A Survey of Automated Deduction

Woody Bledsoe
University of Texas
Computer Sciences Department
Taylor Hall 2.124
(512) 471-9568
ai.bledsoe@r20.utexas.edu

Richard Hodges

January 29, 1988

Abstract

DRAFT

This is an enlarged version of a survey talk given by Woody Bledsoe at the AAAI National Conference, Seattle, Washington, July 16, 1987 and will appear in a collection of survey talks published by Morgan Kaufmann, Los Altos, CA.

Contents

[1]now at Univ. British Columbia

1 INTRODUCTION

What is Automated Deduction?

It includes many things. A part of it involves *proving theorems by computer*, theorems like the Pythagorean theorem from Plane Geometry (Figure 1) or the theorem: If an equilateral triangle is inscribed in a circle, and lines are drawn from its corners to a point on the circumference, then the length of the longest such line is equal to the sum of the lengths of the others. (Figure 1.)

Figure 1 near here.

Or theorems from algebra such as:

> A group for which $x^2 = e$ for each of its elements x, is commutative.
>
> A ring for which $x^3 = x$ is commutative.

Or theorems from analysis such as the *maximum value theorem* and the *intermediate value theorems*, depicted in Figure 2:

Figure 2 near here.

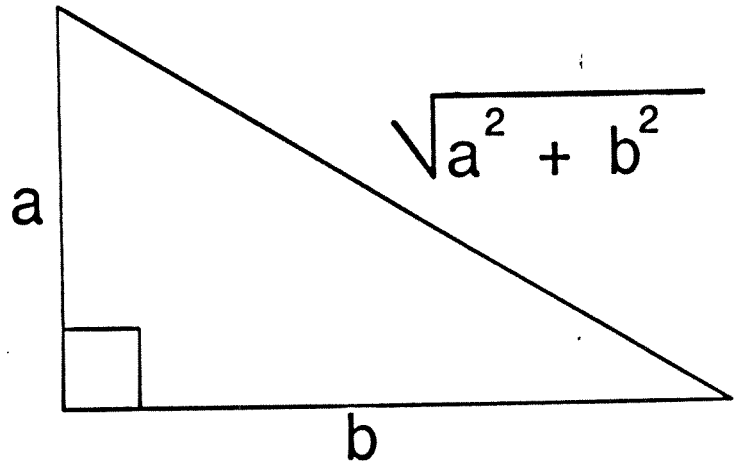
> A Continuous Function f defined on a closed interval $[a, b]$, attains its Maximum (and Minimum) on that interval.
>
> And if $f(a) < 0$ and $f(b) > 0$, then $f(x) = 0$ for some x in $[a, b]$.

Also puzzles such as the *truthtellers and liars* one, can be solved by theorem proving. See [LO85].

> On a certain island the inhabitants are partitioned into those who always tell the truth and those who always lie. I landed on

EXAMPLE THEOREMS FROM GEOMETRY

Pythagorean Theorem:



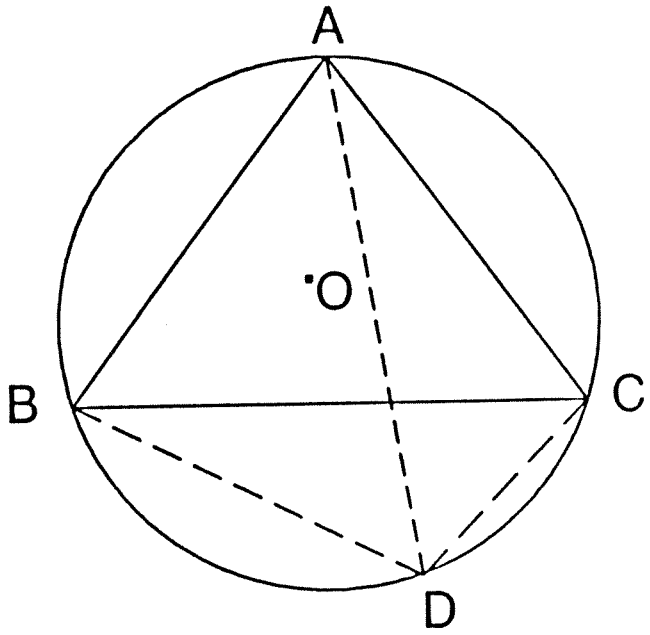
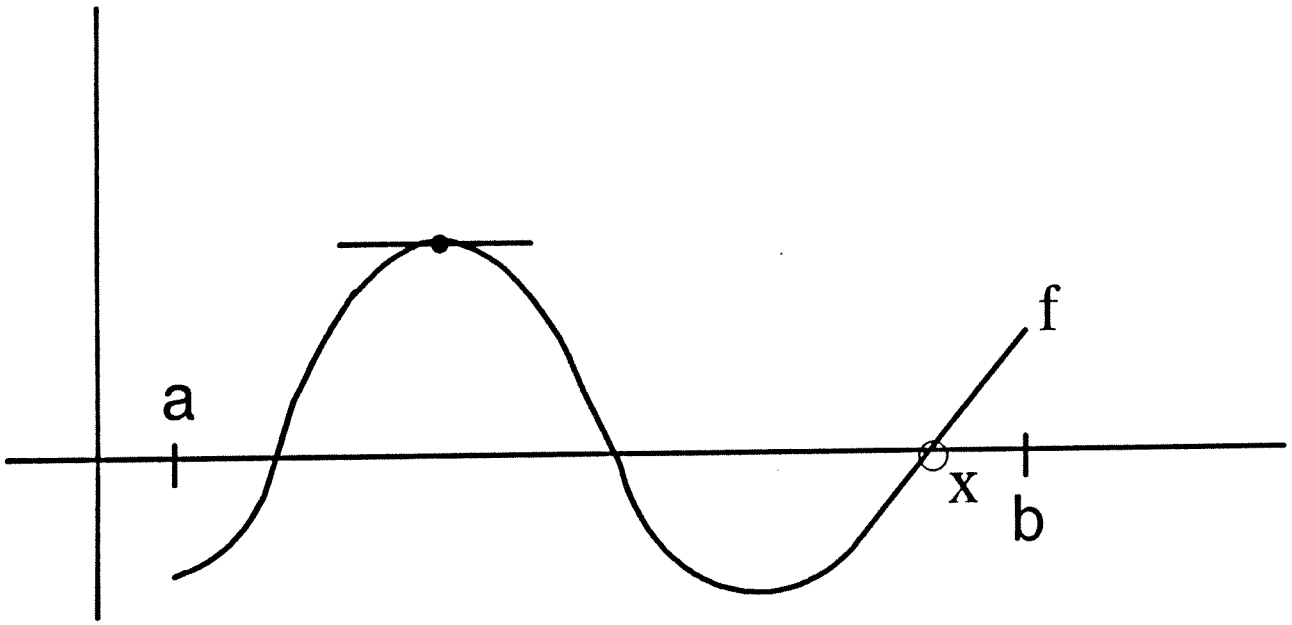


Figure 1

EXAMPLES FROM ANALYSIS



Maximum Value Theorem and Intermediate Value Theorem, for continuous functions.

Figure 2

the island and met three inhabitants A, B, and C. I asked A, 'Are you a truthteller or a liar?' He mumbled something which I couldn't make out. I asked B what A had said. B replied, 'A said he was a liar'. C then volunteered, 'Don't believe B, he's lying!'.

What can you tell about A, B, and C?

The halting problem theorem (figure 3) shows how complicated these theorems can get, and others more so.

Figure 3 near here.

1.1 Facets of Automated Deduction

What is Automated Deduction? It is a number of things. But in all cases one is *making deductions* by computer. It is often called *Automated Theorem Proving* (ATP), or *Automatic Reasoning (AR)*. We will use these terms interchangeably.

Let me list some of the *facets* and *applications* of Automated Deduction. See Figure 4.

Figure 4 near here.

We consider *proof discovery* to be the major component of ATP, because every application of ATP uses some amount of automatic proof discovery. We will tend to concentrate on it in this talk, since we are personally interested in it, and will discuss the others only briefly, if at all. There are a number of review papers and references for each of these areas. One might add to this list: *all non-numeric programming*, since some form of inferencing is involved in all of it.

Automatic proof checking is a very important part of AR (see, for example, [BM82, Con85, Hun85, We77]) but will be discussed only briefly here. The reader is referred to [MS84] for a report on using ATP in CAI.

HALTING PROBLEM IS UNSOLVABLE
(Burkholder)

(1) $(Ex)[Gx \,\&\, (Ay)(Py \rightarrow (Az)Dxyz)] \rightarrow (Ew)[Pw \,\&\, (Ay)(Py \rightarrow (Az)Dwyz)]$

(2) $(Aw)([Pw \,\&\, (Ay)(Py \rightarrow (Az)Dwyz)] \rightarrow (Ay)(Az)([Py \,\&\, H_2yz) \rightarrow (H_2wyz \,\&\, Owg)]$
$\&\, [(Py \,\&\, \neg H_2yz) \rightarrow (H_3wyz \,\&\, Owb)]))$

(3) $(Ew)[Pw \,\&\, (ay)([Py \,\&\, H_2yy) \rightarrow (H_2wyy \,\&\, Owg)] \,\&\, [(Py \,\&\, \neg H\ yy)$
$\rightarrow (H_2wyy \,\&\, Owb)])] \rightarrow (Ev)[Pv \,\&\, (Ay)([Py \,\&\, H_2yy) \rightarrow (H_2vy \,\&\, Ovg)]$
$\&\, [(Py \,\&\, \neg H_2yy) \rightarrow (H_2vy \,\&\, Ovb)])]$

(4) $(Ev)[Pv \,\&\, (Ay)([Py \,\&\, H_2yy) \rightarrow (H_2vy \,\&\, Ovg)] \,\&\, [(Py \,\&\, \neg H_2yy) \rightarrow (H_2vy \,\&\, Ovb)])]$
$\rightarrow (Eu)[Pu \,\&\, (Ay)([(Py \,\&\, H_2yy) \rightarrow \neg H_2uy] \rightarrow \neg H_2uy] \,\&\, [(Py \,\&\, \neg H_2yy)$
$\rightarrow (H\ yy) \rightarrow (H_2uy \,\&\, Oub)])].$

$\longrightarrow$

(5) $\neg(Ex)\,[G\,x \,\&\, (Ay)\,(Py \rightarrow (Az)\,DxyZ)]$

Figure 3

APPLICATIONS OF ATP

Proof Discovery

Proof Checking: Including Computer-Aided Instruction

Interactive Provers (Man-machine)

Logic Programming & Programming Languages

Deductive Data Bases

Program Verification & Automatic Programming

Expert-Systems Inferencing

Algebraic Manipulation (such as Macsyma)

Proof Representation & Manipulation

Figure 4

We will also not discuss *interactive provers*, but consider this to be one of the most important areas of ATP. See [BBr73, BM79,].

We will discuss *logic programming* shortly. Many efforts are underway to combine logic and functional programming languages such as PROLOG and LISP, and to join this with *rapid type inheritance*, to make it easier to write AI applications, and attain greater speed. See, for example, [AN85].

In the near future we expect to see an increased research effort on *deductive data bases*, especially for very large collections of *facts* and *rules*, written in logic, and requiring *a great deal of inferencing* to answer a query. See [GM78] for a review and also [HN84] for an example of *compiling DB* queries, to speed up retrieval.

Such a DB might contain the facts about a corporation and its operating "rules". Similarly for a political situation, such as the Middle East (will country X cut off the oil or go to war), and for military situations. We believe that a structured knowledge base of *general* (common-sense) knowledge, such as [Le86], will play a big role in these efforts.

Program verification (e.g., [Good85, BM79]) and automatic programming [MW85] continue to be significant application areas for ATP. Algebraic manipulation [Buch83], as represented by MACSYMA [MAC] and other systems, has grown to be a sizable part of AR.

Of most interest to the AI community is automatic inference associated with Expert Systems and related "intelligent" programs. In this conference alone there were 46 papers (out of 150) related to automatic reasoning. We expect that trend to continue, especially as AI programs are being based more on traditional logic and extensions of it. Here we could include *non-monotonic* reasoning (e.g., circumscription) [McC80] Truth Maintenance [Do79, deK84], common-sense reasoning [McC, Le86], qualitative reasoning, (see, for example, [deK84, Fo84, Ku86]), meta Reasoning [GGS83, GN87].

1.2 Proof Representation & Manipulation

Another branch of automated deduction studies methods of representing and transforming proofs. Human mathematicians seem to be able to understand a proof as a whole, wheras automated deduction systems tend to have a very narrow view, centered around a single clause or a small group of clauses at any one time.

One reason for wanting to be able to manipulate proofs is to facilitate higher-level strategies for proof discovery. The method of proof by analogy is an area which needs the ability to transform proofs, to extract the abstract content of a proof, and to annotate proofs with additional information such as the "motivation" for a given step. (See Section 5.2.1).

The internal representations used in automated deduction are often not very easy for people to understand. Many theorem provers use clausal resolution. But putting a theorem into clauses often introduces redundancy and obscures the logical structure of the theorem and its proof. Observing that it is often much easier to understand a proof in natural deduction format, Peter Andrews and Dale Miller have developed algorithms for transforming resolution proofs into an intermediate form called an "expansion tree" and then into a natural deduction proof [An81]. Amy Felty, a student of Miller, has recently developed a system to translate proofs into natural English. These systems use "higher order logic" (see section 3.3) and have automatically proven Cantor's theorem and a version of Russell's paradox.

A group of Systems [GMW82, Ne80, Card86, CoH85, Cons86, deB80] have been developed for representing and checking mathematical proofs using a higher order logic based on the Curry-Howard isomorphism between propositions and lambda-types (see section 3.4) These systems have also been used for verifying software and hardware [G087]. Proofs often can be written in a form much closer to that used by a human mathematician than by employing first-order predicate calculus and resolution. So far,

little work has been done on proof-discovery in these systems.

McAllester (MIT) has developed a theorem prover with set theory "built-in" and with a novel concept for proof guidance: the user specifies a "focus object" and the prover tries to forward chain from established facts to prove everything it can about the selected object. The prover can then search using patterns to see if anything useful has been proved. This seems potentially useful as a representation for motivation in proofs. His ONTIC has been used to proof-check the Stone Representation Theorem as well as others [McA87].

Weyrauch [We77, We82] has developed a system called FOL in which the syntax and reasoning rules of a deductive system can be formalized in First Order Logic. In paricular, FOL can formalize its own logic. It can conduct reasoning about proofs and about its own rules of inference. New rules can be verified using the deductive capabilities of FOL and can be added declaratively to the set of meta-theorems representing facts FOL knows about itself.

2 REFERENCES

There have been a number of excellent *review papers* of ATP during the last few years. Perhaps the review by Loveland [lo84] or [Bhe85] (in the first issue of the Journal of Automated Reasoning, 1985) would be the best for the beginner. In that same issue of JAR is an extended review of AR. Those interested in the prehistory and early history of ATP should see Martin Davis' [Da83]. Also see [WH83]. Bill Pase, of I.P. Sharp Associates, has recently revised his 70 page bibliography of Automated Deduction, which is very useful for those serious about this subject. [Pa87]

There are a number of books and collections of important papers which are introductory to the subject. For example, [CL73, Lo78, Bi82-87, Wo84, GN87, Ko79, Bu83, An86, IEEE-C25, Wo87, BM79, Sw83, BL84]. Also

there are chapters on ATP in various books on AI such as [Nil80, Rich83], and various Journals and Conference Proceedings (JAR, AAR Newsletter, CADE Reports, AI Journal, MI Series, AAAI, IJCAI, IEEE Transactions PAMI and SSC, etc.).

Other books of related interest include Konolige [Kon86] on representing the capabilities of intelligent agents with imperfect reason; and Smullyan's books of logic puzzles, especially [Smu85], a good source of challenge problems for ATP systems.

3 BRIEF HISTORY OF AUTOMATED DEDUCTION

Modern ATP was born in the middle 1950's with the "Logic Machine" of Newell, Simon, and Shaw [NSS56]. Gelernter's "Geometry Machine" [Ge59] followed in the late 50's, as well as other interesting work by Hao Wang [WaH60], Davis and Putnam [DP60] and many others (see [Da83]). But it was the advent of J. A. Robinson's RESOLUTION paper [Ro65] in 1965 that forever changed this field.

Also note that Maslov's *inverse method* [Mas68] stems from the mid 60's. (Vladimir Lifschitz has recently completed an excellent paper [Li87] simplifying the presentation of this powerful method.)

Other proof procedures, such as the so called "Natural Deduction" Provers [Wah60, Bl75, Lo78, Bl77, Pl82], Model Elimination, Connection and Mating Methods [An81, Bi82], Interconnectivity graphs [Ko75, Si76], Semantic Tableaux [Op1, Smu68], and the earlier "inverse Methods" of Maslow [Mas 68], have much in common with Resolution and also suffer many of its shortcomings.

Still, we believe that the introduction of resolution represents the single most important event in ATP so far. What is it?

3.1 Resolution

The basic idea of Resolution is simple and is very easy to learn. See, for example, the presentation in [CL73]. It is based upon the *modes ponens* rule, or more generally the *chain rule*. Referring to Figure 5, if the chain rule is converted to *clausal form* (by replacing an expression $x \rightarrow y$ by $(\neg x \vee y)$ then the rule is effected by cancelling the q and $\neg q$ in the upper clauses. Shown at the bottom of Figure 5, is the Resolvent Rule for first order logic, where *unification* is required; here the variable x is bound to the term a.

Figure 5 near here.

Figure 6 shows a resolution proof of a simple theorem. Note that the hypotheses are converted to clausal form and the conclusion is negated. Then clauses are resolved until a contraction, $\square$, is reached.

Figure 6 near here.

For Propositional Logic (where no variables are to be bound) Resolution is quite simple:

RESOLUTION RULE

1. Negate Theorem
2. Put in "Clausal Form"' (i.e., Conjunctive Normal Form, CNF)
3. Resolve until a contradiction, $\square$, is obtained

Now let us look at Resolution for First Order Logic (FOL). Figure 7 shows some expressions in FOL and a theorem. One is dealing here with quantifiers and variables. In order to prove this by resolution we must convert it to *clausal* form. (Figure 8) First each hypothesis is ***skolemized*** by removing the quantifiers.

RESOLVENT RULE

OD

MODES PONENS

$$\frac{p, \quad p \rightarrow q}{q}$$

CHAIN RULE

$$\frac{p \rightarrow q, \quad q \rightarrow r}{p \rightarrow r}$$

RESOLVENT RULE

$$\frac{\neg p \ v \ q, \quad \neg q \ v \ r}{\neg p \ v \ r}$$

$$\frac{\neg p(x) \ v \ q(x), \quad \neg q(a) \ v \ r}{\neg p(a) \ v \ r}$$

Figure 5

EXAMPLE Resolution Proof

Theorem: $[(p \rightarrow q) \,\&\, p] \rightarrow q$

Use CONTRADICTION. (Clauses)

1.	$\neg p \vee q$	4. q	1, 2
2.	p	5. $\square$	3, 4
3.	$\neg q$	"box"	

Actually:

$$\left\{\begin{array}{l}\neg p \vee q \\ p \\ \neg q\end{array}\right\} \rightarrow\rightarrow \left\{\begin{array}{l}\neg p \vee q \\ p \\ \neg q \\ q\end{array}\right\} \rightarrow\rightarrow \left\{\begin{array}{l}\neg p \vee q \\ p \\ \neg q \\ q \\ \square\end{array}\right\}$$

Figure 6

Figure 7 and 8 near here.

In the first hypothesis, the expression is true for all x and y, so we discard the quantifiers, and remember that we can replace x and y by any term we please in the proof. We also convert the implication as before. Similarly in the next hypothesis, except that we require a skolem function. For each p, there exist a z such that Mother(z, p). It is clear that z *depends* on p, so we show that dependence by replacing z my the expression $m(p)$. The conclusion is negated (since resolution uses Contradiction). The z remains a variable that also might be replaced with a term. Figure 9 shows the corresponding clauses and the derivation of $\square$ by resolution. There, x, y, p, and z are variables, and John and m are constants. The proof goes as before except that some of the variables are *bound* in the process. These bindings are called a *substitution*. The process of determining the substitution is called *unification*. Two formulas are UNIFIED (made one) in the process.

Figure 9 near here.

For example, the pair

$$P(g(x), x)$$

$$P(y, x0)$$

are unified by the substitution $[x \leftarrow x0,\ y \leftarrow g(x0)]$ (where x and y are variables and g and $x0$ are function symbols)

But the pair

$$P(g(x), x)$$

$$P(y, h(y))$$

has no unifier. Why?

The first step in trying to unify

$$P(g(x), x)$$

FIRST ORDER LOGIC

Girl (x), Female (x), Person (p)

THEOREM:

$\forall x \quad \forall y \; [\text{Mother } (x,y) \longrightarrow \text{Female } (x)] \;\&$

$\forall p \; [\text{Person } (p) \longrightarrow \exists z \; \text{Mother } (z,p)] \;\&$

Person (John)

$\longrightarrow \exists z \; \text{Female } (z)$

Figure 7

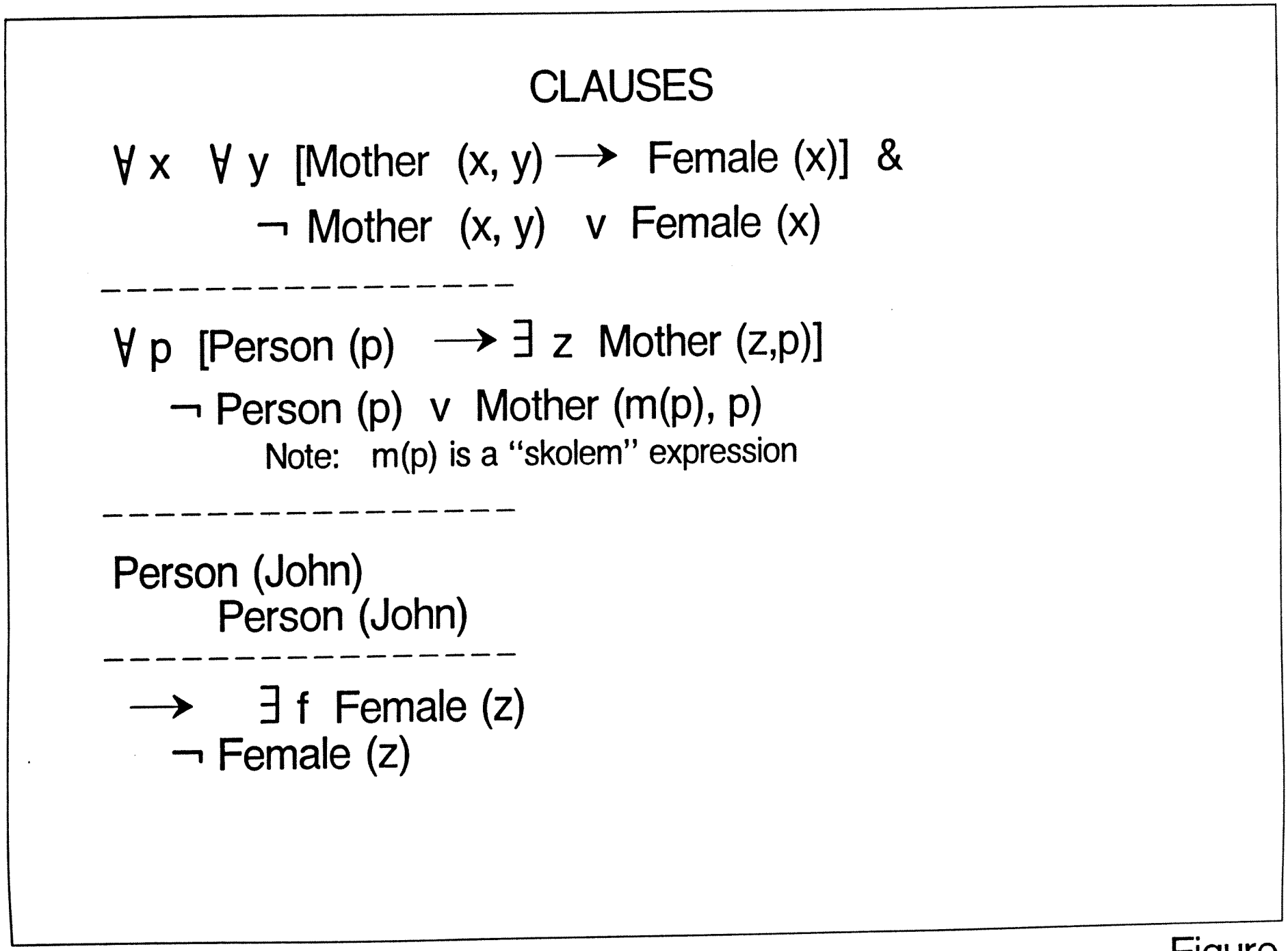

CLAUSES
∀ x ∀ y [Mother (x, y) → Female (x)] &
¬ Mother (x, y) v Female (x)
∀ p [Person (p) → ∃ z Mother (z,p)]
¬ Person (p) v Mother (m(p), p)
Note: m(p) is a "skolem" expression
Person (John)
Person (John)
→ ∃ f Female (z)
¬ Female (z)

Figure 8

PROOF

1. ¬Mother (x, y) v Female (x)
2. ¬Person (p) v Mother (m(p), p)
3. Person (John)
4. ¬Female (z)

5.	Mother (m(John), John)	3, 2, p ← John
6.	Female (m(John))	5, 1, y ← John, x ← m(John)
7.	□	4, 6 z ← m(John)

Figure 9

$$P(y, h(y))$$

yields

$$P(g(x), \quad x \quad)$$

$$P(g(x), h(g(x)))$$

But we cannot finish, because x occurs in $h(g(x))$. If we tried to continue by substituting $[x \leftarrow h\ (g(x))]$ we would get into an infinite loop. We prevent this kind of error by what is called the "occurs check" in the unification algorithm. If we don't use such occurs check, we could prove non-theorems, such as

$$\forall x\ \exists y\ P(y, x) \ \rightarrow\ \exists y\ \forall x\ P(y, x)$$

We will see more on the *occurs check* problem when we discuss *logic programming.*

Resolution is *complete* for first order logic; i.e., any theorem expressed in FOL can be proved by resolution. This is an important result since FOL includes much of mathematics (indeed, can include *all* of mathematics).

However, resolution is not a *decision procedure* for FOL, there is no guarantee that it will detect non-theorems in finite time; in fact FOL has no decision procedure. Higher Order Logic, which we will discuss shortly, has *no* complete proof procedure, let alone a decision procedure.

3.2 Completeness

Completeness is a desirable property of a proof procedure such as resolution; we want to know what it *can* and *cannot* do before we employ it. But completeness alone is not enough. We also need *speed* as well. But Resolution – as well as other proof procedure for FOL – tend to be slow when attempting the discovery of proofs of hard theorems.

We are faced with the classic *combinatorial explosion* problem when we automatically search a *proof tree*, such as the one depicted in Figure 10. The prover searches down along the branches looking for the *goal nodes*, depicted by the asterisks. Finding such a goal finishes the proof.

Figure 10 near here.

Actually, in standard resolution, the search space is not really a tree, since branches often rejoin other branches. Linear formats for organizing resolution search (such as SL-resolution, model elimination, problem reduction) make the search more tree-like. In any case, the "tree" metaphor in the following discussion is useful for intuition.

The number of branches in the tree increases at least exponentially with depth. When the solution nodes lie even moderately deep, brute-force search methods quickly exhaust available resources.

Professional mathematicians have an uncanny way of excluding much of the "brush" of the tree by heading directly toward one of these solution nodes. But the computer - though a million times faster - tends to hopelessly thrash around through all the branches (using depth first or breadth first search methods). The *challenge of this age* for this field is to *shorten* the search time. Attempts to do so can be classified into two categories.

1. Methods that speed up the *inherent reasoning process* by

   (a) Using faster *hardware*, or by

   (b) Clever programming *tricks*, such as *clause compiling*

2. Those that *prune* the search tree.

The effect of the first category is to push down a few layers in the search tree. See Figure 10A. The swath indicates how a *faster* prover might *push farther down* in the tree. This may or may not help, depending, of

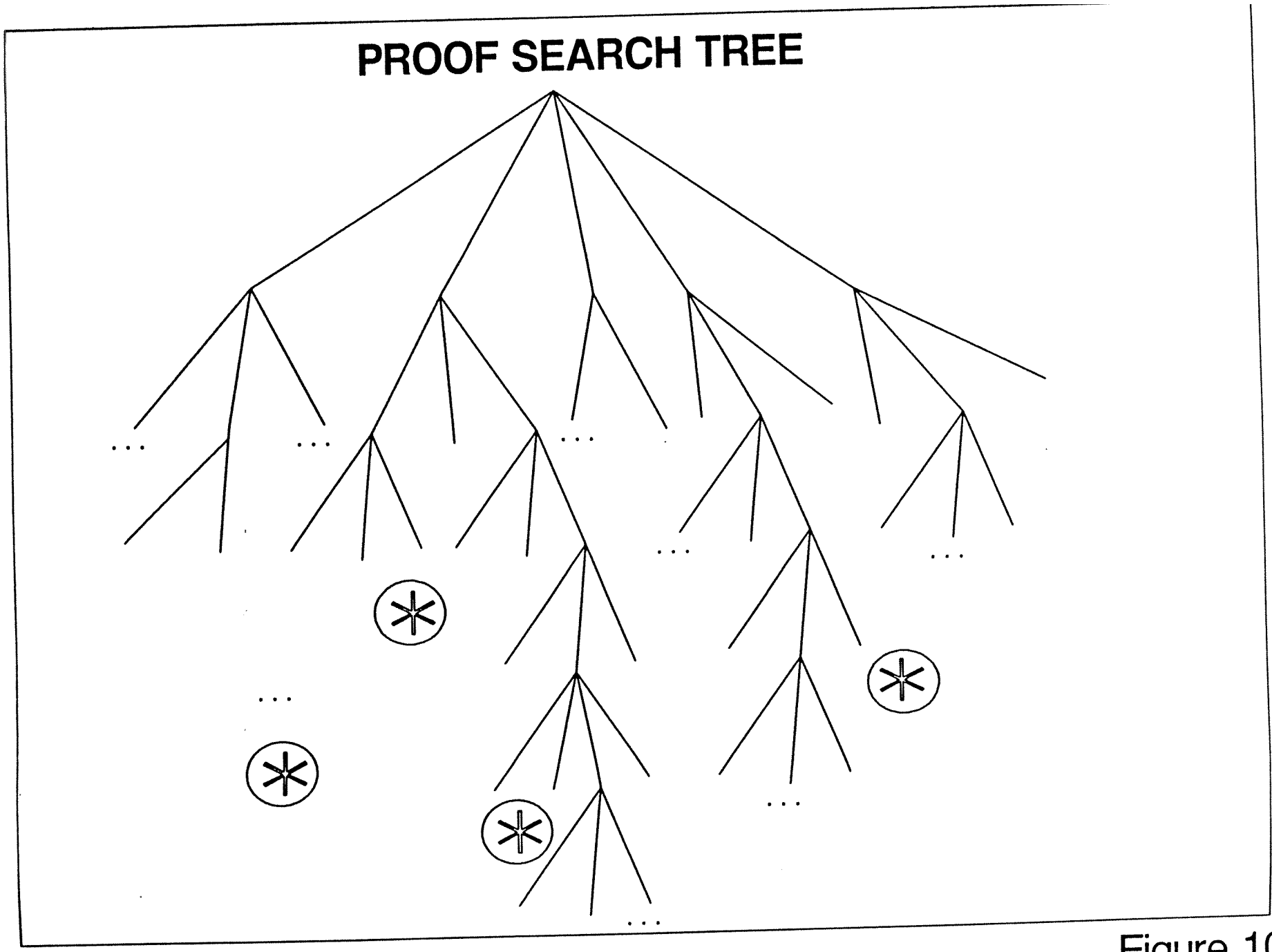


Figure 10

course, on the positions of the goal nodes in the tree. For *many applications* in AI and related fields, it *does* help. A speed up of *one* or *two orders of magnitude*, that seems to be attainable by the new *clause-compiling* techniques coming from the PROLOG community has made possible the proofs of many theorems previously unattainable by automatic methods. This is *good news* for many workers in AI who are beginning to use logic more extensively for *representing rules* for *expert systems* and for entries in *logic data bases*, etc.

Figure 10A near here.

This extended use of logic is placing a greater load on the *"inference engine"* of these systems, and these new *compiling techniques* will help greatly with that load. But it is through the *second category*, the *pruning* strategies, that we can expect satisfactory solutions for the long run. *speed alone* cannot replace the judicial use of *knowledge*. (See our recent paper, *Some Thoughts on Proof Discovery* [Bl86], for a further articulation of this argument.) There were many early attempts to *prune* the search tree. Most of these are *syntactic* in nature, applying equally well from one subfield to another. Some refinements of Resolution to speed up proof discovery are:

- Set-of-Support Resolution
- Hyper-resolution
- SL-resolution (=Model Elimination)
- Connection Method, Matings
- Interconnectivity Graphs
- Locking
- Dozens more.

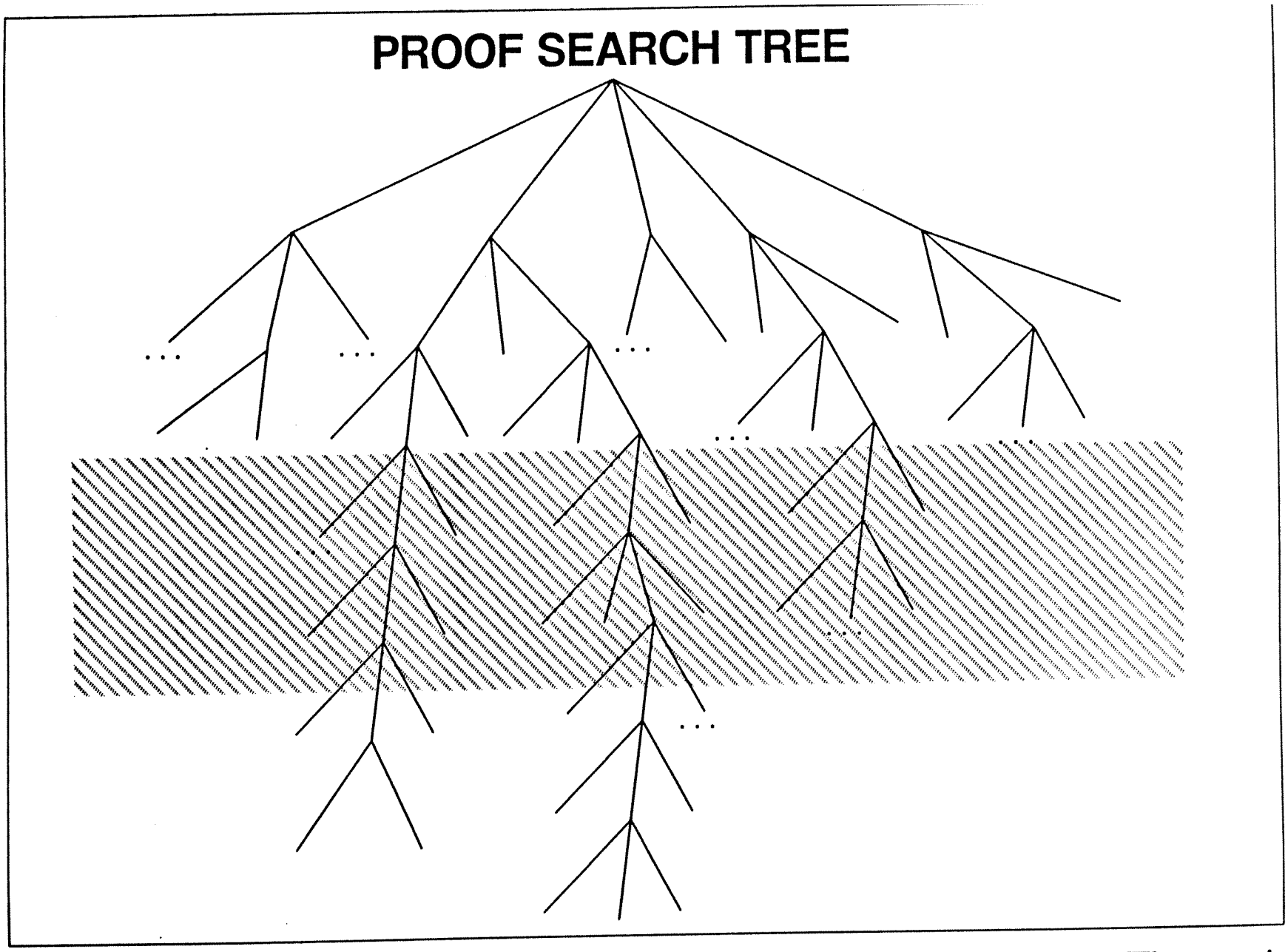


Figure ~~11~~ 10A

One such method, an important one, is the *set of support* strategy [We70], whereby the program works back from the desired goal, and avoids generating unmotivated lemmas that may or may not contribute to the final solution.

Another important one is called *hyperresolution* [Ro65A] wherein a number of resolution steps are combined into one *larger step*, with the program keeping *only* the final resolvent and discarding the intermediate resolvents ("fragments"). (See Section 5 below). This method has been especially powerful in the hands of the *Argonne Group* headed by Larry Wos. Many other pruning strategies have been tried, but these will not be reviewed here [Ko71, Lo68, An84, Bi82, Ko75, Si76, . . .].

It should be noted the *ground* proofs (proofs in which no binding of variables takes place) are hardly ever *difficult.* It is only when we allow the *binding* of *variables* (i.e., the replacement of variables by other terms), through the *unification* process, that we encounter the *combinatorial explosions* that so hamper our provers. There have been developed ground provers which are *enormously fast*, and it is questionable whether further progress in this area is necessary.

We will return to the problem of speeding-up proof discovery shortly, but we first briefly discuss other logics and equality.

3.3 Higher Order Logic

In first order logic we do *not* quantify *function* symbols, *predicate* symbols, or symbols representing *higher* order objects. For example, the formula

$$\forall a\ [\forall x\ P(x)\ \rightarrow\ P(a)] \tag{1}$$

is from the first order logic because only the a and x are quantified. But the formula

$$\forall a\ \exists Q[\forall x\ P(x)\ \rightarrow\ Q(a)] \tag{2}$$

is not, because the predicate symbol Q has been quantified. [2]

Actually (2) is an *easy theorem* for people or machines: we simply replace "Q" by "P", and "x" by "a", but it is part of Higher Order Logic (HOL), which is not even complete, let alone decidable. Inherently, HOL is *harder* than FOL. However, the methods of Unification and Resolution have been extended HOL [Hu73, An84] with a certain amount of success. For example, Andrew's Prover, based on the Huet Unification Algorithm has proved

> **Cantor's Theorem**: If N is the set of integers, and SN is the set of subsets of N, then there is no one-to-one function from N to SN.

More difficult theorems, such as

> **Intermediate Value Theorem**: If f is a continuous function on a non-empty closed interval $[a, b]$, $f(a) < 0$, and $f(b) > 0$, then $f(x) = 0$ for some x in $[a, b]$. (Using the Least Bound Axiom.)

have been proved by special purpose provers such as the one described in [Bl79], but that Prover has limited generality. General Purpose Provers tend to be SLOW, especially for HOL.

3.3.1 Propositions as Types

An interesting approach to HOL has been developed from the so-called Curry-Howard isomorphism. This is an elegant relationship between the typed lambda-calculus and intuitionistic logic. It has been championed, primarily by Martin-Lof [Ma84], as a basis for abstract computer science.

[2] The predicate symbol P is also universally quantifies (implicitly) in (1) and (2), it is only when "existential' type quantifiers are used, where the quantified predicate symbol is to be replaced (bound) in the proof process, that we enter true higher order logic.

Basically, the idea is that if a proposition is viewed as a type and the proof of a proposition is viewed as an object having that type, lambda conversion is formally the same as *modus ponens*. If A and B are propositions (types) and f is a term of type B, the expression

$$(\lambda\ (x : A)\ f)$$

is a function mapping the type A into the type B. The type of this function is symbolized as $A \rightarrow B$, which can be thought of as expressing the implication $A \rightarrow B$, with the meaning that given a proof p of A, we can get a proof $(\lambda(x)\ f)\ (p)$ for B. To prove $A \rightarrow B$ means to demonstrate an object of type $A \rightarrow B$, i.e., an effective procedure for obtaining a proof of B from a proof of A.

This calculus is a sufficient starting point to do mathematics. It is possible to construct definitions of all the usual logical connectives (and, or, not), quantifiers, and equality (using Leibniz' definition of substitutivity of equals). See [CoH85] for an example of how this is done in one system.

The resulting logic is intuitionistic; all objects purported to exist must be constructed, and there is no law of excluded middle. However, if desired, logical connectives and quantifiers obeying the usual non-intuitionistic rules can be constructed from the intuitionistic ones.

A branch of category theory, the theory of Topoi [Top79] leads naturally to the same intuitionistic logic and is a convenient abstract setting for foundational questions in this kind of logic.

Potential advantages of Curry-Howard systems for ATP include: higher order quantifiers are naturally available; we can get a lot of security in the logic from the strong typing; and there is a natural mapping between proofs and programs for constructing objects. So far the only provers using such representations are proof checkers, having very limited search capabilities.

3.4 Other Logics

Many sorted logic brings the idea of typed variables and terms into first-order logic. Walther [Wa83] (see section 6.9) has developed a complete many sorted extension of resolution. Mathematical problems can often be expressed more compactly in many-sorted logic than in standard FOL. There is a significant gain in efficiency of search for proofs, since the types attached to the terms place restrictions on permissible unifications.

An example which has been widely used as an ATP benchmark is "Schubert's Steamroller". (See below.) Fig. 11 shows how many sorted logic can improve the proof length and input sizes for this problem, and also includes data on further improvements which are possible using Cohn's LLAMA logic [Coh87].

Figure 11 near here.

Schubert's Steamroller Problem

> Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore there is an animal that likes to eat a grain eating animal.

For reasoning about the common-sense world, for planning actions, and for communicating with agents (including people), it is necessary to express and reason about ideas like possiblity, belief, knowledge, successiveness

STEAMROLLER PROBLEM STATISTICS

	FOL	Walter's logic	LLAMA
No. of clauses initially	27	12	3
No. of possible inferences	102	12	7
Length of proof	33	10	5

Figure 11

(in time), etc. Modal logics and temporal logics have been developed for this purpose. Proof procedures based on connection methods [Wal86] and semantic tableaux [Smu68] have been developed. See [Tur84] for a review.

Others, particularly Kowalski [Ko79], have argued that modal and temporal logics are unnecessary and that the corresponding reasoning can be formulated and carried out entirely in FOL. The Situation Calculus [McC63, McC69] formulates actions and there effects on states in FOL. Green [Gr69] developed a large working system based on resolution for performing such reasoning.

For recent work in applications of these methods, see [Ko86, Ko86a, Ko86b, Ap82, Moo85]. An excellent textbook covering this area is [GN87].

3.5 Equality

An early problem, a persistent one, is that involving *equality*, the "substitution of equals". For example, the theorem

$$(a = b \ \wedge \ P(a)) \ \rightarrow \ P(b)$$

is rather easy, one simply substitutes a for b, or vice versa (assuming of course that "=" has its traditional meaning). But in more complex examples, like the following theorems,

A group for which $x^2 = e$, is commutative, (Hard)

A ring for which $x^3 = x$, is commutative, (Very Hard)

the proof discovery process is difficult for a computer program, because there are so many ways in which one term can be replaced by another.

The problem arises because, if $a = b$ is hypothesized, then we can replace either a by b, or b by a. This branching factor of 2, when invoked many times, leads to a serious combinatorial explosion. Paramodulation

[Wo70] and E-Resolution [Mo69], provided *complete* solutions to the equality problem, but brought very little to prevent the inherent explosion. Some ATP researchers have greatly tamed the problem by the use of *rewrite rules*. Called *demodulators* by Wos [Wo67] and *reductions* by Bledsoe [Bl71], these procedures rewrite a formula using a set of *reducers* or *rewrite rules*. For example, if we have the rewrite rules

$$x + 0 \rightarrow x$$

$$t \in (A \cap B) \rightarrow t \in A \ \& \ t \in B$$

...

we would rewrite the formula

$$f(t) \in (A(x) \cap B(x+0))$$

as

$$f(t) \in A(x) \ \& \ f(t) \in B(x).$$

The great advantage here is that the substitution on *one-way* only. We replace "$x + 0$" by "x", but do *not* replace "x" by "$x + 0$", as might be possible by paramodulation and E-resolution. Thus a branching factor of 2 is replaced by 1! However, the disadvantage is that such procedures are *incomplete*, some theorems cannot be proved by rewriting alone.

3.5.1 Term Rewriting Systems

An exciting advancement in this area was an attempt to enlarge these sets of rewrite rules to *complete* sets, the so called *complete sets of reductions*. A signal paper in this subarea was [KB70] by Knuth and Bendix that provided a set of ten rewrite rules which constitute a *complete set of reductions* for (non-commutative) group theory. See figure 12. These can be used, by rewriting alone, to prove a variety of theorems in group theory.

Figure 12 near here.

Knuth and Bendix also offered a procedure for *completing* an incomplete set, where that is possible.

This is part of a rapidly growing subfield of ATP called *Term Rewriting* Systems, which includes work on *narrowing* [Sl74] and *unification algorithms with built-in theories* [Fay79].

The first studies concerning the use of complete sets of reductions in resolution was conducted by Dallas Lankford [La75]. It brought together the notion of complete sets of reductions with that of "narrowing" introduced by Slagle [Sl74].

The connection between CSOR's and the study of unification algorithms became closer when independently, Peterson and Stickel [Pe81] and Lankford and Ballantyne [LB77] used the commutative associative unification algorithm [St81] to extend the Knuth-Bendix completion algorithm to handle commutative associative functions. Conversly, Fay [Fay79] used the narrowing algorithm to generate unification algorithms for theories which could be represented by CSORs. Fay's work was extended by Hullot [Hul80]. The study of unification algorithms is now being actively pursued by several research groups, at SRI [Sm87] and Kirchner [Kir86] in particular. See also [CREAS87].

A good survey of the field up to 1980 is found in [HO80]. A more up-to-date survey on completion can be found in [Der87A], and an equally recent survey on the termination of systems of reductions can be found in [Der87b].

COMPLETE SET OF REDUCTIONS

For a Group

KB1 $x + 0 \rightarrow x$

KB2 $0 + x \rightarrow x$

KB3 $x + (-x) \rightarrow 0$

KB4 $(-x) + x \rightarrow 0$

KB5 $(x + y) + z \rightarrow x + (y + z)$

KB6 $-(-x) \rightarrow x$

KB8 $-(x + y) \rightarrow (-y) + (-x)$

KB9 $x + ((-x) + y)) \rightarrow y$

KB10 $(-x) + (x + y) \rightarrow y$

Figure 12

4 LOGIC PROGRAMMING AND CLAUSE-COMPILING

Another giant subarea of ATP is represented by the PROLOG community, or more correctly *Logic Programming*. During the early 1970's Kowalski, Colmereauer, Roussel and others [Ko74, Rou75], discovered that one could use a theorem proving system as a programming language. This is in the spirit of earlier work by Green [Gr69], where an *answer-clause* was used to return the list of bindings of variables, resulting from the proof of a theorem. For example, if one asserts the facts

$$Father(Frank, Mary)$$

$$Mother(Mary, Ted)$$

$$Grandfather(x, z) \leftarrow father(x, y) \ \& \ Mother(y, z),$$

and proves the theorem

$$\exists x \ Grandfather(x, \ Ted),$$

he can obtain the binding

$$x \leftarrow Frank,$$

which gives an answer to the question, "Who is Ted's Grandfather?".

PROLOG is widely used as a programming language, especially in AI, and there are a number of implementations of it. The "standard" version employs ordinary resolution, but

1. allows only horn clauses[3]

2. does not do the "occurs check" during unification.

[3]A clause is horn if it has at most one positive literal. e.g., $\neg P(x) \vee Q(x) \vee \neg R(x, y)$

By restricting use to horn clauses, the implementation can employ a depth-first search, which greatly simplifies the storage allocation problem, and enables high performance via *clause-compiling* (which we will discuss shortly).

There is no apparent difficulty with ignoring the occurs check when PROLOG is used as programming language. But it is unsound as a Theorem Prover, because it would allow the unification of formulas such as

$$P(g(x), x) \text{ and } P(y, h(y)),$$

thereby (as we saw earlier), "proving" non-theorems such as

$$\forall\, x\ \exists y\ P(y, x)\ \rightarrow\ \exists y\ \forall x\ P(y, x)$$

It is also incomplete for FOL, because it employs a depth first search, and is restricted to horn clauses.[4] So why are we interested in PROLOG as a reasoning mechanism, since it is unsound and incomplete? The reason is that during the last few years David Warren (for DEC10 PROLOG) and others have used some compiling techniques (clause-compiling, or rule-compiling) to greatly speed up the process – by orders of magnitude.

Shortly we will (very) briefly describe how clause-compiling is done for PROLOG, and how that is extended to speed up proofs in full first order logic.

Our interest is in Automatic Deduction more than Programming, so we will not report on the enormous literature on Logic-Programming and PROLOG. Those with further interest should consult review papers such as those found in [CT82].

[4]Of course PROLOG, like any other programming language, can be used to implement a sound and complete theorem prover. What is more, Plaisted's SPRF [Pl87] (see Section 6.11) gains much of the speed of PROLOG for ATP.

4.1 Clause Compiling in PROLOG

Clause compiling is like ordinary compiling (of say LISP), in that it involves: structure sharing, clever use of the stack, open coding of unification, and much more. See papers by Warren [War87] and Stickel [St86].

A key to clause-compiling is to have an unchanging set of (original) clauses which will not be enlarged during the proof. So that these can be compiled *once and for all* at the beginning, in a way that makes their use extremely fast. Additionally, there will be one goal literal which continually changes (during the proof search). These original clauses are compiled by anticipating how unification might be accomplished with each of their literals, and constructing a computer program to carry out that unification and other tasks.

This program can be written in some computer language such as C, LISP, or an Abstract Machine Language such as Warren's [War87], and then compiled (ordinary compiling) into machine code. See [War87, St86] for details.

Suppose we have the following input clauses (and others)

1. $(P\ x\ 1) \leftarrow (Q\ x\ z)\ (S\ z)$

2. $(P\ (f z)\ y) \leftarrow (R\ y\ z)$

3. ...

The clause compiler will compile each of the predicates P, Q, S, R, ..., by constructing a LISP[5] Function for each of them, and other supporting functions (not shown here).

Shown here is the function, FUN-P, which has been constructed for the predicate P.

[5]or a C program, etc. We have used LISP here to simplify the presentation.

```
(DE FUN-P  (u v CONTINUATION) (GOAL)
    (PROG (z)
          (COND((UNBOUND-VARIABLE v) (ASSIGN V 1))
                ((NOT (= V 1)) (GO OUT)))
          ( . . .      Allocate, etc . . .)
          ( . . .      Alter CONTINUATION to include the further goal(S z))
          (Q u z CONTINUATION)
    OUT
          (COND((=(FCN-SYM u) 'f) (SETQ Z (ARG1 u)))
                (T (go OUT2)))
           (R v z CONTINUATION).
    OUT2  . . . .  ))
```

Much has been left out, but the main idea is that when a goal literal of the form (P u v) is encountered, to determine whether clause 1. will apply to it (i.e., whether (P x 1) will unify with (P u v)), we can ignore u since x is a variable and hence can be bound to any term; we need only check whether v is 1 or is a variable, and then accomplish the further goal (Q u z).

The *continuation* parameter refers to any additional goals that were carried over from a previous call; we must add to it the subgoal (S z) before proceeding to the goal (Q u z). If (Q u z *continuation*') succeeds, i.e., the goal (Q u z) is accomplished plus the goals of *continuation*', then the proof is finished; if not, then it attempts to apply clause 2. to the goal literal (P u v). This is done at the point OUT in the program.

Similar LISP functions are constructed by the clause-compiler for the other predicates Q, R, S, and any others that appear in the original clause set. All of these LISP functions are then compiled (traditional compiling) to C code or machine code. Of course, as mentioned earlier, the clause-

compiler could avoid LISP altogether. But LISP offers a convenient tool for the clause-compiler and a convenience to us for explaining how this part of clause-compiling works.

4.2 Clause-Compiling for First Order Logic

The phenomenal speeds gotten by clause-compiling in PROLOG were not lost on the rest of the ATP community – they wanted this performance too, but could not use the results from PROLOG unless three major difficulties with it were overcome:

1. the horn clause restriction
2. the depth-first search problem
3. the occurs-check problem

Work on these problems, to bring clause-compiling (and its inherent speeds) to all of first order logic, represents some of the most exciting work in ATP right now. Some systems which extend the PROLOG compiling techniques as follows:

- Stickel's "Prolog Technology Prover" [St86]
- Plaisted's "Simplified Problem Reduction Format" [Pl87]
- Loveland's "Near Prolog" [Lo87]
- Overbeek and Lusk's New Argonne Prover"
- Munich Group's "PROTHEQ" [BAY86]

There are probably a number of others. How do these systems overcome the restriction, 1–3? Let us consider them in order.

The *horn clause* restriction (1) was used in PROLOG to allow a linear search mechanism: once a proof-search is started it can proceed to success or failure without having to backtrack, as is necessary when using ordinary-clauses resolution. This linear format greatly simplifies the search mechanism; one only needs a "stack" and no auxiliary clause storage; only the original clauses are retained, and they can be compiled before the proof search starts.

The way that Stickel [St86] avoids the horn clause restriction for full resolution is to employ a variation of resolution called *model-elimination* (which is essentially *SL-resolution*)[6], which uses chains instead of clauses.

These chains act like clauses, with extra data in them which code the history of how they were constructed in the proof process. This allows a linear format similar to that used in PROLOG, but requires the addition of many contrapositives[7] of input clauses.

Plaisted avoids the horn-clause restriction by using a form of "Case-Splitting", which does not require contrapositives[Pl87].

Loveland uses "multiple-head horn clauses" e.g. $P,\ Q \leftarrow R$, with no contrapositives needed. His technique is similar to Model Elimination but it greatly reduces the amount of extra "history information" recorded with clauses [Lo87].

The *depth-first* search problem (2), is avoided by "iterative deepening", i.e., by repetitively searching to deeper and deeper levels of the search tree. The added cost for recomputing the top parts of the tree is minimal when the search tree is branchy, which is usually the case.

There have been two ways used for avoiding the *occurs check* problem (3):

---

[6]Model Elimination was discovered by Loveland [Lo68, Lo69]; it is equivalent to SL-Resolution, developed independently by Kowalski and Kuener [Ko71].

[7]e.g., for the clause $P \leftarrow Q \wedge R$, we would add the contrapositives $\neg Q \leftarrow (\neg P \wedge R)$ and $\neg R \leftarrow (\neg P \wedge Q)$

(i) by detecting at compile time which literals can possibly have an occurs-check problem e.g., $P(x,\ f(x))$, tagging them, and handling only them during the proof.

(ii) by examining the substitution resulting from any successful unification to determine if there was a problem, and rejecting substitutions with "cyclic" terms, like $x \leftarrow h(g(x))$. (Plaisted, Overbeek and Lusk)

Both methods cause a loss of speed, but not a severe one because such problems rarely occur. (e.g., it is necessary for a variable to occur twice in such a literal for it to present an occurs check problem.)

We believe that clause-compiling will be very important for the future of ATP. These great speeds cannot be ignored. Granted that the ultimate solution is not in speed, but in the better use of knowledge to prune the search tree. Nevertheless, fast reasoning components will be important parts of future technology.

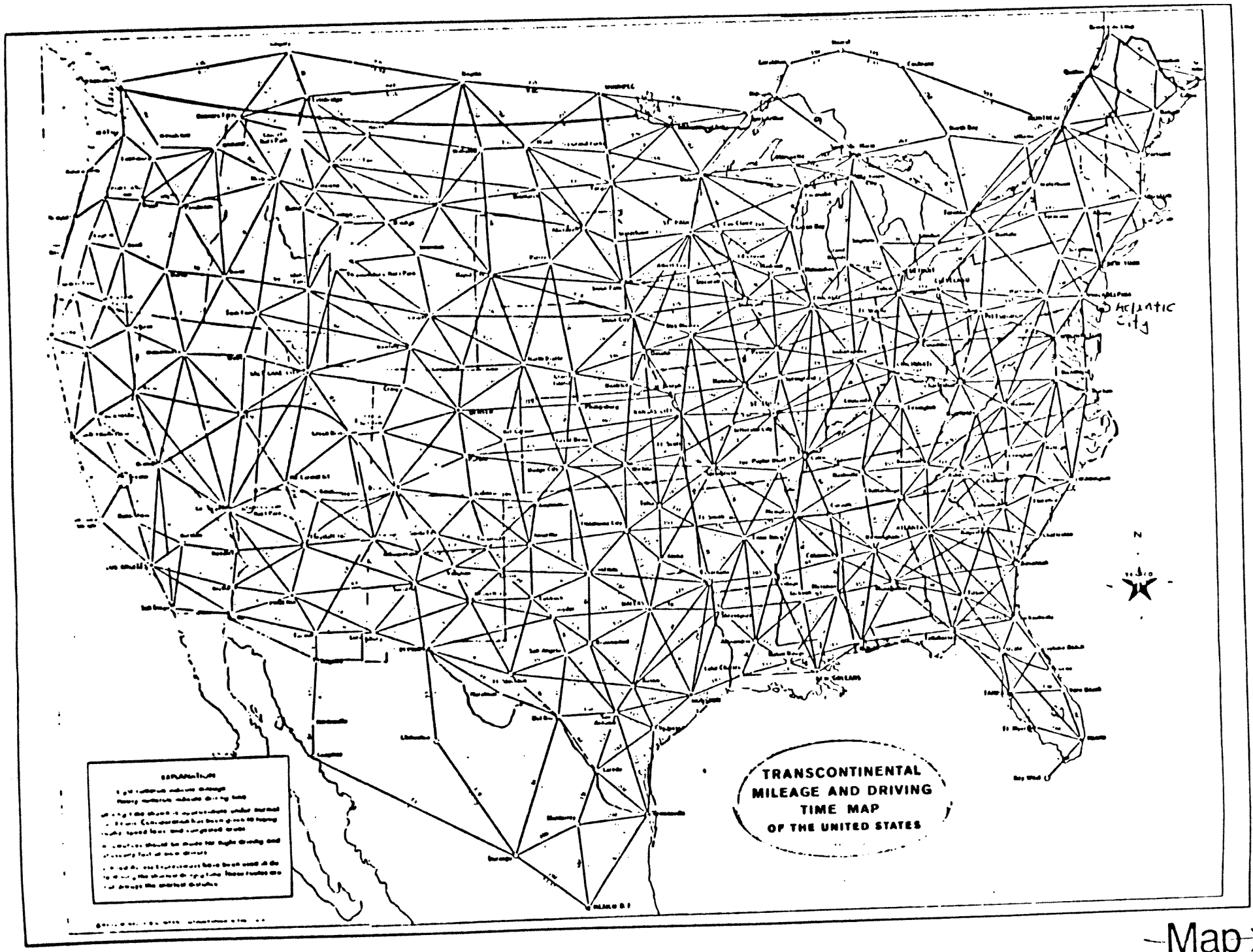
Also, compiling methods of the kind that we have described, are useful for other components of the reasoning process. For example, similar improvements in performance have been obtained for forward chaining [Fo80], rewriting or demodulation [Bo86], inheritance [AN85], and data-base indexing [But86].

5 OVERVIEW OF PROOF DISCOVERY

Now let us give an overview of (our version) of Automated Proof Discovery. How do we classify the research that is being done and should be done?

We feel that building a program for discovering proofs is like designing an autonomous vehicle to cross the USA, say from Atlantic City to Fresno. See Figure 13.

Figure 13 near here.



Map xx

Exhibit 13

To do so one needs:

1. Fast cars;

2. Tactics: For getting from city to city;

3. Strategy: An overall plan of action.

And one needs a map.

But note that speed alone is not enough; dashing off in more less the right direction will not lead to a distant goal without some guidance, no matter how fast the car.

One could liken this to the way that automated proof discovery is being attacked. See Figure 14. Here again we have "fast cars" (fast inference vehicles), tactics and strategy. Let us break this down into more detail.

Figure 14 near here.

Category 1 is easy to define, it consists of those efforts which produce *speed* of inferencing. They are *essential* to the success of ATP. Whatever else we do to prune the tree, it is absolutely necessary that we have great speeds for the "vehicle."

Examples of parallel processing in ATP, are the efforts of Overbeek et al, at Argonne National Lab [REF], the Munich Group [REF], and Waltz and Stanfield at Thinking Machines [REF].

But speed alone is not enough. Again we need overall guidance that comes from tactics and strategy.

It is not so clear what to put in Category 2, tactics, but we feel that those methods which employ "large inference" steps tend to have the "city to city" flavor, as do the special purpose provers. We will discuss these in more detail shortly.

But what do we put in Category 3, strategy? Is there any method being used, that takes an overall,*global* view, that provides and uses an overall