AUTOMATED REASONING

SLIDES Extra:

MISCELLANEOUS
- Semantic Control
  - Model Failure for Horn clauses
  - Abstractions
  - Nagging
- Hyper-linking

KB - AR - 08

Using Semantic Information to prune the search space Eai

The slides Ea give some examples of how additional semantic information might be used to inprove theorem provers. This is an area which has not been exploited much except in data base applications, where semantic information is used to tailor queries: for example, by pruning queries which cannot succeed, or by instantiating variables when there is only one instance that will satisfy a query.

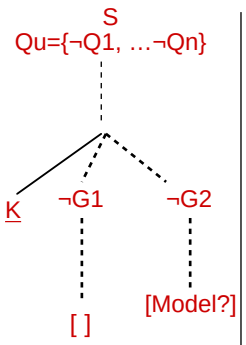
First, recall that finding a model of a Horn clause program that falsifies some goal G (or intermediate goal), represented by ¬G in the tableau, shows G is not provable. Suitable models are usually models with small domains. For instance, models that use a domain which mirrors typing information allow to prune tableau branches in which certain literals are ``badly typed''.

A second way to prune branches that cannot succeed, is to find some extra information (EI) that is consistent with a program P and which together with P implies ¬G. Then, since P+EI is consistent and P+EI |= ¬G, it is also the case that not(P |= G). If this were not so and P|= G, then since P+EI is consistent there is a model M of P and of EI, which by P |= G must also be a model of G, contradicting that P+EI |= ¬G. The argument only works if M exists, which is guaranteed only if P+EI is consistent.

This method can be useful not only to prune branches, but to complete branches in the only possible way, as the example on Eavi shows. In that example, the extra information states that P is a function of its first 2 arguments, and that information together with previously completed branches closes other branches too.

Semantic Control: Model Failure for Horn Clauses (Bundy)

Let S = a set of given *Horn* clauses
Let Qu be goal clause {¬Q1,…,¬Qn} i.e. "show Q1 and Q2 and ... and Qn"



Notice that a closed tableau formed using Horn clauses and started from a top clause with no positive literals has only negative literals at internal nodes and only positive atoms at leaf nodes.

Exercise: Show this by induction on the depth of a closed tableau.

In the tableau on the left, S $|^{=}$ G1 because S + ¬G1 is inconsistent. This can be shown because S are Horn clauses. **(Exercise:** If they were not Horn clauses why could it not be safely concluded?)

On the other hand, suppose a model exists for S + ¬ G2, then we know it is not the case that S |=G2.

It may be possible to find a simple model of S+ ¬ G2 and so prevent searching for a non-existent (and possibly infinite) closed tree below ¬ G2.

Eaii

Model Failure for Horn Clauses - simple example Eaiii

Suppose a model exists for S + ¬ G2, then can conclude that not(S $|^{=}$ G2).
(If S $|^{=}$ G2 then S+¬G2 is inconsistent and would have no models.)

A very **simple example** of the idea:

¬G2 = ¬Q(b) (as a Prolog goal = ?Q(b))
S = {P(x), ¬Q(x)} , Q(a) , {P(x) , ¬R(x)} , R(b)

Try to construct a tableau beneath ¬G2;
in fact, it can be abandoned immediately as there is a model M of S + ¬Q(b):

Let M make {P(a),P(b),Q(a),R(b)} True and other atoms False.
Then M makes all conclusion literals True (and so satisfies S) and
M makes Q(b) False (so ¬Q(b) True).

Question: How do we know there is a model of S+¬Q(b)? And how do we know we can construct M as above?

Answer: Q(b) does not unify with any conclusion literal in S, so choose all conclusion atoms true and Q(b) false..

Of course, in Prolog, the reason the branch is abandoned is exactly because there's no match.

A Useful Kind of Failure Model

Eaiv

Potentially useful models for terminating a branch usually have small domains.
e.g. in a problem using a database that includes individuals of types
Man or Woman, then a model with domain {M,W} may be appropriate.

Example: A program *Pr* includes:
{gm(x,y),¬m(x,z),¬p(z,y)} , m(mary,fred), etc.
p(x,y) reads x is a parent of y and m(x,y) reads x is mother of y.
Goal is ?gm(john, x) i.e. ¬G2 = ¬∃x.gm(john,x)
(find x: John is x's grandmother)

Suppose the database respects the type of the1st argument of predicate m;
(i.e. all persons in 1st arg position of m are Women).
A failure model M (with Domain = {M,W}) is:

each Woman constant maps to W; each Man constant maps to M
eg john maps to M, mary maps to W, etc.
gm(W,x) and m(W,x) are both true for x in {M,W};
p(x,y) is always true for any x and y; other atoms are false.

Pr is true in M but ¬G2 is true also (gm(john,x) is false in M for every x,
hence for no x is gm(john,x) true).
Hence it is not the case that *Pr* $\models$ ∃x.gm(john,x).

Using Extra Information (EI)

Eav

Suppose a programmer has **extra information (EI)** about the predicates used in a Horn program P that is being used to show goal G (i.e top query = ¬G).

Assume P+EI is consistent. Perhaps P |= G, but for some intermediate goal G1 on one branch of the search space, P+EI + G1 |= (ie P+EQ+G1 is inconsistent)
Then P+ E1 |= ¬G1 and hence not(P |= G1).
(**Exercise**: Why does not(P |= G1) follow?)

EI can be used to remove branches of the search space that will fail because P |= G1 is not true.

Example:
In the gm program of Eaiii, assume current goal is gm(john,x);
i.e. G1 = ∃x.gm(john,x)

extra information (EI) might include ¬woman(john) , {¬gm(x,y) ,woman(x)}, etc.

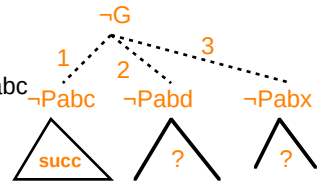
Then, P+EI is consistent and P+E1 |= ¬ ∃x gm(john,x)
(so it is not the case that P |= G1).

Exercise:
1) Why does P+EI |= ¬ ∃x gm(john,x)?
2) Why does the consistency of P+EI matter to conclude not (P |= goal)?

Using Extra Information (Example)

Eavi

Example: Let EI = {¬Pxyz ,¬Pxyw , z=w} ,
¬b=c , ¬a=b, etc. i.e. predicate P is a
function of its first 2 arguments.



Suppose branch 1 closes; then Program |= Pabc
What about branches 2 and 3?

Branch 2: Program+EI |= ¬P(a,b,d);
So not(Program |= P(a,b,d))

Branch 3: Can deduce x==c. Why?

Idea can be extended by including in clause bodies redundant atoms that are implied by the clause bodies. If these are false in a model can fail the goal.

eg clause {Q, ¬A, ¬B} is extended to {Q, ¬A, ¬B, ¬C}, where A & B → C.

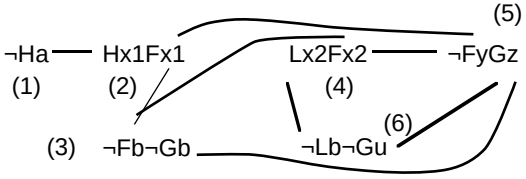
The new goal literals derived from using the clause will be ¬A, ¬B, ¬C.
Suppose a model of the program P makes C false;
then it makes either A or B false and P cannot imply both A and B,
so this branch of the search space cannot succeed and another must be tried
Before extending by C the goal literals would have been only ¬A and ¬B.

Hyper-Linking

Ebi

Hyper-linking (Plaistead, JAR, 1992) uses ideas from hyper-resolution and Davis Putnam. It is a good example of combining different techniques

- Potential unifiers are used to derive a suitable set of instances for each clause - those that might be used in a ground refutation.
- These clauses can be tested for a ground refutation by Davis Putnam
- A ***hyper-link*** θ of clause C is a successful combination of a set ofunifiers, one for each literal in C.
- All instances Cθ of C arising from applying a hyperlink θ to C are taken.



If Cθ is not ground then it is ***Grounded*** by substituting a new name for every varaible. eg {Fx,Gx} is grounded to {Fd, Gd} using d; it indicates no special instance should be used - any instance is possible.

Hyper-links and the resulting instances:
(1): ¬ Ha (2): {Ha Fa} (3): {¬Fb,¬Gb} (4): {Lb, Fb}
(5): {¬ Fx2, Gz } , {¬Fx2, Gb}, {¬Fx1,Gz}, {¬Fx1,Gb} ground to
{¬ Fd, Gd}, {¬ Fd,Gb}, {¬ Fd, Gd}, {¬Fd, Gb}
(6): {¬Lb, ¬Gz} grounds to {¬ Lb, ¬Gd}

If a refutation cannot be found with the instances from round 1, they are added to the graph and a second round of hyper-links and instances are generated; these will include the previous ones.

- e.g. given **Pa, Pfx ∨ ¬Px, ¬Pffa.**

- The first set of instances is **Pa, Pfa ∨ ¬Pa, Pffa ∨ ¬Pfa, ¬Pffa.**
- The 2nd and 3rd instances come from using the self-resolving link on the recursive clause – i.e. the link between Pfx and ¬Px.

- This set is not unsatisfiable so a second round of hyper-linking is made. The (new) instances from round 1 are added to the graph and new links derived from the parent clauses are added.

- When the above instances are added, the extra instances **Pffffa ∨ ¬Pffffa, Pffa ∨ ¬Pfa** are generated.

Exercise: Show this.

- Together with the previous ones, these are unsatisfiable.

- Method is clearly sound. Completeness is a bit more difficult. The proof is based on a notion of distance between a clause and the clause that eventually provides the "grounding" bindings.

Ebii

Hyper-Linking: Ebiii

The *Hyper-linking* procedure shown on Ebi has two parts; part (i) is based on linking potentially complementary literals – a method borrowed from connection graphs [Kowalski] and hyper-resolution; part (ii) is based on the Davis-Putnam procedure and might be viewed as a way of adapting DP to first order clauses.

The difficulty with adapting DP to the general case is that it is not known which ground instances of clauses are necessary for finding a refutation. Since a set of clauses is unsatisfiable if a set of instances of those clauses is unsatisfiable, one could just keep enumerating sets of ground instances and checking them for unsatisfiability. Hyper-linking helps us choose a set of instances that might potentially be unsatisfiable. (**Question:** Why would a clause instance that is not a hyper-link not be part of any set of unsatisfiable ground instances of the given clauses? **Hint:** Recall the definition of a pure literal in DP.)

In part (i) of Hyper-linking, all potentially complementary pairs of literals are linked and for each clause are computed all instances of it for which there is a set of compatible links - one from each literal. A set of links is *compatible* if the unifiers on those links can be *combined* into a single unifier. This is achieved by combining the equations for each unifier and applying the unification algorithm. (See details of example on Slide Ebi.)

In part (ii) these instances are grounded by substituting any variable remaining in an instance by (the same) new ground term and DP is applied. If the procedure succeeds then a set of unsatisfiable ground instances has been found. Otherwise, all the new instances (before grounding) are added to the graph and a new round of hyper-linking is made. The second generation of instances will, of course, include the previous ones.

On slide Ebi, note that the hyper-link of a ground clause (eg clause (1)) is just the clause itself. For clause (2), there is one binding for x1, from the link between clauses (1) and (2). This combines successfully with the binding between clauses (2) and (5) and yields x1==a. That is, apply the unification algorithm to x1=a and x1=y. The combination of bindings x1==a and x1==b (taking links between clauses (2) and (3) does not lead to a unifier. For clause (5) there are potentially 4 different hyper-links, according as to which combination of links to clause (5) is taken. In fact, there are only 2 distinct instances.

In case there are recursive clauses, with links between literals in a single clause, the hyper-linking is carried out by making a variant copy of the clause for the destination literal. On slide Ebii there is a link between Pfx and ¬Px in the clause Pfx ∨¬Px. To find the hyper-link instances first make a variant copy: Pfy ∨¬Py. Then one could take the link between Pfx and ¬Pffffa (fx=fffa) and between ¬Px and Pfy (x=f(y)) in the variant copy. These are compatible yielding x==ffa. Similarly, one could take the link between ¬Px and Pa and between Pfx and ¬Py in the variant copy, yielding x==a.

Ebiv

USING ABSTRACTIONS (Plaistead)

Eci

Another way to reduce the search space is to simplify the given clauses (called *abstraction*) and then look for a refutation amongst the simpler clauses.

If the abstraction is chosen so that the real refutations and the abstracted refutations (using the simpler, abstracted, clauses) have a similar structure and hence that existence of a Real refutation → existence of an abstracted refutation, then one can conclude the contrapositive: that no abstracted refutation → no real refutation and start searching for refutations first among the abstracted clauses.

In case an abstracted and a real refutation exists, then it may be possible to construct the Real refutation from the abstracted refutation.

An *abstraction function* f maps a clause to a set of clauses, s.t. f([]) = [] and
- If R(C1, C2) = C3 and D3 is in f(C3), then ∃ D1 in f(C1) and D2 in f(C2) such that R(D1,D2) subsumes D3. (i.e. f respects resolvents)

There are syntactic abstractions: "forget arguments" or "simplify arguments", or semantic abstractions: "reduce arguments to their types".

e.g."simplify args" f({P(a,f(x)),¬Q(x)} = {P(a,y),¬Q(x)}.
Exercise: Check that this abstraction respects resolvents.

The method is a generalisation of model failure (See Plaistead AI 82,84)

Using Abstractions to reduce the Search Space: Ecii

Slides Ec show the use of abstractions. An *abstraction* function f maps a clause to a set of simpler clauses, such that, if the set of abstracted clauses of a clause-set S (denoted by f(S)) does not possess a refutation, then nor does the clauseset S possess one. A simple abstraction f1 would be to "forget arguments" (i.e. f1({P(a,x), ¬Q(x)}) = {P,¬Q}. Clearly, in this case, if the abstractions cannot be refuted, then nor could the original clauses be, for at least the predicates would have to resolve. In simple cases (and the only ones considered here) f will map a clause to a single clause.

An abstraction f must satisfy some simple requirements (see slide Eci).

The first requirement is that f *respects resolvents*.
e.g. The "forget arguments" abstraction f satisfies this property:
Let C1=P(a)∨ P(b) and C2= ¬P(x) ∨ Q(x) resolve on P(a) and ¬ P(x).
The D3 of the property is f(P(b)∨ Q(a)) = P∨ Q.
Resolving D1=f(C1) and D2=f(C2) gives Q, which subsumes D3.

The second property is that f *respects subsumption*. i.e.
If C1 subsumes C2 then ∀D2 in f(C2) ∃D1 in f(C1) s.t. D1 subsumes D2.
e.g. The "simplify args" abstraction f, in which non-variable non-ground arguments are replaced by variables satisfies this property:
Let C1=P(g(x), y), C2 = P(g(f(a)),a), then f(C1) = P(u,y), f(C2) = P(g(f(a)),a)
C1 subsumes C2 and f(C1) subsumes f(C2).

The third requirement is that f maps the empty clause to itself.

Together, these 3 properties guaranteee that if there is a refutation of the original clauses, there will also be one of the abstracted clauses. The proof is by induction on the depth *n* of a refutation.

The case when n=1 is easy. The empty clause is obtained by resolving facts C1 and ¬C2.
Since f([ ]) = [ ], in order that the resolvent of f(C1) and f(¬C2) subsumes f([ ]), it must also be [ ].

For the induction step, let n>1 and assume as inductive hypothesis (IH) that if R is a refutation of depth <n, then f(R) is also a refutation. Here we assume a step in a refutation is either a factoring step or a resolution step. Let R be a refutation of depth n of initial clauses S and consider the refutation of depth n-1 formed by treating the first derived clause as an additional given G. By the IH there is a corresponding abstracted refutation of S+{G}.

Let C1, C2, C3 be as given in requirement 1, where C1 and C2 are initially given clauses. Then f(C3) is subsumed by R(f(C1), f(C2)). Let all abstractions of the first resolvent G be formed, then each abstraction of G will be subsumed by the resolvents such as R(f(C1), f(C2)). By properties of subsumption (shown in Slides 6), there is also a refutation using these subsuming resolvents in place of the abstractions of G. Therefore, there is an abstract refutation from the original clauses.

Eciii

Parallel Use of Abstractions Edi

Abstractions are also used in the example of *Nagging* on Slide Edii. Here the search for refutations among abstractions is made in parallel. It is assumed that the set of abstracted refutations can be arranged as a tree. Branches in this search tree that contain no solutions correspond also to branches, or groups of branches, in a non-abstract search tree, but the abstracted branches may be abandoned more quickly. Similarly, if the query is existential – ie a binding for some variables in the initial goal is required – such a binding might be found more simply from an abstracted search space. The N-queens example is like this.

The difference with the previous abstraction idea (on Eci) is that it is the whole search tree that is abstracted. As an example, suppose the search space is that of a ME tableau represented as a tree of chains. A master process will search one branch of the abstraction of this tree and parcel out the parallel search of abstractions of the other branches to *minions*. If all abstracted branches beneath a node terminate in failure the original node would do so too, so the branch can be abandoned. Each minion will use a similar approach, abstracting nodes further (if possible). Examples of abstractions are given on the slides.

The N-Queens abstraction abstracts a term representing a partial placement of queens (row by row) to a term representing the same placement but taken column by column. In many cases it is easier to fail to find an extension with the abstracted representation.

NAGGING - using abstractions in parallel (JAR 12/97)

Nagging is a technique that allows a parallel search of a search space - for simplicity, suppose the search space is a tree. Let a node T of a search space have subtrees T1, …, Tn to be searched.

Let f be a function such that f(T) is a simpler search space than T and f(T) has a solution if T has a solution. So no solution in f(T) => no solution in T.

The *master* node searching the tree below T will offer some of its subtrees to "*minions*" , while it continues its own search. If a minion finds no solution in f(Tk) (say) then the subtree below Tk can be abandoned. Each minion can itself use nagging!

What makes a good function f?
- f should be **informative** - f(T) must sometimes contain no solutions;
- f should be **reductive** - f(T) should be smaller than T;

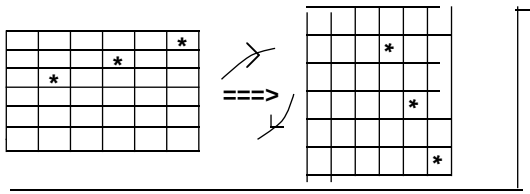
- f(T) is **permutative** if branches of T are interchanged/terminated in f(T).
- f is an **equivalence abstraction** if it maps terms to equivalence classes; e.g. b and c might be identified and each occurrence of b/c is mapped to a term called [bc](b) or [bc](X) for variable X (similarly for c). In this case the clauses that make the search space are altered rather than the search tree.

(See Ediii).

Edii

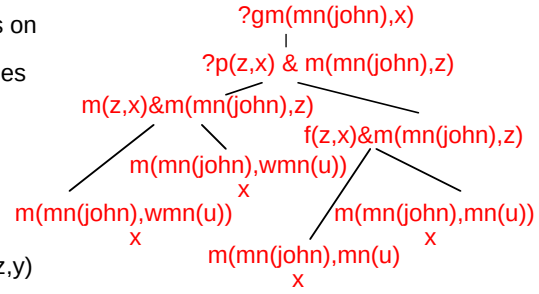
Example: **n-queens** (agreed a rather special case) Ediii

Consider searching all states below a node in which the queens are already placed as in left diagram (placing queens row by row). (No solutions).
Now rotate through 90 and search again - much easier to fail?



gm(x,y) if p(z,y) & m(x,z)
m(mary, fred), etc.
p(x,y) if m(x,y)
p(x,y) if f(x,y)
f(joe, ann), etc.
(x,y,z variables)

Example: gm example; see clauses on right; goal: gm(john, w)
Abstraction: use equivalence classes mn (for man) and wmn (for woman)
New clauses: m(wmn(x),mn(y)); m(wmn(y),wmn(x)); f(mn(y),mn(x)); f(mn(y),wmn(x)), + others.
Minion gets tree to search below ?gm(mn(john),x).
This is much smaller than original.
If first clause is gm(x,y) if m(x,z)&p(z,y) search fails almost immediately.



Equivalence Abstraction Ediv

The *equivalence abstraction* is an abstraction applied to the clauses which generate the abstracted tree. Terms in a clause are mapped to some representative, so several terms might all be mapped to the same representative. This is similar to using types as representatives of terms. e.g. terms b and c might both be mapped to a representative [bc](x) (x is a variable here), representing some term in the equivalence class {b,c}. But they might also, in some clauses, be mapped to [bc](b) or [bc](c) – which depends on the initial abstraction translation. The term [bc](x) will match any other abstract term in the equivalence class [bc] (e.g. [bc](b)).

In the "gm" example on Ediii there are 2 equivalence classes, {men} and {women}. Each term is mapped to mn(x) or wmn(x) as appropriate. Thus john could be mapped either to mn(x) or to mn(john), mary could be mapped either to wmn(x) or to wmn(mary), etc. (Here we use mn to represent the large equivalence class [man1, man2, man3, ...].) By using the more general mapping, all the data clauses reduce to 1 of 4 forms: m(wmn(y),wmn(x)), m(wmn(y),mn(x)) and a similar two for the predicate f. The original negated conclusion is mapped to gm(mn(john),x). The search tree is much smaller than the original would have been.

The non-fact clauses might also have been abstracted. For example, p(x,y) if m(x,y) could become p(wmn(x),y) if m(wmn(x),y).

Summary of Slides Extra

Ee

1.Whilst syntactic control refinements for resolution have been thoroughly investigated, there has been little investigation of semantic control.

2. We presented Semantic failure for controlling Horn clause programs, in which a model of the program that satisfies the negation of the current goal is sought. Finding such a model shows that the goal is not derivable. The model may be an abstraction of the given clauses, or it may be a model of additional information.

3. Abstractions of either individual clauses or of whole search spaces were briefly introduced. The latter leads to a way to control the search space using parallel search of simpler problems, called Nagging.

4. Hyper-linking uses links between potentially complementary literals to find sets of ground instances of given clauses that will be unsatisfiable. The DP method is then applied to test each set. In fact, other methods could be used in place of DP. This is a good example of how different techniques may be combined to generate new or extended methods. Here, hyper-linking can be thought of as an extension of DP to first order clauses.