td>Type7-5</td>
<td>0.015</td>
<td>0.718</td>
<td>0.253</td>
<td>375</td>
<td>2.8</td>
<td>12.1</td>
</tr>
<tr>
<td>Pigeon-2-3</td>
<td>0.005</td>
<td>0.102</td>
<td>0.053</td>
<td>150</td>
<td>1.9</td>
<td>15.4</td>
</tr>
<tr>
<td>Pigeon-3-4</td>
<td>0.115</td>
<td>0.379</td>
<td>0.194</td>
<td>330</td>
<td>1.9</td>
<td>205.4</td>
</tr>
<tr>
<td>Pigeon-4-5</td>
<td>28.1</td>
<td>1.177</td>
<td>1.099</td>
<td>570</td>
<td>1.0</td>
<td>207248</td>
</tr>
<tr>
<td>Ramsey-4</td>
<td>51.9</td>
<td>0.903</td>
<td>0.338</td>
<td>482</td>
<td>2.6</td>
<td>44306.4</td>
</tr>
<tr>
<td>Nqueens-6</td>
<td>8.955</td>
<td>4.337</td>
<td>0.351</td>
<td>490</td>
<td>12.3</td>
<td>1100.8</td>
</tr>
<tr>
<td>Chandra21</td>
<td>22.6</td>
<td>0.058</td>
<td>0.045</td>
<td>35</td>
<td>1.2</td>
<td>61061.6</td>
</tr>
<tr>
<td>Chandra24</td>
<td>221.2</td>
<td>0.024</td>
<td>0.013</td>
<td>14</td>
<td>1.7</td>
<td>299759</td>
</tr>
<tr>
<td>Easy</td>
<td>0.025</td>
<td>0.249</td>
<td>0.099</td>
<td>240</td>
<td>2.5</td>
<td>39.9</td>
</tr>
<tr>
<td>History-ex</td>
<td>0.015</td>
<td>0.382</td>
<td>0.186</td>
<td>307</td>
<td>2.0</td>
<td>23.5</td>
</tr>
<tr>
<td>Selenoid</td>
<td>0.025</td>
<td>0.322</td>
<td>0.128</td>
<td>300</td>
<td>2.5</td>
<td>38.5</td>
</tr>
<tr>
<td>Valve</td>
<td>0.125</td>
<td>0.633</td>
<td>0.182</td>
<td>360</td>
<td>3.4</td>
<td>99.7</td>
</tr>
<tr>
<td>Two-pipes</td>
<td>1.445</td>
<td>0.731</td>
<td>0.249</td>
<td>420</td>
<td>2.9</td>
<td>1259.9</td>
</tr>
<tr>
<td>Three-pipes</td>
<td>19.0</td>
<td>1.533</td>
<td>0.494</td>
<td>600</td>
<td>3.0</td>
<td>11012.4</td>
</tr>
</tbody>
</table>
Deputes and logiciens are taken from [Car66]. Pannes is a base described in [Sie87]. Type-*-* are structured bases defined in [Mat91]. Pigeon, ramsey, and nqueens are well-known. Bases chandra to four-pipes are taken from [FdK93]. Mine-\(x-y\) encodes the mine sweeper game on a \(x \times y\) field. Adder-\(n\) is an \(n\) bits adder. Multi-*-*\(x-y\) are multipliers of \(x\) bits by \(y\), some of them with an additional order constraint on the two operands. Last, 2tree-* and cycle1-* are bases whose graphs contain many cycles.

As we can see, forward chaining on the achieved base is faster than a complete method based on a satisfiability test with a speedup of 2 to 3. The break-even point varies from one base to another and is the larger for bases whose achievement is exponential (pigeon-\(n\)-*, chandra*, cycle1-*,...). However, in most cases, the compilation is amortized after a very small number of inferences (from 100 to 1000). This proves that achievement is a real compilation method whose cost is amortized after several queries.

13. CONCLUSION

Forward chaining is an algorithm for producing implied literals which is often used in expert systems. Unfortunately, this algorithm is not complete as soon as negations or disjunctions are used. In previous articles [MD90, MD94b, MD94a], a logical compilation of knowledge bases was presented which compiles a base once, allowing later a simple forward chaining to compute every implied literal, and this, whatever base of facts is chosen. This article presents a new compilation method called achievement by cycle search, based on the computation of cycles in a graph. This method lets us identify precisely parts of the knowledge for which forward chaining is not complete and which we complete. Thus, we can avoid the production of every prime implicate, avoiding as soon as possible many useless computations. Among other applications, achievement by cycle search shows the writer of a knowledge base which subsets of the base must be completed. This method makes possible the implementation of an editor, telling incrementally, as each rules is entered, what the rules missing for completeness are. Furthermore, such a system can produce a graphical explanation. It is therefore an interesting tool to help developing knowledge bases in propositional calculus. Also, this method gives another explanation of achievement by parts methods [RM95] and lets us derive other segmentation theorems. Last, it is clearly proved that achievement is a real compilation method which can produce substantial speedup.
REFERENCES


