# Symbolic AI

Andre Freitas

•Photo by Vasilyev Alexandr

# Acknowledgements

- Based on the slides of:
  - General Ideas in Inductive Logic Programming (FOPI-RG).
  - Lecture 6: Inductive Logic Programming Cognitive Systems II - Machine Learning.
  - CS 391L: Machine Learning: Rule Learning, Mooney.

# This Lecture

- Getting deeper into ILP.

# Recap: ILP

- Goal is to induce a Horn-clause definition for some target predicate *P*, given definitions of a set of background predicates *Q*.

- Goal is to find a syntactically simple Horn-clause definition, *D*, for *P* given background knowledge *B* defining the background predicates *Q*.
  - For every positive example $p_i$ of *P*

    $$D \cup B \models p_i$$

  - For every negative example $n_i$ of *P*

    $$D \cup B \not\models n_i$$

- Background definitions are provided either:
  - **Extensionally**: List of ground tuples satisfying the predicate.
  - **Intensionally**: Prolog definitions of the predicate.

# Relational Learning and Inductive Logic Programming (ILP)

- **Fixed feature vectors** are a very limited representation of instances.
- Examples or target concept may require a relational representation that includes multiple entities with relationships between them (e.g. a graph with labeled edges and nodes).
- **First-order predicate logic** is a more powerful representation for handling such relational descriptions.
- **Horn clauses** (i.e. if-then rules in predicate logic, Prolog programs) are a useful restriction on full first-order logic that allows decidable inference.
- Allows learning programs from sample I/O pairs.

# Learning Rules

- Rules are fairly easy for people to understand and therefore can help provide insight and comprehensible results for human users.
  - Frequently used in data mining applications where goal is discovering understandable patterns in data.

- Methods for automatically inducing rules from data *have been shown to build more accurate expert systems than human knowledge engineering* for some applications.

# Rule Learning vs. Knowledge Engineering

- An influential experiment with an early rule-learning method (AQ) by Michalski (1980) compared results to knowledge engineering (acquiring rules by interviewing experts).
- Knowledge engineered rules:
  - Weights associated with each feature in a rule
  - Method for summing evidence similar to *certainty factors*.
  - No explicit disjunction
- Data for induction:
  - Examples of 15 soybean plant diseases described using 35 nominal and discrete ordered features, 630 total examples.
  - 290 “best” (diverse) training examples selected for training. Remainder used for testing
    - What is wrong with this methodology?

# Experimental Results

- Rule construction time:
  - Human: 45 hours of expert consultation
  - AQ11: 4.5 minutes training on IBM 360/75
    - What doesn't this account for?

- Test Accuracy:

| | 1st choice correct | Some choice correct |
|---|---|---|
| AQ11 | 97.6% | 100.0% |
| Manual KE | 71.8% | 96.9% |

# Recap: Sequential Covering

- A set of rules is learned one at a time
- each time finding a single rule
- that covers a large number of positive instances
- without covering any negatives,
- removing the positives that it covers,
- and learning additional rules to cover the rest.

  Let *P* be the set of positive examples
  Until *P* is empty do:
  - Learn a rule *R* that covers a large number of elements of *P* but no negatives.
  - Add *R* to the list of rules.
  - Remove positives covered by *R* from *P*

- This is an instance of the greedy algorithm for minimum set covering and does not guarantee a minimum number of learned rules.
- Minimum set covering is an NP-hard problem and the greedy algorithm is a standard approximation algorithm.

# Strategies for Learning a Single Rule

- Top Down (General to Specific):
  - Start with the most-general (empty) rule.
  - Repeatedly add antecedent constraints on features that eliminate negative examples while maintaining as many positives as possible.
  - Stop when only positives are covered.
- Bottom Up (Specific to General)
  - Start with a most-specific rule (e.g. complete instance description of a random instance).
  - Repeatedly remove antecedent constraints in order to cover more positives.
  - Stop when further generalization results in covering negatives.

# Learning a Single Rule in FOIL

- Basic algorithm for instances with discrete-valued features:

```
Let A={} (set of rule antecedents)
Let N be the set of negative examples
Let P the current set of uncovered positive examples
Until N is empty do
    For every feature-value pair (literal) (F_i=V_ij) calculate
        Gain(F_i=V_ij, P, N)
    Pick literal, L, with highest gain.
    Add L to A.
    Remove from N any examples that do not satisfy L.
    Remove from P any examples that do not satisfy L.
Return the rule: A_1 ∧A_2 ∧… ∧A_n → Positive
```

# Rule Pruning in FOIL

- Prepruning method based on **minimum description length (MDL)** principle.

- Postpruning to eliminate unnecessary complexity due to limitations of greedy algorithm.

  For each rule, *R*
    For each antecedent, *A*, of rule
      If deleting *A* from *R* does not cause
        negatives to become covered
      then delete *A*

  For each rule, *R*
    If deleting *R* does not uncover any positives (since they
      are redundantly covered by other rules)
    then delete *R*

# Minimum Description Length

- Devise an encoding that maps a theory (set of clauses) into a bit string.
- Also need an encoding for examples.
- Number of bits required to encode theory <u>should not exceed number of bits to encode +ve examples</u>.

# Rule Learning Issues

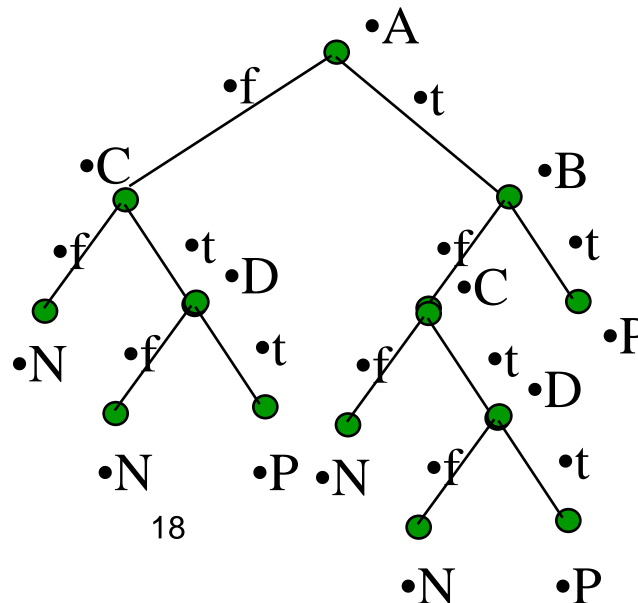
- Which is better top-down or bottom-up search?
  - **Bottom-up** is more subject to noise, e.g. the random seeds that are chosen may be noisy.
  - **Top-down** is wasteful when there are many features which do not even occur in the positive examples (e.g. text categorization).

# Rule Learning Issues

- Which is better rules or trees?
  - **Trees** share structure between disjuncts.
  - **Rules** allow completely independent features in each disjunct.
  - Mapping some rules sets to decision trees results in an exponential increase in size.

- $A \wedge B \rightarrow P$
- $C \wedge D \rightarrow P$

- What if add rule:
- $E \wedge F \rightarrow P$
- ??



# Sequential vs Simultaneous

- **Sequential covering:**
  - learn just one rule at a time, remove the covered examples and
  - repeat the process on the remaining examples
  - many search steps, making independent decisions to select earch precondition for each rule

- **Simultaneous covering:**
  - ID3 learns the entire set of disjunct rules simultaneously as part of a single search for a decision tree
  - Fewer search steps, because each choice influences the preconditions of all rules
  - Choice depends of how much data is available
    - Plentiful: sequential covering (more steps supported)
    - Scarce: simultaneous covering (decision sharing effective)

# Induction as Inverted Deduction

- **Observation:** induction is just the inverse of deduction.

- In general, machine learning involves building theories that explain the observed data.

- Given some *data D* and some *background knowledge B*, learning can be described as generating a *hypothesis h* that, together with B, explains D.

$$(\forall < x_i, f(x_i) > \in D)(B \wedge h \wedge x_i) \vdash f(x_i)$$

- The above equation casts the learning problem in the framework of deductive inference and formal logic.

# Induction as Inverted Deduction

- **Features of inverted deduction:**
  - Subsumes the common definition of learning as finding some general concept.
  - Background knowledge allows a more rich definition of when a hypothesis h is said to "fit" the data.

- **Practical difficulties:**
  - Noisy data makes the logical framework to completely lose the ability to distinguish between truth and falsehood.
  - Search is intractable.
  - Background knowledge often increases the complexity of H.

# Inverting Resolution

- Resolution is a general method for automated deduction
- Complete and sound method for deductive inference
- **Inverse Resolution Operator (propositional form):**
  - 1. Given initial clause $C_1$ and C, find a literal L that occurs in $C_1$ but not in clause C.

$C_2$: KnowMaterial v ~Study

$C_1$: PassExam v

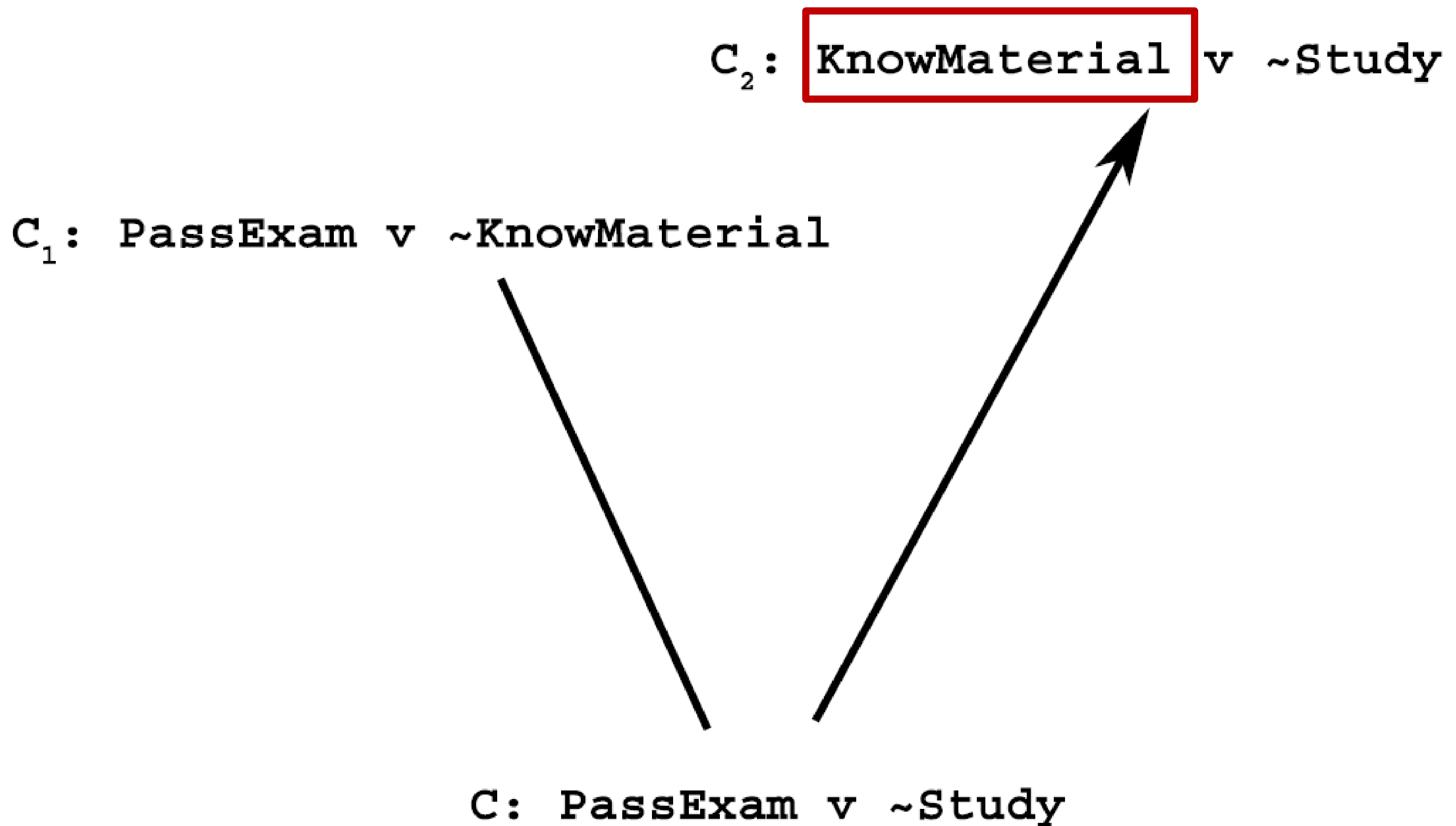
C: PassExam v ~Study

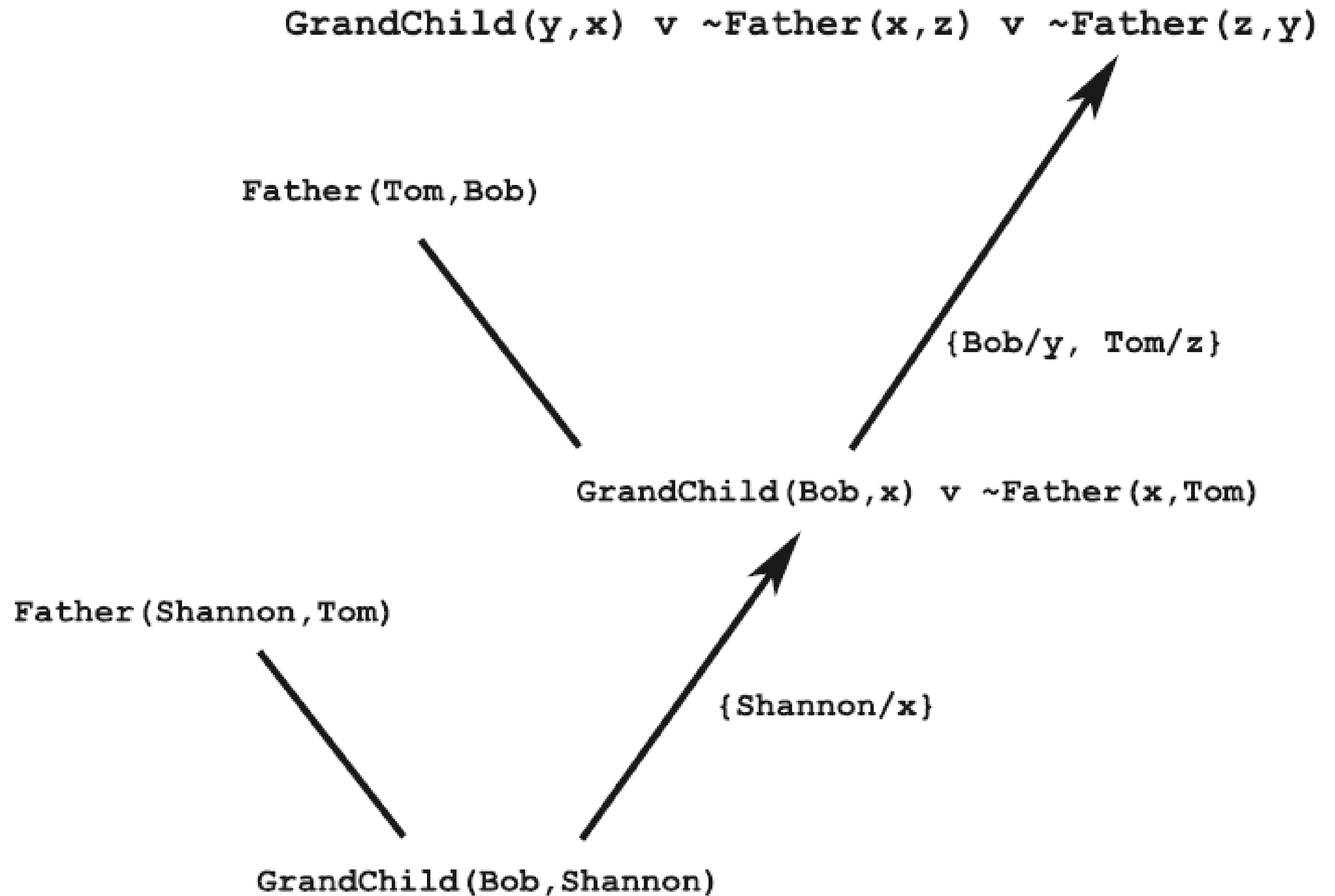
# Inverting Resolution

- Resolution is a general method for automated deduction
- Complete and sound method for deductive inference
- **<u>Inverse Resolution Operator</u> (propositional form):**
  - 1. Given initial clause $C_1$ and C, find a literal L that occurs in $C_1$ but not in clause C.
  - 2. Form the second clause $C_2$ by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{L\}$$

$$C_2 = (C - (C_1 - \{L\})) \cup \{L\}$$






$$D = \{GrandChild(Bob, Shannon)\}$$

$$B = \{Father(Shannon, Tom), Father(Tom, Bob)\}$$

# Generalization, θ-Subsumption, Entailment

interesting to consider the relationship between the $more_general_than$ relation and inverse entailment

hypothesis can also be expressed as $c(x) \leftarrow h(x)$.

$\theta - subsumption$: Consider two clauses $C_j$ and $C_k$, both of the form $H \vee L_1 \vee ... \vee L_n$, where $H$ is a positive literal and the $L_i$ are arbitrary literals. Clause $C_j$ is said to $\theta - subsume$ clause $C_k$ iff $(\exists\theta)[C_j\theta \subseteq C_k]$.

$Entailment$: Consider two clauses $C_j$ and $C_k$. Clause $C_j$ is said to $entail$ clause $C_k$ (written $C_j \vdash C_k$) iff $C_j$ follows deductively from $C_k$.

# ILP Examples

- Learn definitions of family relationships given data for primitive types and relations.

```
uncle(A,B) :- brother(A,C), parent(C,B).
uncle(A,B) :- husband(A,C), sister(C,D), parent(D,B).
```

- Learn recursive list programs from I/O pairs.

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

append([],L,L).
append([X|L1],L2,[X|L12]):-append(L1,L2,L12).
```

# Ensuring Termination in FOIL

- First empirically determines all binary-predicates in the background that form a well-founded partial ordering by computing their transitive closures.
- Only allows recursive calls in which one of the arguments is reduced according to a known well-founded partial ordering.
  - `path(X,Y) :- edge(X,Z), path(Z,Y).`

    `X` is reduced to `Z` by `edge` so this recursive call is OK
- Due to halting problem, cannot determine if an arbitrary recursive definition is guaranteed to halt.

# Inducing Recursive List Programs

- FOIL can learn simple Prolog programs from I/O pairs.
- In Prolog, lists are represented using a logical function `:[Head | Tail]`.

- Since FOIL cannot handle functions, this is re-represented as a predicate:

  `components(List, Head, Tail)`

- In general, an *m*-ary function can be replaced by a (*m*+1)-ary predicate.

# Logic Program Induction in FOIL

- FOIL has also learned
  - `append` given `components` and `null`
  - `reverse` given `append`, `components`, and `null`
  - `quicksort` given `partition, append, components,` and `null`

- Learning recursive programs in FOIL requires a complete set of positive examples for some constrained universe of constants, so that a recursive call can always be evaluated extensionally.

- Negative examples usually computed using a closed-world assumption.
  - Grows combinatorically large for higher arity target predicates.
  - Can randomly sample negatives to make tractable.

# FOIL Limitations

- Search space of literals (branching factor) can become intractable.
  - Use aspects of bottom-up search to limit search.
- Requires large extensional background definitions.
  - Use intensional background via Prolog inference.
- Requires complete examples to learn recursive definitions.
  - Use intensional interpretation of learned recursive clauses.

# FOIL Limitations (cont.)

- Requires a large set of closed-world negatives.
  - Exploit "output completeness" to provide "implicit" negatives.
- Inability to handle logical functions.
  - Use bottom-up methods that handle functions.
- Background predicates must be sufficient to construct definition, e.g. cannot learn `reverse` unless given `append`.
  - Predicate invention
    - Learn `reverse` by inventing `append`
    - Learn `sort` by inventing `insert`

# ILP Settings

Examples: Positive: bird(penguin)
bird(eagle)
bird(crow)
bird(ostrich)

Negative: bird(carp)
bird(bat)
bird(horse)

Background knowledge:

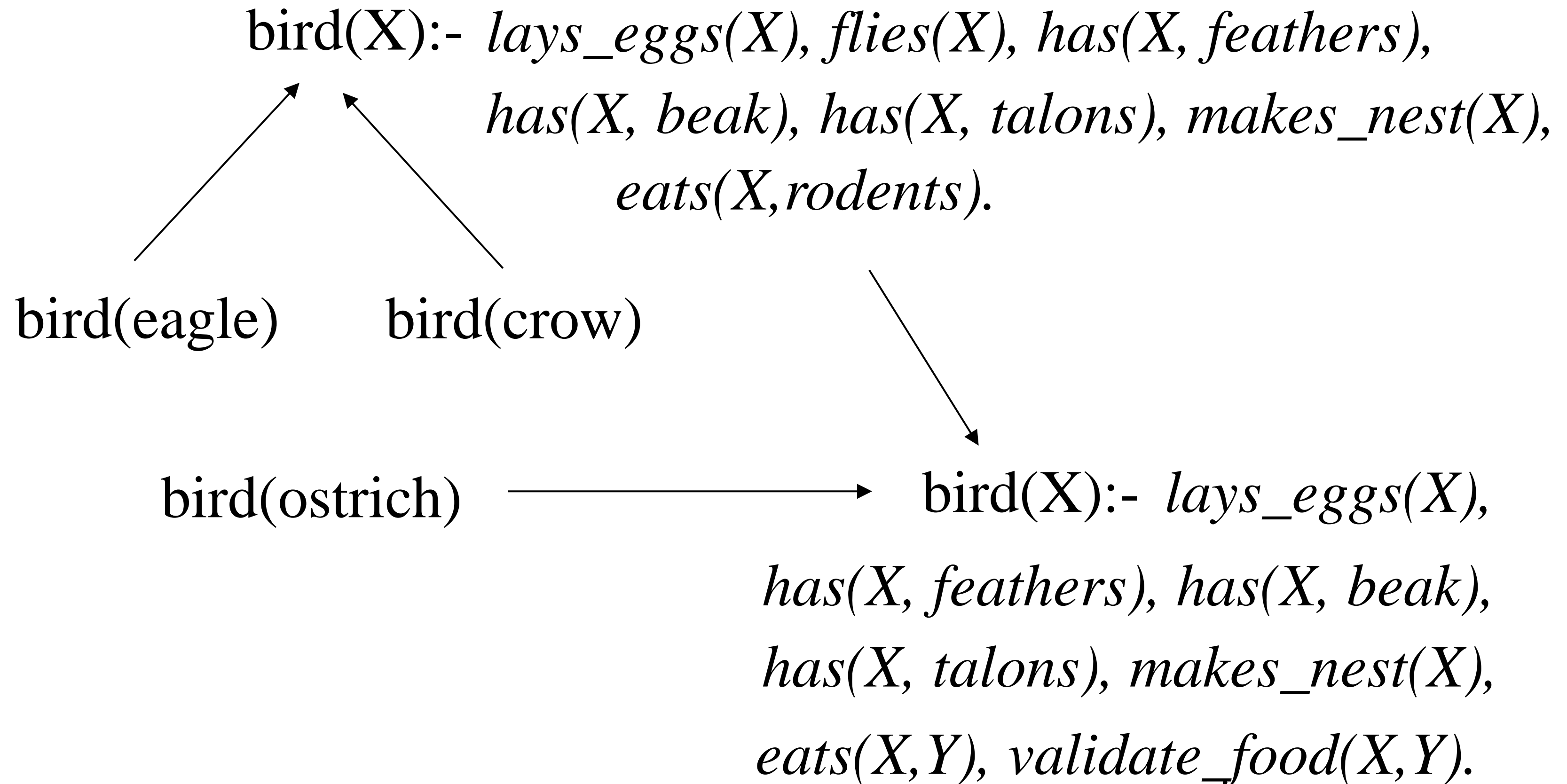
```
lays_eggs(penguin).
lays_eggs(crow).
lays_eggs(eagle).
lays_eggs(ostrich).
lays_eggs(carp).
flies(eagle).
flies(crow).
flies(bat).
swims(carp).
swims(penguin).
runs(horse).
runs(ostrich).
fish(X) :- has_scales(X), swims(X).
mammal(X):- warm_blooded(X), live_young(X).
```

Theory (one or more clauses):

```
bird(penguin).
bird(X):- lays_eggs(X), flies(X).
bird(X):- lays_eggs(X), runs(X).
```

# Bottom-Up Approach

- relative least general generalisation (rlgg)




Used in GOLEM [Muggleton, 90]

# Top-down Approach

Some ILP engines use standard top-down search algorithms: depth-first, breadth-first, A*, etc.

```
bird(X):-.
bird(X):- lays_eggs(X).
bird(X):- flies(X).
bird(X):- lays_eggs(X), flies(X).
…
```

We can improve efficiency by:

- setting a depth-bound (max clauselength).
- paying attention to clause evaluation scores - coverage, MDL.
  - re-ordering candidate clauses based on score
  - pruning candidate clauses below a score threshold
- etc.

# Practical Problem Areas

Most commonly encountered:

- Exploring large search spaces
- Positive-only data sets
- Noisy data

# Search Space

The hypothesis space is bounded by:

- Maximum clause length
- Size of background knowledge (BK)

Techniques to reduce background knowledge include:

- Excluding redundant predicates
  - Feature subset selection
  - Inverse entailment
- Replacing existing BK with compound predicates (feature construction).

# Progol and Aleph's Approach

Uses inverse entailment.

1. Randomly pick a positive example, *p*.
2. Define the space of possible clauses that could entail that example.
   - Generate the bottom clause, $\bot$
   - $\bot$ contains all the literals defined in BK that could cover *p*.
3. Search this space.

# Noisy Data

- Techniques to avoid over-fitting.
  - Pre-pruning: limit length of clauses learned
  - Post-pruning: generalise/merge clauses that have a small cover set.
  - Leniency: don't insist on a perfect theory
- Embed the uncertainty into the learning mechanism
  - Stochastic Logic Programs
  - Fuzzy ILP
  - **Diff ILP**

# Numerical Reasoning

e.g. bird(X):- number_of_legs(X,Y), lessthan(Y, 3).

Many ILP engines don’t handle numerical reasoning without help.

- Lazy evaluation [Srinivasan & Camacho, 99]
- Farm it out to another process [Anthony & Frisch, 97]
- (if possible) add predicates to the background knowledge
- First-Order Regression [Karolic & Bratko, 97]

# Inventing Predicates

Some ILP engines can invent new predicates and add them to the existing BK.

e.g. Progol uses constraints to call a predicate invention routine.

```
:- constraint(invent/2)?

invent(P,X):-    {complicated code that includes asserts}.
```

FOIL only uses extensional BK and so can't use this method.

# ILP Systems

- Top-Down:
  - FOIL (Quinlan, 1990)
- Bottom-Up:
  - CIGOL (Muggleton & Buntine, 1988)
  - GOLEM (Muggleton, 1990)
- Hybrid:
  - CHILLIN (Mooney & Zelle, 1994)
  - PROGOL (Muggleton, 1995)
  - ALEPH (Srinivasan, 2000)

# Aleph

- file.b: contains the background knowledge (intentional and extensional), the search, language restrictions and types restrictions and the system parameters. (as Prolog clauses).

- file.f: contains the positive examples (only ground facts) to be learned with Aleph;

- file.n: contains the negative examples (only facts without variables) - optional.

# Mode Declarations

- Describe the relations (predicates) between the objects and the type of data.

- Declarations inform Aleph if the relation can be used in the head (modeh declarations) or in the body (modeb declarations) of the generated rules.

mode(Recall number, PredicateMode)

- For instance, if we want to declare the predicate parent_of(P,D) the recall should be 2, because the daughter D, has a maximum of two parents P.

- Recall number of grandparents(GP,GD) = ?

- The Modes indicates the predicate format, and can be described as:

  predicate(ModeType1, ModeType2, ... , ModeTypen)

  - ’+’, specifying that when a predicate p appears in a clause, the corresponding argument is an input variable;
  - ’-’, specifying that the corresponding argument is an output variable;
  - ’#’, specifying that the corresponding argument is a constant.

# Mode: Example

- Example: for the learning relation uncle of(U,N) with the background knowledge parent of(P,D) and sister of(S1,S2), the mode declarations could be:

```
:- modeh(1,uncle_of(+person,+person)).
:- modeb(*,parent_of(-person,+person)).
:- modeb(*,parent_of(+person,-person)).
:- modeb(*,sister_of(+person,-person)).
```

# Types

person(john)

person(leihla)

person(richard)

...

# Determinations

- Determination statements declare the predicate that can be used to construct a hypothesis

determination(Target Pred/Arity t, Body Pred/Arity b).

determination(aunt_of/2, parent_of/2).

Determinations are only allowed for 1 target predicate on any given run of Aleph: if multiple target determinations occur, the first one is chosen

# Positive and Negative Examples

- Positive examples: file with an extension .f
- Negative examples: file with an extension .n

…

```
% Mode declarations

:- modeh(1,aunt_of(+person,+person))?
:- modeb(*,parent_of(-person,+person))?
:- modeb(*,parent_of(+person,-person))?
:- modeb(*,sister_of(+person,-person))?

% Types

person(jane).
person(henry).
person(sally).
person(jim).
person(sam).
person(sarah).
person(judy).

% Background knowledge

parent_of(Parent,Child) :- father_of(Parent,Child).
parent_of(Parent,Child) :- mother_of(Parent,Child).

father_of(sam,henry).

mother_of(sarah,jim).

sister_of(jane,sam).
sister_of(sally,sarah).
sister_of(judy,sarah).
```

```
% Examples

aunt_of(jane,henry).
aunt_of(sally,jim).
aunt_of(judy,jim).
```

```
:- aunt_of(henry,sally).
:- aunt_of(judy,sarah).
```

# Output

```
[Generalising aunt_of(jane,henry).]
[Most specific clause is]

aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).

[Learning aunt_of/2 from positive examples]
[C:-0,12,11,0 aunt_of(A,B).]
[C:6,12,4,0 aunt_of(A,B) :- parent_of(C,B).]
[C:6,12,3,0 aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).]
[C:6,12,3,0 aunt_of(A,B) :- parent_of(C,B), sister_of(A,D).]
[C:4,12,6,0 aunt_of(A,B) :- sister_of(A,C).]
[5 explored search nodes]
f=6,p=12,n=3,h=0
[Result of search is]

aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).

[3 redundant clauses retracted]

aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).

[Total number of clauses = 1]

[Time taken 0.02s]
```