Inductive Logic Programming: The Problem Specification

- *Given*:
  - *Examples:* first-order atoms or definite clauses, each labeled positive or negative.
  - *Background knowledge:* in the form of a definite clause theory.
  - *Language bias:* constraints on the form of interesting new clauses.

ILP Specification (Continued)

- *Find:*
  - A hypothesis *h* that meets the language constraints and that, when conjoined with *B*, entails (implies) all of the positive examples but none of the negative examples.
- To handle real-world issues such as noise, we often relax the requirements, so that *h* need only entail significantly more positive examples than negative examples.

A Common Approach

- Use a greedy covering algorithm.
  - Repeat while some positive examples remain uncovered (not entailed):
    - Find a *good clause* (one that covers as many positive examples as possible but no/few negatives).
    - Add that clause to the current theory, and remove the positive examples that it covers.
- ILP algorithms use this approach but vary in their method for finding a *good clause*.

A Difficulty

- Problem: It is undecidable in general whether one definite clause implies another, or whether a definite clause together with a logical theory implies a ground atom.
- Approach: Use *subsumption* rather than implication.

Subsumption for Literals

Literal L_1 subsumes L_2 if and only if there exists a substitution θ such that $L_1\theta = L_2$.

Example: $p(f(X), X)$ subsumes $p(f(a), a)$ but not $p(f(a), b)$.

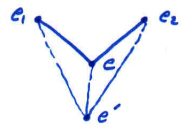
Subsumption for Clauses

Clause C_1 subsumes clause C_2 if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$ (where a clause is viewed as the set of its literals).

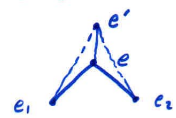
Examples: $p(X, Y) \vee p(Y, Z)$ subsumes $p(W, W)$, using the substitution $\theta = \{X \mapsto W, Y \mapsto W, Z \mapsto W\}$. $p(a, X)$ subsumes $p(a, c) \vee p(Y, b)$ using the substitution $\theta = \{X \mapsto c\}$.

Given a partially-ordered set (poset)...

For any two elements e_1 & e_2, <u>glb</u> of $\{e_1, e_2\}$ is e : $e_1 \geq e$; $e_2 \geq e$; for any e' s.t. $e_1 \geq e'$ & $e_2 \geq e'$, we have $e \geq e'$.



For any two elements e_1 & e_2, lub of $\{e_1, e_2\}$ is e : $e \geq e_1$; $e \geq e_2$; for any e' s.t. $e' \geq e_1$ & $e' \geq e_2$, we have $e' \geq e$.



If any two elements in poset have glb and lub, poset is a <u>lattice</u>.

Least Generalization of Terms

Input : Two terms t_1 and t_2.

Output : least generalization lgg(t_1, t_2).

Let φ be a bijection from (ordered) pairs of terms to variables that do not appear in t_1 or t_2.

If t_1 and t_2 are built from the same primary function symbol f, then $t_1 = f(u_1,\ldots,u_n)$ and $t_2 = f(s_1,\ldots,s_n)$, where the arity of f is $n \geq 0$ and $u_1,\ldots,u_n$ and $s_1,\ldots,s_n$ are terms. Return $f(\text{lgg}(u_1,s_1),\ldots,\text{lgg}(u_n,s_n))$.

Otherwise, return $\varphi(t_1,t_2)$.

Least Generalization of Terms (Continued)

- Examples:
  - lgg(a,a) = a
  - lgg(X,a) = Y
  - lgg($f(a,b),g(a)$) = Z
  - lgg($f(a,g(a)),f(b,g(b))$) = $f(X,g(X))$
- lgg(t_1,t_2,t_3) = lgg(t_1,lgg(t_2,t_3)) = lgg(lgg(t_1,t_2),t_3): justifies finding the lgg of a set of terms using the pairwise algorithm.

Least Generalization of Literals

Input : Literals L_1 and L_2.

Output : Least generalization lgg(L_1, L_2).

If L_1 and L_2 have different predicates or different signs (one is negated and the other unnegated) then return TOP. Otherwise, L_1 has the form $p(u_1,\ldots,u_n)$ and L_2 has the form $p(s_1,\ldots,s_n)$. Return $p($lgg$(u_1,s_1),\ldots,$ lgg$(u_n,s_n)\,)$.

Lattice of Literals

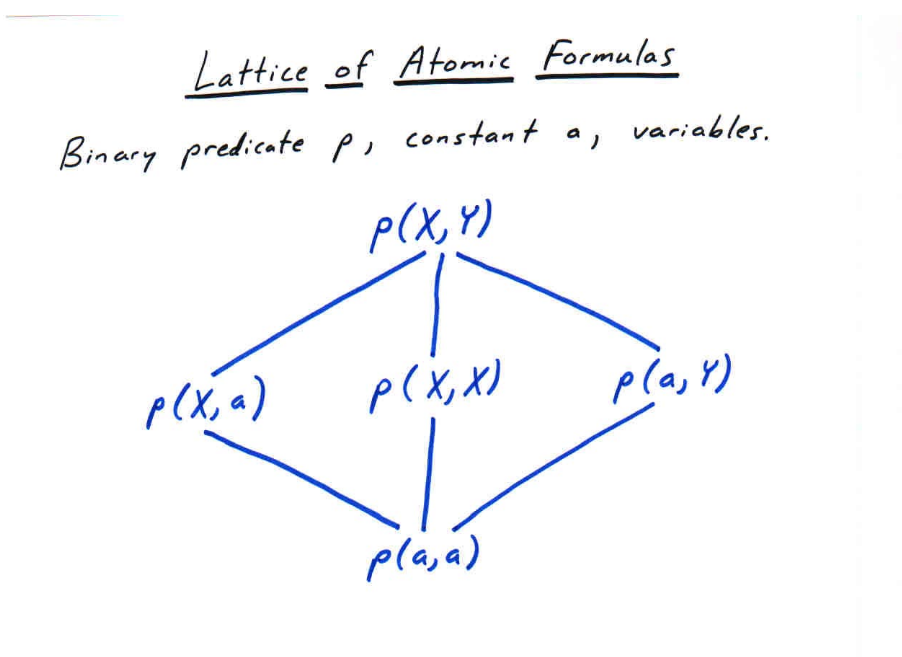
- Consider the following partially ordered set.
- Each member of the set is an equivalence class of literals, equivalent under variance.
- One member of the set is greater than another if and only if one member of the first set subsumes one member of the second (can be shown equivalent to saying: if and only if every member of the first set subsumes every member of the second).

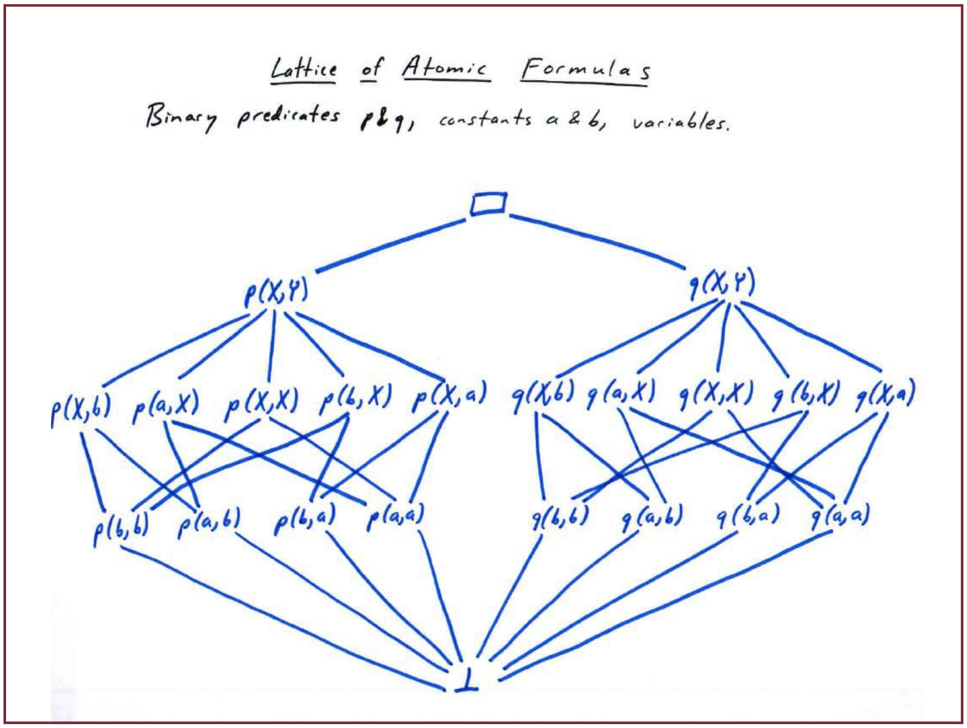
Lattice of Literals (Continued)

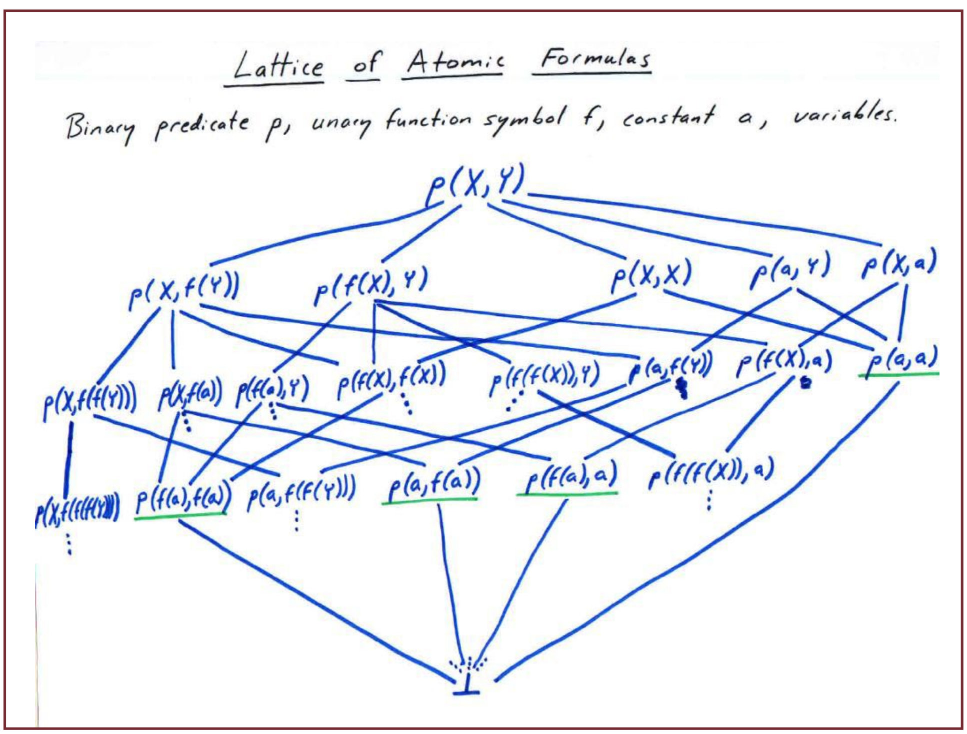
- For simplicity, we now will identify each equivalence class with one (arbitrary) representative literal.
- Add elements TOP and BOTTOM to this set, where TOP is greater than every literal, and every literal is greater than BOTTOM.
- Every pair of literals has a least upper bound, which is their lgg.

Lattice of Literals (Continued)

- Every pair of literals has a greatest lower bound, which is their greatest common instance (the result of applying their most general unifier to either literal, or BOTTOM if no most general unifier exists.)
- Therefore, this partially ordered set satisfies the definition of a lattice.




Lattice of Atomic Formulas
Binary predicates p & q, constants a & b, variables.
p(X,Y)
q(X,Y)
p(X,b) p(a,X) p(X,X) p(b,X) p(X,a)
q(X,b) q(a,X) q(X,X) q(b,X) q(X,a)
p(b,b) p(a,b) p(b,a) p(a,a)
q(b,b) q(a,b) q(b,a) q(a,a)
⊥


Lattice of Atomic Formulas
Binary predicate p, unary function symbol f, constant a, variables.
p(X,Y)
p(X,f(Y)) p(f(X),Y) p(X,X) p(a,Y) p(X,a)
p(X,f(f(Y))) p(X,f(a)) p(f(a),Y) p(f(X),f(X)) p(f(f(X)),Y) p(a,f(Y)) p(f(X),a) p(a,a)
p(f(a),f(a)) p(a,f(f(Y))) p(a,f(a)) p(f(a),a) p(f(f(X)),a)
⊥

Least Generalization of Clauses

Input : Two clauses, $C_1 = l_{1,1} \vee \ldots \vee l_{1,n}$ and $C_2 = l_{2,1} \vee \ldots \vee l_{2,m}$.

Output : Least generalization lgg(C_1, C_2).

Initialize the set of literals in lgg(C_1, C_2) to the empty set. For every pair of literals, one from C_1 and one from C_2, if their lgg is not TOP then add this lgg as a literal of lgg(C_1, C_2). Return the resulting clause.

Example

The lgg of the following two clauses

$p(a, f(a)) \vee p(b,b) \vee \sim p(b, f(b))$ and

$p(f(a), f(a)) \vee p(f(a),b) \vee \sim p(a, f(a))$

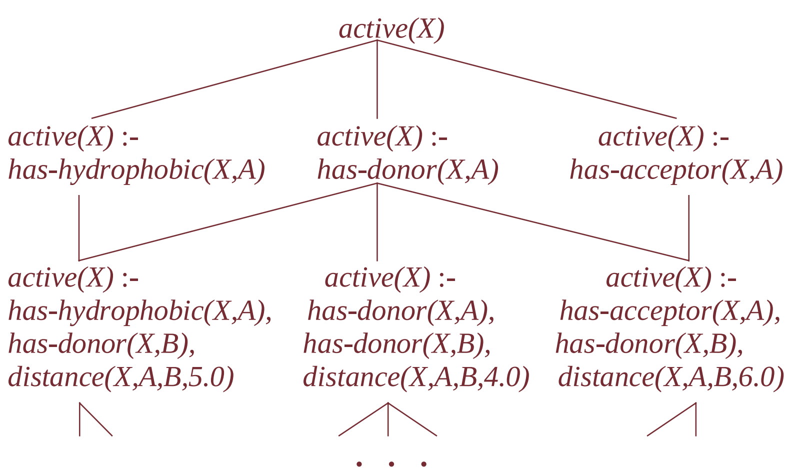
is:

$p(X, f(a)) \vee p(X,Y) \vee p(Z,Z) \vee p(Z,b) \vee \sim p(U, f(U))$

Lattice of Clauses

- We can construct a lattice of clauses in a manner analogous to our construction of literals.
- Again, the ordering is subsumption; again we group clauses into variants; and again we add TOP and BOTTOM elements.
- Again the least upper bound is the lgg, but the greatest lower bound is just the union (clause containing all literals from each).

Lattice of Clauses for the Given Hypothesis Language



Incorporating Background Knowledge: Saturation

- Recall that we wish to find a hypothesis clause *h* that together with the background knowledge *B* will entail the positive examples but not the negative examples.
- Consider an arbitrary positive example *e*. Our hypothesis *h* together with *B* should entail *e*: $B \wedge h \models e$. We can also write this as $h \models B \rightarrow e$.

Saturation (Continued)

- If *e* is an atom (atomic formula), and we only use atoms from *B*, then $B \rightarrow e$ is a definite clause.
- We call $B \rightarrow e$ the *saturation* of *e* with respect to *B*.

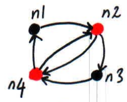
Saturation (Continued)

- Recall that we approximate entailment by subsumption.
- Our hypothesis h must be in that part of the lattice of clauses above (subsuming) $B \rightarrow e$.

Alternative Derivation of Saturation

- From $B \wedge h$ □ e by contraposition: $B \wedge \{\neg e\}$ □ $\neg h$.
- Again by contraposition: h □ $\neg (B \wedge \neg e)$
- So by DeMorgan's Law: h □ $\neg B \vee e$
- If e is an atom (atomic formula), and we only use atoms from B, then $\neg B \vee e$ is a definite clause.

Example of Saturation



g_1

B: node(G,X) :- red(G,X).
node(G,X) :- black(G,X).

path(G,X,Y) :- arc(G,X,Y).
path(G,X,Y) :- arc(G,X,Z),
path(G,Z,Y),
X \= Z, Y \= Z.

black(g1, n1).
red(g1, n2).
black(g1, n3).
red(g1, n4).

arc(g1, n1, n2).
arc(g1, n2, n3).
arc(g1, n3, n4).

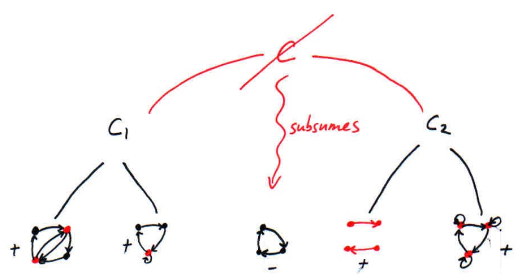
arc(g1, n4, n1).
arc(g1, n2, n4).
arc(g1, n4, n2).

Saturate cyclic(g1) with all variable modes:

cyclic(G) :- black(G,N1), red(G,N2), black(G,N3), red(G,N4),
node(G,N1), node(G,N2), node(G,N3), node(G,N4),
arc(G,N1,N2), arc(G,N2,N3), arc(G,N3,N4),
arc(G,N4,N1), arc(G,N2,N4), arc(G,N4,N2),
path(G,N1,N4), path(G,N4,N1), path(G,N1,N3),
path(G,N3,N1), path(G,N1,N2), path(G,N2,N1),
path(G,N2,N4), path(G,N4,N2), path(G,N2,N3),
path(G,N3,N2), path(G,N3,N4), path(G,N4,N3).

Bottom-up (LGG) Approach (GOLEM)

Start with saturations of all positive examples.
Repeat while possible:
Choose 2 clauses & replace with LGG if the LGG covers (subsumes) no negative examples.

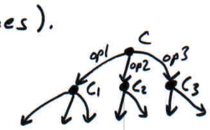


$C_1 \wedge C_2$

- Poor pairings early can throw off results.
- LGG often gives large (incomprehensible) clauses.

Refinement Operators (Top-Down) (MIS - E.Y. Shapiro)

Rewrite a clause to a more specific one (one it subsumes).



Would like the operators to be:

- Finite: rewrite a clause to finite set of clauses.
- Complete: can generate any subsumed clause by repeated application.
- Non-redundant: only one sequence of operators to get from a clause to a subsumed clause.

Nienhuys-Cheng & van der Laag: Can't have all 3!

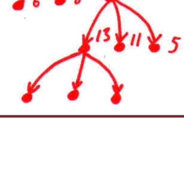
In practice, we give up on the third property.

Top-Down (Refinement Graph) Search

Start with $p(X_1, \dots, X_n)$ where p is target predicate (predicate from which examples are constructed).

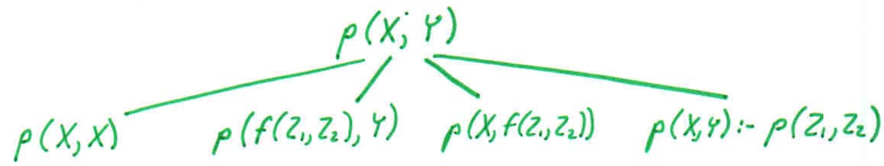
Repeatedly apply refinement operators to a clause to generate children. Score each generated clause by testing which examples it covers (entails when taken with background theory $\mathcal{B}$).

Can do greedy (→ fast, incomplete) search, branch-and-bound (→ slower, complete (admissible)) search, etc.

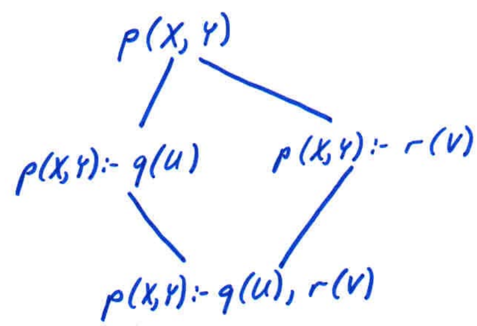


Common Refinement Operators For Definite Clauses

- Substitute for one variable a term built from a functor of arity n and n distinct variables not in the clause: $\theta = \{X \Rightarrow f(Y_1, \ldots, Y_n)\}$
- Substitute for one variable another variable already in the clause (create a "co-reference"): $\theta = \{X \Rightarrow Y\}$
- Add a literal to the body of the clause whose arguments are distinct variables not in the clause.

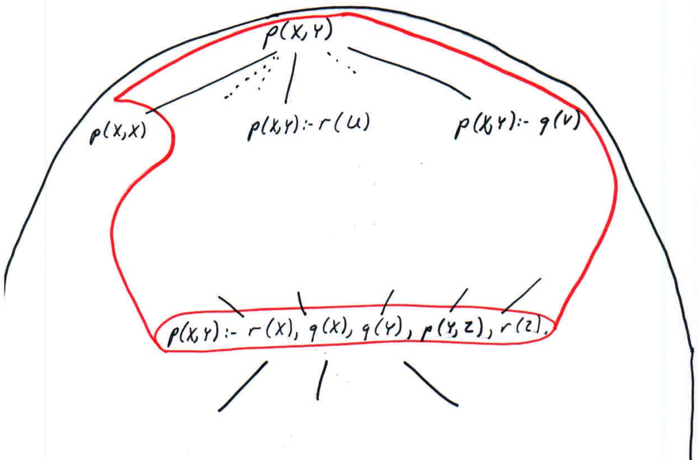


This set of operators is finite and complete, but redundant.



Will end up generating and scoring $p(X, Y) \text{ :- } q(U), r(V)$ twice if doing a complete search.

Refinement graph is the entire lattice of clauses. Can make this smaller by focusing on a "seed" example, saturating it, and only searching the lattice above that saturation.



Overview of Some ILP Algorithms

- GOLEM (bottom-up): saturates every positive example and then repeatedly takes lggs as long as the result does not cover a negative example.
- PROGOL, ALEPH (top-down): saturates first uncovered positive example, and then performs top-down admissible search of the lattice above this saturated example.

Algorithms (Continued)

- FOIL (top-down): performs greedy top-down search of the lattice of clauses (does not use saturation).
- LINUS/DINUS: strictly limit the representation language, convert the task to propositional logic, and use a propositional (single-table) learning algorithm.