LEARNING IN LOGIC

INDUCTIVE LOGIC PROGRAMMING

Ivan Bratko

Faculty of Computer and Info. Sc.

University of Ljubljana, Slovenia

This presentation is shaped after: I. Bratko, Prolog Programming for Artificial Intelligence, 4th edition, Addison-Wesley 2012 (Chapter 21, on ILP; or 3rd edition, 2001, Chapter 19) All programs from the book are available at www.booksites.com

Inductive Logic Programming, ILP

- ILP is an approach to machine learning
- In ILP, hypothesis language = logic
- Usually: first order predicate logic, like Prolog
- In ILP, a hypothesis is a logic program, usually a Prolog program
- ILP is also a form of relational learning – learning relations (as opposed to learning functions)

ILP ENABLES NATURAL USE OF BACKGROUND KNOWLEDGE

- Learner can use knowledge known prior to learning – background knowledge
- E.g. learner may use Pitagora's theorem, or Newton's laws, or commonsense knowledge
- Most machine learning approaches only enable very limited ways of using background knowledge (parameter settings, definition of new attributes, ...)
- ILP supports it in a most general and natural way

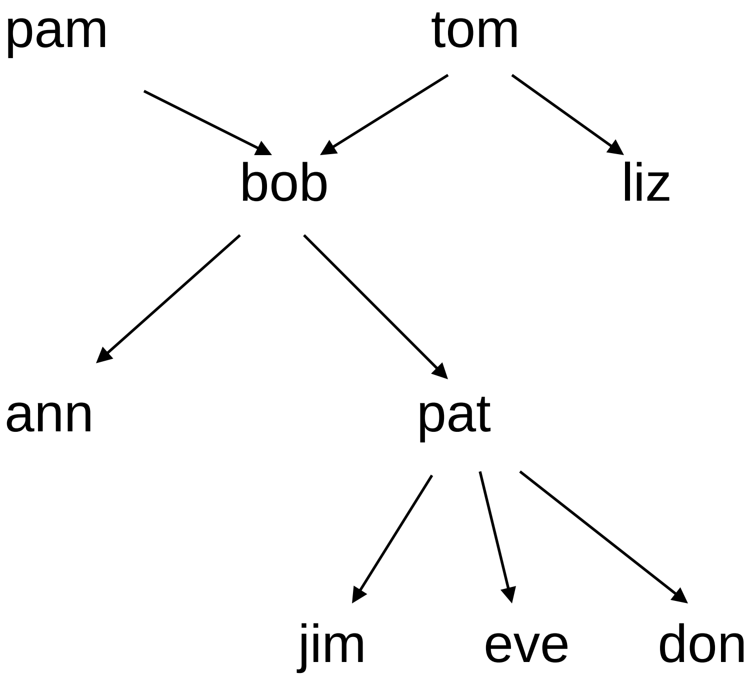
WHY LOGIC AS HYPOTHESIS LANGUAGE?

- In ML, attribute-value representations are more usual (decision trees, rules, SVMs, neural networks...)

- Why predicate logic?
  - More expressive than attribute-value representations
  - Enables flexible use of *background knowledge* (knowledge known to learner prior to learning)

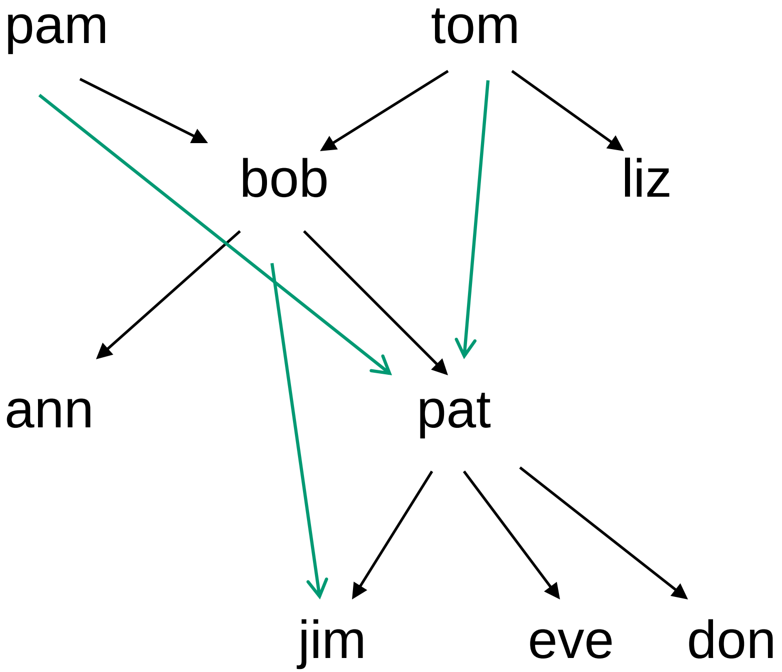
Example: Tree structure as background knowledge

Parent relation given:



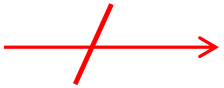
How can definition of relation grandparent be learned?

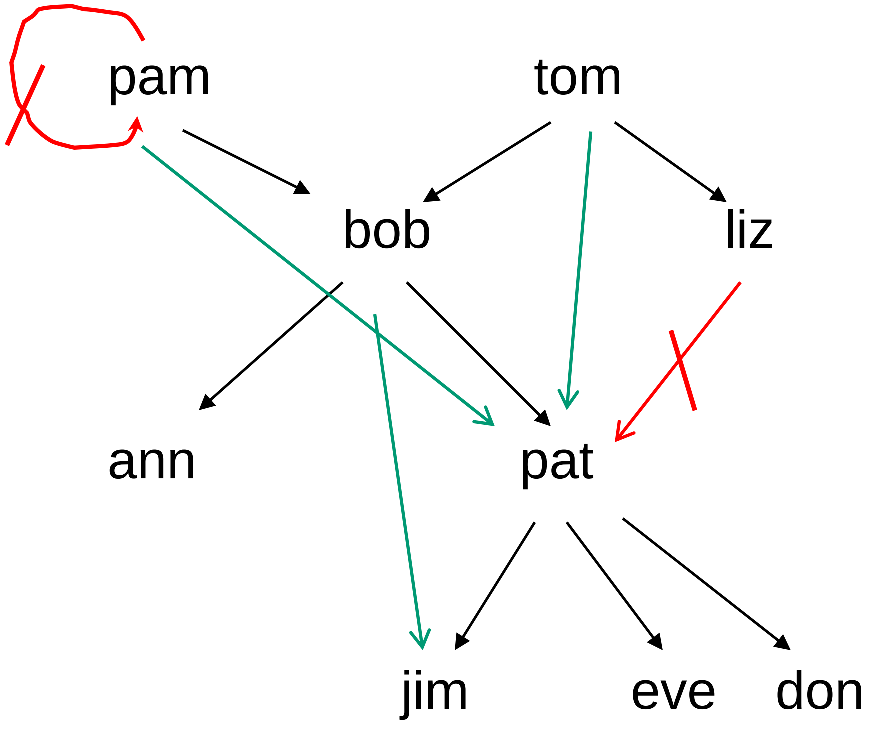
Examples of grandparent relation, green arrows: 



How can definition of relation grandparent be learned?

Negative examples of grandparent:





Definition of grandparent to be learned

For all X and Z: X is grandparent of Z if there is Y such that X is parent of Y and Y is parent of Z

$$
\begin{tikzcd}
X \arrow[d, "\text{parent}"] \\
Y \arrow[d, "\text{parent}"] \\
Z
\end{tikzcd}
$$

In logic, syntax of Prolog language, this is written as:

```
grandparent( X, Z)   :-   parent(X,Y), parent(Y,Z).
```

Problem definition for an ILP system

```
% Background knowledge

parent( pam, bob).
parent( tom, bob).
...

female( pam).
male( tom).
...
```

Problem definition for an ILP system, ctd.

% Positive examples

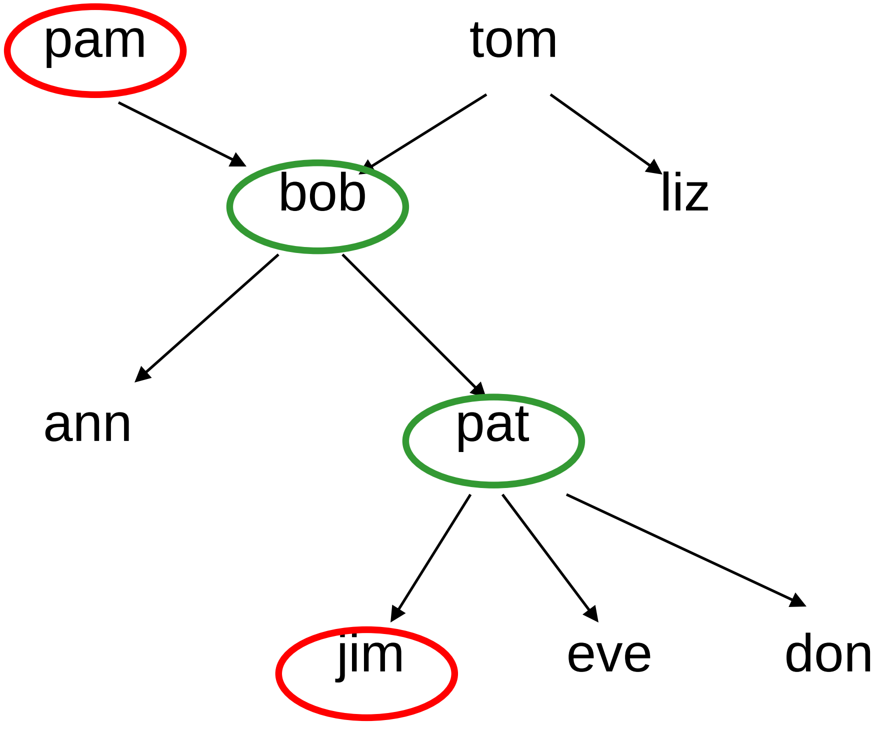
```
ex( grandparent( tom, pat)).      % Tom is grandparent of Pat
ex( grandparent( pam, pat)).
ex( grandparent( bob, jim)).
```

% Negative examples

```
nex( grandparent( pam, pam)).     % Pam is not grandparent of herself
nex( grandparent( liz, pat)).
```

% Target predicate grandparent(X,Y)

WHAT CONCEPT CORRESPONDS TO THESE EXAMPLES?



Green circles: positiveexamples; red circles: negative examples

Learning definition of “has a daughter”

```
% Background knowledge
parent( pam, bob).
parent( tom, liz).
...
female( liz).
male( tom).
...
% Positive examples
ex( has_a_daughter(tom)).      % Tom has a daughter
ex( has_a_daughter(pam)).
...
% Negative examples
nex( has_a_daughter(pam)).     % Pam doesn't have a daughter
...
```

Learning has_a_daughter(X)

Target relation, to be learned from the examples and background knowledge:

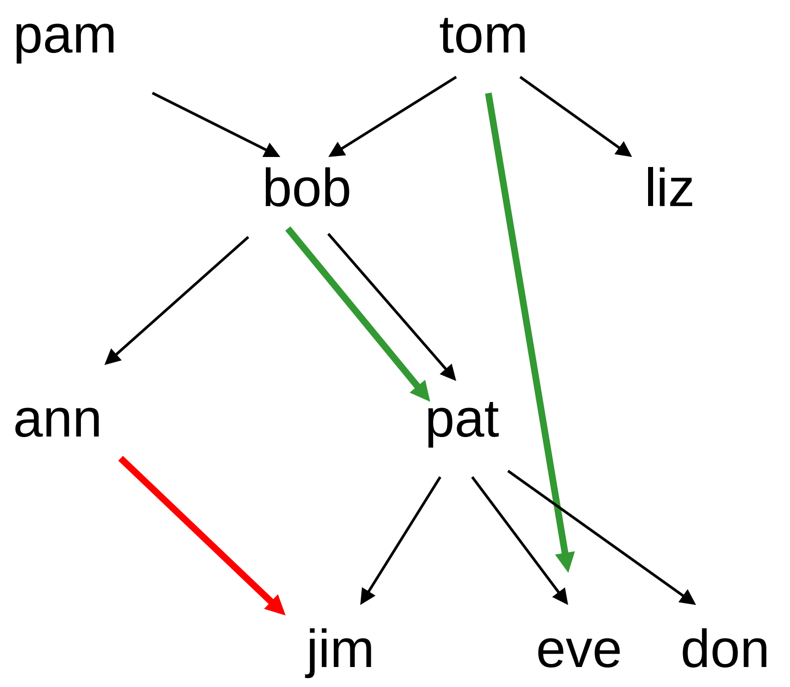
```
has_a_daughter( X)  :-
    parent( X, Y),              % Note: new variable Y!
    female( Y).
```

$$\begin{array}{ll} & X \quad \text{has\_a\_daughter} \\ \text{parent} & \downarrow \\ \text{female} & Y \end{array}$$

Learning predecessor relation

Given parent relation (black arrows), learn relation predecessor pred(X,Y).

Positive examples – green arrows; negative examples – red arrows

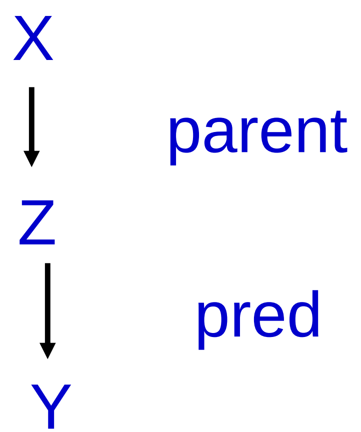


Learning predecessor relation, ctd.

- Predecessor relation pred(X, Y) is defined recursively

```
pred( X, Y)  :-
   parent( X, Y).

pred( X, Y)  :-
   parent( X, Z),
   pred( Z, Y).
```



ILP problem, formally

- Given:

  positive examples E, negative examples N,

  and background knowledge B

- Find:

  hypothesis H, such   that

  B & H |-- E

  and

  For all n in N: not (B & H |--   n)

ILP as automatic programming

- Let:

    p(a) be a positive example

    p(b) be a negative example

- Let B be a given program
- Interaction with B in Prolog:

    ?- p(a).

    no

    ?- p(b).

    no

ILP as automatic programming, ctd.

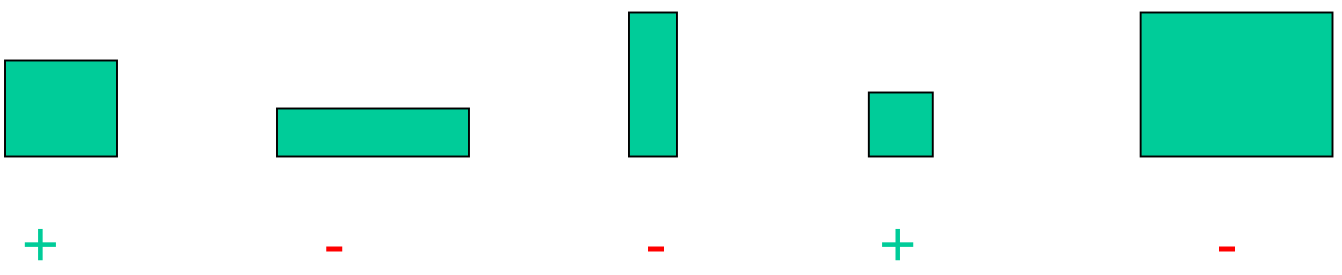
- Task of ILP: Extend B by program H (find H), so that interaction with B+H will be:

```
?-  p(a).          % Positive example
yes
?-  p(b).          % Negative example
no
```

Learning about square

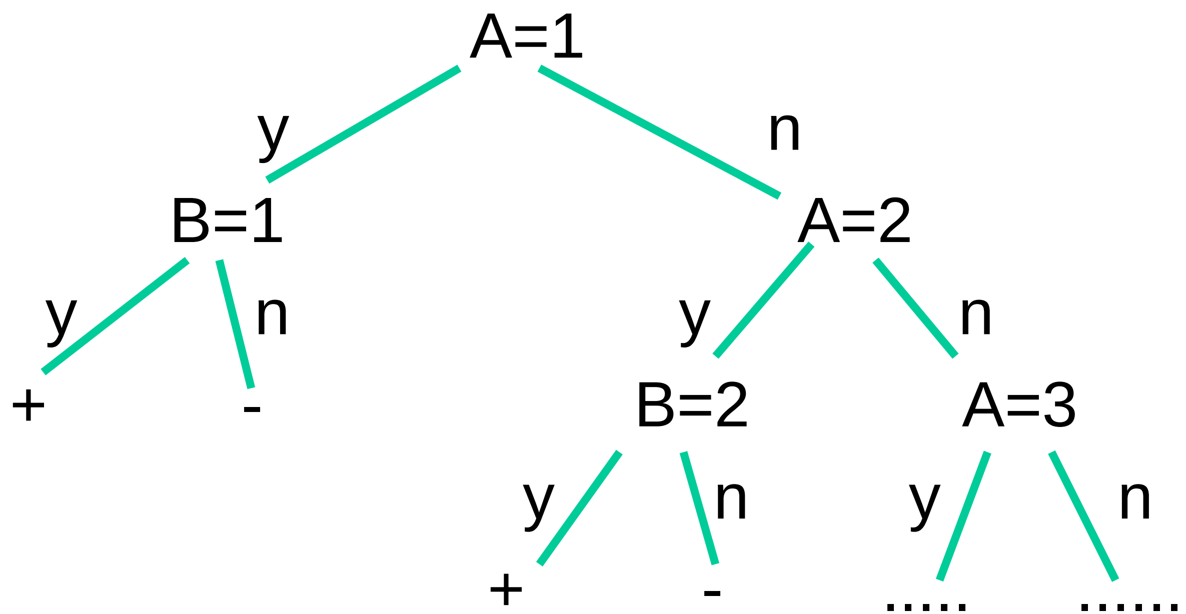
- Attributes: sides A, B of a rectangle
- Target concept: square

- Examples and counter-examples:



Learning about square

- Very awkward to represent in attribute-value learning, e.g. by decision tree



Learning about square

- Very easy to represent target concept in logic:

  For all A: square(A,A)

- Written in Prolog:

```
square( A, A).
```

- Or:

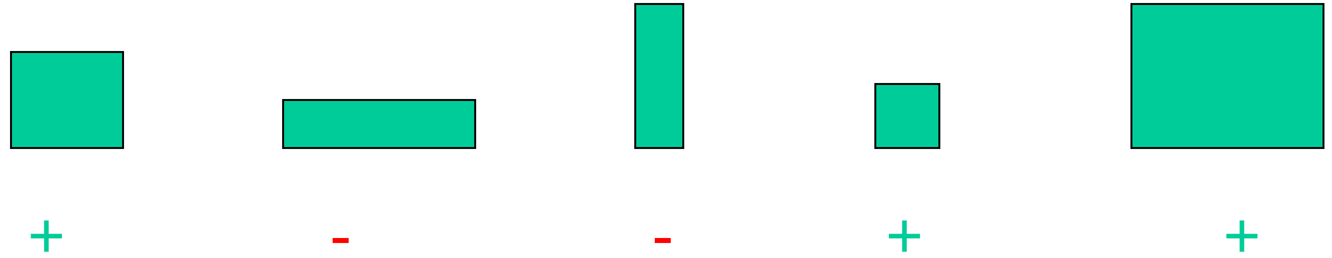
```
square( A, B)  :-  A = B.
```

- This is read as: for all A,B: square(A,B) if A=B

Learning about almost-square

- Attributes: sides A, B of a rectangle
- Target concept: almost a square

- Examples and counter-examples:



Learning about almost-square

- Introduce some “background knowledge”:

```
next( 1, 2).
next( 2, 3).
next( 3, 4).
....
```

- Then target concept is easy to express again:

```
almost_square( A, A).
almost_square( A, B)  :-
   next( A, B).
```

Significance of ILP formulation

- Allows flexible use of background knowledge, that is knowledge known to learner prior to learning

- Background knowledge can be many things:
  - useful auxiliary concepts
  - relations between observed objects
  - properties of objects
  - inter-atom structure in molecules
  - full calculus of qualitative reasoning
  - robot’s “innate” knowledge
  - principles of food networks in ecology
  - ...

EXAMPLE: ECOLOGICAL DISCOVERY

- Problem: discovering food chains (who eats what) from experimental data (Tamadoni-Nezhad et al. 2020)

- Measurement in a large scale field experiments in UK (Farm Scale Evaluations) to study effects of Genetically modified herbicide-tolerant crops (GMHT)

- Observations are in form of facts:

  abundance(Species, Site, Direction)

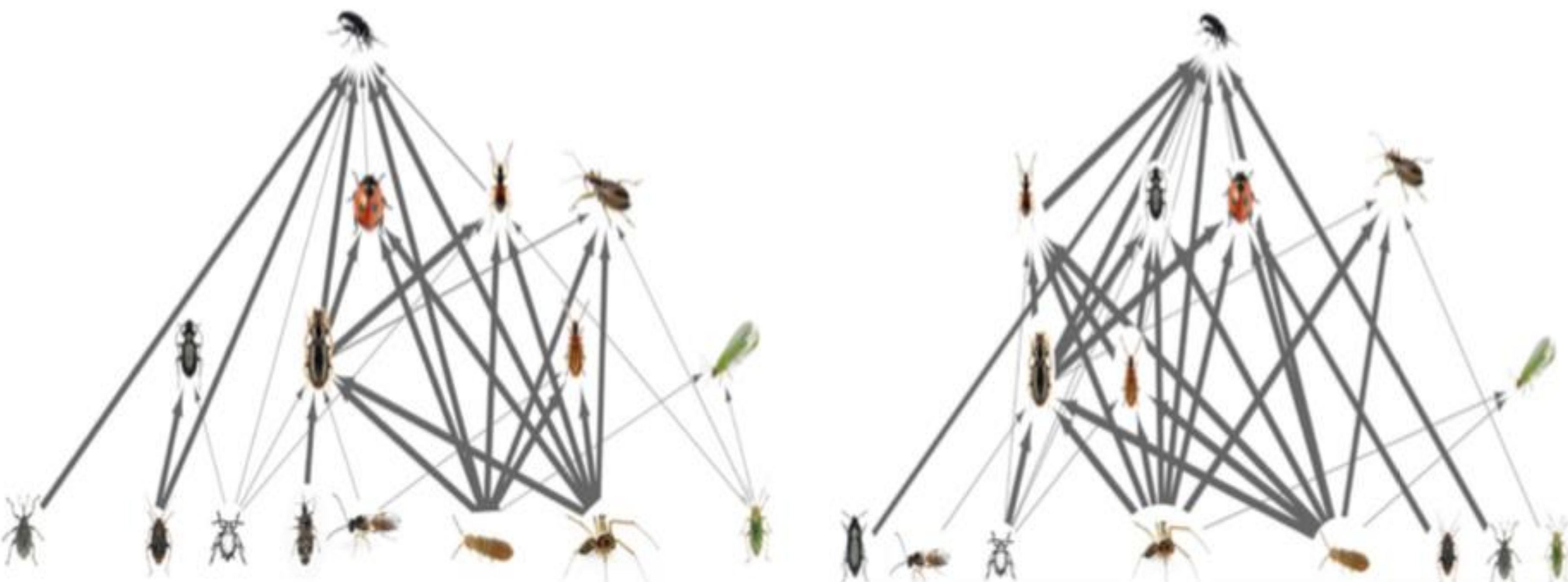
  Abundance of Species at Site has changed in Direction "increase" or "decrease"

BACKGROUND KNOWLEDGE

```
abundance( X, S, Dir):-      % Direction of change of X
    predator(X),             % Species X is predator
    co_occurs(S, X, Y),      % Species X, Y co-occur at S
    bigger_than(X, Y),       % X is bigger than Y
    eats(X, Y),              % X eats Y
    abundance(Y, S, Dir).    % Direction of change of Y
```

To be learned: relation eats(X,Y)   (X eats Y)

EXAMPLES OF LEARNED FOOD CHAINS AMONG INSECTS



(a) (b)

Top-down induction of logic programs

- Employs *refinement* operators
- Typical refinement operators on a clause:
  - Apply a substitution to clause
  - Add a literal to the body of clause
- Refinement graph:
  - Nodes correspond to clauses
  - Arcs correspond to refinements

Part of refinement graph

```
has_a_daughter(X).
```

```
has_a_daughter(X) :-
  male(Y).
```

```
has_a_daughter(X) :-
  male(X).
```

```
...    ...
```

```
has_a_daughter(X) :-
  female(Y).
```

```
has_a_daughter(X) :-
  female(Y),
  parent(S,T).
```

```
...    ...    ...
```

```
has_a_daughter(X) :-
  parent(Y,Z).
```

```
has_a_daughter(X) :-
  parent(X,Z).
```

```
has_a_daughter(X) :-
  parent(X,Z),
  female(U).
```

```
has_a_daughter(X) :-
  parent(X,Z),
  female(Z).
```

EXERCISE

- Construct part of refinement graph for learning about grandparent, given parent relation

- How many refinements are needed in the best case? Start with the most general clause:

gp(X,Y)

EXERCISE

- Construct part of refinement graph for learning about grandparent, given parent relation

- How many refinements are needed in the best case?
- Refinement graph:

gp(X,Y)

X=Y

gp(X,X)

gp(X,Y) :- parent(U,V).

gp(X,Y) :- parent(X,V).

gp(X,Y) :- parent(X,V), parent(S,T).

V = S

Y = T

gp(X,Y) :- parent(X,V), parent(V,Y).

COVERING ALGORITHM

- Input: Set of classified EXAMPLES
- Output: List of induced rules RULELIST

RULELIST := empty
while (EXAMPLES not empty) **do**
  begin
    RULE := InduceOneRule(EXAMPLES)
    Add RULE to RULELIST
    Remove from EXAMPLES examples covered by RULE
  end

Covering approach to ILP

This is the most common structure of ILP systems:

- Induce iteratively clause by clause until all (or most) positive examples are covered

- In each iteration construct a clause by clause refinement process

- Heuristic: among alternative refinements, prefer those clauses that cover many positive and few negative examples

- Note: Covering is a greedy search

  Clauses optimised locally, not hypotheses globally

  Limited to covering examples by individual clauses, or clauses so far
- Consider mutual recursion: even(...) :- odd(...). odd(...) :- even(...).

  Problem: In which order should clauses be induced?

ILP based on covering

- Most ILP systems use covering approach
- Some important ILP systems:
  - FOIL (First Order Inductive Logic), Quinlan
  - PROGOL, Muggleton
  - ALEPH, Srinivasan
  - They all use covering approach

- Problem with covering approach: When inducing recursive definitions, complete chain of recursive calls must be given among examples (e.g. learning “ancestor”)

Typical problems of covering algorithms

- Local optimisation of individual clauses
- Unnecessarily long hypotheses
- Difficulties in handling recursion
- Difficulties in learning multiple predicates simultaneously
- Non-covering (refining complete hypothesis) better in these respects

Non-covering approach

- Develop complete hypotheses (i.e. set of clauses) as a whole
- Search step: refine complete hypothesis
- Combinatorial complexity:
  - Search for individual clauses exponential
  - Search for complete hypothesis expected to be even much worse!
  - Experiments with HYPER program show that in practice complexity is better than expected

Learning relation member(X, L)

- Target definition:

```
member( X, [X | L]).

member( X, [Y | L]):-
    member( X, L).
```

Refining complete hypotheses

Learning member(X, List)

```
member(X1,L1).        Start hypothesis
member(X2,L2).

    Refine term L1 = [X3|L3]

member(X1,[X3|L3]).
member(X2,L2).

    Match X1 = X3

member(X1,[X1|L3]).
member(X2,L2).
```

Refining complete hypotheses, ctd.

```
member(X1,[X1|L3]).
member(X2,L2).
```

Refine term L2 = [X4|L4]

```
member(X1,[X1|L3]).
member(X2,[X4|L4]).
```

Add literal member(X5,L5) and match input L5 = L4

```
member(X1,[X1|L3]).
member(X2,[X4|L4])   :-   member(X5,L4).
```

Match X2 = X5

```
member(X1,[X1|L3]).
member(X2,[X4|L4])   :-   member(X2,L4).
```

Program HYPER

- HYPER = Hypothesis Refiner
- HYPER uses ***non-covering*** approach
- HYPER finds a ***complete*** and ***correct*** hypothesis
- An implementation in Prolog of hypothesis search by refining *complete* hypotheses

- See I. Bratko, Prolog Programming for Artificial Intelligence, 4th edition, Addison-Wesley 2012 (Chapter 21, on ILP)

- All programs from the book available at

  www.booksites.com

Each refinement is a *specialisation* of an existing hypothesis

If refine(H1, H2) then
coverage(H2) is a subset of coverage(H1)

Therefore all refinements that do not cover all positive examples are discarded

Heuristic

Evaluation function on hypotheses F = cost(H):

(1) F increases with # covered neg. examples by H

(2) F increases with size of H

Representation of background knowledge in HYPER

- Background literals, with typed arguments
- Background literals with input and output arguments
- Term refinement rules
- “Prolog predicates”, evaluated by Prolog’s interpreter
- Target predicates evaluated by a special interpreter:
  max. “proof effort”, answers “yes”,”no”, “maybe”

Learning odd/1 and even/1 with HYPER
Definition of the learning problem

```
backliteral( even( L), [ L:list], []).
backliteral( odd( L), [ L:list], []).

term( list, [X|L], [ X:item, L:list]).
term( list, [], []).

start_clause([ odd( L) ] / [ L:list]).
start_clause([ even( L) ] / [ L:list]).

ex( even( [])).
ex( odd( [a])).
ex( odd( [a,b,c,d,e])).
ex( even( [a,b])).
ex( odd( [b,c,d])).
ex( even( [a,b,c,d])).

nex( even( [a])).
nex( odd( [])).
nex( odd( [a,b,c,d])).
nex( even( [a,b,c])).
nex( odd( [a,b])).
```

Learning even and odd, results with HYPER

```
 ?- induce( HYP), show_hyp( HYP).

Hypotheses generated: 85
Hypotheses refined:   16
To be refined:        29

Induced hypothesis:

even([]).

even([A,B|C]):-
  even(C).

odd([A|B]):-
  even(B).
```

Learning even and odd, results with HYPER ctd.

Next Prolog answer:

```
Hypotheses generated: 115
Hypotheses refined:   26
To be refined:        32

even([]).
odd([A|B]):-
  even(B).
even([A|B]):-
  odd(B).
```

Mutually recursive definition!

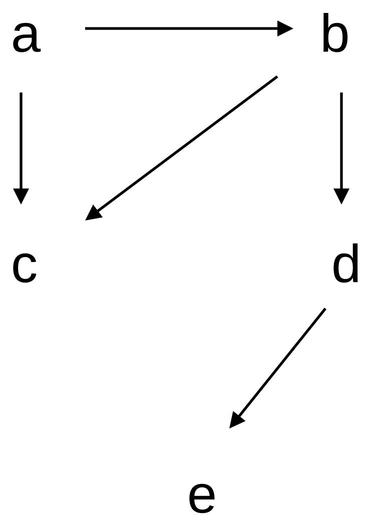
Some other exercises with HYPER

Learning about:

- conc(List1, List2, List3)
- insert_sort(List, SortedList)
- sort(List1, List2)
- path(Node1, Node2, Path)
- arch(Block1, Block2, Block3)

Learning path(A,B,L)

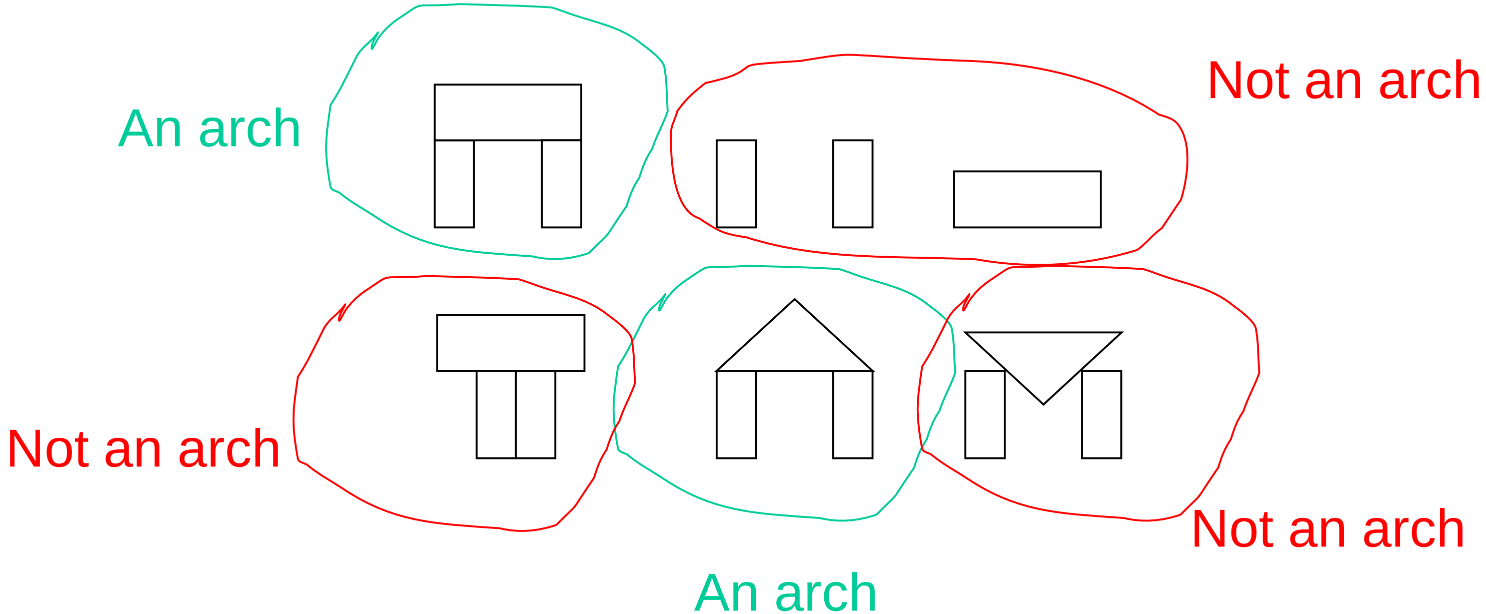
- Given BK as a graph:



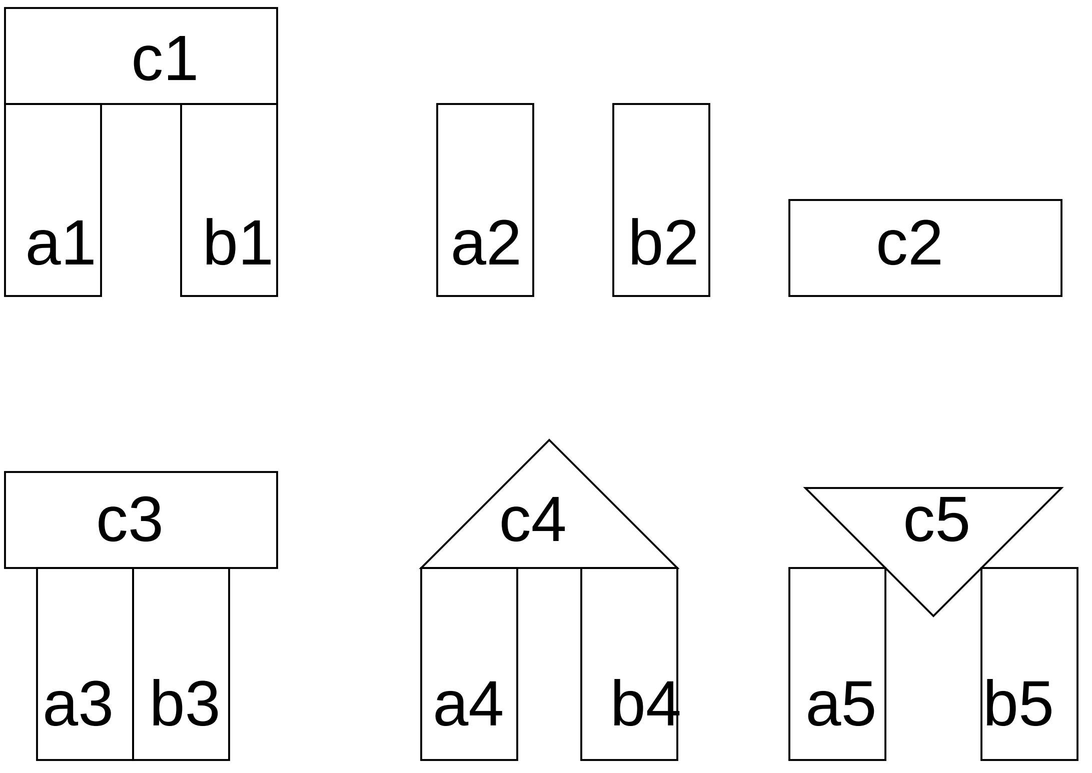
To learn a graph, learn relation link(X,Y): Abductive Logic Programming

HOW COULD A ROBOT LEARN THE CONCEPT OF AN ARCH WITH HYPER?

Learning about arch: examples



Label objects



State relations between objects as part of Background Knowledge

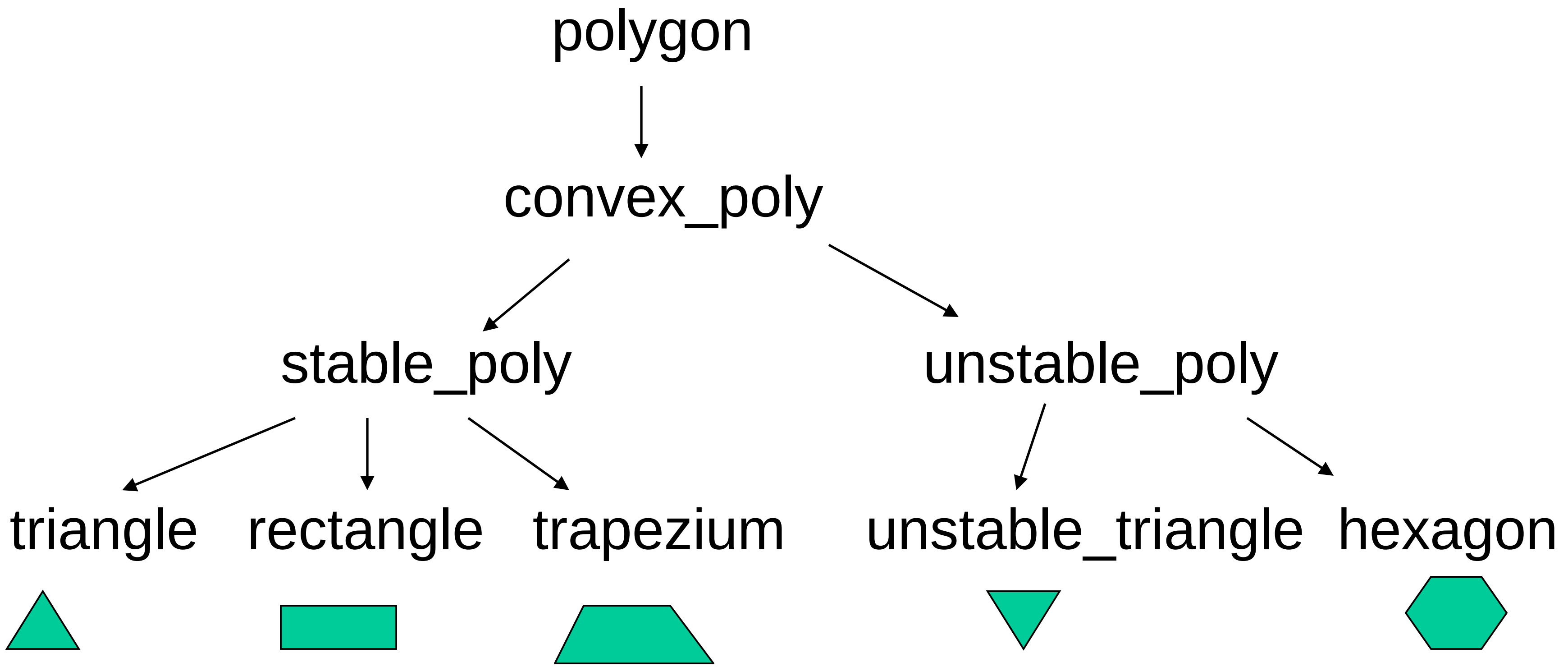
```
support(a1,c1).    support(b1,c1).
support(a3,c3).    support(b3,c3).
support(a4,c4).    support(b4,c4).
support(a5,c5).    support(b5,c5).

touch(a3,b3).
```

State positive and negative examples

```
ex( arch(a1,b1,c1)).
ex( arch(a4,b4,c4)).
nex( arch(a2,b2,c2)).
nex( arch(a3,b3,c3)).
nex( arch(a5,b5,c5)).
nex( arch(a1,b2,c1)).
nex( arch(a2,b1,c1)).
```

Introduce a taxonomy, part of BK



Taxonomy

```
ako( polygon, convex_poly).          % Convex polygon is a kind of polygon
ako( convex_poly, stable_poly).      % Stable polygon is a kind of convex polygon
ako( convex_poly, unstable_poly).    % Unstable polygon is a kind of convex poly
ako( stable_poly, triangle).         % Triangle is a kind of stable polygon
...

ako( rectangle, a1).                 % a1 is a rectangle
ako( rectangle, a2).
ako( triangle, c4).
....
```

Transitivity of "is a"

```
isa( Figure1, Figure2)  :-          % Figure1 is a Figure2
  ako( Figure2, Figure1).

isa( Fig0, Fig) :-
  ako( Fig1, Fig0),
  isa( Fig1, Fig).
```

Background literals

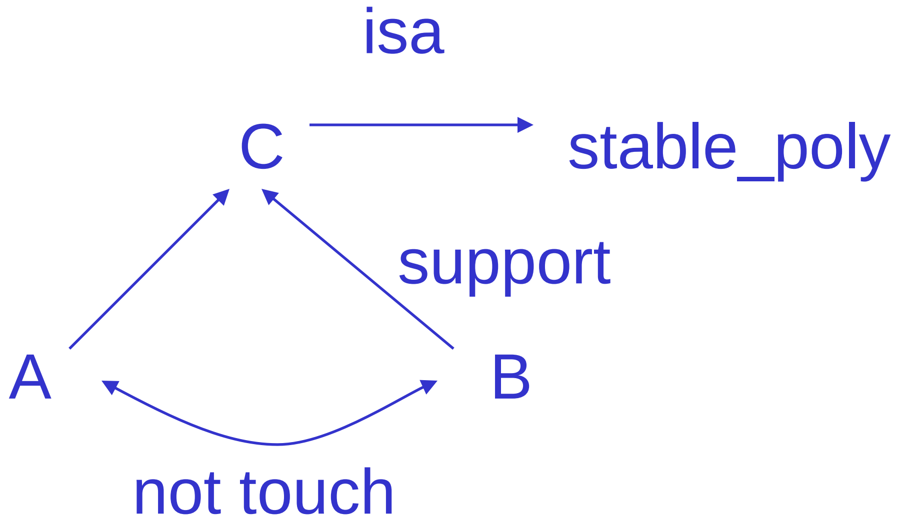
```
backliteral( isa(X,Y), [X:object], [ ])  :-
  member( Y , [polygon,convex_poly,stable_poly,unstable_poly,
    triangle, rectangle, trapezium, unstable_triangle, hexagon]).

backliteral( support(X,Y), [X:object, Y:object], [ ]).
backliteral( touch(X,Y), [X:object, Y:object], [ ]).
backliteral( not G, [X:object,Y:object], [ ])  :-
  G = touch(X,Y); G = support(X,Y).

prolog_predicate( isa(X,Y)).
prolog_predicate( support(X,Y)).
prolog_predicate( touch(X,Y)).
prolog_predicate( not G).
```

Induced hypothesis about arch

```
arch(A,B,C):-
    support(B,C),
    isa(C,stable_poly),
    not touch(A,B),
    support(A,C).
```



Learning about arch (formulation in book)

- Hypotheses generated: 368
- Hypotheses refined: 10
- Time needed in the order of 50 msec

PROGRAM CORRECTNESS PROOFS

- For loops we need invariants, i.e. conditions that are always true
- How can invariants be obtained?
- Common answer: By guessing
- But there is another, less known and more elegant possibility: automatically with ILP
- How do we know that invariant was correctly guessed?
- When program proof succeeds

Example: Integer division a divided by b

```
% result := a div b, remainder := a mod b

     begin,
       res := 0,  rem := a,
       invariant ( res*b + rem = a )  and ( 0 =< rem )
       while (rem >= b) do
             begin, res := res + 1, rem := rem - b, end,
      end
```

Correctness proof includes a proof that invariant is always true.

Trace of dividing 8 by 3

a	b	res	rem := rem - b	
8	3	0	8	
		1	5	
		2	2	2 < 3, rem < b, stop

invar(A, B, Res, Rem)

ex(invar(8, 3, 0, 8))

ex(invar(8, 3, 1, 5))

....

How can negative examples be obtained? By spoiling positive examples

LEARNING INVARIANT FOR INTEGER DIVISION WITH ILP

- ILP problem formulation: Background knowledge includes arithmetic operations, i.e. predicates add and mult:

```
add( A, B, C)        % C = A + B
mult( A, B, C)       %  C = A * B
```

- Induced definition of invariant:

```
inv(A,B,C,D):-       %  A = B*C + D
   mult(C,B,E),
   add(E,D,A).
```

HYPER uses non-covering approach

- Develop complete hypotheses (i.e. set of clauses) as a whole
- Search step: refine complete hypothesis
- Combinatorial complexity:
  - Search for individual clauses exponential
  - Search for complete hypothesis expected to be even much worse!

COMPLEXITY OF HYPER

- Combinatorial complexity of HYPER - super exponential
- This is main limitation!
- Still, better than many would expect
- Pruning of many hypotheses during general-to-specific search
- When a hypothesis is too specific, no refinement (specialisation) will help

HYPER: Some surprising results

Learning problem	Backgr. pred.	Pos. exam.	Neg. exam.	Refine depth	Hypos. refined	To be refined	All generated	Total size
member	0	3	3	5	20	16	85	1575
append	0	5	5	7	14	32	199	$> 10^9$
even + odd	0	6	5	6	23	32	107	22506
path	1	6	9	12	32	112	658	$> 10^{17}$
insort	2	5	4	6	142	301	1499	540021
arches	4	2	5	4	52	942	2208	3426108
invariant	2	6	5	3	123	2186	3612	18426

- “Total size” = # all hypotheses in search space within refine depth
- Cf. Total search space size and #hypos. refined
- What is easier: path(A,B) or path(A,B,Path) ?

THE “ART” OF ILP

- Much of success depends on problem formulation
- How is knowledge represented?
- What background knowledge is used?
- Background knowledge: not too little, not too much!

PREDICATE INVENTION

- Can we discover new predicates with HYPER?
- Yes, by adding “dummy” background predicates which HYPER may use as “seeds” for new predicates not mentioned elsewhere in problem definition

From Russell and Norvig, AI – A Modern Approach, 2010

"Some of the deepest revolutions in science come from the invention of new predicates and functions – for example, Galileo's invention of acceleration, or Joule's invention of thermal energy. Once these terms are available, the discovery of new laws becomes (relatively) easy."