AUTOMATED REASONING

SLIDES 6:

CONTROLLING RESOLUTION

Simple Restrictions:
- Subsumption
- Tautology removal
- Factoring

Saturation Search improved
Search Spaces
Introduction to Refinements

KB - AR - 09

Controlling Resolution:

In this group of slides we'll look at some basic ways to control resolution. It is easy to make resolution steps, but for even a medium sized problem the number of resolvents increases very quickly, so some method is needed to control their generation.

In Slides 3 we introduced saturation search and considered factoring. Here we'll look in more detail at subsumption and its relation to factoring. In Slides 7 and 8 we'll consider other ways of restricting the resolvents.

A number of difficulties for theorem provers have been presented by Wos and are repeated at the end of these slides. Larry Wos led the group at Argonne that produced Otter – a wonderful theorem prover that you will use soon. The successor is Prover9, but Otter is easier for beginners. This prover uses a saturation search as its basic strategy, but with many additional ways of restricting resolvents. The important thing is that the strategy is systematic.

You already saw that unrestricted resolution generates many redundant clauses. There are some very simple restrictions that are almost universally imposed in theorem provers, called *Tautology deletion*, *safe factoring* and *subsumption*. We consider these next.

Some of the material in the notes sections of these Slides (mainly proofs) is presented for information only, as we will not have time to cover it all in detail in class.

Subsumption

A clause C ***θ subsumes*** clause D if $C\theta \subseteq D$ for some θ

A clause C ***subsumes*** clause D if $\forall C \models \forall D$, where $\forall C$ means that all variables in clause C are explicitly unversally quantified. Equivalently, C subsumes D if $\{C+\neg D\}$ has no H-models (or if $C+\neg D \Longrightarrow^* [\,]$).

A clause C **strictly θsubsumes** D if C θsubsumes D without necessary factoring in Cθ. Each literal in Cθ matches a different literal in D.

Example: $Px \vee Qx$ subsumes $Pa \vee Qa$ (and θsubsumes it)
$Pxa \vee Pyx$ θsubsumes Paa but not strictly
$Pf(x) \vee \neg Px$ subsumes $Pf(f(y)) \vee \neg Py$ but does not θsubsume it.

Exercise
Say whether first clause θsubsumes the second and if so, whether strictly.

$Qxx \vee Qxy \vee Qyz$ and Qaa
$Qax \vee \neg Rxa$ and $Qab \vee Qac$
Qxx and Quv
$Sf(x)x$ and $Sug(u))$

Qaa and $Qxx \vee Qxy \vee Qyz$
Qzy and Quv
Quv and Qxx
$Sf(x)y$ and $Sug(u))$

Note: **Identica**l literals in a clause are always merged, so $Pa \vee Pa$ is always Pa and $Px \vee Px$ is always Px. They both strictly subsume $Pa \vee Qa$. If C θsubsumes D, but not strictly, can first factor C to C' which strictly θsubsumes D.

Subsumption in Use

There are two species of subsumption:

Forward subsumption: a resolvent is subsumed (no need to generate it).
Backward subsumption: a resolvent subsumes (can remove other clauses).

In a saturation search

- forward subsumption can occur either:
(i) as soon as a subsumed resolvent is generated, or (ii) after each stage
- backwards subsumed clauses can be removed either:
(i) when a subsuming resolvent is generated, or (ii) at the end of each stage.

Exercise:
2. $\neg Dxy \vee Cxy$ 3. $\neg Tx \vee \neg Cxb$ 4. Tc 5. $\neg Dcz$
8. (1,5) Dcb 9. (2,3) $\neg Dxb \vee \neg Tx$ 10. (3,4) $\neg Ccb$

Compare:
a) for forwards subsumption removal of a subsumed clause immediately or at the end of a stage
b) for backwards subsumption removal of a subsumed clause immediately or at the end of a stage
What do you recommend as a good strategy?

An Important Property of Subsumption:

6biii

Subsumed clauses can be removed from S without affecting satisfiability:

If C , D in S and C subsumes D then S is unsatisfiable iff S-D is unsatisfiable
Hence S ⇒* [] iff S- {D} ⇒* []

But note that the sets of derivations using S and using S-D are not the same
eg Backwards subsumption can mean some "proofs" are lost.

Example
1. Pxy ∨ ¬Qx ∨ ¬Ry   2. ¬Puv   3. Qa   4. Rc   5. Qb

6. (1,2) ¬Qx ∨ ¬Ry causes 1 to be removed
7. (6,3) ¬Ry causes 6 to be removed   8. (4,7) []
There are no other proofs even if 8 is not used to remove all other clauses.
Without backward subsumption there is another derivation using (6).

Relation between θsubsumption and full subsumption:

θsubsumption implies subsumption
(but not the converse - find a counter example involving a recursive clause).
(Usually checks are made for strict θsubsumption only.)

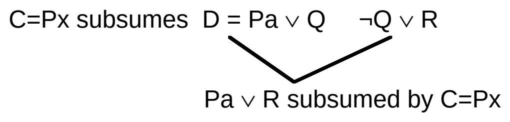
A Constructive View of θSubsumption Deletion

Using θsubsumed clauses leads to redundancy in proof construction in 2 ways.
Assume for simplicity that factoring is unnecessary for this slide and the next.

If C θsubsumes D, C≠D, and D resolves with E giving R1 then
either (i) C resolves with E to give R2 that θsubsumes R1,
or (ii) C θsubsumes R1   **(*)**

e.g. Let C = Px, and D=Pa ∨ Q; then C θsubsumes D.

Suppose D is resolved with {¬Q,R},
then the resolvent {Pa,R} is subsumed by Px (i.e. by C).



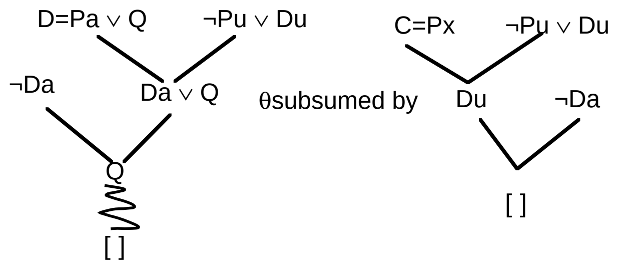
Using D in this case simply leads to further θsubsumed clauses.
Which of the two cases of (*) does this illlustrate?

6biv

If C θsubsumes D, C≠D, and D resolves with E giving R1 then
either (i) C resolves with E to give R2 that subsumes R1,
or (ii) C θsubsumes R1   **(*)**

6bv

Again let C = Px and D = Pa ∨ Q.
Suppose D is resolved with ¬Pu ∨ Du, then the resolvent Da ∨ Q is θsubsumed
by the resolvent formed by resolving C with ¬Pu ∨ Du, i.e. by Du.



Using D like this yields clauses that can be θsubsumed if C is used instead.
Which of the two cases of (*) does this illustrate?

If C θsubsumes D,
then using C instead of D gives a shorter refutation.

All about Tautologies

A clause is a ***tautology*** if <u>all</u> its instances contain an atom and its negation.

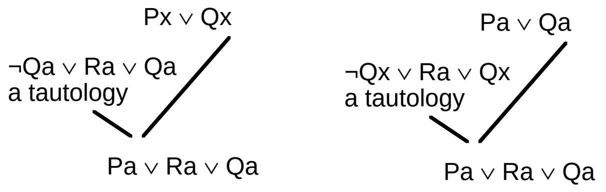
Important Property of Tautologies

If T is a tautology and T is in S, then S is satisfiable iff S-T is satisfiable.
i.e. T can be removed S.

Note: ¬ Qx ∨ Qy is not a tautology but ¬ Qx ∨ Qx is.

Resolving a tautology T with S leads to a resolvent R that is subsumed by S:
e.g. if T=¬Qa ∨ Ra ∨ Qa, S=Px ∨ Q, then R=Pa ∨ Ra ∨ Qa is ***subsumed*** by S

Two Examples:



6ci

Question: Does Prolog have to worry about tautologies? Explain.

Factoring again

6cii

F is a **factor** of E1 ∨... ∨En ∨H if F=(E ∨ H)θ , where θ=mgu{Ei} and E=Eiθ.

Examples Px ∨ Py factors to Px   Qua ∨ Qvv factors to Qaa
Qxy ∨ Qaz ∨ Rxy factors to Qaz ∨ Raz

Factoring is not easy to implement efficiently, but sometimes it **is** necessary.
Earlier we saw that you cannot refute {Px ∨ Py , ¬Pa ∨ ¬Pb} without factoring.

The above definition of a factor concentrates on one predicate symbol at a time. When applying θ it could be that other literals become identical and can be merged.

eg factor Px and Py in Qxx ∨ Qxy ∨ Px ∨ Py gives Qxx ∨ Px
factor Qxx and Qay in Qxx ∨ Qay ∨ Qxy gives Qaa

Or, a factor can be further factored:
eg factor Px,Py in Qxx ∨Qxy ∨Px ∨Py ∨Pa to give Qxx ∨Px ∨Pa
which can be further factored to Qaa ∨ Pa

We use “factor” to mean either a basic factor, as defined above, or the result of several steps of factoring

Exercise: Write down all the factors of Qxx ∨Qxy ∨Px ∨Py ∨Pa

Safe Factoring

6ciii

F is a **safe factor** (or a *reduction*) of C if F is a factor of C <u>and</u> F subsumes C

Exercise: Write down all the safe factors of Qxx ∨Qxy ∨Px ∨Py ∨Pa

Actually, only **safe factoring** (or *reduction*) is necessary:

If C subsumes D, then C |= D. If C is also a factor of D then D |= C. (Why?)
Then C ≡ D and C can replace D.

If C is a factor of D but does **not** subsume D then C *cannot* replace D.

e.g. Qaa doesn't subsume Qua ∨ Qvv; the latter might be needed with some other substitutions for u and v (e.g. Qba ∨ Qbb) so it cannot be discarded.

Another characterisation of safe factor:

C safely factors if F iff for some G and H, C=G∨H, where G or H may be disjunctions of ≥1 literals, and F = H = (G∨H)θ for some θ.
(**Exercise**: Show H <u>is</u> then a safe factor of G∨H)

On the next slide is an outline program for saturation refinement.The program performs subsumption at the end of each stage. To check for backwards / forwards subsumption immediately a resolvent is formed requires `resolveall` to include `Sold` as an argument and to be more sophisticated in its second call from `satref`.

An Outline PROLOG program for Saturation Refinement

6civ

```
satref(Snew,Sold,K):- member([],Snew).
satref([],Sold,K):- writenl(['failed'], fail.
satref(Snew,Sold,K):- Snew≠[],  not  member([],Snew),
    resolveall(Snew,Snew,R1),resolveall(Snew,Sold,R2),
    append(R1,R2,R3), forwardsubsumed(R3,Snew,Sold,R4),
    backsub(R4,Snew,Sold, R5,Snew1,Sold1),
    append(Snew1,Sold1,Sold2),K1 is K+1,
    satref(R5,Sold2,K1).
```

- `satref(New,Old,K)` holds if `New` are resolvents formed from `Old` at stage K of saturation search, and `New` **union** `Old` is unsatisfiable.
- `resolveall(X,Y,Z)` holds if Z are all non-tautologous safe-factored resolvents between clauses in X and Y.
- `forwardsubsumed(X,Y,Z,W)` holds if removing clauses from X that are subsumed by a clause in Y or Z leaves W.
- `backsub(X,Y,Z,X1,Y1,Z1)` holds if clauses from X,Y and Z that are backward subsumed by clauses from X are removed leaving, respectively, `X1,Y1,Z1` .
- Initial call `satref(Init, [ ],0)`.

Using Prolog to program a Saturation Search Theorem Prover

6cv

The Program on 6civ is only an outline. How might clauses be represented in Prolog? The easiest way is to represent a clause as a list of literals, but for quicker pattern matching, maybe a clause could be represented by a pair of lists - the first list of the pair being the positive literals and the second the negative ones. e.g. P(x)∨Q(x)∨¬R(x) becomes the pair ([P(x),Q(x)],[R(x)]). The data is then a list of such pairs. The empty clause would be the pair `( [], [])`. In that case, for example, the condition `not member([],Snew)` in clause 3 would need to be changed to `not member(([],[]),Snew)`. Much of the work is done by `resolveall` which has to form all resolvents between the clauses in its first two arguments and then remove tautologies and form safe factors of the remainder, if any.

As an example of how to form safe factors, consider the clause {P(x,x), P(a,z), P(a,a), Q(z,v), Q(a,u)}. In this case, by factoring literals in pairs, we could reach successively {P(x,x), P(a,a), Q(a,v), Q(a,u)}, {P(a,a), Q(a,v),Q(a,u)}, {P(a,a), Q(a,u)}, which subsumes the original. So, does this approach always lead to all factors?
Exercise: Try to show it does. You might also try to program it and to consider its efficiency. Since most factors are <u>*not*</u> safe factors, there are clever heuristics to detect when safe factoring <u>isn't</u> possible, before trying all possibilities.

Heuristics can also be useful for detecting (strict) θ-subsumption (or when it does not occur). Most clauses don't subsume each other and this can be detected before a full check for subsumption is made. E.g. if C has 5 literals and D has 4, then C cannot (strictly) θ-subsume D. If the predicates in the two clauses don't subsume each other, then nor will the clauses. E.g. P(...)∨P(...)∨Q(....) won't strictly subsume P(..)∨R(...) or vice versa, whatever the arguments. In conclusion, checking for subsumption is not an easy task (see Problem sheets and Slide 6cvii).

Continued from Slide 6cv

Programs mostly test for strict θ-subsumption and simple cases of safe-factoring only. An example of what can happen otherwise is that some factor of C may subsume D, even though it isn't a safe factor. e.g. C=P(x,y) ∨P(y,x) θ subsumes D=P(x,x)∨R (but not strictly); C[x==y] = P(x,x) is a non-safe factor of C which non-strictly subsumes D. You can begin to see the kinds of problems faced when constructing an efficient theorem prover.

As for the program on 6civ, notice that it contains two calls to `append`. It would be more efficient to represent the lists of clauses as *difference lists*, so that they can be appended in constant time. A general difference list representation has the form of a pair of lists, (Z, W), where W is a suffix of Z. E.g. ([1,2,3,4|W], W) represents the list [1,2,3,4]; i.e. the difference between the list consisting of 1,2,3,4 followed by some W, and W. Appending ([1,2,3,4|W],W) to ([5,6|Z],Z) results in the binding W/[5,6|Z] and the new list ([1,2,3,4,5,6|Z],Z). The single clause for append using difference lists is append((X,Y),(Y,Z),(X,Z)).

To resolve two clauses such as ([P(x),Q(x)],[R(x)]) and ([S(v),R(v)][ ]) using Prolog, first use `copy_term` to make a copy of the two clauses with fresh variables (so their variables do not become bound by Prolog when unifying R(x) and R(v)). Then the resolvent is formed, in this case ([P(x1),Q(x1),S(x1)],[ ]), where x1 is the fresh variable for clause1, v1 is the fresh variable for clause2 and ¬R(v1) is resolved with R(x1) with unifier v1/x1. Prolog helpfully propagates this binding to other occurrences of v1 to give the desired resolvent. A copy operation is also needed to implement a subsumption check. The potentially subsuming clause C is copied and the potentially subsumed clause D is grounded to Dg - its variables are bound to new ground terms, using the `numbervars` predicate. Then a check is made of whether C is a subset of Dg for some instance of C. (**Exercise**: Check why this works.) 6cvi

Further Properties of subsumption and factoring. 6cvii

Full subsumption is not usually checked as it can be a theorem proving problem itself that may not terminate. E.g. ¬ P(x)∨P(f(x)) subsumes ¬P(x)∨P(f(f(x))) (i.e. the first clause implies the second) but it does not θsubsume it. Even checking for θsubsumption can be hard. If C is a clause with (say) 5 literals, all positive and all of predicate P, and D is a similar kind of clause, how many possible ways are there that C might subsume D? Lots! One simple way to check for θsubsumption is given in the exercise solutions.

Checking for factoring is also not easy. If every factor of a clause is added then the number of clauses can get very large very quickly. But factors *are* often useful, especially if they instantiate a clause. The factored clause might resolve with fewer clauses than the original, so fewer resolvents are considered. One strategy might be to favour factored clauses. However, the original clause cannot normally be discarded. Factors are also sometimes necessary – see Slide 3ci for an example.

A factor of C is implied by C (**Why?**). If also the factor subsumes C, then it implies C and hence C and the factor are equivalent. We call this *safe factoring* (or *reduction*). In this case C *can* be discarded. Finding safe-factors is worthwhile – the factor is smaller than C as at least two literals have been made identical. Moreover, it has been shown that these are the only kinds of factors that might be necessary in order to find a refutation, although smaller proofs might be found if other factors may be generated and used.

(By the way, note that a refutation is sometimes called a proof, since it is a proof of a contradiction, the empty clause.)

Continued from Slide 6cvii 6cviii

As illustrated on Slides 6biv/v, using subsumed clauses leads to redundancy in a proof. We can show that the following *Property SubFree* holds for refutations formed using saturation search. (See Appendix1 for the proof.) The proof given there uses the notion of *maximum depth of a refutation*, which is the stage in the generation of resolvents in a refutation by saturation search at which the empty clause is formed. A resolvent R is *derived in a refutation at depth k* if k is the stage in the saturation search at which R is derived.

Property SubFree:
Let S be a set of unsatisfiable clauses such that none subsumes any other in S. Then, there is a refutation R from S such that for each clause Ck at depth k≥0 and used in R, Ck is not subsumed by any different clause that is in S or derived from S at a depth ≤k.

The proof of Property SubFree uses this fact (illustrated on slides 6biv/bv):
if C subsumes D and a step in a refutation uses D (resolving with K) to derive R,
then either C subsumes R,
or resolving C and K leads to resolvent R' that subsumes R.
The proof of this fact is not difficult and is left as an exercise.

The proof given in Appendix1 constructs a subsumption free refutation in stages from an arbitrary refutation. It assumes that any factoring needed is carried out at the time of the resolution step, which avoids the restriction to strict subsumption. Try to construct an example of a refutation that violates the SubFree Property and then find another refutation that satisfies the property. If interested, you could follow the construction.

Search Spaces and Refinements 6di

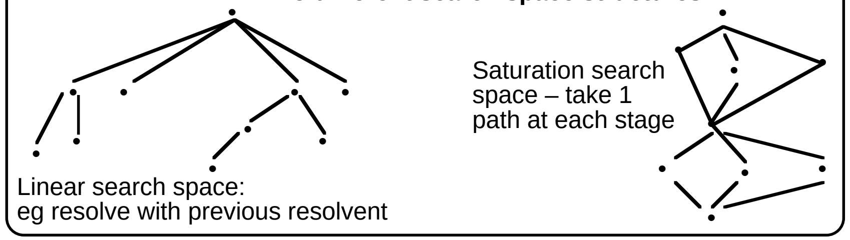
Fact: Unrestricted resolution leads to formation of far too many resolvents.

Some simple restrictions we have already seen are to:

- delete a tautology (clause with a literal and its negation eg {P(x), Q(x,y), ¬P(x)}
- delete a subsumed clause (which is redundant)
- generate factors (maybe just safe factors)

More generally we need further ways to restrict resolution. The most popular methods exploit the syntactic form of a clause to reduce the search space.

Two different search space structures



Types of refinement

6dii

Within a systematic search for a refutation, there will still be choices:

The possibilities give rise to a ***Search Space*** and we may ask how to control its size and in what order to search it, or even does it contain all required proofs.

- *strategy refinements* concern which resolvents will be formed –
e.g. except at first step can only resolve with a previous resolvent

A strategy refinement affects the structure of the search space and so controls its size and the particular refutations that are possible.

- *order refinements* concern the order in which resolvents are formed –
e.g. take resolvents with unit clauses first

An order refinement affects how the search space is searched and (in case only one refutation is desired), which refutation that would be.

In the next few lectures we'll look at a selection of syntactic refinements.

Study of refinements
- of resolution began in about 1970 and continues still
- of Tableaux started in about 1984 and continues still
- of equational reasoning started in about 1985 and continues still
- of natural deduction started in 1960 but has not been actively pursued

Example 1:*Saturation Search*

6diii

uses the "forwards subsumption" strategy refinement,
uses the "subsumption/safe-factoring at end, stage-by-stage, or breadth-first" order refinement,
and looks for one or all proofs within the search space

Other strategy refinements might be to enforce order of literal selection, or to make each step a combination of resolution steps.

Example 2: *Prolog*
uses the "selection rule and linear" strategy refinement,
uses the "clause order and depth-first search" order refinement,
and looks for all proofs in the search space

Other order refinements might be parallel search,
or reject useless paths first – eg goal ?...,L,... and no matching clause for L

<u>Question</u>:
a) Explain how Prolog uses resolution, and.
b) What feature(s) of Prolog makes it unsound in some circumstances
(that is, leads it to give the wrong answer)?
c) Can you suggest additional strategy or order refinements that could be used for Logic Programming (i.e. not just Prolog).

<u>Miscellaneous notes on Search Spaces</u>

6div

A search space for a strategy refinement may be searched in (at least) 3 kinds of ways and in practice aspects of all 3 ways are used.. The saturation search is just one way. Although space consuming, it is in common use. It is guaranteed to find a short proof in the search space if one exists. Two others are:

(1) Each path is taken in order and followed to its conclusion. This is not, in general, possible as a path might not terminate. Instead, a depth d is chosen and all paths are followed to this depth. If no proof is found then d is increased and the process is repeated. The partial proofs found previously to depth d are repeated. This method could miss a short proof if it did not happen to be explored first and d is initially too large.

(2) Search according to some heuristics, often chosen to be data dependent. eg one might be able to remove early on paths that become obviously useless.

We usually require a strategy refinement to be *complete*:

- the search space generated should contain all required solutions (proofs).

although a weaker form ensures the search space contains at least one proof (if any exists).

We also require an order refinement to be search-complete, or *fair*. That is, every branch will eventually be developed, or shown to be redundant. eg depth-first search with no depth limit is generally not fair, because of the possibility of infinite branches.

Summary of Slides 6

6ei

1. Without control, resolution generally produces too many resolvents, many of which are redundant.

2. Some simple control methods are forwards and backwards subsumption, tautology deletion and safe factoring. Subsumption removes clauses that would most likely, if used, lead to longer derivations. Tautologies, if used, lead to subsumed resolvents. Safe factoring replaces a clause with an equivalent, but smaller clause.

3. Generally, subsumption detection is limited to strict θ-subsumption, as other, stronger forms of subsumption are expensive to detect, and, in the case of general subsumption may be undecidable

4. Deletion of sbsumed clauses and tautologies does not affect unsatisfiability.

5. Control of resolution (and indeed of other techniques) is a much researched area, and continues to be so. Most strategy refinements are syntactic. Semantic refinements tend to be in specialised domains.

6. A simple Prolog program was introduced for saturation search. A saturation search proceeds by stages, generating resolvents in groups, at each stage using at least one resolvent from the immediately preceeding stage in every resolution step and no resolvents generated in the current stage.

7. A search space is the set of possible steps that can be made. For a given problem and general strategy there may be several different search spaces that can be generated, depending on the particular *strategy refinement*. Any search space may often be searched in different ways as well, depending on the *order refinement*.

For example, in Logic Programming, different search spaces will result depending on the selection of literal from each query. In Prolog, the selection is always the leftmost literal of the most recent literals added to the query. But other choices are possible. Prolog searches its search space from left to right and depth-first. It is also possible to search each branch to depth 1, then each branch to depth 2 and so on, although this uses up rather more space than depth-first. But depth-first search is not complete if branches may be infinite, as they often are in Prolog. (**Exercise**: Find such an example.)

8. A refutation (using subsumption) can constructively be transformed into a simpler refutation that does not use subsumption.

Obstacles to the Automation of Reasoning (Wos) (For interest only)

1 Data retention: the program keeps too much information in its data base.

Remedy: throw away some data, perhaps needing to recompute when needed. BUT which data to delete?

2 Redundant information: the program keeps generating the same information, or subsumed information, over and over again.
eg if A in data, no need for A or B. Or $\forall x. A(x)$, no need for $A(b)$.
Remedy: throw away the redundant info.
BUT: requires checking to find redundancy.

3 Inadequate focus: the program gets lost too easily and wanders down useless paths.

Remedy: use some sort of weighting to decide on useful deductions.
BUT how can this weighting be formulated?

1,2,3 usually result in the program generating too many conclusions, many of which are redundant or irrelevant.

4 The inference rules may be too fine-grained, resulting in the problems 1-3, or they may be too large, or too restrictive, resulting in the problem of too little information being drawn. (We'll consider several examples.)

5 There are no general guidelines for selecting the appropriate means to control the problems in 1-4.

Remedy: control by strategy
– syntactic kinds prohibit certain paths – easy for the program;
– semantic kinds harder – but focus on paths likely to solve problem.
– More generally, could allow deductions only if they yield facts;
– or use equations from left to right only – e.g. a=b can be used to replace a by b but not b by a.

– e.g. simulation of LP by resolution: start from negative clause and generate next resolvent from current resolvent and an input clause; select literals in a fixed order; order input clauses; could check constraints to remove failing paths quickly;

6 The program may not use an appropriate representation of the information pertinent to the problem.
Remedy: lots of experience.

7 Ordinary computing difficulties such as indexing in the database and finding appropriate information.