RESEARCH CONTRIBUTIONS

Artificial Intelligence and Language Processing

Chris Montgomery and Lloyd Fosdick Editors

Matings in Matrices

WOLFGANG BIBEL *Technische Universität München, West Germany*

ABSTRACT: This paper gives an overview of the connection method, developed by the author in Automated Theorem Proving. Its prominent features are illustrated with a number of examples. Well-defined measures of efficiency have shown it to outperform standard proof methods. Some of its features are also present in Andrews' independent approach via matings. The relationship between these two methods is clarified.

Wolfgang Bibel's current research interests include mechanization of reasoning, natural language processing, and computer architecture dedicated to knowledge representation and processing.

Author's Present Address: Dr. Wolfgang Bibel, Institut für Informatik der Technischen Universität München, D-8000, München 2, Postfach 20 24 20, West Germany

A preliminary version of this paper was presented as an invited lecture to the German Workshop on Artificial Intelligence, [20].

0001-0782/83/1100-0844 75¢

1. INTRODUCTION

An essential feature of human thinking is the capacity for logical reasoning. Everyone uses it all the time, mostly in an unconscious way. It was this capacity which eventually led to man's scientific activity. Therefore, it is not surprising that since the dawn of history, man has reflected upon the very nature of logical reasoning. He has modeled it within natural language, and isolated general rules operative in the human brain in everyday experience. In the course of time, the rules of logic were abstracted from natural language and expressed within formal languages, which model natural languages to the extent that they keep the logical structure intact. These formal languages together with the rules are called *formal*, or *logical systems*.

There are many such formal systems to date, each developed according to the purpose it was meant to serve. For instance, one formal system might be developed as a tool to be used by a mathematician to prove theorems within some theory, while another might be of a purely machine-oriented nature. Formal systems developed within non-monotonic logic must account for contingent truths; clearly, such systems, which admit the possibility of contingency, would be inappropriate for proving mathematical theorems.

Most of these formal systems are based on *first-order logic* (FOL). Its rules are so fundamental that any (existing or future) system of practical importance will probably have incorporated them in some form. This is, for instance, true of systems for both higher-order and modal logic which simply are extensions of FOL. Further, FOL is both natural and powerful enough to model much of our reasoning adequately. We will, therefore, focus our attention on FOL keeping in mind that, while this restriction may be artificial, it is certainly reasonable as a point of departure.

Although this paper is written with a particular formal system for FOL in mind, we will not specify the details here. No trouble should arise for readers familiar with the basic concepts of FOL. As usual, we have the class of (*well-formed*) *formulas* which correspond to syntactically correct fragments of text in natural language.

The most fundamental problem in FOL, as in any other formal system, is the development of a (hopefully efficient) procedure which for a given set of formulas $F_0, \ldots, F_n$, $n \geq 0$, decides whether F_0 is a consequence of $F_1, \ldots, F_n$ according to the rules in FOL. Such a procedure is called a *decision procedure*. According to a well-known result of Church (e.g., see [15], p. 170), this problem, in its most general form, has no solution, unless the F_i's are taken from certain subclasses of formulas (which frequently is the case for formulas of interest). For arbitrary F_i's, there are only so-called *semi-decision* or *proof procedures*, that is, procedures which are guaranteed to give a result in the affirmative case only, and in the negative case may give an answer (the usual case in practice) or run forever. These are the type of procedures we will study.

If we could develop an *efficient* such procedure this would have an enormous practical impact for many kinds of applications in science because of the general nature of FOL discussed before. Some of these applications are described by Nilsson [18]. It has occasionally been argued against this kind of general approach with reference to results (and conjectures) from complexity theory which seem to indicate that *in principle* such procedures cannot be efficient for arbitrary formulas. But even were this the case, it would not say very much about the feasibility of proof procedures *in practice* because of the worst-case nature of such general complexity results (*cf.*, Sect. IV.3 in [7]). Therefore, the development of efficient proof procedures does remain a challenging and promising research goal.

This is not to say that current proof procedures are completely inefficient. On the contrary, running deductive systems have proved rather deep mathematical theorems automatically; they have even solved a number of open mathematical problems (none of the famous ones, of course) for which a human proof was not found. Further they are in daily use as programming aids, generating or verifying pieces of programming code. (The interested reader is referred to the Proceedings of the Conferences on Automated Deduction [9, 14, 17, 21]. In many respects, these achievements are still modest, however, when compared with human performance.

The reasons for their deficiencies seem to be of two different kinds. First, human beings seem to adapt, quickly, powerful strategies which speed up the search. Research is just beginning to study such adaptive mechanisms. Second, the existing systems are based on proof procedures which work in such a redundant way that it is amazing that they are at all successful. Researchers, like all people, tend to a monotheistic attitude in such a situation, expecting the cure by solving *one* of these two kinds of problems. We believe, however, that *both* kinds of deficiencies necessarily have to be removed in order to substantially enhance the performance of running systems.

This paper is concerned with only one of these kinds of deficiencies: redundancy. It plagues all the popular proof procedures, in particular those based on the resolution principle introduced by Robinson [19], as well as most of those based on a natural deduction-like approach [10].

In the last decade, however, there have been two essentially independent but closely related developments, which provide an improvement in this direction. One is due to Andrews, the other to the author. The results have been published in a number of papers of a rather technical nature (see [1–6] for the most recent ones). Therefore, we attempt to provide a more expository overview of this method which is provably less redundant than any other known proof method. In the course of this overview, we clarify the relationship between Andrews' and the author's approach. Occasionally, we compare our method with standard proof methods. For a comprehensive treatment, the reader is referred to [7].

It is hoped that the presentation is such that not only an expert in the field will be able to quickly grasp the essence of this method, but also a non-expert with some familiarity with FOL will get a feeling for the enhancement achieved (a summary of which appears in Section 8). Due to the nature of the exposition and to limitations in space, the examples discussed are necessarily trivial and consequently do not reflect the generality of application. For more complex problems, the many details of a technical nature have to be left to the computer in actual implementations. In fact, the method has been implemented both by Andrews and the author together with their associates (e.g., see [17], pp. 50–69). A more advanced implementation is currently in progress (project "Beweisverfahren" supported by the Deutsche Forschungsgemeinschaft).

2. THE BASIC CONCEPTS FOR THE CONNECTION METHOD

As our first example, we choose a very old syllogism saying that the man, Socrates, is mortal since every man is mortal. The logic of such a statement in natural language is often ambiguous. For this reason, the formal first-order language has been developed in order to express the statement in the following logically equivalent but unambiguous way (see [15], or Chapter 4 in [18] for an introduction).

*A*1:	$\forall$**x (MANx** $\rightarrow$ **MTLx)**	: "every man is mortal"
*A*2:	**MANsocrates**	: "Socrates is a man"
TH:	**MTLsocrates**	: "Socrates is mortal"

With these partial statements, the whole statement says: from the *axioms*, *A*1 *and A*2, we may infer the *theorem*, *TH*. In fact, we may express the whole statement as the single formula $A1 \wedge A2 \rightarrow TH$ wherein *A*1, *A*2, and *TH* abbreviate the respective formulas above. Note the usage of the convention that $\wedge$ binds more than $\rightarrow$ in order to save parentheses. (Such conventions will henceforth be assumed). For purely didactic reasons, in this formula, the implication sign $\rightarrow$ is replaced equivalently by negation $\neg$ and disjunction $\vee$ and the scope of each occurrence of $\neg$ is made to be atomic by applying the well known equivalence rules relating logical connectives (*cf.*, III.1.4 in [7]). The resulting formula then reads

F:$\exists$**x(MANx** $\wedge$ $\neg$**MTLx)** $\vee$ $\neg$**MANsocrates** $\vee$ **MTLsocrates**

The original inference from *A*1 and *A*2 to *TH* is a valid one if and only if *F* is a *valid* formula or a *theorem* in the sense of FOL without non-logical axioms. The problem is how to test the validity of *F* (and of other theorems) as efficiently as possible.

For the following, it is illustrative to display such a formula in a two-dimensional format by listing the parts connected by $\vee$ from left to right, and within each such part connected by $\wedge$ from top down. *F* represented in this way reads

$$\exists x \begin{Bmatrix} \mathbf{MANx} \\ \neg\mathbf{MTLx} \end{Bmatrix} \{\neg\mathbf{MANsocrates}\} \{\mathbf{MTLsocrates}\}$$

Without a quantifier, $\exists$x, this structure is called a *matrix in normal form* which, by definition, is a set of sets of *literals* or shortly a set of *clauses*. In this particular example, we have a set of three clauses listed from left to right, the literals in each clause listed top–down. If one crosses such a matrix in two-dimensional format from left to right, visiting exactly one

literal in each clause, one obtains a good illustration for what is called a *path through* the matrix. Doing this with our matrix, also denoted by *F*, only the first clause gives us a choice of whether to visit the top or the bottom literal. Hence, there are exactly two paths through *F*

{MANx, ¬MANsocrates, MTLsocrates}

and

{¬MTLx, ¬MANsocrates, MTLsocrates}

An unordered pair of literals such as {MANx, ¬MANsocrates} with one and the same predicate symbol—here MAN—one literal unnegated, the other negated, and both contained in some path through a matrix, is called a *connection* in that matrix. A set of connections is called *spanning* for a matrix if each path through it contains such a connection (as a subset). Obviously, there are exactly two connections in *F* which, in fact, are spanning for *F* and are illustrated by

$$\exists x \left\{ \begin{matrix} \mathbf{MANx} \\ \mathbf{\neg MTLx} \end{matrix} \right\} \{\mathbf{\neg MANsocrates}\} \quad \{\mathbf{MTLsocrates}\}$$

Now, according to the results of Andrews and the author (see Corollary III.6.4 in [7]), *F* is in fact a theorem if, and only if, there is a substitution of some term for the variable x such that after this substitution, each of the two spanning connections consists of two *complementary* literals, *i.e.*, a literal *L* and its negated form ¬*L*. This obviously is the case if we substitute **socrates** for x; hence, *F*, in fact, is a theorem or, in other words, **MTLsocrates** is a logical consequence of the two axioms *A*1 and *A*2.

It was pointed out before that the elimination of the implication sign serves didactic purposes only. We now can see why it does not affect the essence of the method at all. We need just redefine the crucial notion of a spanning set of connections for arbitrary formulas via the equivalence rules mentioned above, a simple exercise indeed. For our original formula we thus obtain

$$\forall x(\mathbf{MANx} \rightarrow \mathbf{MTLx}) \wedge \mathbf{MANsocrates} \rightarrow \mathbf{MTLsocrates}$$

This demonstrates that it is a negligible technical detail whether we prefer to work with the originally given formula or the equivalent two-dimensional display. In this paper, we mainly use the latter since it displays the paths, the connections, and the spanning properties in a more transparent way.

Let us be sure that among all these comments and definitions we do not miss the crux of this discussion: For establishing the proof of our theorem, *F*, all we have to do is: (i) locate the two spanning connections within *F*; and (ii) test the existence of an appropriate substitution. It is important to note that this does not require any storage for copies of parts of the given formula, which, for this method, holds in general and not only with this trivial example.

For comparison, it is interesting to have a look at the popular resolution method [19]. For reasons which today may be regarded as historical, the given formula is first negated. The resulting formula is transformed into *clausal form* similar to what we did with *F* above. From the resulting set of clauses (or matrix), the empty set (or clause) is derived by two applications of the resolution rule as shown in Figure 1.

If we now recall our previous proof, it becomes obvious that each resolution step, resolving upon two literals, corresponds exactly to locating the connection between these two literals in the given formula and vice versa. The empty set is derived as soon as the set of connections thus obtained becomes spanning and vice versa. We notice that in the clausal form of resolution, the position of the negation in each connection is switched due to negating the original formula. Less trivial is the fact that in the course of the resolution proof a new clause, namely {**MTLsocrates**}, has been generated. Since, in general, no clauses may be deleted, this not only requires additional memory space but also increases the search space since we may resolve any two clauses, old or new. Much research effort has been invested in resolution in order to avoid these drawbacks which may become disastrous in more realistic problems where tens of thousands of generated clauses are not unusual. In our approach, these problems do not arise at all.

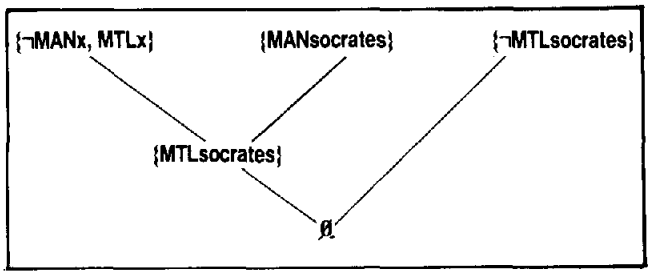


FIGURE 1. The Resolution Proof for F.

From this perspective, then, our method appears as a clever representation of resolution avoiding some of its drawbacks. That it is more will become clear as we proceed with additional examples. Perhaps at this point it might be appropriate to introduce some name for this new method. Over the years, the author used to call it the *systematic* method to stress the point that it affects a more systematic proof search than other methods. This is not a very distinctive name, however, since all these methods are more or less systematic. In view of the fact that locating connections in the formula may be regarded as its characteristic activity as we have already seen, *connection principle* or *method* will be used in this paper.

The connection method as described up to this point is identical with Andrews' *general matings* method except for the following notational differences. Following the resolution tradition Andrews starts with negating the given formula. Consequently, his paths run top–down rather than left–right in the matrix. Incidentally, he refers to a formula like the one explicitly named *F* above as in *negation normal form* where the scope of each explicit or implicit (e.g., via →) negation is atomic.

A set of connections in his terminology is a *potential mating*. A set of connections such that there is a substitution making all connected pairs of literals complementary is called a *mating*. If, additionally, it is spanning, *p-acceptable* in his terminology, then it is a *refutation mating*. The literals in a single connection are called *potential mates* (with respect to the empty mating). More substantial differences between the two methods than merely these notational ones will emerge as we proceed with further examples.

3. GENERATING SPANNING SETS OF CONNECTIONS

The previous section demonstrated that a proof with the connection method requires two things—namely, a spanning set of connections and an appropriate substitution. Although a realistic proof procedure will not perform these two tasks separately, a separate discussion will certainly be helpful for the reader. Therefore, in this section, we shall set aside all aspects related to substitution. This can be achieved by as-

suming that the appropriate substitution has been determined in advance. The formula F from the previous section in matrix form would then read

$$\begin{Bmatrix}\textbf{MANsocrates}\\ \neg\textbf{MTLsocrates}\end{Bmatrix}\{\neg\textbf{MANsocrates}\}\{\textbf{MTLsocrates}\}$$

By abbreviating **MANsocrates** by L and **MTLsocrates** by K, and deleting the braces we obtain simply $\begin{matrix}L\\ \neg K\end{matrix}$ $\neg L$ K.

In this case, where the literals are simply (possibly negated) propositional variables, our basic question whether F is a theorem is reduced to whether this matrix in propositional logic (sometimes called the *ground level*) is a tautology. As before, this is the case if, and only if, each of its paths contains a connection (see Theorem II.3.4 in [7]). We are now going to describe an algorithm which determines exactly that. Let us call it SSC for "spanning set of connections." Since our matrix is a little too simple for illustrating its behavior, we add four more literals, yielding

$$\begin{matrix}L & \neg L & \neg M & K & L\\ \neg K & M & & & K\end{matrix}$$

Initially, SSC chooses any clause, say the leftmost one, and in it any literal, say L. A data structure for storing the whole matrix is easily designed such that a clause containing the negation of the chosen literal can be found immediately, for instance, by looking it up in a table which contains all occurring literals in a determined order together with a reference to those clauses in which they occur. Doing this establishes a

first connection in our matrix $\rightarrow$ $\overset{\frown}{L}\ \overset{\frown}{\neg L}$ $\neg K$ M $\neg M$ K L K.
$\uparrow$

At the same time a pointer is set in the first clause (the little arrow in the picture) noting that L has been processed but the rest of the clause—in this case, just $\neg K$—remains to be processed. This completes the first step of SSC in which all paths containing this selected connection $\{L, \neg L\}$—only two in this particular example, but obviously there may be many more in general—have been checked and will *never be considered* by SSC.

The paths yet to be processed may be partitioned into those which contain L (but not $\neg L$) from the first connection and those which do not. SSC proceeds with the first ones while the second ones have been stored on a stack simply by the reference depicted by the pointer above. At this stage, the situation is essentially the same as the one after the selection of the first clause, the selected clause now being the second one from the left, marked with a vertical arrow. SSC chooses any member literal except $\neg L$. Here, this must be M since this is the only one left. As before, from the *remaining* clauses (to the right of the vertical arrow), we choose one which contains $\neg M$, thus establishing the second connection as shown

$$\begin{matrix}\overset{\frown}{L}\ \overset{\frown}{\neg L} & & & & L\\ \rightarrow\ \neg K & M & \neg M & K & K\\ & \rightarrow & \uparrow & & \end{matrix}$$

If there had been more literals in the second clause, this would have provided a new entry on the stack as illustrated with the second horizontal arrow pointing to nothing in this particular example. After this second step, all paths containing one of the selected connections have been checked.

SSC must continue to process those paths containing L (but not $\neg L$, and not both M and $\neg M$ any more). If there were such paths left, the same partition would be made with respect to M as before with respect to L; thus, we would proceed with the third step as before in the second step, and so forth. Note that such a chain is never longer than the clauses in the whole matrix since new clauses in this process are always selected from those not already involved in the present chain. Due to the particular situation of this example, there is no such third step; however, since all paths containing L also contain one of the selected connections. SSC notices this since the third clause does not contain any literal other than $\neg M$. In such a situation, it backs up by considering the topmost (nonempty) entry on the stack which is depicted by the leftmost horizontal arrow, thus starting a new chain from the situation illustrated by

$$\begin{matrix}\overset{\frown}{L}\ L & & & & L\\ \neg K & M & \neg M & K & K\\ \uparrow & & & & \end{matrix}$$

As before, SSC selects any literal in the actual clause marked by the vertical arrow, which has not been processed before, and any clause containing its complement, thus establishing a third connection as shown in

$$\begin{matrix}\overset{\frown}{L\ \ K} & \overset{\frown}{\neg L} & \neg M & L\\ \neg K & M & \neg M & K\\ \rightarrow & \uparrow & & \end{matrix}$$

where the reordering of the clauses is required for clarity of presentation. The horizontal arrow depicts later processing of further literals in the first clause (none in this case). The new chain after this step has already been completed. Since the stack is empty, SSC terminates with success. The three selected connections which, incidentally, need not be stored explicitly, are in fact spanning for this matrix. The rightmost clause and any connections containing its literals were redundant for the proof.

This completes the description of algorithm SSC for testing any matrix for a set of spanning connections. Above, we have already stressed that no extra storage of copies of the parts of the given formula is required. Only the pointers directing the chaining through the formula have to be stored in addition to a single copy of the formula and something like the table (of the size of the formula) mentioned at the beginning of this section. This fact is characteristic for SSC, not only in its simple form just described with a trivial example, but also in its full version applicable to arbitrary formulas.

A full version of this has been developed in [4] (see also pp. 326–341 in [17]). It is very general and has been designed to avoid certain redundant steps arising in special situations, for which reason its algorithmic details are rather complicated. It has been demonstrated [4] that this full version may simulate any known refinement of resolution with fewer or equal number of steps (and less storage) in the search for a proof, the amount of processing required for a single step being about the same in both cases. Since there are formulas for which SSC requires *strictly fewer* steps we see that the connection method provides a real (and provable) advantage over known resolution methods *in addition* to the representational advantage mentioned several times before. (There is another major advantage to be discussed in later sections.) As a guard against too much optimism we mention that determining a spanning set of connections is known as a hard problem in

general (actually co-NP-hard in the terminology of complexity theory); hence, no miracles should be expected for the worst case.

Because of its generality, the full version, in fact, may deal with *arbitrary* formulas or matrices not only ones in normal form (as our examples have considered thus far). For instance, any literal, e.g., M in the previous matrix, might be replaced by a whole (non-normal form) matrix; SSC still would be able to process such a matrix as before and without any change of its structure. The reason for this generalization lies in the fact that the characterization of tautologies via connections in each path, which was mentioned before, holds for arbitrary formulas. (See Theorem II.3.4 in [7].)

Andrews has demonstrated the disadvantages of the transformation to clausal form, which is required for resolution, with several examples. For instance, in [1], it has been shown that the simple mathematical statement $f(S \cup T) = f(S) \cup f(T)$ for a function f and two sets S and T after elimination of the defined operations (like $\cup$ but not $\leftrightarrow$) leads to a formula with 12 literals, compared with 104 literals in the corresponding clausal form. Now imagine the effect when the proof process *searches* among 104 rather than among 12 literals!

It might be helpful for the reader to consider SSC as a proof rule rather than an algorithm. Namely, for any given matrix, A, SSC in ech step essentially adds a single connection w to the set W of those connections obtained in previous steps; hence, the rule is $(A, W) \vdash (A, W \cup \{w\})$. (For more details, see Sections II.4 and II.5 in [7].)

Note that the formula, A, does not change, in contrast to any other logical rules. Initially, $W = \emptyset$ and the termination criterion for a successful derivation is the spanning property of W for A. If the process gets stuck before this criterion is fulfilled, then A is shown to be invalid. Thus, in the sense defined in the Introduction, SSC is actually a decision procedure for the subclass of propositional formulas (of the class of all formulas in FOL).

4. UNIFICATION

In Section 2, we have seen that a proof of a theorem involves the problem of

(i) determining a spanning set of connections such that

(ii) there is a substitution of terms for variables which makes the connected literals complementary.

In the previous section, we described the basic idea of the algorithm SSC for solving (i) on the ground level. Obviously, such an algorithm is applicable also on the *general* level, *i.e.*, in FOL, by simply neglecting the terms in the literals. Therefore, we now take the existence of such an algorithm for granted and ask for a solution of the subproblem (ii).

Since the test for (ii) may be performed by a fast algorithm, and a potential failure might be detected after any step of SSC, it is preferable to check for (ii) after each step. For example, after the first step of SSC is applied to the matrix F of Section 2, we would have to test whether there is a substitution, say σ_1, which makes *MANx* and **¬MANsocrates** complementary. Of course, $\sigma_1 = \{x \leftarrow \textbf{socrates}\}$ will do and thus is kept for the following steps. Thus, the situation after this first step may be illustrated as in the previous section with the substitution σ_1 added.

After having obtained the second connection in the second step, subproblem (ii) now requires a substitution σ_2 such that the two literals ***¬MTLx*** and **MTLsocrates**, after application of σ_1 and σ_2, become complementary. Obviously $\sigma_2 = \emptyset$ since σ_1 alone is sufficient in this particular case, thus completing the proof already presented in Section 2.

In general, we proceed this way considering in the nth step two literals, L_1 and L_2, and the composition $\sigma_{n-1}, \sigma_{n-2}, \ldots, \sigma_1$ of the substitutions obtained in the previous steps (substituting terms for variables) and test for a substitution σ_n such that $\sigma_n L_1' = \sigma_n \neg L_2'$ where $L_i' = \sigma_{n-1} \cdots \sigma_1 L_i$ for $i = 1, 2$. This problem of determining σ_n is known as the *unification* problem [19] which has been thoroughly studied with fast solutions (running in linear time) for the general case. We do not discuss any of these unification algorithms in detail (See Sections III.5 and IV.9 in [7].) assuming that the reader will grasp somewhat of their nature from further examples. But we must point out that their application is subject to an essential restriction.

Consider any predicate Q with two arguments and the formula

$$\forall c\ \exists x\ (\overbrace{Qcx \rightarrow Qxc})$$

Applying the connection method this turns out to be a theorem since the substitution $\sigma_1 = \{x \leftarrow c\}$ solves subproblem (ii). Let us now exchange the two quantifiers to yield

$$\exists x\ \forall c\ (\overbrace{Qcx \rightarrow Qxc})$$

Simply by reading the formula as a statement with his natural language, the reader will see that this cannot be a true statement for arbitrary Qs although our method, as described thus far, would result in the same proof with $\sigma_1 = \{x \leftarrow c\}$. Of course, σ_1 cannot be a correct solution since according to the formula the existence of object x is claimed *independent* of the choice of c while σ_1 would suggest a *dependent* solution.

Standard proof methods take care of that by introducing so-called *Skolem functions* for each $\forall$ quantifier with the dominating $\exists$-quantified variables as arguments. In this example, $\exists x$ dominates $\forall c$ in the (tree) structure of the formula which is expressed briefly by $x < c$; hence, c is replaced by fx to yield $\exists x\ (\overbrace{Q(fx)x \rightarrow Qx(fx)})$ where f is any new function symbol. Obviously, this prevents the two connected literals from being unified since for no substitution can x and fx yield the same term.

As an alternate solution to this, one may regard $c = \sigma_1 x$ as a tree-ordering relation $c <\cdot x$. Such a substitution is then called *acceptable* if the transitive closure $\lhd$ of the union $< \cup <\cdot$ of the relations $<$ and $<\cdot$ has no cycles. In this sense, σ_1 for our example is not acceptable since $c <\cdot x < c$ obviously leads to the cycle $c \lhd c$. Thus, all we need do is restrict unification to acceptable substitutions; this has several technical advantages over the solution via Skolem functions. Namely, some fast unification methods, after having introduced Skolem functions, construct $\lhd$ from them—hence, their introduction is in fact redundant—and test for cycles anyway. (See IV.9.1 in [7].) Also, our solution fits elegantly with what will be discussed in later sections.

We complete the description of the most basic aspects of the connection method by pointing out that it requires (selective) backtracking whenever an acceptable substitution does not exist as in the following example:

$$\forall b\ \exists x\ \forall c\ (\overbrace{Q(fcx)x \rightarrow Qxc \vee \exists y\ Qyb})$$

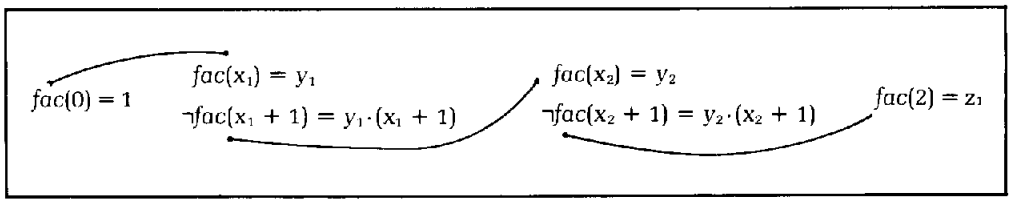


FIGURE 2. An Example of Amplification.

Here, the dotted connection has no acceptable substitution. Hence, SSC has to back up and consider the alternate (fully lined) connection which yields the proof with the acceptable substitution

$$\{x \leftarrow b,\ y \leftarrow fcx\}$$

At this point, it is appropriate to complete the comparison with Andrews' work. As in the standard resolution methods, he uses Skolem functions. His ground-level procedure is less elaborate than our full version of SSC. Finally, his method lacks the features described in the following two sections.

5. IMPLICIT AMPLIFICATION

Consider the formula

$$fac(0) = 1 \wedge \forall xy\,(fac(x) = y \rightarrow fac(x + 1) = y\cdot(x + 1)) \rightarrow \exists z\, fac(2) = z$$

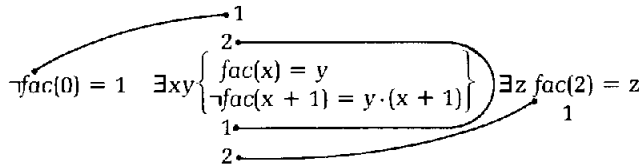
It claims the existence of a value for the factorial function for the argument 2 which obviously is a valid statement. As before, we prefer the matrix representation which is

$$\neg fac(0) = 1 \quad \exists xy\left\{\begin{matrix} fac(x) = y \\ \neg fac(x + 1) = y\cdot(x + 1) \end{matrix}\right\} \quad \exists z\, fac(2) = z$$

An attempt to prove this theorem with the connection method as described in the previous sections will fail, however, since this description is still incomplete in an essential detail. Namely, according to the nature of FOL, we must allow for an arbitrary number of independent copies of any $\exists$-quantified part in the matrix (which explains why no decision procedure exists for FOL). Adding further such copies is called *amplification* by Andrews. For our example, a proof is obtained with one additional copy of the second clause as shown in Figure 2.

Here, the copies are distinguished by indices added to the variables. For instance, x and y from the original second clause are replaced by x_1 and y_1 in its first copy and by x_2 and y_2 in its second copy, respectively. Although only a single copy of the rightmost clause is needed so that a replacement is not actually required here, z has nevertheless been replaced by z_1 for reasons of uniformity.

Evidently, the three depicted connections are spanning. The corresponding acceptable substitution is $\sigma = \{x_1 \leftarrow 0,\ y_1 \leftarrow 1,\ x_2 \leftarrow 1,\ y_2 \leftarrow 1,\ z_1 \leftarrow 2\}$; for instances, σ unifies (here in a slightly generalized sense as the alert reader might notice) the respective pairs of terms corresponding to the second connection, viz., $\sigma(x_1 + 1) = (0 + 1) = 1 = \sigma x_2$ and $\sigma(y_1\cdot(x_1 + 1)) = 1\cdot(0 + 1) = 1 = \sigma y_2$. Note that for the "answer" variable, z_1, this proof yields the expected result of computing the factorial of 2, viz., $\sigma z_1 = 2$. We now encode this proof within a single copy of the original matrix in the following way.



This representation is obtained from the previous one simply by projecting the two copies of the second clause into the single original one, while the information concerning their differences is encoded in the indices now labeling the nodes of the connections (each by definition, given in Section 1, referring to the adjacent literal). For instance, there are two such nodes adjacent to the literal $fac(x) = y$, one labeled with 1, the other with 2.

It is obvious, for this example, that all information contained in the previous *explicit* presentation (with the explicitly added second copy) can be recovered from this *implicit* encoding; conversely, the implicit representation is also uniquely determined by the explicit one. This one-to-one correspondence between these two kinds of representations in fact holds in general. Hence, the connection method need not explicitly generate such copies of $\exists$-quantified formula parts, since the illustrated simple indexing technique serves the same purpose. The technical and rather complicated details realizing this intuitively simple idea may be found in the Sections III.6 and IV.8 of [7], while a version of our algorithm SSC, adapted to the present complication, is spelled out in III.7.2 of [7].

This kind of connection proof in implicit representation may easily be generalized to arbitrary input $n \geq 1$ for the present example (as well as for others, of course) by introducing *schemes* of connections. This is illustrated as follows, this time using again the original formula structure (Figure 3).

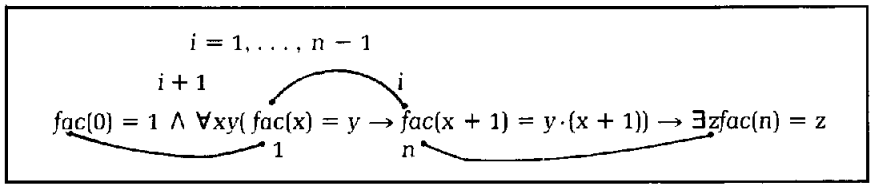


FIGURE 3. An Example of a Connection Scheme.

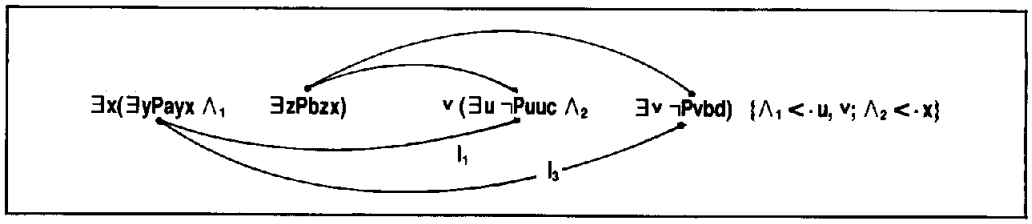
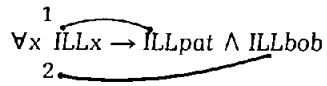


FIGURE 4. A Theorem and its Proof with the Connection Method.

It represents both the natural specification of the problem to compute the value of the factorial for input n (viz., the logical formula) and a proof, respectively a program, determining how to compute this value efficiently (viz., the connection scheme). Note that specification and program are neatly separated, whereas in conventional programs the logical and dynamic structure are mixed together. It is this mixture which causes so many problems in software production. Such a *logic program* (i.e., formula + scheme) may be compiled automatically as usual to run as efficiently as any conventional program computing *fac*(*n*). (For a discussion of the close connection between these kind of proofs and programs, and for further information concerning the predicate logic as a suitable programming environment, see [8], or Section V.2 in [7], and the references given there, such as Reference 3 in [8]).

6. SPLITTING BY NEED

Consider the formula



presented together with its proof along the lines of the previous section. This looks like a satisfactory solution but, in fact, has a major drawback. Imagine that the three literals are replaced by matrices of considerable size with many more required connections. Then, the implicit generation of a *second* copy of the matrix to the left of the implication will certainly increase the search space for appropriate connections substantially since in particular connections between the two copies have to be taken into consideration as the previous example has demonstrated. The second copy, however, is not actually required (not even implicitly) since the two conclusions are independent from each other because of the separating conjunction; hence, the search for the proof may take advantage of what is called *splitting* in conventional methods and is applied there in a processing step.

Here, we briefly discuss an alternate form of *splitting by need* which is more flexible and more general than previous splitting methods, and which elegantly fits into the connection method.

Recall the relations $<$ (tree structure of formula), $\lessdot$ (substitutional structure), and $\lhd$ (transitive closure of $< \cup \lessdot$), introduced in Section 4. In order to account for splitting, $\lessdot$ will now serve an additional purpose. Namely, consider the situation where the proof has located the first connection with $\sigma_1 = \{x \leftarrow pat\}$ in the above example, and is now considering the second one, still with a single copy of ILLx in mind. The unification fails since $\sigma_2 = \{x \leftarrow bob\}$ is apparently not compatible with σ_1. At this point, i.e., when the need arises rather than in advance, the process may consider the possibility of splitting (among other possibilities such as alternate connections or further copies) the way to be explained below.

Splitting means to consider separable subcases separately. This, in turn, requires the separation (represented by $\wedge$) to be carried out *before* processing each subcase, in particular before performing the substitutions on x. Instead of carrying out this separation explicitly, this requirement is encoded in our method as a condition on the relation $\lessdot$, hence also on $\lhd$. In this example, this condition reads $\wedge \lessdot x$, where $\wedge$ refers to the particular occurrence of conjunction in the formula. With this notation, the definition of an *acceptable* substitution can be extended in a straightforward way such that it covers this intended meaning of $\wedge \lessdot x$. (For the awkward technical details, see Section IV.10 in [7]). Thus, both σ_1 and σ_2 would be acceptable in the present context, resulting in the following proof with only a single (explicit or implicit) copy of ILLx:

The merits of this sophisticated technique become more visible in more complicated problems, the proofs of which yield more than just one such conditions. This is illustrated in Figures 4 and 5 with an example that (for reasons of space) is artificially elaborated but shows the effect as it may well occur in complicated theorems.

Figure 4 shows the four-step proof with the connection method involving no search at all. It leads to three such conditions shown in the figure. On their basis, the substitutions determined by the depicted connections, in fact, become acceptable. Even without the formal definition at hand, this can be verified by the reader on the basis of the intended interpretation of these conditions as splittings. Here we have two splits, namely, one on $\wedge_1$ and one on $\wedge_2$. The conditions imply that the latter has to precede the former since the condition $\wedge_2 \lessdot x$ together with $x < \wedge_1$ (given by the formula structure) implies $\wedge_2 \lhd \wedge_1$, which expresses exactly that precedence. Performing these splits in that sequence leads to four separate subproblems each of which can be proved immediately (an easy exercise which is left to the reader). But note the essential point that the method itself does *not* carry out any such splits since the test for cycles (present in unification anyway as explained in Section 3), now incorporating the extended relation $\lessdot$, takes care of that in a very efficient way.

For comparison, Figure 5 shows an optimal eight-step proof with resolution for the same formula. For resolution fans, it is a healthy exercise to attempt it by hand *before* looking at the picture since in relation to the small size of the problem it involves considerable search. The comparison is fair since, for resolution, further splitting is excluded if a slight change is made in the example which, in contrast, has no effect on the connection method:

$\exists xw\ (\exists y\ Payxw \wedge \exists z\ Pbzxw) \vee \exists s\ (\exists u\ \neg Puucs \wedge \exists v\ \neg Pvbds)$

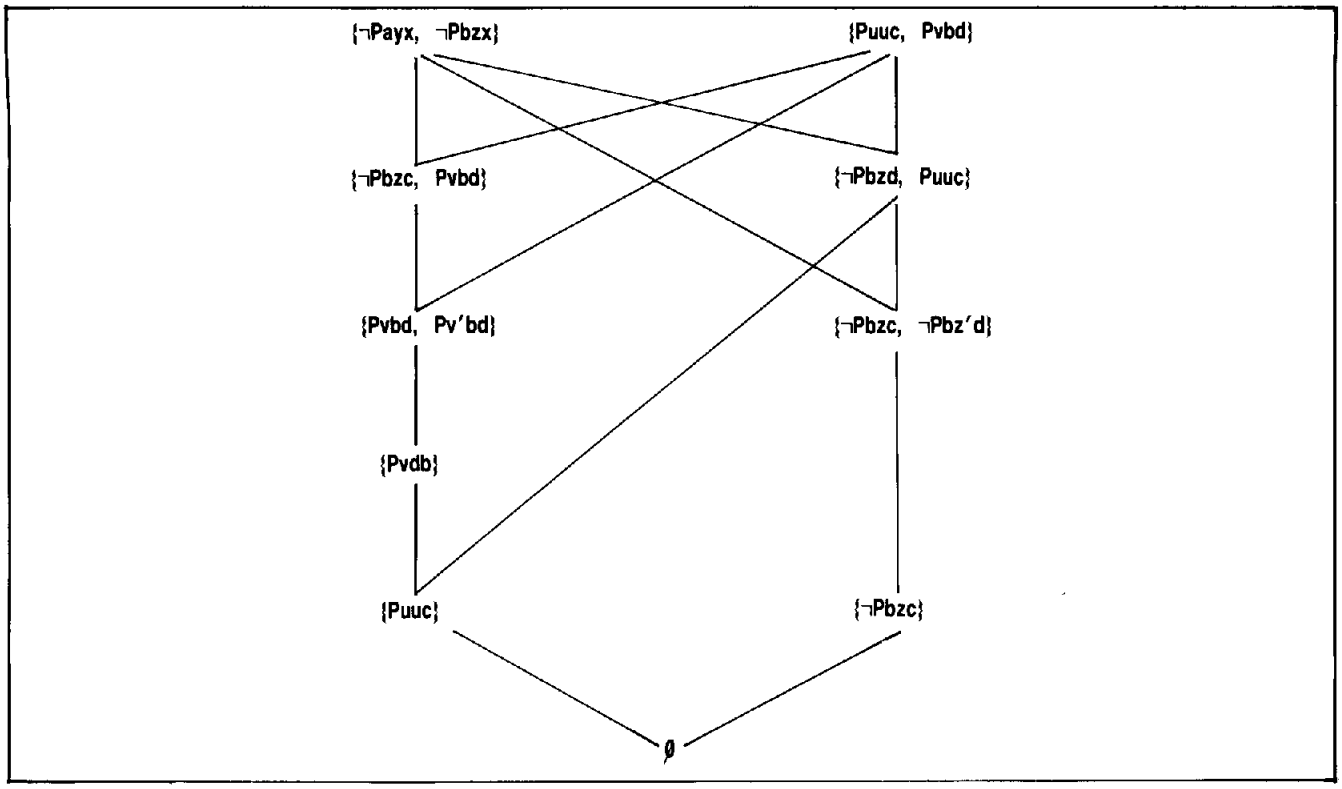


FIGURE 5. The Resolution Derivation of the Theorem in Figure 2.

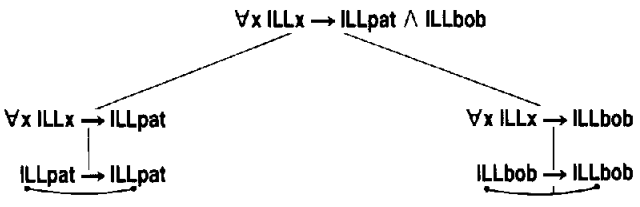
Remember the proof rule introduced for the connection method in Section 3. Here, and in Section 4, we have seen that each step not only adds a connection, w, to formula A but possibly also extends the relation $\lessdot$ to $\lessdot'$ which initially is empty. Hence, its complete form is $(A, W, \lessdot) \vdash (A, W \cup \{w\}, \lessdot')$, where A never changes.

7. NATURAL DEDUCTION PROOFS

The results reported in this paper were inspired by a thorough study of natural deduction proofs [12]. It is not surprising, then, that the connection proofs we describe are closely related to natural deduction proofs which may even be printed out immediately once the connection proof has been achieved. In this transformation, the relation $\lessdot$ provides the sequence of the deduction steps, and the connections encode both the instantiated terms (via unification) and the property qualifying the formulas at the leaves of the deduction tree as logical axioms. This is illustrated in Figure 6 with the natural deduction proof corresponding to the (second) connection proof for the first example of the previous section. Note that the $\wedge$ splitting is performed prior to the instantiation of the terms *pat* and *bob* for x due to the information given by $\wedge \lessdot x$ which reveals the proper nature of this relation. (For the natural deduction proof obtained from the proof in Figure 2, see Figure 3 in [5].)

In Section 2, we already pointed out the close relation between the connection method and resolution (or its refine-

FIGURE 6. The Natural Deduction Proof Obtained from the Connection Proof ∀x ILLx → ILLpat ∧ ILLbob, ∧ <· x.



ments) which was studied in detail [4]. Here, we now see its close relation with natural deduction proofs. This justifies regarding the connection method as a *higher level* proof method; it provides a deeper insight into the characteristic features of theorems (versus formulas) than do the resolution or the natural deduction proof method.

8. SUMMARY

In this paper, we have given an overview of a proof method for theorems in first-order logic called the *connection method*. With a number of examples, we have illustrated a number of its prominent features. These and further ones are now summarized.

(*i*) It may be regarded as a higher level proof method providing deeper insight into the characteristic features of theorems (versus formulas) than do conventional proof methods (see Section 7).

(*ii*) It operates exclusively on a single copy of the given formula (see Sections 3 and 5, and the proof rule at the end of Section 6).

(*iii*) In particular, no transformation into any normal form of the given formula is required which allows the use of any standard logical connectives and the speeding up of search by antiprenexing (see Section 2, and [7], p. 209).

(*iv*) Not even Skolem functions need be introduced, *i.e.*, only the terms in the original formula need be unified (see Section 4).

(*v*) Unification is generalized to include an optimal splitting by need (see Section 6).

(*vi*) Amplification is accomplished by a systematic indexing without violating feature (*ii*) (see Section 5).

(*vii*) A connection proof can easily be transformed into a natural, thus immediately comprehensible deduction of the given formula (see Section 7).

(*viii*) Special handling of equality and other algebraic relations (extensively studied under the key word "rewrite rules") smoothly fits into the connection method, since from its perspective such specialized rules simply encode a deterministic control for locating connections, possibly a whole scheme of connections within one step (see Sections V.3 and 4 in [7]).

(*ix*) Induction may appropriately be incorporated following the standard—or a new, in fact promising—line (see Section V.5 in [7]); the generalization of the method to higher-order logic (or type theory) is similarly straightforward (see Section V.6 in [7]); the same may be expected for other extensions of FOL such as modal logic.

(x) The location of connections in a given formula structure has a potential of being realized on the chip level.

All these features have been developed by the author in a number of technical papers. (See [3–6] and their references.) A number of them were also obtained by Andrews in an independent approach with *matings*. (See [1–2] and their references.) This paper has clarified this relationship.

Due to its intended nature, this paper has not provided any technical details such as precise definitions, theorems, proofs or other justifications, and algorithms. For these details, the interested reader is referred to the references mentioned before or to the comprehensive treatment given in [7].

Acknowledgments. I thank Peter Haddaway, Alex Kumjian, Bernard Meltzer, and the referee for numerous valuable suggestions for improving this text.

REFERENCES

1. Andrews, P.B. Theorem proving via general matings. *J. ACM 28*, 2 (April 1981), 193–214.
2. Andrews, P.B. Transforming matings into natural deduction proofs. In W. Bibel, R. Kowalski, eds. *Proc. 5th Conf. on Autom. Deduction*, LN in *Comp. Sci. 87* (Springer, Berlin, 1980), 281–292.
3. Bibel, W. On matrices with connections. *J. ACM 28*, 4 (October 1981), 633–645.
4. Bibel, W. A comparative study of several proof procedures. *Artif. Intell. 18*, (1982), 269–293.
5. Bibel, W. The complete theoretical basis for the systematic proof method. Bericht ATP-6-XII-80, Institut für Informatik, TUM (1980). Submitted to J. ACM.
6. Bibel, W. Computationally improved versions of Herbrand's theorem. In Stern, ed. *Proc. of the Herbrand Colloquium*, North Holland, Amsterdam, 1982), 11–28.
7. Bibel, W. Automated theorem proving. Vieweg: Braunschweig, 1982.
8. Bibel, W. Syntax-directed, semantics-supported program synthesis. *Artif. Intell. 14*, (1980), 243–261.
9. Bibel, W., and Kowalski, R., eds. *5th Conference on Automated Deduction*, LN in *Comp. Sc. 87* Berlin, Springer, 1980.
10. Bledsoe, W.W. Non-resolution theorem proving. *Artif. Intell 9*, (1977), 1–35.
11. Chang, C.-L., and Lee, R.C.-T. Symbolic logic and mechanical theorem proving. New York: Academic Press, 1973.
12. Gentzen, G. Untersuchunge über das logische Schließen I, *Mathemat. Zeitschrift* 39, (1935), 176–210.
13. Herbrand, J. Recherches sur la Theorie de la Demonstration, Traveaux de la Societé des Sciences et des Lettres de Varsovie, Classe III sciences mathematiques et physiques, *33* (1930).
14. Joyner, W., (ed.) 4th Workshop on Automated Deduction (Austin, Texas, 1979).
15. Kalish, D., and Montague, R. Logic. New York: Harcourt Brace Jovanovich, 1964.
16. Loveland, D.W. *Automated theorem proving*. Amsterdam: North Holland, 1978.
17. Loveland, D.W. (ed.) 6th Conference on Automated Deduction, LN in *Comput. Sc. 138*, Berlin: Springer, 1982.
18. Nilsson, N.J. Principles of artificial intelligence. Palo Alto: Tioga, 1980.
19. Robinson, J.A. Logic: form and function, Edinburgh University Press (1979).
20. Siekmann, J. (ed.) GWAI-81, Informatik Fachberichte 47. Berlin: Springer, 1981.
21. Wos, L. Solving open questions with an automated theorem-proving program, in [17].

CR Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*mechanical theorem proving, logic programming*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*deduction, resolution, logic programming*; I.2.4. [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*predicate logic*

General Terms: Theory, Algorithms

Additional Key Words and Phrases: logic, theorem proving, connection method, matings, resolution, unification, splitting, natural deduction, systematic proof procedure, complementary matrices, spanning sets of connections, structure sharing, refinements of resolution.

Received 3/81; revised 2/83; accepted 4/83