Answer-Set Programming: a Declarative Knowledge Representation Paradigm

Ilkka Niemelä and Mirek Truszczyński

ESSLLI 2001

Main topics

1. Answer-set programming (ASP) — an introduction
2. Stable logic programming (SLP)
3. *Smodels* — implementation of an SLP system
4. Logic PS+ and its implementation — assps
5. Applications
6. Final comments

ASP — an introduction

- Motivation — why do we need it?
- Roots — where does it come from?
  - KR
  - Satisfiability planning
  - Logic programming with negation
- ASP — what is it?
- ASP — properties

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 floor 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 floor

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

Wanted!

- A general-purpose approach for modeling and solving such problems
- Issues:
  - Diverse domains
  - Spatial and temporal reasoning
  - Constraints
  - Incomplete information
  - Frame problem
- Our proposal: Answer Set Programming (ASP) paradigm!

ASP — where does it come from?

- Knowledge Representation: logic-based formalisms
- Satisfiability planning
- Declarative programming: logic programming with negation

Roots — KR

- How to model:
  - agent's belief sets
  - commonsense reasoning
  - defeasible inferences
  - elaboration tolerance
  - the frame problem
- Approach: use a logic-based formalism

Roots — KR

- First-order logic and theorem proving methods
  - problems with modeling: FOL is monotone
  - computational complexity: FOL is undecidable
- Nonmonotonic logics: address the modeling issue
  - Default logic; Reiter 1980
  - Autoepistemic logic; Moore 1984
  - Circumscription; McCarthy 1980
  - Logic programming with stable model semantics; Gelfond-Lifschitz 1988

Default logic

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

$$\frac{\alpha : \beta_1, \ldots, \beta_k}{\gamma}$$

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

  then γ is known
- We will focus on propositional case here

Default theory

- (D, W)
  - D is a set of defaults
  - W is a set of propositional formulas
- Example

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

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
- Set: $\Gamma_\Delta(S) = \{\varphi : \vdash_{\Delta,S} \varphi\}$
- S is an *extension* of a default theory Δ if

$$S = \Gamma_\Delta(S)$$

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)

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

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

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

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

Default logic — properties

- Operator Γ_Δ is antimonotone
- Extensions of a default theory form an antichain (that is, no extension is contained in another one)
  - assume that S_1 and S_2 are extensions of $\Delta = (D, W)$ and that $S_1 \subseteq S_2$
  - it follows that

$$S_1 = \Gamma_\Delta(S_1) \supseteq \Gamma_\Delta(S_2) = S_2$$

  - hence, $S_1 = S_2$

Interpretation of default theories

- Extensions often thought as representations of different possible belief sets
- Reasoning based on computation of extensions and not on proof theory
  - no proof theory, in a common sense of the term, available
- A default theory thought of as an encoding of a *collection of sets* (possible belief sets)

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
- Changes the way logic is used in KR (extensions as opposed to proofs)

Satisfiability planning

- Given a *start* configuration, a *goal* configuration, and a set of actions, built a plan to change state from *start* to *goal*
- Common approach: a proof of the *goal* description from the theory describing the problem yields variable substitutions that define a plan
- Satisfiability planning (Kautz, Selman and collaborators, mid 90s):
  - Radical departure: represent a planning problem as a propositional theory so that *models* not proofs describe solutions

Satisfiability planning

- Exploit fast sat checkers (*satz*, *Walksat*, etc.)
- For programming use propositional schemata
- Fix the time horizon to ensure finiteness of theories and models
- *Satplan*, *BlackBox* — highly successful

Roots — declarative programming

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

Some logic terminology

- Language: *constant*, *variable*, *function* and *predicate* symbols
- *Terms*: strings built recursively from constant, variable and function symbols
- $c,\ X,\ f(c,X),\ f(f(c,X),f(X,f(X,c)))$
- *Herbrand universe*: collection of all terms
- *Ground terms*: no variable symbols

Horn logic programming

- *Atoms*: built of predicate symbols and terms
- $p(X,c,f(a,Y))$
- *Ground atoms*: atoms built of predicate symbols and ground term
- $p(a,c,f(a,a))$
- *Herbrand base*: collection of all ground atoms

Horn logic programming

- Horn clause:

$$p \leftarrow q_1, \ldots, q_k$$

  where p, q_i are atoms
- Clauses are *universally* quantified
- Interpretation of a clause: a shorthand for the set of its *ground instantiations*
  - ground instantiation: replace variables with ground terms - uniformly

Horn logic programming

- Horn program: a collection of Horn clauses
- Herbrand universe of P, $HU(P)$
- Herbrand base of P, $HB(P)$
- *grnd*(P)

Horn logic programming — semantics

- Herbrand models
- A subset M of the Herbrand base of a Horn program P is a *Herbrand model* of P if M is a model of *grnd*(P)
- Least Herbrand model
  - intersection of the set of Herbrand models of a Horn program is a Herbrand model of the program
  - Herbrand base is a Herbrand model
- *Single* intended Herbrand 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
- Queries (goals)
  - proofs provide answers
  - SLD-resolution
  - unification - basic mechanism to manipulate data structures
- Extensive use of recursion

Example

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

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

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

Example, cont'd

- Problem: reverse list `[a,b,c]`
  - Ask query `?- reverse([a,b,c],X).`
  - A proof of the query yields a substitution: `X=[c,b,a]`
  - The substitution constitutes an answer
- Query `?- reverse([a|X],[b,c,d,a])` returns `X=[d,c,b]`
- Query `?- reverse([a|X],[b,c,d,b])` returns no substitutions (there is no answer)

Example, cont'd

- **Techniques to search for proofs — the key**
- Understanding of the resolution mechanism is important
- It may make a difference which logically equivalent form is used:
  - `reverse([X|Y],Z) :- append(U,[X],Z), reverse(Y,U) .`
  - `reverse([X|Y],Z) :- reverse(Y,U), append(U,[X],Z) .`
  - termination vs. non-termination for query:
    `?- reverse([a|X],[b,c,d,b])`
- Is it truly declarative programming?

Negation in the body

- Why negation?
  - Natural linguistic concept
  - Facilitates declarative descriptions (definitions)
  - Needed for programmers convenience
- Clauses of the form:

$$p(\vec{X}) \leftarrow q_1(\vec{X_1}), \ldots, q_k(\vec{X_k}), \mathbf{not}(r_1(\vec{Y_1})), \ldots, \mathbf{not}(r_l(\vec{Y_l}))$$

- Things get more complex!

Semantics of programs with negation

- Usually — no least Herbrand model!
- Program

  $a \leftarrow \mathbf{not}(b)$
  $b \leftarrow \mathbf{not}(a)$

  has two minimal Herbrand models: $M_1 = \{a\}$ and $M_2 = \{b\}$.
- Choosing a single *intended* model difficult

Semantics of programs with negation

- Great Logic Programming Schism
  - single intended model approach
  - multiple preferred model approach

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
- Many of these concerns apply to (common) use of first-order logic in KR

Multiple preferred models

- Semantics assigns to a program (theory) not one but **many** intended models!
  - for instance, all stable models (to be introduced soon)
- How to interpret these semantics?
  - skeptical reasoning — resolution-based approaches too complex
  - preferred models represent different solutions to a problem
    - analogy with the way default theories are interpreted, and with the idea behind satisfiability planning!!!

Answer-set programming — ASP

- Fundamental concept: models not proofs represent solutions!
- Techniques to compute models (not proofs) are needed

Answer-set programming — ASP

- $\mathcal{L}$ — formal language (e.g. language of predicate logic)
- Formal system $\mathcal{F}$ in $\mathcal{L}$ with well-defined syntax and semantics
- Semantics:
  - $T \subseteq \mathcal{L} \mapsto Sem(T) \subseteq 2^{HB(T)}$
  - $HB(T)$ — Herbrand base for T
  - for a finite T, elements of $Sem(T)$ are finite!
- No function symbols in the language!

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 problems

- 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

- Given a search problem Π and an instance $I \in D_\Pi$, reduce it to the problem of computing models of theories in $\mathcal{F}$
  - *encode* (Π, I) as a finite theory $T(\Pi, I)$ in $\mathcal{F}$
  - so that solutions to Π for the instance I are represented by models of $T(\Pi, I)$
  - and can be reconstructed from the models

Solving search problems with ASP

- Mapping $T(\Pi, I)$ must be polynomial-time computable (in the size of I)
- $sem(T(\Pi, I)) = \emptyset$ iff $S_\Pi(I) = \emptyset$
- There is a polynomial-time computable *partial* function

$$sol_\Pi : \bigcup \{sem(T) : T \text{ is a theory in } \mathcal{F}\} \longrightarrow \bigcup \{S_\Pi(I) : I \in D_\Pi\}$$

such that

  - if $M \in sem(T(\Pi, I))$ then M is in the domain of sol_Π and $sol_\Pi(M) \in S_\Pi(I)$
  - all solutions in $S_\Pi(I)$ are of such form

Solving search problems with ASP

- Π can be solved for an instance I by:
  - constructing the theory $T(\Pi, I)$
  - finding one of the models $M \in sem(T(\Pi, I))$
  - reconstructing from M a solution $sol_\Pi^I(M)$
- if $sem(T(\Pi, I)) = \emptyset$, report no solutions

Solving search problems with ASP

- Propositional logic as an ASP
  - express an instance of a decision problems as an instance of the satisfiability problem
  - use a satisfiability checker
- All decision problems in NP can be solved in this way

Example — graph 3-colorability

- $G = (V, E)$, where
  - $V = \{v_1, \ldots, v_n\}$
  - $E \subseteq \{\{x, y\} : x \neq y,\ x, y \in V\}$
- Three colors: r, g, and b
- Propositional atoms
  - variables $v(x, c)$ for each combination of vertex x and color c
  - meaning: x has color c

Example, cont'd — clauses

- For every vertex x:

  $v(x, r) \vee v(x, g) \vee v(x, b)$
  $\neg v(x, r) \vee \neg v(x, g)$
  $\neg v(x, r) \vee \neg v(x, b)$
  $\neg v(x, g) \vee \neg v(x, b)$

- For every edge $\{x, y\}$:

  $\neg v(x, r) \vee \neg v(y, r)$
  $\neg v(x, g) \vee \neg v(y, g)$
  $\neg v(x, b) \vee \neg v(y, b)$

- Theory $T(G, r, g, b)$

Example, cont'd — key property

- $G = (V, E)$ is a graph
- If a function $f : V \longrightarrow \{r, g, b\}$ is a 3-coloring of G then

$$\{v(x, f(x)) : x \in V\}$$

  is a model of $T(G, r, g, b)$
- If M is a model of $T(G, r, g, b)$, then $M = \{v(x, f(x)) : x \in V\}$, where f is a 3-coloring of G

Uniform solutions

- $T(\Pi, I) = \mathit{data}_\Pi(I) \cup T_\Pi$, where
  - $\mathit{data}_\Pi(I)$ — an effective encoding of input instances to Π as collections of formulas in $\mathcal{F}$ (usually, ground atoms in $\mathcal{F}$); data
  - T_Π — a finite (instance independent) theory in $\mathcal{F}$ encoding the search problem; a program
- Separation of input data ($data_\Pi(I)$) from problem representation (T_Π)
- Π has a *uniform* solution in $\mathcal{F}$
- Variables needed!

Expressive power

- $\mathcal{F}$ — answer-set programming formalism
- $\exp(\mathcal{F})$ — the class of all search problems that have uniform solutions in $\mathcal{F}$
- Key theoretical issues: given $\mathcal{F}$, determine $\exp(\mathcal{F})$

The class NP-search

- A search problem Π is in the class *NP-search* if there is a nondeterministic Turing Machine *TM* such that:
  - *TM* runs in polynomial time, and
  - For every instance $I \in D_\Pi$, the set of strings left on the tape when accepting computations for I terminate is precisely the set of solutions $S_\Pi(I)$
- All decision problems that are in NP are included in the class NP-search
- Expressive power of many current ASP formalisms is precisely the class NP-search

ASP — examples

- Stable Logic Programming (SLP)
- Propositional Schemata with CWA (PS+)
- Disjunctive Logic Programming with answer-set semantics (DLP)
- Default logic
- Circumscription
- Most formal systems can be cast as ASP formalisms

ASP - desiderata

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

Stable Logic Programming

Overview

- Logic programming — syntax (propositional case)
- Semantics of stable models (propositional case)
- Properties
- Programs with variables
- Programming in SLP
- Expressive power

Logic programming — syntax

- Propositional case first!
- Propositional language over a set of atoms At
- Clause

$$a \leftarrow b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n)$$

  - a, b_i, c_j are atoms
  - a is the *head* of the clause
  - literals b_i, $\mathbf{not}(c_j)$ form the *body* of the rule

Stable model semantics

- 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
- Rules with no literal **not**(a) in the body such that $a \in M$ are called M*-applicable*

Stable model semantics

- 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

- Embedding

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

- One-to-one correspondence between stable models and extensions
- The operator GL_P given by

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

  corresponds to the operator Γ_Δ
- Stable models form an antichain

Stable models — example

- New notation, consistent with the syntax used by most implementations (in particular, *Smodels*)
- $p \leftarrow q, \mathbf{not}(r)$ $\mapsto$ `p :- q, not r.`

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

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

Stable models — example

- The case of $M_1 = \{\texttt{p}, \texttt{q}\}$
- P^{M_1}:

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

- $LM(P^{M_1}) = \{\texttt{p}, \texttt{q}\} = M_1$

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.
```

- $\{\texttt{p}, \texttt{q}\}$ is the only stable model

"Killing" clauses — constraints

- `p :- not p.`
- This program has NO stable models
- Let P be a program and p — a 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.
```

"Killing" clauses — proof

- $Q = P \cup \{p \leftarrow q, \mathbf{not}(r), \mathbf{not}(p)\}$
- M is a stable model of Q if and only if M is a stable model of P and it satisfies $\neg q \vee r$ (does not contain q or contains r)
- Assume that M is a stable model of Q
- If $p \in M$ then $p \leftarrow q, \mathbf{not}(r), \mathbf{not}(p)$ is M-applicable (contributes to the reduct Q^M)
- Contradiction! This is only possible if $p \notin M$
- Thus, $p \notin M$

"Killing" clauses — proof

- $Q = P \cup \{p \leftarrow q, \mathbf{not}(r), \mathbf{not}(p)\}$
- Assume that $r \in M$
- Then $Q^M = P^M$ and $M = LM(Q^M) = LM(P^M)$
- Thus, M is a stable model of P and it contains r

"Killing" clauses — proof

- $Q = P \cup \{p \leftarrow q, \mathbf{not}(r), \mathbf{not}(p)\}$
- Assume that $r \notin M$
- Then $Q^M = P^M \cup \{p \leftarrow q\}$
- Since $M = LM(Q^M)$ and $p \notin M$, $q \notin M$
- It follows that $LM(Q^M) = LM(P^M)$
- Thus, M is a stable model of P and $q \notin M$
- The converse implication can be proved in a similar fashion

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 *grnd*(P)
- $M \subseteq HB(P)$ is a stable model of P if M is a stable model of *grnd*(P)

Computing with stable semantics

- A logic program P defines:

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

- 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 ($data$)
- Program:
  - a *finite* collection of clauses
  - intensional database (idb)
- Uniform solution

SLP — expressivity

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

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

SLP — expressivity

- (Marek-Remmel, 2001) A search problem Π can be solved uniformly by a finite logic program if and only if P is in the class NP-search.

Example — graph colorability

- Data ($data$)
  - Ground atoms `vtx(v)` — define vertices
  - Ground atoms `edge(v,u)` — 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).
```


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) .
```

Smodels — an implementation of SLP

Overview

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

Implementing SLP

- Typically a two phase architecture employed
  - Grounding:
    Given program P with variables, generate a set of ground instances of the rules (a subset of $grnd(P)$) which preserves the stable models.
  - Stable models for ground programs:
    Davis-Putnam type backtracking search
- Used in: Smodels, `dlv`, aspps, ...

Smodels

- Computing stable models for normal logic programs with extensions
- Helsinki University of Technology
  Lab. for Theoretical Computer Science
- Developers: Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen
- Available under GLP:
  `http://www.tcs.hut.fi/Software/smodels/`
- Implemented in C++

Smodels—cont'd

- Two phase architecture:
  - `lparse` (grounding)
  - `smodels` (stable models for ground programs)

Smodels programs

- Smodels accepts **domain-restricted programs**
- In a program, predicates are partitioned (automatically) into
  - **domain predicates**
    (no choices/no recursion through negation)
  - **non-domain predicates**
    (all others)
- Rules domain-restricted:
  Each variable in a rule must appear in a positive domain predicate in the body of the rule.

Smodels programs—examples

- Domain predicates

```
% Facts
clr(b). clr(g). clr(r).
vtx(v). vtx(u). vtx(w).
edge(v,u). edge(u,w).

% Non-recursive predicates
e2(X,Y) :- edge(X,Z), edge(Z,Y), not edge(X,Y).

% Recursive predicates
tc(X,Y) :- edge(X,Y).
tc(X,Y) :- edge(X,Z), tc(Z,Y), vtx(Y).
```


Smodels programs—examples

- Domain-restricted rules

```
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).

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


Domain-restricted programs

- Function symbols can be used (with domain restriction)

```
p(s(X),X) :- d(X).
```

- Built-in (e.g. arithmetic) functions supported

```
d(0..n).
odd(X+1) :- d(X), X < n, not odd (X).
```

- For domain-restricted programs grounding done in two steps:
  - Evaluate domain predicates
  - Generate for each rule with variables a set of ground instances that are compatible with the evaluation of domain predicates

Grounding

- Rules defining domain predicates have a unique stable model M_{dom} which can be computed by deductive database techniques.
- M_{dom} is a subset of every stable model of the program.
- Grounding considers for each rule all its ground instances, i.e., every possible ground rule obtainable by replacing each variable in the rule (uniformly) by a ground term.
- Only those ground instances need to be taken where the domain predicates in the body hold (are in M_{dom}).
- Ground instances generated using database techniques (join evaluation).

Grounding — example

- Suppose that `vtx(V)` is a domain predicate and has been evaluated with the unique stable model M_{dom} containing

```
vtx(v1).
...
vtx(vn).
```

- Then for a rule,

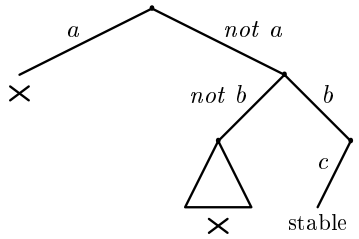
```
clrd(X,r) :- vtx(X), not clrd(X,b), not clrd(X,g).
```

grounding produces

```
clrd(v1,r) :- vtx(v1), not clrd(v1,b), not clrd(v1,g).
...
clrd(vn,r) :- vtx(vn), not clrd(vn,b), not clrd(vn,g).
```


Search in `smodels`

- Stable model computation:
  - 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**)
  - Each node in the search tree corresponds to a partial model, i.e., a set of literals derivable by the propagation rules from current choices
- Low level implementation using Dowling-Gallier type data structures

Propagation rules

- Given a partial model A (say with a new choice added), propagation rules extend A.
- Propagation rules do not eliminate stable models:
  If literal l is derivable by a propagation rule from A, then l holds in every stable model compatible with A.
- Examples:
  - Unit propagation:
    Given $q \leftarrow p_1, \mathbf{not}(p_2)$ and $\{p_1, \mathbf{not}(p_2)\} \subseteq A$ **derive** q
    Given $q \leftarrow p_1, \mathbf{not}(p_2)$ and $\{\mathbf{not}(q), p_1\} \subseteq A$ **derive** p_2
  - Also stronger rules used:
    Given only one rule for q: $q \leftarrow p_1, \mathbf{not}(p_2)$
    and $q \in A$ **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	$q \leftarrow p_1, \mathbf{not}(p_2)$	$p_2, \mathbf{not}(p_3)$	$\mathbf{not}(q)$
	$q \leftarrow p_3, \mathbf{not}(p_4)$		

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	$p_2 \leftarrow p_1, \mathbf{not}(q_1)$	$-$	$\mathbf{not}(p_1)$,
	$p_3 \leftarrow p_2, \mathbf{not}(q_2)$		$\mathbf{not}(p_2)$,
	$p_1 \leftarrow p_3, \mathbf{not}(q_3)$		$\mathbf{not}(p_3)$

Computing stable models

- Each branch of the search tree where propagation rules do not derive new literals and where there are no conflicts (both a and $\mathbf{not}(a)$ derived) provides a stable model (given by the positive atoms in the branch).

Program:

$p \leftarrow \mathbf{not}(q), \mathbf{not}(u)$

$q \leftarrow \mathbf{not}(p), \mathbf{not}(u)$

$p \leftarrow q$

Search tree:

$\mathbf{not}(u)$ (F-2)

↙ ↘

$\mathbf{not}(p)$	(Choice)	p	(Choice)
q	(F-1/BP-2)	$\mathbf{not}(q)$	(F-2)
p	(F-1)		
$\times$		stable model : $\{p\}$	

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

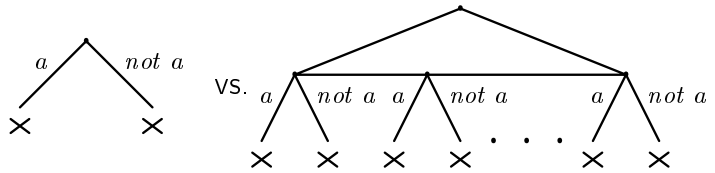
- **expand**(P, A) can be computed in quadratic time:
  - closure of (Fitting-1 – BP-2) in linear time
  - Upper Bound in linear time
- **expand** is the key procedure in Smodels (but e.g. `dlv` does not use Upper bound)

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

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 ($A_{\textbf{not}(p)}$)
- 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 ] .
maximize [ 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).
```

- **Domain declarations**:

```
#domain vtx(V), clr(C).
1 { clrd(V,CL):clr(CL) } 1.
:- edge(V,U), clrd(V,C), clrd(U,C).
```

- 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)
```

- Further information: `lparse` User's manual

Running Smodels — cont'd

```
UNIX> lparse demo.lp | smodels 0
smodels version 2.26. 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
```

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).
```

```
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).
```

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

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

    2^n stable models:

    ```
    { }
    ...
    { 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

  $a \vee \neg b$

  $\neg b \vee \neg a$

  $b \vee a$

  Program P_T

  ```
  :- not a, b.
  :- a, b.
  :- not a, not b.
  { a }. { b }.
  ```

  Stable model

  ```
  { a }
  ```

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

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.
```

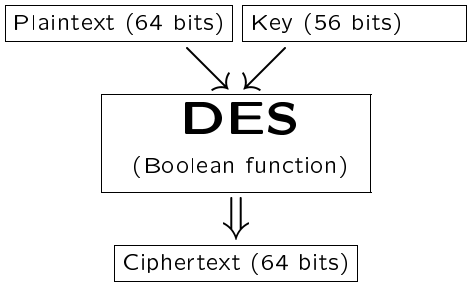
  Stable model:

```
{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

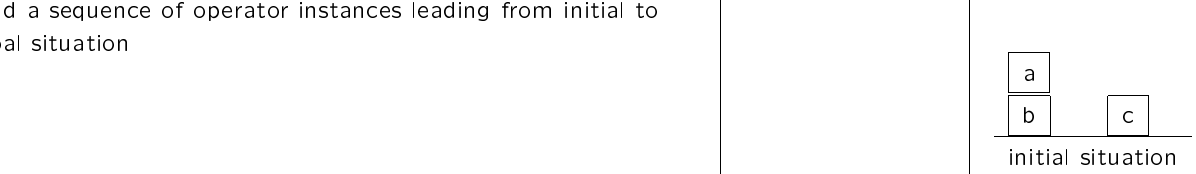
Planning — cont'd

- 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>)))
```



solution:
moveop(a,floor,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,floor,0).
on(c,floor,0).
```

Planning — cont'd

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

```
object(floor).
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(floor,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 floor to floor
:- block(X), time(T),
       moveop(X,floor,T), on(X,floor,T).

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

Logic PS+: an overview

- Syntax and semantics
- Implementation — system *aspps*
- Two examples:
  - 15-puzzle
  - Wire routing

Logic PS+ — syntax

- Language of predicate logic
- Constant, variable and predicate symbols
- Predicates of three types:
  - *data* predicates
  - *program* predicates
  - *predefined* predicates ($=$, $\neq$)
- *Term tuple*:
  - a tuple of constants, variables and underscores _
  - $t = (X, c, _, a, Y, Z)$

Atoms in PS+

- Special atoms **F** and **T** (**false** and **true**)
- An atom: $p(t)$
  - p is a predicate
  - t is a term tuple
- A *ground* atom: $p(t)$
  - t is a tuple term of *constant* symbols only

Atoms in PS+

- An *e-atom*, $p(t, _)$
  - p is a program predicate
  - q is a data predicate
  - t is a term tuple
  - more than one underscore symbol may appear
- Meaning:

$$\exists X p(t, X)$$

Atoms in PS+: examples

- $p(X, c)$
  - variables start with an upper-case letter
  - constants start with a lower-case letter
- $p(a, c)$: ground atom
- $p(X, _, a)$: e-atom

Clauses in PS+

- Logical connectives:
  - ',' stands for **and**
  - '|' stands or **or**
  - '$\rightarrow$' stands for **implication**
- Clause – expression of the form

$$A_1, \ldots, A_m \rightarrow B_1 | \ldots | B_n$$

  - A_i, B_j: atoms or e-atoms

Clauses in PS+: examples

- $\mathbf{T} \rightarrow \mathit{clrd}(X, _)$ (the same as $\mathit{clrd}(X, _)$)
- $\mathit{clrd}(X, C) \rightarrow \mathbf{F}$ (the same as $\neg \mathit{clrd}(X, C)$)
- $\mathit{clrd}(X, C) \rightarrow \mathit{vtx}(X)$
- $\mathit{edge}(X, Y), \mathit{clrd}(X, C), \mathit{clrd}(Y, D) \rightarrow C \neq D$

Theories in PS+

- *Input data*: a collection of *ground* atoms built of data predicates
- *Program*: a collection of clauses
- *Theory* in PS+: a pair (D, P)
  - D: input data
  - P: program

Semantics of PS+

- Key ideas:
  - A clause is assumed to be *universally* quantified
  - A clause is a shorthand for a collection of its ground (propositional) instantiations
  - Grounding converts a PS+ theory into a propositional theory
  - Grounding allows us to *lift* to PS+ the semantics of propositional logic
  - Caveat: program must not change input data!

Grounding of a PS+ theory (D, P)

- Data part D is already grounded
- For each clause in program P:
  - replace variables with constant symbols in theory
  - do it uniformly and in all possible ways
- *grnd*(D, P)

Grounding

- Some care needed for e-atoms
  - ground to disjunctions: $p(t, _)$ is interpreted as

    $$\exists X p(t, X)$$

    or

    $$p(t, c_1) | \ldots | p(t, c_k)$$

    where $c_1, \ldots, c_k$ are all the constants of the theory
  - handled differently depending on their location in a clause

Grounding

- a, b, c: all constants in the theory
- p and q are program predicates
- $q(X) \rightarrow p(X, _)$
- Grounding:

  $q(a) \rightarrow p(a, a) | p(a, b) | p(a, c)$
  $q(b) \rightarrow p(b, a) | p(b, b) | p(b, c)$
  $q(c) \rightarrow p(c, a) | p(c, b) | p(c, c)$

Grounding

- a, b: all constants in the theory; p, q: program predicates
- $p(X, _) \rightarrow q(X)$
- $\exists Y p(X, Y) \rightarrow q(X)$
- $p(X, Y) \rightarrow q(X)$

  $p(a, a) \rightarrow q(a)$
  $p(a, b) \rightarrow q(a)$
  $p(b, a) \rightarrow q(b)$
  $p(b, b) \rightarrow q(b)$
- **No need for e-atoms in the bodies of rules**

Grounding predefined predicates

- Evaluate atoms built from predefined predicates ($=$, $\neq$)
- When grounding, delete a *clause* if
  - the body contains an atom evaluated to **false**, or
  - the head contains an atom evaluated to **true**
- When grounding, delete an *atom* from a clause if
  - the atom evaluates to **true** and appears in the body of the clause, or
  - the atom evaluates to **false** and appears in the head of the clause

Semantics of PS+

- A set of ground atoms M is a *model* of (D, P) or *answer set* for (D, P) if
  - M is a model of $grnd(D, P)$
  - for a ground atom p, built of a data predicate, $p \in M$ if and only if $p \in D$
- **Programs do not recompute (change) input data**

Example: graph coloring

- Given: graph $G = (V, E)$ and a set of colors $C = \{1, \ldots, k\}$
- Compute a proper vertex coloring of G with colors in C
- Data (a set of ground atoms):

$$D_{gcl} = \{vtx(v) : v \in V\} \cup \{edge(v, w) : \{v, w\} \in E\} \cup \{color(c) : c \in C\}$$

Example: graph coloring

- Program (a collection of clauses), P_{gcl}:
  - $clrd(X, C) \rightarrow vtx(X)$
  - $clrd(X, C) \rightarrow color(C)$
  - $vtx(X) \rightarrow clrd(X, _)$
  - $clrd(X, C), clrd(X, D) \rightarrow C = D$
  - $edge(X, Y), clrd(X, C), clrd(Y, C) \rightarrow \mathbf{F}$
- First two rules: typing
- Models of the PS+ theory (D_{gcl}, P_{gcl}) determine proper colorings and each proper coloring corresponds to a model

Expressive power

- Finite PS+ programs capture precisely the whole class NP-search
- The same expressive power as that of SLP

Implementation

- *aspps*: **a**nswer **s**et **p**rogramming with **p**ropositional **s**chemata
  - grounder: *grnd*
  - solver: *aspps*
  - available at: `http://www.cs.uky.edu/ai/aspps/`
- Key differences
  - typing
  - predefined function symbols and more predefined predicates
  - restricted-domain e-atoms
  - Cardinality atoms

Typing

- Some data predicates define *data types*
  - *clrd*$(X, C) \rightarrow$ *vtx*(X)
  - ensures that the domain of X is restricted to the extension of the predicate *vtx* as specified in D
  - In *aspps*, X is declared to be of type *vtx* and the clause is not needed!
  - `var vtx X.`

Typing

- Arguments of program predicates are restricted to values of appropriate types
  - *clrd* is declared in the program file to be a binary predicate with arguments of type *vtx* and *color*
  - `pred clrd ( vtx , color ).`
- Simplifies programming and grounding

Predefined functions and predicates

- Arithmetic operators
- Support restricted-domain arithmetic
  - apply to variables of types that are subsets of integers and to integer constants
  - are evaluated first at grounding
  - if the result is out of range, the atom is evaluated to **false**
- More predefined predicates (arithmetic ordering): $\geq$, $>$, $\leq$, $<$

Restricted-domain e-atoms

- $p(X, Y) : d(Y)$
  - p is a program predicate
  - d is a data predicate
  - Meaning

    $$p(X, c_1) | \ldots | p(X, c_n),$$

    where $c_1, \ldots, c_n$ form the extension of d, as given by D
- Related to e-atom $p(X, _)$
  - restricted domain of constants that define the disjunction (define the scope of the existential quantifier)

Cardinality atoms

- $m\{p(t) : d_1(t_1) : \ldots : d_k(t_k)\}n$
  - m and n are non-negative integers
  - p is a program predicate
  - d_i are data predicates or predefined predicates (conditions)

Cardinality atoms

- $m\{p(t) : d_1(t_1) : \ldots : d_k(t_k)\}n$
- The expression $\{p(t) : d_1(t_1) : \ldots : d_k(t_k)\}$ defines a list of ground atoms:
  - all those ground atoms $p(t^g)$, such that for every i, the projection of t^g onto appropriate arguments satisfies d_i
    - according to D, if d is a data predicate
    - or, if d_i is a predefined predicate, according to the standard interpretation of d_i
- Meaning: at least m and no more than n atoms on the list are true

Running aspps

- `grnd -c t=12 -d data -r program`
  - Produces a ground theory in the file: `12_data_program.tdc`
  - option `-c` allows the user to specify constants from the command line
- `aspps -f 12_data_program.tdc`
  - adds one line summarizing the result (sat or unsat, running time, etc) to the file `aspps.stat`

15-puzzle problem

- 4×4 grid filled with 16 movable tiles labeled $0, 1, \ldots, 15$
- Goal: rearrange the tiles so that the tiles are arranged from 0 to 15 in the *row-major* order
- In a single step 0 can be swapped with one of its direct neighbors
- Hard search problem
- An example of a planning problem

15-puzzle: data file

```
% 12 moves required
%   1  5  2  7
%   4  0  3 11
%   8  9  6 15
%  12 13 10 14

   time[0..t].     pos[1..4].     entry[0..15].

   in0(1,1,1).   in0(1,2,5).  in0(1,3,2).  in0(1,4,7).
   in0(2,1,4).   in0(2,2,0).  in0(2,3,3).  in0(2,4,11).
   in0(3,1,8).   in0(3,2,9).  in0(3,3,6).  in0(3,4,15).
   in0(4,1,12).  in0(4,2,13). in0(4,3,10). in0(4,4,14).
```


15-puzzle: program file

```
% Declare program predicates

   pred in(time,pos,pos,entry).
   pred move(time,pos,pos).
   pred goal(time).

% Declare variables

   var time T.
   var pos X,Y,X1,Y1.
   var entry A.
```

15-puzzle: program file

```
% for each time T, define an assignment of numbers
% 0, 1, ..., 15 to 16 locations in the array

   1{in(T,X,Y,A):entry(A)}1.
   1{in(T,X,Y,A):pos(X):pos(Y)}1.

% At time T=0, computed assignment must coincide with
% the initial assignment

   in(0,X,Y,A) -> in0(X,Y,A).
```


15-puzzle: program file

```
% For each time T<t, select exactly one location neighboring
% with 0 for the swap

   T<t -> 1{move(T,X,Y): pos(X):pos(Y)}1.
   in(T,X,Y,0), move(T,X1,Y1) -> abs(X-X1)+abs(Y-Y1)==1.

% Do not undo the last move

   in(T,X,Y,0), in(T+2,X,Y,0) -> .
```


15-puzzle: program file

```
% Assignment at time T+1, must correspond to the assignment at
% time T modulo swapped values

   T<t, in(T+1,X,Y,A) -> in(T,X,Y,A) |
                         in(T,X,Y,0) |
                         move(T,X,Y).

   T<t, in(T,X,Y,0), move(T,X1,Y1) -> in(T+1,X1,Y1,0).
```


15-puzzle: program file

```
% Specify the goal configuration
%  0  1  2  3
%  4  5  6  7
%  8  9 10 11
% 12 13 14 15

   in(t,1,1,0).   in(t,1,2,1).   in(t,1,3,2).   in(t,1,4,3).
   in(t,2,1,4).   in(t,2,2,5).   in(t,2,3,6).   in(t,2,4,7).
   in(t,3,1,8).   in(t,3,2,9).   in(t,3,3,10).  in(t,3,4,11).
   in(t,4,1,12).  in(t,4,2,13).  in(t,4,3,14).  in(t,4,4,15).
```

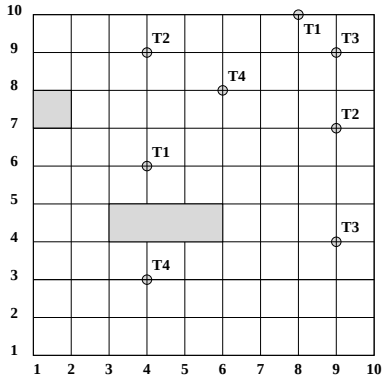
Performance

Distance	first plan	optimality
12	1.5	4.9
13	10.2	8.7
14	4.9	18.8
15	95.4	54.6
16	79.0	90.9
17	367.1	205.3
18	468.1	557.4

Wire routing

- Given pairs of terminals on a grid and a set of obstacles connect terminals by wires so that wires:
  - do not intersect
  - do not enter obstacles
  - do not share grid edges

chip10.1



Representing data

```
wire(1).  wire(2).  wire(3).  wire(4).

crd(1).  crd(2).  crd(3).  crd(4).  crd(5).
crd(6).  crd(7).  crd(8).  crd(9).  crd(10).

terminal(6,4,1).  terminal(10,8,1).
terminal(7,9,2).  terminal(9,4,2).
terminal(4,9,3).  terminal(9,9,3).
terminal(8,6,4).  terminal(3,4,4).

blocked(4,3).  blocked(4,4).  blocked(4,5).  blocked(4,6).
blocked(5,3).  blocked(5,4).  blocked(5,5).  blocked(5,6).
blocked(7,1).  blocked(7,2).  blocked(8,1).  blocked(8,2).
```

Modeling problem constraints

```
% Declare program predicate path: point (I,J) is on wire W

  pred path(crd , crd , wire).

% Declare variables

  var crd I, J, K, L, M, N.
  var wire W.

% Terminal points are on the path

  terminal(I,J,W) -> path(I,J,W).
```


Modeling problem constraints

```
% Terminal points have exactly one adjacent point on the path

  terminal(I,J,W) -> 1 { path(L,M,W):
                         crd(L):
                         crd(M):
                         abs(I-L)+abs(J-M)==1 } 1.
```

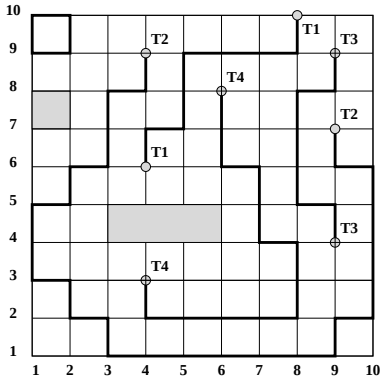

Modeling problem constraints

```
% All other points on the path have exactly two adjacent points

   path(I,J,W) -> terminal(I,J,W) |
                  2 { path(L,M,W):
                      crd(L):
                      crd(M):
                      abs(I-L)+abs(J-M)==1 } 2
```


Modeling problem constraints

```
% Only one wire can pass through any point

   { path(I,J,W): wire(W) } 1.

% Blocked points are unavailable

   blocked(I,J), path(I,J,W) -> .
```

Independent cycles, wandering paths



Controlling search space

```
% At most m points on each wire

   { path(I,J,W):crd(I):crd(J) } m.

% Do not choose points at distance m or more from
% a terminal point

   terminal(I,J,W), path(L,M,W), m < abs(I-L)+abs(J-M) -> .
```

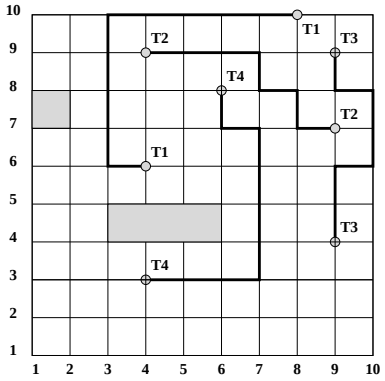

Controlling search space

```
% Eliminate cycles of length 4

   path(I,J,W),
   path(I+1,J,W),
   path(I+1,J+1,W),
   path(I,J+1,W) -> .
```


Controlling search space

```
% Limit search space to rectangle formed by terminal points
% (or its relaxation - by a constant k)

   terminal(I,J,W), terminal(M,N,W), path(K,L,W), K < I-k -> .
   terminal(I,J,W), terminal(M,N,W), path(K,L,W), M+k < K -> .
   terminal(I,J,W), terminal(M,N,W), path(K,L,W), L < J-k -> .
   terminal(I,J,W), terminal(M,N,W), path(K,L,W), N+k < L -> .
```

Rectangle w/o 4 cycles



Applications: an overview

- Planning
- Product configuration
- Computer aided verification
- Wire routing, feature interaction, security, ...

Applications — planning

- Planning has been studied widely, see e.g.

  Dimopoulos, Nebel, Koehler: Encoding planning problems in nonmonotonic logic programs

  Lifschitz: Answer set planning

  Son, Baral, McIlraith: Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs.

- USAdvisor project at Texas Tech:

  A decision support system for the flight controllers of space shuttles

Applications — product configuration

- Modelling configurable products

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

  Soininen, Niemelä, Tiihonen, Sulonen: Representing configuration knowledge with weight constraint rules.

- Software configuration

  Syrjänen: A rule-based formal model for software configuration.

  Syrjänen: A Linux configurator

- WeCoTin project (`http://www.soberit.hut.fi/WeCoTin/`).

  Web Configuration Technology

Applications — CAV

- Fixed point engine for model checking
  Liu, Ramakrishnan, and Smolka: Fully Local and Efficient Evaluation of Alternating Fixed Points
- Reachability analysis
  Varpaaniemi: Stable Models for Stubborn Sets
  See the `prod` tool (`http://www.tcs.hut.fi/Software/prod/`)
  Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. See `mcsmodels`: `http://www.tcs.hut.fi/~kepa/tools/`
- Bounded model checking
  Heljanko and Niemelä: Bounded LTL model checking with stable models.

Applications — cont'd

- Wire routing
  Erdem, Lifschitz, Wong: Wire routing and satisfiability planning
  East, Truszczyński: More on wire routing with ASP
- Dynamic Constraint Satisfaction
  Soininen and I. Niemelä and E. Gelle: A fixpoint definition of dynamic constraint satisfaction
- Network management
  Son, Lobo: Reasoning about policies using logic programs.
- Feature interaction
  Accorsi, Areces and Maarten: Towards Feature Interaction via Stable Models

Applications — cont'd

- Logical cryptanalysis
  Hietalahti, Massacci, and Niemelä: DES: a Challenge Problem for Nonmonotonic Reasoning Systems
- Security protocol analysis
  Aiello,Massacci: An executable specification language for planning attacks to security protocols.
- Network inhibition analysis
  Aura, Bishop, Sniegowski: Analyzing single-server network inhibition

Applications — cont'd

- Diagnosis
  Gelfond, Galloway: Diagnosing dynamic systems in A-Prolog
- Computing (total and partial) stable models for disjunctive programs
  Janhunen, Niemelä, Simons, You: Unfolding Partiality and Disjunctions in Stable Model Semantics

Other ASP systems

- `dlv` (disjunctive logic programs)
  `http://www.dbai.tuwien.ac.at/proj/dlv/`
- 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 LP-based 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:

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

- Colorability:

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


Conclusions

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