Practical Tools for Knowledge Representation and Nonmonotonic Reasoning

Ilkka Niemelä and Mirek Truszczyński

AAAI 2000 — Tutorial SA2

Focus

- Part 1: Stable Logic Programming (SLP)
- Part 2: Smodels — implementation of an SLP system

Overview — part 1

- Motivation
- Knowledge representation roots
- Default logic primer
- Logic programming roots
- Stable Logic Programming (SLP)
- Answer Set Programming (ASP)
- Examples

French Phrases, Italian Soda

- Mark T. Zegareli, *Dell Logic Puzzles*, page 43, June 1999
- Six people sit at a round table
- Each drinks a different kind of soda
- Each plans to visit a different French-speaking country
- Based on clues — find out:

  — what each of them is drinking

  — where each of them is going

French Phrases, Italian Soda

- The person who is planning a trip to Quebec, who drank either blueberry or lemon soda, didn't sit in seat number one.
- Jeanne didn't sit next to the person who enjoyed the kiwi soda.
- The person who has a plane ticket to Belgium, who sat in seat four or seat five, didn't order the tangelo soda.

How to find a good route?

- Given:
  - a set of places to visit
  - the set of pairs of places that have direct connections
- Find a closed route that takes you through each place exactly once
- Hamiltonian cycle problem
- Traveling salesman problem

How to move blocks around?

- Find a plan!
- Blocks on the table are arranged in stacks
- They need to be rearranged into different stacks
- At each step
  - only top block in each stack can be moved
  - can be placed on top of another stack or on the table

Other problems

- Classical AI puzzles and problems:
  - missionaries and cannibals, 15-puzzle, magic squares, ...
- Constraint satisfaction problems
  - propositional satisfiability
- Combinatorics
  - graph problems, combinatorial optimization, combinatorial structures

Goal

- WANTED: An approach for modeling and solving such problems
- Diverse domains
- Spatial and temporal reasoning
- Constraints
- Incomplete information
- Frame problem

Solution

- Answer Set Programming (ASP) paradigm
- Stable Logic Programming (SLP) — its implementation

ASP — where does it come from?

- Knowledge Representation
  - Commonsense reasoning
  - Defeasible reasoning
- Declarative programming
  - Logic programming

Roots — KR

- Commonsense reasoning
- Defeasible inferences
- Elaboration tolerance
- Representing the frame problem
- Problems with FOL
  - monotonicity

Roots — KR

- Nonmonotonic logics
  - Default logic; Reiter 1980
  - Autoepistemic logic; Moore 1984
  - Circumscription; McCarthy 1980

Default logic

- Extension of FOL by *defeasible* inference rules — *defaults*:

$$\frac{\alpha\colon \beta_{\mathbf{1}}, \ldots, \beta_{\mathbf{k}}}{\gamma}$$

- If
  - α is known and
  - if it is known that all β_is are possible

  then γ is known

- Default theory: (D, W)

Default logic — semantics

- Assume what's possible
  - that is, pick a theory S
  - β is possible if $S \not\vdash \neg\beta$
- default

$$\frac{\alpha : \beta}{\gamma} \quad \text{yields } S\text{-applicable rule} \quad \frac{\alpha}{\gamma}$$

Default logic — semantics

- Default proof — $\vdash_{\Delta,S}$
- Uses:
  - W
  - tautologies
  - modus ponens
  - S-applicable rules
- S is an *extension* of a default theory Δ if

$$S = \{\varphi : \vdash_{\Delta,S} \varphi\}$$

Default logic — example

- $$W = \{p\} \qquad D = \left\{ \frac{p : \neg q}{r}, \quad \frac{p : \neg r}{q}, \quad \frac{r : \neg s}{s} \right\}$$
- Exactly one extension: $S = Cn(p, q)$
- Default proofs w.r.t. S use:

$$p, \quad \frac{p}{q} \quad \text{and} \quad \frac{r}{s}$$

- Exactly the consequences of p and q can be derived!

Default logic — example

- $$W = \{p\} \qquad D = \left\{ \frac{p : \neg q}{r}, \quad \frac{p : \neg r}{q}, \quad \frac{r : \neg s}{s} \right\}$$
- $S = Cn(p, r, s)$ is NOT an extension
- Default proofs w.r.t. S use

$$p, \quad \text{and} \quad \frac{p}{r}$$

- No way to derive s!

Default logic — example

- Modeling inertia (change requires an action)

$$\frac{in\_location(X, L, T) : \neg abnormal(X, T)}{in\_location(X, L, T + 1)}$$

$$move(X, L1, L2, T) \Rightarrow in\_location(X, L2, T + 1)$$

$$move(X, L1, L2, T) \Rightarrow abnormal(X, T)$$

Default logic — example

- Modeling inertia (change requires an action)

$$\frac{in\_location(X, L, T) : \neg abnormal(X, T)}{in\_location(X, L, T + 1)}$$

$$move(X, L1, L2, T) \Rightarrow in\_location(X, L2, T + 1)$$

$$move(X, L1, L2, T) \Rightarrow abnormal(X, T)$$

$$destroyed(X, L, T) \Rightarrow abnormal(X, T)$$

Importance of default logic

- Generalizes Closed World Assumption
- Generalizes logic programming with stable model semantics
- Captures propositional circumscription
- Captures an important fragment of autoepistemic logic

Roots — declarative programming

- Algorithm = Logic + Control (Kowalski, 1979)
- First-order logic as a programming language
- Expectations, hopes
  - easy programming, fast prototyping
  - handle on program verification
  - advancement of software engineering

Horn logic programming

- $p(\vec{X}) \leftarrow q_1(\vec{X}), \ldots, q_k(\vec{X})$
- Herbrand models
- *single* intended Herbrand model
- least model

Computing with Horn programs

- A Horn program P *specifies* a subset X of the Herbrand universe for P, $HU(P)$, if for some predicate symbol p occurring in P we have:

$$X = \{t \in HU(P) : p(t) \in LM(P)\}.$$

- Finite Horn programs specify precisely the r.e. sets and relations

- Smullyan, 1968, Andreka and Nemeti, 1978

Computing with Horn programs

- Function symbols necessary!
- List constructor $[\cdot|\cdot]$ used to define higher-order objects
- Terms - basic data structures
- Unification - basic mechanism to manipulate data structures
- SLD-resolution
- Extensive use of recursion

Example

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

reverse([],[]).
reverse([X|Y],Z) :-
reverse(Y,U),append(U,[X],Z).
```

- both relations defined recursively
- terms represent complex objects: lists, sets, ...

Negation in the body

- Things get more complex!
- Clauses of the form:

$$p(\vec{X}) \leftarrow q_1(\vec{X}), \ldots, q_k(\vec{X}), \textbf{not}(r_1(\vec{X})), \ldots, \textbf{not}(r_l(\vec{X}))$$

- Why negation?
  - Natural linguistic concept
  - Facilitates declarative descriptions (definitions)

Semantics of programs with negation

- Usually — no least Herbrand model!
- Choosing a single *intended* model difficult
- Great logic programming schism
  - single intended model approach
  - multiple preferred model approach
- Nonmonotonicity — the roots converge!

Single intended model approach

- Extensions of Horn logic programming
  - Perfect semantics of *stratified* programs
  - 3-valued well-founded semantics for (arbitrary) programs
- Top-down computing based on unification and resolution
- XSB – Warren's group, SUNY Stony Brook

Were the expectations met?

- Logic programming (single intended model approach)
  - Requires significant logical background
  - Terms hard to think of as data structures
  - Extensive (often not intuitive) use of recursion
  - Unification difficult

Multiple preferred models

- Attractive alternative
- Semantics assigns to a program (theory) not one but many intended models!
  - for instance, all stable models
- How to interpret these semantics?
  - skeptical reasoning — resolution-based approaches too complex
  - preferred models represent different solutions to a problem

Stable model semantics

- Propositional case first!
- P — logic program
- M — set of atoms
- Reduct P^M (Gelfond, Lifschitz)
  - for each $a \in M$ remove rules with **not**(a) in the body
  - remove literals **not**(a) from all other rules

Stable model semantics

- M-applicable clauses
- Reduct P^M is a Horn program
- It has the least model $LM(P^M)$
- M is a stable model of P if

$$M = LM(P^M)$$

- Connection to default logic

$$p \leftarrow q, \textbf{not}(s) \qquad \mapsto \qquad \frac{q : \neg s}{p}$$

Stable models — example

- $p \leftarrow q, \textbf{not}(r)$ $\mapsto$ `p :- q, not r.`

```
p :- q, not s.
r :- p, not q , not s.
s :- not q.
q :- not s.
```

- Two stable models: M_1 ={p,q} and M_2 ={s}

Stable models — example

- The case of M_1 ={p,q}
- P_{M_1}:

```
p :- q.
q.
```

- $LM(P_{M_1})$ ={p,q}= M_1

Connection to DL — example

- $$W = \{p\} \qquad D = \left\{ \frac{p : \neg q}{r}, \quad \frac{p : \neg r}{q}, \quad \frac{r : \neg s}{s} \right\}$$

- Corresponding program:

```
p.
r :- p, not q.
q :- p, not r.
s :- r, not s.
```

- {p,q} is the only stable model

“Killing” clauses — constraints

- `p :- not p.`
- This program has NO stable models
- P — program; p — new atom
- Adding

  ```
  p :- not p.
  ```

  to P “kills” all stable models of P

“Killing” clauses — constraints

- Adding

```
p :- q, not r, not p.
```

  to P “kills” all stable models of P that:

  – contain q, and

  – do not contain r

- Abbreviation:

```
:- q, not r.
```

Programs with variables

- Herbrand models only!
- Program clause — a shorthand for all its ground substitutions

```
edge(1,2).
edge(1,3).
edge(2,4).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Programs with variables

- Clause `path(X,Y) :- edge(X,Y).` represents:

```
path(1,1) :- edge(1,1).
path(1,2) :- edge(1,2).
path(2,1) :- edge(2,1).
path(2,2) :- edge(2,2).
path(1,3) :- edge(1,3).
...
```

- Grounding

Programs with variables

- Grounding yields:
  - a propositional logic program
  - built of atoms from the Herbrand base of P, $HB(P)$
  - denoted $ground(P)$
- $M \subseteq HB(P)$ is a stable model of P if M is a stable model of $ground(P)$

Computing with stable semantics

- A logic program P defines:

$$SS(P) = \{M : M \ \textit{stable model of} \ P\}$$

- This is a *family of subsets* of $HB(P)$ (Herbrand base for P)
- $SS(P)$ may be "large" and complex
- Sets in $SS(P)$ are, in general, infinite and complex
- Restrict syntax — *eliminate function symbols*

Problem solving with SLP

- Encode the problem as a logic program
- So that stable models represent solutions
- Find one (or a specified) number of stable models
- Reconstruct solution(s)

SLP - how to program?

- Rudiments of methodology
  - Generate-and-test programming:
    - generate (possible structures)
    - weed out (unwanted ones)
    - by adding constraints ("killing" clauses)
  - Separate data from program (DATALOG$^{\neg}$)

Separating data from program

- Data:
  - a *finite* collection of ground atoms
  - extensional database (edb)
- Program:
  - a *finite* collection of clauses
  - intensional database (idb)

Problem-specific approach

- Requires deep understanding of the problem
- Exploits the global character of the semantics
- May result in more efficient programs

SLP — expressivity

- (Schlipf, 1994) Given an NP-problem $\mathcal{P}$, there are:
  - a finite program P, and
  - a representation scheme $edb_{\mathcal{P}}$ of problem instances

  so that for every instance I of $\mathcal{P}$:
  - $edb_{\mathcal{P}}(I)$ has size polynomial in $size(I)$
  - I has a solution if and only if $P \cup edb_{\mathcal{P}}(I)$ has a stable model.

Example — graph colorability

- Data (*edb*)
  - Ground atoms `vtx(a)` — define vertices
  - Ground atoms `edge(a,b)` — define edges
- Assignment of colors

```
clrd(X,r) :- vtx(X), not clrd(X,b),
                     not clrd(X,g).
clrd(X,b) :- vtx(X), not clrd(X,r),
                     not clrd(X,g).
clrd(X,g) :- vtx(X), not clrd(X,r),
                     not clrd(X,b).
```

Example — graph colorability

- Imposing constraints: colorability condition

  ```
  f :- clr(C), edge(X,Y), clrd(X,C), clrd(Y,C), not f.
  ```

- Colorability condition — differently

  ```
  :- clr(C), edge(X,Y), clrd(X,C), clrd(Y,C).
  ```

Answer-set programming —— ASP

- $\mathcal{L}$ —— formal language
  - e.g., language of predicate logic
- Formal system $\mathcal{F}$ in $\mathcal{L}$
  - syntax
  - semantics
- Semantics
  - $T \subseteq \mathcal{L} \mapsto Sem(T) \subseteq 2^{HB(T)}$
  - $HB(T)$ —— Herbrand base for T

ASP — examples

- Stable Logic Programming
- LP with other semantics
- Classical logic
- CSP
- Circumscription
- Disjunctive logic programming (answer sets)
- DATALOG with constraints
- Default logic

What is ASP good for?

- Search problems!
- Search problem Π:
  - a set of finite *instances*, D_Π
  - for each instance $I \in D_\Pi$:
    - a finite set $S_\Pi(I)$ of all *solutions* of Π

Solving search problem with ASP

- An algorithm A solves a search problem Π if:
  - for each $I \in D_\Pi$, A returns the answer "NO" when $S_\Pi(I) = \emptyset$, and
  - any solution $s \in S_\Pi(I)$, otherwise
- Decision problems can be cast as special search problems

Solving search problems with ASP

- For a search problem Π, we need:
  - an effective encoding of search problem instances as collections of ground atoms — edb_Π
  - a finite (instance independent) theory in $\mathcal{F}$ encoding the search problem — T_Π
  - so that solutions to Π for an instance I can be reconstructed from models of $edb_\Pi(I) \cup T_\Pi$

Solving search problems with ASP

- Formally, we require that:
  - $Sem(edb_\Pi(I) \cup T_\Pi) = \emptyset$ iff $S_\Pi(I) = \emptyset$, and
  - there is a polynomially computable *onto* function sol_Π that for every $I \in D_\Pi$ maps elements of $Sem(edb_\Pi(I) \cup T_\Pi)$ onto $S_\Pi(I)$

Solving search problems with ASP

- Π can be solved for an instance I by:
  - constructing the theory $edb_\Pi(I) \cup T_\Pi$
  - finding one of its models $s \in Sem(edb_\Pi(I) \cup T_\Pi)$
  - reconstructing from s a solution $sol_\Pi^I(s)$
- Uniform solution for Π
- Problem-specific approaches possible but harder

ASP - desiderata

- Expressive power
- Ease of modeling
  - intuitive semantics
  - concise encodings
  - modular programming
- Performance (speed)

How do they stack up?

- SLP vs PL vs DLP vs DC?
- No clear answer
- More experimental work needed
- Nonmonotonicity of the underlying system seems to be advantageous:
  - conciseness
  - modeling (knowledge representation)

French Phrases, Italian Soda

- The person who is planning a trip to Quebec, who drank either blueberry or lemon soda, didn't sit in seat number one

```
:- position_visits(1,quebec) .
soda_visits(blueberry,quebec) :-
               not soda_visits(lemon,quebec) .
soda_visits(lemon,quebec) :-
               not soda_visits(blueberry,quebec) .
```

French Phrases, Italian Soda

- Jeanne didn't sit next to the person who enjoyed the kiwi soda

```
:- position(Position1), position(Position2),
   person_position(jeanne,Position1),
   position_soda(Position2,kiwi),
   differ_position_1(Position1,Position2) .
```

French Phrases, Italian Soda

- Closure rules

```
person_soda(Person,Soda) :-
        person(Person), soda(Soda), visits(Visits),
        person_visits(Person,Visits),
        soda_visits(Soda,Visits) .
```

Overview — part 2

- Smodels— introduction
- Grounding — `lparse`
- Searching — `smodels`
- Using Smodels— examples:
  - propositional satisfiability, Hamiltonian cycles, blocks world planning
- Applications
- Conclusions

Implementing SLP

- Typically a two phase technique employed
  - Grounding:

    Given program P with variables, generate a set of ground instances of the rules (a subset of $ground(P)$) which preserves the stable models.
  - Stable models for ground programs:

    Davis-Putnam type backtracking search
- Used in: Smodels, `dlv`, dcs, ...

Smodels

- Normal logic programs with extensions
- Helsinki University of Technology
- Developers: Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen
- `http://www.tcs.hut.fi/Software/smodels/`
- Implemented in C++
- Two modules:
  - `lparse` (grounding)
  - `smodels` (stable models for ground programs)

Rules with variables

- Function symbols are not allowed (decidability)
- Built-in (e.g. arithmetic) functions supported
- A rule with variables is mapped to a set of ground rules by
  - evaluating first "easy" predicates (range/domain predicates for variables) and
  - generating for the remaining predicates ground instances compatible with the "easy" ones

Grounding

- Suppose that `clr(C)` and `edge(X,Y)` are "easy" and have been evaluated
- Then for a rule,

```
:- clr(C), edge(X,Y), clrd(X,C), clrd(Y,C).
```

grounding produces all ground rules

```
:- clr(r), edge(a,b), clrd(a,r), clrd(b,r).
```

for which `clr(r)` and `edge(a,b)` hold

Grounding — cont'd

- "Easy" predicates:
  - in Smodels: non-recursive (database views)
- Consider rules:

```
e2(X,Y) :- edge(X,Z), edge(Z,Y), not edge(X,Y).
d2(Y) :- e2(X,Y).
```

- If `edge(X,Y)` is an "easy" predicate, then so are `e2(X,Y)`, `d2(Y)`

Grounding — cont'd

- For safety: domain-restricted programs
- Each variable has a domain ("easy") predicate

```
clrd(X,r) :- vtx(X),
        not clrd(X,b), not clrd(X,g).
```

- Straightforward to incorporate built-in functions, e.g., arithmetic (no floundering problems)

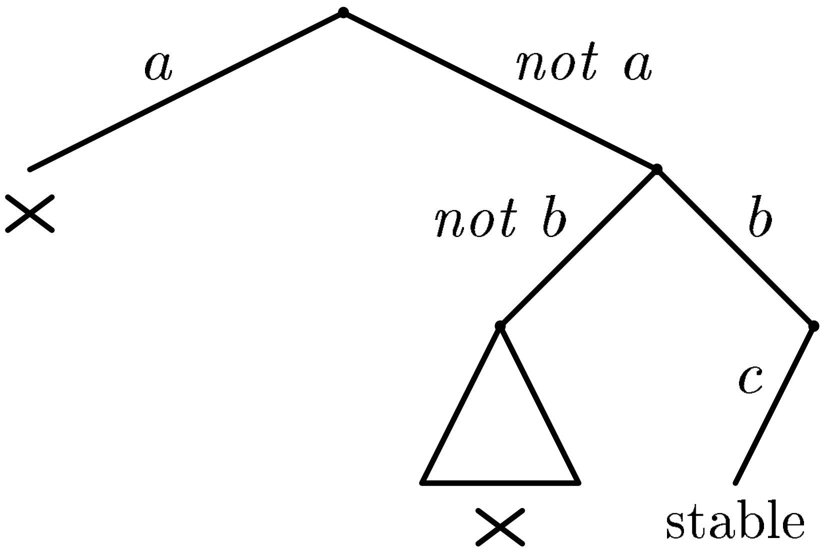
```
even(X+1) :- d(X), not even(X).
```

Grounding — cont'd

- For domain-restricted programs, grounding can be implemented locally and efficiently using database techniques
- Stronger notions of "easy" predicates possible
- In `dlv`:
  - range-restricted programs
  - "easy" predicates evaluated using deductive database techniques

Search in `smodels`

- Stable model computation:
  - Bottom-up backtracking search building a binary search tree
  - choice points on atoms:



Search in `smodels`

- To prune the search space:
  - choices are made one at a time (**heuristic**)
  - before making a new choice, the set of remaining choice points is pruned and conflicts are detected (**propagation rules**)
- Low level implementation using Dowling-Gallier type data structures

Propagation rules

- Similar to unit propagation:

  Given $q \leftarrow p_1, \mathbf{not}(p_2)$ and $\{p_1, \mathbf{not}(p_2)\}$ **derive** q

  Given $q \leftarrow p_1, \mathbf{not}(p_2)$ and $\{\mathbf{not}(q), p_1\}$ **derive** p_2

- In addition, some stronger rules are used:

  Given only one rule for q: $q \leftarrow p_1, \mathbf{not}(p_2)$ and $\{q\}$ **derive** $p_1, \mathbf{not}(p_2)$.

Propagation rules — cont'd

Rule	P	A	Derived
Fitting-1	$q \leftarrow p_1, \mathbf{not}(p_2)$	$p_1, \mathbf{not}(p_2)$	q
Fitting-2	$q \leftarrow p_1, \mathbf{not}(p_2)$ $q \leftarrow p_3, \mathbf{not}(p_4)$	$p_2, \mathbf{not}(p_3)$	$\mathbf{not}(q)$

Propagation rules — cont'd

Rule	P	A	Derived
BP-1	$q \leftarrow p_1, \mathbf{not}(p_2)$	q	$p_1, \mathbf{not}(p_2)$
BP-2	$q \leftarrow p_1, \mathbf{not}(p_2)$	$\mathbf{not}(q), p_1$	p_2
Upper Bound	$p_2 \leftarrow p_1, \mathbf{not}(p_3)$ $p_3 \leftarrow p_2, \mathbf{not}(p_1)$ $p_1 \leftarrow p_3, \mathbf{not}(p_2)$	$-$	$\mathbf{not}(p_1)$, $\mathbf{not}(p_2)$, $\mathbf{not}(p_3)$

Propagation rules — cont'd

- BP-1 and Upper bound have no counter part in SAT checkers
- Denote by **expand**(P, A) the closure of A under all the rules (Fitting-1 — Upper Bound)
- **expand**(P, A) generalizes the well-founded model
- Specifically: **expand**$(P, \emptyset)$ is the well-founded model of P

Propagation rules — cont'd

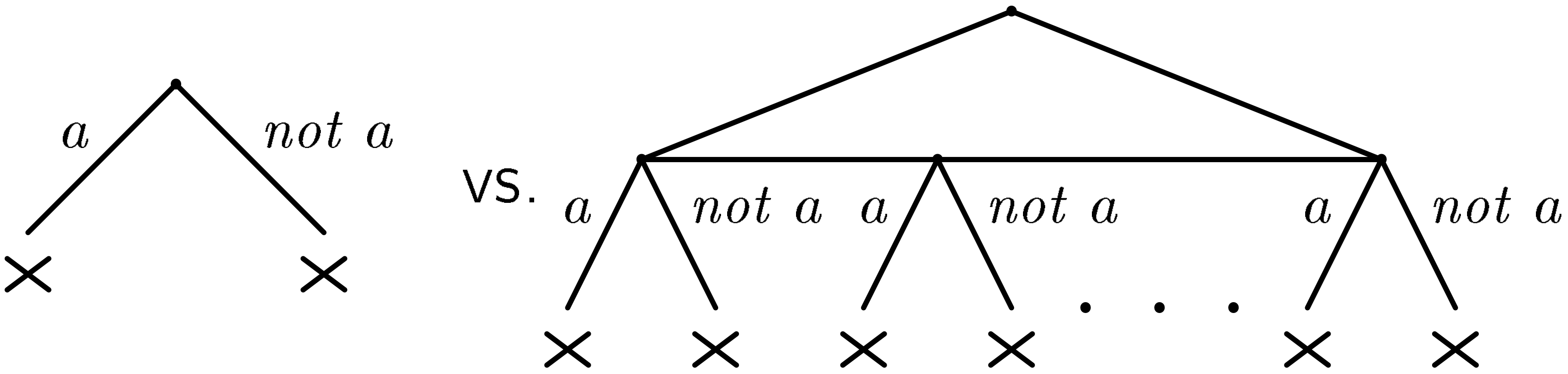
- **expand**(P, A) can be computed in quadratic time:
  - closure of (Fitting-1 – BP-2) in linear time
  - Upper Bound in linear time
- **expand** is the key procedure in Smodels
  (but e.g. `dlv` does not use Upper bound)

Lookahead

- Smodels uses an even stronger propagation rule
- **Lookahead** rule
  - Given program P and literals A:
    - if for a literal L, **expand**$(P, A \cup \{L\})$ contains a conflict $(p, \mathbf{not}(p))$
    - then derive the complement of L

Lookahead — cont'd

- Lookahead can yield a smaller search space:



Search heuristics

- Typical approaches to selecting choice points:
  - Maximum Occurrences in rules of Minimum size (MoM's) heuristics
  - Unit propagation counting
  - Preferring choice points with the smallest combined remaining search space (Smodels)

Heuristics in `smodels`

- Computed while doing lookahead
- For each undetermined atom p, two lookaheads:

  $A_p = \textbf{expand}(P, A \cup \{p\})$

  $A_{\mathbf{not}(p)} = \textbf{expand}(P, A \cup \{\mathbf{not}(p)\})$

Heuristics in `smodels` — cont'd

- If p is the next choice point, an upper bound on the remaining search space is

  $$E(p) = 2^{u} + 2^{u'}$$

  where u (u') is the number of undetermined atoms in A_p ($A_{\mathbf{not}(p)}$)

- Heuristic: choose an atom p with the smallest estimated remaining search space $E(p)$

Further extensions

- **Choices over subsets**
- `{ a1,..., an } :- b, not c.`
  - If the body holds, then any subset of `{a1,...,an}` (including empty) can be derived
- Choice rules

Choice rules — an example

- Rule `{ a } :- b.`
  - Corresponds to
    ```
    a :- not na, b.
    na :- not a.
    ```
  - One stable model: `{}`
- Program:
  ```
  { a } :- b.
  b.
  ```
  - Two stable models: {b}, {b,a}

Further extensions — cont'd

- **Cardinality constraints**
- $l \ \{\ p_1, \ldots, p_n\ \}\ u$
  - satisfied in a model S if the number of the literals from $\{p_1, \ldots, p_n\}$ satisfied in S is between integers l and u (inclusive)
- Example: vertex cover of size less than 12:

```
1 { vi,vj } 2 .  % for all edges (vi,vj)
:- 12 { v1,...,vn } .
```

Further extensions — cont'd

- **Weight constraints**
- $l \leq \{p_1 = w_1, \ldots p_n = w_n\} \leq u$
  - satisfied in S if the sum of the weights w_i of the literals from $\{p_1, \ldots, p_n\}$ satisfied in S is between integers l and u (inclusive)

Further extensions — cont'd

- Example: Knapsack with the sum of weights at most w but the sum of the values exceeding v

```
0 [ i1=w1,...,in=wn ] w.
:- [ i1=v1,...,in=vn ] v.
```

- In the ground case, complexity still in NP

Further extensions — cont'd

- **Optimization**: `minimize { p1=w1,...,pn=wn } .`
- Example: Cardinality minimal model

```
minimize  { a1=1,...,an=1 }.
```

Further Extensions — cont'd

- **Type definitions (conditions)**:

```
1 { clrd(V,C):clr(C) } 1 :- vtx(V).
```

- Two-phase implementation of the extensions:
  - Grounding (`lparse`) handles type definitions/conditions
  - Stable model computation for ground programs (`smodels`) extended to support directly cardinality and weight constraints.

Running Smodels

- Typical Unix command line:

```
UNIX> lparse [options] files | smodels [number]
```

- Some options:

```
-c const=n : define 'const' as number 'n'
-d : domain predicates to be output
-t : output plain text
number determines the number of stable
   models computed (0 indicates all)
```

Running Smodels — cont'd

```
UNIX> lparse demo.lp | smodels 0
smodels version 2.25. Reading...done
Answer: 1
Stable Model: r(a) p(a) s(a,b,c)
Answer: 2
Stable Model: r(a) q(a) s(a,b,c)
False
```

```
UNIX> lparse -d none -t demo.lp
r(a) :- p(a).
r(a) :- q(a).
q(a) :- not p(a).
p(a) :- not q(a).
```

demo.lp:

```
s(a,b,c).
s(X,Y,Z):- t(Y,X,Z).
p(X):- not q(X), s(X,Y,Z).
q(X):- not p(X), s(X,Y,Z).
r(X):- p(X), s(X,Y,Z).
r(X):- q(X), s(X,Y,Z).
```

Using answer-set programming

- A basic methodology: generate and test
  - **Generator**: provides candidate answer sets (typically encoded using odd loops/disjunctions/choice constructs)
  - **Tester**: eliminates non-valid candidates (typically encoded using killing clauses/integrity constraints)

Generate-and-test

- Integrity constraints
- `:- a1,..., an, not b1,..., not bm.`
- eliminate stable models but cannot introduce new ones:
  - Let P be a program and IC a set of integrity constraints
  - Then S is a stable model of $P \cup IC$ iff:
    * S is a stable model of P, and
    * S satisfies all ICs

Example: satisfiability

- **Generator**
  - Candidate answer sets: propositional models, i.e., sets of atoms
  - For each atom `ai`, rule `{ ai }.`
  - For program $\qquad$ 2^n stable models:

`{ a1 }.`	`{ }`
...	...
`{ an }.`	`{ a1,...,an }`

Satisfiability — cont'd

- **Tester** for $a1 \vee \cdots \vee an \vee \neg b1 \vee \cdots \vee \neg bm$:

```
:- not a1,..., not an, b1,..., bm.
```

- Clauses T / Program P_T / Stable model

Clauses T	Program P_T	Stable model
$a \vee \neg b$	`:- not a, b.`	`{ a }`
$\neg b \vee \neg a$	`:- a, b.`	
$b \vee a$	`:- not a, not b.`	
	`{ a }. { b }.`	

- Models of T and stable models of P_T correspond

Satisfiability — cont'd

- Consider general formulae (not in CNF):

  $(a \vee \neg b) \wedge (\neg a \leftrightarrow b)$

- Generator (for each atom `x`, rule `{ x }.`):

```
{ a }.

{ b }.
```

Satisfiability — cont'd

- Tester — evaluates a formula q recursively
- For each subformula:
  - the conditions under which it is true are given
  - false cases by default: it is false unless otherwise stated
- A satisfying truth assignment: a stable model satisfying

```
:- not q.
```

Satisfiability — cont'd

- Tester encoding

Subformula p	Rules
$l_1 \wedge \cdots \wedge l_n$	$p \leftarrow p_{l_1}, \ldots, p_{l_n}$
$l_1 \vee \cdots \vee l_n$	$p \leftarrow p_{l_1}$
	$\cdots$
	$p \leftarrow p_{l_n}$
$\neg l$	$p \leftarrow \mathbf{not}(p_l)$
$l_1 \leftrightarrow l_2$	$p \leftarrow p_{l_1}, p_{l_2}$
	$p \leftarrow \mathbf{not}(p_{l_1}), \mathbf{not}(p_{l_2})$

Satisfiability — cont'd

- Consider formula p_1: $\underbrace{(a \vee \neg b)}_{p_2} \wedge \underbrace{(\neg a \leftrightarrow b)}_{p_3}$

- Program:

```
{ a }.  { b }.
:- not p1.    p1:- p2, p3.
p2:- a.       p3:- a, not b.
p2:- not b.   p3:- not a, b.
```

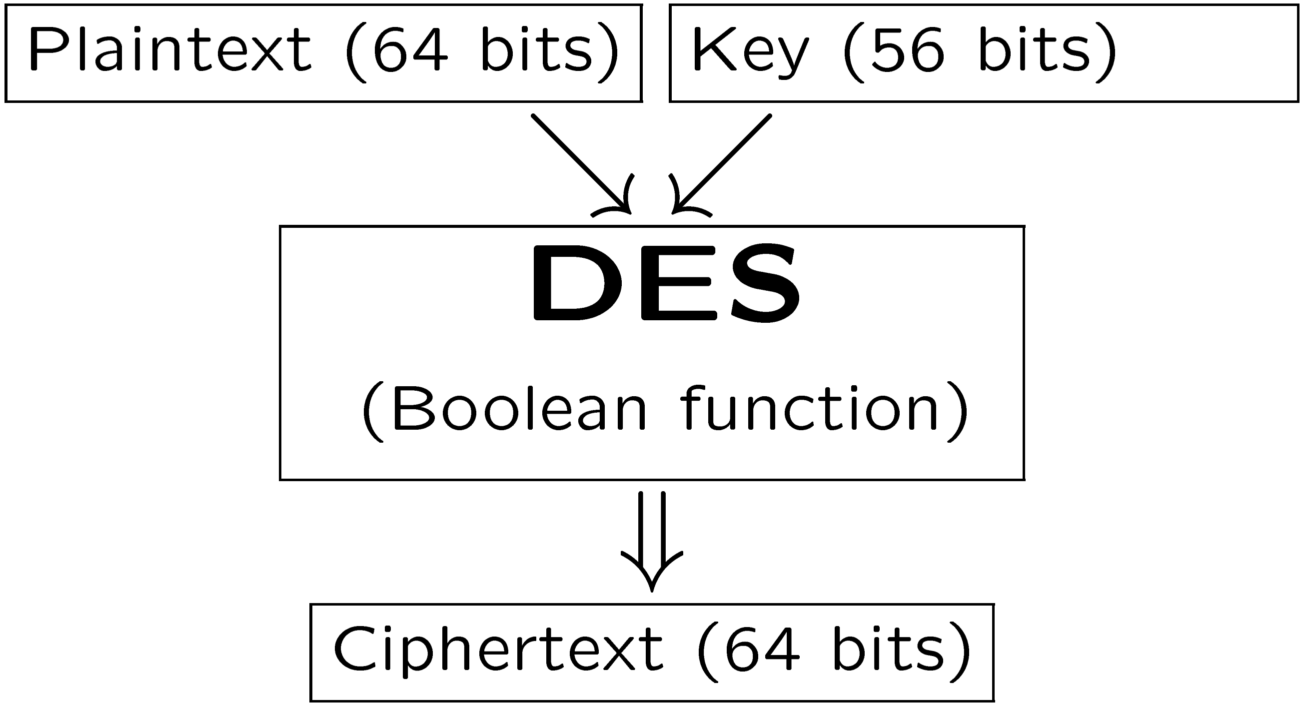
Stable model:

```
{a,p1,p2,p3}
```

- Satisfying truth assignments for p_1 and the stable models of the program correspond

Experimental results — DES

- DES (Boolean function) encoded directly as rules



DES — cont'd

- Experimenting with key search problems
- Known plaintext attack:
  given plaintext and ciphertext find a possible key
  for a limited version of DES (1–3 rounds)
- Smodels handles 3 rounds

DES — cont'd

- State-of-the art SAT-checkers (SATO/ntab/walksat) using an optimized encoding: 2 rounds
- A tuned SAT-checker (rel_sat based) using an optimized encoding: 3 rounds

Example — Hamiltonian cycles

- A Hamiltonian cycle: a *closed* path that visits all vertices of the graph exactly once
- Input: a graph
  - `vtx(a),...`
  - `edge(a,b),...`
  - `initialvtx(a0)`, for some vertex `a0`

Hamiltonian cycles — cont'd

- Candidate answer sets: subsets of edges
- Generator:

  ```
  { hc(X,Y) } :- edge(X,Y).
  ```

- Stable models of the generator given a graph:
  - input graph +
  - a subset of the ground facts `hc(a,b)` for which there is an input fact `edge(a,b)`.

Hamiltonian cycles — cont'd

- Tester (i):

  Each vertex has at most one chosen incoming edge and one outgoing edge

  ```
  :-hc(X,Y), hc(X,Z), edge(X,Y), edge(X,Z), Y!=Z.
  :-hc(Y,X), hc(Z,X), edge(Y,X), edge(Z,X), Y!=Z.
  ```

- Only subsets of chosen edges `hc(v,u)` forming paths (possibly closed) pass the test

Hamiltonian cycles — cont'd

- Tester (ii):

  Every vertex is reachable from a given initial vertex through chosen `hc(v,u)` edges:

  ```
  :- vtx(X), not r(X).
  r(Y) :- hc(X,Y), edge(X,Y), initialvtx(X).
  r(Y) :- hc(X,Y), edge(X,Y), r(X),
          not initialvtx(X).
  ```

- Only Hamiltonian cycles pass the tests (i–ii)

Hamiltonian cycles — cont'd

- Given:
  - the graph, the generator rule, and the tester rules (i–ii)

  Hamiltonian cycles and stable models correspond
- A cycle: facts `hc(v,u)` in a stable model
- Note that Hamiltonian cycles are challenging for SAT solvers as no compact SAT encodings (with linear number of atoms) are known

Hamiltonian cycles — cont'd

- Cardinality constraints enable an even more compact encoding
- Tester (i) using 2 variables:

```
:- 2 { hc(X,Y):edge(X,Y) }, vtx(X).
:- 2 { hc(X,Y):edge(X,Y) }, vtx(Y).
```

Example: planning

- Given:
  - a set of operators
  - initial situation and goal
- find a sequence of operator instances leading from initial to goal situation

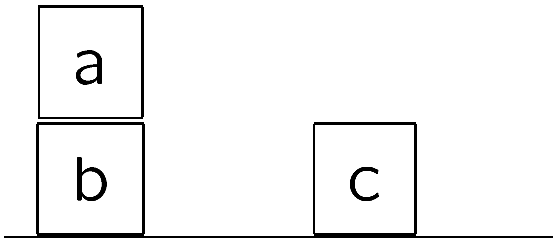
Planning — cont'd

- Planning is PSPACE-complete
- Planning with:
  - deterministic operators
  - complete knowledge about the initial situation, and with
  - an upper bound on the length of the plan

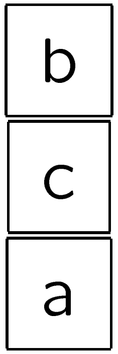
  is NP-complete

Block-world planning

```
(operator moveop
  (params (<X> OBJECT) (<Y> OBJECT) )
  (preconds  (clear <X>) (clear <Y>))
  (effects   (on <X> <Y>)  (clear <X>)))
```



initial situation



goal

solution:

moveop(a,table,0),

moveop(c,a,1),

moveop(b,c,2)

Mapping planning to rules

- Devise a logic program such that stable models correspond to plans:
  - of length at most n
  - that are valid
  - and that reach the goal

Mapping planning to rules

- Candidate answer sets: valid execution sequences (of length $\leq n$) of operator instances from the initial conditions
- Tester: eliminates those sequences that do not reach the goal

Planning — cont'd

- Preliminaries
  - Add to each predicate a situation argument
  - `on(X,Y,T)`: X is on Y in T
  - `moveop(X,Y,T)`: X is moved onto Y in T
  - Length bound n: `time(0..n).`
  - ```
    nextstate(Y,X) :- time(X), time(Y),
                Y = X + 1.
    ```
    ```

Planning — cont'd

- Available blocks: `block(a).`

  `block(b).`

  `block(c).`

- Initial conditions: `on(a,b,0).`

  `on(b,table,0).`

  `on(c,table,0).`

Planning — cont'd

- Auxiliary concepts make encoding easier
- Rules make it straightforward to define auxiliary predicates:

```
object(table).
object(X) :- block(X).
covered(X,T) :- block(Z), block(X), time(T),
        on(Z,X,T).
```

Planning — cont'd

- Further predicates:

```
on_something(X,T) :-
        block(X), object(Z), time(T),
        on(X,Z,T).
available(table,T) :- time(T).
available(X,T) :- block(X), time(T),
        on_something(X,T).
```

Planning — cont'd

- Generator: execution sequences of operators
- An operator can be applied if preconditions hold:

```
{ moveop(X,Y,T) }:-
        time(T), block(X), object(Y),
        X != Y, on_something(X,T),
        available(Y,T),
        not covered(X,T),
        not covered(Y,T).
```

Planning — cont'd

- Operator effects:

```
on(X,Y,T2) :- block(X), object(Y),
        nextstate(T2,T1),
        moveop(X,Y,T1).
```

Planning — cont'd

- Frame axioms (as rules with exceptions):

```
on(X,Y,T2) :- block(X), object(Y),
        nextstate(T2,T1),
        on(X,Y,T1),
        not moving(X,T1).
% the exceptions
moving(X,T) :-  time(T), block(X), object(Y),
        moveop(X,Y,T).
```

Planning — cont'd

- In addition rules for blocking conflicting operator instances are needed
- This set depends on how much concurrency in the search of a plan is allowed
- Computationally advantageous to allow concurrency to decrease search space explosion due to interleavings of independent operators

Planning — cont'd

- Blocking conditions for `moveop` (no concurrent actions):

```
:- 2 { moveop(X,Y,T):block(X):object(Y) },
      time(T).
```

Planning — cont'd

- Blocking conditions for `moveop` (with concurrent actions) I–II:

```
% A block cannot be moved to two destination
:- 2 { moveop(X,Y,T):object(Y) },
       block(X), time(T).
% The destination cannot be moving
:- block(X), object(Y), time(T),
        moveop(X,Y,T),
        moving(Y,T).
```

Planning — cont'd

- Blocking conditions for `moveop` (with concurrent actions) III:

```
% No two blocks moved onto the same block
:- 2 { moveop(X,Y,T):block(X) },
       block(Y), time(T).
```

Planning — cont'd

- Tester: excludes models where the goal has not been reached

```
:- not goal.
goal :- time(T), goal(T).
goal(T2) :- nextstate(T2,T1), goal(T1).
% Actual goal conditions
goal(T) :- time(T),
        on(b,c,T),
        on(c,a,T).
```

Planning — cont'd

- Plans correspond to stable models:
  - there is a stable model iff there is a valid sequence of moves that leads to goal and can be executed concurrently in at most n steps
- A valid plan
  - facts `moveop(x,y,t)` in a model ordered by the argument `t` where facts with the same `t` can be taken in any linear order.

Planning — cont'd

- Easy to add optimizations:

```
% Stop when the goal has been reached
:- block(X), object(Y), time(T),
        moveop(X,Y,T),
        goal(T).
```

Planning — cont'd

- Further optimizations (pruning rules):

```
% No move from table to table
:- block(X), time(T),
       moveop(X,table,T), on(X,table,T).

% No move on something and then to table
:- nextstate(T2,T1), block(X), object(Y),
        moveop(X,Y,T1), moveop(X,table,T2).
```

Planning — cont'd

- Encoding with card. constraints (Ext) is more compact and efficient than that using normal rules (Nor).

Blocks world problem	n	Ground rules Nor/Ext	Time (s)	Nor/Ext
large.c	8	81 681/19 311	first plan	9.2/2.3
(15 blocks)	7	72 527/17 087	no plan	1.4/0.4
large.d	9	127 999/27 359	first plan	21/4.4
(17 blocks)	8	115 109/24 533	no plan	3.1/0.6
large.e	10	191 621/37 379	first plan	40/6.7
(19 blocks)	9	174 099/33 879	no plan	5.0/0.8

Future work

- Improving implementation
- Development tools (debuggers, tracers)
- Integration with other systems and tools (SAT, CSP, Databases, programming languages)
- Programming methodology

Applications

- Planning

  Dimopoulos, Nebel, Koehler: Encoding Planning Problems in Nonmonotonic Logic Programs

  Lifschitz: Answer Set Planning

- Model checking

  Liu, Ramakrishnan, and Smolka: Fully Local and Efficient Evaluation of Alternating Fixed Points

Applications — cont'd

- Reachability analysis

  Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets

  See, the `mcsmodels` tool:

  `http://www.tcs.hut.fi/~kepa/tools/`

  Varpaaniemi: Stable Models for Stubborn Sets

Applications — cont'd

- Product configuration

  Soininen and Niemelä: Developing a declarative rule language for applications in product configuration

  Syrjänen: A Linux configurator

- Dynamic Constraint Satisfaction

  Soininen and I. Niemelä and E. Gelle: A Fixpoint Definition of Dynamic Constraint Satisfaction

Applications — cont'd

- Feature interaction
  Accorsi, Areces and Maarten: Towards Feature Interaction via Stable Models
- Logical cryptanalysis
  Hietalahti, Massacci, and Niemelä: DES: a Challenge Problem for Nonmonotonic Reasoning Systems

Applications — cont'd

- Network inhibition analysis

  Aura, Bishop, Sniegowski: Analyzing single-server network inhibition

- Computing (total and partial) stable models for disjunctive programs

  Janhunen, Niemelä, Simons, and You: Unfolding Partiality and Disjunctions in Stable Model Semantics

Other ASP systems

- `dlv` (disjunctive logic programs)
  `http://www.dbai.tuwien.ac.at/proj/dlv/`
- dcs (Datalog with constraints)
  `http://www.cs.engr.uky.edu/dcs/`
- LDL^{++} (Stratified programs with choice constructs and aggregates)
  `http://www.cs.ucla.edu/ldl/`
- CCALC (tight programs)
  `http://www.cs.utexas.edu/users/tag/cc/`

Other systems — dlv

- Disjunctive logic programming
- Technical University of Vienna
- Nicola Leone and his group
- `http://www.dbai.tuwien.ac.at/proj/dlv/`

Dlv

- Disjunctive clauses:
- Clauses of the form:

$$p_1 | \ldots | p_k \leftarrow q_1, \ldots, q_m, \textbf{not}(r_1), \ldots, \textbf{not}(r_n)$$

- Colorability:

```
clr(X,r) | clr(X,g) | clr(X,b) :- vtx(X).
```

Dcs

- DATALOG with constraints
- University of Kentucky
- Developers: Deborah East, Mirosław Truszczyński
- `http://www.cs.engr.uky.edu/dcs/`

Dcs

- Theories consist of
  - Constraints and post-constraints
  - Horn clauses (DATALOG)
- Semantics: M is a model if:
  - M is a model of constraints
  - the closure of M under Horn rules is a model of post-constraints

Conclusions

- ASP emerges as a viable KR tool
- Efficient implementations (Smodels, `dlv`, dcs)
- Expanding functionality and ease of use
- Growing range of applications
- Use it!