Relative Least General Generalization

Goal: To find an analogue to *least general generalizations* in a *relative* generality order.

Definition: The *relative least general generalization* of two clauses c_1 and c_2 relative to background knowledge B is defined as the least general clause within the θ-subsumption lattice such that adding this clause to B allows to proof c_1 and c_2 by using the clause only once.

$$B \wedge rlgg_B(c_1, c_2) \vdash c_1 \wedge c_2$$

and

$$(B \wedge c \vdash c_1 \wedge c_2) \text{ implies } c \succeq rlgg_B(c_1, c_2)$$

Construction of $rlgg$s for ground atoms

The most specific clause that can be used to proof a ground atom e_1 from background knowledge B is

$$c_1 = e_1 \text{ :- } b_1, b_2, \ldots$$

where the b_i represent a *ground model* of B, i.e. all ground instances that can be derived from B.

Given two ground atoms e_1 and e_2, the $rlgg_B(e_1, e_2)$ can be constructed as

$$\begin{aligned} c_1 &= e_1 \text{ :- } b_1, b_2, \ldots \\ c_2 &= e_2 \text{ :- } b_1, b_2, \ldots \\ rlgg_B(e_1, e_2) &= lgg(c_1, c_2). \end{aligned}$$

Note: FOIL also uses ground atoms as background knowledge!

$rlgg$ – Example

B :
```
append([1,2],[3,4],[1,2,3,4]).   append([2],[3,4],[2,3,4]).
append([],[],[]).                append([a],[],[a]).
```

e_1 = `append([1,2],[3,4],[1,2,3,4]).`
e_2 = `append([a],[],[a]).`

c_1 =
```
append([1,2],[3,4],[1,2,3,4]) :-
        append([1,2],[3,4],[1,2,3,4]), append([],[],[]),
        append([],[],[]), append([2],[3,4],[2,3,4]).
```

c_2 =
```
append([a],[],[a]) :-
        append([1,2],[3,4],[1,2,3,4]), append([],[],[]),
        append([],[],[]), append([2],[3,4],[2,3,4]).
```

$rlgg_B(e_1, e_2) = lgg(c_1, c_2) =$
```
append([A|B],C,[A|D]) :-
   append([1,2],[3,4],[1,2,3,4]), append([A|B],C,[A|D]),
   append(W,C,X), append([S|B],[3,4],[S,T,U|V]),
   append([R|G],K,[R|L]), append([a],[],[a]),
   append(Q,[],Q), append([P],K,[P|K]), append(N,K,O),
   append(M,[],M), append([],[],[]), append(G,K,L),
   append([F|G],[3,4],[F,H,I|J]), append([E],C,[E|C]),
   append(B,C,D), append([2],[3,4],[2,3,4]).
```

By (Flach, 1994)

Pruning of $rlgg$s

Logical Reduction: A literal l is logically redundant in a clause c with respect to B when $B \wedge c \models c \setminus \{l\}$ and can thus be removed. Problem: inefficiency.

Functional Reduction: Make sure that the input and output variables of the head are linked, i.e. each output variable in the head is either directly computed from the input variables or computed from variables that are linked to input variables (similar to ITOU).

Using Negative Examples: Remove literals from the $rlgg$ (literals with bigger variable depths first) as long as doing so will not cover any negative examples.

Problems with $rlgg$s

Infinite Ground Models of B: Models of B (in particular the *least Herbrand model*) may be infinite (even for definite background knowledge). Thus $rlgg_B(e_1, e_2)$ may be infinite (even if logically reduced).

Size of $rlgg$s: Even when a finite subset of a ground model of B is used, the length of $rlgg_B(e_1, e_2)$ may be very large ($(|B|^2 + 1)$). The size of the $rlgg$ grows exponentially with the number of examples e_i.

Multiple Clause Hypotheses: Target concepts with multiple clauses cannot be learned since the $rlgg$ of a set of examples is a single clause.

GOLEM (Muggleton & Feng, 1990)

To deal with these problems, GOLEM employs the following restrictions to construct $rlgg$s that are not exponential in size:

extensional background knowledge: B consists only of ground atoms (as in FOIL)

h-**easy models:** extensional background knowledge B can be generated from intensional background knowledge B_i by generating all ground atoms derivable from B_i in at most h resolution steps

generative clauses: All variables in the head of a $rlgg$ must also appear in its body

ij-**determinacy:** The body of a $rlgg$ is restricted to contain only determinate literals with at most j input variables and a maximal variable depth of i

GOLEM — Algorithm

1. Randomly select s pairs of positive examples.

2. Construct $rlgg(S)$ for each pair S and select the pair S_{best} that covers the most positive examples.

3. While the number of covered positive examples increases

   (a) Select s uncovered positive examples e_i

   (b) Find e_{best} such that $rlgg(S_{best} \cup e_{best})$ covers the most positive examples

   (c) $S = S_{best} \cup e_{best}$

4. Reduce the clause using functional reduction and negative examples.
   Note: This is the only place where negative examples are used.

Repeat this to learn clauses until all positive examples are covered.

Relationship between $rlgg$ and Inverse Resolution

$$\boxed{rlgg_B(c,d) = lgg(\textsc{Saturate}(B,c), \textsc{Saturate}(B,d))}$$

The the $rlgg$ of two clauses relative to badckground knowledge B is equal to the lgg of two clauses that have been *exhaustively saturated* relative to B (i.e. clause heads of B have been added to the clause by applying a least general saturation operator until no further clause body in B matches a subset of the current clause body).

Subsumption and Implication

It is easy to see that $c \succeq d$ implies $c \rightarrow d$.

The converse, however, does not hold for certain types of self-resolving clauses (not even when no background knowledge is present).

Counter-Examples:

1. $c =$ `ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).`
   $d =$ `ancestor(a,d) :-`
   `ancestor(a,b), ancestor(b,c), ancestor(c,d).`

   θ-subsumption would have to map `Y` to `c`.
   However, $c \rightarrow d$.

2. $c =$ `list([V|W]) :- list(W).`
   $d =$ `list([X,Y,Z|T) :- list(T).`

   θ-subsumption would have to map `W` to `Y,Z|T` and `T`.
   However, $c \rightarrow d$.

Important Results on Subsumption and Implication

Deduction Theorem: $c \models d \Leftrightarrow \models (c \rightarrow d)$

Subsumption Theorem: If $B \models c$ then c is a tautology or there exists a resolution proof of a clause c' such that $c' \succeq c$. This trivially implies refutation completeness ($c = \Box$).

Corollary 1: $c \rightarrow d$ iff d is a tautology, or $c \succeq d$, or a clause e can be constructed by repeatedly self-resolving c such that $e \succeq d$.

Corollary 2: If c is not self-resolving and d is not tautological $c \rightarrow d$ is equivalent to $c \succeq d$.

Gottlob's Lemma: If d is not a tautology, then a necessary condition for $c \rightarrow d$ is that $c^+ \succeq d^+$ and $c^- \succeq d^-$ where c^+ are the positive literals of c and c^- are the negative literals of c.

Undecidability: For general clauses c and d, the question whether $c \rightarrow d$ is undecidable (Schmidt-Schauss, 1988).

Inverse Entailment

Learning a single generalization c from a single example clause e in the presence of background knowledge B is to find c such that

$$B \wedge c \models e$$

By the deduction theorem this can be rewritten as

$$B \wedge \overline{e} \models \overline{c}$$

As c and e are single clauses, $\overline{e}$ and $\overline{c}$ consist only of ground skolemized unit atoms. In particular, if e is a ground atom, $\overline{e}$ is also a ground atom. As in GOLEM, the most specific clause s for which this relation holds, is the clause

$$s = e \leftarrow b_1, b_2, \ldots \text{ with } \overline{s} = \overline{e} \wedge b_1 \wedge b_2 \wedge \ldots$$

where the b_i form a ground model of B. For any c holds:

$$B \wedge \overline{e} \models \overline{s} \models \overline{c} \text{ and thus } c \models s$$

PROGOL (Muggleton, 1995)

1. Select a definite clause e from the background knowledge whose head matches the target concept

2. Construct the most special clause s_i with maximal variable depth i that obeys certain user-specified type and mode restrictions and subsumes a saturated version of the input clause e

3. Find the best generalization of s of s_i using *best-first search*

4. Goto 1. until all positive examples are covered.

PROGOL — Construction of s_i

1. Initialize the head of s_i with a variabilized head of e (equal terms will be bound to equal variables in the entire algorithm)

2. $V =$ all input variables (variables with *mode* +) associated with the corresponding terms in the head of e

3. $k = 0$

4. loop
   - If $k = i$ return s_i else $k = k + 1$
   - For all literals in the background knowledge
     - Find all variabilization with variables of V for the input variables that obeys the type restrictions
     - and find all substitutions that replace all variables with terms that have been previously associated with these variables
     - try to proof each resulting literal with at most h resolution steps
     - add proven literals to s_i replacing old terms with old and new terms with new variables. Add new variables to V.

PROGOL's Search Heuristic

PROGOL searches for the best generalization of s_i using *best-first search*. The clause s_i is processed in order from left to right (starting with the head) and a queue of current candidate clauses and a pointer to the next literal in s_i is maintained.

A clause s is evaluated with an admissible heuristic f_s

$$g_s = p_s - c_s - g_s \text{ and } f_s = g_s - n_s$$

where p_s (n_s) are the positive (negative) examples covered by the clause, c_s is the current length of the clause, and h_s is an (optimistic) estimate of the number of literals needed to make the clause *I/O-complete* (derive a value for all output variables of its head). g_s is an upper bound of f_r for all refinements r of s.

The search stops when it is guaranteed that no better clause with a length $c_s \leq c$ can be found, where c is a parameter that can be specified by the user.

PROGOL (Ctd.)

1. remove candidate s with highest f_s from the queue

2. if $n_s = 0, f_s > 0$ and $f_s >= g_r$ for all r that have been visited so far then return s.

3. unless $n_s = 0$, $g_s \leq 0$ or $c_s \geq c$ refine s

   - add the next literal of s_i to s

   - add the next literal of s_i but introduce new variables for some of its output variables (*variable splitting*)

   - let s as it is, but increment the pointer so that refinements with the same literal are not considered again

4. add all refinements to the queue

5. unless the queue is empty goto 1.

This algorithm for finding the best generalization of s_i is used in a covering strategy as in FOIL.

Characteristics of Progol

- Avoids limitations of generalizing in the θ-subsumption lattice
- Control structure quite similar to Foil, but employs a *best-first* search, instead of *hill-climbing*
- Is not limited to determinate literals
- For longer clauses ($c > 3$) best-first search is costly
- Uses type and mode restrictions, maximal variable depth and maximal clause length to keep the search space tractable
- Background knowledge can be general horn clauses
- Examples can ge definite clauses can be general horn clauses (in that case the skolemized body literals are added to B). Progol can even generalize negative examples (clauses with head `false`).