Version Space Search

- Version space search is one of the first Machine Learning algorithms.
- For us, introduction to Inductive Logic Programming.
- Our (Tom Mitchell's) toy data:

Example (Tennis Dataset)

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Overcast	Mild	High	Weak	Yes
D5	Overcast	Mild	High	Strong	Yes
D6	Overcast	Hot	Normal	Weak	Yes
D7	Rain	Mild	High	Strong	No

Version Space Search

- Our **hypothesis** is a conjunction of attribute tests that imply $PlayTennis = yes$.
  - $h = \langle ?, Cold, High, ?, ?, ? \rangle$ represents the hypothesis $Temperature = cold \ \& \ Humidity = high \Rightarrow PlayTennis = yes$.
    - ? is satisfied by any value
    - $\emptyset$ cannot be satisfied
  - For binary attributes, we have $3^{|\#attributes|} + 1$ hypotheses
    - hypotheses with $\emptyset$ are not satisfiable, therefore they are equivalent.
    - We perform a systematic search.
    - The hypothesis space is partially ordered by the subsumption.

Definition (More general, more specific)

- The hypothesis h_g is **more general** than the hypothesis $h_g \succeq h_s$ iff any sample that satisfies h_s satisfies also h_g.
- In the above case, the hypothesis $h_s, h_g \succeq h_s$ is called **more specific that** h_g.

- $\langle ?, ?, ?, ? \rangle$ is more general than $\langle Sunny, \ldots, Same \rangle$.
- The most general hypothesis $\langle ?, ?, ?, ? \rangle$ is satisfied by all data.
- The most specific hypothesis $\langle \emptyset, \ldots \rangle$ is not satisfied by any data.
- The hypothesis space for a **lattice** partially ordered by the 'more general' relation.

Find–S

- We search for a hypothesis satisfied by all positive examples and no negative example.

Find–S (to be improved)

procedure FIND-S:(X dataset with the goal attritute *yes*/*no*)
    $h \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ # the most specific hypothesis
    for each positive data sample x_i **do**
        for each attribute condition $X_j = x_{i,j}$ in h **do**
            if x_i does not satisfy $X_j = x_{i,j}$ **then**
                replace the condition by
                    a closest more general condition satisfied by x_i
            end if
        end for
    end for
    return h
end procedure

Version Space Search

Example (Tennis Dataset)

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Overcast	Mild	High	Weak	Yes
D5	Overcast	Mild	High	Strong	Yes
D6	Overcast	Hot	Normal	Weak	Yes
D7	Rain	Mild	High	Strong	No

Version Space

- Now we look for all hypotheses consistent with the data.

Definition (Version Space)

- **The version space** for the hypothesis space H and the data X is a subset of H that is consistent with X

$$VS(H,X) = \{h \in H | Consistent(h,X)\}.$$

- The version space is characterized by the most general and the most specific boundary.
- Any hypothesis between these boundaries is consistent with the data.

Definition (General Boundary)

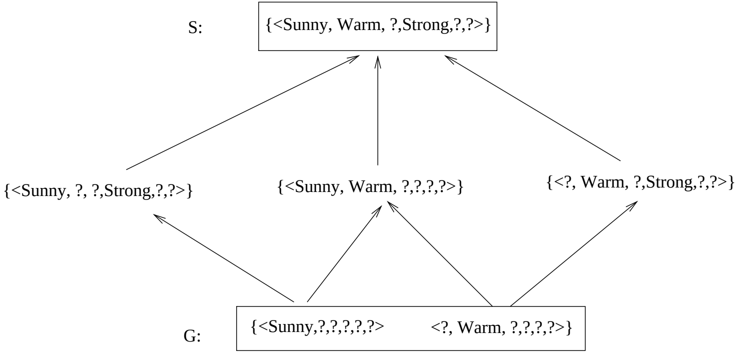
- **The general boundary** for the hypothesis space H and the data X is a set of most general hypothesis from H that are consistent with X

$$G(H,X) = \{g \in H | Consistent(g,X) \& (\nexists g_1 \in H)[g_1 \succ g \& Consistent(g_1,X)]\}.$$

Definition (Specific Boundary)

- **The specific boundary** for the hypothesis space H and the data X is a set of most specific hypothesis from H that are consistent with X

$$S(H,X) = \{s \in H | Consistent(s,X) \& (\nexists s_1 \in H)[s \succ s_1 \& Consistent(s_1,X)]\}.$$



- We search for a hypothesis satisfied by all positive examples and no negative example.

```
procedure CANDIDATE–ELIMINATION:(X data,the goal att. yes/no)
    G ← {⟨?, ?, ?, ?⟩}, S ← {⟨∅, ∅, ∅, ∅⟩} # general,specific
    for each data sample x_i do
        if x_i is positive then
            remove from G all h inconsistent with x_i
            for each s ∈ S inconsistent with x_i do
                add to S all minimal generalizations h
                Consistent(h, x_i)&(∃g ∈ G)(g ⪰ h)
                remove from S {s|(∃s_1 ∈ S)(s ≻ s_1)} # not most specific
            end for
        else x_i is negative example
            remove from S all h inconsistent with x_i
            for each g ∈ G inconsistent with x_i do
                add to G all minimal specifications h
                Consistent(h, X)&(∃s ∈ S)(h ⪰ s)
                remove from G {g|(∃g_1 ∈ G)(g_1 ≻ g)} # not most gen.
            end for
        end if
    end for
    return G, S
end procedure
```

Literature

- A. Cropper and S. Dumancic. Inductive logic programming at 30: a new introduction. CoRR, abs/2008.07912, 2020.
- S. Muggleton & all.: Meta-interpretive learning: application to grammatical inference, http://www.doc.ic.ac.uk/~shm/FLOC_ILP/Paper03.pdf

Predicate Logic

- Recall predicate logic.
- CNF, DNF the conjunctive and disjunctive normal form
- **clause**: a disjunction of literals $father(X, Y) \vee \neg parent(X, Y) \vee \neg male(X)$
- **Horn clauses** with at most one positive literal, written as a rule
  - **definite clause** $father(X, Y) : -male(X), parent(X, Y)$.
  - **fact** - no negative literal $male(adam)$.
  - **goal clause** - no positive literal $false : -father(X, bob)$.
- **Ground term, clause** - a term, a clause without variables.
- We have our data in the form of a set of clauses B, E^+, E^-,
  - the background knowledge B is a set of (Horn) clauses,
  - the positive and examples E^+, E^- are sets of ground literals (facts).

Example

$$B = \left\{ \begin{array}{c} lego\_builder(alice). \\ enjoys\_lego(A) := lego\_builder(A). \\ estate\_agent(dave). \\ enjoys\_lego(alice). \\ enjoys\_lego(claire). \end{array} \right\}$$

$$E^+ = \{happy(alice).\}$$

$$E^- = \left\{ \begin{array}{c} happy(bob). \\ happy(claire). \\ happy(dave). \end{array} \right\}$$

Substitution, Subsumption

- Clauses ale implicitly generally quantified.
- They should not have a variable with the same name.

Definition (Substitution, Subsumption)

- Given a **substitution** $\theta = \{v_i/t_i\}$ and formula F. $F\theta$ is formed by replacing every variable v_i in F by t_i.
- Substitution θ **unifies** atom A and B in the case $A\theta = B\theta$.
- Atom A subsumes atom B, $A \succeq B$, iff there exists a substitution θ such that $A\theta = B$.
- **Clause C subsumes clause D**, $C \succeq D$, iff there exists a substitution θ such that $C\theta \subseteq D$.

Example

- $C_1 = f(A, B) : -head(A, B)$.
- $C_2 = f(X, Y) : -head(X, Y), empty(Y)$.
- C_1 subsumes C_2 since $C_1\theta \subseteq C_2$ with $\theta = \{A/X, B/Y\}$.

Definition (Generalisation)

- Clause C **is more general than** clause D, iff $C \models D$.
- Clause C is more general than clause D with respect to B, iff $B, C \models D$.
  - B is the **background knowledge**.

Example

- Statement A: Daffy Duck can fly. $can\_fly(daffy)$
- Statement B: All ducks can fly. $can\_fly(X) \succeq can\_fly(daffy)$.

Example

- Statement C: Marek lives in London.
- Statement D: Marek lives in England.

- $lives(marek, london)$
- $lives(marek, england)$
- Background knowledge $lives(x, england) : -lives(x, london)$.
- $B, C \models D$, 'C is more general than D with respect to B'.
- $C \succeq D$ with respect to B.
- `http://www.doc.ic.ac.uk/~shm/FLOC_ILP/Lecture1.1.pdf`

ILP general logical setting

Definition (Hypothesis Properies)

The background knowledge B and the hypothesis H should entail E, that is:

Necessity	B	$\not\models$	E^+	we need H
Sufficiency (úplnost)	$B \,\& H$	$\models$	E^+	H explains positive examples
Weak consistency	$B \,\& H$	$\not\models$	$\bot$	H does not contradict B
(Strong) **consistency**	$B \,\& H \,\& E^-$	$\not\models$	$\bot$	... neither negative examples

ILP task

- Given
  - B background knowledge (logic program)
  - E^+, E^- examples – sets of ground unit clauses
- Given B, E find a logic program H such that is necessary, sufficient and consistent.
- Often, we assume noisy data and accept some errors, but we try to minimize them.

Example

$$B = \left\{ \begin{array}{c} lego\_builder(alice). \\ lego\_builder(bob). \\ estate\_agent(claire). \\ estate\_agent(dave). \\ enjoys\_lego(alice). \\ enjoys\_lego(claire). \end{array} \right\}$$

$$E^+ = \{ happy(alice). \}$$

$$E^- = \left\{ \begin{array}{c} happy(bob). \\ happy(claire). \\ happy(dave). \end{array} \right\}$$

Our hypothesis space:

$$\mathcal{H} = \left\{ \begin{array}{c} h_1 : \; happy(A) :- lego\_builder(A). \\ h_2 : \; happy(A) :- estate\_agent(A). \\ h_3 : \; happy(A) :- enjoys\_lego(A). \\ h_4 : \; happy(A) :- lego\_builder(A), estate\_agent(A). \\ h_5 : \; happy(A) :- lego\_builder(A), enjoys\_lego(A). \\ h_6 : \; happy(A) :- estate\_agent(A), enjoys\_lego(A). \end{array} \right\}$$

- $B \cup h_1 \models happy(bob)$ therefore h_1 is inconsistent.
- $B \cup h_2 \not\models happy(alice)$ therefore h_2 is incomplete.
- $B \cup h_3 \models happy(claire)$ therefore h_3 is inconsistent.
- $B \cup h_4 \not\models happy(alice)$ therefore h_4 is incomplete.
- h_5 is both complete and consistent.
- $B \cup h_6 \not\models happy(alice)$ therefore h_1 in incomplete.

Hypothesis Space

- To specify (restrict) the hypothesis space usually mode declarations are used.

Definition (Mode declarations)

Mode declarations denote which literals may appear in the head/body of a rule. A mode declaration is of the form:

$$mode(recall, pred(m_1, m_2, \ldots, m_a))$$

where *recall* is the maximum number of occurrences of the predicate
m_i are the argument types and they may be assigned as input $+$, output $-$, constant $\#$.

Example

modeb(2,parent(+person,-person)).
modeh(1,happy(+person)).
modeb(*,member(+list,-element)).
modeb(1,head(+list,-element)).

A. Cropper and S. Dumancic. Inductive logic programming at 30: a new introduction. CoRR, abs/2008.07912, 2020.

Non-monotonic reasoning

- In Prolog, there is negation as a failure.

Example

$$Program = \left\{ \begin{array}{c} sunny. \\ happy :- sunny, not\ weekday. \end{array} \right\}$$

- Prolog tries to prove *weekday*.
- It does not prove it, therefore it concludes *happy*.
- With additional knowledge *weekday* some of entailments are not true any more.

Definition (Normal logic program)

Normal logic programs may include negated literals in the body of a clause, e.g.

$$h :- b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m.$$

Aleph ILP system (based on Progol)

- Given
  - A set of mode declaration M
  - Background knowledge B in the form of a normal program allows negation, with the semantics negation as a failure
  - Positive E^+ and negative E^- examples as a set of ground facts
- Return: A normal program hypothesis H that:
  - H is consistent with M
  - $\forall e \in E^+$, $H \cup B \models e$ (H is complete)
  - $\forall e \in E^-$, $H \cup B \not\models e$ (H is consistent).

Aleph

1. Select a positive example to generalize.
2. Construct the most specific clause consistent with M that entails the example (the bottom clause).
3. Search for a clause more general than the bottom clause.

- Add the clause to the hypothesis and remove all examples covered.
- If a positive example left, return to step 1.

Bottom Clause Construction

- The purpose is to bound the search in the step in 3.
- Without mode declarations, the bottom clause may have infinite cardinality.

Definition (Bottom clause)

Let H be a clausal hypothesis and C be a clause. The bottom clause $\bot(C)$ is the most specific clause such that:

$$H \cup \bot(C) \models C.$$

Example (Bottom clause)

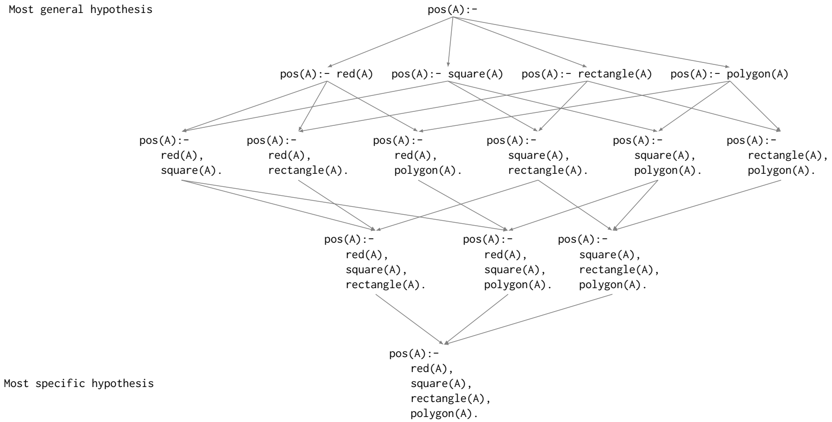
$$M = \left\{ \begin{array}{c} :-modeh(*, pos(+shape)). \\ :-modeb(*, red(+shape)). \\ :-modeb(*, square(+shape)). \\ :-modeb(*, triangle(+shape)). \\ :-modeb(*, polygon(+shape)). \end{array} \right\} B = \left\{ \begin{array}{c} red(s1). \\ blue(s2). \\ square(s1). \\ triange(s2). \\ polygon(A) :- rectangle(A). \\ rectangle(A) :- square(A). \end{array} \right\}$$

Let e be the positive example $pos(s1)$. Then:

$$\bot(e) = pos(A) :- red(A), square(A), rectangle(A), polygon(A).$$

Clause Search

- Aleph performs a bounded breadth-first search to enumerate the shorter clauses before longer ones.
- The search is bounded by several parameters (max. clause size, max. proof depth).



Aleph 2, Popper, Flex

- Aleph default evaluation function is **coverage** defined as $P - N$,
  - P is the number of positive examples covered by the clause
  - N is the number of negative examples covered by the clause
  - that means it accepts some noise.
- It starts from the most general one $pos(A) : -.$
- It tries to specialize the clause
  - by adding literals to the body of it, which it selects from the bottom clause
- or by instantiating variables.
- Each specialization is called **refinement**.
- Aleph Advantages
  - one Prolog file, easy to download and use.
    - https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html
  - It has good empirical performance.
  - Allows numerical reasoning, user defined cost functions, handles noisy data.
- Aleph Disadvantages
  - It has many parameters to tune.
  - It struggles to learn recursive programs and optimal programs
    - since it learns only a single clause a time.

Metagol

- Given
  - A set of metarules M
  - Background knowledge B in the form of a normal program
  - Positive E^+ and negative E^- examples as a set of facts (atoms).
- Return: A definite program hypothesis H that:
  - H is consistent with M
  - $\forall e \in E^+$, $H \cup B \models e$ (H is complete)
  - $\forall e \in E^-$, $H \cup B \not\models e$ (H is consistent)
  - $\forall h \in H$, $\exists m \in M$ such that $h = m\theta$
    - where θ is a substitution that grounds all the existentially quantified variables in m.

Example (Metarule)

- An example is the **chain** metarule $P(A, B) \leftarrow Q(A, C), R(C, B)$
- that allows Metagol to induce programs such as

$$
\begin{aligned}
f(A, B) &\ :-\ tail(A, C), tail(C, B). \\
grandparent(A, B) &\ :-\ parent(A, C), parent(C, B).
\end{aligned}
$$

Metagol

- Metagol is a form of ILP besed on a Prolog meta-interpreter.

Metagol

1. Select a positive example to generalize.
    - If none exists, test the hypothesis on the negative examples.
        - If the hypothesis does not entail any negative example stop and return the hypothesis.
        - otherwise backtrack to a choice point at step 2 and continue.
2. Try to prove the atom by:
    - using given BK or an already induced clauses
    - unifying the atom with the head of a metarule
    - binding the variables in a metarule to symbols in the predicate and constant signatures
    - save the substitution
    - try to prove the body of the metarule by treating the body atoms as examples and applying step 2 to them.

Recursion

- Metagol can learn recursive programs.

Example (Reachability)

Consider learning the concept of *reachability* in a graph. Without recursion, with the maximal depth 4 we could learn:

$$
\begin{aligned}
reachable(A, B) &:- \; edge(A, B).\\
reachable(A, B) &:- \; edge(A, C), edge(C, B).\\
reachable(A, B) &:- \; edge(A, C), edge(C, D), edge(D, B).\\
reachable(A, B) &:- \; edge(A, C), edge(C, D), edge(D, E), edge(E, B).
\end{aligned}
$$

With recursion, we can learn:

$$
\begin{aligned}
reachable(A, B) &:- \; edge(A, B).\\
reachable(A, B) &:- \; edge(A, C), reachable(C, B).
\end{aligned}
$$

Iterative deepening

iterative deepening

- Metagol uses iterative deepening to search for hypotheses.
  - at depth $d = 1$, at least one metasub.
  - at iteration d, it introduces $d - 1$ new predicate symbols and is allowed to use d clauses.

Metagol Example

Example (Kinship example)

$$B = \left\{ \begin{array}{c} mother(ann, amy).mother(ann, andy). \\ mother(amy, amelia), mother(amy, bob). \\ mother(linda, gavin). \\ father(steve, amy).father(steve, andy). \\ father(andy, sponegebob).father(gavin, amelia). \end{array} \right\}$$

$$metarule(ident, [P, Q], [P, A, B], [[Q, A, B]]).$$
$$metarule(chain, [P, Q, R], [P, A, B], [[Q, A, C], [R, C, B]]).$$

$$E^+ = \left\{ \begin{array}{c} grandparent(ann, amelia). \\ grandparent(steve, amelia). \\ grandparent(ann, spongebob). \\ grandparent(linda, amelia). \end{array} \right\}$$
$$E^- = \{grandparent(amy, amelia).\}$$

Tracing Metagol

- It select the first example to generalize $grandparent(ann, amelia)$.
- It tries to prove it from BK and induced clauses. It fails.
- Metagol tries to use the first metarule:

$$grandparent(ann, amelia) :- Q(ann, amelia).$$

  stores $sub(ident, [grandparent, Q])$
- and tries to unify Q, but fails.
- Metagol tries to use the second metarule:

$$grandparent(ann, amelia) :- Q(ann, C), R(C, amelia).$$

  stores $sub(chain, [grandparent, Q, R])$
- and recursively tries to prove $Q(ann, C)$ and $R(C, amelia)$.
- It succeedes with the metasum $sub(chain, [grandparent, mother, mother])$
- and induces the first clause;

$$grandparent(A, B) :- mother(A, C), mother(C, B).$$

Metagol Trace 2

- Then, it select the second example to generalize $grandparent(steve, amelia)$.
- It tries to prove it from BK and induced clauses. It fails.
- Metagol can again use the second metarule with another substitution: stores $sub(chain, [grandparent, father, mother])$
- and induces the second clause;

$$grandparent(A, B) :- father(A, C), mother(C, B).$$

- Given no bound on the program size, the Metagol would prove the other two examples the same way and form the program:

$$\begin{aligned} grandparent(A, B) &:- mother(A, C), mother(C, B). \\ grandparent(A, B) &:- father(A, C), mother(C, B). \\ grandparent(A, B) &:- father(A, C), father(C, B). \\ grandparent(A, B) &:- mother(A, C), father(C, B). \end{aligned}$$

In praxis, it learns:

$$\begin{aligned} grandparent(A, B) &:- grandparent\_1(A, C), grandparent\_1(C, B). \\ grandparent\_1(A, B) &:- father(A, B). \\ grandparent\_1(A, B) &:- mother(A, B). \end{aligned}$$

Tail Recursive Metarule

Example (Tail Recursive Metarule)

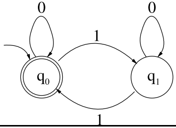
- An example is the **tail recursive** metarule $P(A, B) \leftarrow Q(A, C), P(C, B)$
- Metagol can also learn mutually recursive programs, such:

$$
\begin{aligned}
&even(0). \\
even(A) &:- \quad successor(A, B), even\_1(B). \\
even\_1(A) &:- \quad successor(A, B), even(B).
\end{aligned}
$$

We even do not have to provide the concept of an odd number. We can let the Metagol to invent such predicate ($even\_1$).

Automata Example

(a)

Finite acceptor	Production rules	Definite Clause Grammar (DCG)
	$q_0 \rightarrow$	$q_0([], []) \leftarrow$
	$q_0 \rightarrow 0\ q_0$	$q_0([0\|A], B) \leftarrow q_0(A, B)$
	$q_0 \rightarrow 1\ q_1$	$q_0([1\|A], B) \leftarrow q_1(A, B)$
	$q_1 \rightarrow 0\ q_1$	$q_1([0\|A], B) \leftarrow q_1(A, B)$
	$q_1 \rightarrow 1\ q_0$	$q_1([1\|A], B) \leftarrow q_0(A, B)$

(b)

E^+	E^-	Meta-interpreter	Ground facts
λ	1	$parse(S) \leftarrow parse(q0, S, [])$.	$acceptor(q0) \leftarrow$
0	01		$delta1(q0, 0, q0) \leftarrow$
00	10	$parse(Q, [], []) \leftarrow acceptor(Q)$.	$delta1(q0, 1, q1) \leftarrow$
11	001	$parse(Q, [C\|X], Y) \leftarrow$	$delta1(q1, 0, q1) \leftarrow$
000	010	$delta1(Q, C, P)$,	$delta1(q1, 1, q0) \leftarrow$
011	100	$parse(P, X, Y)$.	
101	111		

Fig. 1 (**a**) Parity acceptor with associated production rules, DCG; (**b**) positive examples (E^+) and negative examples (E^-), Meta-interpreter and ground facts representing the Parity grammar

http://www.doc.ic.ac.uk/~shm/FLOC_ILP/Paper03.pdf

Louise Example

Example (Module)

```
:-module(anbn, [background_knowledge/2
   ,metarules/2
   ,positive_example/2
   ,negative_example/2
   ,a/2
   ,b/2
   ]).
```

Example (Background knowledge)

```
background_knowledge(s/2,[a/2,b/2]).
a([a|T],T).
b([b|T],T).
```

Example (Metarules)

metarules(s/2,[chain]).
% (Chain) $\exists$.P,Q,R $\forall$.x,y,z: P(x,y)$\leftarrow$ Q(x,z),R(z,y)

Louise Example Continued

Example (Positive Examples)

```
positive_example(s/2,E):-
  member(E, [%s([a,b],[])
   s([a,a,b,b],[])
   ]).
```

Example (Negative Examples)

```
negative_example(s/2,E):-
  member(E,[s([a,a],[])
   ,s([b,b],[])
   ,s([a,a,b],[])
   ,s([a,b,b],[])
   ]).
```

Example (Parameter Tuning)

```
:- auxiliaries:set_configuration_option(clause_limit, [3]).
:- auxiliaries:set_configuration_option(max_invented, [1]).
:- auxiliaries:set_configuration_option(unfold_invented, [true]).
```

Louise Example Continued 2

Example (Learned Program)

```
?- learn(s/2).
s(A,B):-a(A,C),b(C,B).
s(A,B):-a(A,C),s(C,D),b(D,B).
true.
```

ASPAL algorithm

- ASPAL uses Answer Set Programming.
- ASP program can have one, many, or none models (answer sets).
- Computation in ASP is the process of finding models.
- We may specify the range of the number of clauses from a set beeing true.
  $0\{sunny., weekday., happy(A) :- lego\_builder(A)\}3$
- We may specify an evaluation function to optimize (like to minimize the number of 'true' clauses, e.g. the size of the hypothesis.

ASPAL

- Generate all possible rules consistent with the given mode declarations.
  Assign each rule a unique identifier and add an guessable atom in each rule.
- Use an ASP solver to find a minimal subset of the rules
  by formulating the problem as an ASP optimization problem.

Example (ASPAL)

$$B = \left\{ \begin{array}{c} bird(alice). \\ bird(betty). \\ can(alice, fly). \\ can(betty, swim). \\ ability(fly). \\ ability(swim). \end{array} \right\}$$

$$M = \left\{ \begin{array}{c} modeh(1, penguin(+bird)). \\ modeb(1, bird(+bird)). \\ modeb(*, not\ can(+bird, \#ability)). \end{array} \right\}$$

$$\begin{array}{rcl} E^{+} & = & \{penguin(betty).\} \\ E^{-} & = & \{penguin(alice).\} \end{array}$$

Given the modes, the possible rules are:

$$\begin{array}{rcl} penguin(X) & :- & bird(X). \\ penguin(X) & :- & bird(X), not\ can(X, swim). \\ penguin(X) & :- & bird(X), not\ can(X, fly). \\ penguin(X) & :- & bird(X), not\ can(X, swim), not\ can(X, fly). \end{array}$$

ASPAL replaces constants and adds extra literal:

$$\begin{array}{rcl} penguin(X) & :- & bird(X), rule(r1). \\ penguin(X) & :- & bird(X), not\ can(X, C1), rule(r2, C1). \\ penguin(X) & :- & bird(X), not\ can(X, C1), not\ can(X, C2), rule(r3, C1, C2). \end{array}$$

ASPAL passes to an ASP solver:

$bird(alice).$

$bird(betty).$

$can(alice, fly).$

$can(betty, swim).$

$ability(fly).$

$ability(swim).$

$penguin(X) :- bird(X), rule(r1).$

$penguin(X) :- bird(X), not\ can(X, C1), rule(r2, C1).$

$penguin(X) :- bird(X), not\ can(X, C1), not\ can(X, C2), rule(r3, C1, C2).$

$0\{rule(r1), rule(r2, fly), rule(r2, swim), rule(r3, fly, swim)\}4$

$goal :- penguin(betty), not\ penguin(alice).$

$:- not\ goal.$

- The answer is: $rule(r2, c(fly))$
- Which is translated to a program:

$$penguin(A) :- bird(A), not\ can(A, fly).$$

ILP aplications

- Bioinformatics
  - ILP can make predictions based on the (sub)structured biological data.
  - Predict mutagenic activity of molecules and alert the causes of chemical cancers
  - learning protein folding signatures.
- Robot scientist.
  - BK knowledge represents the relationship between protein-coding sequences, enzymes, and metbolites in pathway.
  - Automatically generates hypotheses, run experiments, iterprets results.
- Games
  - Sokoban
  - Bridge
  - Checkers.

Table of Contens