ANSWER SET PROGRAMMING

Tran Cao Son

Department of Computer Science

New Mexico State University

Las Cruces, NM 88011, USA

tson@cs.nmsu.edu

HTTP://WWW.CS.NMSU.EDU/~TSON

October 2005

Acknowledgement

This tutorial contains some materials from tutorials on answer set programming and on knowledge representation and logic programming from those provided by

- Chitta Baral, available at `www.public.asu.edu/~cbaral`.
- Michael Gelfond, available at `www.cs.ttu.ued/~mgelfond`.

Introduction — Answer Set Programming

Answer set programming is a new programming paradigm. It is introduced in the late 90's and manages to attracts the intention of different groups of researchers thanks to its:

- *declarativeness*: programs do not specify how answers are computed;
- *modularity*: programs can be developed incrementally;
- *expressiveness*: answer set programming can be used to solve problems in high complexity classes (e.g. Σ_P^2, $\Pi_2 P$, etc.)

Answer set programming has been applied in several areas: reasoning about actions and changes, planning, configuration, wire routing, phylogenetic inference, semantic web, information integration, etc.

Purpose

- Introduce answer set programming
- Provide you with some initial references, just in case
- . . . you get excited about answer set programming

Outline

- Foundation of answer set programming: logic programming with answer set semantics (syntax, semantics, early application).
- Answer set programming: general ideas and examples
- Application of answer set programming in
  - Knowledge representation
  - Constraint satisfaction problem
  - Combinatoric problems
  - Reasoning about action and change
  - Planning and diagnostic reasoning
- Current issues

LOGIC PROGRAMMING AND ANSWER SET SEMANTICS

Terminologies – many brorrowed from classical logic

- variables: X, Y, Z, etc.
- object constants (or simply constants): a, b, c, etc.
- function symbols: f, g, h, etc.
- predicate symbols: p, q, etc.
- terms: variables, constants, and $f(t_1, \ldots, t_n)$ such that $t_i s$ are terms.
- atoms: $p(t_1, \ldots, t_n)$ such that $t_i s$ are terms.
- literals: atoms or an atom preceded by $\neg$.
- naf-literals: atoms or an atom preceded by **not**.
- gen-literals: literals or a literal preceded by **not**.
- ground terms (atoms, literals) : terms (atoms, literals resp.) without variables.

First Order Language, Herbrand Universe, and Herbrand Base

- $\mathcal{L}$ – a first order language with its usual components (e.g., variables, constants, function symbols, predicate symbols, arity of functions and predicates, etc.)
- $U_{\mathcal{L}}$ – Herbrand Universe of a language $\mathcal{L}$: the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.
- $B_{\mathcal{L}}$ – Herbrand Base of a language $\mathcal{L}$: the set of all ground atoms which can be formed with the functions, constants and predicates in $\mathcal{L}$.
- Example:
  - Consider a language $\mathcal{L}_1$ with variables X, Y; constants a, b; function symbol f of arity 1; and predicate symbol p of arity 1.
  - $U_{\mathcal{L}_1} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.
  - $B_{\mathcal{L}_1} = \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))),$
    $p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$.

Logic Programs – Syntax

A logic program rule is of the form

$$a_0 \leftarrow a_1, \ldots, a_n, \mathbf{not}\ a_{n+1}, \ldots, \mathbf{not}\ a_{n+k} \quad (1)$$

where a's are atoms (of a first order language $\mathcal{L}$); **not** a is called a naf-atom.

- a_0 – head (the left hand side)
- $a_1, \ldots, a_n$ – body (the right hand side)

Definition 1 *A logic program is a set of logic programming rules.*

The language $\mathcal{L}$ of a program Π is often given *implicitly*.

Remark 1 *From now on, we will say a rule (resp. a program) instead of a logic programming rule (resp. a logic program), for short;*

Special cases:

- $a_0 \leftarrow$ – $(n = 0)$ is called a *fact*;
- $\leftarrow a_1, \ldots, a_n$ – (a_0 is missing) is called a *constraint* or a *goal*;
- $k = 0$ – the rule is called a positive rule.

Main Definitions

- $ground(r, \mathcal{L})$: the set of all rules obtained from r by all possible substitution of elements of $U_{\mathcal{L}}$ for the variables in r.
- **Example 1** *Consider the rule "*$p(f(X)) \leftarrow p(X).$*" and the language* $\mathcal{L}_1$*. Then* $ground(r, \mathcal{L}_1)$ *will consist of the following rules:*

  $p(f(a)) \leftarrow p(a).$
  $p(f(b)) \leftarrow p(b).$
  $p(f(f(a))) \leftarrow p(f(a)).$
  $p(f(f(b))) \leftarrow p(f(b)).$
  $\vdots$
- For a program Π:
  - $ground(\Pi, \mathcal{L}) = \cup_{r \in \Pi} ground(r, \mathcal{L})$
  - $\mathcal{L}_\Pi$: The language of a program Π is the language consists of the constants, variables, function and predicate symbols (with their corresponding arities) occurring in Π. In addition, it contains a constant a if no constant occurs in Π.
  - $ground(\Pi) = \cup_{r \in \Pi} ground(r, \mathcal{L}_\Pi)$.

Example 2 $-\ \Pi$*:*

$p(a).$
$p(b).$
$p(c).$
$p(f(X)) \leftarrow p(X).$

$-\ ground(\Pi)$*:*

$p(a) \leftarrow.$
$p(b) \leftarrow\ .$
$p(c) \leftarrow\ .$
$p(f(a)) \leftarrow p(a).$
$p(f(b)) \leftarrow p(b).$
$p(f(c)) \leftarrow p(c).$
$p(f(f(a))) \leftarrow p(f(a)).$
$p(f(f(b))) \leftarrow p(f(b)).$
$p(f(f(c))) \leftarrow p(f(c)).$
$\vdots$

Intuition and Examples of Using LP (Positive Programs)

- Meaning of a rule
  - A positive rule "$a_0 \leftarrow a_1, \ldots, a_n$" can be viewed as the disjunction $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_n$ which says that if $a_1, \ldots, a_n$ are true then a_0 is true.
  - A rule "$a_0 \leftarrow a_1, \ldots, a_n, \mathbf{not}\ a_{n+1}, \ldots, \mathbf{not}\ a_{n+k}$" states that if $a_1, \ldots, a_n$ are true and none of the $a_{n+1}, \ldots, a_{n+k}$ can be proven to be true then a_0 is true.
  - A constraint "$\leftarrow a_1, \ldots, a_n, \mathbf{not}\ a_{n+1}, \ldots, \mathbf{not}\ a_{n+k}$" holds if some a_i $(1 \leq i \leq n)$ is false or some a_j $(n+1 \leq j \leq n+k)$ is true.
- Knowledge representation using LP-rules

$flies(X) \leftarrow bird(X).$	"if X is a bird, then X flies"
$wet\_grass \leftarrow sprinkler\_on.$	"The grass is wet if the sprinkler is on"
$rain \leftarrow humid, hot.$	"hot and humid causes rain"
$love(X,Y) \leftarrow love(Y,X).$	"Love is a symmetric relationship"

Herbrand Interpretation

Definition 2 *The Herbrand universe (resp. Herbrand base) of* Π*, denoted by* U_Π *(resp.* B_Π*), is the Herbrand universe (resp. Herbrand base) of* $\mathcal{L}_\Pi$*.*

Example 3 *For*

$$\Pi = \{p(X) \leftarrow q(f(X), g(X)). \quad r(Y) \leftarrow\}$$

the language of Π *consists of*

- *two function symbols:* f *(arity 1) and* g *(arity 2);*
- $p, q,$ *and* r *are predicate symbols;*
- *variables* X, Y*; and*
- *a (added) constant* a*.*

$U_\Pi = U_{\mathcal{L}_\Pi} = \{a, f(a), g(a), f(f(a)), g(f(a)), g(f(a)), g(g(a)), f(f(f(a))), \ldots\}$
$B_\Pi = B_{\mathcal{L}_\Pi} = \{p(a), q(a, a), r(a), p(f(a)), q(a, f(a)), r(f(a)), \ldots\}$

Definition 3 (Herbrand Interpretation) *A Herbrand interpretation of a program* Π *is a set of atoms from its Herbrand base.*

Semantics – Positive Programs without Constraints

Let Π be a positive proram and I be a Herbrand interpretation of Π. I is called a Herbrand model of Π if for every rule "$a_0 \leftarrow a_1, \ldots, a_n$," if $a_1, \ldots, a_n$ are true with respect to I (or $a_1, \ldots, a_n \subseteq I$) then a_0 is also true with respect to I.

Definition 4 *The least Herbrand model for a program* Π *is called the* minimal model *of* Π *and is denoted by* M_Π.

Computing M_P. Let Π be a program. We define a fixpoint operator T_Π that maps a set of atoms (of program Π) to another set of atoms as follows.

$$\begin{aligned} T_\Pi(X) = \{a \mid\ & a \in B_\Pi, \\ & \text{there exists a rule} \\ & a \leftarrow a_1, \ldots, a_n \text{in } \Pi \text{ s. t. } a_i \in X\} \end{aligned} \tag{2}$$

Note: By $a \leftarrow a_1, \ldots, a_n$ in Π we mean there exists a rule $b \leftarrow b_1, \ldots, b_n$ in Π (that might contain variables) and a ground substitution σ such that $a = b\sigma$ and $a_i = b_i\sigma$.

Remark 2 *The operator* T_Π *is often called the van Emden and Kowalski's iteration operator.*

Some Examples

For $\Pi = \{p(f(X)) \leftarrow p(X). \quad q(a) \leftarrow p(X).\}$
we have

$$U_\Pi = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$$

and

$$B_\Pi = \{q(a), p(a), p(f(a)), p(f(f(a))), \ldots\}$$

Computing $T_\Pi(X)$:

- For $X = B_\Pi$, $T_\Pi(X) = \{q(a)\} \cup \{p(f(t)) \mid t \in U_\Pi\}$.
- For $X = \emptyset$, $T_\Pi(X) = \emptyset$.
- For $X = \{p(a)\}$, $T_\Pi(X) = \{q(a), p(f(a))\}$.
- We have that $M_\Pi = \emptyset$ (Why?).

Properties of T_Π

- T_Π is monotonic: $T_\Pi(X) \subseteq T_\Pi(Y)$ if $X \subseteq Y$.
- T_Π has a least fixpoint that can be computed as follows.
  1. Let $X_1 = T_\Pi(\emptyset)$ and $k = 1$
  2. Compute $X_{k+1} = T_\Pi(X_k)$. If $X_{k+1} = X_k$ then stops and return X_k.
  3. Otherwise, increase k and repeat the second step.

Note: The above algorithm will terminate for positive program Π with finite B_Π. We denote the least fix point of T_Π with $T_\Pi^\infty(\emptyset)$ or $lfp(T_\Pi)$.

Theorem 1 $M_\Pi = lfp(T_\Pi)$.

Theorem 2 *For every positive program* Π *without constraint,* M_Π *is unique.*

More Examples

- For $\Pi_1 = \{p(X) \leftarrow q(f(X), g(X)). \quad r(Y) \leftarrow\}$

  we have that

  $U_{\Pi_1} = \{a, f(a), g(a), f(f(a)), g(f(a)), g(f(a)), g(g(a)), f(f(f(a))), \ldots\}$

  $B_{\Pi_1} = \{p(a), q(a,a), r(a), p(f(a)), q(a, f(a)), r(f(a)), \ldots\}$

  Computing M_Π:

  $X_0 = T_{\Pi_1}(\emptyset) = \{r(a), r(f(a)), r(g(a)), r(f(f(a))), ...\}$

  $X_1 = T_{\Pi_1}(X_0) = X_0$

  So, $lfp(\Pi_1) = \{r(a), r(f(a)), r(g(a)), r(f(f(a))), ...\}$.

- For $\Pi_2 = \{p(f(X)) \leftarrow p(X). \quad q(a) \leftarrow p(X).\}$

  $U_{\Pi_2} = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$

  $B_{\Pi_2} = \{q(a), p(a), p(f(a)), p(f(f(a))), \ldots\}$

  Computing M_{Π_2}:

  $X_0 = T_{\Pi_2}(\emptyset) = \emptyset$

  $X_1 = T_{\Pi_2}(X_0) = X_0$

  So, $lfp(\Pi_2) = \emptyset$.

- For $\Pi_3 = \{p(f(X)) \leftarrow p(X). \quad q(a) \leftarrow p(X). \quad p(b).\}$

  $U_{\Pi_3} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), \ldots\}$

  $B_{\Pi_3} = \{q(a), q(b), p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), \ldots\}$

  Computing M_{Π_3}:

  $X_0 = T_{\Pi_3}(\emptyset) = \{p(b)\}$
  $X_1 = T_{\Pi_3}(X_0) = \{p(b), q(a), p(f(b))\}$
  $X_2 = T_{\Pi_3}(X_1) = \{p(b), q(a), p(f(b)), p(f(f(b)))\}$
  $\ldots$

  So, $lfp(T_{\Pi_3}) = \{q(a)\} \cup \{p(f^i(b)) \mid i = 0, 1, \ldots\}$.

- For $\Pi_4 = \{p \leftarrow a. \quad q \leftarrow b. \quad a \leftarrow .\}$, $M_{\Pi_4} = \{a, p\}$.

- For $\Pi_5 = \{p \leftarrow p.\}$, $M_{\Pi_5} = \emptyset$

- For $\Pi_6 = \{p \leftarrow p. \quad q \leftarrow .\}$, $M_{\Pi_6} = \{q\}$.

- For $\Pi_7 = \{p(b). \quad p(c). \quad p(f(X)) \leftarrow p(X).\}$,$M_{\Pi_7} = \{p(f^n(b)) \mid n = 0, \ldots, \} \cup \{p(f^n(c)) \mid n = 0, 1\ldots, \}$

Entailment

For a program Π and an atom a, Π *entails* a (with respect to the minimal model semantics), denoted by $\Pi \models a$, iff $a \in M_\Pi$.
We say that Π entails $\neg a$ (with respect to the minimal model semantics), denoted by $\Pi \models \neg a$, iff $a \in M_\Pi$.

Example 4 *Let*

$$\Pi = \begin{cases} p(f(X)) \leftarrow p(X). \\ q(a) \leftarrow p(X). \\ p(b). \end{cases}$$

We have that $M_\Pi = lfp(T_\Pi) = \{q(a)\} \cup \{p(f^i(b)) \mid i = 0, 1, \ldots\}$
where $f^i(b) = f(f(\ldots(f(b))))$ *(*f *repeated* i *times)*
So, we say:

$\Pi \models q(a)$,

$\Pi \models \neg q(b)$, *and*

$\Pi \models p(f^i(b))$ *for* $i = 0, 1, \ldots$

Entailment – Another Example

Example 5 • *Consider the parent-child database with facts of the form* $p(X, Y)$ *(*X *is a parent of* Y*). We can define the ancestor relationship,* $a(X, Y)$ *(*X *is an ancestor of* Y*), using the following rules*

$$\Pi_a = \begin{cases} a(X,Y) \leftarrow p(X,Y). \\ a(X,Y) \leftarrow p(X,Z), a(Z,Y). \end{cases}$$

Given the set of facts $I = \{p(a,b), p(b,c), p(c,d)\}$*, let* $\Pi = \Pi_a \cup I$*. We can easily compute*

$$M_\Pi = I \cup \{a(X,Y) \mid p(X,Y) \in I\} \cup \{a(a,c), a(a,d), a(b,d)\}$$

So, $\Pi \models a(a,c)$ *and* $\Pi \models a(a,d)$*, i.e.,* a *is an ancestor of* c *and* d*; on the other hand,* $\Pi \models \neg a(d,a)$*, i.e.,* d *is not an ancestor of* a*.*

- *Consider a directed graph G described by a set of atoms of the form $edge(X, Y)$. The following program can be used to determine whether there is a path connecting two nodes of G.*

$$\Pi_G = \begin{cases} reachable(X,Y) \leftarrow edge(X,Y) \\ reachable(X,Y) \leftarrow edge(X,Z), reachable(Z,Y) \\ \\ edge(a,b) \leftarrow \\ edge(b,c) \leftarrow \\ edge(c,a) \leftarrow \\ \ldots \end{cases}$$

It can be shown that for every pair of nodes p and q of the graph G, $reachable(p, q)$ belongs to M_{Π_G} **iff** *there exists a path from p to q in the graph G.*

Remark 3 *Reasoning using positive programs assumes the closed world assumption (CWA): anything, that cannot be proven to be true, is false.*

Semantics – General Logic Programs without Constraints

Recall that a program is a collection of rules of the form

$$a \leftarrow a_1, \ldots, a_n, \mathbf{not}\ a_{n+1}, \mathbf{not}\ a_{n+k}.$$

Let Π be a program and X be a set of atoms, by Π^X we denote the program obtained from $ground(\Pi)$ by

1. Deleting from $ground(\Pi)$ any rule $a \leftarrow a_1, \ldots, a_n, \mathbf{not}\ a_{n+1}, \mathbf{not}\ a_{n+k}$ for that $\{a_{n+1}, \ldots, a_{n+k}\} \cap X \neq \emptyset$, i.e., the body of the rule contains a naf-atom $\mathbf{not}\ a_l$ and a_l belongs to X; and
2. Removing all of the naf-atoms from the remaining rules.

Remark 4 *The above transformation is often referred to as the Gelfond-Lifschitz transformation.*

Remark 5 Π^X *is a positive program.*

Definition 5 *A set of atoms X is called an* answer set *of a program Π if X is the minimal model of the program Π^X.*

Theorem 3 *For every positive program Π, the minimal model of Π, M_Π, is also the unique answer set of Π.*

Detailed Computation

- Consider $\Pi_2 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$. We will show that its has two answer sets $\{a\}$ and $\{b\}$

$S_1 = \emptyset$	$S_2 = \{a\}$	$S_3 = \{b\}$	$S_4 = \{a, b\}$
$\Pi_2^{S_1}$: $a \leftarrow$ $b \leftarrow$	$\Pi_2^{S_2}$: $a \leftarrow$	$\Pi_2^{S_3}$: $b \leftarrow$	$\Pi_2^{S_4}$:
$M_{\Pi_2^{S_1}} = \{a, b\}$	$M_{\Pi_2^{S_2}} = \{a\}$	$M_{\Pi_2^{S_3}} = \{b\}$	$M_{P^{S_4}} = \emptyset$
$M_{\Pi_2^{S_1}} \neq S_1$	$M_{\Pi_2^{S_2}} = S_2$	$M_{\Pi_2^{S_3}} = S_3$	$M_{\Pi_2^{S_4}} \neq S_4$
NO	YES	YES	NO

- Assume that our language contains two object constants a and b and consider $\Pi = \{p(X) \leftarrow \textbf{not } q(X). \quad q(a) \leftarrow\}$. We show that $S = \{q(a), p(b)\}$ is an answer set of Π. We have that $\Pi^S = \{p(b) \leftarrow \quad q(a) \leftarrow\}$ whose minimal model is exactly S. So, S is an answer set of Π.

- $\Pi_4 = \{p \leftarrow \textbf{not } p.\}$ We will show that P does not have an answer set.

  $S_1 = \emptyset$, then $\Pi_4^{S_1} = \{p \leftarrow\}$ whose minimal model is $\{p\}$. $\{p\} \neq \emptyset$ implies that S_1 is not an answer set of Π_4.

  $S_2 = \{p\}$, then $\Pi_4^{S_2} = \emptyset$ whose minimal model is $\emptyset$. $\{p\} \neq \emptyset$ implies that S_2 is not an answer set of Π_4.

  This shows that P does not have an answer set.

In computing answer sets, the following theorem is useful:

Theorem 4 *Let Π be a program.*

1. *Let r be a rule in $ground(P)$ whose body contains an atom a that does not occur in the head of any rule in $ground(P)$. Then, S is an answer set of Π iff S is an answer set of $ground(P) \setminus \{r\}$.*
2. *Let r be a rule in $ground(P)$ whose body contains a naf-atom* **not** *a that does not occur in the head of any rule in $ground(P)$. Let r' be the rule obtained from r by removing* **not** *a. Then, S is an answer set of P iff S is an answer set of $ground(P) \setminus \{r\} \cup \{r'\}$.*

Examples about Answer Sets

Remark 6 *A program may have zero, one, or more than one answer sets.*

- $\Pi_1 = \{a \leftarrow \textbf{not } b.\}$.

  Π_1 has a unique answer set $\{a\}$.

- $\Pi_2 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$.

  The program has two answer sets: $\{a\}$ and $\{b\}$.

- $\Pi_3 = \{p \leftarrow a. \quad a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$

  The program has two answer sets: $\{a, p\}$ and $\{b\}$.

- $\Pi_4 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } c. \quad d \leftarrow .\}$

  Answer sets: $\{d, b\}$.

- $\Pi_5 = \{p \leftarrow \textbf{not } p.\}$

  No answer set.

- $\Pi_6 = \{p \leftarrow \textbf{not } p, d. \quad r \leftarrow \textbf{not } d. \quad d \leftarrow \textbf{not } r.\}$

  Answer set $\{r\}$.

Entailment w.r.t. Answer Set Semantics

- For a program Π and an atom a, Π *entails* a, denoted by $\Pi \models a$, if $a \in S$ for every answer set S of Π.
- For a program Π and an atom a, Π *entails* $\neg a$, denoted by $\Pi \models \neg a$, if $a \notin S$ for every answer set S of Π.
- If neither $\Pi \models a$ nor $\Pi \models \neg a$, then we say that a is *unknown* with respect to Π.

Remark 7 Π *does not entail* a *DOES NOT IMPLY that* Π *entails* $\neg a$*, i.e., reasoning using answer set semantics does not employ the closed world assumption.*

Example 6

- $\Pi_1 = \{a \leftarrow \textbf{not } b.\}$.

  Π_1 *has a unique answer set* $\{a\}$. $\Pi_1 \models a$, $\Pi_1 \models \neg b$.

- $\Pi_2 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$.

  The program has two answer sets: $\{a\}$ *and* $\{b\}$*. Both* a *and* b *are unknown w.r.t.* Π_2.

- $\Pi_3 = \{p \leftarrow a. \quad a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$

  The program has two answer sets: $\{a, p\}$ *and* $\{b\}$*. Everything is unknown.*

- $\Pi_3 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } c. \quad d \leftarrow .\}$

  Answer sets: $\{d, b\}$. $\Pi_3 \models b$; $\Pi_3 \models \neg a$; *etc.*

- $\Pi_4 = \{p \leftarrow \textbf{not } p.\}$

  No answer set. p *is unknown.*

- $\Pi_5 = \{p \leftarrow \textbf{not } p, d. \quad r \leftarrow \textbf{not } d. \quad d \leftarrow \textbf{not } r.\}$

  Answer set $\{r\}$. $\Pi_5 \models r$; $\Pi_5 \models \neg p$; *etc.*

- $\Pi_6 = \{p \leftarrow \textbf{not } a. \quad p \leftarrow \textbf{not } b. \quad a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$

  Two answer sets: $\{p, a\}$ *and* $\{p, b\}$. *So,* $\Pi_6 \models p$ *but* $\Pi_6 \not\models a$ *and* $\Pi_6 \not\models \neg a$? *(likewise* b*).*

- $\Pi_7 = \{q \leftarrow \textbf{not } r. \quad r \leftarrow \textbf{not } q. \quad p \leftarrow \textbf{not } p. \quad p \leftarrow \textbf{not } r.\}$

  One stable model: $\{p, q\}$. *So,* $\Pi_7 \models p$ *and* $\Pi_7 \models q$.

Further intuitions behind the semantics

- A set of atoms S is **closed under** a program Π if for all rules of the form

  $a_0 \leftarrow a_1, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$

  in Π, $\{a_1, \ldots, a_m\} \subseteq S$ and $\{a_{m+1}, \ldots, a_n\} \cap S = \emptyset$ implies that $a_0 \in S$.
- A set of atoms S is said to be **supported by** Π if for all $p \in S$ there is a rule of the form $p \leftarrow a_1, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$

  in Π, such that $\{a_1, \ldots, a_m\} \subseteq S$ and $\{a_{m+1}, \ldots, a_n\} \cap S = \emptyset$.
- A set of atoms S is an answer set of a program Π **iff** (i) S is closed under Π and (ii) there exists a level mapping function λ (that maps atoms in S to a number) such that for each $p \in S$ there is a rule in Π of the form $p \leftarrow a_1, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$ such that $\{a_1, \ldots, a_m\} \subseteq S$, $\{a_{m+1}, \ldots, a_n\} \cap S = \emptyset$ and $\lambda(p) > \lambda(a_i)$, for $1 \leq i \leq m$.
- Note that (ii) above implies that S is supported by Π.

LOGIC PROGRAMMING AND KNOWLEDGE REPRESENTATION

Commonsense Reasoning

Our knowledge

- is often *incomplete* (it does not contain complete information about the world), and
- contains *defaults* (rules which have exceptions, also called normative sentences).
- contains *preferences* between defaults (prefer a conclusion/default).

For this reasons, we often jump to conclusions (ignore what we do not now), and know to deal with exceptions and preferences.

Example 7 *We know*

Normally, birds fly.
Normally, computer science students can program.
Normally, students work hard.
Normally, things do not change.
Normally, students do not watch TV.
Normally, the speed limit on highways is 70 mph.
Normally, it is cold in December.

From this, we make conclusions such as Tweety *flies if we know that* Tweey is a bird*;* Monica *can program if* she is a computer science student*; etc.*

Representing Defaults

Normally, a's are b's. $b(X) \leftarrow a(X), \textbf{not } ab(r, X)$[1]

Normally, birds fly. $flies(X) \leftarrow bird(X), \textbf{not } ab(bird\_fly, X)$

Normally, animals have four legs.

$numberoflegs(X, 4) \leftarrow animal(X), \textbf{not } ab(animal, X)$

Normally, fishs swim. $swim(X) \leftarrow fish(X), \textbf{not } ab(fish, X)$

Normally, computer science students can program.

$can\_program(X) \leftarrow student(X), is\_in(X, cs), \textbf{not } ab_p(X)$

Normally, students work hard.

$hard\_working(X) \leftarrow student(X), \textbf{not } ab(X)$

Typically, classes start at 8am.

$start\_time(X, 8am) \leftarrow class(X), \textbf{not } ab(X)$

[1] r is the 'name' of the statement; $ab(X)$ or $ab_r(X)$ can also be used.

Example 8 (Birds) *Suppose that we know*

- r_1*: Normally, birds fly.*
- r_3*: Penguins are birds.*
- r_3*: Penguins do not fly.*
- r_4*: Tweety is a penguin.*
- r_5*: Tim is a bird.*

$$\Pi_b = \begin{cases} r_1: & flies(X) & \leftarrow & bird(X), \textbf{not } ab(r_1, X) \\ r_2: & bird(X) & \leftarrow & penguin(X) \\ r_3: & ab(r_1, X) & \leftarrow & penguin(X) \\ r_4: & penguin(tweety) & \leftarrow & \\ r_5: & bird(tim) & \leftarrow & \end{cases}$$

Answer set of Π_b*: ?*

$$\Pi_b = \begin{cases} r_1: & flies(X) & \leftarrow & bird(X), \textbf{not } ab(r_1, X) \\ r_2: & bird(X) & \leftarrow & penguin(X) \\ r_3: & ab(r_1, X) & \leftarrow & penguin(X) \\ r_4: & penguin(tweety) & \leftarrow & \\ r_5: & bird(tim) & \leftarrow & \end{cases}$$

Answer set of Π_b*:*

$\{bird(tim), flies(tim), penguin(tweety), ab(r_1, tweety), bird(tweety)$

$\Pi_b \models flies(tim)$ *and* $\Pi_b \models \neg flies(tweety)$

Defaults – Example

Example 9 (Animals) *Consider the following information:*

- *Normally lions and tigers are cats.*
- *Sam and John are lions.*
- *Sam is not a cat.*
- *Sam is a sea lion.*

This information can be represented by the following program

$$\Pi_a = \begin{cases} r_1: & cat(X) & \leftarrow & lion(X), \textbf{not } ab(r_1, X) \\ r_2: & cat(X) & \leftarrow & tiger(X), \textbf{not } ab(r_2, X) \\ r_3: & lion(X) & \leftarrow & sea\_lion(X) \\ r_4: & ab(r_1, X) & \leftarrow & sea\_lion(X) \\ r_5: & sea\_lion(sam) & \leftarrow & \\ r_6: & lion(john) & \leftarrow & \end{cases}$$

We can check that Π_a entails that sam is a lion but not a cat while john is a cat which is a lion.

Defaults – More Examples

Example 10 • *We know that computers are normally fast machines, the commodore is a slow machine because it is an old one. This can be represented by the following program:*

$$\Pi_c = \begin{cases} r_1: & fast\_machine(X) & \leftarrow & computer(X), \textbf{not } ab(X) \\ r_2: & old(X) & \leftarrow & comodore(X) \\ r_3: & computer(X) & \leftarrow & comodore(X) \\ r_4: & ab(X) & \leftarrow & old(X) \end{cases}$$

- *The Boeing 747, concord, and FA21314 are airplanes. Airplanes normally fly unless they are out of order. FA21314 is out of order.*

$$\Pi_{airplanes} = \begin{cases} r_1: & flies(X) & \leftarrow & airplane(X), \textbf{not } ab(X) \\ r_2: & ab(X) & \leftarrow & out\_of\_order(X) \\ r_3: & airplane(boeing\_747) & \leftarrow & \\ r_4: & airplane(concord) & \leftarrow & \\ r_5: & airplane(fa21324) & \leftarrow & \\ r_6: & out\_of\_order(fa21324) & \leftarrow & \end{cases}$$

EXTENSIONS OF LOGIC PROGRAMMING AND COMPUTING ANSWER SETS

Additional Features for Knowledge Representation and Reasoning

To add expressiveness and make logic programming a more suitable language for knowledge representation and reasoning, additional features and constructors are introduced:

- *classical negation*: instead of atoms, literals are used in the rule

$$l_0 \leftarrow l_1, \ldots, l_n, \textbf{not } l_{n+1}, \ldots, \textbf{not } l_{n+k}$$

  where l_i is a literal (an atom a or its negation $\neg a$). This will allow us to represent and reason with negative information.

- *epistemic disjunction*: rule is allowed to have epistemic disjunction in the head

$$l_0 \; or \; \ldots \; or \; l_m \leftarrow l_{m+1}, \ldots, l_{m+n}, \textbf{not } l_{m+n+1}, \ldots, \textbf{not } l_{m+n+k}$$

  where *or* is a epistemic disjunction. This rule states that if $l_{m+1}, \ldots, l_{m+n}$ are true and there is no reason for believing that the $l_{m+n+1}, \ldots, l_{m+n+k}$ are true then at least one of the $l_0, \ldots, l_m$ is believed to be true.

- *nested expression*: allowing **not not** l.

- *weighted-atom*:

$$\mathtt{l}\{l_0 = w_0, \ldots, l_k = w_k, \mathbf{not}\ l_{k+1} = w_{k+1}, \ldots, \mathbf{not}\ l_{k+n} = w_{k+n}\}\mathtt{u}$$

where l_i's are a literal, w_i are integers, and `l` and `u` are two integers. This atom is true with respect to a set of literals S if

$$\mathtt{l} \leq \sum_{l_j \in S}^{0 \leq j \leq k} w_j + \sum_{l_j \notin S}^{k+1 \leq j \leq k+n} w_j \leq \mathtt{u}$$

Special case: *choice atom* – $w_i = 1$ for every i.

Programs with weight constraints are defined by allowing weight atoms to appear in place of atoms.

- *Answer set semantics*: defined accordingly. See for example,
  - M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," New Generation Computing, 1991, pp. 365-385.
  - V. Lifschitz, L. R. Tang and H. Turner, "Nested expressions in logic programs," Annals of Mathematics and Artificial Intelligence, Vol. 25, 1999, pp. 369-389.
  - I. Niemelä and P. Simons. Extending the Smodels System with Cardinality and Weight Constraints.Logic-Based Artificial Intelligence, pp. 491-521. Kluwer Academic Publishers, 2000.

Using Classical Negation

$$\Pi_b = \begin{cases} r_1: & flies(X) & \leftarrow & bird(X), \textbf{not } ab(r_1, X) \\ r_2: & bird(X) & \leftarrow & penguin(X) \\ r_3: & ab(r_1, X) & \leftarrow & penguin(X) \\ r_4: & penguin(tweety) & \leftarrow & \\ r_5: & bird(tim) & \leftarrow & \end{cases}$$

- In the bird example, "Penguins do not fly" is more intuitive than "Normally, penguins do not fly." So, r_3 of Π_b should be changed to

$$r_3' \quad : \quad \neg flies(X) \leftarrow penguin(X).$$

Doing so will make the program Π_b inconsistent! Obviously, r_1 does not account for the class of birds who do not fly. We should change the rule r_1 to

$$r_1' \quad : \quad flies(X) \leftarrow bird(X), \textbf{not } ab(r_1, X), \textbf{not } \neg flies(X)$$

The new program, $\Pi_b' = \Pi_b \setminus \{r_1, r_3\} \cup \{r_1', r_3'\}$ will correctly answer the same questions such as "does Tim flies" and "does Tweety fly?"

- Suppose that we have a list of professor and their courses:

Professor	Course
mike	ai
sam	db
staff	C

where $staff$ stands for "someone" – an unknown professor – who might be different than $mike$ and sam.

This list can be expressed by a set of atoms of the form $teach(P, C)$ (P teaches C): $teach(mike, ai)$, $teach(sam, db)$, and $teach(staff, c)$.

By default, we know that if a professor P teaches the course C, then (P, C) will be listed (and hence the atom $teach(P, C)$ will be present.) Thus, by default, professor P does not teach the course C if $teach(P, C)$ is not present. The exception to this rule are the courses taught by "staff". This leads to the following two rules:

$$\begin{array}{lll} \neg teach(P, C) & \leftarrow & \textbf{not}\ teach(P, C), \textbf{not}\ ab(P, C). \\ ab(P, C) & \leftarrow & teach(staff, C) \end{array}$$

This will allow us to conclude that $mike$ teaches ai but we do not know whether he teaches c or not.

Answer Sets of Programs with Constraints

For a set of ground atoms S and a constraint c

$$\leftarrow a_0, \ldots, a_n, \textbf{not } a_{n+1}, \ldots, \textbf{not } a_{n+k}$$

we say that c is **satisfied by** S if $\{a_0, \ldots, a_n\} \setminus S \neq \emptyset$ or $\{a_{n+1}, \ldots, a_{n+k}\} \cap S \neq \emptyset$.
Let Π be a program with constraints. Let

$$\Pi_O = \{r \mid r \in \Pi,\ r \text{ has non-empty head}\}$$

(Π_O is the set of normal logic program rules in Π) and

$$\Pi_C = \Pi \setminus \Pi_O$$

(Π_C is the set of constraints in Π).

Definition 6 *A set of atoms S is an answer sets of a program Π if it is an answer set of Π_O and satisfies all the constraints in ground(Π_C).*

Example 11 *$\Pi_2 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$ has two answer sets $\{a\}$ and $\{b\}$.*
But, $\Pi_2' = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a. \quad \leftarrow \textbf{not } a\}$ has only one answer set $\{a\}$.

Computing Answer Sets

- **Complexity**: The problem of determining the existence of an answer set for finite propositional programs (programs without function symbols) is NP-complete. For programs with disjunctions, function symbols, etc. it is much higher.

  A consequence of this property is that there exists no polynomial-time algorithm for computing answer sets.

  Special cases: answer sets of positive programs without function symbols can be computed in polynomial time in the size of the program.

  A good survey on the complexity and expresiveness of different classes of logic programs can be found in

  – Chitta Baral. Knowledge representation, reasoning and declarative problem solving, Cambridge University Press, 2003.

- **Answer set solvers**: Programs that compute answer sets of (finite and grounded) logic programs. Two main approaches:

- *Direct implementation*: Due to the complexity of the problem, most solvers implement a variation of the generate-and-test algorithm. The advantage of this approach is that we are free to invent! Many systems:
  - SMODELS available at `http://www.tcs.hut.fi/Software/smodels/`
  - DLV available at `http://www.dbai.tuwien.ac.at/proj/dlv/`
  - **deres** available at `http://www.cs.engr.uky.edu/ai/deres.html`
  - **nomore** available at `http://www.cs.uni-potsdam.de/~linke/nomore/`
- *Using SAT solvers*: A program Π is translated into a satisfiabilty problem F_Π and a call to a SAT solver is made to compute solution of F_Π. The main task of this approach is to write the program for the conversion from Π to F_Π. Recent discoveries ensure that there is an one-to-one correspondence between solutions of F_Π and answer sets of Π (e.g., see paper from the **ASSAT** web site). Two systems:
  - CMODELS available at `http://www.cs.utexas.edu/users/tag/cmodels.html`
  - **ASSAT** available at `http://assat.cs.ust.hk/`

Remark 8 *Most of the above systems make use of* **lparse**, *a grounding program for logic programs with function symbols (typed) and weight constraints. The program is available at the* SMODELS *web site.*

Remark 9 *A Java implementation of* Smodels, *called* Jsmodels, *is available at* `http://www.cs.nmsu.edu/~hle/jsmodel.html`.

Usage of lparse and Smodels

- **lparse**: a preprocessor for programs using as input to Smodels. Some restrictions apply:
  - *Finite domains:* every variable is typed and has a finite domain;
  - *Safe rule:* variables appear in the head of a rule must occur in the body of the rule, in a predicate that specifies the domain of the variable;
  - *Built-in predicates*: *assign*, *plus*, *minus*, etc.. (see **lparse** user's manual for more). Do not use them in your programs or you will get errors.
- *Command line:*

  lparse <options> prog1 prog2 ... | **smodels** number_of_models

ANSWER SET PROGRAMMING

Brief History and References

- Answer set programming was introduced in
  - V. W. Marek, M. Truszczynski. Stable models and an alternative logic programming paradigm. In The Logic Programming Paradigm, K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren (eds.), pp. 375-398. Springer-Verlag, 1999.
  - I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
- Take off thanks to the development of several answer set solvers (Smodels and Dlv),
- the application of answer set programming in several applications such as planning, system configuration, cryptography, NASA's application, etc.
- 2001: first symposium on answer set programming: "Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning," `http://www.cs.nmsu.edu/~tson/ASP2001/csp01.html`.
- 2002 – now: several symposia on ASP (`http://wasp.unime.it/`).
- 2001 – now: several papers on answer set programming have been published.

Main Idea

Given a problem P, whose solutions are $S_1, \ldots, S_n$ which can be computed by

- generating a *hypothetical* solution and
- testing whether it is *really* a solution.

This process is very similar to the process of computing answer sets of a logic program Π: given a program Π its answer sets can be computed by

- generating a set of atoms X and
- checking whether X is an answer set of Π

This gives us the idea: Solve P by representing P as a logic program Π_P whose answer sets correspond one-to-one to $S_1, \ldots, S_n$ using the follong steps:

- Repesenting P as Π_P;
- Computing answer sets of Π_P; and
- Converting answer sets of Π_Π to solutions of P

EXAMPLES

Graph Coloring

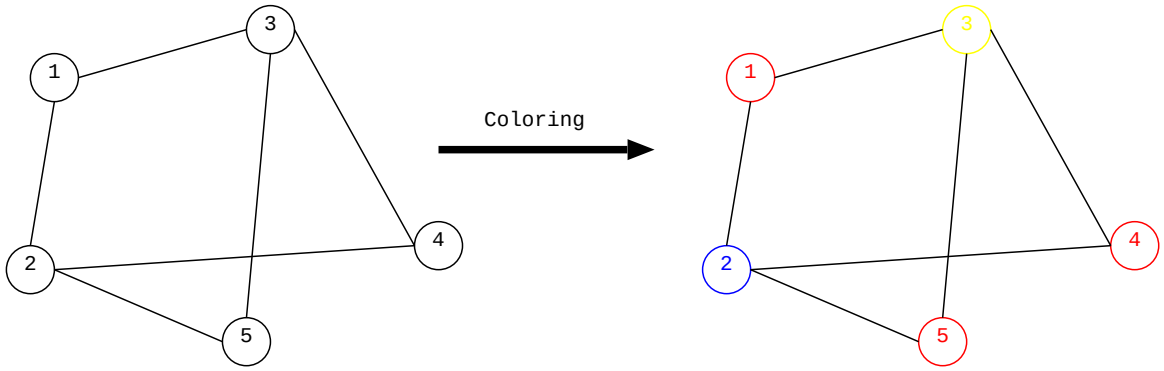


Figure 1: Graph Coloring Problem

Problem: Given a (bi-directed) graph and three colors *red*, *green*, and *yellow*. Find a color assignment for the nodes of the graph such that no edge of the graph connects two nodes of the same color.

Graph Coloring

Example 12 (Graph coloring problem) *Given a (bi-directed) graph and three colors* red, green, *and* yellow. *Find a color assignment for the nodes of the graph such that no edge of the graph connects two nodes of the same color.*

- **Representation:** *A graph can be specified by*
  - *the set of nodes and*
  - *the set of edges.*
- **Solution:** *A mapping from the set of nodes to the set of colors* $\{red, green, yellow\}$ *such that no edge connects two nodes of the same color.*
- *Solving the problem (manualy):*
  - *Numbering the nodes from 1 to n*
  - *Assigning each node a color*
  - *Checking if the assignment is a solution, i.e., satisfies the constraints* no edge connects two nodes of the same color

Graph Coloring – Program

Writing a program to solve the graph coloring problem:

- Graph representation:
  - The nodes: $node(1), \ldots, node(n)$.
  - The edges: $edge(i, j)$.
- Solution repesentation: use the predicate $color(X, Y)$ - node X is assigned the color Y.
- Generating the solutions: Each node is assigned one color. The three rules

$$color(X, red) \leftarrow \textbf{not}\ color(X, green), \textbf{not}\ color(X, yellow). \quad (3)$$
$$color(X, green) \leftarrow \textbf{not}\ color(X, red), \textbf{not}\ color(X, yellow). \quad (4)$$
$$color(X, yellow) \leftarrow \textbf{not}\ color(X, green), \textbf{not}\ color(X, red). \quad (5)$$

  or the weighted rule

$$1\{color(X, red), color(X, yellow), color(X, green)\}1 \leftarrow node(X).$$

  can be used. (The weighted rule says that each node should be colored using *exactly one* color.)

- Checking for a solution: needs to make sure that no edge connects two nodes of the same color. This can be represented by a constraint:

$$\leftarrow edge(X, Y), color(X, C), color(Y, C). \quad (6)$$

```
%% prog1
%% representing the graph
node(1).   node(2).   node(3).   node(4).   node(5).

edge(1,2). edge(1,3). edge(2,4). edge(2,5). edge(3,4). edge(3,5).

%% each node is assigned a color
color(X,red):- not color(X,green), not color(X, yellow).
color(X,green):- not color(X,red), not color(X, yellow).
color(X,yellow):- not color(X,green), not color(X, red).

%% constraint checking
:- edge(X,Y), color(X,C), color(Y,C).
```

Try with `lparse prog1 | smodel 1` or `lparse prog1 | smodel 0` and see the result.

`prog1` can be divided into three group of rules

- Rules for *describing the graph*;
- Rules for *generating the hypothetical solutions*; and
- Rules for *checking the correctness of the solutions*.

It is a good practice to separate the rules into two files: (a) the first file contains rules for describing the graph (let us called this file `prog11`, for our example, this program contains the first two lines of `prog1`); and (b) the second file contains other rules. (let us called this file `prog12` which contains the other rules of `prog1`).

```
Command line:  lparse prog11 prog12 | smodels
```

Correctness of `prog1`

Let us denote the program `prog1` developed for a graph G by Π_G.

- **What is to prove?** (one-to-one mapping between solutions of G and answer sets of Π_G.) Intuitively, this means
  - If Π_G has an answer set then the 3-coloring problem for G has a solution and vice versa.
  - If Π_G does not have an answer set then the coloring problem of G has no solution.
- **How can we prove this?**
  - Take an answer set S of Π_G, construct a solution for the coloring problem of G from S.
  - Take a color mapping M, which is a solution for the problem, construct an answer set for Π_G.

We can prove the following theorems.

Theorem 5 *Let S be an answer set of Π_G. Then, the 3-coloring problem of G has a solution corresponds to S.*

Proof. We prove the following:

1. For every node k of the graph G, S contains one and only one atom from the set $C = \{color(k, red), color(k, green), color(k, yellow)\}$, i.e., $S \cap C$ has only one element. Assume that it is not the case. Then, there are only three cases: S contains zero, two, or three elements of the set C. Assume that

Case 1: S does not contain any element from C. Then, Π_G^S contains

$color(k, red) \leftarrow node(k).$ (because of rule (3))

$color(k, green) \leftarrow node(k).$ (because of rule (4))

$color(k, yellow) \leftarrow node(k).$ (because of rule (5))

Because k is a node of G, we can easily see that $color(k, red)$, $color(k, yellow)$, and $color(k, green)$ belong to the minimal model of Π_G^S. Thus, S cannot be an answer set of Π_G because it cannot be equal the minimal model of Π_G^S. This contradicts the assumption that S is an answer set of Π_G. Hence, this case cannot happen.

Case 2: S contains two elements from C. Since the three colors are equivalent and so, without loss of generality, we assume that S contains $color(k, red)$ and $color(k, green)$; and it does not contain $color(k, yellow)$. Then, Π_G^S will not contain any rule whose head is an atom belonging to C. (all the rules of the form (3)-(5) for $X = k$ are

removed). This implies that the minimal model of Π_G^S cannot contain any element of C. This implies that S cannot be an answer set of Π_G because it cannot be equal the minimal model of Π_G^S. Again, this contradicts the assumption that S is an answer set of Π_G. Hence, this case cannot happen.

Case 3: S contains all elements of C. This is similar to the second case, i.e., Π_G^S does not contain any rule whose head is a member of C, and hence, S would not be an answer set of Π_G.

The three cases show that for each node k, S contains 1-and-only-1 member of the set $\{color(k, red), color(k, green), color(k, yellow)\}$.

2. Now we need to show that the color mapping specified by the answer set S is a solution to the coloring problem.

Let $1, \ldots, n$ be the nodes of the graph and $c_1, \ldots, c_n$ be the color such that $color(i, c_i) \in S$ for $i = 1, \ldots, n$. We need to show that if $edge(i, j)$ belongs to G then $c_i \neq c_j$. Again, we prove by contradiction. Let assume that there is an edge (p, q) in G and $c_p = c_q$. This means that the body of the rule (6) for $edge(p, q)$, $color(p, c_p)$, and $color(q, c_q)$ is satisfied. This means that S is not an answer set of Π_G, i.e., our assumption contradicts the fact that S is an answer set of Π_G. Thus, our asssumption is incorrect, i.e., we have proved that for every edge (i, j) of G, $c_i \neq c_j$. This shows that S corresponds to a solution of the 3-coloring problem for G.

Theorem 6 *If the 3-coloring problem for G has a solution M then Π_G has an answer set correcsponds to M.*

Proof. Let $1, \ldots, n$ be the nodes of the graph. Consider a solution for the 3-coloring problem for G. Let $c_1, \ldots, c_n$ be the color of the node $i = 1, \ldots, n$, respectively. We will show that the set of atoms

$$\begin{aligned} S \;=\; & \{node(i) \mid i = 1, \ldots, n\} \cup \{edge(i,j) \mid (i,j) \text{ is an edge of } G\} \cup \\ & \{color(i, c_i) \mid i = 1, \ldots, n\} \end{aligned}$$

is an answer set of Π_G.

Let us compute Π_G^S. We can see that Π_G^S consists of the following rules:

- the rules defining the graph, i.e., the rules $node(1) \leftarrow$ .... $node(n) \leftarrow$ (nodes of the graph) and the rules $edge(i,j) \leftarrow$ if (i,j) is an edge of G.
- for each node i, one rule of the form $color(i, c_i) \leftarrow node(i)$ which comes from one of the rules (3)-(5).
- for each edge (i,j), three constraints:
  $\leftarrow edge(i,j), color(i, red), color(j, red)$
  $\leftarrow edge(i,j), color(i, yellow), color(j, yellow)$
  $\leftarrow edge(i,j), color(i, green), color(j, green)$

We need to show that S is the minimal model of Π_G^S that does not violates any of the constraints.
Let $X_0 = \{node(i) \mid i = 1, \ldots, n\} \cup \{edge(i,j) \mid (i,j) \text{ is an edge of } G\}$.
Obviously, $T_{\Pi_G^S}(\emptyset) = X_0$ and
$T_{\Pi_G^S}(X_0) = S$, and $T_{\Pi_G^S}(S) = S$. (*)
Furthermore, because for every edge (i,j), $c_i \neq c_j$, we can conclude that there exists no edge (i,j) in G and a color $C \in \{red, green, yellow\}$ such that S contains $color(i,C)$ and $color(j,C)$. This is equivalent to say that the constraints in Π_G^S are satisfied by S.
Together with (*), we conclude that S is an answer set of Π_G.

n-Queens

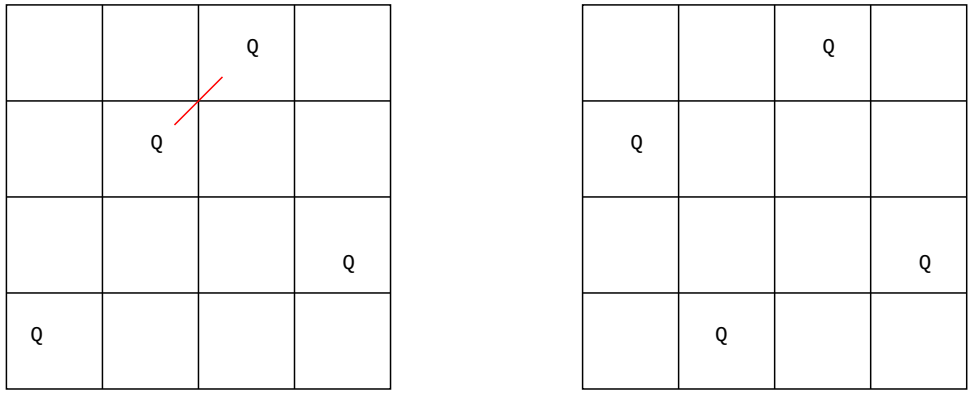


Figure 2: 4-queens Problem

Problem: Place n queens on a $n \times n$ chess board so that no queen is attacked (by another one).

n-Queens

- **Representation:** the chess board can be represented by a set of cells $cell(i, j)$ and the size n.
- **Solution:** Each cell is assigned a number 1 or 0. $cell(i, j) = 1$ means that a queen is placed at the position (i, j) and $cell(i, j) = 0$ if no queen is placed at the position (i, j)
- **Generating a possible solution:**
  - $cell(i, j)$ is either true or false
  - select n cells, each on a column, assign 1 to these cells.
- **Checking for the solution:** ensures that no queen is attacked

n-Queens – writing a program

Use a constant n to represent the size of the board

$col(1..n).$ // n columns

$row(1..n).$ // n rows

Since two queens can not be on the same column, we know that each column has to have one and only one queen. Thus, using the weighted-rule

$$1\{cell(I, J) : row(J)\}1 \leftarrow col(I).$$

we can make sure that only one queen is placed on one column. To complete the program, we need to make sure that the queens do not attack each other.

- No two queens on the same row
$$\leftarrow cell(I, J1), cell(I, J2), J1 \neq J2.$$
- No two queens on the same column (not really needed)
$$\leftarrow cell(I1, J), cell(I2, J), I1 \neq I2.$$
- No two queens on the same diagonal
$$\leftarrow cell(I1, J1), cell(I2, J2), |I1 - I2| = |J1 - J2|$$

```
%% prog2
%% representing the board, using n as a constant
col(1..n).   % n column
row(1..n).   % n row

%% generating solutions
1 {cell(I,J) : row(J)}:-  col(I).

% two queens cannot be on the same row/column
:- col(I), row(J1), row(J2), neq(J1,J2), cell(I,J1), cell(I,J2).
:- row(J), col(I1), col(I2), neq(I1,I2), cell(I1,J), cell(I2,J).

% two queens cannot be on a diagonal
:- row(J1), row(J2), J1 > J2, col(I1), col(I2), I1 > I2,
   cell(I1,J1), cell(I2,J2), eq(I1 - I2, J1 - J2).

:- row(J1), row(J2), J1 > J2, col(I1), col(I2), I1 < I2,
   cell(I1,J1), cell(I2,J2), eq(I2 - I1, J1 - J2).
```

Command line: `lparse -c n=?? prog2 | smodels`

Tile covering problem

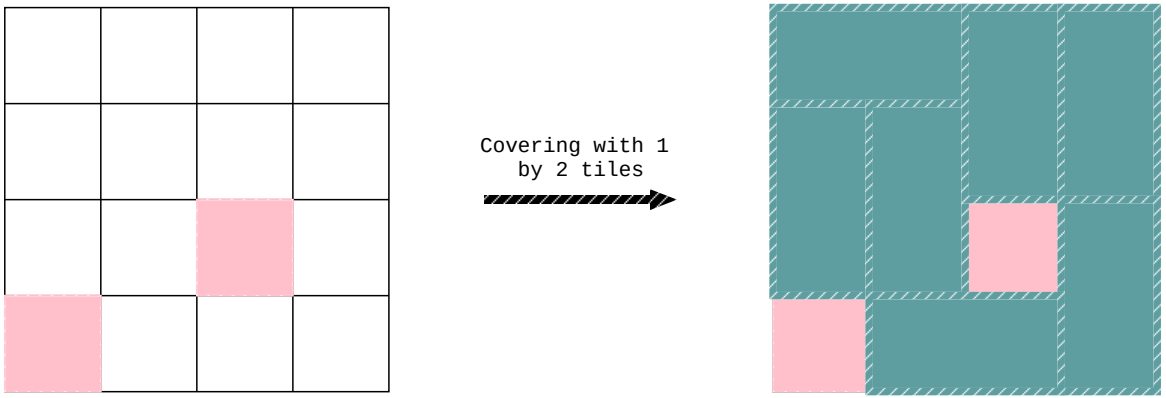


Figure 3: 4 x 4 Tile Covering Problem

Problem: Given a slightly damaged chess board ($n \times n$). Find a covering of the board using m 1×2-tiles so that all the good squares on the board are covered. If no such covering exists, report there is no solution.

Tile covering problem

- **Problem**: Given a slightly damaged chess board ($n \times n$). Find a covering of the board using m 1×2-tiles so that all the good squares on the board are covered. If no such covering exists, report there is no solution.
- **Representation**:
  - the board
    - dimension
    - damaged/good cells
  - the tiles with their coverage

  As with N-queens problem, we can use the two predicates $row()$ and $col()$ to represent the possible board cells

  We can use $bad(i, j)$ to indicate that the cell (i, j) is damaged We can use the predicate $cell(i, j, t)$ to represent the information that the cell (i, j) is covered by the tile t

- **Generating a possible solution**: We assign each tile a number. Then we can assign a tile to a location. Since each tile covers two and only two cells, we can used the weighted-rule:

$$2cell(I, J, T) : col(I) : row(J)2 \leftarrow tile(T).$$

  to generate a possible solution.

- **Checking for a solution**: We need to check for the following constraints:
  - bad cell needs not be covered
  - no two tiles on the same cell
  - The cells covered by a tile must be neightbor
  - The cells covered by a tile cannot lie on a diagonal

The result program `prog3`:

```
% representation of the board
% col(.) and row(.) as in the queen example
% bad(.) for damaged cell
% ntiles is a constant for the number of tiles

col(1..n).      row(1..n).      tile(1..ntiles).
```

```
bad(1,1).  bad(3,2).

% generating possible solutions

2 {cell(I,J,T) : col(I) : row(J) } 2 :- tile(T).

% bad cell needs not be covered

:- col(I), row(J), tile(T), bad(I,J), cell(I,J,T).

% one tile over one cell only

:- col(I),row(J),tile(T1),tile(T2),neq(T1,T2),cell(I,J,T1),cell(I,J,T2).

% cell covered by the same tile must be neightbor
% (not far from each other)

:- tile(T), cell(I1,J1,T), cell(I2,J2,T), col(I1), col(I2),
   row(J1), row(J2), I1 - I2 > 1.
```

```
:- tile(T), cell(I1,J1,T), cell(I2,J2,T), col(I1), col(I2),
   row(J1), row(J2),  J1 - J2 > 1.

% they cannot be on a diagonal

:- tile(T), cell(I1,J1,T), cell(I2,J2,T), col(I1), col(I2),
   row(J1), row(J2), neq(I1,I2), neq(J1,J2).
```

Making it Shorter

```
col(1..n).      row(1..n).      tile(1..ntiles).       %represenation

#domain tile(T;T1;T2).
#domain col(I;I1;I2).
#domain row(J;J1;J2).

bad(1,1).  bad(3,2).

2 {cell(I,J,T) : col(I) : row(J) } 2 :- tile(T).        %generating ..

:- bad(I,J), cell(I,J,T).                          % bad cell needs not ..

:- cell(I,J,T1),cell(I,J,T2),neq(T1,T2).           % one cell one tile

:- cell(I1,J1,T), cell(I2,J2,T), I1 - I2 > 1.  % neightbor only ..
:- cell(I1,J1,T), cell(I2,J2,T), J1 - J2 > 1.
:- cell(I1,J1,T), cell(I2,J2,T), neq(I1,I2), neq(J1,J2). % no diagonal
```

K-clique

- **Problem**: Given a number k and a graph G. We say that G has a clique of size k if there is a set of k different vertices (nodes) in G such that each pair of vertices from this set is connected through an egde.
- **Representation**:
  - Graph ($node()$ and $edge()$)
  - Clique ($clique(N)$) to say that node N belongs to the clique if $clique(N)$ is true; otherwise it does not belong to the clique.
- **Generating a solution**: Selecting k nodes – this is equivalent to assigning k atoms of the set

$$\{clique(1), \ldots, clique(n)\}$$

the truth value true. This can be achieved by the rule

$$k\{clique(N) : node(N)\}k.$$

- **Checking for a solution**: if every pair of the selected nodes is connected then this is a solution; otherwise it is not a solution. This means that there exists no pair (I, J) such that $clique(I)$ and $clique(J)$ are true but $edge(I, J)$ is not true.

$$\leftarrow clique(I), clique(J), I \neq J, \textbf{not } edge(I, J).$$

```
% prog4
node(1..5).                          % nodes

#domain node(N1;N2).

edge(1,2). edge(1,3). edge(2,3).     % edges
edge(1,4). edge(1,5).

edge(N1,N2):- edge(N2,N1).           % bi-directed

k {clique(N):node(N)} k.             % generating solution

:- clique(N1), clique(N2), neq(N1,N2), not edge(N1,N2).
```

Constraint Satisfaction Problem

- **Problem**: A constraint satisfactory problem (**CSP**) consists of
  - a set of variables $v_1, \ldots, v_k$;
  - a set of possible values for each variable, called the $d$omain of the variable;
  - a set of constraints where a constraint can be either a *allowed combination* of variables or a *prohibited combination* of variables;
  - *sometime*, an optimal constraint on some objective function.
- **Solution**: An assignment of variables such that non of the constraints is violated.
- **Idea for solving using CSP answer set programming**:
  - rules for specifying the domains of variables
  - rules for generating a possible solution
  - rules for checking the constraints

Examples of CSP – n-Queens Problem

The n-queens problem could be viewed as a CSP:

- Variables: n-pairs $(x_1, y_1), \ldots, (x_n, y_n)$, each represents a queen
- Domains of each variable: $1 \leq x_i, y_i \leq n$
- Prohibited constraints: for every pair of $i \neq j$, $x_i \neq x_j$, $y_i \neq y_j$, and $|x_j - x_i| \neq |y_j - y_i|$.
- We do not have constraints for allowed combination.

Examples of CSP – Combinatorial Auction

A company sells A a number of products $p_1, \ldots, p_n$ to m customers $c_1, \ldots, c_m$. Each customer bids on some of the items. The set of the items for customer c_i will be denoted by s_i. A customer wins a bid would imply that he gets everything he wants. The company wants to get maximal profit. How should A decide who wins the auction?
Let b_i denote the bid of customer i. Then, the problem is to find the value for b_i, i.e., the variables of the problem are b_i's.
The domain of each variable is $\{0, 1\}$.
We do not allow that two customers get the same item. This gives the prohibited combination: $\neg(b_i \wedge b_k)$ for every pair $i \neq k$ if $s_i \cap s_k \neq \emptyset$.
The company wants to get maximal profit means that the sum

$$\Sigma_{i=1}^{n} b_i pb_i$$

should be maximum where pb_i is the value of the item p_i offered by customer i.

Example 13 *There are three items a, b, and c. Customer 1 wants a for \$US 2 and b for \$US 4. Customer 2 wants a for \$US 1 and c for \$US 6. Who should win?*
The problem has two variables: b_1 *and* b_2*;* $s_1 = \{a, b\}$ *and* $s_2 = \{b, c\}$*.* $s_1 \cap s_2 \neq \emptyset$*.*
This means that we have a constraint $\neg(b_1 \wedge b_2)$*, i.e., we cannot sell to both 1 and 2.*
Then, the constraint to maximize $\sum_{i=1}^{n} b_i pb_i$ *makes us decide that 2 wins, i.e, we should sell to 2.*

Solving CSP by Answer Set Programming

Given a CSP problem P, we define Π_P as a program consists of the following rules

- for each variable v_i and each element c_{ij} in the domain of v_i, Π_P contains a rule

$$v_i(c_{ij})$$

- for each variable v_i, Π_P contains the rule

$$1\{v_i(c_{i1}), \ldots, v_i(c_{in_j})\}1$$

- for each allowed combination co which allows $v_{i_1}, ldots, v_{i_j}$ to take a value $c^*_{i_1}, \ldots, c^*_{i_j}$, Π_P contains the following three rules:
  - $constraint(co) \leftarrow$
  - $satisfied(co) \leftarrow v_{i_1}(c^*_{i_1}), \ldots, v_{i_j}(c^*_{i_j})$
  - $\leftarrow constraint(co), \textbf{not}\ satisfied(co).$

  which have the final effect of requiring v_{i_l} to be assigned to the value c_{i_j}.

- for each prohibited combination *co* that disallows $v_{i_1}, ldots, v_{i_j}$ to take a value $c^*_{i_1}, \ldots, c^*_{i_j}$, Π_P contains the constraint:
  $\leftarrow v_1(c^*_1), \ldots, v_n(c^*_n)$.

  NOTE: We simplify the problem a little bit here. A constraint *co* might place conditions on only few variables – not all. Also, a variable might occur in more than one allowed combination.

A Program for n-Queens as CSP

Describing the variables:

$x_1(1).\quad x_1(2).\quad \ldots\quad x_1(n).\quad \ldots\quad x_2(1).\quad x_2(2).\quad \ldots\quad x_2(n).$

$y_1(1).\quad y_1(2).\quad \ldots\quad y_1(n).\quad \ldots\quad y_n(1).\quad y_n(2).\quad \ldots\quad y_n(n).$

For the prohibited combination $x_i \neq x_j$ we have:

$\leftarrow x_1(1), x_2(1).\quad \leftarrow x_1(1), x_3(1)\quad \ldots\quad \leftarrow x_1(1), x_n(1).$

$\leftarrow x_1(2), x_2(2).\quad \leftarrow x_1(2), x_3(2)\quad \ldots\quad \leftarrow x_1(2), x_n(2).$

$\ldots$

$\leftarrow x_1(n), x_2(n).\quad \leftarrow x_1(n), x_3(n)\quad \ldots\quad \leftarrow x_1(n), x_n(n).$

Similar rules are written for the prohibited combination $y_i \neq y_j$.

Rules for the prohibited combination $|x_i - x_i| \neq |y_i - y_j|$: for each pair $i \neq j$

$\leftarrow x_i(c1), y_i(r1), x_j(c2), y_j(r2),$

$\quad abs(c1, r1, a1), abs(c2, r2, a2), a1 \neq a2.$

where $abs(p, q, p - q) \leftarrow p \geq q$ and $abs(p, q, q - p) \leftarrow p < q$ are the two additional rules.

REASONING ABOUT ACTIONS AND CHANGES

An Example – A Variation of the Yale Shooting Problem

Consider the story: *Matt* – a turkey – is walking along the road. *Jimmy* – a hunter – is coming from the opposite direction. He takes out a loaded gun and shoots at *Matt*.

There are several questions that arise given the above story:

- Is *Matt* still alive?
- Is the gun still loaded?
- Is *Jimmy* walking?
- Does *Jimmy* have the same number of guns?
- etc.

Problems:

- the *frame problem* – compact representation of what does not change after the execution of an action.
- the *ramification problem* – reasoning about indirect effects of actions.

Frame and Ramification Problem

Consider the story: *Matt* – a turkey – is walking along the road. *Jimmy* – a hunter – is coming from the opposite direction. He takes out one of his loaded guns and shoots at *Matt*.

From the story, we know that the action "shoot" occurs. Commonsense tells us that the turkey will be dead and the gun becomes unloaded if it can hold at most one bullet. However, several other properties of the environment stay unchanged after the action has completed. For instance,

- the road does not change its direction;
- the number of bullets in the other guns of Jimmy does not change;
- the number of guns belonging to Jimmy does not change
- the amount of water in the lake nearby does not change; etc.
- etc.

We also know that if Matt is hit by the gun, he will be dead and if he is dead he cannot not continue walking.

The Language $\mathcal{A}$

$\mathcal{A}$ is a high-level action language for representing and reasoning about actions and change. It has a simple and independent semantics based on transition system. It is introduced in

- M. Gelfond and V. Lifschitz: "Representing Actions and Change by Logic Programs", Journal of Logic Programming, vol. 17, Num. 2,3,4, pp. 301–323, 1993.

Several extensions of $\mathcal{A}$ have been proposed. We will use $\mathcal{AL}$, which is introduced in

- C. Baral, M. Gelfond: "Reasoning agents in Dynamic Domains." Logic Based Artificial Intelligence , Edited By J. Minker, Kluwer 2000

In $\mathcal{AL}$, an action theory is defined over two disjoint sets, a set of fluents (a fluent is a property whose value changes over time) and a set of actions, and is a set of propositional propositions of the form

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \tag{7}$$

$$f \textbf{ if } p_1, \ldots, p_n \tag{8}$$

$$\textbf{initially } f \tag{9}$$

where f and p_i's are fluent literals (a *fluent literal* is either a fluent g or its negation $\neg g$, written as $neg(g)$) and a is an action. (7), referred as *dynamic law*, represents the (conditional) effect of action a. (8) is a *static law* which represents the relationship between fluents. Propositions of the form (9), also called *v-propositions*, are used to describe the initial situation. An action theory is given by a pair (D, I) where D consists of propositions of the form (7)-(9) and I consists of propositions of the form (9). D and I will be called the *domain description* and *initial state*, respectively. We assume that for each fluent f, **initially** f or **initially** $\neg f$ belongs to I but not both.

Example 14 (Story) • *To say that initially, the turkey is walking and not dead, we write*

$$\textbf{initially}\ \neg dead \quad \text{and} \quad \textbf{initially}\ walking$$

- *Initially, the gun is loaded*

$$\textbf{initially}\ loaded$$

- *Shooting causes the turkey to be dead if the gun is loaded can be expressed by*

$$shoot\ \textbf{causes}\ dead\ \textbf{if}\ loaded \quad \text{and} \quad shoot\ \textbf{causes}\ \neg loaded\ \textbf{if}\ loaded$$

- *Un/Loading the gun causes the gun to be un/loaded*

$$load\ \textbf{causes}\ leaded \quad \text{and} \quad unload\ \textbf{causes}\ \neg loaded$$

- *Dead turkeys cannot walk*

$$\neg walking\ \textbf{if}\ dead$$

So, the action theory is

$$I_y = \{\ \textbf{initially}\ \neg dead,\ \textbf{initially}\ walking,\ \textbf{initially}\ loaded\}$$

and

$$D_y = \left\{ \begin{array}{ll} shoot\ \textbf{causes}\ dead\ \textbf{if}\ loaded & shoot\ \textbf{causes}\ \neg loaded\ \textbf{if}\ loaded \\ load\ \textbf{causes}\ leaded & \neg walking\ \textbf{if}\ dead \end{array} \right\}$$

Semantics

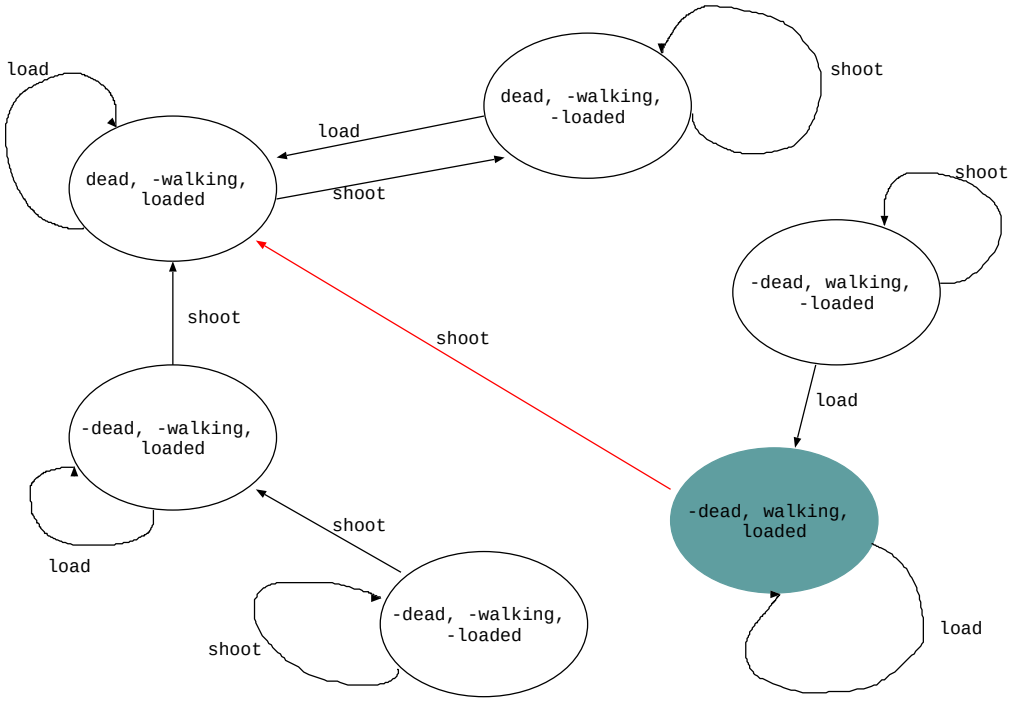


Figure 4: Transition Diagram for (D_y, I_y)

Each node represents a *state*: a consistent and complete set of fluent literals satisfying all the static laws in D_y. (Picture)

- Each node represents a *state*: a consistent and complete set of fluent literals satisfying all the static laws in D_y.
- A fluent literal l holds in a state s if $l \in s$.
- Each directed link between two states s_1 and s_2 represents a *transition* from s_1 to s_2 due to the execution of the action (the label of the link). s_2 is called a possible next state.
- Transition function Φ specifies the set of possible next states:

$$\Phi(a, s) = \{s' \mid s' \text{ is a possible next state}\}.$$

  For example,

$$\Phi(shoot, s_1) = \{s_2\}$$

  where $s_1 = \{loaded, walking, \neg dead\}$ and $s_2 = \{\neg loaded, \neg walking, dead\}$.
- Φ is extended to define $\hat{\Phi}$ by
  - $\hat{\Phi}([], s) = \{s\}$
  - $\hat{\Phi}([a_1, \ldots, a_n], s) = \cup_{s' \in \Phi(a_1, s)} \hat{\Phi}([a_2, \ldots, a_n], s')$.

The entailment relation of (D, I) is defined by:

$$D \models f \textbf{ after } \ a_1, \ldots, a_n \text{ iff } f \text{ holds in every state in } \hat{\Phi}([a_1, \ldots, a_n], s_0). \qquad (10)$$

Interested Questions in Reasoning About Actions and Changes

Given an action theory (D, I), we are often interest in the following questions/problems:

- *Projection:* What will be true/false after the execution of the sequence of action $a_1, \ldots, a_m$ from the initial state? Or, whether

$$(D, I) \models f \textbf{ after } \ a_1, \ldots, a_n \tag{11}$$

  holds for a given fluent literal f.

- *Planning:* Which sequence of actions will changes the world from the initial state into a state that satisfies a given fluent literal f? Or, find a sequence of action $a_1, \ldots, a_m$ such that f will be true after the execution of the sequence of action $a_1, \ldots, a_m$ from the initial state.

Remark 10 *We can easily generalize the result to answer questions with respect to a fluent formula.*

A Logic Program for Projection (Computing $\models$)

Given an action theory (D, I) and a sequence of actions $\alpha = [a_1, \ldots, a_n]$, we want to know what is true and what is false after the execution of α from the initial state. In other words, for every fluent f, we want to know whether

$$(D, I) \models f \textbf{ after } \ a_1, \ldots, a_n$$

holds or not.

We will compute $\models$ using logic programming. For each action theory (D, I), we define a program $\pi(D, I)$ as follows. The language of $\pi(D, I)$ contains the following predicates:

- $holds(F, T)$: this states that the fluent literal F holds at time T.
- $occ(A, T)$: action A occurs at time moment T.
- $time(T)$: denotes the time moment T.

and we make sure that the following property holds:

$$(D, I) \models f \textbf{ after } \ a_1, \ldots, a_n \quad iff \quad \pi(D, I) \models holds(f, n)$$

where the first $\models$ symbol represents the entailment relation defined in (10) and the second $\models$ symbol represents the entailment relation defined by the answer set semantics of $\pi(D, I)$.

The program $\pi(D, I)$ is defined as follows:

- For each dynamic law "a **causes** f **if** $f_1, \ldots, f_n$", $\pi(D, I)$ contains the following rule:

$$holds(f, T+1) \leftarrow time(T), occ(a, T), holds(f_1, T), \ldots, holds(f_n, T). \quad (12)$$

- For each static law "f **if** $f_1, \ldots, f_n$" , $\pi(D, I)$ contains the rule:

$$holds(f, T+1) \leftarrow time(T), holds(f_1, T), \ldots, holds(f_n, T). \quad (13)$$

- For each v-proposition " **initially** f", $\pi(D, I)$ contains the rule

$$holds(f, 0). \quad (14)$$

- For each fluent f, $\pi(D, I)$ will also contain the well-known *inertial rules*:

$$\begin{array}{lll} holds(f, T+1) & \leftarrow & time(T), fluent(f), holds(f, T), \textbf{not } holds(\neg f, T+1). \\ holds(\neg f, T+1) & \leftarrow & time(T), fluent(f), holds(\neg f, T), \textbf{not } holds(f, T+1). \end{array} \quad (15)$$

Remark 11 *$\pi(D, I)$ also contains some rules that make the encoding more compact. (See example).*

Let π^n be the set of rules of $\pi(D, I)$ in which the time variable takes the value from 0 to n. Let $\pi = \pi^n \cup \{occ(a_1, 0), \ldots, occ(a_n, n-1)\}$. The following can be proven:

Theorem 5 *For every action theory (D, I), a sequence of actions $a_1, \ldots, a_n$, and a fluent f,*

$$(D, I) \models f \textbf{ after } \ a_1, \ldots, a_n$$

iff

$$\pi \models holds(f, n),$$

i.e., $holds(f, n)$ belongs to every answer set of the program π.

Example – One Gun

For the Yale-Shooting problem we have the following rules (See PROGA1):

```
% Defining time   %defining fluents
time(0..length).  fluent(loaded).   fluent(dead).  fluent(walking).

% Defining actions
action(load).     action(shoot).

% The initial state
holds(loaded, 0).   holds(neg(dead), 0).   holds(walking, 0).

% Representing action's effects
holds(dead, T+1) :- time(T), occ(shoot, T), holds(loaded, T).
holds(neg(loaded), T+1) :- time(T), occ(shoot, T).
holds(loaded, T+1) :- time(T), occ(load, T).

% Static law
holds(neg(walking), T) :- holds(dead, T).
```

```
% Defining fluent literals/Contrary literals
literal(F):- fluent(F).                literal(neg(F)):- fluent(F).
contrary(F, neg(F)):- fluent(F).       contrary(neg(F), F):- fluent(F).

% The inertial rule
holds(F,T+1):-  literal(F), time(T), holds(F,T),
                contrary(G,F), not holds(G,T+1).
```

To determine whether

$$(D, I) \models \mathit{dead}\ \mathbf{after}\ \ \mathit{shoot},$$

we compute the answer sets of the program $\pi = \pi^1 \cup \mathit{occ}(\mathit{shoot}, 0)$ using the command

```
lparse -c length=1 proga1 a1inst | smodel 0
```

where the file A1INST contains the two facts: $\mathit{occ}(\mathit{shoot}, 0)$. It is easy to verify that $\mathit{holds}(\mathit{dead}, 1)$ belongs to every answer set of π and thus $(D, I) \models \mathit{dead}$ **after** shoot.

Remark 12 *The constant* length *is used to specify the length of the action sequence.*

Considering Executability Condition

Sometime, an action requires some condition for it to be executable. In $\mathcal{AL}$, this is called an *executability condition* and is described by a proposition of the form

$$a \textbf{ executable\_if } p_1 \ldots, p_n \tag{16}$$

which says that a can only be executed if the fluent literals $p_1 \ldots, p_n$ hold.
For instance, *Jimmy* (the hunter) can only execute the action *shoot* if he is not *wounded*. This is expressed by

$$shoot \textbf{ executable\_if } \neg wounded$$

Introducing executability condition requires some changes:

- In the transition diagram, the link with the label a between s and s' exists only if a is executable in s and s' is a possible state reached after executing a.

- In the program $\pi(D, I)$, for each proposition (16), we add a rule

$$possible(A, T) \leftarrow holds(p_1, T), \ldots, holds(p_n, T).$$

Furthermore, we add a constraint

$$\leftarrow occ(A, T), \textbf{not } possible(A, T)$$

to $\pi(D, I)$ which forbids A to occur when its executability condition is not satisfied.

$\pi(D, I)$ can be used in the same way for projection as before.

Another Example — The Block World Domain

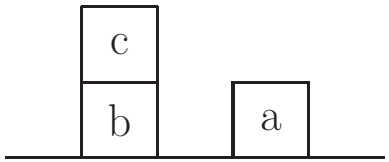


Figure 5: A Block World Domain

In this domain we have 3 blocks a, b, c. Each block is either on the table or on top of another block. They form tower like b and c (a tower of 2 blocks), and a (a tower of one block). The top block of a tower can be moved to the table or on top of another tower. To move a block, that is not on top of a tower, we need to move all the blocks above it. Furthermore, we are interested in building a new tower or finding out where a block is located, whether a block is on the table, on top of some other block, etc.

We will write $on(X, Y)$ to denote that block X is on block Y; We will also use t as a constant to denote the table. We write $move(X, Y)$ to represent the action that moves X on top of Y.

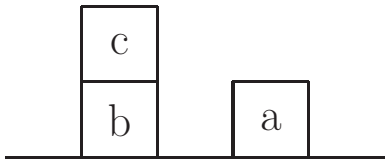


Figure 6: A Block World Domain

The set of fluents **F** is

$$\{on(X,Y) \mid X,Y \in \{a,b,c,t\}\} \setminus (\{on(X,X) \mid X \in \{a,b,c,t\} \cup \{on(t,X) \mid X \in \{a,b,c\}).$$

Furthermore, the set of actions **A** is

$$\{move(X,Y) \mid X,Y \in \{a,b,c\}, X \neq Y\} \cup \{move(X,t) \mid X \in \{a,b,c\}\}.$$

We begin with describing the initial state:

$$I_b = \begin{cases} \textbf{initially} \;\; on(c,b) \\ \textbf{initially} \;\; on(b,t) \\ \textbf{initially} \;\; on(a,t) \end{cases}$$

In addition, we also have that
initially $\neg on(X,Y)$ where $on(X,Y) \in \mathbf{F} \setminus \{on(c,b), on(b,t), on(a,t)\}$. The domain D_b contains of the following propositions

$$D_b = \begin{cases} move(a,b) \; \textbf{executable\_if} \; \neg on(c,a), \neg on(c,b), \neg on(b,a), \neg on(a,b) \\ move(a,t) \; \textbf{executable\_if} \; \neg on(c,a), \neg on(b,a) \\ \ldots \\ move(a,b) \; \textbf{causes} \; on(a,b) \\ move(a,t) \; \textbf{causes} \; on(a,t) \\ \ldots \\ \neg on(a,t) \; \textbf{if} \;\; on(a,b) \\ \neg on(a,c) \; \textbf{if} \;\; on(a,b) \\ \ldots \end{cases}$$

A Program for the Block World Domain

```
% Defining the time constants and objects
time(0..length).      block(a).   block(b).   block(c).

% Defining fluents
fluent(on(X,Y)):- block(X), block(Y), neq(X,Y).
fluent(on(X,t)):- block(X).

% Defining an additional fluent whose value is determined by others
fluent(clear(X)):- block(X).

% Defining actions

action(move(X,Y)):- block(X), block(Y), neq(X,Y).
action(move(X,Y)):- block(X), table(Y).
```

```
% Static law
holds(neg(on(X,Y)), T):-  time(T),
      block(X), block(Y), neq(X,Y),
      block(Z), neq(X,Z), neq(Y,Z), holds(on(X,Z), T).

holds(neg(on(X,Y)), T):- time(T),
      block(X), block(Y), neq(X,Y), holds(on(X,t), T).

holds(neg(on(X,t)), T):- time(T),
      block(X), block(Y), neq(X,Y), holds(on(X,Y), T).

holds(neg(clear(X)), T):- time(T),
      block(X), block(Y), neq(X,Y),
      fluent(on(Y,X)), holds(on(Y,X), T).

holds(clear(X), T):- time(T),
      block(X), not holds(neg(clear(X)), T).

% Executability condition
possible(move(X,t),T):- block(X), time(T), holds(clear(X), T).
```

```
possible(move(X,Y),T):- time(T), block(X), block(Y), neq(X,Y),
      holds(clear(X), T), holds(clear(Y), T).

:- action(A), time(T), occ(A,T), not possible(A,T).

% Representing action effects

holds(on(X,Y), T+1):- time(T),
      block(X), block(Y), neq(X,Y), occ(move(X,Y), T).

holds(on(X,t), T+1):- time(T), block(X), occ(move(X,t), T).

% The initial state
holds(on(c,b), 0).
holds(on(a,t), 0).
holds(on(b,t), 0).
holds(neg(F), 0):- fluent(F), not holds(F, 0).

% Defining fluent literals/Contrary literals
```

```
literal(F):- fluent(F).          literal(neg(F)):- fluent(F).
contrary(F, neg(F)):- fluent(F). contrary(neg(F), F):- fluent(F).

% The inertial rule
holds(on(X,Y), T+1) :-
      block(X), block(Y), neq(X,Y),
      time(T),
      holds(on(X,Y), T),
      not holds(neg(on(X,Y)), T+1).

holds(on(X,t), T+1) :-
      block(X), block(Y), neq(X,Y),
      time(T),
      holds(on(X,t), T),
      not holds(neg(on(X,t)), T+1).
```

We then can use SMODELS to answer query like

$$(D_b, I_b) \models on(a, c) \wedge on(c, t) \textbf{ after } [move(c, t), move(a, c)]$$

Planning

Answer set planning was introduced in

- V. Lifschitz, "Answer set planning," in Proceedings of the 1999 International Conference on Logic Programming, 1999, pp. 23-37.
- V. Subrahmanian and C. Zaniolo, "Relating stable models and ai planning domains," In Proceedings of the International Conference on Logic Programming, 1995, pp. 233–247.

A *planning problem* is specified by a triple $\langle D, I, \varphi \rangle$ where (D, I) is an action theory and φ is a fluent formula (or *goal*), representing the goal state. A sequence of actions $a_1, \ldots, a_m$ is a *plan for* φ if

$$(D, I) \models \varphi \textbf{ after } \ a_1, \ldots, a_m.$$

Given a planning problem $\langle D, I, \varphi \rangle$, answer set planning solves it by translating it into a logic program $\Pi(D, I, \varphi)$ that has two components:

- The program $\pi(D, I)$: this program describes the action theory (D, I).
- The set of rules for describing the goal and generating action occurrences:
  - for each $f \in \varphi$, $\Pi(D, I, \varphi)$ contains the rule

$$\leftarrow \ not\ holds(f, length). \tag{17}$$

    which guarantees that f holds at the time moment $length$
  - a rule of the form

$$1\{occ(A, T) : action(A)\}1 \leftarrow time(T), T < length. \tag{18}$$

    which states that at any moment of time, one and only one action must occur. We add the condition $T < length$ to not allow actions to occur at the time $length$.

Remark 13 *Correctness of* $\Pi(D, I, \varphi)$ *can be found in*

- *Tran Cao Son, Chitta Baral, Tran Hoai Nam, and Sheila McIlraith, "Domain-Dependent Knowledge in Answer Set Planning," ACM TOCL.*

Answer Set Planning – Examples

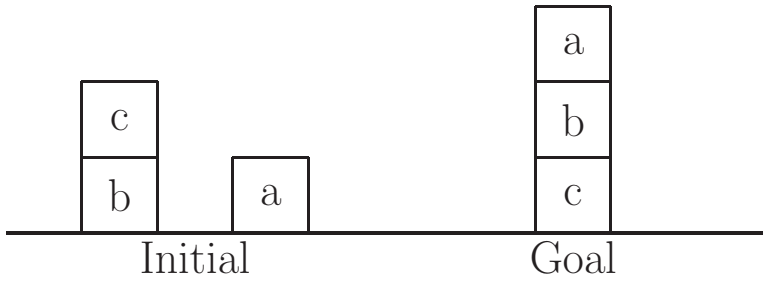
- Suppose that we are interested in the planning problem $(D_y, I_y, dead)$, we need to add the following rules to this program:

```
:- not holds(dead, length).

1{occ(A,T) : action(A)} 1 :- time(T), T < length.
```

  Generating an answer set for this, we will find a plan *shoot*.

- For the block world domain, to check whether the goal situation in the next Figure can be achieved



we add the following rule to the program:

```
1 {occ(A,T) : action(A) } 1 :- time(T), T < length.

:- not goal(length).

goal(T):- time(T), holds(on(c,t), T),
      holds(on(b,c), T), holds(on(a,b), T).
```

Computing an answer set, we obtain an answer set containing the atoms `occ(move(b,c),1) occ(move(a,b),2) occ(move(c,t),0)` which represent the plan $move(c,t), move(b,c), move(a,b)$.

Another Example — Missionaries and Cannibals

```
% Defining the constants
time(0..length).
number(0..3).
missionary(1..3).       cannibal(1..3).
location(l1).           location(l2).      % two banks

% X missionaries and Y cannibals are at location L
fluent(at(X,Y,L)):-      missionary(X),cannibal(Y),location(L).

% the boat is at location L
fluent(boat_at(L)):-     location(L).

% X missionaries and Y cannibals move from one bank to another bank
action(cross(I,J,L)):-
   number(I), number(J), location(L), I+J <= 2, I+J > 0.
```

```
% effects of crossing
holds(at(M+P,N+Q,L1),T+1):-
   time(T),
   number(M), number(N), location(L),
   number(P), number(Q), action(cross(M,N,L)),
   occ(cross(M,N,L), T), location(L1), neq(L,L1),
   holds(at(P,Q,L1), T).

holds(boat_at(L1),T+1):-
   time(T),
   number(M), number(N), location(L),
   action(cross(M,N,L)), location(L1), neq(L,L1),
   occ(cross(M,N,L), T).

holds(neg(boat_at(L)),T+1):-
   time(T),
   number(M), number(N), location(L),
   action(cross(M,N,L)),
   occ(cross(M,N,L),T).
```

```
holds(at(P-M,Q-N,L),T+1):-
   time(T),
   number(M), number(N), location(L),
   number(P), number(Q), action(cross(M,N,L)),
   occ(cross(M,N,L), T),
   holds(at(P,Q,L), T).

% executability condition
possible(cross(I,J,L), T):-
   time(T),
   number(I), number(J), location(L),
   number(P), number(Q), action(cross(I,J,L)),
   holds(boat_at(L), T),
   holds(at(P, Q, L), T),  P>=I, Q>=J.

% initial condition
holds(at(3,3,l1),0).
holds(at(0,0,l2),0).
holds(boat_at(l1),0).
```

```
% constraint
:- number(I), number(J), location(L), I+J <= 2, I+J>0,
   time(T), occ(cross(I,J,L), T), not possible(cross(I,J,L), T).

not_good :- number(I), number(J), location(L),
   time(T), holds(at(I,J,L), T), I < J, I > 0.

:- not_good.

% goal
:- not goal(length).

goal(T):- time(T), holds(at(3,3,l2), T).
goal(T+1):- time(T), goal(T).

1 { occ(A, T): action(A) } 1 :- time(T), not goal(T).
```

Diagnosis

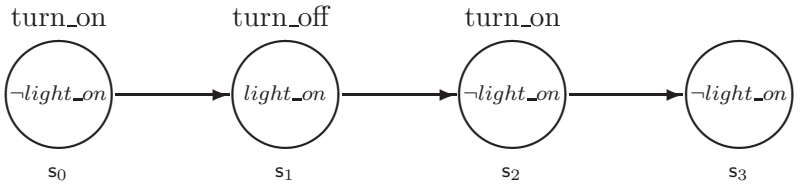
Reference

- Chitta Baral, Sheila McIlraith, and Tran Cao Son. Formulating diagnostic problem solving using an action language with narratives and sensing, Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning (KRR'00), 2000, pages 311-322.

Consider the following narrative

- *9am:* John arrived at work and turned the light on. As usual, the light went on and John started his daily work.
- *12 pm:* John turned off the light and went to lunch.
- *1pm:* John turned on the light when he got back from lunch. The light did not go on.
- *1:15pm*: The company's electrician arrived. He replaced the bulb and turned on the light, the light did not go on. He then checked the fuses and replaced one which was blown. The light is back.

Narrative Description – General Idea



The narrative can be described by a triple $(SD, COMPS, OBS)$ where

- SD is an action theory describing actions (e.g. *turn_on, turn_off, replace_bulb, etc.*) and their expected outcomes and relationships between fluents. It includes also actions that are beyond the control of the agents such as $break(bulb)$ causes the bulb to be broke.
- $COMP$ is a set of objects that can be broken (*bulb, fuse, ...*); for each object o, an action of the form $break(o)$ with the outcome $ab(o)$, indicating that o is broken, is included in SD.
- OBS is a set of observations that describes the history of the world (might be incomplete) in term of which actions were executed, when they were executed relative to each other, and what are the outcomes of the actions;

The Narrative as a System

We illustrate the concepts using the example. Let $Sys = (SD, \{bulb\}, OBS)$ be a ssytem with

$$SD = \begin{cases} (\text{r1}) & turn\_on \textbf{ causes } light\_on \textbf{ if } \neg ab(bulb) \\ (\text{r2}) & turn\_off \textbf{ causes } \neg light\_on \\ (\text{r3}) & \neg light\_on \textbf{ if } ab(bulb) \\ (\text{r4}) & break(bulb) \textbf{ causes } ab(bulb) \end{cases}$$

and

$$OBS = \begin{cases} (\text{o1}) & turn\_on \textbf{ occurs\_at } \mathsf{s}_0 \\ (\text{o2}) & turn\_off \textbf{ occurs\_at } \mathsf{s}_1 \\ (\text{o3}) & turn\_on \textbf{ between } \mathsf{s}_2, \mathsf{s}_3 \\ (\text{o4}) & \mathsf{s}_0 \textbf{ precedes } \mathsf{s}_1 \\ (\text{o5}) & \mathsf{s}_1 \textbf{ precedes } \mathsf{s}_2 \\ (\text{o6}) & \mathsf{s}_2 \textbf{ precedes } \mathsf{s}_3 \\ (\text{o7}) & \neg light\_on \textbf{ at } \mathsf{s}_0 \\ (\text{o8}) & light\_on \textbf{ at } \mathsf{s}_1 \\ (\text{o9}) & \neg light\_on \textbf{ at } \mathsf{s}_2 \\ (\text{o10}) & \neg light\_on \textbf{ at } \mathsf{s}_3 \end{cases}$$

Diagnostic Reasoning Process

Address the questions

- when does a system need a diagnosis?

  Answer: When there are inconsistency between observations and expected outcomes; or when the system does not have a model if we remove all actions $break(o)$ from SD.

- what are the diagnoses?

  Answer: Additional action occurrences that help explain the indescrepancies between observations and expected outcomes.

- how to fix a system that needs a diagnosis?

  Answer: Collecting enough information and executing test actions if necessary so that a diagnosis is singed out. Thereafter, executing the repair actions necessary to fix the system.

Diagnostic Reasoning Process – A Summary

Answer the question: when a system needs a diagnosis and what are the diagnoses.

- generating candidate diagnoses based on an incomplete history of events that have occurred and observations that have been made.
- in the event of multiple candidate diagnoses, performing actions to enable observations that will discriminate candidate diagnoses. The selection of a particular action is often biased towards confirming the most likely diagnosis, or the one that is easiest to test.
- generating (possibly with conditional) plans, comprising both world-altering actions and sensing actions, to discriminate candidate diagnoses.
- updating the space of diagnoses in the face of changes in the state of the world, and in the face of new observations.

The Narrative as a Logic Program – General Idea

Given a system description $(SD, COMPS, OBS)$ with

- SD: a set of propositions of the form a **causes** f **if** $p_1, \ldots, p_n$ or f **if** $p_1, \ldots, p_n$; this set of propositions describes the normal system behavior;
- $COMPS$: a set of components that can be broken
- OBS: a set of observations representing a narrative of the system.

We will write a logic program $\Pi(SD, COMPS, OBS)$ (or Π for short) to compute diagnoses. Π will need to contain the following parts:

- rules for generating a sequence of actions
- rules for checking if the generated sequence of actions is a possible diagnosis; this includes
  - rules that assign situations to time moments – this assignement must respect the ordering between situations in the observations
  - rules that make sure that observations are satisfied.

Elements of Π – Part 1

Constants: time, situations, actions, fluents, and literals

```
time(0..length).

% define situations
sit(s0).          sit(s1).          sit(s2).    sit(s3).

% actions
action(turn_on).  action(turn_off).  action(break(bulb)).

% fluents
fluent(light_on).                   fluent(ab(bulb)).

% literal
literal(F):- fluent(F).             literal(neg(F)):- fluent(F).
contrary(F, neg(F)):- fluent(F).    contrary(neg(F), F):- fluent(F).
```

Elements of Π – Part 2

Effects of actions (similar to what has been done in the part on reasoning about actions/planning).

```
holds(light_on, T+1):-       time(T),
   occ(turn_on, T), holds(neg(ab(bulb)), T).

holds(neg(light_on), T+1):-  time(T), occ(turn_off, T).

holds(ab(bulb), T+1):-       time(T), occ(break(bulb), T).

holds(neg(light_on), T):-    time(T), holds(ab(bulb), T).

% inertial axiom

holds(F, T+1):-  time(T), literal(F), contrary(F, G),
   holds(F, T), not holds(G, T+1).
```

Elements of Π – Part 3

Satisfying all the observations

```
% specifying the set of observations situation order

prec(s0,s1).   prec(s1,s2).   prec(s2,s3).

% fluent observations

at(neg(light_on), s0).    at(neg(ab(bulb)), s0).    at(light_on, s1).
at(neg(light_on), s2).    at(neg(light_on), s3).

% action observations

between(s0,s1,turn_on).   between(s2,s3,turn_on).
```

Elements of Π – Part 4

Satisfying all the observations

```
% generating situation order each situation happens at a time moment
1 {happens(S, T): time(T) } 1 :- sit(S).

% s0 is always at the time moment 0
:- time(T), happens(s0,T), T > 0.
:- time(T), happens(s3,T), T < length.

:- time(T1), time(T2), sit(S1), sit(S2),
   happens(S1,T1), happens(S2,T2), prec(S1,S2), T1>T2.

:- time(T1), time(T2), sit(S1), sit(S2), action(A),
   happens(S1,T1), happens(S2,T2), between(S1,S2,A), neq(T2,T1+1).

% generating action occurrences
{occ(A, T): action(A)} 1 :- time(T), T < length.
```

Elements of Π – Part 4

Satisfying all the observations

```
%  fluent observations
holds(F, T):-  time(T), literal(F), sit(S), at(F, S),  happens(S, T).

% constraints, making sure that action occurrences happen
% as they are observed

:- time(T1), time(T2), action(A), action(A1),
   between(S1,S2,A), happens(S1,T1), occ(A1,T1), neq(A1, A).

occ(A, T1):- time(T1), action(A), between(S1,S2,A), happens(S1,T1).

:- time(T), action(A1), action(A2), neq(A1,A2), occ(A1, T), occ(A2, T).

% constraints that eliminate inconsistency model
:- time(T), fluent(F), holds(F, T), holds(neg(F), T).
```

```
lparse -c length=5 bulb | smodels
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
occ(turn_on,0)
occ(turn_off,1)
occ(turn_on,2)
occ(break(bulb),3)
occ(turn_on,4)
happens(s3,5) happens(s2,4) happens(s1,1) happens(s0,0)
prec(s2,s3) prec(s1,s2) prec(s0,s1)
```

Other Applications – References

- E. Erdem, V. Lifschitz and M. Wong, Wire routing and satisfiability planning, in Proc. CL-2000, 2000.
- E.Erdem, V. Lifschitz and D. Ringe, Temporal phylogenetic networks and logic programming, Theory and Practice of Logic Programming.
- T. Eiter: Data Integration and Answer Set Programming. LPNMR 2005: 13-25
- E. Erdem, V. Lifschitz, L. Nakhleh and D. Ringe, Reconstructing the evolutionary history of Indo-European l anguages using answer set programming, PADL-2003.
- M. Balduccini and M. Gelfond: Diagnostic Reasoning with A-Prolog, TPLP, 3(4-5):425-461, 2003.
- M.Balduccini, M.Gelfond, M.Nogueira, R.Watson,M.Barry: An A-Prolog decision support system. for the space shuttle, AAAI Spring 2001 Symposium, Mar 2001
- K. Heljanko and I. Niemelä. Bounded LTL Model Checking with Stable Models, TPLP, 3 (4-5): 519-550, 2003.

CURRENT ISSUES

Theoretical Issues

- *Language development:* extensions to allow new language constructors that are useful for knowledge representation. For example, there is a growing interest in developing logic programs with
  - aggregates,
  - ordered disjunction, and
  - constraint atoms

  This includes the introduction of new constructors and the semantics of the programs with these constructors. Some references:
  - N. Pelov, Semantics of logic programs with aggregates, Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, April, 2004.
  - W. Faber, N. Leone, and G. Pfeifer, Recursive aggregates in disjunctive logic programs: Semantics and complexity. Proceedings of JELIA 04.
  - L. Liu and M. Truszczynski, Properties of Programs with Monotone and Convex Constraints. Proceedings of AAAI-05.

  - G. Brewka, I. Niemelä, and T. Syrjänen. Logic Programs with Ordered Disjunction. To appear in Computational Intelligence, 20(2), 333-357, 2004.
  - T. C. Son, E. Pontelli, and I. Elkabani: A Translational Semantics for Aggregates in Logic Programming, NMSU-CS-2005-005.

- *New characterization of answer sets*: this includes the study of the semantics here-and-there, the characterization of answer sets using rule graphs, or the relationship between graph coloring and answer sets. Some references:

  - V. Lifschitz, D. Pearce and A. Valverde, Strongly equivalent logic programs, ACM Transactions on Computational Logic, Vol. 2, 2001, pp. 526-541.
  - P. Ferraris and V. Lifschitz, Weight constraints as nested expressions, Theory and Practice of Logic Programming, Vol. 5, 2005, pp. 45–74.
  - K. Konczak, T. Linke, T. Schaub: Graphs and Colorings for Answer Set Programming: Abridged Report. LPNMR 2004: 127-140
  - S. Costantini, O. M. D'Antona, A. Provetti: On the equivalence and range of applicability of graph-based representations of logic programs. Inf. Process. Lett. 84(5): 241-249 (2002)

- *Study of properties of programs*: building block results for normal logic programs as well as programs within the new language (e.g. with new language constructor) are

studied. In particular, theorems

- on the conditions for equivalence between logic programs (strong/weak equivalent)
- that can easy the process of proving properties of a programs (e.g. correctness of programs),
- that provide new insight in the definition of answer set semantics,
- that open the door for new way to compute answer sets,

are sought.

More references

- F. Lin and X. Zhao, On odd and even cycles in normal logic programs In Proc. of AAAI-04. pp. 80-85
- F. Lin, Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic In Proc. of KR-02.
- F. Lin, Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers
- J. Lee, A Model-Theoretic Counterpart of Loop Formulas, Proc IJCAI-05.
- M. Gebser and T. Schaub, Loops: Relevant or Redundant, LPNMR-05.
- F. Lin, Y. Chen, Discovering Classes of Strongly Equivalent Logic Programs, IJCAI-05.

Implementation Issues

- *Development of answer set solvers*: Current answer set solvers have restrictions on the type of atoms that can be considered (e.g., ground programs, no recursive aggregates, etc.) Providing new answer set solvers that can deal with more expressive programs is the goal of developers of new answer set solvers. Some attempts can be found in
  - T. Eiter, M. Fink, H. Tompits, S. Woltran: Strong and Uniform Equivalence in Answer-Set Programming: Characterizations and Complexity Results for the Non-Ground Case. AAAI 2005: 695-700.
  - T. Eiter, W. Faber, M. Fink, G. Pfeifer, S. Woltran: Complexity of Answer Set Checking and Bounded Predicate Arities for Non-ground Answer Set Programming. Answer Set Programming 2003
  - T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, S. Perri, G. Pfeifer: System Description: DLV with Aggregates. LPNMR 2004: 326-330
  - I. Elkabani, E. Pontelli, T. C. Son: Smodels^A - A System for Computing Answer Sets of Logic Programs with Aggregates. LPNMR 2005: 427-431

  - E. Pontelli, T. C. Son, I. Elkabani: Smodels with CLP? A Treatment of Aggregates in ASP. LPNMR 2004: 356-360.

- *Development of tools and programming environment*: References on the integration of answer set programming into other languages and an interactive environment for answer set programming can be found in

  - O. El-Khatib, E. Pontelli, T. C. Son: ASP-PROLOG: A System for Reasoning about Answer Set Programs in Prolog. PADL 2004: 148-162
  - O. El-Khatib, E. Pontelli, T. C. Son: Justification and debugging of answer set programs in ASP. AADEBUG 2005: 49-58