Answer Set Programming

Theory and Applications

Tran Cao Son

Department of Computer Science
New Mexico State University
MSC CS, PO Box 30001
Las Cruces, New Mexico 88003

Fall 2018/AI 2

Outline

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.edu/~mgelfond`.
- Torsten Schaub, available at `http://potassco.sourceforge.net/teaching.html`

Introduction — Answer Set Programming

Answer set programming is a new programming paradigm. It was introduced in the late 90's and manages to attract 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.

Outline

Rules and Constraints

r : b_1 *or* $\ldots$ *or* $b_m \leftarrow a_1, \ldots, a_n$, **not** $a_{n+1}, \ldots,$ **not** a_{n+k}

- a_i, b_j: atom of a language $\mathcal{L}$ ($\mathcal{L}$ can either be propositional or first order)
- **not** a: a *negation-as-failure atom* (naf-atom).

Rules and Constraints

r : b_1 *or* $\ldots$ *or* $b_m \leftarrow a_1, \ldots, a_n$, **not** $a_{n+1}, \ldots,$ **not** a_{n+k}

- a_i, b_j: atom of a language $\mathcal{L}$ ($\mathcal{L}$ can either be propositional or first order)
- **not** a: a *negation-as-failure atom* (naf-atom).

Reading 1

If $a_1, \ldots, a_n$ are true and none of $a_{n+1}, \ldots, a_{n+k}$ can be proven to be true then at least one of $b_1, \ldots, b_m$ must be true.

Rules and Constraints

r : b_1 *or* $\ldots$ *or* $b_m \leftarrow a_1, \ldots, a_n$, **not** $a_{n+1}, \ldots,$ **not** a_{n+k}

- a_i, b_j: atom of a language $\mathcal{L}$ ($\mathcal{L}$ can either be propositional or first order)
- **not** a: a *negation-as-failure atom* (naf-atom).

Reading 1

If $a_1, \ldots, a_n$ are true and none of $a_{n+1}, \ldots, a_{n+k}$ can be proven to be true then at least one of $b_1, \ldots, b_m$ must be true.

Reading 2

If $a_1, \ldots, a_n$ are believed to be true and there is no reason to believe that any of $a_{n+1}, \ldots, a_{n+k}$ is true then at least one of $b_1, \ldots, b_m$ must be true.

Notations

$$r: \quad \underbrace{b_1 \text{ or } \ldots \text{ or } b_m}_{head(r)} \leftarrow \underbrace{a_1, \ldots, a_n, \textbf{not } a_{n+1}, \ldots, \textbf{not } a_{n+k}}_{body(r)}$$

- $head(r) = \{b_1, \ldots, b_m\}$
- $pos(r) = \{a_1, \ldots, a_n\}$ (also: $body^+(r) = \{a_1, \ldots, a_n\}$)
- $neg(r) = \{a_{n+1}, \ldots, a_{n+k}\}$ (also: $body^-(r) = \{a_{n+1}, \ldots, a_{n+k}\}$)

Special cases

- $n = k = 0$: r encodes a fact;
- $k = 0$: r is a positive rule; and
- $m = 0$: r encodes a constraint.

Program

- Program: a set of rules.
- Herbrand universe: the set of ground terms constructed constructed from function symbols and constants occurring in the program. (U_π)
- Herbrand base: the set of ground atoms constructed from predicate symbols and ground terms from the Herbrand universe. (B_π)
- Rule with variables: shorthand for the collection of its ground instances. ($ground(r)$)
- Program with variables: collection of ground instances of its rules. ($ground(\pi)$)

$\mathcal{L}$ is a propositional language

$\mathcal{L}$: set of propositions such as p, q, r, a, b ...

$$P_1 = \begin{cases} a \leftarrow \\ b \leftarrow a, c \\ c \leftarrow a, p \\ c \leftarrow \end{cases} \qquad P_2 = \begin{cases} a \leftarrow \textbf{not } b \\ b \leftarrow \textbf{not } a, c \\ p \leftarrow a, \textbf{not } p \\ c \leftarrow \end{cases} \qquad P_3 = \begin{cases} a \leftarrow \\ b \leftarrow c \end{cases}$$

$\mathcal{L}$ is a first order language

$\mathcal{L}$ has one function symbol f (arity: 1) and one predicate symbol p (arity 1)

$$Q_1 = \{\ p(f(X)) \leftarrow p(X)$$

$$Q_2 = \{\ p(f(f(X)) \leftarrow p(f(X)), \textbf{not}\ p(X)$$

$$Q_3 = \left\{ \begin{array}{l} p(f(X)) \leftarrow \\ p(f(f(f(X)))) \leftarrow p(X) \end{array} \right.$$

Semantics: Positive Propositional Programs

For a program without **not** and every rule $m = 1$. So, every rule in P is of the form: $a \leftarrow a_1, \ldots, a_n$

Definition

For a positive program P,

$$T_P(X) = \{a \mid \exists(a \leftarrow a_1, \ldots, a_n) \in P.[\forall i.(a_i \in X)]\}$$

Observations

- every fact in P belongs to $T_P(X)$ for every X
- If $X \subseteq Y$ then $T_P(X) \subseteq T_P(Y)$
- $\emptyset \subseteq T_P(\emptyset) \subseteq T_P(T_P(\emptyset)) \subseteq \ldots \subseteq T_P^n(\emptyset) \subseteq$ and $T^n(\emptyset)$ for $n \to \infty$ converges to $lfp(T_P)$

Computing T_P: Example 1

$\mathcal{L}$: set of propositions such as p, q, r, a, b ...

$$P_1 = \begin{cases} a \leftarrow \\ b \leftarrow a, c \\ c \leftarrow a, p \\ c \leftarrow \end{cases}$$

$T_{P_1}(\emptyset) = \{a, c\}$
$T^2_{P_1}(\emptyset) = T_{P_1}(T_{P_1}(\emptyset)) = T_{P_1}((\{a, c\}) = \{a, c, b\}$
$T^3_{P_1}(\emptyset) = T_{P_1}(T^2_{P_1}(\emptyset)) = T_{P_1}(\{a, c, b\}) = \{a, c, b\} = lfp(T_{P_1})$

Computing T_P: Example 2

$\mathcal{L}$: set of propositions such as p, q, r, a, b ...

$$P_2 = \begin{cases} a \leftarrow b \\ b \leftarrow a, c \\ p \leftarrow a, p \\ c \leftarrow \end{cases}$$

$T_{P_2}(\emptyset) = \{c\}$
$T^2_{P_2}(\emptyset) = T_{P_2}(T_{P_2}(\emptyset)) = T_{P_2}(\{c\}) = \{c\} = lfp(T_{P_2})$

Computing T_P: Example 3

$$P_3 = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow c \end{array} \right.$$

$T_{P_3}(\emptyset) = \{a\}$
$T^2_{P_3}(\emptyset) = T_{P_3}(T_{P_3}(\emptyset)) = T_{P_3}(\{a\}) = \{a\} = lfp(T_{P_3})$

Computing T_P: Example 4 and 5

$$P_4 = \begin{cases} a \leftarrow b \\ b \leftarrow a \end{cases}$$

$T_{P_4}(\emptyset) = \emptyset = lfp(T_{P_4})$

$$P_5 = \begin{cases} a \leftarrow \\ b \leftarrow a, b \end{cases}$$

$T_{P_5}(\emptyset) = \{a\}$
$T^2_{P_5}(\emptyset) = T_{P_5}(T_{P_5}(\emptyset)) = T_{P_5}(\{a\}) = \{a\} = lfp(T_{P_5})$

Terminologies – many borrowed 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.

FOL, 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.
  - $U_{\mathcal{L}_1} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.
  - $B_{\mathcal{L}_1} = \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$.

Programs with FOL Atoms

$r: \quad b_1 \text{ or } \ldots \text{ or } b_m \leftarrow a_1, \ldots, a_n, \textbf{not } a_{n+1}, \ldots, \textbf{not } a_{n+k}$

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

Rules with Variables

$ground(r, \mathcal{L})$: the set of all rules obtained from r by all possible substitution of elements of $U_{\mathcal{L}}$ for the variables in r.

Example

Consider the rule "$p(f(X)) \leftarrow p(X)$." and the language $\mathcal{L}_1$ (with variables X, Y; constants a, b; function symbol f of arity 1; and predicate symbol p of arity 1). Then $ground(r, \mathcal{L}_1)$ will consist of the following rules:

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

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.
- For a program Π:
  - $ground(\Pi, \mathcal{L}) = \bigcup_{r \in \Pi} ground(r, \mathcal{L})$
  - $\mathcal{L}_\Pi$: 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.
  - $ground(\Pi) = \bigcup_{r \in \Pi} ground(r, \mathcal{L}_\Pi)$.

Example 2

- Π:

  $p(a).\quad p(b).\quad p(c).$

  $p(f(X)) \leftarrow p(X).$

- $ground(\Pi)$:

  $p(a) \leftarrow.$

  $p(b) \leftarrow .$

  $p(c) \leftarrow .$

  $p(f(a)) \leftarrow p(a).$

  $p(f(b)) \leftarrow p(b).$

  $p(f(c)) \leftarrow p(c).$

  $p(f(f(a))) \leftarrow p(f(a)).$

  $p(f(f(b))) \leftarrow p(f(b)).$

  $p(f(f(c))) \leftarrow p(f(c)).$

  $\vdots$

  $p(f^{k+1}(x)) \leftarrow p(f^k(x)).$ for $x \in \{a, b, c\}$

Herbrand Interpretation I

Definition

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 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); two predicate symbols: p (arity 1), q (arity 2) and r (arity 1); variables X, Y; and a (added) constant a.

$U_\Pi = U_{\mathcal{L}_\Pi} = \{a, f(a), g(a), f(f(a)), g(f(a)), g(f(a)),$
$\quad g(g(a)), f(f(f(a))), g(f(f(g(a)))), \ldots\}$

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

Definition (Herbrand Interpretation)

A Herbrand interpretation of a program Π is a set of atoms from its Herbrand base.

Semantics – Positive Programs without Constraints I

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 "$a_0 \leftarrow a_1, \ldots, a_n$" in $ground(\Pi)$, $a_1, \ldots, a_n$ are true with respect to I (or $\{a_1, \ldots, a_n\} \subseteq I$) then a_0 is also true with respect to I (or $a_0 \in I$).

Definition

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.

$$T_\Pi(X) = \{a \mid a \in B_\Pi, \text{ there exists a rule } a \leftarrow a_1, \ldots, a_n \text{in } \Pi \text{ s. t. } a_i \in X\} \quad (1)$$

Semantics – Positive Programs without Constraints II

Note: By $a_0 \leftarrow a_1, \ldots, a_n$ in $ground(\Pi)$ we mean there exists a rule $b_0 \leftarrow b_1, \ldots, b_n$ in Π (that might contain variables) and a ground substitution σ such that $a_0 = b_0\sigma$ and $a_i = b_i\sigma$.

Remark

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

$$U_\Pi = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\} = \{f^i(a) \mid i = 0, 1, \ldots, \}$$

and

$$B_\Pi = \{q(f^i(a)), p(f^i(a)) \mid i = 0, \ldots, \}$$

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

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

Theorem

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

More Examples I

- 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:
  $X_0 = T_{\Pi_1}(\emptyset) =$
  $\{r(a), r(f(a)), r(g(a)), r(f(f(a))), r(g(f(a))), r(f(g(a))), ...\}$
  $X_1 = T_{\Pi_1}(X_0) = X_0$
  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, $lfp(\Pi_2) = \emptyset$.

More Examples II

- 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 $a \notin M_\Pi$.

Example

Let

$$\Pi = \left\{ \begin{array}{lcl} p(f(X)) & \leftarrow & p(X). \\ q(a) & \leftarrow & p(X). \\ p(b). & & \end{array} \right.$$

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 I

- 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.

Entailment – Another Example II

- 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.

Entailment – Another Example III

Remark

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 I

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

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

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

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

Semantics – General Logic Programs without Constraints II

Remark

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

Remark

Π^X *is a positive program.*

Definition

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

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

Detailed Computation I

- 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 $\{b\}$

$S_1 = \emptyset$	$S_2 = \{a\}$	$S_3 = \{b\}$	$S_4 = \{a, b\}$
$\Pi_2^{S_1}$: $a \leftarrow$ $b \leftarrow$	$\Pi_2^{S_2}$: $a \leftarrow$	$\Pi_2^{S_3}$: $b \leftarrow$	$\Pi_2^{S_4}$:
$M_{\Pi_2^{S_1}} = \{a, b\}$	$M_{\Pi_2^{S_2}} = \{a\}$	$M_{\Pi_2^{S_3}} = \{b\}$	$M_{\Pi_2^{S_4}} = \emptyset$
$M_{\Pi_2^{S_1}} \neq S_1$	$M_{\Pi_2^{S_2}} = S_2$	$M_{\Pi_2^{S_3}} = S_3$	$M_{\Pi_2^{S_4}} \neq S_4$
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.

Detailed Computation II

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

Detailed Computation III

In computing answer sets, the following theorem is useful:

Theorem

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* $ground(P) \setminus \{r\} \cup \{r'\}$*.*

More Examples

Remark

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 I

- 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

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.*

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

Entailment w.r.t. Answer Set Semantics II

- $\Pi_3 = \{p \leftarrow a. \quad a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$
  The program has two answer sets: $\{a, p\}$ and $\{b\}$. Everything is unknown.
- $\Pi_4 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } c. \quad d \leftarrow .\}$
  Answer sets: $\{d, b\}$. $\Pi_4 \models b$; $\Pi_4 \models \neg a$; etc.
- $\Pi_5 = \{p \leftarrow \textbf{not } p.\}$
  No answer set. p is unknown.
- $\Pi_6 = \{p \leftarrow \textbf{not } p, d. \quad r \leftarrow \textbf{not } d. \quad d \leftarrow \textbf{not } r.\}$
  Answer set $\{r\}$. $\Pi_6 \models r$; $\Pi_6 \models \neg p$; etc.
- $\Pi_7 = \{p \leftarrow \textbf{not } a. \quad p \leftarrow \textbf{not } b. \quad a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$
  Two answer sets: $\{p, a\}$ and $\{p, b\}$. So, $\Pi_7 \models p$ but $\Pi_7 \not\models a$ and $\Pi_7 \not\models \neg a$? (likewise b).
- $\Pi_8 = \{q \leftarrow \textbf{not } r. \quad r \leftarrow \textbf{not } q. \quad p \leftarrow \textbf{not } p. \quad p \leftarrow \textbf{not } r.\}$
  One stable model: $\{p, q\}$. So, $\Pi_7 \models p$ and $\Pi_7 \models q$.

Entailment w.r.t. Answer Set Semantics III

Brave Reasoning vs. Skeptical Reasoning The entailment defined earlier required that $\Pi \models a$ iff a belongs to every answer set of Π. This is termed as *skeptical reasoning* w.r.t. answer set semantics.

Brave reasoning is an alternative that can be useful in different situations. Brave reasoning relaxes the notion of entailment as follows: for a program Π and an atom a, Π *bravely entails* a (resp. $\neg a$), denoted by $\Pi \models^b a$ (resp. $\Pi \models^b \neg a$), if $a \in S$ (resp. $a \notin S$) for **some answer set** S of Π.

Brave reasoning is the semantics underlying answer set programming.

Computation of Answer Sets I

Consider $\Pi_3 = \{p \leftarrow a. \quad a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$
The Herbrand base of this program is $\{p, a, b\}$. So, this program has eight possible answer sets. Each answer set is a subset of $\{p, a, b\}$. Let us consider each of them.

1. $X_0 = \emptyset$. Computing $\Pi_3^{X_0}$, we have that

$$\Pi_3^{X_0} = \begin{cases} p \leftarrow a \\ a \leftarrow \\ b \leftarrow \end{cases}$$

The minimal model of this program is: $M_{\Pi_3^{X_0}} = \{p, a, b\}$.
$M_{\Pi_3^{X_0}} = \{p, a, b\} \neq X_0$. This implies that X_0 is not an answer set.

Computation of Answer Sets II

2. $X_1 = \{p\}$. Computing $\Pi_3^{X_1}$, we have that

$$\Pi_3^{X_1} = \begin{cases} p \leftarrow a \\ a \leftarrow \\ b \leftarrow \end{cases}$$

The minimal model of this program is: $M_{\Pi_3^{X_1}} = \{p, a, b\}$.
$M_{\Pi_3^{X_1}} = \{p, a, b\} \neq X_1$. This implies that X_1 is not an answer set.

3. $X_2 = \{a\}$. Computing $\Pi_3^{X_2}$, we have that

$$\Pi_3^{X_2} = \begin{cases} p \leftarrow a \\ a \leftarrow \end{cases}$$

The minimal model of this program is: $M_{\Pi_3^{X_2}} = \{p, a\}$.
$M_{\Pi_3^{X_2}} = \{p, a\} \neq X_2$. This implies that X_2 is not an answer set.

Computation of Answer Sets III

4. $X_3 = \{b\}$. Computing $\Pi_3^{X_3}$, we have that

$$\Pi_3^{X_3} = \begin{cases} p \leftarrow a \\ b \leftarrow \end{cases}$$

The minimal model of this program is: $M_{\Pi_3^{X_3}} = \{b\}$.
$M_{\Pi_3^{X_3}} = \{b\} = X_3$. This implies that X_3 is an answer set.

5. $X_4 = \{p, a\}$. Computing $\Pi_3^{X_4}$, we have that

$$\Pi_3^{X_4} = \begin{cases} p \leftarrow a \\ a \leftarrow \end{cases}$$

The minimal model of this program is: $M_{\Pi_3^{X_4}} = \{p, a\}$.
$M_{\Pi_3^{X_4}} = \{p, a\} = X_4$. This implies that X_4 is an answer set.

Computation of Answer Sets IV

6. $X_5 = \{p, b\}$. Computing $\Pi_3^{X_5}$, we have that

$$\Pi_3^{X_5} = \left\{ \begin{array}{l} p \leftarrow a \\ b \leftarrow \end{array} \right.$$

The minimal model of this program is: $M_{\Pi_3^{X_5}} = \{b\}$.
$M_{\Pi_3^{X_5}} = \{b\} \neq X_5$. This implies that X_5 is not an answer set.

7. $X_6 = \{a, b\}$. Computing $\Pi_3^{X_6}$, we have that

$$\Pi_3^{X_6} = \left\{ \begin{array}{l} p \leftarrow a \end{array} \right.$$

The minimal model of this program is: $M_{\Pi_3^{X_6}} = \emptyset$.
$M_{\Pi_3^{X_6}} = \emptyset \neq X_6$. This implies that X_6 is not an answer set.

Computation of Answer Sets V

8. $X_7 = \{p, a, b\}$. Computing $\Pi_3^{X_7}$, we have that

$$\Pi_3^{X_7} = \{\ p \leftarrow a$$

The minimal model of this program is: $M_{\Pi_3^{X_7}} = \emptyset$.

$M_{\Pi_3^{X_7}} = \emptyset \neq X_7$. This implies that X_7 is not an answer set.

So, the program has two answer sets: $\{b\}$ and $\{p, a\}$.
Notice that in the computation, I did not detail how the minimal model of a positive program is computed. One should notice the difference between minimal model of the program $\Pi_3^{X_5}$ and $\Pi_3^{X_5}$ and know the reason why it is so.

Further intuitions behind the semantics I

- 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 $1 \leq i \leq m$.

Further intuitions behind the semantics II

- Note that (ii) above implies that S is supported by Π.
- It is known that if S is an answer set of Π then
  1. S must be closed under Π and
  2. S must be supported by Π.

The above notions are useful for the computation of answer sets. They allow us to eliminate possible answer sets quickly. Let us look at our computation of the answer sets of Π_3.

1. Let us consider $X_0 = \emptyset$. Consider the rule $a \leftarrow$ **not** b in Π_3. We have that the set of positive atoms in the body of this rule is empty (i.e., the set $\{a_1, \ldots, a_m\}$ in the definition of closedness) and the set of negative atoms in the body of this rule is $\{b\}$ and $\{b\} \cap X_0 = \emptyset$. This means that X_0 violates the closedness condition, i.e., X_0 is not closed under the rule $a \leftarrow$ **not** b of Π_3 and hence it is not closed under Π_3. As such, X_0 cannot be an answer set of Π_3.

Further intuitions behind the semantics III

2. Let us consider $X_7 = \{p, a, b\}t$. Consider the atom $a \in X_7$. The only rule in Π_3 whose head is a is the rule $a \leftarrow$ **not** b. We have that the set of positive atoms in the body of this rule is empty (i.e., the set $\{a_1, \ldots, a_m\}$ in the definition of supportedness) and the set of negative atoms in the body of this rule is $\{b\}$ and $\{b\} \cap X_7 \neq \emptyset$. This means that a has no rule to support it in Π_3, i.e., X_7 is not supported by Π_3. As such, X_7 cannot be an answer set of Π_3.

Answer Sets of Programs with Constraints I

For a set of ground atoms S and a constraint c of the form

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

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

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 $\mathit{ground}(\Pi_C)$.

Answer Sets of Programs with Constraints II

$\Pi_2 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a.\}$: two answer sets $\{a\}$ and $\{b\}$.

- $R_1 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a. \quad \leftarrow \textbf{not } a.\}$ has only one answer set $\{a\}$.
- $R_2 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a. \quad \leftarrow \textbf{not } a, \textbf{not } b.\}$ has again two answer sets $\{a\}$ and $\{b\}$.
- $R_3 = \{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a. \quad p \leftarrow a. \quad \leftarrow \textbf{not } p.\}$ has only one answer set $\{b\}$.

Disjunctive Logic Programs I

We define the general logic program rule as

$$b_1 \textit{ or } \ldots \textit{ or } b_m \leftarrow a_1, \ldots, a_n, \textbf{not } a_{n+1}, \ldots, \textbf{not } a_{n+k}$$

This rule is often called a disjunctive logic program rule (or simply disjunction rule). A rule of this form says that whenever $a_1, \ldots, a_n$ are true and $a_{n+1}, \ldots, a_{n+k}$ are false then at least one of $b_1, \ldots, b_m$ must be true. There are several situations where disjunctive logic programs are necessary since disjunctive logic program (DLP) is strictly more expressive than normal logic program.
An example is the map coloring problem:

$$\textit{colored}(N, \textit{red}) \textit{ or } \textit{colored}(N, \textit{blue}) \textit{ or } \textit{colored}(N, \textit{yellow}) \leftarrow \textit{node}(N).$$

Disjunctive Logic Programs II

Semantics of DLP is also defined by answer sets. The definition also makes use of the Gelfond-Lifschitz reduct: given a DLP program π and a set of atoms X, π^X is the result of the Gelfond-Lifschitz reduct of π over X.

The program π^X does not contain negation-as-failure atoms.

A set of atoms X is an answer set of π if and only if X is a minimal set of atoms satisfying π^X.

Examples

- $\pi_1 = \{a \text{ or } b \leftarrow c. \quad a \leftarrow c. \quad b \leftarrow c. \quad c.\}$

  π_1 is a positive program. It has only one answer set: $\{c, a, b\}$.

- $\pi_2 = \{a \text{ or } b \leftarrow c. \quad c.\}$

  π_2 is also a positive program. It has two answer sets: $\{c, a\}$ and $\{c, b\}$.

Disjunctive Logic Programs III

It is useful for representing incomplete knowledge. For example, consider the story

John's Injury

John ran and broke one of his arms. We do not know whether his left or right arm was broken but we are sure that only one was broken.

This story can be represented by the disjunctive program consisting of the fact:

$$broken(left)\ or\ broken(right) \leftarrow$$

Outline

Running `clingo`

- Download from `https://potassco.org`, set up, etc.
- Run `clingo` <params>
- Set of examples
- Let solve some problems
- Some syntactical notes:
  - "$\leftarrow$" is replaced by ":-"
  - "*or*" is replaced by "|"

Graph Coloring I

Problem

Given a undirected graph G. Color each node of the graph by red, yellow, or blue so that no two adjacent nodes have the same color.

Approach

We will solve the problem by writing a logic program Π_G such that each answer set of Π_G gives us a solution to the problem. Furthermore, each solution of the problem corresponds to an answer set.

The program Π_G needs to contain information about the graph and then the definition of the problem. So,

- Graph representation:
  - The nodes: $node(1), \ldots, node(n)$.
  - The edges: $edge(i, j)$.
- Solution representation: use the predicate $colored(X, Y)$ - node X is assigned the color Y.

Graph Coloring II

- Generating the solutions: Each node is assigned one color. The three rules

$$\begin{aligned} \mathit{colored}(X, \mathit{red}) &\leftarrow \textbf{not}\ \mathit{colored}(X, \mathit{blue}), \textbf{not}\ \mathit{colored}(X, \mathit{yellow}) && (2)\\ \mathit{colored}(X, \mathit{blue}) &\leftarrow \textbf{not}\ \mathit{colored}(X, \mathit{red}), \textbf{not}\ \mathit{colored}(X, \mathit{yellow}). && (3)\\ \mathit{colored}(X, \mathit{yellow}) &\leftarrow \textbf{not}\ \mathit{colored}(X, \mathit{blue}), \textbf{not}\ \mathit{colored}(X, \mathit{red}). && (4) \end{aligned}$$

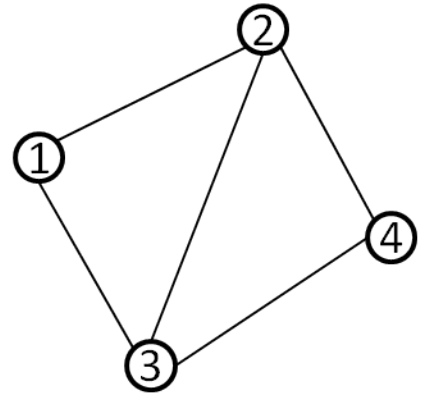
- Checking for a solution: needs to make sure that no edge connects two nodes of the same color. This can be represented by a constraint:

$$\leftarrow \mathit{edge}(X, Y), \mathit{colored}(X, C), \mathit{colored}(Y, C). \qquad (5)$$

Graph Coloring III

%% graphcolor.lp

%% 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

colored(X,red):- node(X), not colored(X,blue), not colored(X, yellow).
colored(X,blue):- node(X), not colored(X,red), not colored(X, yellow).
colored(X,yellow):- node(X), not colored(X,blue), not colored(X, red).

%% constraint checking

:- edge(X,Y), colored(X,C), colored(Y,C).

Try with

```
clingo graphcolor.lp
```
or
```
clingo 0 graphcolor.lp
```

and see the result. The atom node(X) in the rules with the head *colored*$(X, \_)$ is needed because the solver's "*unsafe variable*" error.

Graph Coloring IV

`graphcolor.lp` 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 `graphcolor1.lp`, for our example, this program contains the first three lines of `graphcolor.lp`); and (b) the second file contains other rules. (let us called this file `graphcolor2.lp` which contains the other rules of `graphcolor2.lp`).

```
Command line:  clingo graphcolor1.lp graphcolor2.lp
```

Correctness of `graphcolor.lp` I

Let us denote the program `graphcolor.lp` 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.

Correctness of `graphcolor.lp` II

We can prove the following theorems.

Theorem

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
$C = \{colored(k, red), colored(k, blue), colored(k, yellow)\}$, 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

$colored(k, red) \leftarrow node(k).$ (because of rule (2))
$colored(k, blue) \leftarrow node(k).$ (because of rule (3))
$colored(k, yellow) \leftarrow node(k).$ (because of rule (4))

Correctness of `graphcolor.lp` III

Because k is a node of G, we can easily see that $colored(k, red)$, $colored(k, yellow)$, and $colored(k, blue)$ 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 $colored(k, red)$ and $colored(k, blue)$; and it does not contain $colored(k, yellow)$. Then, Π_G^S will not contain any rule whose head is an atom belonging to C. (all the rules of the form (2)-(4) 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.

Correctness of `graphcolor.lp` IV

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 $\{colored(k, red), colored(k, blue), colored(k, yellow)\}$.
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 $colored(i, c_i) \in S$ 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 (5) for $edge(p, q)$, $colored(p, c_p)$, and $colored(q, c_q)$ 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 assumption is incorrect, i.e., we have proved that for every edge (i, j) of G, $c_i \neq c_j$. This

Correctness of `graphcolor.lp` V

shows that S corresponds to a solution of the 3-coloring problem for G.

Theorem

If the 3-coloring problem for G has a solution M then Π_G has an answer set corresponds 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

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

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:

Correctness of `graphcolor.lp` VI

- 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 $colored(i, c_i) \leftarrow node(i)$ which comes from one of the rules (2)-(4).
- for each edge (i,j), three constraints:

$$\leftarrow edge(i,j), colored(i, red), colored(j, red)$$
$$\leftarrow edge(i,j), colored(i, yellow), colored(j, yellow)$$
$$\leftarrow edge(i,j), colored(i, blue), colored(j, blue)$$

We need to show that S is the minimal model of Π_G^S that does not violate 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 $T_{\Pi_G^S}(S) = S$. (*)

Correctness of `graphcolor.lp` VII

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 $C \in \{red, blue, yellow\}$ such that S contains $colored(i, C)$ and $colored(j, C)$. 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.

Seating Arrangements I

Problem

Ann, John, Sally, and Mike often have lunch in the break room of the department. The room has two tables.

- Mike wants to be alone or with Ann.
- John does not care.
- Ann likes to be either with Mike and Sally or with Sally and John.
- Sally does not want to be alone.

Find a seat arrangement that can make everybody happy.

Seating Arrangements II

- Problem representation: $table(1)$ and $table(2)$; $person(mike)$, $person(sally)$, $person(ann)$, and $person(john)$.
- Result representation: $at(mike, 1)$, $at(mike, 2)$, etc.
- The rules:

$$at(X, 1) \leftarrow \textbf{not } at(X, 2).$$
$$at(X, 2) \leftarrow \textbf{not } at(X, 1).$$

- `clingo` and see "X is unsafe"
- add "what is X" to the rule: $person(X)$ or $table(X)$?
- run the command `clingo 0 seats.lp` and see the possible arrangements.
- add constraints to remove unwanted answer sets.

Syntactic Extensions of Logic Programming

Choice Atoms

colored(X,red):- node(X), not colored(X,blue), not colored(X, yellow).
colored(X,blue):- node(X), not colored(X,red), not colored(X, yellow).
colored(X,yellow):- node(X), not colored(X,blue), not colored(X, red).
replaced by

```
1 {colored(X, C) : is_color(C)} 1 :- node(X).
```

and a set of atoms

```
is_color(yellow).   is_color(red).   is_color(blue).
```

Choice atoms allow for a succinct representation. General form of choice atoms is

$$\mathbf{l}\,\{p_1, p_2, \ldots, p_k\}\,\mathbf{u}$$

where $0 \leq \mathbf{l} \leq \mathbf{u}$ are integers and p_i's are atoms. Expression of the from $\{p(\vec{X}) : q(\vec{Y})\}$ where all variables in $\vec{Y}$ appear in $\vec{X}$. A choice atom is true with respect to a set of atoms S if $\mathbf{l} \leq |\{p_i \mid p_i \in S\}| \leq \mathbf{u}$.

Syntactic Extensions of Logic Programming

Weighted Atoms

$\mathbf{l}\{l_0 = w_0, \ldots, l_k = w_k, \textbf{not } l_{k+1} = w_{k+1}, \ldots, \textbf{not } l_{k+n} = w_{k+n}\}\mathbf{u}$ where l_i's are atoms, w_i are integers, and $\mathbf{l} \leq \mathbf{u}$ are integers. This atom is true with respect to a set of literals S if

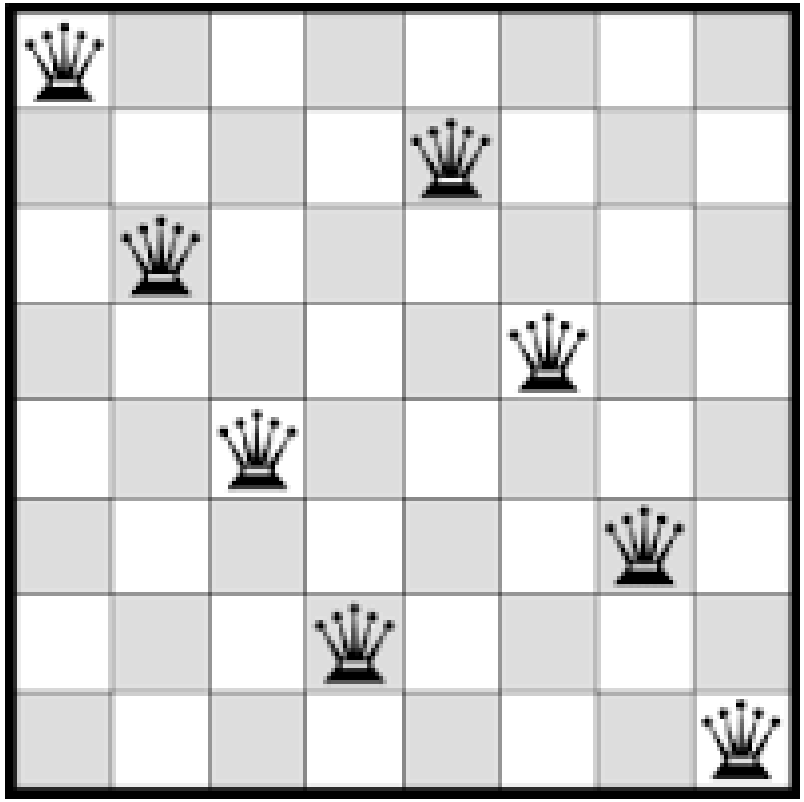
$$\mathbf{l} \leq \sum_{l_j \in S}^{0 \leq j \leq k} w_j + \sum_{l_j \notin S}^{k+1 \leq j \leq k+n} w_j \leq \mathbf{u}$$

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

Aggregates

$Sum(\Omega)$, $Count(\Omega)$, $Average(\Omega)$, $Min(\Omega)$, $Max(\Omega)$ where Ω denotes a multiset (e.g., $\{p(a, X) \mid X \in \{1, 2, 3\}\}$)

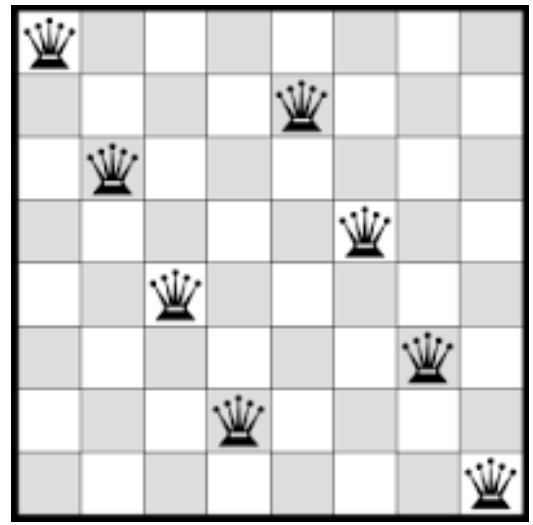
n-Queens



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 choice atom in the 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
$$\leftarrow cell(I1, J1), cell(I2, J2), |I1 - I2| = |J1 - J2|$$

Code

```
% 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) } 1:- col(I).
% two queens cannot be on the same row/column
:- col(I), row(J1), row(J2), J1!=J2, cell(I,J1), cell(I,J2).
:- row(J), col(I1), col(I2), 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), I1 - I2 == J1 - J2.
:- row(J1), row(J2), J1 > J2, col(I1), col(I2), I1 < I2, cell(I1,J1),
cell(I2,J2), I2 - I1 == J1 - J2.
Command line:  clingo -c n=input queens.lp
```

Sudoku

Facts : $c(1..9).\ r(1..9).\ num(1..9).$
$an(1,1,2).\ an(1,3,9).\ an(1,4,6)....$
$ar(1..3,1..3,1).\ ar(1..3,4..6,2).\ ar(1..3,7..9,3).$
$ar(4..6,1..3,4).\ ar(4..6,4..6,5).\ ar(4..6,7..9,6).$
$ar(7..9,1..3,7).\ ar(7..9,4..6,8).\ ar(7..9,7..9,9).$

2		9	6		5			
	5	7	9				1	
	1	3			7		6	9
5		1		7				4
			5		6			
9				8		2		5
4	9		1			3	5	
	6				3	9	4	
			8		4	1		6

Generation :
$1\{an(X,Y,N) : num(N)\}1 : -c(X), r(Y).$

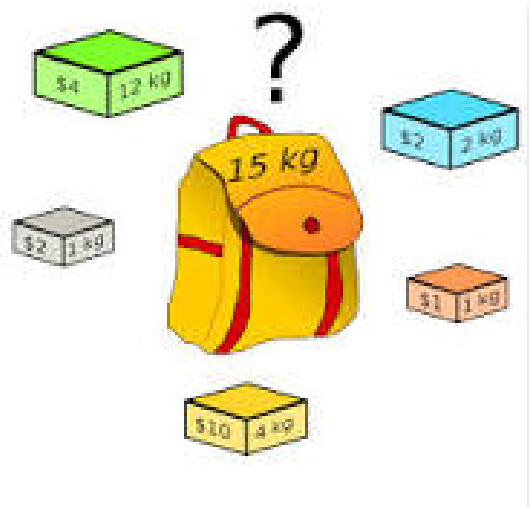
Check :
$: -c(Y), c(X), X \neq Y, r(R), an(R,X,N), an(R,Y,N).$
$: -r(Y), r(X), X \neq Y, c(C), an(Y,C,N), an(X,C,N).$
$: -ar(X,Y,Z), ar(P,Q,Z), X \neq P, an(X,Y,N), an(P,Q,N).$
$: -ar(X,Y,Z), ar(P,Q,Z), Y \neq Q, an(X,Y,N), an(P,Q,N).$

(c - column, r - row, num - number, an - answer, ar - area)

Knapsack Problem

Problem

There are several items. Each has some weight and value. There is a backpack that can only cary a maximal weight of M. Identify the set of items that should be put in the backpack so that the total value is maximal.



Example

Five items: i_1 (12kg, \$4), i_2 (2kg, \$2), i_3 (1kg, \$1), i_4 (4kg, \$10), i_5 (1kg, \$2).
Backpack: maximal 15 kg.

Knapsack Problem

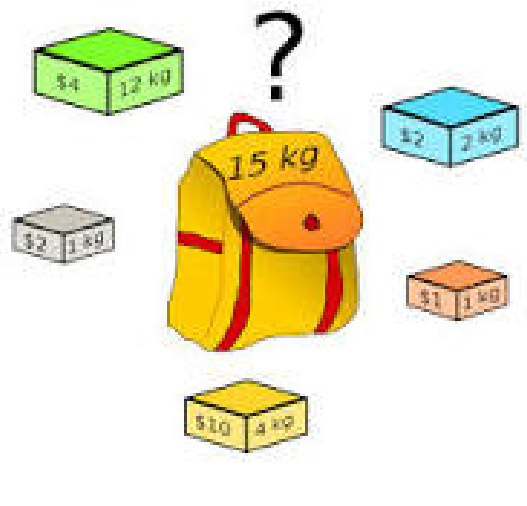
- Problem representation: items, values, weights
- Solution generation:
- Checking for solution:

Knapsack Problem

- Problem representation: items, values, weights
  item(1..n). value(1,4). weight(1,12). ...
- Solution generation:
  {in(I)} :- item(I).
- Checking for solution:
  Compute the total weight
  total_weight(W) :- W = #sum {N : in(I), weight(I,N)}.
  :- total_weight(W), W > 15.
  Compute total value
  total_value(V) :- V = #sum {VA : in(I), value(I, VA)}.
  #maximize {V : total_value(V)}.

Knapsack Problem: Code

```
item(1..5).
value(1,4). value(2,2).
value(3,1). value(4,10).
value(5,2).
weight(1,12). weight(2,2).
weight(3,1). weight(4,4).
weight(5,1).
{in(I)} :- item(I).
total_weight(W) :- W = #sum
{N : in(I), weight(I,N)}. :-
total_weight(W), W > 15.
total_value(V) :- V = #sum {VA
: in(I), value(I, VA)}.
#maximize {V : total_value(V)}.
```



Some Notes on Syntax used by clingo

```
item(1..5).
value(1,4). value(2,2). value(3,1). value(4,10). value(5,2).
weight(1,12). weight(2,2). weight(3,1). weight(4,4). weight(5,1).
{in(I)} :- item(I).
```

Old

```
total_weight(W) :- W = #sum {N : in(I), weight(I,N)}.
:- total_weight(W), W > 15.
total_value(V) :- V = #sum {VA : in(I), value(I, VA)}.
#maximize {V : total_value(V)}.
```

New

```
total_weight(W) :- W = #sum {N,I : in(I), weight(I,N)}.
:- total_weight(W), W > 15.
total_value(V) :- V = #sum {VA,I : in(I), value(I, VA)}.
#maximize {V : total_value(V)}.
```

General Syntax: **guide.pdf (2.0)**

$$s_1 \prec_1 \alpha \{t_1 : L_1;\ t_2 : L_2; \ldots;\ t_n : L_n\} \prec_2 s_2$$

where

- t_i and L_i are non-empty tuples of terms and atoms, respectively.
- α can either be $\#count$, $\#sum$, $\#min$, $\#max$, $\#sum+$ (sum only positive weights)
- $\prec_1$ and $\prec_2$: comparison operator ($=, \leq, \geq, <, >$, default is $\leq$)

Example

$\#$sum $\{$ 3 : bananas; 25 : cigars; 10 : broom $\} \leq 30$.
$\lhd$ if bananas, cigars, broom are true then it results in $38 \leq 30$ (false)
$\lhd$ if bananas, cigars are true (broom is false) it results in $28 \leq 30$ (true)
$\#$count $\{$ 42 : a; 42 : a; t : not b; t : not b $\} = 2$.
$\lhd$ if a and b are true then it results in $\{42\} = 2$ (false)
$\lhd$ if a is true and b is false then it results in $\{42;t\} = 2$ (true)

Knapsack Revisited

Set atom with variable

total_weight(W) :- W = #sum {N,I : in(I), weight(I,N)}.

The variables: I and N

This stands for

```
total_weight(W) :- W = #sum {   12,1 : in(1);
                                2,2 : in(2);
                                1,3 : in(3);
                                4,4 : in(4);
                                1,5 : in(5)    }.
```

K-clique

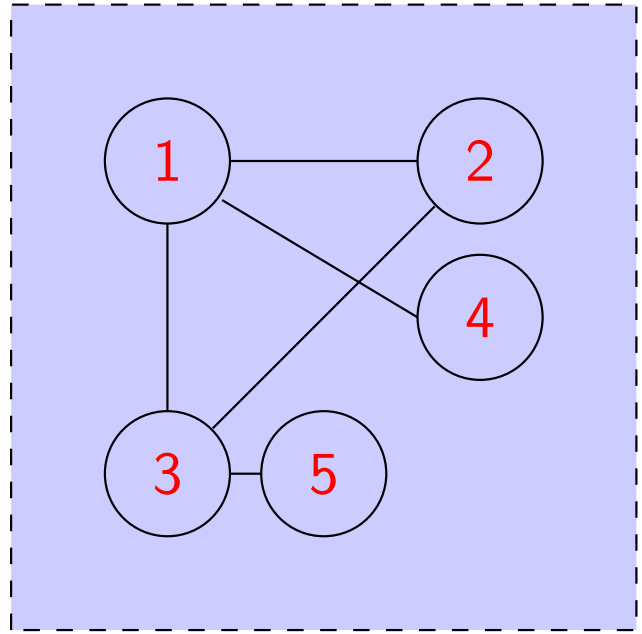
- **Problem**: Given a number k and a graph G. G has a clique of size k if there is a set of k different vertices (nodes) in G such that each pair of vertices from this set is connected through an egde.
- **Representation**:
  - Graph ($node()$ and $edge()$)
  - Clique ($clique(N)$) to say that node 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
- **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.

K-clique

- **Problem**: Given a number k and a graph G. 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. $\leftarrow clique(I), clique(J), I \neq J,$ **not** $edge(I, J),$ **not** $edge(J, I).$

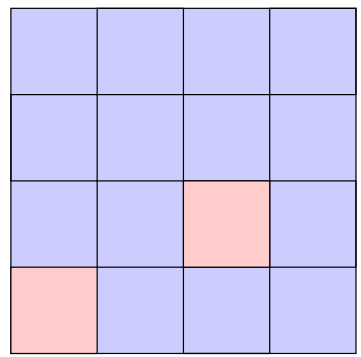
K-clique: Code

```
% Graph
node(1..5).
edge(1,2). edge(1,3). edge(2,3).
edge(1,4). edge(1,5).
edge(N1,N2):- edge(N2,N1).
% generating solution
k {clique(N):node(N)} k.
% checking solution
:- clique(N1), clique(N2), N1 !=N2,
      not edge(N1,N2).
```



Tile covering problem

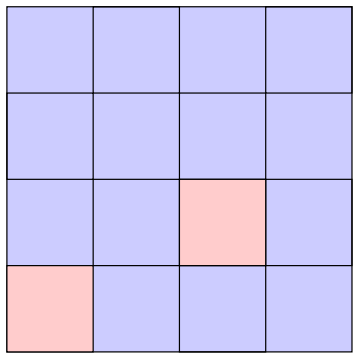
Problem: Given a slightly damaged chess board ($n \times n$). Find a covering of the board using m 1×2-tiles so that all the good squares on the board are covered. If no such covering exists, report there is no solution.

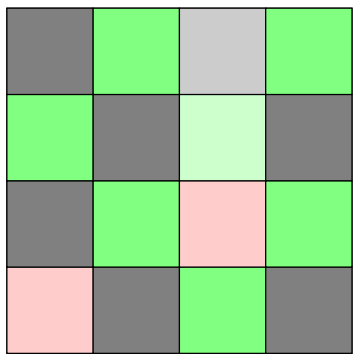


4 x 4 Problem

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.





4 x 4 Problem

Tile covering problem I

- **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

Tile covering problem II

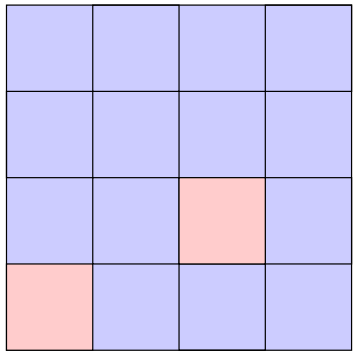
- **Generating a possible solution**: We assign each tile a number. Then we can assign a tile to a location. Since each tile covers two and only two cells, we can used the weighted-rule:

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

  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

Tile covering problem III

```
% Board representation
col(1..n). row(1..n). tile(1..ntiles).
bad(1,1). bad(3,2).
% generating solution
2 {cell(I,J,T) : col(I),row(J) } 2 :- tile(T).
% checking solution
% bad cell needs not ..
:- bad(I,J), cell(I,J,T).
% one cell one tile ....
:- cell(I,J,T1),cell(I,J,T2),T1!=T2.
% neightbor only ....
:- cell(I1,J1,T), cell(I2,J2,T), I1 - I2 > 1.
:- cell(I1,J1,T), cell(I2,J2,T), J1 - J2 > 1.
% no diagonal ...
:- cell(P,Q,T), cell(R,S,T), P!=R, Q!=S.
```

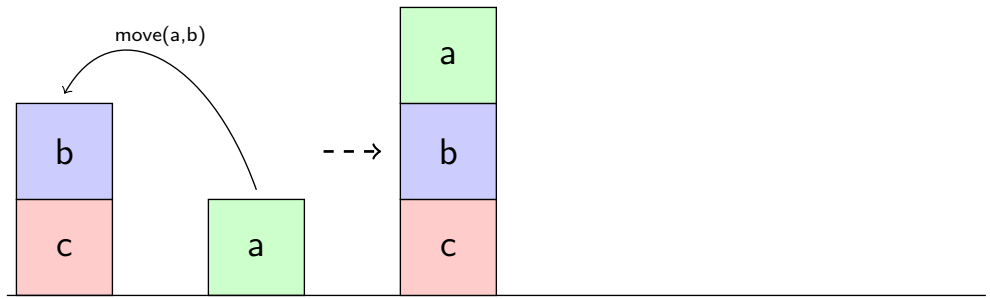


Outline

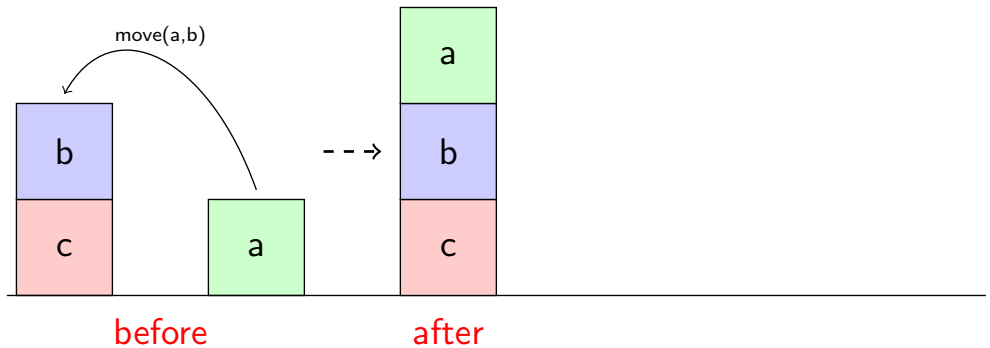
Block World Domain

Block World Domain

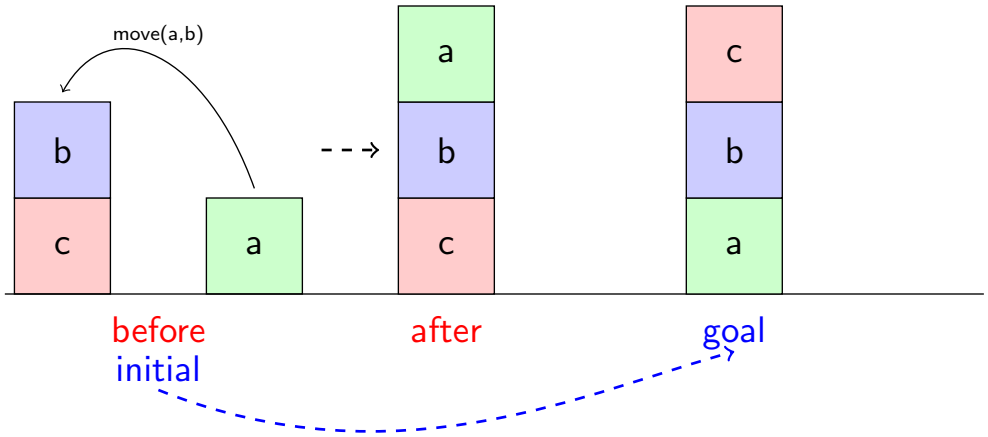
Block World Domain



Block World Domain



Block World Domain



Dynamic Domains

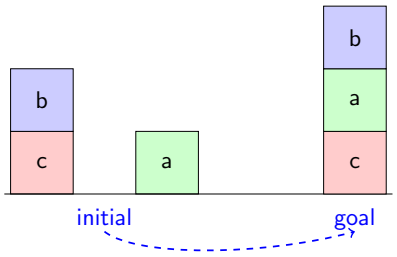
The state of the world continuously changes through the execution of actions (by some agent). State of the world can be described by a set of fluents (properties of the world whose truth values change over time). We are interested in

- predicting the state of the world after the execution of an action sequence;
- identifying ways to change the world to a pre-defined state;
- explaining the discrepancies that we observe.

Modeling the Block World Domain: Fluents and Actions

Fluents

- $on(X, Y)$: block X is on block Y
- $onTable(X)$: block X is on the table
- $clear(X)$: block X is clear
- $holding(X)$: the agent holds the block X
- $handEmpty$: the agent does not hold anything



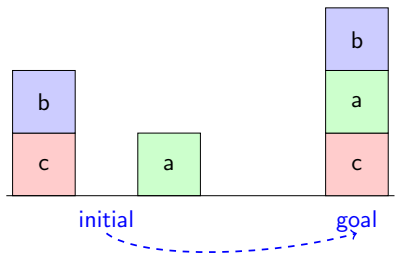
Actions

- $un/stack(X, Y)$: un/stack block X is on block Y
- $pickup/putdown(X)$: pickup or put down block X from/on to the table

Modeling the Block World Domain: Action Description I

Action Preconditions and Effects:

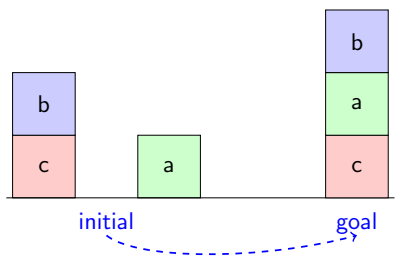
- *stack*(X, Y)
  - Precondition:
    - block Y is clear
    - the agent holds the block X
  - Effects:
    - X is clear
    - X is on Y
    - Y is no longer clear
    - the agent does not hold anything
- *putdown*(X)
  - Precondition:
    - the agent holds the block X
  - Effects:
    - X is clear
    - X is on the table
    - the agent does not hold anything



Modeling the Block World Domain: Action Description II

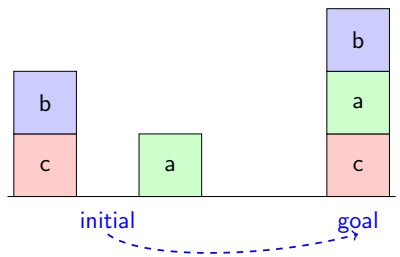
Action Preconditions and Effects:

- *unstack*(X, Y)
  - Precondition:
    - X is clear
    - X is on Y
    - the agent does not hold anything
  - Effects:
    - the agent holds the block X
    - Y becomes clear
    - X is not clear
- *pickup*(X)
  - Precondition:
    - X is clear
    - X is on the table
    - the agent does not hold anything
  - Effects:
    - the agent holds the block X
    - X is no longer on the table and not clear



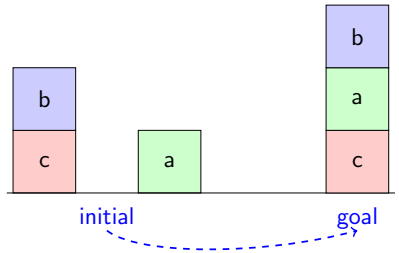
Block World Domain: Encoding I

```
% 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),X!=Y.
fluent(onTable(X)):- block(X).
fluent(clear(X)):- block(X).
fluent(holding(X)):- block(X).
fluent(handEmpty).
% Defining actions
action(stack(X,Y)):- block(X), block(Y), X!=Y.
action(unstack(X,Y)):- block(X),block(Y),X!=Y.
action(putdown(X)):- block(X).
action(pickup(X)):- block(X).
```



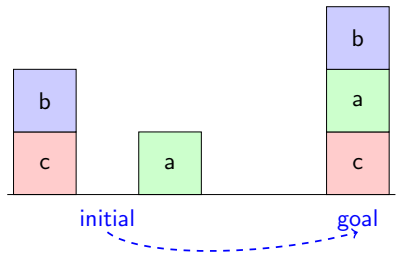
Block World Domain: Encoding II

```
% Action effects & precondition: pickup
excutable(pickup(X), T):- holds(clear(X), T),
   holds(handEmpty, T),
   holds(onTable(X), T).
:- occ(pickup(X),T), not executable(pickup(X),T).
 holds(neg(clear(X)), T+1):- occ(pickup(X), T).
 holds(neg(onTable(X)), T+1):- occ(pickup(X), T).
 holds(neg(handEmpty), T+1):- occ(pickup(X), T).
 holds(holding(X), T+1):- occ(pickup(X), T).
 % unstack
 executable(unstack(X,Y), T) :- holds(clear(X), T), holds(on(X,Y), T),
    holds(handEmpty, T).
 holds(neg(clear(X)), T+1):- occ(unstack(X,Y), T).
 holds(neg(on(X,Y)), T+1):- occ(unstack(X,Y), T).
 holds(holding(X), T+1):- occ(unstack(X,Y), T).
 holds(neg(handEmpty), T+1):- occ(pickup(X), T).
 holds(clear(Y), T+1):- occ(unstack(X,Y), T).
 :- occ(unstack(X,Y),T), not executable(unstack(X,Y),T).
```



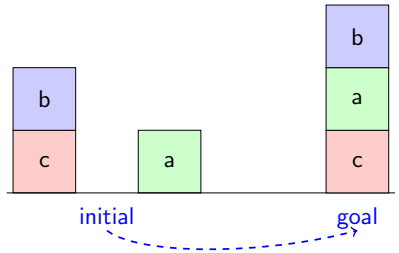
Block World Domain: Encoding III

```
% Action effects & precondition: putdown
executable(putdown(X), T):- holds(holding(X), T).
holds(clear(X), T+1):- occ(putdown(X), T).
holds(onTable(X), T+1):- occ(putdown(X), T).
holds(neg(holding(X)),T+1):- occ(putdown(X), T).
holds(handEmpty,T+1):- occ(putdown(X), T).
:- occ(putdown(X),T),not executable(putdown(X),T).
  % stack
  executable(stack(X,Y), T):- holds(clear(Y), T), holds(holding(X), T).
  holds(clear(X), T+1):- occ(stack(X,Y), T).
  holds(on(X,Y)), T+1):- occ(stack(X,Y), T).
  holds(neg(holding(X)), T+1):- occ(stack(X,Y), T).
  holds(neg(clear(Y)), T+1):- occ(stack(X,Y), T).
  holds(handempty,T+1) :- occ(stack(X,Y), T).
  :- occ(stack(X,Y),T), not executable(stack(X,Y),T).
```



Block World Domain: Initial State, Inertial Axiom

```
holds(on(b,c), 0).     holds(onTable(b), 0).
holds(onTable(a), 0).     holds(clear(a), 0).
holds(clear(b), 0).     holds(handEmpty,0).
holds(neg(F), 0) :- fluent(F), not holds(F,0).
```




```
% inertial axioms
holds(F, T+1) :- fluent(F), holds(F, T), not holds(neg(F), T+1).
holds(neg(F), T+1) :- fluent(F), holds(neg(F), T), not holds(F, T+1).
```

Planning

```
% action occurrences generation
1 {occ(A,T) : action(A) } 1 :- time(T), T ¡ length.
% Goal checking
goal(T) :- holds(on(b,a), T), holds(on(a,c), T), holds(onTable(c), T).
:- not goal(length).
```

Reasoning Problems in the Block World Domain

Let $\Pi_{block}(n)$ be the program for the block world domain with length sets to n without the set of rules for planning (the rules in the block below, denoted by $Plan(n)$).

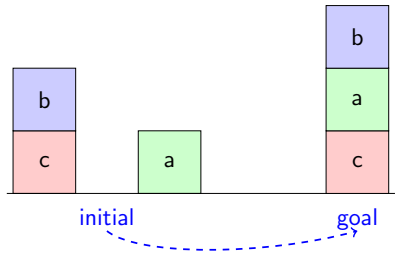
Plan(n)

```
% action occurrences generation
1 {occ(A,T) : action(A) } 1 :- time(T), T < length.
% Goal checking
goal(T) :- holds(on(b,a), T), holds(on(a,c), T), holds(onTable(c), T).
:- not goal(length).
```

This program can be used for two tasks: planning and hypothetical reasoning.

Planning in the Block World Domain

Planning is the problem of computing an action sequence that transforms the state of the world from the initial state to the goal state.



To use the program for planning, we need to add the planning module ($Plan(n)$) to the program. This module generates the action occurrences, defines the goal, and checks for the satisfiability of the goal.
To run the code, we need to provide an estimated length or use a script to run with length=1, 2, ... : **clingo -c length=<n> block.lp**
Answer sets of $\Pi_{block}(n) \cup Plan(n)$ contain atoms of the form

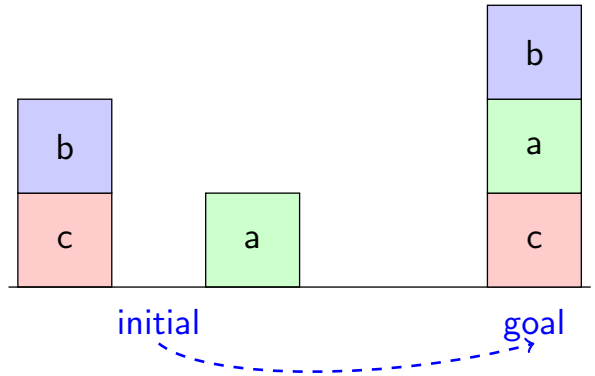
- $occ(a, t)$ where a is an action and t is an integer.
- $holds(l, t)$ where l is a fluent literal ($f / neg(f)$) and t is an integer.
- others such as $executable(a, t)$, etc.

We can show that if A is an answer set of the program $\Pi_{block}(n)$ $\cup Plan(n)$ then A will contain the atoms $occ(a_0, 0)$, ..., $occ(a_{n-1}, n-1)$ and $[a_0, \ldots, a_{n-1}]$ is a solution (soundness); and if the problem has a plan $[b_0, \ldots, b_{n-1}]$, then $\Pi_{block}(n) \cup Plan(n)$ has an answer set B containing $occ(b_0, 0), \ldots, occ(b_{n-1}, n-1)$ (completeness).

Example 1

How do we encode the problem?

- the initial state
- the goal state



- Initial state:

```
% Expressing what are true!
```

holds(on(b,c),0). holds(onTable(c),0). holds(onTable(a),0).
holds(clear(a),0). holds(clear(b),0). holds(handEmpty,0).

```
% Expressing what we do not state as true are false!
```

holds(neg(F),0) :- fluent(F), not holds(F,0).

- The goal state:

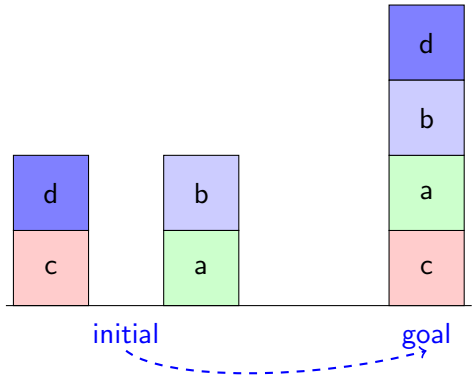
```
% Defining what should be the goal!
```

goal(T) :- holds(on(b,a),T), holds(on(a,c),T), holds(onTable(c),T).

Example 2

- What should be changed?
- How to encode the changes?

There is one additional block (Block d).
The initial state and the goal state change.



- The object: add block(d).
- % Initial state: Expressing what are true!

  holds(on(d,c),0). holds(onTable(c),0). holds(clear(d),0).
  holds(on(b,a),0). holds(onTable(a),0). holds(clear(b),0).
  holds(handEmpty,0).

  % Expressing what we do not state as true are false!

  holds(neg(F),0) :- fluent(F), not holds(F,0).
- % The goal state: Defining what should be the goal!

  goal(T) :- holds(on(d,b),T), holds(on(b,a),T),
  holds(on(a,c),T), holds(onTable(c),T).

Organizing Code

The ASP code for the block world domain is often organized into different files:

- The action description: this contains the description of the fluents, the actions, the action effects, the inertial axioms.
- The instance: this includes the rules defining the objects, the initial state, and the goal state.

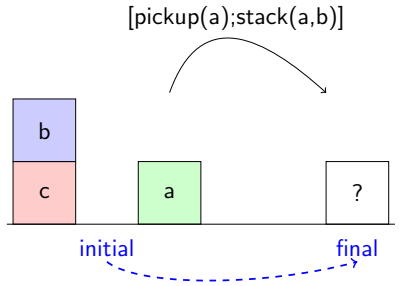
Alternative Goal Encoding

For a set of literal $\{l_1, \ldots, l_n\}$, the goal $l_1 \wedge l_2 \ldots \wedge l_n$ can be encoded using the set of rules:

```
goal(l_1).   ...   goal(l_n).
:- goal(l), not holds(l, length).
```

Hypothetical reasoning in the Block World Domain

Hypothetical reasoning: predicting what state of the world will be after the execution of an action sequence.



Assume that we would like to execute the sequence of actions $\alpha = [a_0, \ldots, a_{k-1}]$. To use $\Pi_{block}(n)$ for hypothetical reasoning we need to:

- set the value of *length* to be k the length of the action sequence
- add to the program the collection of facts $OCC(\alpha) = \{occ(a_0, 0), \ldots, occ(a_{k-1}, k-1)\}$.

To find out whether a fluent f is true (false) after the execution of the action sequence α, we need to

- compute all answer sets of $\Pi_{block}(k) \cup OCC(\alpha)$
- check whether $holds(f, k)$ is in each answer set; if it is, then answer YES.

Examples

- $unstack(b); putdown(b)$
- $unstack(b); putdown(b); stack(b, a)$
- $unstack(b); putdown(b); pickup(a); stack(a, c); pickup(b); stack(b, a)$
- $unstack(b); putdown(b); pickup(a); stack(a, c); pickup(b); stack(b, a)$
- $pickup(a); putdown(a); unstack(b); putdown(b); pickup(a); stack(a, c); pickup(b); stack(b, a)$

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*.

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.

How do we answer the above questions?

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*.

Frame Problem

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 number of bullets in 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.

Ramification Problem

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.

Fluents and Actions

We consider the situation where Jimmy has only one gun for now.

- Fluents: walking, loaded, dead
  We know that *alive* and *dead* are "two sides of a coin." For now, let us consider only one of them.
- Actions: shoot, load
  - *shoot*: if the is loaded then the turkey to be dead; otherwise, nothing happens
    - Precondition: *True*
    - Effects: two situations (*loaded* then *dead*; $\neg$*loaded* then nothing)
  - *load*:
    - Precondition: the gun is unloaded
    - Effects: the gun is loaded

ASP Encoding

```
% Defining the steps
time(0..length).
% Defining fluents
fluent(loaded). fluent(dead). fluent(walking).
% Defining actions
action(load). action(shoot).
% Representing action's effects
executable(shoot, T):- time(T).
holds(dead, T+1) :- occ(shoot, T), holds(loaded, T).
holds(-loaded, T+1) :- time(T), occ(shoot, T).
executable(load, T):- time(T), holds(-loaded, T).
holds(loaded, T+1) :- time(T), occ(load, T).
% The inertial rule
holds(F,T+1):- time(T), fluent(F), holds(F,T), not holds(-F,T+1).
holds(-F,T+1):- time(T),fluent(F),holds(-F,T),not holds(F,T+1).
% The initial state
holds(loaded, 0). holds(-dead, 0). holds(walking, 0).
```

Let us denote this program with $\boxed{\Pi_{Turkey}(length)}$.

Answering Questions

- Is *Matt* still alive? (after the shoot action!)
- Is the gun still loaded?
- Is *Jimmy* walking?
- Does *Jimmy* have the same number of guns?

Answering Questions

- Is *Matt* still alive? (after the shoot action!)
  - $length = 1$ (only one action occurrence)
  - Add the $occ(shoot, 0)$ to $\Pi_{Turkey}(1)$
  - Computer answer sets of the resulting program and check for answer
- Is the gun still loaded?
- Is *Jimmy* walking?
- Does *Jimmy* have the same number of guns?

Answering Questions

- Is *Matt* still alive? (after the shoot action!)
  - $length = 1$ (only one action occurrence)
  - Add the $occ(shoot, 0)$ to $\Pi_{Turkey}(1)$
  - Computer answer sets of the resulting program and check for answer
- Is the gun still loaded? Same as above
- Is *Jimmy* walking?
- Does *Jimmy* have the same number of guns?

Answering Questions

- Is *Matt* still alive? (after the shoot action!)
  - $length = 1$ (only one action occurrence)
  - Add the $occ(shoot, 0)$ to $\Pi_{Turkey}(1)$
  - Computer answer sets of the resulting program and check for answer
- Is the gun still loaded? Same as above
- Is *Jimmy* walking?
  We cannot answer the question as our modeling does not consider this! How to change this?
- Does *Jimmy* have the same number of guns?
  Again, our modeling does not consider this! How to change this?

Answering Questions

- Is *Matt* still alive? (after the shoot action!)
  - $length = 1$ (only one action occurrence)
  - Add the $occ(shoot, 0)$ to $\Pi_{Turkey}(1)$
  - Computer answer sets of the resulting program and check for answer
- Is the gun still loaded? Same as above
- Is *Jimmy* walking?
  Change the fluent walking to walking(.) and model it as follows.
  individual(matt). individual(jimmy).
  fluent(walking(I)):- individual(I).
- Does *Jimmy* have the same number of guns?

Answering Questions

- Is *Matt* still alive? (after the shoot action!)

  - $length = 1$ (only one action occurrence)
  - Add the $occ(shoot, 0)$ to $\Pi_{Turkey}(1)$
  - Computer answer sets of the resulting program and check for answer

- Is the gun still loaded? Same as above
- Is *Jimmy* walking?
  Change the fluent walking to `walking(.)` and model it as follows.
  individual(matt). individual(jimmy).
  fluent(walking(I)):- individual(I).
- Does *Jimmy* have the same number of guns?
  Add a fluent `number_of_guns(.)` and represent it as follows.
  number(0..10).
  fluent(number_of_guns(I)):- number(I).

After adding a fluent

- specify the truth values of these fluents in the initial state
- make sure that they are modeled correctly in action effects

Jimmy walking?

```
individual(matt). individual(jimmy).
fluent(walking(I)):- individual(I).
holds(walking(jimmy), 0). holds(walking(matt, 0).
```

There exists no action that affects walking ...

number_of_guns

```
number(0..10).
fluent(number_of_guns(I)):- number(I).
holds(number_of_guns(1), 0).
```

There exists no action that affects number_of_guns ...

Is Matt Walking?

It does not matter how we change the program, the current formalization will not allow us to conclude that Matt is not walking, i.e., for every answer set of the program $\Pi_{turkey}(1) \cup \{occ(shoot, 0)\}$ will contain $h(walking, 1)$.
What is the reason?
How to correct this?

Is Matt Walking?

It does not matter how we change the program, the current formalization will not allow us to conclude that Matt is not walking, i.e., for every answer set of the program $\Pi_{turkey}(1) \cup \{occ(shoot, 0)\}$ will contain $h(walking, 1)$.

What is the reason? there is no rule with the head -walking

How to correct this?

- Because dead implies -walking
- We add the rule

```
h(-walking,T) :- time(T), h(dead,T).
```

Ramification Problem

We know that if Matt is hit by the gun, he will be dead and if he is dead he cannot not continue walking. Therefore, we add the rule:

h(-walking,T) :- time(T), h(dead,T).

This rule represents a static causal law (or state constraint). It states a relationship between fluents and can cause a fluent to change its value even though the action does not directly change it. In general form, a static causal law is given by the statement of the form

If $l_1, \ldots, l_n$ are true then so is l.

This static causal law is written in answer set programing by the rule:

$h(l, T) : -time(T), h(l_1, T), \ldots, h(l_n, T)$.

Representing and reasoning with static causal laws are often called the ramification problem.

Another Example — Missionaries and Cannibals I

https://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem

River-Crossing Problem

Three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board. And, in some variations, one of the cannibals has only one arm and cannot row.

Jealous Husbands Problem

Three married couples, with the constraint that no woman can be in the presence of another man unless her husband is also present. Under this constraint, there cannot be both women and men present on a bank with women outnumbering men, since if there were, some woman would be husbandless. Therefore, upon changing men to missionaries and women to cannibals, any solution to the jealous husbands problem will also become a solution to the missionaries and cannibals problem.

First Encoding

```
% Defining the constants
time(0..length). number(0..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
action(cross(I,J,L)):-
   number(I), number(J), location(L), I+J ≤ 2, I+J > 0.
```

▸ back to the discussion

First Encoding I

```
% Effects of (M,N) crossing from one bank to another
holds(at(M+P,N+Q,L1),T+1):- time(T), number(P), number(Q),
   action(cross(M,N,L)), location(L1), L!=L1,
   occ(cross(M,N,L), T), holds(at(P,Q,L1), T).
holds(at(P-M,Q-N,L),T+1):- time(T), number(P), number(Q),
   action(cross(M,N,L)), occ(cross(M,N,L), T), holds(at(P,Q,L), T).
holds(boat_at(L1),T+1):- time(T),location(L1), L!=L1,
   action(cross(M,N,L)), occ(cross(M,N,L), T).
% executability condition
executable(cross(I,J,L), T):- time(T), number(P), number(Q),
   action(cross(I,J,L)), holds(boat_at(L), T),
   holds(at(P, Q, L), T), P≥I, Q≥J.
% constraint
:- time(T), action(cross(I,J,L)),
   occ(cross(I,J,L), T), not executable(cross(I,J,L), T).
:- number(I), number(J), location(L),
   time(T), holds(at(I,J,L), T), I < J, I> 0.
```

First Encoding II

```
% initial condition
holds(at(3,3,l1),0).
holds(at(0,0,l2),0).
holds(boat_at(l1),0).
% goal
:- not goal(length).
goal(T):- holds(at(3,3,l2), T).
1 { occ(A, T): action(A) } 1 :- time(T), not goal(T).
```

Differences from previous encoding

- We ignore the negative information (neg(.)). Why? (the negative information is implicit!)
- We add an extra constraint to represent the requirement "...for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals..." (we could have included it in the executability condition but we make use of the flexibility of logic programming)

The Second Encoding I

```
% Defining the constants
time(0..length).
% Three missionaries/three cannibals
number(0..3).
% Two banks of the river
location(l1). location(l2).
% X missionaries and Y cannibals are at location L
fluent(at(X,Y,L)):- number(X),number(Y),location(L).
% X missionaries and Y cannibals are in the boat
fluent(in_boat(X,Y)):- number(X),number(Y).
% the boat is at location L
fluent(boat_at(L)):- location(L).
% The boat moves from one bank to another
action(cross(L)):- location(L).
% X missionaries and Y cannibals depart the boat
action(depart(I,J)):- number(I), number(J), I+J>0.
% X missionaries and Y cannibals get on the boat
action(board(I,J)):- number(I), number(J), I+J>0.
```

The Second Encoding II

```
% Effects of departing from the boat
% increasing the number of individuals on the bank
holds(at(M+P,N+Q, L),T+1):- time(T), occ(depart(M,N), T),
   holds(boat_at(L), T), holds(at(P,Q,L), T),
   holds(in_boat(X,Y), T), M ≤ X, N ≤ Y.
% reducing the number in the boat
holds(in_boat(P,Q),T+1):- time(T), occ(depart(M,N), T),
   holds(boat_at(L), T), fluent(in_boat(P,Q)),
   holds(in_boat(M+P,N+Q), T).
% does not change the number of individuals on the other bank
holds(at(P,Q,L1),T+1):- time(T), occ(depart(M,N), T),
   holds(boat_at(L), T), holds(at(P,Q,L1), T),
   location(L1), L1!=L.
% does not change the location of the boat
holds(boat_at(L), T+1):- time(T), location(L),
   holds(boat_at(L), T), action(depart(M,N)),
   occ(depart(M,N), T).
```

The Second Encoding III

```
% boarding to the boat
% reducing the number of individuals on the bank
holds(at(M-P,N-Q,L),T+1):- time(T), occ(board(P,Q), T),
   holds(boat_at(L), T), holds(at(M,N,L), T),
   M >= P, N >= Q.
% increasing the number of individuals in the boat
holds(in_boat(M+P,N+Q),T+1):- time(T), occ(board(P,Q), T),
   holds(boat_at(L), T), holds(in_boat(M,N), T).
% does not change the number of individuals on the other bank
holds(at(M,N,L1),T+1):- time(T), occ(board(P,Q), T),
   holds(boat_at(L), T), holds(at(M,N,L1), T),
   location(L1), L1!=L.
% does not change the location of the boat
holds(boat_at(L), T+1):- time(T), location(L),
   action(board(P,Q)), holds(boat_at(L), T),
   occ(board(P,Q), T).
```

The Second Encoding IV

```
% boat crossing
% change the location of the boat
holds(boat_at(L1),T+1):- time(T), location(L1), L!=L1,
   occ(cross(L), T).
% does not change the number of individuals in the boat
holds(in_boat(M,N), T+1):- time(T), holds(in_boat(M,N), T),
   occ(cross(L), T).
% does not change the number of individuals on the banks
holds(at(M,N,L1),T+1):- time(T), occ(cross(L), T),
   holds(at(M,N,L1), T), location(L1).
```

The Second Encoding V

```
% executability condition
executable(cross(L), T):-
    time(T),
    action(cross(L)),
    holds(boat_at(L), T).
executable(board(M,N), T):-
    time(T),
    action(board(M,N)),
    holds(boat_at(L), T),
    holds(in_boat(P,Q), T),
    holds(at(X,Y,L), T),
    X>=M, Y>=N, M+N>0, M+N+P+Q<=2.
executable(depart(M,N), T):-
    time(T),
    action(depart(M,N)),
    holds(in_boat(P,Q), T),
    M<=P, N<=Q, M+N>0.
```

The Second Encoding VI

```
% initial condition
holds(at(3,3,l1),0).
holds(at(0,0,l2),0).
holds(boat_at(l1),0).
holds(in_boat(0,0),0).
% constraint
:- time(T), action(A), occ(A, T), not executable(A, T).
:- number(I), number(J), location(L),
   time(T), holds(at(I,J,L), T), I<J, I>0.
% goal
:- not goal(length).
goal(T):- holds(at(3,3,l2), T).
1 { occ(A, T): action(A) } 1 :- time(T), T < length.
```

This program has the shortest plan with 28 actions. `clingo` takes more than 1 hour and cannot even verify it!

Modeling and Encoding in ASP

- The right level of abstraction: one type of actions ($cross(I, J, L)$) vs. three types ($cross(L)$, $board(X, Y)$, $depart(X, Y)$). The second provides more details. However, the program would not even run!

  Trade off: details vs. computability!

- Use the right constructs and vocabularies: When we have one type of actions ($cross(X, Y, L)$), we need to encode all information (who is crossing and from which bank), we could have used $cross(X, Y, L1, L2)$ to denote the direction of movement of the boat. However, the second representation is not necessary in this problem. It will be necessary to introduce the second parameter if the setting changes. For example, the river is replaced by a triangle pond!

High Level Action Languages I

$\mathcal{A}$ 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

High Level Action Languages II

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

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \tag{6}$$

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

$$\textbf{initially } f \tag{8}$$

$$a \textbf{ executable\_if } p_1 \ldots, p_n \tag{9}$$

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. (6), referred as *dynamic law*, represents the (conditional) effect of action a. (7) is a *static law* which represents the relationship between fluents. Propositions of the form (8), 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 (6)-(8) and I consists of propositions of the form (8). 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.

High Level Action Languages III

- To say that initially, the turkey is walking and not dead, we write
  initially $\neg$*dead* and
  initially *walking*
- Initially, the gun is loaded:
  initially *loaded*
- Shooting causes the turkey to be dead if the gun is loaded can be expressed by
  shoot **causes** *dead* **if** *loaded* and
  shoot **causes** $\neg$*loaded* **if** *loaded*
- Un/Loading the gun causes the gun to be un/loaded
  load **causes** *leaded* and
  unload **causes** $\neg$*loaded*
- Dead turkeys cannot walk
  $\neg$*walking* **if** *dead*
- A gun can be loaded only when it is not loaded
  load **executable_if** $\neg$*loaded*

High Level Action Languages IV

So, the action theory is

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

and

$$D_y = \left\{ \begin{array}{l} shoot \ \textbf{causes} \ dead \ \textbf{if} \ loaded \\ shoot \ \textbf{causes} \ \neg loaded \ \textbf{if} \ loaded \\ load \ \textbf{causes} \ leaded \\ \neg walking \ \textbf{if} \ dead \\ shoot \ \textbf{executable\_if} \ true \\ load \ \textbf{executable\_if} \ \neg loaded \end{array} \right\}$$

Why High Level Action Languages?

Each action theory can be easily translated into an ASP program. In fact,

- each law a **causes** f **if** $p_1, \ldots, p_k$ is translated into the rule:
  $holds(f, T+1) \leftarrow occ(a, T), holds(p_1, T), \ldots, holds(p_k, T).$
- each law f **if** $p_1, \ldots, p_k$ is translated into the rule:
  $holds(f, T) \leftarrow holds(p_1, T), \ldots, holds(p_k, T).$
- each law a **executable_if** $p_1, \ldots, p_k$ is translated into the rules:
  $executable(a, T) \leftarrow holds(p_1, T), \ldots, holds(p_k, T).$
- each law **initially** f is translated into the rule:
  $holds(f, 0).$

The other rules that we have seen are the same for all theories:

$\leftarrow occ(a, T),$ **not** $executable(a, T).$

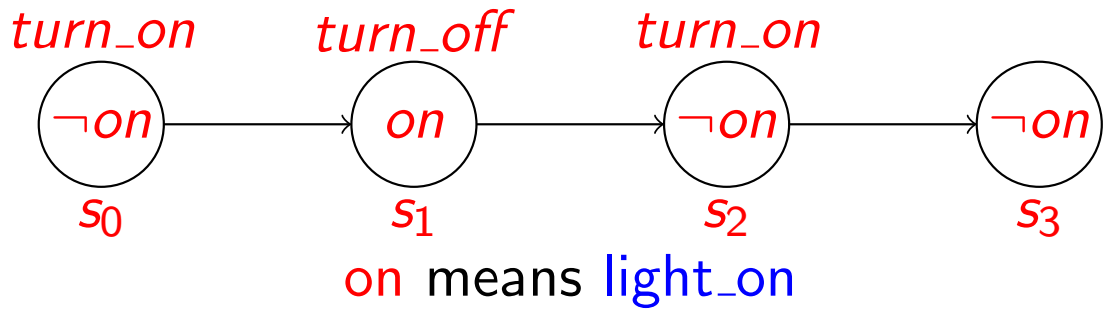
$1\{occ(a, T) : action(A)\}1.$

Diagnosis

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

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

and

$$OBS = \begin{cases} (o1) & turn\_on \textbf{ occurs\_at } s_0 \\ (o2) & turn\_off \textbf{ occurs\_at } s_1 \\ (o3) & turn\_on \textbf{ between } s_2, s_3 \\ (o4) & s_0 \textbf{ precedes } s_1 \\ (o5) & s_1 \textbf{ precedes } s_2 \\ (o6) & s_2 \textbf{ precedes } s_3 \\ (o7) & \neg light\_on \textbf{ at } s_0 \\ (o8) & light\_on \textbf{ at } s_1 \\ (o9) & \neg light\_on \textbf{ at } s_2 \\ (o10) & \neg light\_on \textbf{ at } s_3 \end{cases}$$

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

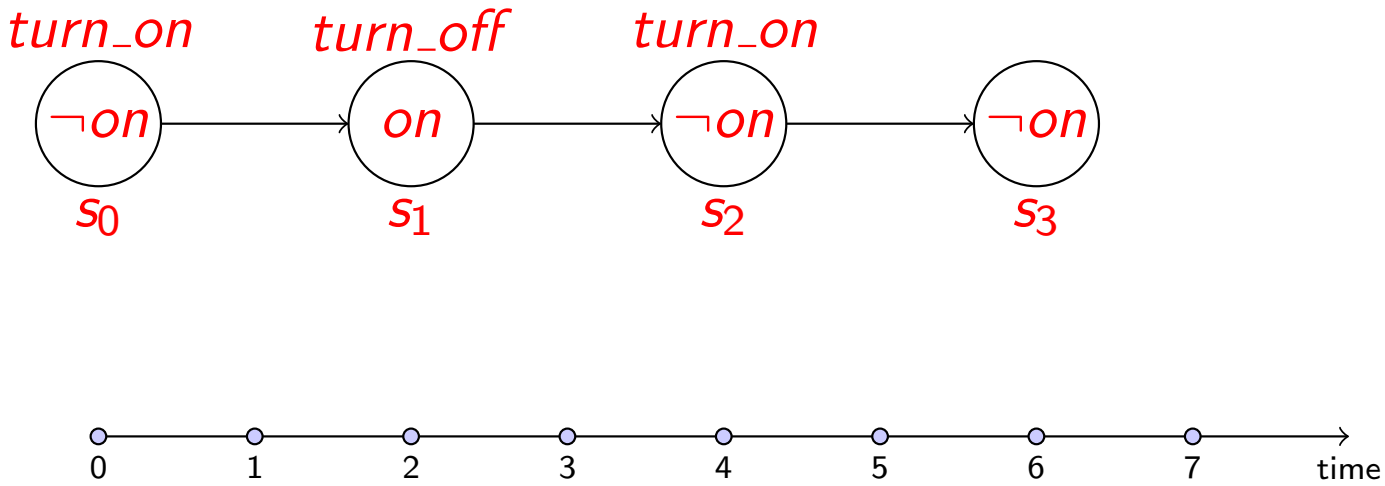
Given a system description $(SD, COMPS, OBS)$ with

- 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.

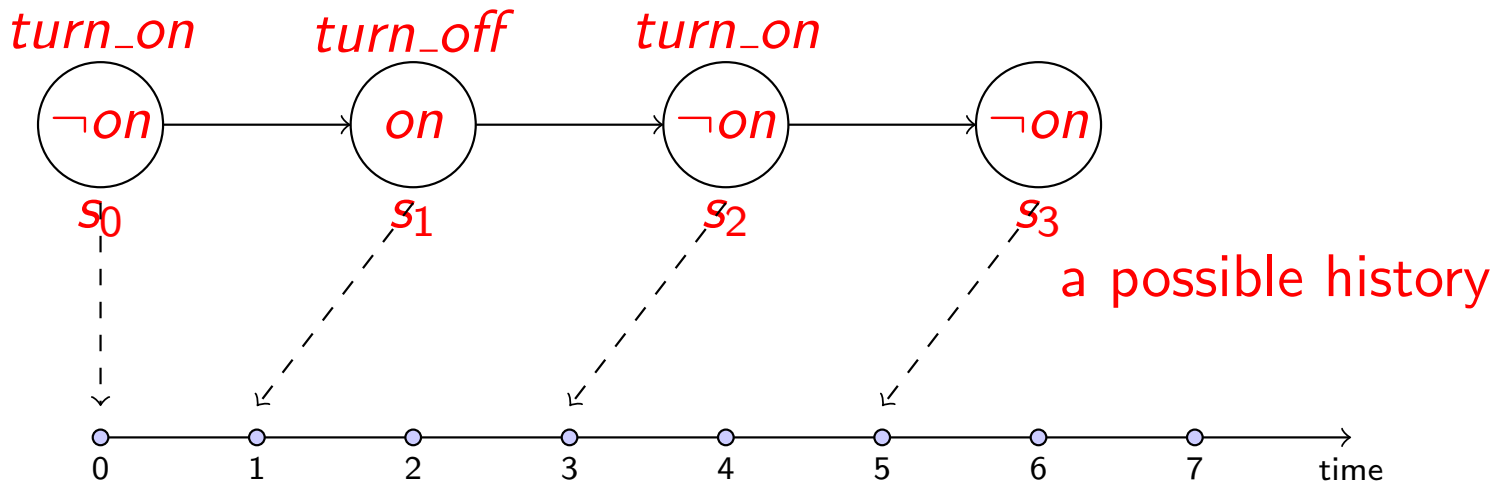
General Idea



The narrative happens over a course of time (0, 1, 2, ...).

- Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.

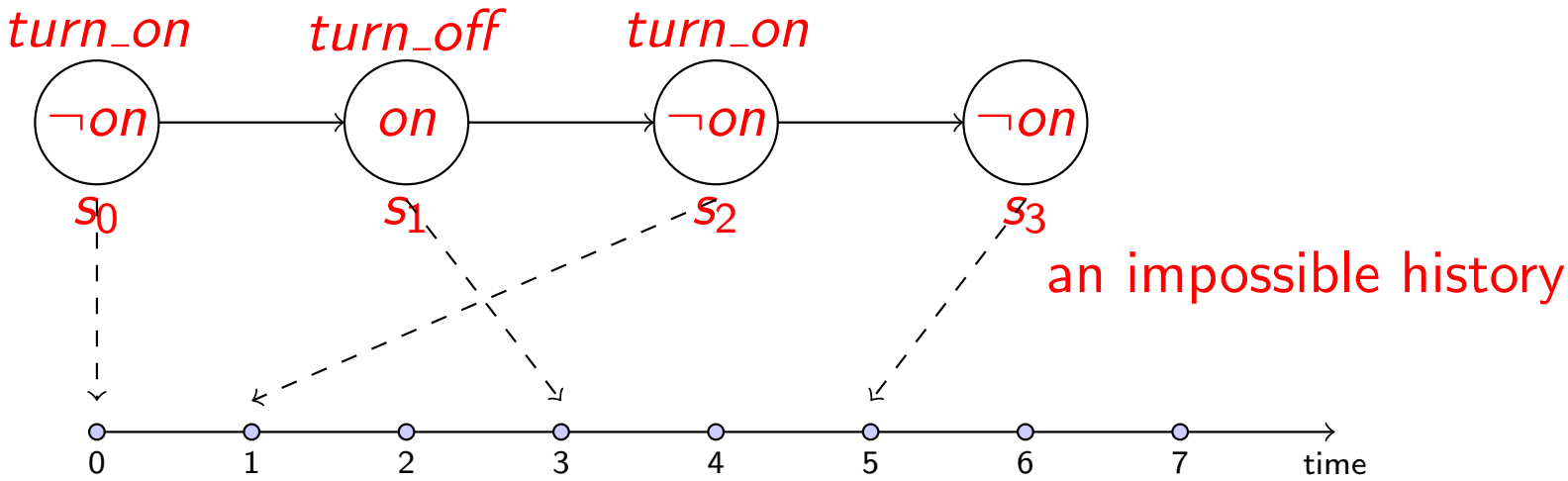
General Idea



The narrative happens over a course of time (0, 1, 2, ...).

- Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.

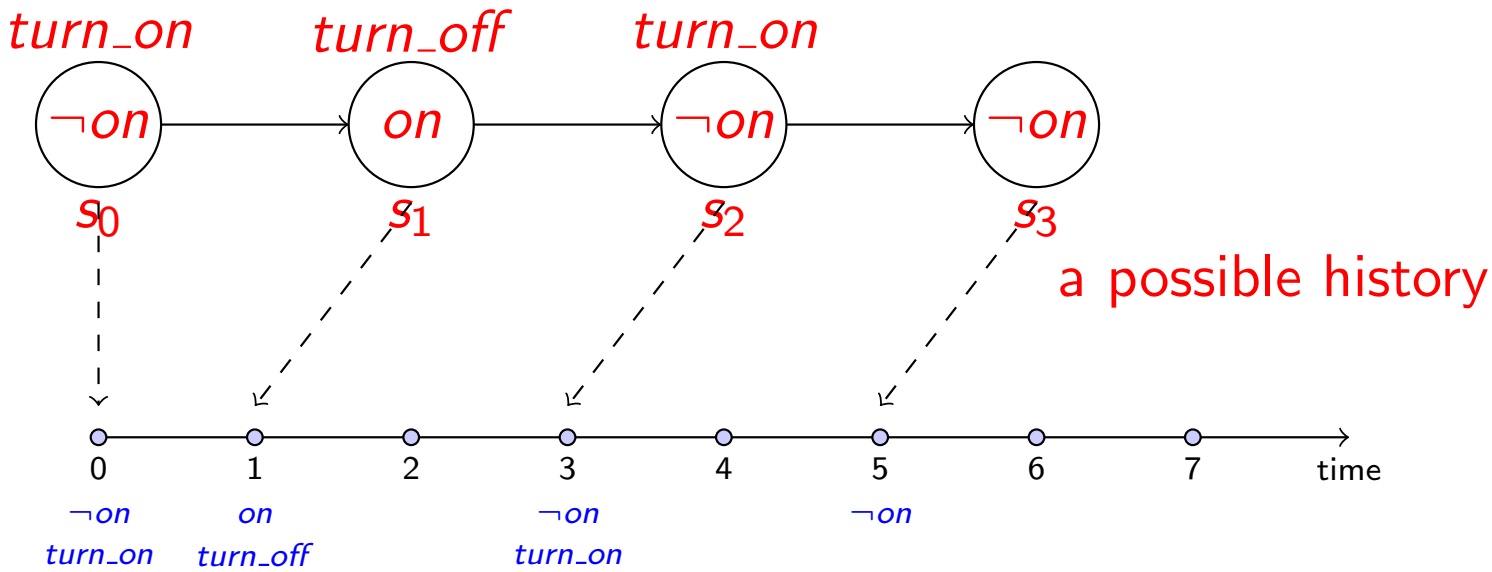
General Idea



The narrative happens over a course of time (0, 1, 2, ...).

- Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.

General Idea



The narrative happens over a course of time (0, 1, 2, ...).

- Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.
- Assume that the assignment is good. We then need to generate action occurrences and ensure that observations are true.

Elements of Π I

```
% Constants: time, situations, actions, fluents
time(0..length).
% 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)).
% 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), fluent(F), holds(F, T), not holds(neg(F), T+1).
holds(neg(F),T+1):-time(T),fluent(F),holds(neg(F),T),not holds(F,T+1)
```

Elements of Π II

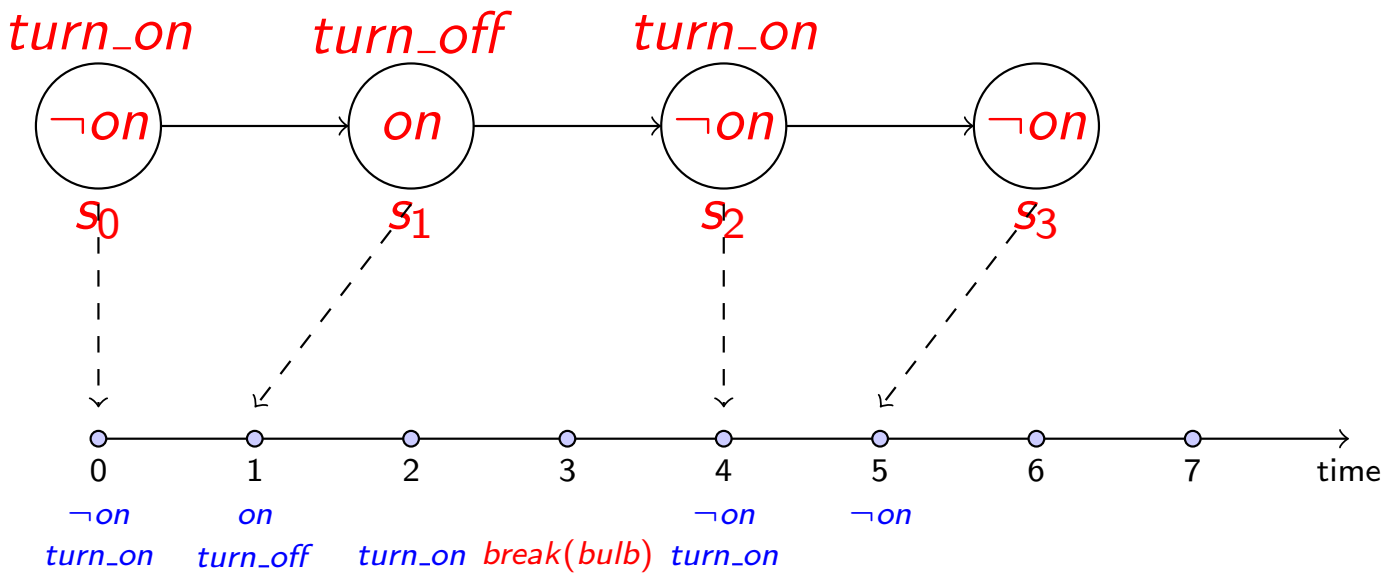
```
% specifying the set of observations situation order
prec(s0,s1).   prec(s1,s2).   prec(s2,s3).
% fluent observations
at(neg(light_on), s0).   at(neg(ab(bulb)), s0).   at(light_on, s1).
at(neg(light_on), s2).   at(neg(light_on), s3).
% action observations
between(s0,s1,turn_on).   between(s2,s3,turn_on).
exec_at(s1,turn_off).
% 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), T2 !=T1+1.
```

Elements of Π III

```
% generating action occurrences
1{occ(A, T): action(A) } 1 :- time(T), T < length.
% satisfying all the observations
% fluent observations
holds(F, T):- time(T), fluent(F), sit(S), at(F, S), happens(S, T).
holds(neg(F),T):-time(T),fluent(F),sit(S),at(neg(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), A1!= A.
occ(A, T1):- time(T1), action(A), between(S1,S2,A), happens(S1,T1).
:- time(T), action(A1), action(A2), A1 != A2, occ(A1, T), occ(A2, T).
% constraints that eliminate inconsistency model
:- time(T), fluent(F), holds(F, T), holds(neg(F), T).
```

Running the program

clingo -c length=5 bulb.lp
that gives an answer set with the following:
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)



Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!

Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!

Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Reason

It is the inertial axiom!

Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Reason

It is the inertial axiom!

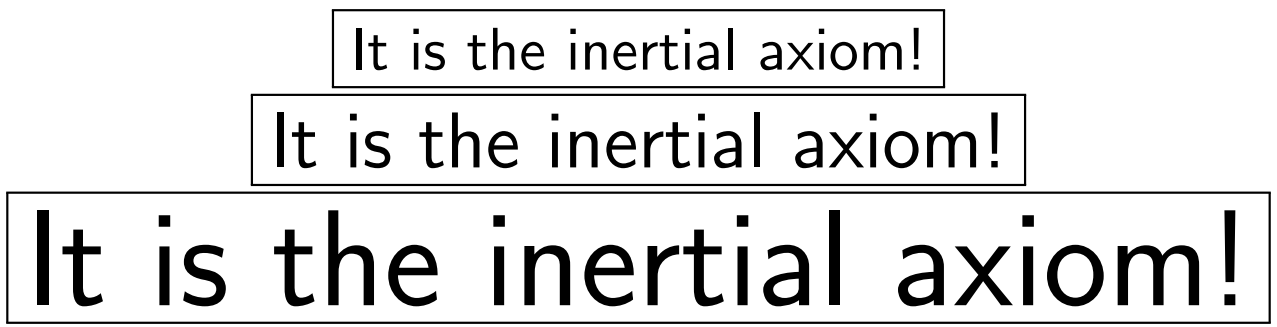
It is the inertial axiom!

Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Reason

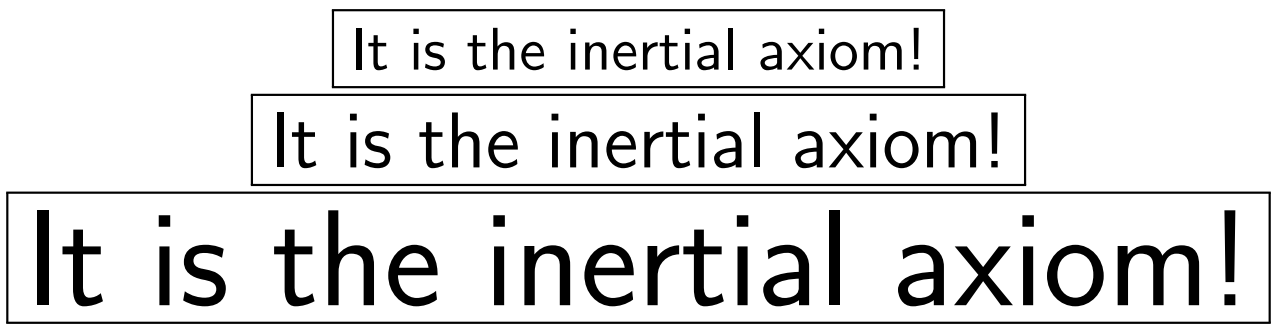


Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Reason



What is it? The inertial law says that object at rest stays at rest. Translated to reasoning about effects of actions and changes: what is true/false stays true/false if it has not been changed by the action execution.

Inertial Axioms in Our Examples

Different ways to do it

- See ▸ Formalization in the block world domain
  holds(F, T+1) :- fluent(F), holds(F, T), not holds(neg(F), T+1).
  holds(neg(F),T+1):-fluent(F),holds(neg(F),T),not holds(F,T+1).
- See ▸ Formalization in the Yale shooting problem
  Same as in the block world domain.
- The formalization in the missionaries and cannibals is different. See ▸ MC problem
  It seems that we forgot inertial rules! NO, we just did it in a different way.

Frame Axioms in MC Problem I

We have fluents: at(M,N,L), boat_at(L). $0 \leq M, N \leq 3$. They can be divided into three types: at(M,N,I1), at(M,N,I2), boat_at(L) (color coded: red, blue, black)

If you display the fluents from the program you will see the following fluents:

fluent(at(0,0,I1)) fluent(at(1,0,I1)) fluent(at(2,0,I1)) fluent(at(3,0,I1))
fluent(at(0,1,I1)) fluent(at(1,1,I1)) fluent(at(2,1,I1)) fluent(at(3,1,I1))
fluent(at(0,2,I1)) fluent(at(1,2,I1)) fluent(at(2,2,I1)) fluent(at(3,2,I1))
fluent(at(0,3,I1)) fluent(at(1,3,I1)) fluent(at(2,3,I1)) fluent(at(3,3,I1))
fluent(at(0,0,I2)) fluent(at(1,0,I2)) fluent(at(2,0,I2)) fluent(at(3,0,I2))
fluent(at(0,1,I2)) fluent(at(1,1,I2)) fluent(at(2,1,I2)) fluent(at(3,1,I2))
fluent(at(0,2,I2)) fluent(at(1,2,I2)) fluent(at(2,2,I2)) fluent(at(3,2,I2))
fluent(at(0,3,I2)) fluent(at(1,3,I2)) fluent(at(2,3,I2)) fluent(at(3,3,I2))
fluent(boat_at(I1)) fluent(boat_at(I2))

Frame Axioms in MC Problem II

For each action $cross(I, J, L)$ we have one rule that changes the value of

holds(at(I,J,L),T+1) :- ... occ(cross(P,Q,L),T), ...

holds(at(I,J,L1),T+1) :- ... occ(cross(P,Q,L),T), L1!=L,...

holds(boat_at(L1),T+1) :- ... occ(cross(P,Q,L),T), L1!=L,...

The three rules guarantee that there will be one fluent of each type that is true in the next time step (T+1) if the action is executable.

Since we represent negative information implicitly (there is no neg(.) in the program), we account for all the fluents in the problem (holds(at(.,.,l), t)) records what is true; the missing fluents are false).

For example, when we add occ(cross(1,1,l1),0) to the program and display atoms of the form holds(.,.) we can see the following

holds(at(3,3,l1),0) holds(at(0,0,l2),0) holds(boat_at(l1),0)

holds(at(2,2,l1),1) holds(at(1,1,l2),1) holds(boat_at(l2),1)

Frame Axioms in MC Problem III

Why a new way?

We could represent the negative information to the MC program by the rule (Closed World Assumption):

holds(neg(F),0) :- fluent(F), not holds(F, 0).

and the inertial rules

holds(F, T+1) :- fluent(F), holds(F, T), not holds(neg(F), T+1).

holds(neg(F),T+1):-fluent(F),holds(neg(F),T),not holds(F,T+1).

then we get the following results after the action $cross(1, 1, l1)$ occurs, i.e., $occ(cross(1, 1, l1), 0)$ is added to the program (red for positive information and black for negative information, which are not good!):

holds(at(3,3,l1),0) holds(at(0,0,l2),0) holds(boat_at(l1),0)
holds(boat_at(l2),1) holds(boat_at(l1),1) holds(at(2,2,l1),1)
holds(at(1,1,l2),1) holds(at(3,3,l1),1) holds(at(0,0,l2),1)
holds(boat_at(l2),2) holds(boat_at(l1),2) holds(at(2,2,l1),2)
holds(at(1,1,l2),2) holds(at(3,3,l1),2) holds(at(0,0,l2),2)
holds(neg(at(0,0,l1)),0) holds(neg(at(1,0,l1)),0) holds(neg(at(2,0,l1)),0)

Frame Axioms in MC Problem IV

holds(neg(at(3,0,l1)),0) holds(neg(at(0,1,l1)),0) holds(neg(at(1,1,l1)),0)
holds(neg(at(2,1,l1)),0) holds(neg(at(3,1,l1)),0) holds(neg(at(0,2,l1)),0)
holds(neg(at(1,2,l1)),0) holds(neg(at(2,2,l1)),0) holds(neg(at(3,2,l1)),0)
holds(neg(at(0,3,l1)),0) holds(neg(at(1,3,l1)),0) holds(neg(at(2,3,l1)),0)
holds(neg(at(1,0,l2)),0) holds(neg(at(2,0,l2)),0) holds(neg(at(3,0,l2)),0)
holds(neg(at(0,1,l2)),0) holds(neg(at(1,1,l2)),0) holds(neg(at(2,1,l2)),0)
holds(neg(at(3,1,l2)),0) holds(neg(at(0,2,l2)),0) holds(neg(at(1,2,l2)),0)
holds(neg(at(2,2,l2)),0) holds(neg(at(3,2,l2)),0) holds(neg(at(0,3,l2)),0)
holds(neg(at(1,3,l2)),0) holds(neg(at(2,3,l2)),0) holds(neg(at(3,3,l2)),0)
holds(neg(boat_at(l2)),0) holds(neg(at(0,0,l1)),1)
holds(neg(at(1,0,l1)),1) holds(neg(at(2,0,l1)),1) holds(neg(at(3,0,l1)),1)
holds(neg(at(0,1,l1)),1) holds(neg(at(1,1,l1)),1) holds(neg(at(2,1,l1)),1)
holds(neg(at(3,1,l1)),1) holds(neg(at(0,2,l1)),1) holds(neg(at(1,2,l1)),1)
holds(neg(at(3,2,l1)),1) holds(neg(at(0,3,l1)),1) holds(neg(at(1,3,l1)),1)
holds(neg(at(2,3,l1)),1) holds(neg(at(1,0,l2)),1) holds(neg(at(2,0,l2)),1)
holds(neg(at(3,0,l2)),1) holds(neg(at(0,1,l2)),1) holds(neg(at(2,1,l2)),1)
holds(neg(at(3,1,l2)),1) holds(neg(at(0,2,l2)),1) holds(neg(at(1,2,l2)),1)

Frame Axioms in MC Problem V

holds(neg(at(2,2,l2)),1) holds(neg(at(3,2,l2)),1) holds(neg(at(0,3,l2)),1)
holds(neg(at(1,3,l2)),1) holds(neg(at(2,3,l2)),1) holds(neg(at(3,3,l2)),1)
holds(neg(at(0,0,l1)),2) holds(neg(at(1,0,l1)),2) holds(neg(at(2,0,l1)),2)
holds(neg(at(3,0,l1)),2) holds(neg(at(0,1,l1)),2) holds(neg(at(1,1,l1)),2)
holds(neg(at(2,1,l1)),2) holds(neg(at(3,1,l1)),2) holds(neg(at(0,2,l1)),2)
holds(neg(at(1,2,l1)),2) holds(neg(at(3,2,l1)),2) holds(neg(at(0,3,l1)),2)
holds(neg(at(1,3,l1)),2) holds(neg(at(2,3,l1)),2) holds(neg(at(1,0,l2)),2)
holds(neg(at(2,0,l2)),2) holds(neg(at(3,0,l2)),2) holds(neg(at(0,1,l2)),2)
holds(neg(at(2,1,l2)),2) holds(neg(at(3,1,l2)),2) holds(neg(at(0,2,l2)),2)
holds(neg(at(1,2,l2)),2) holds(neg(at(2,2,l2)),2) holds(neg(at(3,2,l2)),2)
holds(neg(at(0,3,l2)),2) holds(neg(at(1,3,l2)),2) holds(neg(at(2,3,l2)),2)
holds(neg(at(3,3,l2)),2)

Frame Axioms in MC Problem VI

The reason for the wrong results lies in that we do not have rules for negative effects. For example, if we add

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

Then we solve the problem with holds(boat_at(l2),1) holds(boat_at(l1),1).

How do we solve the problem with holds(at(3,3,l1),0) holds(at(3,3,l1),1)?

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

How do we solve the problem with holds(at(1,1,l2),1) holds(at(0,0,l2),1)?

```
holds(neg(at(P,Q,L1)), T+1):- time(T), action(cross(M,N,L)),
   occ(cross(M,N,L), T), L!=L1, holds(at(P,Q,L1), T).
```

We get the following results:

holds(at(3,3,l1),0) holds(at(0,0,l2),0) holds(boat_at(l1),0)
holds(boat_at(l2),1) holds(at(2,2,l1),1) holds(at(1,1,l2),1)
holds(at(2,2,l1),2) holds(at(1,1,l2),2) holds(neg(boat_at(l1)),1)
holds(neg(at(0,0,l1)),0) holds(neg(at(1,0,l1)),0) holds(neg(at(2,0,l1)),0)
holds(neg(at(3,0,l1)),0) holds(neg(at(0,1,l1)),0) holds(neg(at(1,1,l1)),0)
holds(neg(at(2,1,l1)),0) holds(neg(at(3,1,l1)),0) holds(neg(at(0,2,l1)),0)
holds(neg(at(1,2,l1)),0) holds(neg(at(2,2,l1)),0) holds(neg(at(3,2,l1)),0)

Frame Axioms in MC Problem VII

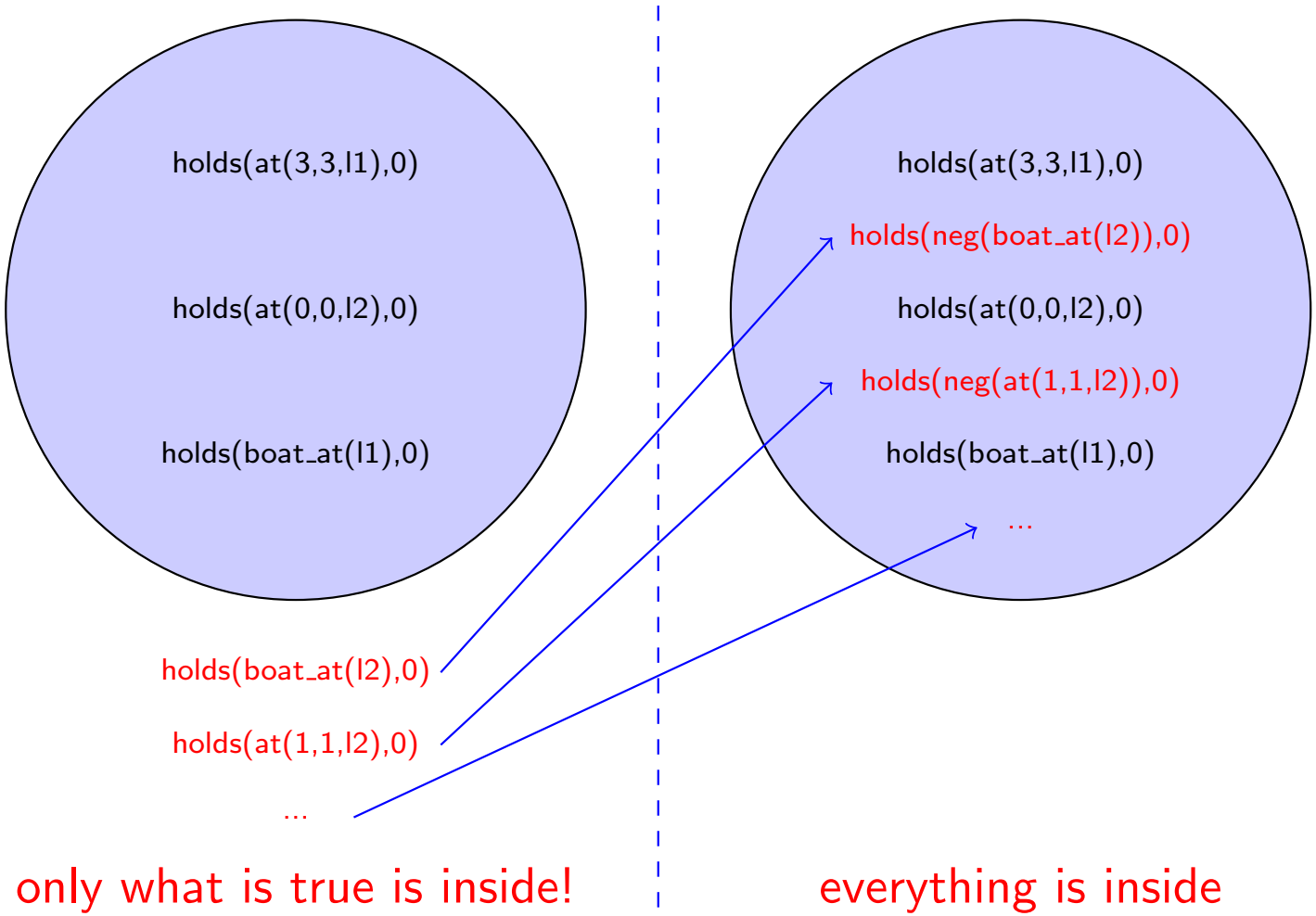
holds(neg(at(0,3,l1)),0) holds(neg(at(1,3,l1)),0) holds(neg(at(2,3,l1)),0)
holds(neg(at(1,0,l2)),0) holds(neg(at(2,0,l2)),0) holds(neg(at(3,0,l2)),0)
holds(neg(at(0,1,l2)),0) holds(neg(at(1,1,l2)),0) holds(neg(at(2,1,l2)),0)
holds(neg(at(3,1,l2)),0) holds(neg(at(0,2,l2)),0) holds(neg(at(1,2,l2)),0)
holds(neg(at(2,2,l2)),0) holds(neg(at(3,2,l2)),0) holds(neg(at(0,3,l2)),0)
holds(neg(at(1,3,l2)),0) holds(neg(at(2,3,l2)),0) holds(neg(at(3,3,l2)),0)
holds(neg(boat_at(l2)),0) holds(neg(boat_at(l1)),2) holds(boat_at(l2),2)
holds(neg(at(0,0,l1)),1) holds(neg(at(1,0,l1)),1) holds(neg(at(2,0,l1)),1)
holds(neg(at(3,0,l1)),1) holds(neg(at(0,1,l1)),1) holds(neg(at(1,1,l1)),1)
holds(neg(at(2,1,l1)),1) holds(neg(at(3,1,l1)),1) holds(neg(at(0,2,l1)),1)
holds(neg(at(1,2,l1)),1) holds(neg(at(3,2,l1)),1) holds(neg(at(0,3,l1)),1)
holds(neg(at(1,3,l1)),1) holds(neg(at(2,3,l1)),1) holds(neg(at(1,0,l2)),1)
holds(neg(at(2,0,l2)),1) holds(neg(at(3,0,l2)),1) holds(neg(at(0,1,l2)),1)
holds(neg(at(2,1,l2)),1) holds(neg(at(3,1,l2)),1) holds(neg(at(0,2,l2)),1)
holds(neg(at(1,2,l2)),1) holds(neg(at(2,2,l2)),1) holds(neg(at(3,2,l2)),1)
holds(neg(at(0,3,l2)),1) holds(neg(at(1,3,l2)),1) holds(neg(at(2,3,l2)),1)
holds(neg(at(3,3,l2)),1) holds(neg(at(3,3,l1)),1) holds(neg(at(0,0,l2)),1)
holds(neg(at(0,0,l1)),2) holds(neg(at(1,0,l1)),2) holds(neg(at(2,0,l1)),2)
holds(neg(at(3,0,l1)),2) holds(neg(at(0,1,l1)),2) holds(neg(at(1,1,l1)),2)

Frame Axioms in MC Problem VIII

holds(neg(at(2,1,l1)),2) holds(neg(at(3,1,l1)),2) holds(neg(at(0,2,l1)),2)
holds(neg(at(1,2,l1)),2) holds(neg(at(3,2,l1)),2) holds(neg(at(0,3,l1)),2)
holds(neg(at(1,3,l1)),2) holds(neg(at(2,3,l1)),2) holds(neg(at(1,0,l2)),2)
holds(neg(at(2,0,l2)),2) holds(neg(at(3,0,l2)),2) holds(neg(at(0,1,l2)),2)
holds(neg(at(2,1,l2)),2) holds(neg(at(3,1,l2)),2) holds(neg(at(0,2,l2)),2)
holds(neg(at(1,2,l2)),2) holds(neg(at(2,2,l2)),2) holds(neg(at(3,2,l2)),2)
holds(neg(at(0,3,l2)),2) holds(neg(at(1,3,l2)),2) holds(neg(at(2,3,l2)),2)
holds(neg(at(3,3,l2)),2) holds(neg(at(3,3,l1)),2) holds(neg(at(0,0,l2)),2)

Frame Axioms in MC Problem IX

Two views of the same coin



Adding Actions and Programs Stop Working I

Problem

I added an action that allows for the boat to move to an island.

The program stops working.

The changes you might have made:

```
number(0..4).        % increase number of MC
location(island).    % add the island
action(cross(I,J,L,L1)):- number(I), number(J), I+J ≤ 2, I+J > 0,
   location(L), L!=L1, location(L1).    % change actions
... replace cross(I,J,L) in the old program to cross(I,J,L,L1) ...
holds(at(0,0,island),0).    % specify number of MC on island
goal(T):- holds(at(4,4,I2), T).    % make the goal
```

Adding Actions and Programs Stop Working II

Why so?

Remove the goal and adding occ(cross(1,1,l1,l2),0) to the program, run with *length* = 1, you will see
holds(at(4,4,l1),0) holds(at(0,0,l2),0) holds(boat_at(l1),0) holds(at(0,0,island),0)
holds(at(3,3,l1),1) holds(at(1,1,l2),1) holds(boat_at(l2),1)
Oh, I expect that **holds(at(0,0,island),1)** be there!
Where does it go?
In fact, if you add occ(cross(1,1,l1,island),0) to the program, then **holds(at(0,0,l2),1)** is expected but not present.
And, that was the source of the problem: something we expect to be true but disappear! The reason was inertial axioms are missing.
By introducing the island, we create the fourth type of fluents (analogous to our previous analysis, at(P,Q,island)). Our program change the value of only three types of fluents after one action is executed. The fourth type must stay the same.

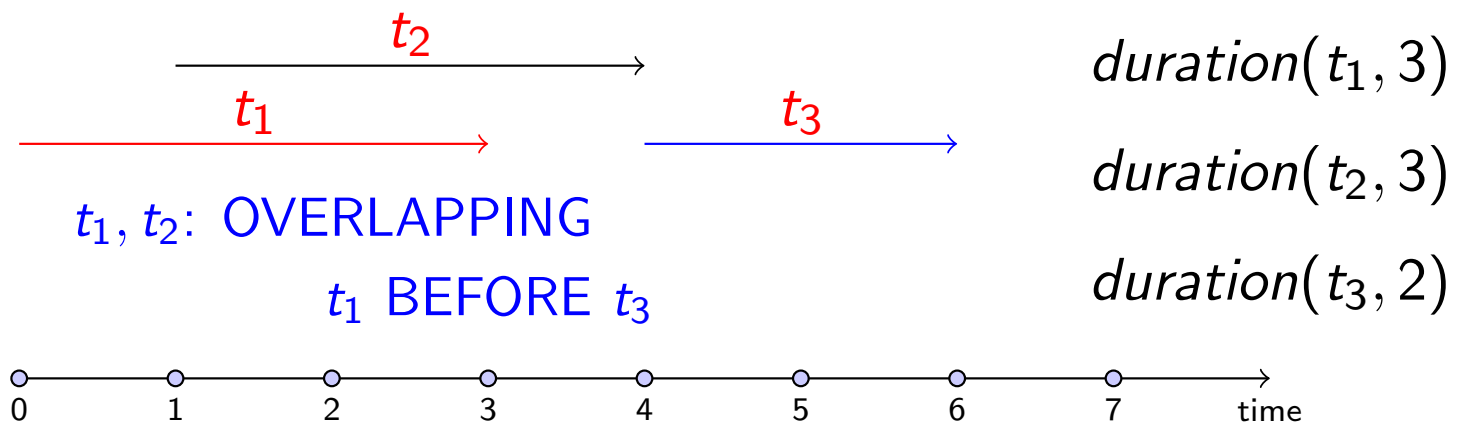
Scheduling

Problem

We have several tasks $t_1, \ldots, t_n$. For every i, we have

- a unique atom $duration(t_i, d_i)$ that encodes the duration of the task t_i (we assume that d_i is a positive integer);
- a collection of atoms of the form $prec(t_i, t_j)$ which says that t_i has to be completed before t_j can start.
- a collection of atoms of the form $non\_overlap(t_i, t_j)$ which says that t_i and t_j cannot be overlapped.

Goal: find a schedule to complete the $t_1, \ldots, t_n$ with minimal span (total time).

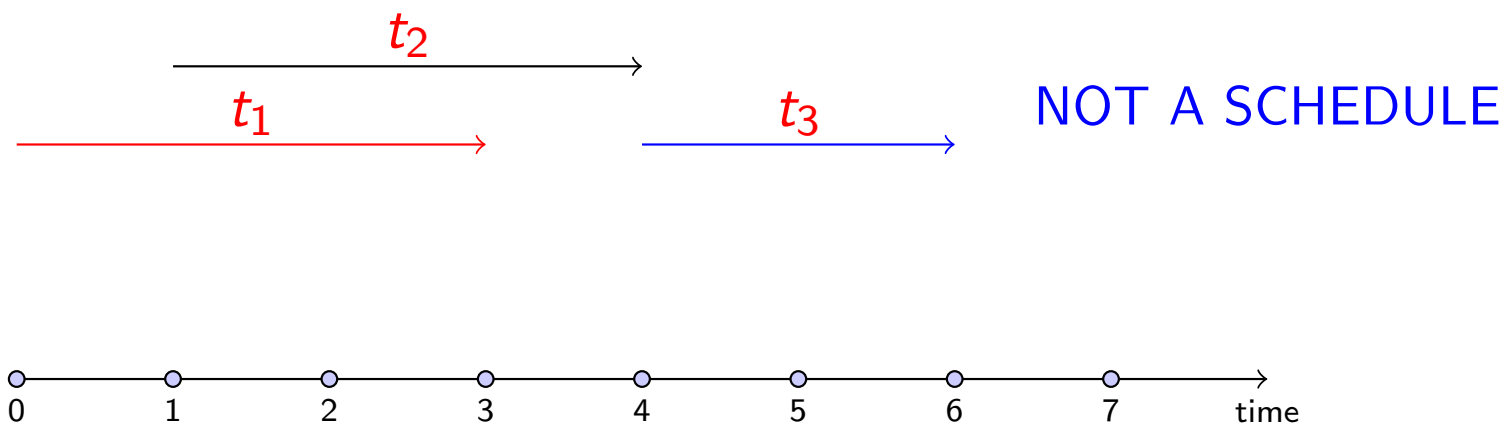


A Schedule for a Set of Tasks: Definition

A schedule for the set of tasks $T = \{t_1, \ldots, t_n\}$ is a mapping of T to the set of non-negative integers $\mathbf{N}$, denoted by $start : T \longrightarrow \mathbf{N}$, such that

- if $prec(t_i, t_j)$ is true then $start(t_i) + d_i \leq start(t_j)$ (t_i completed before t_j)
- if $non\_overlap(t_i, t_j)$ is true then $start(t_i) + d_i \leq start(t_j)$ or $start(t_j) + d_j \leq start(t_i)$

Given three tasks t_1, t_2, t_3 with $duration(t_1, 3)$, $duration(t_2, 3)$, $duration(t_3, 2)$, and the constraints $prec(t_1, t_3)$, $non\_overlap(t_1, t_2)$ then the assignment represents in the top half of the figure is not a schedule for the set of tasks $\{t_1, t_2, t_3\}$;

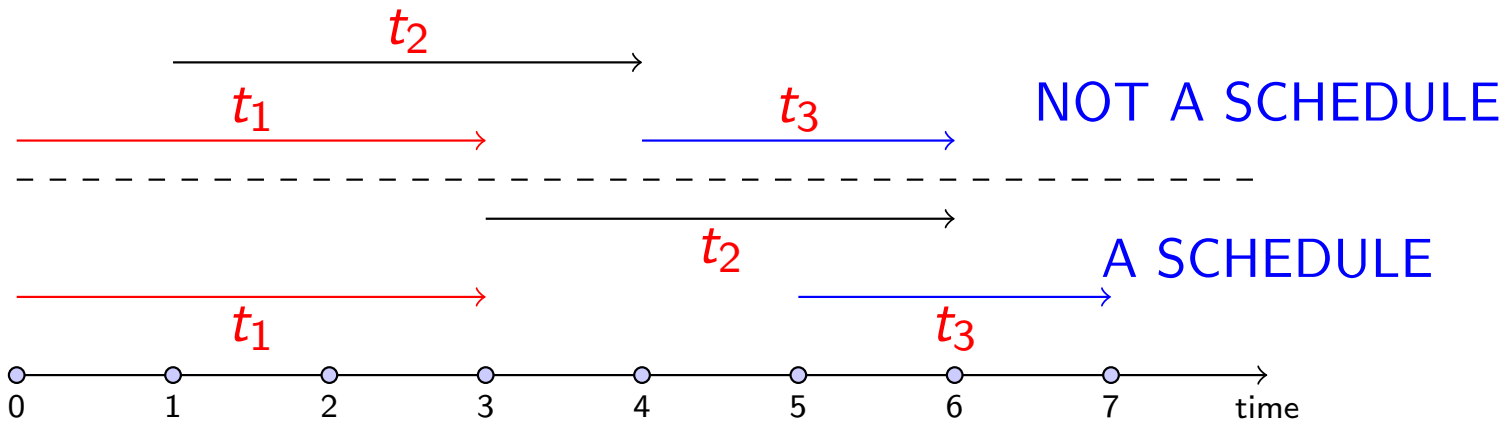


A Schedule for a Set of Tasks: Definition

A schedule for the set of tasks $T = \{t_1, \ldots, t_n\}$ is a mapping of T to the set of non-negative integers $\mathbf{N}$, denoted by $start : T \longrightarrow \mathbf{N}$, such that

- if $prec(t_i, t_j)$ is true then $start(t_i) + d_i \leq start(t_j)$ (t_i completed before t_j)
- if $non\_overlap(t_i, t_j)$ is true then $start(t_i) + d_i \leq start(t_j)$ or $start(t_j) + d_j \leq start(t_i)$

Given three tasks t_1, t_2, t_3 with $duration(t_1, 3)$, $duration(t_2, 3)$, $duration(t_3, 2)$, and the constraints $prec(t_1, t_3)$, $non\_overlap(t_1, t_2)$ then the assignment represents in the top half of the figure is not a schedule for the set of tasks $\{t_1, t_2, t_3\}$; the assignment represents in the bottom half is.

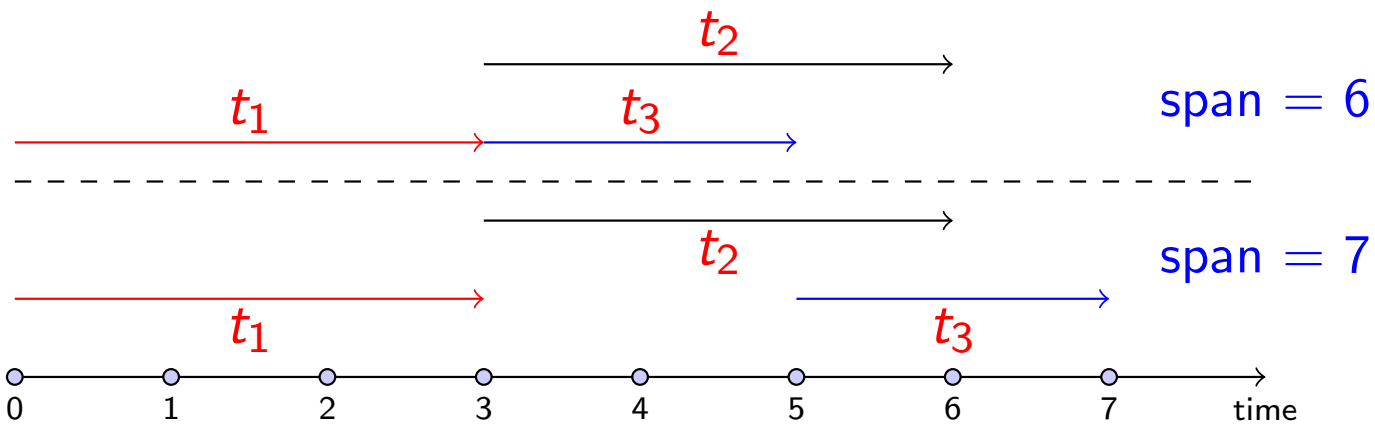


Span of a Schedule: Definition

A schedule for the set of tasks $T = \{t_1, \ldots, t_n\}$ is a mapping of T to the set of non-negative integers $\mathbf{N}$, denoted by $start : T \longrightarrow \mathbf{N}$, such that

- if $prec(t_i, t_j)$ is true then $start(t_i) + d_i \leq start(t_j)$ (t_i completed before t_j)
- if $non\_overlap(t_i, t_j)$ is true then $start(t_i) + d_i \leq start(t_j)$ or $start(t_j) + d_j \leq start(t_i)$

The span of a schedule is defined by the formula
$span = max\_end - min\_start$ where
$max\_end = \max\{start(t_i) + d_i \mid i = 1, \ldots, n\}$ and
$min\_start = \min\{start(t_i) \mid i = 1, \ldots, n\}$.



ASP Encoding for Scheduling

Input: assume that the problem is given ...

$task(t_1), \ldots, task(t_n)$, $duration(t_1, d_1), \ldots, duration(t_n, d_n)$
$prec(t_i, t_j), \ldots,$ $non\_overlap(t_i, t_j), \ldots,$

Code

ASP Encoding for Scheduling

Input: assume that the problem is given ...

$task(t_1), \ldots, task(t_n)$, $duration(t_1, d_1), \ldots, duration(t_n, d_n)$
$prec(t_i, t_j), \ldots,$ $non\_overlap(t_i, t_j), \ldots,$

Code

```
time(0..length).
% generating start time
1 { start(T, S) : time(S) } 1 :- task(T).
```

ASP Encoding for Scheduling

Input: assume that the problem is given ...

$task(t_1), \ldots, task(t_n)$, $duration(t_1, d_1), \ldots, duration(t_n, d_n)$
$prec(t_i, t_j), \ldots,$ $non\_overlap(t_i, t_j), \ldots,$

Code

```
time(0..length).
% generating start time
1 { start(T, S) : time(S) } 1 :- task(T).
% checking prec
:- prec(T1,T2),start(T1,S1),start(T2,S2),duration(T1,D1), S2<S1+D1.
```

ASP Encoding for Scheduling

Input: assume that the problem is given ...

$task(t_1), \ldots, task(t_n)$, $duration(t_1, d_1), \ldots, duration(t_n, d_n)$
$prec(t_i, t_j), \ldots,$ $non\_overlap(t_i, t_j), \ldots,$

Code

```
time(0..length).
% generating start time
1 { start(T, S) : time(S) } 1 :- task(T).
% checking prec
:- prec(T1,T2),start(T1,S1),start(T2,S2),duration(T1,D1), S2<S1+D1.
% non-overlap
:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T1, D1),
   S2 < S1+D1, S2 ≥ S1.
:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T2, D2),
   S1 < S2+D1, S1≥ S2.
```

ASP Encoding for Scheduling

Input: assume that the problem is given ...

$task(t_1), \ldots, task(t_n)$, $duration(t_1, d_1), \ldots, duration(t_n, d_n)$
$prec(t_i, t_j), \ldots,$ $non\_overlap(t_i, t_j), \ldots,$

Code

```
time(0..length).
% generating start time
1 { start(T, S) : time(S) } 1 :- task(T).
% checking prec
:- prec(T1,T2),start(T1,S1),start(T2,S2),duration(T1,D1), S2<S1+D1.
% non-overlap
:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T1, D1),
   S2 < S1+D1, S2 ≥ S1.
:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T2, D2),
   S1 < S2+D1, S1≥ S2.
% minimizing span
max_end(M):- M=#max {D+S : task(T),duration(T,D),start(T,S)}.
min_start(MS) :- MS = #min {S : task(T), start(T,S)}.
span(MA - MS) :- max_end(MA), min_start(MS).
#minimize {S : span(S)}.
```

Planning with Durative Actions and Numerical Fluents

Assumptions in Planning

- Actions are duration-less: every action takes one time step to complete (sometime, it is also referred to as instantaneous) (not realistic, e.g., driving takes time)
- Fluents are boolean: simplified the modeling but could increase the grounding (e.g., the amount of gasoline in the tank is a real number)

Example 1

John is at the train station and wants to go to class. It is 7am and John has a class at 8am. The train starts at 7:05 am and will take 40 minutes to get to the school. Walking from the train station to the class will require 15 minutes. John would like to have a cup of coffee before the class (otherwise, he will just sleep in the class :-). Standing in line to get the coffee at the coffee stand will take 10 minutes. He could also buy coffee on the train. What should John do?

Planning with Durative Actions and Numerical Fluents

Assumptions in Planning

- Actions are duration-less: every action takes one time step to complete (sometime, it is also referred to as instantaneous) (not realistic, e.g., driving takes time)
- Fluents are boolean: simplified the modeling but could increase the grounding (e.g., the amount of gasoline in the tank is a real number)

Example 2

John needs to catch his flight out of El Paso airport at 9 am. Driving from Las Cruces to El Paso airport takes 45 minutes during non-rush hours. It could take 1 hour and 15 minutes during rush hours. Rush hours is between 7:30 am and 9:30 am. John needs to be at the airport 15 minutes earlier to check in and pass through security. When should John start driving?

Planning with Durative Actions and Numerical Fluents

Problems and Ideas

- Representation of durative actions: this can be done similar to what we use in scheduling; however, there is something more than that
  - fixed duration: the train trip between two stations (John's home and school) takes 40 minutes. In this case, we can use a fact of the form $duration(a, d)$ to denote that the execution of a takes d time units.
  - variable duration: the time needed for driving from Las Cruces to El Paso depends on how fast John drives. In this case, we need to introduce a rule that defines $duration(a, d)$, for example,
    duration(drive_lc_elp, 45/Speed) :- speed(drive_lc_elp, Speed).
    duration(drive(A,B),D/S):-speed(A,B,S),distance(A,B,D).
- Representation of numerical fluents: use equality, for example,
  holds(amount_gasoline = 10, T+1) :- ....

References

The encoding of dynamic domains using ASP is developed by several authors. It has its root in the investigation of reasoning about actions and changes and the development of action languages. Some representative references are:

- Michael Gelfond, Vladimir Lifschitz (1998). Action Languages, Linköping Electronic Articles in Computer and Information Science, vol 3, nr 16.
- Tran Cao Son, Chitta Baral, Tran Hoai Nam, and Sheila McIlraith, Domain-Dependent Knowledge in Answer Set Planning, ACM TOCL, 2006.
- 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.

Outline

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 know), and deal with exceptions and preferences whenever they arise.

Representing Defaults

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: Example 1

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

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

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.

Representing Defaults: Example 2 (Birds and Penguins) I

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.

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

Answer set of Π_b: ?

Representing Defaults: Example 2 (Birds and Penguins) II

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

Answer set of Π_b:
$\{bird(tim), fly(tim), penguin(tweety), ab(r_1, tweety), bird(tweety)$
$\Pi_b \models fly(tim)$ and $\Pi_b \models \neg fly(tweety)$

Defaults – Example 3 (Animals) I

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 I

- 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:

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

Defaults – More Examples II

- The Boeing 747, concord, and FA21314 are airplanes. Airplanes normally fly unless they are out of order. FA21314 is out of order. $\Pi_{airplanes}$ is given below:

$$
\begin{array}{lll}
r_1: & fly(X) & \leftarrow \; airplane(X), \textbf{not } ab(X) \\
r_2: & ab(X) & \leftarrow \; out\_of\_order(X) \\
r_3: & airplane(boeing\_747) & \leftarrow \\
r_4: & airplane(concord) & \leftarrow \\
r_5: & airplane(fa21324) & \leftarrow \\
r_6: & out\_of\_order(fa21324) & \leftarrow
\end{array}
$$

Inheritance Reasoning I

In practical applications, there might be more than one defaults and they interact with each others.

Example

Consider the knowledge base with the following statements:

- Normally, students are young adults.
- Normally, students are single.
- Young adults are adults.
- Normally, adults are married.

How to answer the following questions?

- Sam is a young adult. Is Sam married?
- Marry is an adult and she is a student.
- John is an adult. Is he a student?

Inheritance Reasoning II

The normal encoding of $\Pi_{adult\_student}$ is given below:

$$
\begin{array}{lll}
r_1: & young\_adult(X) & \leftarrow student(X), \textbf{not } ab(r_1, X) \\
r_2: & \neg married(X) & \leftarrow student(X), \textbf{not } ab(r_2, X) \\
r_3: & adult(X) & \leftarrow young\_adult(X) \\
r_4: & married(X) & \leftarrow adult(X), \textbf{not } ab(r_4, X) \\
r_5: & young\_adult(sam) & \leftarrow \\
r_6: & adult(marry) & \leftarrow \\
r_7: & student(marry) & \leftarrow \\
r_8: & adult(john) & \leftarrow
\end{array}
$$

This program is inconsistent because of the conflict information about Marry! The information that Marry is a student is more specific than that of her being an adult. Therefore, information about her being single should override that she is married.

Inheritance Reasoning III

Adding rules that allow for information about Marry being married to be overridden:

$$
\begin{array}{lll}
r_1: & young\_adult(X) & \leftarrow student(X), \textbf{not } ab(r_1, X) \\
r_2: & \neg married(X) & \leftarrow student(X), \textbf{not } ab(r_2, X) \\
r_3: & adult(X) & \leftarrow young\_adult(X) \\
r_4: & married(X) & \leftarrow adult(X), \textbf{not } ab(r_4, X) \\
r_4': & ab(r_4, X) & \leftarrow student(X), \textbf{not } ab(r_4', X) \\
r_5: & young\_adult(sam) & \leftarrow \\
r_6: & adult(marry) & \leftarrow \\
r_7: & student(marry) & \leftarrow \\
r_8: & adult(john) & \leftarrow
\end{array}
$$

This rule says that normally, students are exceptions to the default r_4 (normally, adults are married!). It is derived from the principle:

Specificity principle

normally, more specific information overrides less specific one!

Outline

Additional Features for KRR

To add expressiveness and to make logic programming more suitable 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.
- *nested expression*: allowing **not not** l.

Using Classical Negation I

$$\Pi_b = \left\{ \begin{array}{lll} r_1: & fly(X) & \leftarrow bird(X), \textbf{not } ab(r_1, X) \\ r_2: & bird(X) & \leftarrow penguin(X) \\ r_3: & ab(r_1, X) & \leftarrow penguin(X) \\ r_4: & penguin(tweety) & \leftarrow \\ r_5: & bird(tim) & \leftarrow \end{array} \right.$$

- 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

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

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

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

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 fly" and "does Tweety fly?"

Using Classical Negation II

- Suppose that we have a list of professors 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:

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

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

Closed World Assumption (CWA)

The Closed World Assumption is invented by Ray Reiter in his study on deductive database. It has since been used in several areas of knowledge representation. The key idea of the CWA is the presumption that whatever is true is also known to be true. In other words, whatever is not true is **false**.

We have used the CWA in representing the initial state of planning problems. For example, to describe the initial state of the block world domain, we list all fluents that are true and add a rule that completes the initial state:

```
holds(on(a,b), 0).
holds(ontable(b), 0).
....
holds(clear(a), 0).
```

```
holds(neg(F), 0) :- fluent(F), not holds(F, 0).
```

The last rule is often called CWA rule.

Outline

Advanced Problems in ASP

- Planning with durative actions and resources
- Integration of ASP with external solvers
- Multi-agent systems with ASP

Planning with durative actions and resources

Problem

Given a planning problem with durative actions and resources. How to solve this type of problems using ASP?
The general idea is simple but scalable and efficient solution is yet to be developed.

Example 1

In the block world domain, we can imagine that each block has some weights. The cost of moving a block to the table or atop of another block is given in some way (e.g., the robot arm needs gasoline to move around and the amount of gasoline is proportional to the weight of the block).

Planning with durative actions and resources

Problem

Given a planning problem with durative actions and resources. How to solve this type of problems using ASP?

The general idea is simple but scalable and efficient solution is yet to be developed.

Example 2

Suppose that we are at the headquarter of FedEx. We need to create a delivery plan for packets that we receive in the day. Assume that the packets have been sorted, i.e., origins and destinations and type (e.g., express, normal, one-day, etc.) of packets are known. FedEx has airplanes and trucks for transportation. The delivery plan would need to take into consideration information such as

- the duration of each action ($fly(from, to)$ or $drive(from, to)$)
- the type of packet which implies the dateline for delivery
- the availability of the transportation means

Integration of ASP with external solvers

- Early ASP-solver is developed to run in batch mode. There is no interaction with users during the answer set computation process. This might not be an optimal way to deal with problems that are, for example, interactive in nature. For instance, if we were to use ASP to analyze video stream of an airport to identify terrorists, we cannot wait until we get the complete video. It will never be complete or if it is complete then the terrorist might have been escaped already.
- The method of computing answer sets by ASP-solvers is dominantly two steps: first the system grounds the program and then it computes the answer sets. As such, scalability is a challenge for ASP-based applications. For example, we cannot consider variables with continuous domains such as the amount of gasoline, probability, etc.

Integration of ASP with external solvers: Current Directions

- Reactive ASP: for stream/reactive reasoning (e.g., elevator controller)
- Multi-shot ASP: connection with Python (e.g., user interface, game)
- Integration with external solver
  - database (or ontology) solver for Semantic Web applications
  - constraint solver for dealing with real number
- Integration with natural language processor to acquire knowledge, represent it as an ASP program, and reason about it. See presentation on ASP and HRI.

Multi-Agent Systems

- See agent based modeling slides
- Platform for agent based modeling