Lecture 9
Resolution Theorem Proving

To recap, we have looked at logic and some rules of deduction in order to understand automated reasoning in the general case. In the last lecture we looked at a particular rule of inference, the resolution rule. We know that the application of this rule produces a complete search method, which means that it will prove any true theorem which can be written in first order logic. Application of the rule relies on having two sentences in conjunctive normal form (CNF) where one literal of one unifies with the negation of a literal in the other. We described how to write first order sentences in CNF, and how to find unifying substitutions in lecture 8. In this lecture, we look at exactly how to use the resolution rule to prove first order theorems in practice.

9.1 Overview of Resolution

Recall that resolution uses proof by refutation, where we add the negation of the theorem and the axioms to the knowlegde base, and deduce the False statement from it. This method demonstrates that the theorem being false causes an inconsistency with the axioms, hence the theorem must have been true all along. It uses only one rule of deduction: the generalised resolution rule we saw in the last lecture, used to combine two **parent clauses** into a **resolved clause**.

We can express the full **resolution rule of inference** concisely using 'big $\vee$' notation:

$$\frac{\bigvee_{i \in A} L_i \qquad \bigvee_{i \in B} L_i}{\bigvee_{i \in C} Subst(\theta, L_i)}$$

$$Unify(L_j, \neg L_k)$$
$$j \in A, k \in B$$
$$C = (A \cup B) \backslash \{j, k\}$$

The 'big $\vee$' is just a more concise way of writing clauses, where underneath the V we specify a set of indices for the literals L. For example, if A = {1,2,7} then the first parent clause is $L_1 \vee L_2 \vee L_7$. (We can use a similar 'big $\wedge$' notation to express conjunctions.) The rule resolves literals P_j (a negative literal) and P_k (a positive literal). We just remove j and k from the set of indices to get the resolved clauses.

We repreatedly resolve clauses until eventually two sentences resolve together to give the **empty clause**, which contains no literals. This is taken to be the False statement. To see why this is a natural thing to do, remember that to perform a resolution step, a literal L_1 in one clause must have unified with the negation of a literal L_2 in another clause. If a resolution step took place and left nothing, then the clauses resolved together must have been single literals, with one unifying to the negation of the other. This means that there was a way, call it mu,

of writing L_1 and L_2 so that Subst(mu, L_1) and Subst(mu, L_2) became a literal and its negation. Furthermore, this means that both Subst(mu, L_1) and Subst(mu, L_2) were true clauses in the knowledge base, hence both true at the same time. But they can't be true at the same time, as one is the negation of the other. Hence, if two clauses resolve to give an empty clause, there must be an inconsistency in the database, and False has indeed been derived.

Once false has been found, we know the theorem is true, and if the agent has been careful to remember how each new sentence was added to the knowledge base (i.e., which two sentences resolved together to produce it), then it can work backwards from the False statement and find the chain of inference to the axioms. Reading the chain forward will give a proof of the falsity of the negated theorem, which proves the theorem itself. Producing the proof may be time consuming, and some implementations do not produce the proof. This means that they can tell you that a statement in predicate logic is true, but not tell you why.

9.2 Search Spaces

We can specify resolution proving as a search problem using notions laid down in section 3.1 of lecture 3:

- **Initial State**: A knowledge base (KB) consisting of negated theorem and axioms in CNF.
- **Operators**: The full resolution rule of inference picks two sentences from KB and adds a new sentence.
- **Goal Test**: Does KB contain False?

The states in this search space are the individual knowledge bases. The search is either looking for an artefact (whether the initial KB is unsatisfiable) or a path (a **proof** that initial KB is unsatisfiable).

Let's illustrate the concept of a **resolution search space** with the simple example from Aristotle we've seen before. Apparently, all men are mortal and Socrates was a man. Given these words of wisdom, we want to prove that Socrates is mortal. We saw how this could be achieved using the Modus Ponens rule, and it is instructive to use Resolution to prove this as well.

The initial KB (including the negated theorem) in CNF is:

```
1) is_man(socrates)
2) ¬is_man(X) ∨ is_mortal(X)
3) ¬is_mortal(socrates)
```

We can apply resolution to get TWO different solutions. The first alternative is that we combine (1) and (2) to get the state A:

```
1) is_man(socrates)
2) ¬is_man(X) ∨ is_mortal(X)
3) ¬is_mortal(socrates)
4) is_mortal(socrates)
```

Then combine (3) and (4) to get the state B:

```
1) is_man(socrates)
2) ¬is_man(X) ∨ is_mortal(X)
3) ¬is_mortal(socrates)
4) is_mortal(socrates)
5) False
```

Alternatively, we could initially combine (2) and (3) to get the state C:

```
1) is_man(socrates)
2) ¬is_man(X) ∨ is_mortal(X)
3) ¬is_mortal(socrates)
4) ¬is_man(socrates)
```

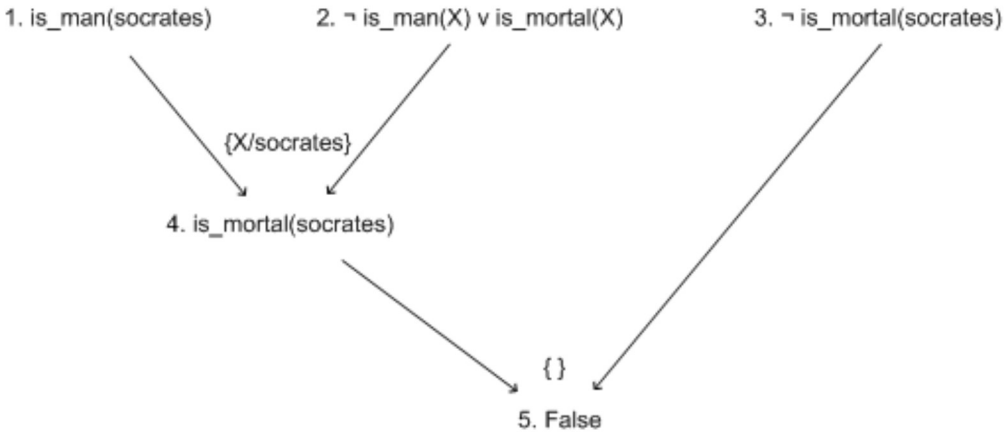
We then resolve again to get state D:

```
1) is_man(socrates)
2) ¬is_man(X) ∨ is_mortal(X)
3) ¬is_mortal(socrates)
4) ¬is_man(socrates)
5) False
```

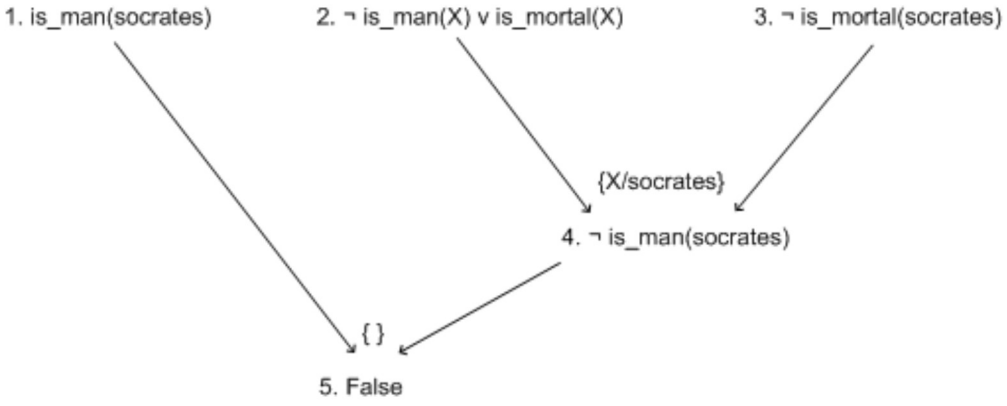
So, we have a search space with two alternative paths to a solution: Initial --> A --> B and Initial --> C --> D.

9.3 Proof Trees

Important though the idea of a resolution search space is, it is tedious to write out the whole knowledge base again to illustrate every state. Instead, it is often more convenient to visualise the developing proof. On the top line we can write the clause of our initial KB, and draw lines from the two parent clauses to the new clause, indicating what substitution was required, if any. Repeating this process for each step we get a **proof tree**. Here's the finished proof tree for the path Initial --> A --> B in our example above:



An here's the proof tree for the alternative path Initial --> C --> D:



Complex proofs require a bit effort to lay out, and it is ususally best not to write out all the initial clauses on

the top line to begin with, but rather introduce them into the tree as they are required.

Resolution proof trees make it easier to recontruct a proof. Considering the latter tree, we can read the proof by working backwards from False. We could read the proof to Aristotle thus:

> "You said that all men were mortal. That means that for all things X, either X is not a man, or X is mortal [CNF step]. If we assume that Socrates is not mortal, then, given your previous statement, this means Socrates is not a man [first resolution step]. But you said that Socrates *is* a man, which means that our assumption was false [second resolution step], so Socrates must be mortal."

We see that, even in this simple case, it is difficult to translate the resolution proof into a human readable one. Due to the popularity of resolution theorem proving, and the difficulty with which humans read the output from the provers, there have been some projects to translate resolution proofs into a more human readable format. As an exercise, generate the proof you would give to Aristotle from the first proof tree.

In the slides accompanying these notes is an example taken from Russell and Norvig about a cat called Tuna being killed by Curiosity. We will work through this example in the lecture.

9.4 Dealing with Equality

The resolution method we have described will find proofs to all first order theorems *eventually*. However, more effort is required in order to turn the theory into practice. Firstly, we have to deal with equality (this section). Secondly, we need heuristic strategies to make provers more efficient (the next section).

The unification algorithm described in the previous lecture will not unify pairs of sentences such as (i) `ispresident(g_bush)` and (ii) `ispresident(george_bush)`, and rightly so, because *syntactically*, the constants `g_bush` and `george_bush` are quite different. We would not want our theorem proving agent to go around making assumptions about the equality of constants just based on their names, as this would make the system unsound. However, what if we also had the sentence: `g_bush = george_bush` in the knowledge base, i.e., we had informed the system of the equality of these two constants. Surely we would want it to be able to unify the two `is_president` predicates, as this will allow it to do more resolution steps.

One way to get around the problem with equality is to add lots of extra clauses to our knowledge base, expressing everything we wanted to say about equality. These are known as **equality axioms**. In particular, we would have to say that it was reflexive, symmetric and transitive:

$\forall$ X (X = X). [symmetric]
$\forall$ X, Y (X=Y $\rightarrow$ Y=X). [reflexive]
$\forall$ X, Y, X (X=Y $\wedge$ Y=Z $\rightarrow$ X=Z). [transitive]

However, this would not be enough to enable the system to realise when it can unify two constants. We would have to put in additional statements about equality for each of the predicates in our knowledge base. So, for example, if the predicate P of arity 1 was mentioned in the knowledge base, then we would have to add the following to the knowledge base:

$\forall$ X, Y (X = Y $\rightarrow$ P(X) = P(Y)).

This kind of thing will have to be done for every predicate and function if we are to be sure not to miss any opportunity for unification.

An alternative way to get a proving agent to use equality to its full is to employ another inference rule called **demodulation**. Like the resolution rule, this takes two sentences as input, but one of them must be an equality sentence relating two terms. Then, if, in the other sentence, there is a term T which can be unified with the left hand side of the equality sentence, then the unifying substitution is applied to the *right hand* side, and this side is substituted for T. It's more obvious when written as the following rule:

$$\frac{X = Y, \quad A[T]}{\text{Subst}(\theta, A[Y])...)} \quad \text{Unify}(X, S) = \theta$$

We can also allow demodulation to work the other way, i.e. replace Y with X, but as with rewriting we need to take care not to get stuck in any loops. Our presidential example would be a trivial case for the demodulation rule:

$$\frac{\text{george\_bush} = \text{g\_bush}, \quad \text{is\_president(george\_bush)}}{\text{is\_president(g\_bush)}}$$

In this case, the unifying substitution is just the trivial identity substitution.

Most state of the art resolution theorem provers have resolution at their heart, but use a number of other inference rules, including demodulation, and its big brother **paramodulation**, which deals with cases where we only know that $x=y \vee P(x)$.

9.5 Heuristic Strategies

It is important to remember that, for theorems of any interest, the initial knowledge base will be quite sizeable. Moreover, each resolution step adds another sentence to the knowledge base. Hence, a strategy such as breadth first search, which takes every pair of sentences in turn and tries to resolve them will often get nowhere near to proving even fairly trivial theorems.

Automated Reasoning is an AI task that can be represented as a search problem. Therefore, it is not surprising that people began to think about some ways in which to improve the performance by using rules of thumb (heuristics). This was mainly motivated by the fact that early implementations failed to perform as expected - there were lots of simple looking theorems that they could not prove. We look at four heuristics for choosing what to resolve at what stage and what to leave out of resolution steps entirely.

The **Unit Preference** strategy is to be greedy when it comes to clauses which are just single literals, known as **unit clauses**. That is, whenever possible, this strategy chooses steps which involve the resolution of a unit clause with something else. The idea is that we are aiming for a clause with no literals, so if we can keep the length of clauses down to a minimum, then perhaps this is a good strategy. Of course, continually resolving with unit clauses will produce a reduction in clause sizes. Unit preference doesn't actually reduce the branching rate in medium-size problems enough to make an impact. However, the use of this strategy produced a speed up in resolution provers for simpler problems, enabling implementations to prove many theorems which were out of their range before.

The **Set of Support** (SOS) strategy restricts the resolutions which are allowed to occur to just a subset of those possible. Specifically each resolution must involve a clause from a subset called the set of support. The idea is

that the set of clauses excluded from the set of support are consistent within themselves, so the path to the solution must involve a resolution with one of the clauses in the SOS. This has an analogy with a heuristic we teach children to help them with problem solving: if you haven't used everything mentioned in the question, then you're probably not going in the right direction. In our situation, given that the axioms are consistent, then we know that the path to the solution will involve at least one resolution of the negated theorem statement with something else. Of course, the result of that resolution might also lead to the solution, and so on. This indicates the heuristic most used in set of support strategies: start with just the negated theorem in the SOS and keep adding in those clauses which are generated from the resolution of anything with an SOS member.

A special case of the SOS strategy is the **Input Resolution** strategy. This restricts the SOS to include only the clauses that were in the knowledge base before the search started, namely the axioms or negated theorems. Hence each resolution step involves at least one of the clauses provided as input. This will clearly reduce the size of the search space. This strategy is not complete for first order sentences in general, but is complete for Horn clause databases (such as Prolog Programs).

A final heuristic is called **Subsumption** and this checks when a newly formed clause is **subsumed** by one existing in the database already. One clause, C, is subsumed by another, D, if C is more specific than D. As the more specific clauses can be implied by the more general ones, then the more specific ones can be discarded without loss of generality, i.e., it will still be possible to prove the theorem. Of course, the reduction in the size of the knowledge base will produce a smaller search space.

In practice, a naive way to perform subsumption is to check whether one clause, C_1 could be unified with (a subset of) the literals in another clause, C_2 in such a way that the substitution does not affect C_1 other than renaming variables, i.e. no extra function or constant symbols are substituted into C_1, the more specific clause. For example, the clause `p(george)` $\vee$ `q(X)` unifies with a subset of the literals in the clause `p(A)` $\vee$ `q(B)` $\vee$ `r(C)`. The substitution is {A/george, X/B}, which instantiates A in the second (more general) clause, but only renames X to B, thus leaving the first (more specific clause) much the same as before.

There are less naive ways of performing subsumption checks which are much quicker, but subsumption checking of every clause against a new clause can be computationally expensive. Therefore, the reduction in search space will have to outweigh this expense.

9.6 Applications to Mathematics

The Logic Theorist

One of the first applications of automated theorem proving was the use of Newell, Shaw and Simon's Logic Theory Machine to prove theorems from Whitehead and Russell's Principia Mathematica. The program proved 38 of the 52 theorems they presented to it, and actually found a more elegant proof to theorem 2.85 than provided by Whitehead and Russell. On hearing of this, Russell wrote to Simon in November 1956:

'I am delighted to know that Principia Mathematica can now be done by machinery ... I am quite willing to believe that everything in deductive logic can be done by machinery.'

Newell, Shaw and Simon submitted an article about theorem 2.85, co-authored by the Logic Theory Machine, to the *Journal of Symbolic Logic*. However, it was refused publication as it was co-authored by a program.

An informative page on Automated Deduction, including the Logic Theorist is here.

In the 1960s and 70s, when it was realised that resolution was a complete procedure and implementations started to proliferate, there were some people who speculated that soon enough every mathematician would have an automated theorem prover on his or her desk which they could turn to in times of need to prove their theorems. This has not turned out to be the case, and should be taken as an illustrative example of why we shouldn't make such overly optimistic claims about AI. It also points out just how much more difficult the problem of automating mathematical theorem proving has been in comparison to playing chess, for example.

It's perhaps unfair to do so, but we can compare the impact on mathematics of automated reasoning with its sister subject, computer algebra systems (CAS), which perform symbolic manipulations and giant calculations. CAS itself grew out of AI research undertaken by mathematicians and computer scientists and has had a major impact on mathematics. Most undergraduate maths courses now offer modules in standard CAS applications such as Maple and Mathematica, and much research mathematics is facilitated at least in part by some CAS or another. In fact, Andrew Wiles, who in 1995 proved Fermat's Last theorem, purportedly only using a computer to write the proof up, is very much the exception to the rule, and he is a member of a dying breed.

Back to the comparison: there is no comparison. Automated reasoning programs have barely made a scratch on the surface of mathematics education or research. Partly due to this lack of success, and partly due to the success in other, well funded, areas such as hardware and software verification, the active interest in automating mathematics has largely dwindled. Note that hardware and software verification are large areas of research, which are beyond the scope of this lecture course.

There are, however, still a few of us who continue to pursue the dream of having an intelligent, creative AI agent help us make discoveries in mathematics, and below are a couple of the more interesting projects which have actually added to the mathematical literature, both of which use resolution theorem proving.

- Proving Algebraic Theorems

Automated theorem proving in general attempts to find proofs to theorems which are usually assumed to be true. This is, of course, not how mathematics proceeds in general. Only in rare cases is a theorem written down and then a concerted effort is made to prove it. Famous examples include Fermat's Last Theorem and Goldbach's conjecture (that every even number is the sum of two primes). In most cases, theorems are derived and are found at the end of lengthy reasoning processes including calculation, induction of patterns from observations and deduction of facts following from other known facts. Indeed, by the time most theorems are properly written down for the first time, they have been proved already. Having said that, there are cases where the "know it is true and then find the proof" approach can be useful, and other cases where an exhaustive search over a large set of theorems to find some with certain properties can lead to interesting discoveries.

In particular, Bill McCune and Larry Wos at the Argonne National Laboratories have been building and applying state of the art resolution theorem provers to mathematical discovery for many years. Their biggest accomplishment was with a prover called EQP, which solved an open conjecture known as the Robbins Algebra Conjecture. Herbert Robbins conjectured that commutative, associative algebras with the extra

condition that $n(n(x) + y)+n(n(x)+n(y)) = x$ for some function n, are Boolean algebras. The attempts to prove or disprove this conjecture outwitted mathematicians for 60 years. After many years of development of the problem (with the development also using automated deduction), it was finally in a form whereby they could run EQP to attempt to find a proof using resolution. EQP ran for around 8 days and used about 30 Mb of memory to find the proof in 1996. This still remains the biggest mathematical achievement of any automated theorem prover.

Although the Robbins result has been the most high-profile, the Argonne team have worked on many projects which have added to mathematics. In particular, Bill McCune has worked with a mathematician called Padmanabhan. He sends McCune conjectures, and McCune uses his latest resolution prover, called Otter, to attempt to prove the conjectures, sending back any proofs that he finds. This has been a very fruitful partnership, leading to enough results to fill a book about cubic curves.

Another particularly interesting project has been to determine the smallest way to axiomatise certain algebras such as group theory. Group theory is a domain of mathematics which involves symmetries. Normally it is described as a set of objects with an operation, *, which takes a pair of elements of the set and returns a third element. This operation is subject to a set of rules. Normally, these are expressed as three axioms: inverse, identity and associativity. By performing a search over smaller representations for the axioms, and trying to prove using Otter that each was equivalent to the group theory axioms, they found new, very succinct representations for the group theory axioms, and a host of other algebras.

A web page describing the work of the Argonne automated reasoning laboratory is here.

- **Automated Conjecture Making**

One of the course lecturers (Simon) has for a long time been involved in the development of a system called HR which performs automated theory formation in scientific domains. A theory consists of a set of concepts with examples and definitions, a set of conjectures about those concepts, and a set of proofs of the conjectures. HR is really a machine learning program, but it has had special application to the discovery of concepts and conjectures in domains of mathematics such as number theory, group theory and graph theory. More to the point of this lecture, HR uses Otter to do its theorem proving.

HR is often good at identifying non-obvious conjectures which we may not have thought of on our own. For instance, we looked at the domain of anti-associative algebras, mainly because we could not find any research done in this area. These are algebras over one operator with just one axiom: that no triple of elements is associative, that is, for all x, y and z, $(x * y) * z$ is not equal to $x * (y * z)$. We gave HR only these axioms to work with, and it used the MACE model generator (a sister program to Otter, which generates examples as counterexamples to false theorems), to generate examples of anti-associative algebras. HR conjectured and Otter proved many interesting conjectures, that we probably would never have stumbled across, including: that there must be two different elements on the diagonal of the multiplication table, that anti-associative algebras cannot be quasigroups and that they cannot have an identity element. Of course, because these results are *proved* by Otter, and because Otter is a very highly respected theorem prover (i.e., if it says it's proved, then it always is proved), we know that they are true in the general case.

Perhaps the most interesting results have come in number theory. In this domain, we use Otter in a slightly different way, because resolution provers are not as effective in numerical domains as they are in algebraic ones such as group theory. Because any conjectures that Otter can prove are unlikely to be at all difficult for a human to prove, we get HR to discard them because they won't be interesting. Some nice results (try to prove them if you can) from HR in number theory include:

- If the sum of divisors of a number is prime, then the number of divisors will be prime.
- The sum of divisors of a square number is an odd number
- Perfect numbers are pernicious. Perfect numbers are equal to half the sum of their divisors. So, for example, 6 is a perfect number, because the sum of its divisors is 1 + 2 + 3 + 6 = 12, which is twice 6. They are called perfect numbers because the first is 6 and the next is 28. God created the earth in 6 days, and the lunar cycle is 28 days. As the next perfect number was not known for many years, people thought these numbers had divine significance. However, Euler found the next one to be 496, which took away some of the mystique! HR discovered that when you write these in binary, you get a prime number of 1s (pernicious numbers).

None of these theorems were known to the user when they were reported, and each one takes around 10 lines, and a fair bit of number theoretic knowledge, to prove. In addition, HR has been responsible for around 30 new number types being added to the Encyclopedia of Integer Sequences - the only non-human to do so. A nice example is refactorable numbers (where the number of divisors is itself a divisor). HR also pointed out some nice results about these, for example that odd refactorable numbers must be squares.

9.7 Other Topics in Automated Deduction

Below are some other areas of automated theorem proving you may come across at some stage.

- **Interactive Theorem Proving**

Given that, as discussed above, automated theorem provers are still far from being proficient at proving, it makes sense not to go for a fully automated approach, but rather to build more interactive theorem provers. Some systems, such as the Theorema package built by Bruno Buchberger's group in Linz, act like assistants to mathematicians, proving simple results as the user explores a theory. Other systems enable the user to guide the proof of more complicated theorems: they step in whenever there is an important choice point such as a case split. One important topic is the representation of a proof as it unfolds, and there are various techniques for graphically displaying and annotating proof trees so that the user knows what is going on (in order to correctly guide the proof at choice points).

- **Higher Order Theorem Proving**

Higher order theorem proving is what you would expect it to be - the automation of theorem proving in higher order logics as discussed in lecture 4. For an example of a higher order theorem prover (called HOL), see here. HOL has been used for a number of verification tasks, including type safety proofs for Java and the verification of cryptographic protocols.

- **Inductive Theorem Proving**

Here 'inductive' is meant in the sense of *mathematical induction* and **not** inductive reasoning (machine learning). This involves deduction with induction rules of inference. For example, a standard argument by mathematical induction over the natural numbers 0,1,2,3... is expressed in the following rule:

$$\frac{P(0), \quad \forall X.\ (P(X) \longrightarrow P(X+1))}{\forall X.\ P(X)}$$

Mathematical induction is possible over other structures apart from numbers, and is particularly useful for

deductive reasoning about recursion or iteration over such structures. As a result it has considerable application to software and hardware verification, given the ubiquity of recursion/iteration over data structures in computer systems.

- **Proof Planning**

Proof planning is a technique developed, amongst others, in Alan Bundy's automated reasoning group in Edinburgh. A proof plan is an outline or plan of a proof and proof planning is a technique for guiding the search for a proof in automated theorem proving. To prove a conjecture, proof planning first constructs the proof plan for a proof and then uses it to guide the construction of the proof itself. Common patterns in proofs are identified and represented in computational form as general-purpose tactics, i.e., programs for directing the proof search process. These tactics are then formally specified with methods using a meta-language.

Standard patterns of proof failure and appropriate patches to the failed proofs attempts are represented as critics, and to form a proof plan for a conjecture, the proof planner reasons with these methods and critics. The proof plan consists of a customised tactic for the conjecture, whose primitive actions are the general-purpose tactics. This customised tactic directs the search of a tactic-based theorem prover. Proof planning has been particularly useful for proving theorems by mathematical induction.

For more information about proof planning, see a FAQ here.

The TPTP Library
and CASC Competitions

Largely fuelled by the enthusiasm of Geoff Sutcliffe and others, the annual CASC competition is often the venue for much revelry in automated reasoning, and sometimes even a little controversy. This competition attempts to determine who has the fastest, most accurate first order (usually resolution) theorem prover on the planet. It draws unseen problems from the TPTP library (thousands of problems for theorem provers), and these are used to test the provers. The competition is split into various categories, with an overall "CNF" category being perhaps the most prestigious one to win. The winner in 2008 was a theorem prover called Vampire 10 written by Andrei Voronkov and Alexandre Riazanov in Manchester.

CASC is similar to the CASP competition in bioinformatics, where protein structures are withheld so that computers (and humans) can try to determine their structure from their genetic sequence. Such competitions are becoming very popular in AI, although there are many people who believe they are a waste of time, which encourage people to fine tune their systems only to work in competition situations.

The website for CASC is here (in particular, have a look at the T-shirts!).

The website for the TPTP library is here.

The website for Vampire is here.