Declarative Programming

Inductive Logic Programming

Yann-Michaël De Hauwere
ydehauwe@vub.ac.be
• Introduction to declarative programming

• Clausal logic

• Logic programming in Prolog

• Search

• Natural language processing

• Reasoning using incomplete information

• Inductive logic programming
• Introduction to declarative programming
• Clausal logic
• Logic programming in Prolog
• Search
• Natural language processing
• Reasoning using incomplete information
• Inductive logic programming
### Reasoning with incomplete information

<table>
<thead>
<tr>
<th><strong>default reasoning</strong></th>
<th><strong>abduction</strong></th>
<th><strong>induction</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>assume normal state of affairs, unless there is evidence to the contrary</td>
<td>choose between several explanations that explain an observation</td>
<td>generalize a rule from a number of similar observations</td>
</tr>
<tr>
<td>“If something is a bird, it flies.”</td>
<td>“I flipped the switch, but the light doesn’t turn on. The bulb mist be broken.”</td>
<td>“The sky is full of dark clouds. It will rain.”</td>
</tr>
</tbody>
</table>
Inductive Logic Programming

Inductive reasoning

Given:

B: background theory (clauses of logic program)
P: positive examples (ground facts)
N: negative examples (ground facts)

Find a hypothesis H such that

H “covers” every positive example given B
\[ \forall p \in P: B \cup H \models p \]

H does not “cover” any negative example given B
\[ \forall n \in N: B \cup H \not\models n \]
Inductive reasoning & learning

• Concept learning tries to find a suitable concept in a description space where descriptions are related via generalization/specialization relationships. Examples are at the “bottom” of the generalization hierarchy.

• A concept is suitable if it covers (generalizes) all positive and none of the negative examples.

• Learning capabilities depend on the characteristics of the description space: too rough makes learning impossible, too fine leads to trivial concepts (e.g. when the description space supports disjunction).

• A well-known algorithm is Mitchell’s candidate elimination algorithm where upper and lower bounds of possible solutions are updated according to input examples.
Inductive Logic Programming

Inductive reasoning vs abduction

Abduction:
given a theory $T$ and an observation $O$,
find an explanation $E$ such that $T \cup E \models O$

Try to adapt the abductive meta-interpreter:
inducible/1 defines the set of possible hypothesis

```prolog
induce(E,H) :-
    induce(E,[],H).
induce(true,H,H).
induce((A,B),H0,H) :-
    induce(A,H0,H1),
    induce(B,H1,H).
induce(A,H0,H) :-
    clause(A,B),
    induce(B,H0,H).
induce(A,H0,[(A:-B)|H]) :-
    inducible((A:-B)),
    not(member((A:-B),H0)),
    induce(B,H0,H).
induce(A,H0,H) :-
    member((A:-B),H0),
    induce(B,H0,H).
```
Inductive Logic Programming

Inductive reasoning vs abduction

bird(tweety).
has_feathers(tweety).
bird(polly).
has_beak(polly).

inducible((flies(X):-bird(X),has_feathers(X),has_beak(X))).
inducible((flies(X):-has_feathers(X),has_beak(X))).
inducible((flies(X):-bird(X),has_beak(X))).
inducible((flies(X):-bird(X),has_feathers(X))).
inducible((flies(X):-bird(X))).
inducible((flies(X):-has_feathers(X))).
inducible((flies(X):-has_beak(X))).
inducible((flies(X):-true)).
Listing all inducible hypothesis is impractical. Better to **systematically search** the **hypothesis space** (typically large and possibly infinite when functors are involved).

**Avoid overgeneralisation** by including **negative examples** in search process.
Inductive Logic Programming

Inductive reasoning

Inductive reasoning:
a hypothesis search involving successive
generalisation and specialisation steps of a current hypothesis

ground fact for the predicate of which a definition is to be induced that is either true
(+ example) or false (- example) under the intended interpretation

<table>
<thead>
<tr>
<th>example</th>
<th>action</th>
<th>hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ p(b,[b])</td>
<td>add clause</td>
<td>p(X,Y).</td>
</tr>
<tr>
<td>- p(x,[])</td>
<td>specialize</td>
<td>p(X,[V</td>
</tr>
<tr>
<td>- p(x,[a,b])</td>
<td>specialize</td>
<td>p(X,[X</td>
</tr>
<tr>
<td>+ p(b,[a,b])</td>
<td>add clause</td>
<td>p(X,[X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p(X,[V</td>
</tr>
</tbody>
</table>
Learning append/3

?- induce_rlgg([+append([1,2],[3,4],[1,2,3,4]),
+append([a],[a],[a]),
+append([],[a],[a]),
+append([],[1,2,3],[1,2,3]),
+append([2],[3,4],[2,3,4]),
+append([],[3,4],[3,4]),
-append([a],[b],[b]),
-append([c],[b],[c,a]),
-append([1,2],[1,3]),
], Clauses).

RLGG of append([1,2],[3,4],[1,2,3,4]) and append([a],[a],[a]) is
append([X\|Y],Z,[X\|U]) :- [append(Y,Z,U)].
Covered example: append([1,2],[3,4],[1,2,3,4])
Covered example: append([a],[a],[a])
Covered example: append([2],[3,4],[2,3,4])
RLGG of append([],[],[]) and append([],[1,2,3],[1,2,3]) is
append([],X,X) :- []
Covered example: append([],[],[
Covered example: append([],[1,2,3],[1,2,3])
Covered example: append([],[3,4],[3,4])
Clauses = [[append([],X,X) :- []],
(append([X\|Y],Z,[X\|U]) :- [append(Y,Z,U)]].]
Generalising clauses: $\theta$-subsumption

A clause $c_1$ $\theta$-subsumes a clause $c_2$ $\iff$ $\exists$ a substitution $\theta$ such that $c_1\theta \subseteq c_2$

\[
H_1;...;H_n :\neg B_1;...;\neg B_m \\
H_1 \lor ... \lor H_n \lor \neg B_1 \lor ... \lor \neg B_m
\]

**$\theta$-subsumes**

\[\text{element}(X,V) :- \text{element}(X,Z)\]
\[\text{element}(X,[Y|Z]) :- \text{element}(X,Z)\]

using $\theta = \{V \rightarrow [Y|Z]\}$

**$\theta$-subsumes**

\[a(X) :- b(X)\]
\[a(X) :- b(X), c(X)\]

using $\theta = \text{id}$
Inductive Logic Programming

Generalising clauses:
θ-subsumption versus \( \not\models \)

\( H_1 \) is at least as general as \( H_2 \) given \( B \) \( \iff \)

\( H_1 \) covers everything covered by \( H_2 \) given \( B \)
\[ \forall p \in P: B \cup H_2 \not\models p \Rightarrow B \cup H_1 \not\models p \]

\( B \cup H_1 \not\models H_2 \)

\text{clause } c_1 \ \theta\text{-subsumes } c_2 \Rightarrow c_1 \models c_2

The reverse is not true:

\begin{align*}
\text{list([], } &. \ % c_0 \\
\text{list([H1,H2|T]) } :&= \text{ list(T). } \ % c_1 \\
\text{list([A|B]) } :&= \text{ list(B). } \ % c_2
\end{align*}

\( c_1 \models c_2 \), but there is no substitution \( \theta \) such that \( c_1 \theta \subseteq c_2 \)
Inductive Logic Programming

Generalising clauses:
Testing for $\theta$-subsumption

A clause $c_1$ $\theta$-subsumes a clause $c_2$ $\iff$ $\exists$ a substitution $\theta$ such that $c_1\theta \subseteq c_2$

\[
\text{theta_subsumes}((H1:-B1),(H2:-B2)):-
\text{verify}((\text{ground}((H2:-B2)),H1=H2,\text{subset}(B1,B2))).
\]

\[
\text{verify}(\text{Goal}) :-
\text{not}(\text{not}(\text{call}(	ext{Goal}))).
\]

\[
\text{ground}(	ext{Term}) :-
\text{numbervars}(	ext{Term},0,N).
\]

% instantiate vars in Term to terms of the form
% '$VAR'(i) where i is different for each distinct
% var, first i=0, last = N-1
Generalising clauses:
Testing for $\theta$-subsumption

A clause $c_1$ $\theta$-subsumes a clause $c_2$ $\iff$ $\exists$ a substitution $\theta$ such that $c_1\theta \subseteq c_2$

?- theta_subsumes((element(X,V):- []),
                (element(X,V):- [element(X,Z)])).
yes.

?- theta_subsumes((element(X,a):- []),
                (element(X,V):- [])).
no.
Generalising clauses: generalising two clauses

\[ \text{generalising two clauses} \]

\[ \text{a1: member(1,[1]).} \]

\[ \text{a2: member(z,[z,y,x]).} \]

\[ \text{a3: member(X,[X|Y]).} \]

\[ \text{subsumes using } \theta = \{X/1, Y/[]\} \]

\[ \text{subsumes using } \theta = \{X/z, Y/[y,x]\} \]

\[ \text{member(X,[X|Y]).} \]

happens to be the **least general** (or most specific) generalisation because all other atoms that \( \theta \)-subsume a1 and a2 also \( \theta \)-subsume a3:

\[ \text{member(X,[Y|Z]).} \]

\[ \text{member(X,Y).} \]
Generalising clauses:

generalising two clauses

t₁ is more general than t₂ ⇔ for some substitution \( \theta \): t₁\( \theta \) = t₂

greatest lower bound of two terms: unification

specialization = applying a substitution

least upper bound of two terms: anti-unification

generalization = applying an inverse substitution (terms to variables)
Generalising clauses:

dual of unification

compare corresponding argument terms of two atoms, replace by variable if they are different replace subsequent occurrences of same term by same variable

?- anti_unify(2*2=2+2,2*3=3+3,T,[],S1,[],S2).
T = 2 * _G191 = _G191 + _G191
S1 = [2 <- _G191]
S2 = [3 <- _G191]

will not compute proper inverse substitutions: not clear which occurrences of 2 are mapped to _G191 (all but the first) BUT we are only interested in the $\theta$-LGG
:- op(600,xfx,'<-').
anti_unify(Term1,Term2,Term) :-
    anti_unify(Term1,Term2,Term,[],S1,[],S2).
anti_unify(Term1,Term2,Term1,S1,S1,S2,S2) :-
    Term1 == Term2,
    !.
anti_unify(Term1,Term2,V,S1,S1,S2,S2) :-
    subs_lookup(S1,S2,Term1,Term2,V),
    !.
anti_unify(Term1,Term2,Term,S10,S1,S20,S2) :-
    nonvar(Term1),
    nonvar(Term2),
    functor(Term1,F,N),
    functor(Term2,F,N),
    !,
    functor(Term,F,N),
    anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2).
anti_unify(Term1,Term2,V,S10,[Term1<-V|S10],S20,[Term2<-V|S20]).

Generalising clauses:
Generalising clauses:

\begin{verbatim}
anti_unify_args(0,Term1,Term2,Term,S1,S1,S2,S2).
anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2):-
    N>0,
    N1 is N-1,
    arg(N,Term1,Arg1),
    arg(N,Term2,Arg2),
    arg(N,Term,ArgN),
    anti_unify(Arg1,Arg2,ArgN,S10,S11,S20,S21),
    anti_unify_args(N1,Term1,Term2,Term,S11,S1,S21,S2).
\end{verbatim}

\begin{verbatim}
subs_lookup([T1<-V|Subs1],[T2<-V|Subs2],Term1,Term2,V) :-
    T1 == Term1,
    T2 == Term2,
    !.
subs_lookup([S1|Subs1],[S2|Subs2],Term1,Term2,V):-
    subs_lookup(Subs1,Subs2,Term1,Term2,V).
\end{verbatim}
Inductive Logic Programming

Generalising clauses:

C1 is more general than C2 \iff \text{for some substitution } \theta: C1 \theta \subseteq C2

greatest lower bound of two clauses: \theta\text{-MGS}

specialization = applying a substitution and/or adding a literal

least upper bound of two clauses: \theta\text{-LGG}

generalization = applying an inverse substitution and/or removing a literal
Generalising clauses:

computing the \( \theta \) least-general generalization

similar to, and depends on, anti-unification of atoms

but the body of a clause is (logically speaking) unordered

therefore have to compare all possible pairs of atoms (one from each body)

?- theta_lgg((element(c,[b,c]):-[element(c,[c])]),
          (element(d,[b,c,d]):-[element(d,[c,d]),element(d,[d])]),
          C).

C = element(X,[b,c\{Y\}]):-[element(X,[c\{Y\}]),element(X,[X])]

obtained by anti-unifying original heads

obtained by anti-unifying element(c,[c]) and element(d,[c,d])

obtained by anti-unifying element(c,[c]) and element(d,[d])
Inductive Logic Programming

Generalising clauses:

computing the $\theta$ least-general generalization

\begin{verbatim}
theta_lgg((H1:-B1),(H2:-B2),(H:-B)):-
    anti_unify(H1,H2,H,[],S10,[],S20),
    theta_lgg_bodies(B1,B2,[],B,S10,S1,S20,S2).

define_theta_lgg_bodies([]),B2,B,B,S1,S1,S2,S2).
    theta_lgg_bodies([Lit|B1],B2, B0,B, S10,S1, S20,S2):-
        theta_lgg_literal(Lit,B2, B0,B00, S10,S11, S20,S21),
        theta_lgg_bodies(B1,B2, B00,B, S11,S1, S21,S2).

theta_lgg_literal(Lit1,[], B,B, S1,S1, S2,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
    same_predicate(Lit1,Lit2),
    anti_unify(Lit1,Lit2,Lit,S10,S11,S20,S21),
    theta_lgg_literal(Lit1,B2,[Lit|B0],B, S11, S1,S21,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
    not(same_predicate(Lit1,Lit2)),
    theta_lgg_literal(Lit1,B2,B0,B,S10,S1,S20,S2).
same_predicate(Lit1,Lit2) :-
    functor(Lit1,P,N),
    functor(Lit2,P,N).
\end{verbatim}
Generalising clauses:

computing the $\theta$ least-general generalization

?- theta_lgg((\text{reverse}([2,1],[3],[1,2,3])):-[\text{reverse}([1],[2,3],[1,2,3])]),
(\text{reverse}([a],[], [a]):-[\text{reverse}([], [a],[a])]),
C).

\[ C = \text{reverse}([X|Y], Z, [U|V]) :- [\text{reverse}(Y, [X|Z], [U|V])] \]
Inductive Logic Programming

**Bottom up induction**

specific-to-general search of the hypothesis space

relative least general generalization \( rlgg(e_1, e_2, M) \)
of two positive examples \( e_1 \) and \( e_2 \)
relative to a partial model \( M \) is defined as:
\[
rlgg(e_1, e_2, M) = lgg((e_1 :- \text{Conj}(M)), (e_2 :- \text{Conj}(M)))
\]

conjunction of all positive examples plus ground facts for the background predicates
Inductive Logic Programming

Generalising clauses:

relative least general generalization

M
e1
append([1,2],[3,4],[1,2,3,4]).
append([a],[],[a]).
append([],[],[]).
append([2],[3,4],[2,3,4]).

rlgg(e1,e2,M)

?- theta_lgg((append([1,2],[3,4],[1,2,3,4]) :-
    [append([1,2],[3,4],[1,2,3,4]),
     append([a],[],[a]), append([],[],[]),
     append([2],[3,4],[2,3,4])],
    (append([a],[],[a]) :-
     [append([1,2],[3,4],[1,2,3,4]),
     append([a],[],[a]), append([],[],[]),
     append([2],[3,4],[2,3,4])]),
    C))
Inductive Logic Programming

Generalising clauses:

relative least general generalization

rlgg(e1,e2,M)

```
append([X|Y], Z, [X|U]) :- [
    append([2], [3, 4], [2, 3, 4]),
    append(Y, Z, U),
    append([V], Z, [V|Z]),
    append([K|L], [3, 4], [K, M, N|O]),
    append(L, P, Q),
    append([], [], []),
    append(R, [], R),
    append(S, P, T),
    append([A], P, [A|P]),
    append(B, [], B),
    append([a], [], [a]),
    append([C|L], P, [C|Q]),
    append([D|Y], [3, 4], [D, E, F|G]),
    append(H, Z, I),
    append([X|Y], Z, [X|U]),
    append([1, 2], [3, 4], [1, 2, 3, 4])
]
```

variables in body are proper subset of variables in head

Too Complex!!
Generalising clauses:

relative least general generalization (algorithm)

```prolog
rlgg(E1,E2,M,(H:- B)):-
    anti_unify(E1,E2,H,[],S10,[],S20),
    varsin(H,V),
    rlgg_bodies(M,M,[],B,S10,S1,S20,S2,V).
```

```prolog
rlgg_bodies([],B2,B,B,S1,S1,S2,S2,V).
rlgg_bodies([L|B1],B2,B0,B,S10,S1,S20,S2,V):-
    rlgg_literal(L,B2,B0,B00,S10,S11,S20,S21,V),
    rlgg_bodies(B1,B2,B00,B,S11,S1,S21,S2,V).
```

```prolog
rlgg_bodies(B0,B1,BR0,BR,S10,S1,S20,S2,V): rlgg all literals in B0 with all literals in B1, yielding BR (from accumulator BR0) containing only vars in V
```
Generalising clauses:

relative least general generalization (algorithm)

rlgg_literal(L1,[],B,B,S1,S1,S2,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
    same_predicate(L1,L2),
    anti_unify(L1,L2,L,S10,S11,S20,S21),
    varsin(L,Vars),
    var_proper_subset(Vars,V),
    !,
    rlgg_literal(L1,B2,[L|B0],B,S11,S1,S21,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
    rlgg_literal(L1,B2,B0,B,S10,S1,S20,S2,V).
Generalising clauses:
relative least general generalization (algorithm)

\begin{align*}
\text{var\_remove\_one}(X, [Y|Ys], Ys) & : - \\
& \quad X == Y. \\
\text{var\_remove\_one}(X, [Y|Ys], [Y|Zs]) & : - \\
& \quad \text{var\_remove\_one}(X, Ys, Zs). \\
\end{align*}

\begin{align*}
\text{var\_proper\_subset}([], Ys) & : - \\
& \quad Ys \neq []. \\
\text{var\_proper\_subset}([X|Xs], Ys) & : - \\
& \quad \text{var\_remove\_one}(X, Ys, Zs), \\
& \quad \text{var\_proper\_subset}(Xs, Zs). \\
\end{align*}

\begin{align*}
\text{varsin}(\text{Term}, Vars) & : - \\
& \quad \text{varsin}(\text{Term}, [], V), \\
& \quad \text{sort}(V, Vars). \\
\text{varsin}(V, Vars, [V|Vars]) & : - \\
& \quad \text{var}(V). \\
\text{varsin}(\text{Term}, V0, V) & : - \\
& \quad \text{functor}(\text{Term}, F, N), \\
& \quad \text{varsin\_args}(N, \text{Term}, V0, V). \\
\end{align*}

\begin{align*}
\text{varsin\_args}(0, \text{Term}, Vars, Vars). \\
\text{varsin\_args}(N, \text{Term}, V0, V) & : - \\
& \quad N > 0, \\
& \quad N1 \text{ is } N-1, \\
& \quad \text{arg}(N, \text{Term}, \text{ArgN}), \\
& \quad \text{varsin}(\text{ArgN}, V0, V1), \\
& \quad \text{varsin\_args}(N1, \text{Term}, V1, V). \\
\end{align*}
Generalising clauses:
relative least general generalization (algorithm)

?- rlgg(append([1,2],[3,4],[1,2,3,4]),
       append([a],[[],[a]]),
       append([1,2],[3,4],[1,2,3,4]),
       append([a],[[],[a]]),
       append([],[],[]),
       append([2],[3,4],[2,3,4]),
       (H:- B)).

H = append([X|Y], Z, [X|U])
B = [append([2], [3, 4], [2, 3, 4]),
     append(Y, Z, U),
     append([], [], []),
     append([a], [], [a]),
     append([1, 2], [3, 4], [1, 2, 3, 4])]
Inductive Logic Programming

Bottom up induction

main algorithm

construct rlgg of two positive examples

remove all positive examples that are extensionally covered by the constructed clause

further generalize the clause by removing literals

as long as no negative examples are covered
Inductive Logic Programming

Bottom up induction

main algorithm

```prolog
induce_rlgg(Exs,Clauses):-
    pos_neg(Exs,Poss,Negs),
    bg_model(BG),
    append(Poss,BG,Model),
    induce_rlgg(Poss,Negs,Model,Clauses).

induce_rlgg(Poss,Negs,Model,Clauses):-
    covering(Poss,Negs,Model,[],Clauses).

pos_neg([],[],[]).
pos_neg([+E|Exs],[E|Poss],N)gs):-
    pos_neg(Exs,Poss,Negs).
pos_neg([-E|Exs],Poss,[E|Negs]):-
    pos_neg(Exs,Poss,Negs).
```
# Bottom up induction

**main algorithm (covering)**

```prolog
covering(Poss, Negs, Model, Hyp0, NewHyp) :-
    construct_hypothesis(Poss, Negs, Model, Hyp),
    !,
    remove_pos(Poss, Model, Hyp, NewPoss),
    covering(NewPoss, Negs, Model, [Hyp | Hyp0], NewHyp).
covering(P, N, M, H0, H) :-
    append(H0, P, H).
remove_pos([], M, H, []).
remove_pos([P|Ps], Model, Hyp, NewP) :-
    covers_ex(Hyp, P, Model),
    !,
    write('Covered example: '),
    writeLn(P),
    remove_pos(Ps, Model, Hyp, NewP).
remove_pos([P|Ps], Model, Hyp, [P|NewP]) :-
    remove_pos(Ps, Model, Hyp, NewP).
```

```prolog
covers_ex((Head::- Body),
            Example, Model):-
    verify((Head=Example,
            forall(element(L,Body),
                    element(L,Model))))).
```
Bottom up induction

main algorithm (hypothesis construction)

construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
    write('RLGG of '), write(E1),
    write(' and '), write(E2), write(' is'),
    rlgg(E1,E2,Model,Cl),
    reduce(Cl,Negs,Model,Clause),
    !,
    nl,tab(5), write_ln(Clause).
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
    write_ln(' too general'),
    construct_hypothesis([E2|Es],Negs,Model,Clause).

Inductive Logic Programming
reduce((H:-B₀),Negs,M,(H:-B)):-
    setof0(L,
    (element(L,B₀), not(var_element(L,M))),
    B₁),
    reduce_negs(H,B₁,[],B,Negs,M).

var_element(X,[Y|Ys]):-
    X == Y.
var_element(X,[Y|Ys]):-
    var_element(X,Ys).

setof0(X,G,L):-
    setof(X,G,L),!.
settof0(X,G,[]).
Inductive Logic Programming

Bottom up induction

main algorithm (hypothesis reduction)

\[
\text{reduce_negs}(H, [\text{\texttt{L|Rest}}], \text{\texttt{B0}}, B, \text{\texttt{Negs}}, \text{\texttt{Model}}) : -
\begin{align*}
  & \text{append} \left( \text{\texttt{B0}}, \text{\texttt{Rest}}, \text{\texttt{Body}} \right), \\
  & \text{not(\text{\texttt{covers_neg}}((H:-\text{\texttt{Body}}), \text{\texttt{Negs}}, \text{\texttt{Model}}, N))}, \\
  & !, \\
  & \text{\texttt{reduce_negs}}(H, \text{\texttt{Rest}}, \text{\texttt{B0}}, B, \text{\texttt{Negs}}, \text{\texttt{Model}}).
\end{align*}
\]

\[
\text{reduce_negs}(H, [\text{\texttt{L|Rest}}], \text{\texttt{B0}}, B, \text{\texttt{Negs}}, \text{\texttt{Model}}) : -
\begin{align*}
  & \text{reduce_negs}(H, \text{\texttt{Rest}}, [\text{\texttt{L|B0}}], B, \text{\texttt{Negs}}, \text{\texttt{Model}}).
\end{align*}
\]

\[
\text{reduce_negs}(H, [], \text{\texttt{Body}}, \text{\texttt{Body}}, \text{\texttt{Negs}}, \text{\texttt{Model}}) : -
\begin{align*}
  & \text{not(\text{\texttt{covers_neg}}((H:-\text{\texttt{Body}}), \text{\texttt{Negs}}, \text{\texttt{Model}}, N))}.
\end{align*}
\]

\[
\text{covers_neg}(\text{\texttt{Clause}}, \text{\texttt{Negs}}, \text{\texttt{Model}}, N) : -
\begin{align*}
  & \text{element}(N, \text{\texttt{Negs}}), \\
  & \text{\texttt{covers_ex}}(\text{\texttt{Clause}}, N, \text{\texttt{Model}}).
\end{align*}
\]
Inductive Logic Programming

Bottom up induction

example

RLGG of append([1,2],[3,4],[1,2,3,4]) and append([a],[a],[a]) is
append([X|Y],Z,[XIU]) :- [append(Y,Z,U)]
Covered example: append([1,2],[3,4],[1,2,3,4])
Covered example: append([a],[a],[a])
Covered example: append([2],[3,4],[2,3,4])

RLGG of append([],[],[]) and append([],[1,2,3],[1,2,3]) is
append([],X,X) :- []
Covered example: append([],[],[])
Covered example: append([],[1,2,3],[1,2,3])
Covered example: append([],[3,4],[3,4])

Clauses = [(append([],X,X) :- []),
(append([X|Y],Z,[XIU]) :- [append(Y,Z,U)])]
Bottom up induction example

RLGG of listnum([],[]) and
- listnum([2,three,4],[two,3,four]) is too general
- listnum([4],[four]) is
tolistnum([X|Xs],[Y|Ys]):-[num(X,Y),listnum(Xs,Ys)]
Covered example: listnum([2,three,4],[two,3,four])
Covered example: listnum([4],[four])

RLGG of listnum([3,three,4],[two,3,four]) and
- listnum([2,three,4],[three,4],[two,3,four],[four]) is
tolistnum([V|Vs],[W|Ws]):-[num(W,V),listnum(Vs,Ws)]
Covered example: listnum([3,three,4],[two,3,four])
Covered example: listnum([two],[two])

Clauses =[(listnum([V|Vs],[W|Ws]):-[num(W,V),listnum(Vs,Ws)]),
          (listnum([X|Xs],[Y|Ys]):-[num(X,Y),listnum(Xs,Ys)]),listnum([],[])]
Top down induction

main algorithm

Start with the most general definition

further specialize the clause by adding literals

as long as negative examples are covered
Top down induction

main algorithm
Inductive Logic Programming

Top down induction

main algorithm

\[
\text{induce_spec(Examples,Clauses):=}
\]
\[
\quad \text{process_examples([],[],Examples,Clauses).}
\]

% process the examples
\[
\text{process_examples(Clauses,Done,[] ,Clauses).}
\]
\[
\text{process_examples(Cls1,Done,[Ex|Exs],Clauses):=}
\]
\[
\quad \text{process_example(Cls1,Done,Ex,Cls2),}
\]
\[
\quad \text{process_examples(Cls2,[Ex|Done],Exs,Clauses).}
\]

\[
\text{literal(append(X,Y,Z),[list(X),list(Y),list(Z)]).}
\]
\[
\text{term(list([]),[]).}
\]
\[
\text{term(list([X|Y]),[item(X),list(Y)]).}
\]
Inductive Logic Programming

Top down induction

main algorithm

\[
\begin{align*}
\text{process_example}(\text{Clauses}, \text{Done}, +\text{Example}, \text{Clauses}) : & - \\
& \text{covers}(\text{Clauses}, \text{Example}). \\
\text{process_example}(\text{Cls}, \text{Done}, +\text{Example}, \text{Clauses}) : & - \\
& \text{not covers}(\text{Cls}, \text{Example}), \\
& \text{generalise}(\text{Cls}, \text{Done}, \text{Example}, \text{Clauses}). \\
\text{process_example}(\text{Cls}, \text{Done}, -\text{Example}, \text{Clauses}) : & - \\
& \text{covers}(\text{Cls}, \text{Example}), \\
& \text{specialise}(\text{Cls}, \text{Done}, \text{Example}, \text{Clauses}). \\
\text{process_example}(\text{Clauses}, \text{Done}, -\text{Example}, \text{Clauses}) : & - \\
& \text{not covers}(\text{Clauses}, \text{Example}).
\end{align*}
\]

\[
\begin{align*}
\text{covers}(\text{Clauses}, \text{Example}) : & - \\
& \text{proved}(10, \text{Clauses}, \text{Example}). \\
\text{prove_d}(D, \text{Cls}, \text{true}) : & -!.
\end{align*}
\]

\[
\begin{align*}
\text{prove_d}(D, \text{Cls}, (A, B)) : & -!, \\
& \text{prove_d}(D, \text{Cls}, A), \\
& \text{prove_d}(D, \text{Cls}, B). \\
\text{prove_d}(D, \text{Cls}, A) : & - \\
& D > 0, D1 \text{ is } D-1, \\
& \text{copy_element}((A:-B), \text{Cls}), \\
& \text{prove_d}(D1, \text{Cls}, B). \\
\text{prove_d}(D, \text{Cls}, A) : & - \\
& \text{prove_bg}(A).
\end{align*}
\]

\[
\begin{align*}
\text{prove_bg}(\text{true}) : & -!.
\end{align*}
\]

\[
\begin{align*}
\text{prove_bg}((A, B)) : & -!, \\
& \text{prove_bg}(A), \\
& \text{prove_bg}(B). \\
\text{prove_bg}(A) : & - \\
& \text{bg}((A:-B)), \\
& \text{prove_bg}(B).
\end{align*}
\]
Top down induction

generalise(Cls,Done,Example,Clauses):-
    search_clause(Done,Example,Cl),
    write('Found clause: '),write(Cl),nl,
    process_examples([Cl|Cls],[],+[Example|Done],Clauses).

search_clause(Exs,Example,Clause):-
    literal(Head,Vars),
    try((Head=Example)),
    search_clause(3,a((Head:-true),Vars),Exs,Example,Clause).

search_clause(D,Current,Exs,Example,Clause):-
    write(D),write('..'),
    search_clause_d(D,Current,Exs,Example,Clause),!.

search_clause(D,Current,Exs,Example,Clause):-
    D1 is D+1,
    !,search_clause(D1,Current,Exs,Example,Clause).

search_clause_d(D,a(Clause,Vars),Exs,Example,Clause):-
    covers_ex(Clause,Example,Exs), % goal
    not((element(-N,Exs),covers_ex(Clause,N,Exs))),!.

search_clause_d(D,Current,Exs,Example,Clause):-
    D>0,D1 is D-1,
    specialise_clause(Current,Spec),
    search_clause_d(D1,Spec,Exs,Example,Clause).
specialise(Cls,Done,Example,Claus):-
    false_clause(Cls,Done,Example,C),
    remove_one(C,Cls,Cls1),
    write('...refuted: '),write(C),nl,
    process_examples(Cls1,[],[-Example|Done],Clau).
**Inductive Logic Programming**

**Top down induction**

**Auxiliaries**

```prolog
covers_ex((Head:-Body),Example,Exs):-
    try((Head=Example,covers_ex(Body,Exs))).

covers_ex(true,Exs):-!.
covers_ex((A,B),Exs):-!,
    covers_ex(A,Exs),
    covers_ex(B,Exs).
covers_ex(A,Exs):-
    element(+A,Exs).
covers_ex(A,Exs):-
    prove_bg(A).
```
Top down induction

Auxiliaries

specialise_clause(Current,Spec):-
   add_literal(Current,Spec).
specialise_clause(Current,Spec):-
   apply_subs(Current,Spec).

add_literal(a((H:-true),Vars),a((H:-L),Vars)):-!,
   literal(L,LVars),
   proper_subset(LVars,Vars).
add_literal(a((H:-B),Vars),a((H:-L,B),Vars)):-
   literal(L,LVars),
   proper_subset(LVars,Vars).

apply_subs(a(Clause,Vars),a(Spec,SVars)):-
   copy_term(a(Clause,Vars),a(Spec,Vs)),
   apply_subs1(Vs,SVars).
apply_subs1(Vars,SVars):-
   unify_two(Vars,SVars).
apply_subs1(Vars,SVars):-
   subs_term(Vars,SVars).

unify_two([X|Vars],Vars):-
   element(Y,Vars),
   X=Y.
unify_two([X|Vars],[X|SVars]):-
   unify_two(Vars,SVars).

subs_term(Vars,SVars):-
   remove_one(X,Vars,Vs),
   term(Term,TVars),
   X=Term,
   append(Vs,TVars,SVars).
Top down induction

Example

?-induce_spec([+append([], [b, c], [b, c]),
               -append([], [a, b], [c, d]),
               -append([a, b], [c, d], [c, d]),
               -append([a], [b, c], [d, b, c]),
               -append([a], [b, c], [a, d, e]),
               +append([a], [b, c], [a, b, c]) ],
               Clauses)

3..Found clause:  append(X, Y, Z):-true
    ...refuted:  append([], [a, b], [c, d]):-true
3..Found clause:  append(X, Y, Y):-true
    ...refuted:  append([a, b], [c, d], [c, d]):-true
3..Found clause:  append([], Y, Y):-true
3..4..Found clause:  append([X|Xs], Ys, [X|Zs]):- append(Xs, Ys, Zs)

Clauses = [ (append([X|Xs], Ys, [X|Zs]):-append(Xs, Ys, Zs)), (append([], Y, Y):-true) ]