Logic Programming

Foundations and Applications

Tran Cao Son[1], Enrico Pontelli[1], Marcello Balduccini[2]

[1] Department of Computer Science
New Mexico State University
Las Cruces, New Mexico

[2] Department of Computer Science
Drexel University
Philadelphia, PA

Roadmap

Background

- Answer Set Programming
- Constraint Logic Programming
- Constraint Answer Set Programming

Reasoning about Action and Change

- Action Description Languages

ASP and CLP Planning

- Answer Set Planning and CLP Planning

Applications

- Scheduling
- Goal Recognition Design
- Generalized Target Assignment and Path Finding
- Distributed Constraint Optimization Problems

- Conclusions

Outline

Introduction

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, Π_P^2, 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.

Rules and Constraints

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

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

Rules and Constraints

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

- a_i, b_j: atom of a language L (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

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

- a_i, b_j: atom of a language L (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

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

- $head(r) = \{b_1, \ldots, b_m\}$
- $pos(r) = \{a_1, \ldots, a_n\}$ (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. ($ground(\pi)$)

L is a propositional language

L: set of propositions such as p, q, r, a, b ...

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

L is a first order language

L has one function symbol f (arity: 1) and one predicate symbol p (arity 1)

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

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

$$Q_3 = \begin{cases} p(f(X)) \leftarrow \\ p(f(f(f(X)))) \leftarrow p(X) \end{cases}$$

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

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

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

$$P_3 = \begin{cases} a \leftarrow \\ b \leftarrow c \end{cases}$$

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

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

- L – a first order language with its usual components (e.g., variables, constants, function symbols, predicate symbols, arity of functions and predicates, etc.)
- U_L – Herbrand Universe of a language L: the set of all ground terms which can be formed with the functions and constants in L.
- B_L – Herbrand Base of a language L: the set of all ground atoms which can be formed with the functions, constants and predicates in L.
- Example: Consider a language L_1 with variables X, Y; constants a, b; function symbol f of arity 1; and predicate symbol p of arity 1.
  - $U_{L_1} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.
  - $B_{L_1} = \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$.

Programs with FOL Atoms

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

The language L of a program Π is often given *implicitly*.

Rules with Variables

ground(r, L): the set of all rules obtained from r by all possible substitution of elements of U_L for the variables in r.

Example

Consider the rule "$p(f(X)) \leftarrow p(X)$." and the language 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, 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

- $ground(r, L)$: the set of all rules obtained from r by all possible substitution of elements of U_L for the variables in r.
- For a program Π:
  - $ground(\Pi, L) = \bigcup_{r \in \Pi} ground(r, L)$
  - L_Π: 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.
  - $ground(\Pi) = \bigcup_{r \in \Pi} ground(r, L_\Pi)$.

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 L_Π.

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.

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

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

Herbrand Interpretation II

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.

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

Semantics – Positive Programs without Constraints II

Note: By $a_0 \leftarrow a_1, \ldots, a_n$ in *ground*(Π) 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 $M_\Pi = \emptyset$.

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.

Semantics – General Logic Programs without Constraints I

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

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

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

1. Deleting from $ground(\Pi)$ any rule $a \leftarrow a_1, \ldots, a_n, \text{ not } a_{n+1}, \text{ not } a_{n+k}$ 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

- 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*

Remark

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

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 $1 \leq i \leq m$.
- Note that (ii) above implies that S is supported by Π.
- It is known that if S is an answer set of Π then

Further intuitions behind the semantics II

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. Consider $\Pi_3 = \{p \leftarrow a. \ ; a \leftarrow \textbf{not } b. \ b \leftarrow \textbf{not } a.\}$

1. For $X_0 = \emptyset$. Take $a \leftarrow \textbf{not } b$: its set of positive atoms in the body is empty and its set of negative atoms in the body of this rule is $\{b\}$ and $\{b\} \cap X_0 = \emptyset$. So, X_0 violates the closedness condition hence it is not closed under Π_3. As such, X_0 cannot be an answer set of Π_3.
2. For $X_7 = \{p, a, b\}t$. Take $a \in X_7$: the only rule in Π_3 whose head is a is the rule $a \leftarrow \textbf{not } b$. The set of positive atoms in the body of this rule is empty 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 and hence 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

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

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

$$\Pi_O = \{r \mid r \in \Pi, \ r \text{ has non-empty head}\} \quad \textit{and} \quad \Pi_C = \Pi \setminus \Pi_O$$

(Π_O and Π_C: set of normal logic program rules and constraints in Π, respectively.

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

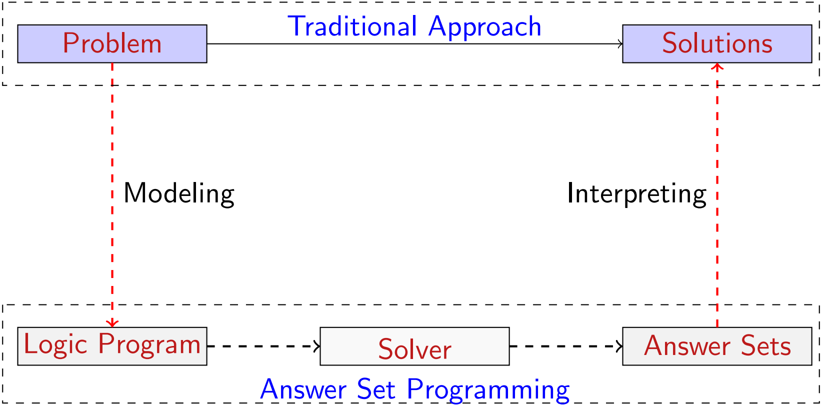
Answer Set Solver clingo

- Download from `https://github.com/potassco/`, set up, etc.
- Run `clingo` <params>
- For example, if $\Pi_2 = \{a \leftarrow \textbf{not}\ b. \quad b \leftarrow \textbf{not}\ a.\}$ is stored in a file named `test.lp`, `clingo test.lp` would output something like the following:

```
clingo test.lp
clingo version 5.2.0
Reading from test.lp
Solving...
Answer: 1
a
SATISFIABLE
...
```

Answer Set Programming

General Methodology



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: $edge(i, j)$.

Graph Coloring II

- Solution representation: use the predicate $color(X, Y)$ - node X is assigned the color Y.
- Generating the solutions: Each node is assigned one color. The three rules

$$color(X, red) \leftarrow \text{not } color(X, blue), \text{ not } color(X, yellow). \quad (2)$$

$$color(X, blue) \leftarrow \text{not } color(X, red), \text{ not } color(X, yellow). \quad (3)$$

$$color(X, yellow) \leftarrow \text{not } color(X, blue), \text{ not } color(X, red). \quad (4)$$

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

$$\leftarrow edge(X, Y), color(X, C), color(Y, C). \quad (5)$$

Graph Coloring III

```
%% description of 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).
```

```
%% generating solution: each node is assigned a color
color(X,red):- node(X), not color(X,blue), not color(X, yellow).
color(X,blue):- node(X), not color(X,red), not color(X, yellow).
color(X,yellow):- node(X), not color(X,blue), not color(X, red).
```

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

(Informal) Theorem

Let G be a graph and Π_G be a program constructed from G. Each solution of G corresponds to an answer set of Π_G and vice versa.

Syntactic Extensions of Logic Programming

Choice Atoms

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

1 {color(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

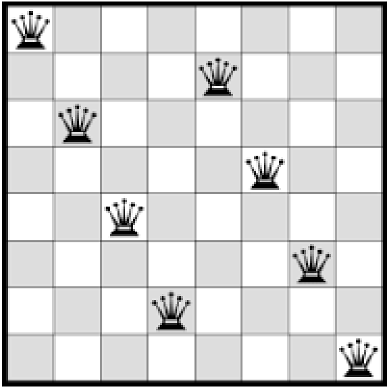
$\mathbf{l}\{l_0 = w_0, \ldots, l_k = w_k, \text{ not } \; l_{k+1} = w_{k+1}, \ldots, \text{ not } \; l_{k+n} = w_{k+n}\}\mathbf{u}$ where l_i's are atoms, w_i are integers, and $\mathbf{l} \leq \mathbf{u}$ are integers. This atom is true with respect to a set of literals S if $\mathbf{l} \leq \sum_{l_j \in S}^{0 \leq j \leq k} w_j + \sum_{l_j \notin S}^{k+1 \leq j \leq k+n} w_j \leq \mathbf{u}$.
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., $\{p(a, X) \mid X \in \{1, 2, 3\}\}$)

Semantics of extensions are well-defined. All features are implemented in answer set solvers.

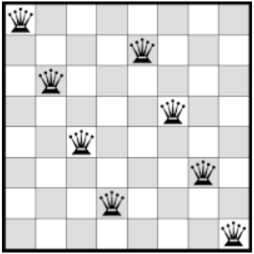
n-Queens



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.
```

Outline

- Explore alternative approaches to encode action languages (e.g., $\mathcal{B}$) using different logic programming (LP) paradigms
  - Explore advantages offered by different paradigms
  - Relate action language features with features from the LP paradigm
- Influence action language design (e.g., $\mathcal{B}^{MV}$)
- Comparative experimental performance testing

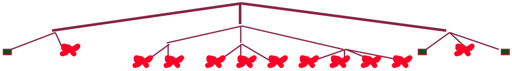
Intuition

- Traditional Logic Programming:
  - Terms are uninterpreted
    ```
    p(X) :- X=square(2).
    ?- p(X).          ?- 3=2+1.   ?- 3=X+1.
    X=square(2)       no          no
    ```
  - Prolog: extra-logical predicates
    ```
    p(X,Y) :- X is Y*2.
    ?- p(6,Y).
    ERROR: is/2:  Arguments not instantiated
    ```
  - Prolog: forces a generate & test style
    ```
    p(X,Y) :- domain(X), domain(Y), X > Y.
    ```
- CLP:
  - Embed interpreted syntax fragments (predefined function symbols and predicates)
  - Embed dedicated solvers to handle them
  - Enable a constrain & generate style

- LP paradigm

  generate then test

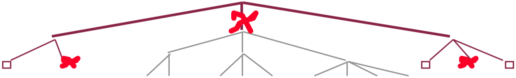
  $\Rightarrow$ many unuseful branches explored



- CLP paradigm

  test then generate

  $\Rightarrow$ cuts branches soon, avoiding exploration



Constraint Logic Programming

$$
\begin{array}{cccccc}
 & S & E & N & D & \\
 & M & O & R & E & + \\
\hline
M & O & N & E & Y &
\end{array}
$$

Constraint Logic Programming

```
solve_naive([S,E,N,D,M,O,R,Y]):-
     Digits1_9 = [1,2,3,4,5,6,7,8,9],
     Digits0_9 = [0|Digits1_9],
     member(S, Digits1_9),
     member(E, Digits0_9), E\=S,
     member(N, Digits0_9), N\=S, N\=E,
     member(D, Digits0_9), D\=S, D\=E, D\=N,
     member(M, Digits1_9), M\=S, M\=E, M\=N, M\=D,
     member(O, Digits0_9), O\=S, O\=E, O\=N, O\=D, O\=M,
     member(R, Digits0_9), R\=S, R\=E, R\=N, R\=D, R\=M, R\=O,
     member(Y, Digits0_9), Y\=S, Y\=E, Y\=N, Y\=D, Y\=M, Y\=O, Y\=R,
     1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E =:=
                              10000*M + 1000*O + 100*N + 10*E + Y.
```

Declarative
1,393,690 Backtrackings

Constraint Logic Programming

```
solve_better([S,E,N,D,M,O,R,Y]):-
        Digits1_9 = [1,2,3,4,5,6,7,8,9],
        Digits0_9 = [0|Digits1_9],
                % D+E = 10*P1+Y
        member(D, Digits0_9),
        member(E, Digits0_9), E\=D,
        Y is (D+E) mod 10, Y\=D, Y\=E,
        P1 is (D+E) // 10, % carry bit
                % N+R+P1 = 10*P2+E
        member(N, Digits0_9), N\=D, N\=E, N\=Y,
        R is (10+E-N-P1) mod 10, R\=D, R\=E, R\=Y, R\=N,
        P2 is (N+R+P1) // 10,
                % E+O+P2 = 10*P3+N
        O is (10+N-E-P2) mod 10, O\=D, O\=E, O\=Y, O\=N, O\=R,
        P3 is (E+O+P2) // 10,
                % S+M+P3 = 10*M+O
        member(M, Digits1_9), M\=D, M\=E, M\=Y, M\=N, M\=R, M\=O,
        S is 9*M+O-P3,
        S>0,S<10, S\=D, S\=E, S\=Y, S\=N, S\=R, S\=O, S\=M.
```

Less Declarative
More Efficient
Requires Order of Execution

Constraint Logic Programming

```
sendmore(Vars) :-
          Vars = [S,E,N,D,M,O,R,Y],
          Vars ins 0..9,
          S*1000+E*100+N*10+D + M*1000+
                O*100+R*10+E #= M*10000+
                O*1000+N*100+E*10+Y,
          S #\= 0, M #\= 0,
          all_different(Vars),
          labeling(Vars).
```

Declarative
More Efficient

declarative programming

Constraint Solving (CS)

Logic Programming (LP)

Constraint Logic Programming (CLP)

- Logic programming extended with terms and predicates defined on non-Herbrand domains
- CLP(X): X is the constraint domain
- Meaning of constraint formulae defined by theory X, not by rules

```
X #< Y :-
                    .....
```

- Prolog solving accumulates substitutions
- CLP solving accumulates constraints (conditions on variables)
- Replace unification with constraint solving
- Constraint Solvers
  - Verify consistency (existence of solutions)
  - Simplify/solve constraints (e.g., reduces to Variable = Value)

- Logic programming extended with terms and predicates defined on non-Herbrand domains
- CLP(X): X is the constraint domain
- Meaning of constraint formulae defined by theory X, not by rules

~~X #< Y :- .....~~

X	Y	X#<Y
0	0	×
0	1	✓
0	2	✓
1	0	×

....

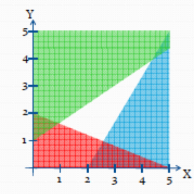
- Prolog solving accumulates substitutions
- CLP solving accumulates constraints (conditions on variables)
- Replace unification with constraint solving
- Constraint Solvers
  - Verify consistency (existence of solutions)
  - Simplify/solve constraints (e.g., reduces to Variable = Value)

- Certain symbols in the syntax have predefined meaning (e.g., $+$, $*$, $/$)
- Variables in constraint terms have specific domains (e.g., integers)
- Note difference:
  - Prolog: $X = Y + 3$ variable assigned term $Y + 3$
  - CLP: $X = Y + 3$ value of X is an integer that satisfies the condition $Y + 3$
- For Example:

```
:- use_module(library(clpr)).
p(X,Y) :- {X=Y*3}, q(X,Y).
q(X,Y) :- {X - 2 = Y}.
```

```
:- p(X,Y).
X=3.0
Y=1.0
```

```
prog(X,Y) :-
        Y #=< -2/5*X+2,
        10*Y #>= 10 + X*7,
        Y #=< (X-2)*5/3.
```




```
?- prog(X,Y).
No
```

Syntax: CLP(X)

- X is a constraint theory
  - A signature $\Sigma_X = (F_X, \Pi_X)$
  - An interpretation structure D
  - A class L of legal formulate (constraints)
    - Typically closed under propositional combination
    - Typically closed under variable renaming
  - Atomic formulae in L: *primitive constraints*

Example: CLP(X)

- $\Sigma_X = (F_X, \Pi_X)$ where
  - $F_X = \{+, -\}$
  - $\Pi_X = \{=, \neq, \leq\} \cup \{\in_n^m \mid n \leq m\}$
- $D = \mathbb{Z}$
- Primitive Constraints: atoms based on $=$, $\neq$, $\leq$, and $\in_n^m$
- Constraints in L:
  - Conjunctions of primitive constraints
  - Each variable should appear in a $\in_n^m$ constraint

- Various X have been formalized

Instance	**Sort**	**Solver**
CLP(FD)	Finite Domains	Local Consistency
CLP(R)	Real, Linear Constraints	Simplex
CLP(Q)	Rationals, Linear Constraints	Simplex
CLP(SET)	Hereditarily Finite Sets	Ad-Hoc, ACI
Prolog-II	Finite Trees	Ad-Hoc

- Signature $\Sigma = (F, \Pi)$ where
  - $F = F_X \cup F_P$
  - $\Pi = \Pi_X \cup \Pi_P$
- Program Rule:

$$p(t_1, \ldots, t_n) :- B_1, \ldots, B_k$$

  where each B_i is a Π_P atom or $B_i \in L$
- Goal:

$$?-\ B_1, \ldots, B_k$$

  where B_i is a Π_P atom or $B_i \in L$

- Example: *In a farmyard, there are only chickens and rabbits. It is known that there are 18 heads and 58 feet. How many chickens and rabbits are there?*

```
counting(Ch,Ra) :- [Ch,Ra] ins 1..58,
               X+Y #= 18, 2*X+4*Y #= 58.
```

Semantics

- State: $\langle G, C\rangle$ where $C \in L$ and G is a goal
  - $\langle \emptyset, C\rangle$ success if *consistent*(C)
  - $\langle G, C\rangle$ failed if $\neg$*consistent*(C)
- Selection Function: $\alpha(G) = B_i$
- Derivation Step: $\langle G, C\rangle \Rightarrow \langle G', C'\rangle$ if $\alpha(G) = B_i$ and
  - $B_i = p(s_1, \ldots, s_n)$ is a Π_P atom and there is a rule $p(t_1, \ldots, t_n) : -\vec{B}$ then
    - $G' = G \setminus \{B_i\} \cup \vec{B} \cup \{s_1 = t_1, \ldots, s_n = t_n\}$
    - $C' = C$
  - $B_i \in L$ then
    - $C' = C \wedge \{B_i\}$
    - $G' = G \setminus \{B_i\}$ if *consistent*(G') or $G' = \emptyset$ if $\neg$*consistent*(G')
- Derivation: $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$ where
  - $S_0 = \langle G, \texttt{true}\rangle$
  - $S_i \Rightarrow S_{i+1}$ for $0 \leq i < n$

CLP(FD)

- $\Pi_X = \{=, \neq, <, >, \leq, \geq\} \cup \{\in_m^n \mid n \leq m\}$
- $F_X = \{+, -, *, /\}$
- SWI-Prolog
  - `X in 1..10        Y in 1..4 \/ 9..12`
  - `X+1 #= Y     X-2#\=Y+1`
  - `X in 1..10, X #< 2 #\/ X #> 9`
- Global Constraints:
  - `sum([X,Y,Z], #>, 10)`
  - `all_different([X,Y,Z])`
- During resolution:
  - Check consistency
  - Possibly simplify constraints:

```
?- X in 1..10, X#>8.
X in 9..10
```

CLP(FD)

- Search:
  - `labeling(+Options,+Variables)`
  - Variable Selection Strategies:
    - `ff`
    - `ffc`
    - `leftmost`
  - Branching Strategy:
    - `step`: `X#=V` or `X#\=V`
    - `enum`: `X=V1` or `X=V2` or ...
    - `bisect`: `X #< M` or `X #>= M`
- Alternating Labeling and propagation
- Optimization: `labeling([max(Expression)], Variables)`

Outline

Constraint Satisfaction: Syntax

A *Constraint Satisfaction Problem (CSP)* is a triple $\langle X, D, C \rangle$, where:

- $X = \{x_1, \ldots, x_n\}$ is a set of variables
- $D = \{D_1, \ldots, D_n\}$ is a set of domains, such that D_i is the domain of variable x_i (i.e. the set of possible values that the variable can be assigned)
- C is a set of constraints.

Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho \rangle$ where σ is a list of variables and ρ is a subset of the Cartesian product of the domains of such variables.

Intensional representation of constraints: an expression involving variables, e.g.

$$x < y$$

A *global constraint* is a constraint that captures a relation between a non-fixed number of variables, such as $sum(x, y, z) < w$ and $all\_different(x_1, \ldots, x_k)$.

Constraint Satisfaction: Example

- Find three integers between 1 and 10 whose sum is 23.
- Additional constraint: only odd integers are allowed.

– How can we formalize it as a CSP? –

Constraint Satisfaction: Example

- Find three integers between 1 and 10 whose sum is 23.
- Additional constraint: only odd integers are allowed.

$$
\begin{array}{l}
P = \langle \\
\ \{x, y, z\}, \\
\ \{\{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}\}, \\
\ \{sum(x, y, z) = 23, x \ \% \ 2 = 1, y \ \% \ 2 = 1, z \ \% \ 2 = 1\} \\
\rangle
\end{array}
$$

Constraint Satisfaction: Semantics

Definition

An *assignment* is a pair $\langle x_i, a \rangle$, where $a \in D_i$

Intuitive meaning: variable x_i is assigned value a.

Definition

- A *compound assignment* is a set of assignments to distinct variables from X.
- A *complete assignment* is a compound assignment to all the variables in X.

Intuitively, a constraint $\langle \sigma, \rho \rangle$ specifies the acceptable assignments for the variables from σ. We say that such assignments *satisfy* the constraint.

Definition

A *solution* to a CSP $\langle X, D, C \rangle$ is a complete assignment satisfying every constraint from C.

Constraint Satisfaction: Example (cont'd)

- Find three integers between 1 and 10 whose sum is 23.
- Additional constraint: only odd integers are allowed.

$$
\begin{array}{l}
P = \langle \\
\ \{x, y, z\}, \\
\ \{\{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}\}, \\
\ \{sum(x, y, z) = 23, x \ \% \ 2 = 1, y \ \% \ 2 = 1, z \ \% \ 2 = 1\} \\
\rangle
\end{array}
$$

- $x := 0$ does not satisfy the constraint $x \ \% \ 2 = 1$.
- $x := 1$ satisfies the constraint $x \ \% \ 2 = 1$.
- $\{x := 9, y := 9, z := 5\}$ is a solution to the CSP.

Constraint ASP (CASP)

- Combines ASP and CSP languages.
- Provides a way of describing a CSP using ASP. Unlike Prolog, host and embedded language feature similar KR paradigms

Example:

given variables: $x, range\ [1, 10]$; $y, range\ [1, 10]$; $z, range\ [1, 10]$

$sum(x, y, z) = 23.$
$x \ \% \ 2 = 1.$
$y \ \% \ 2 = 1.$
$z \ \% \ 2 = 1.$

- Answer set: $\{sum(x, y, z) = 23, x \ \% \ 2 = 1, y \ \% \ 2 = 1, z \ \% \ 2 = 1\}$
- Automatically translated to a CSP
- Solved using a CSP solver

Leveraging both ASP and CSP

given variables: $x, range\ [1, 10]$; $y, range\ [1, 10]$; $z, range\ [1, 10]$

$hard \vee \neg hard.$

$sum(x, y, z) = 23.$
$x\ \%\ 2 = 1.$
$y\ \%\ 2 = 1 \leftarrow hard.$
$z\ \%\ 2 = 1 \leftarrow hard.$

- Answer sets under the ASP semantics:
  1: $\{hard, sum(x, y, z) = 23, x\ \%\ 2 = 1, y\ \%\ 2 = 1, z\ \%\ 2 = 1\}$
  2: $\{\neg hard, sum(x, y, z) = 23, x\ \%\ 2 = 1\}$
- Each answer set is translated to a CSP
- A solution is found when an answer set yields a feasible CSP

Theoretical Foundations of CASP: Language EZCSP

Syntax: extends ASP by pre-interpreted atoms (*CSP atoms*) that encode CSP constraints:

1. *domain*(d): *domain declaration*, e.g.
   - *domain*(*fd*) for numerical constraints over finite domains
   - *domain*(*nlp*) for non-linear constraints
2. *var*(x): x is a CSP variable
   Variant: *var*(x, l, u): x is a CSP variable with range $[l, u]$.
3. *required*(γ): γ is required to be in the CSP.

Possible, but out-of-scope: using EZCSP beyond CASP

Example of Syntax

% Resources
$resource(1).\ resource(2).\ resource(3).$
$available(I) \leftarrow resource(I), \text{not } \neg available(I).$
$\neg available(2).$

% CSP definition:
% $0 < x(I) < 5$ for every I that is available
$domain(fd).$

$var(x(I)) \leftarrow available(I).$

$required(x(I) > 0) \leftarrow available(I).$
$required(x(I) < 5) \leftarrow available(I).$

EZCSP Semantics: Translation Function

- Assumed to exist for a given CSP language
- Bijective
- Maps every CSP constraint η to a valid ASP ground term γ
  - E.g.: $\tau(x > 2.4)$ is $gt(x, \text{“2.4”})$
  - For illustration proposes, we abuse notation and still write $x > 2.4$
- Inverse τ^{-1}: extended to a literal l:

$$\tau^{-1}(l) = \begin{cases} \eta \text{ if } l \text{ is of the form } \mathit{required}(\gamma) \text{ and } \gamma = \tau(\eta) \\ \top \text{ otherwise} \end{cases}$$

Example:

$$\tau^{-1}(\mathit{required}(gt(x, \text{“2.4”}))) \quad \text{is} \quad x > 2.4$$

Answer Sets under the EZCSP Semantics

Given program Π:

- Answer set A of Π "under the ASP semantics": as defined for ASP
- $V(A) = \{v \mid var(v) \in A\}$

Definition

A pair $\langle A, N \rangle$ is an *answer set* of a program Π *under the* EZCSP *semantics* (or an EZCSP *solution*) if-and-only-if:

- A is an answer set of Π under the ASP semantics; and
- N is a solution of the CSP $\langle V(A), \tau^{-1}(A) \rangle$.

Example of Semantics

$$
\begin{array}{l}
resource(1).\ resource(2).\ resource(3).\\
available(I) \leftarrow resource(I), \text{not } \neg available(I).\\
\neg available(2).\\
domain(fd).\\
var(x(I)) \leftarrow available(I).\\
required(x(I) > 0) \leftarrow available(I).\\
required(x(I) < 5) \leftarrow available(I).
\end{array}
$$

Answer set of Π under ASP semantics:

$$
\begin{array}{ll}
A = \{ & resource(1), resource(2), resource(3),\\
& available(1), \neg available(2), available(3),\\
& domain(fd), var(x(1), var(3),\\
& required(x(1) > 0), required(x(1) < 5), required(x(3) > 0), \ldots\}
\end{array}
$$

Example of Semantics

Answer set of Π under ASP semantics:

$$A = \{resource(1), resource(2), resource(3),$$
$$available(1), \neg available(2), available(3),$$
$$domain(fd), var(x(1), var(3),$$
$$required(x(1) > 0), required(x(1) < 5), required(x(3) > 0), \ldots\}$$

- $V(A) = \{x(1), x(3)\}$
- $\tau^{-1}(A) = \{x(1) > 1, x(1) < 5, x(3) > 1, x(3) < 5\}$
- Solutions of $\langle V(A), \tau^{-1}(A)\rangle$:
  $\{x(1) = 1, x(2) = 1\}$, $\{x(1) = 1, x(2) = 2\}$, $\{x(1) = 1, x(2) = 3\}, \ldots$

Answer sets of Π under the EZCSP semantics:

$\langle A, \{x(1) = 1, x(2) = 1\}$

$\langle A, \{x(1) = 1, x(2) = 2\}$

...

Advanced Example

- An object that can travel with linear motion or be idle.
- The object is rotated at an angle of 30 degrees w.r.t. the horizontal axis.
- If the object is held, it remains idle.
- If it is (being) pushed, it travels with constant velocity of 1 m/s in the direction it is facing, unless it is stuck, in which case it remains idle.
- *If the object is not stuck and is pushed, what is its position relative to the origin after 2 seconds?*

Advanced Example

Non-linear dynamics

$$domain(nlp).$$

Advanced Example

If the object is held, it remains idle.

$$
\begin{array}{l}
var(x).\ \ var(y).\\
\neg in\_motion \leftarrow held.\\
required(x = 0) \leftarrow \neg in\_motion.\\
required(y = 0) \leftarrow \neg in\_motion.
\end{array}
$$

Advanced Example

If it is (being) pushed, it travels with constant velocity of 1 m/s in the direction it is facing, unless it is stuck, in which case it remains idle.

$$
\begin{array}{l}
var(a).\ \ var(t).\\
in\_motion \leftarrow pushed, \text{not } \neg in\_motion.\\
required(x = cos(a \cdot \pi/180) \cdot t) \leftarrow in\_motion.\\
required(y = sin(a \cdot \pi/180) \cdot t) \leftarrow in\_motion.
\end{array}
$$

Advanced Example

The object is rotated at an angle of 30 degrees w.r.t. the horizontal axis. If the object is not stuck and is pushed, what is its position relative to the origin after 2 seconds?

$$
\begin{array}{l}
required(a = 30). \\
required(t = 2). \\
pushed. \\
\neg stuck.
\end{array}
$$

Advanced Example

Consider Π_1 as above and $A_1 = Q_1 \cup P_1$ where:
$Q_1 = \{pushed, \neg stuck, in\_motion\}$

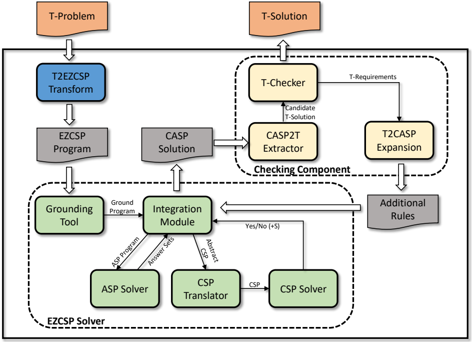
$$P_1 = \begin{cases} domain(nlp), \\ var(x), \quad var(y), \quad var(a), \quad var(t), \\ required(x = cos(a \cdot \pi/180) \cdot t), \quad required(y = sin(a \cdot \pi/180) \cdot t), \\ required(a = 30), \quad required(t = 2) \end{cases}$$

Clearly:

$$\tau^{-1}(A_1) = \tau^{-1}(P_1) = \begin{cases} x = cos(\frac{a \cdot \pi}{180}) \cdot t \\ y = sin(\frac{a \cdot \pi}{180}) \cdot t \\ a = 30 \\ t = 2 \end{cases}$$

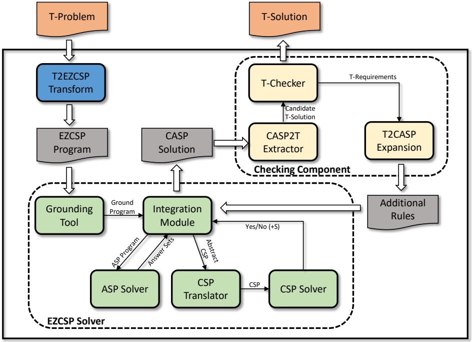
Hence, $\langle A_1, \{t = 2, a = 30, x = 1.7305081, y = 1\}\rangle$ is the answer set of Π_1 under the EZCSP semantics

EZCSP as a Research Tool



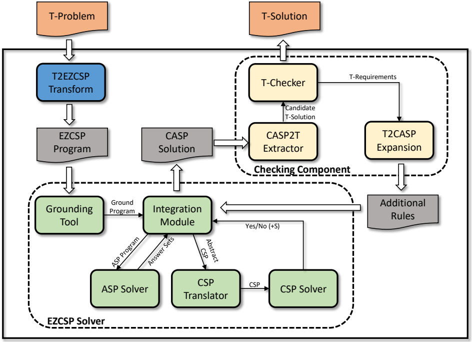
- ASP solver finds an answer set. CP solver finds assignments
- Arbitrary ASP and CP solvers
  - CLP: Sicstus, SWI-Prolo, B-Prolog
  - CSP: Gecode, any Minizinc solvers
  - Algebraic modeling: GAMS
  - Adding new solvers: small translation functions
- Support various degrees of loose-coupling and tight-coupling

EZCSP as a Research Tool



- *T2EZCSP Transform*: T-problem → EZCSP, preserving T-solutions
- *CASP2T Extractor*: answer set → candidate T-solution
- *T-checker*: verifies candidate T-solution
- *T2CASP Expansion*: rejected candidate T-solution → EZCSP rules

EZCSP as a Research Tool



Use case: PDDL+ planning

- *T-problem*: a PDDL+ model
- *T2EZCSP Transform*: PDDL+ to EZCSP
- *CASP2T Extractor*: extracts plans from answer sets
- *T-checker*: VAL [Howey et al., 2004] for extended invariant check
- *T2CASP Expansion*: adds invariants violated by previous answer sets

Performance of EZCSP

- Absolute performance not a main focus...
- ...but still competitive for practical applications
- Allows for prototyping of solving architectures

Domains	EZCSP				CASP solvers	
	CLASP+BP	CLASP+MZN	CLASP+GECODE	PCLASP+BP	CLINGCON	EZCSP+Z3
RF	(11) 10,093.93	N/A	N/A	(10) 9,697.69	**(2) 1,383.02**	–
WS	(30) 18,000.00	(30) 18,000.00	(30) 18,000.00	(13) 7,800.00	**(0) 182.58**	–
IS	(16) 11,176.68	N/A	(14) 8,960.42	(15) 10,182.94	**(12) 7,445.32**	–
*IS**	(21) 14,372.70	(19) 12,466.03	(14) 9,449.68	(20) 13,854.50	N/A	**(5) 3,113.78**

* ASP/CASP competition problems

Domain	EZCSP				CASP solvers	
	CLASP+BP	CMODELS+BP	WASP+BP	PCLASP+BP	CLINGCON	EZSMT+Z3
Car	(0) 0.42	(0) 0.43	(0) 0.78	(0) 2.38	**(0) 0.08**	–
Generator	(2) 1,430.95	(4) 2,400.97	(3) 1,854.91	**(1) 855.54**	(5) 3,3123.84	–

* PDDL+ planning, linear domains

Outline

Motivation

Research on planning requires the resolution of two key problems [GelfondL91]:

- Declarative and Elaboration Tolerant Languages to describe planning domains
- Efficient and Scalable Reasoning Algorithms

Motivation

Research on planning requires the resolution of two key problems [GelfondL91]:

- Declarative and Elaboration Tolerant Languages to describe planning domains
- Efficient and Scalable Reasoning Algorithms

Action Description Languages are formal models to represent knowledge on actions and change (e.g., $\mathcal{A}$ and $\mathcal{B}$ Gelfond and Lifschitz (1991))

Motivation

Research on planning requires the resolution of two key problems [GelfondL91]:

- Declarative and Elaboration Tolerant Languages to describe planning domains
- Efficient and Scalable Reasoning Algorithms

Action Description Languages are formal models to represent knowledge on actions and change (e.g., $\mathcal{A}$ and $\mathcal{B}$ Gelfond and Lifschitz (1991))

Specifications are given through declarative assertions that permit

- to describe actions and their effects on states
- to express queries on the underlying transition system

Action description languages for planning

A *planning domain* D can be described through an action domain description, which defines the notions of

- **Fluents** i.e., *variables* describing the state of the world, and whose value can change
- **States** i.e., possible configurations of the domain of interest: an assignment of values to the fluents
- **Actions** that affect the state of the world, and thus cause the transition from a state to another

A Planning Problem $P = \langle D, I, O \rangle$ includes

- Description (complete or partial) of the Initial state
- Description of the Final state

The language $\mathcal{B}$

Let a be an action and ℓ be a Boolean literal. We have:

- Executability conditions:
  `executable(a, [list-of-preconditions])`
  asserting that the given preconditions have to be satisfied in the current state for the action a to be executable
- Dynamic causal laws:
  `causes(a,` ℓ`, [list-of-preconditions])`
  describes the effect (the fluent literal ℓ) of the execution of action a in a state satisfying the given preconditions
- Static causal laws:
  `caused([list-of-preconditions],` ℓ`)`
  describes the fact that the fluent literal ℓ is true in a state satisfying the given preconditions

The language $\mathcal{B}$: Initial State and Goal

- *Initial state*

$$\texttt{initially}(\ell)$$

asserts that ℓ holds in the initial state.

- *Goal*

$$\texttt{goal}(\ell)$$

asserts that ℓ is required to hold in the final state.

Action description: Example I

- To say that initially, the turkey is walking and not dead, we write

  $\texttt{initially}(\neg \textit{dead})$ and

  $\texttt{initially}(\textit{walking})$

- Initially, the gun is loaded:

  $\texttt{initially}(\textit{loaded})$

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

  $\texttt{causes}(\textit{shoot}, \textit{dead}, [\textit{loaded}])$ and

  $\texttt{causes}(\textit{shoot}, \neg \textit{loaded}, [\textit{loaded}])$

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

  $\texttt{causes}(\textit{load}, \textit{loaded}, [])$ and

  $\texttt{causes}(\textit{unload}, \neg \textit{loaded}, [])$

- Dead turkeys cannot walk

  $\texttt{caused}(\neg \textit{walking}, [\textit{dead}])$

Action description: Example II

- A gun can be loaded only when it is not loaded

  $\texttt{executable}(load, \neg loaded)$

So, an action theory for the Yale Shooting problem is

$$I_y = \{\ \texttt{initially}(\neg dead),\ \texttt{initially}(walking),\ \texttt{initially}(loaded)\}$$

and

$$D_y = \begin{cases} \texttt{causes}(shoot, dead, [loaded]) \\ \texttt{causes}(shoot, \neg loaded, [loaded]) \\ \texttt{causes}(load, loaded, []) \\ \texttt{causes}(unload, \neg loaded, []) \\ \texttt{caused}(\neg walking, [dead]) \\ \texttt{executable}(shoot, []) \\ \texttt{executable}(load, [\neg loaded]) \end{cases}$$

$\mathcal{B}$ vs. PDDL (mostly a 1-1 correspondence, difference in static causal laws)

Domain: D_y in PDDL representation

```
(define (domain yale)
    (:predicates (dead))
    (:action shoot
       :precondition (and (loaded))
       :effect (and (dead) (not loaded)))
...
)
```

Problem: Initial State and Goal in PDDL representation

```
(define (problem yale-1) (:domain yale)
(:objects )
(:init walking )
(:goal (not dead)))
```

$\mathcal{B}$ vs PDDL

$\mathcal{B}$	PDDL
Action	$\surd$
Fluent	Predicate
Conditional Effect	$\surd$
Executability condition	Precondition
Static causal law (allow cyclic)	Defined fluent or axiom (no cyclic)
Ground Instantiations (Variables: shorthand)	Typed Variables

Notes

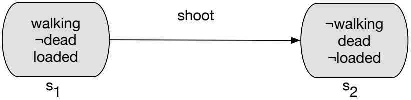
1. Dealing directly with static causal laws is advantageous Thiebaux et al. (2003).
2. Not many planners deal with static causal laws directly.

Action language $\mathcal{B}$ (Semantics) — Intuition

Given an action theory (D, δ), the action domain D encodes a transition system consisting of elements of the form $\langle s_1, a, s_2 \rangle$ where s_1 and s_2 are states of the theory and a is an action that, when executed in s_1, changes the state of the world from s_1 into s_2.

Example

Execution of the action shoot in the domain D_y in the initial state creates the transition $\langle s_1, \mathtt{shoot}, s_2 \rangle$:



Action language $\mathcal{B}$, Complete Information (Semantics)

Given an action domain D, a fluent literal l, sets of fluent literals σ and ψ

- $\sigma \models l$ iff $l \in \sigma$; $\sigma \models \psi$ iff $\sigma \models l$ for every $l \in \psi$.
- σ satisfies a static causal law φ **if** ψ if $\sigma \models \psi$ implies that $\sigma \models \varphi$.
- Closure: $Cn_D(\sigma)$, called the closure of σ, is the smallest set of literals that contains σ and satisfies all static causal laws
- State: *complete* and *consistent* set of fluent literals which *satisfies all* static causal laws.
- Transition Function:
  $\Phi : Actions \times States \rightarrow States$ where

$$\Phi(a, s) = \begin{cases} \{s' \mid s' = Cn_D(de(a, s) \cup (s \cap s'))\} \\ \text{ if } D \text{ contains } a \texttt{ executable} \varphi \text{ and } s \models \varphi \\ \\ \Phi(a, s) = \emptyset \quad \text{otherwise} \end{cases}$$

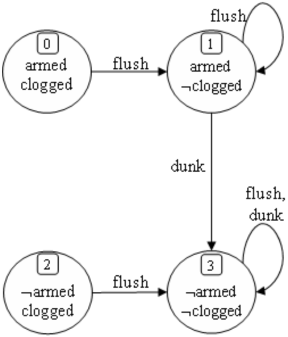
Bomb-In-The-Toilet

There may be a bomb in a package. Dunking the package into a toilet disarms the bomb. This action can be executed only if the toilet is not clogged. Flushing the toilet makes it unclogged.

- Fluents: *armed*, *clogged*
- Actions: *dunk*, *flush*
- Action domain:

$$\mathcal{D}_b = \left\{ \begin{array}{l} \texttt{causes}(dunk, \neg armed, [armed]) \\ \texttt{causes}(flush, \neg clogged, []) \\ \texttt{executable}(dunk, [\neg clogged]) \\ \texttt{executable}(flush, [])^* \end{array} \right.$$

(* — present unless otherwise stated)

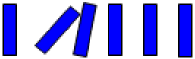


Entailments: $(\mathcal{D}_b, \{armed, clogged\}) \models \neg armed$ **after** $\langle flush, dunk \rangle$

Dominoes

n dominoes $1, 2, \ldots, n$ line up on the table such that if domino i falls down then $i+1$ also falls down.

$$D_d = \begin{cases} \texttt{caused}(down(n+1), [down(n)]) \\ \texttt{causes}(touch(i), down(i), []) \end{cases}$$



It can be shown that

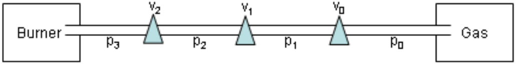
$$(D_d, \delta_d) \models down(n) \textbf{ after } \; touch(i)$$

for every δ_d and i.

Gas Pipe

$n+1$ sections of pipe (pressured/unpressured) connected through n valves (opened/closed) connects a gas tank to burner. A valve can be opened only if the valve on its right is closed. Closing a valve causes the pipe section on its right side to be unpressured. The burner will start a flame if the pipe section connecting to it is pressured. The gas tank is always pressured.

- Fluents: *flame*, *opened*(V), *pressured*(P), $0 \leq V \leq n$, $0 \leq P \leq n+1$,
- Actions: *open*(V), *close*(V)



- Action domain:

$$D_g = \begin{cases} \texttt{executable}(open(V), [\neg opened(V+1)]) \\ \texttt{causes}(open(V), opened(V), []) \\ \texttt{causes}(close(V), \neg opened(V), []) \\ \texttt{caused}(pressured(V+1), [opened(V), pressured(V)]) \\ \texttt{caused}(pressured(0), []) \\ \texttt{caused}(flame, [pressured(n+1)]) \end{cases}$$

Outline

ASP & CLP in Classical Planning

Classical Planning Complexity

Definition (Planning Problem)

- Given: an $\mathcal{B}$-action theory (D, δ), where δ is a state of D, and a set of fluent literals G.
- Determine: a sequence of actions α such that $(D, \delta) \models G \textbf{ after } \alpha$

From [Liberatore (1997); Turner (2002)]:

Theorem (Complexity)

- (D, δ) *is deterministic: NP-hard even for plans of length 1, NP-complete for polynomial-bounded length plans (Classical Planning).*
- (D, δ) *is non-deterministic:* Σ_P^2*-hard even for plans of length 1,* Σ_P^2*-complete for polynomial-bounded length plans (Conformant Planning in non-deterministic theories).*

Early Development of Answer Set Planning

1. Start with [Dimopoulos et al. (1997); Lifschitz (2002); Subrahmanian and Zaniolo (1995)]
2. Planning using answer set programming: prototypical implementation

Given a planning problem $P = (D, I, G)$ in the language B and an integer N, P is encoded as a program $\Pi(P, \mathrm{N})$ consisting of the following sets of rules for

1. declaring the fluents, actions (constants)
2. defining the initial state
3. defining when an action is executable
4. generating action occurrences
5. computing effects of actions, solving frame problem, ramification problem
6. for checking goal conditions

Theorem

Answer sets of $\Pi(P, \mathrm{N})$ 1-to-1 correspond to solutions of length $<$ N of P.

ASP Encoding of $\mathcal{B}$

Ideas from [Gelfond and Lifschitz 92]

We designed a Prolog program that translate an action description D, with initial and final constraints O and a plan length N, in an ASP program $\Pi_D(\mathrm{N}, O)$.

ASP Encoding of $\mathcal{B}$

Ideas from [Gelfond and Lifschitz 92]

We designed a Prolog program that translate an action description D, with initial and final constraints O and a plan length N, in an ASP program $\Pi_D(\mathrm{N}, O)$.

$$\mathrm{time}(0..\mathrm{N})$$

ASP Encoding of $\mathcal{B}$

Ideas from [Gelfond and Lifschitz 92]

We designed a Prolog program that translate an action description D, with initial and final constraints O and a plan length N, in an ASP program $\Pi_D(\mathrm{N}, O)$.

$$\mathrm{time}(0..\mathrm{N})$$

$$\mathrm{fluent}(f). \qquad\qquad \mathrm{action}(a).$$

ASP Encoding of $\mathcal{B}$

Ideas from [Gelfond and Lifschitz 92]

We designed a Prolog program that translate an action description D, with initial and final constraints O and a plan length N, in an ASP program $\Pi_D(\mathrm{N}, O)$.

$$\mathrm{time}(0..\mathrm{N})$$

$$\mathrm{fluent}(f). \qquad \mathrm{action}(a).$$

$$\begin{array}{ll} \mathrm{literal}(F) \text{:} -\mathrm{fluent}(F). & \mathrm{literal}(\mathrm{neg}(F)) \text{:} -\mathrm{fluent}(F). \\ \mathrm{complement}(F, \mathrm{neg}(F)). & \mathrm{complement}(\mathrm{neg}(F), F). \end{array}$$

ASP Encoding of $\mathcal{B}$

The predicate holds(FluentLiteral,Time) is defined using the axioms:

- $\texttt{executable}(a, [\ell_1^1, \ldots, \ell_{r_1}^1])$, ..., $\texttt{executable}(a, [\ell_1^h, \ldots, \ell_{r_h}^h])$:

$$\text{possible}(a, T)\text{:} - \text{time}(T), \text{holds}(\ell_1^1, T), \ldots, \text{holds}(\ell_{r_1}^1, T).$$

$$\ldots$$

$$\text{possible}(a, T)\text{:} - \text{time}(T), \text{holds}(\ell_1^h, T), \ldots, \text{holds}(\ell_{r_h}^h, T).$$

ASP Encoding of $\mathcal{B}$

The predicate holds(FluentLiteral,Time) is defined using the axioms:

- $\texttt{executable}(a, [\ell^1_1, \ldots, \ell^1_{r_1}])$, ..., $\texttt{executable}(a, [\ell^h_1, \ldots, \ell^h_{r_h}])$:

  $$\text{possible}(a, T) \text{:} -\text{time}(T), \text{holds}(\ell^1_1, T), \ldots, \text{holds}(\ell^1_{r_1}, T).$$
  $$\ldots$$
  $$\text{possible}(a, T) \text{:} -\text{time}(T), \text{holds}(\ell^h_1, T), \ldots, \text{holds}(\ell^h_{r_h}, T).$$

- Static causal laws: $\texttt{caused}([\ell_1, \ldots, \ell_r], \ell)$:

  $$\text{holds}(\ell, T) \text{:} -\text{time}(T), \text{holds}(\ell_1, T), \ldots, \text{holds}(\ell_r, T).$$

ASP Encoding of $\mathcal{B}$

The predicate holds(FluentLiteral,Time) is defined using the axioms:

- $\texttt{executable}(a, [\ell_1^1, \ldots, \ell_{r_1}^1])$, ..., $\texttt{executable}(a, [\ell_1^h, \ldots, \ell_{r_h}^h])$:

$$\mathrm{possible}(a, T) :- \mathrm{time}(T), \mathrm{holds}(\ell_1^1, T), \ldots, \mathrm{holds}(\ell_{r_1}^1, T).$$
$$\ldots$$
$$\mathrm{possible}(a, T) :- \mathrm{time}(T), \mathrm{holds}(\ell_1^h, T), \ldots, \mathrm{holds}(\ell_{r_h}^h, T).$$

- Static causal laws: $\texttt{caused}([\ell_1, \ldots, \ell_r], \ell)$:

$$\mathrm{holds}(\ell, T) :- \mathrm{time}(T), \mathrm{holds}(\ell_1, T), \ldots, \mathrm{holds}(\ell_r, T).$$

- Dynamic causal laws: $\texttt{causes}(a, \ell, [\ell_1, \ldots, \ell_r])$:

$$\begin{aligned}\mathrm{holds}(\ell, T+1) :- \;& \mathrm{time}(T), \mathrm{occ}(a, T), \\ & \mathrm{holds}(\ell_1, T), \ldots, \mathrm{holds}(\ell_r, T).\end{aligned}$$

ASP Encoding of $\mathcal{B}$

- State consistency:

$$:-\text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\text{neg}(F), T).$$

ASP Encoding of $\mathcal{B}$

- State consistency:

$$:-\text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\text{neg}(F), T).$$

- Frame problem:

$$\begin{aligned}\text{holds}(L, T+1) \ :- \ & \text{time}(T), \text{literal}(L), \text{holds}(L, T),\\ & \text{complement}(L, L_1), \textbf{not} \ \ \text{holds}(L_1, T+1).\end{aligned}$$

ASP Encoding of $\mathcal{B}$

- State consistency:

$$:-\mathrm{time}(T), \mathrm{fluent}(F), \mathrm{holds}(F, T), \mathrm{holds}(\mathrm{neg}(F), T).$$

- Frame problem:

$$\begin{array}{lcl} \mathrm{holds}(L, T+1) & :- & \mathrm{time}(T), \mathrm{literal}(L), \mathrm{holds}(L, T), \\ & & \mathrm{complement}(L, L_1), \textbf{not } \mathrm{holds}(L_1, T+1). \end{array}$$

- Initial state and goal:

$$\mathrm{holds}(L, 0) :- \mathrm{initially}(L). \qquad :- \mathrm{goal}(L), \textbf{not } \mathrm{holds}(L, \mathrm{N}).$$

ASP Encoding of $\mathcal{B}$

- State consistency:

$$:-\mathrm{time}(T), \mathrm{fluent}(F), \mathrm{holds}(F, T), \mathrm{holds}(\mathrm{neg}(F), T).$$

- Frame problem:

$$\begin{aligned}\mathrm{holds}(L, T+1) :- \ & \mathrm{time}(T), \mathrm{literal}(L), \mathrm{holds}(L, T), \\ & \mathrm{complement}(L, L_1), \textbf{not}\ \mathrm{holds}(L_1, T+1).\end{aligned}$$

- Initial state and goal:

$$\mathrm{holds}(L, 0){:}-\mathrm{initially}(L). \qquad :-\mathrm{goal}(L), \textbf{not}\ \mathrm{holds}(L, \mathrm{N}).$$

- One action per time:

$$1\{\mathrm{occ}(A, T) \ : \ \mathrm{action}(A)\}1{:}-\mathrm{time}(T), T < \mathrm{N}.$$
$$:-\mathrm{action}(A), \mathrm{time}(T), \mathrm{occ}(A, T), \textbf{not}\ \mathrm{possible}(A, T).$$

ASP Encoding of $\mathcal{B}$

- State consistency:

$$:-\text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\text{neg}(F), T).$$

- Frame problem:

$$\begin{aligned}\text{holds}(L, T+1)\ :-\ & \text{time}(T), \text{literal}(L), \text{holds}(L, T),\\ & \text{complement}(L, L_1), \textbf{not}\ \text{holds}(L_1, T+1).\end{aligned}$$

- Initial state and goal:

$$\text{holds}(L, 0) :- \text{initially}(L). \qquad :-\text{goal}(L), \textbf{not}\ \text{holds}(L, \mathrm{N}).$$

- One action per time:

$$\begin{aligned}&1\{\text{occ}(A, T)\ :\ \text{action}(A)\}1 :- \text{time}(T), T < \mathrm{N}.\\ &:-\text{action}(A), \text{time}(T), \text{occ}(A, T), \textbf{not}\ \text{possible}(A, T).\end{aligned}$$

In absence of static laws a simplified mapping has been implemented (leading to smaller ASP programs, but not always to faster executions)

Example: Bomb-in-the-Toilet

%fluents and actions
fluent(*armed*). fluent(*clogged*). action(*dunk*). action(*flush*).

% executable(*dunk*, [¬*clogged*])
executable(*dunk*, T): −time(T), holds(neg(*clogged*), T).
% executable(*flush*, [])
executable(*dunk*, T): −time(T), holds(neg(*clogged*), T).

% causes(*dunk*, ¬*armed*, [*armed*])
holds(neg(*armed*), T): −time(T), occ(*dunk*, T), holds(*armed*, T).
% causes(*flush*, ¬*clogged*, [])
holds(neg(*clogged*), T): −time(T), occ(*flush*, T).

CLP in Classical Planning

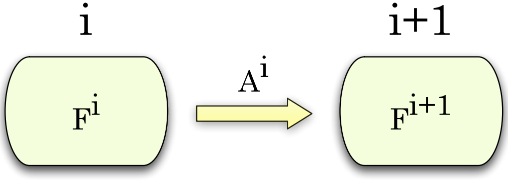
From [Dovier et al. (2010)]

- Action descriptions are mapped to finite domain constraints
- Constrained variables are introduced for fluents and action occurrences
- Executability conditions and causal laws are rendered by imposing constraints
- Solutions of the constrains identify plans and trajectories

Why?

- Declarative encoding, highly elaboration tolerant
- Propagation techniques to prune planning search space
- Sophisticated search techniques
- Global constraints to capture trajectory properties (e.g., control knowledge)
- Natural extensions to action languages (e.g., multi-valued fluents, non-Markovian, costs)

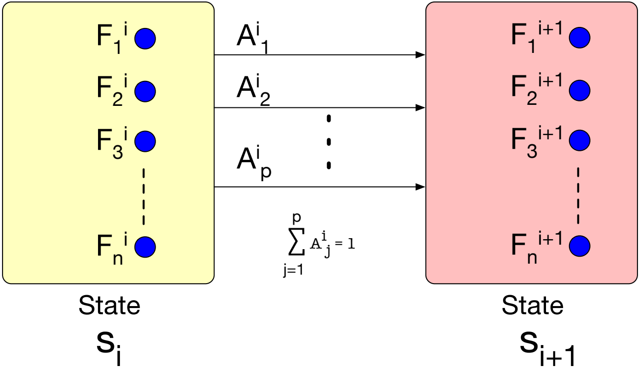
Main idea



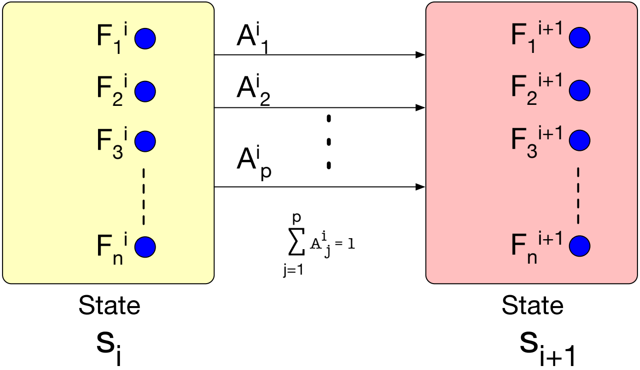
For all states s_i $(0 \leq i \leq \mathrm{N})$:

- Every fluent F is represented by Boolean variable F^i:
  F^i is the value of fluent F in state s_i
- Every action A is represented as a Boolean variable A^i $(i < \mathrm{N})$:
  $A^i = 1$ iff action A is executed in state s_i

Main idea


F_1^i
F_2^i
F_3^i
F_n^i
A^i_1
A^i_2
A^i_p
$\sum_{j=1}^{p} A^i_j = 1$
F_1^{i+1}
F_2^{i+1}
F_3^{i+1}
F_n^{i+1}
State
S_i
State
S_{i+1}

Main idea



$F^{i+1} = 1 \Leftrightarrow$ Action a sets F to true ($A^i = 1$), or
No action that sets F to false is fired and $F^i = 1$

Some Terminology

Formula Projection

φ^i: formula φ projected to state s_i

$$(at(door, x) \land \neg door(closed))^i \Rightarrow at(door, x)^i \land \neg door(closed)^i$$

- $\hat{\alpha}_j$ condition of action A_j making F true
- $\hat{\beta}_k$ condition of action A_k making F false
- $\hat{\delta}_h$ conditions of static causal law making F true
- $\hat{\gamma}_h$ conditions of static causal law making F false
- $\hat{\eta}_r$ condition in one of executability conditions for action A_r

Some constraints

General Conditions

$$A^i_j = 1 \quad \rightarrow \bigvee_{j=1}^{q} \hat{\eta}^i_{r_j}$$ *Only executable actions can occur*

$$\sum_{a_j \in \mathcal{A}} A^i_j = 1$$ *Single action at a time*

Some constraints

General Conditions

$$A^i_j = 1 \quad \rightarrow \bigvee_{j=1}^{q} \hat{\eta}^i_{r_j}$$ *Only executable actions can occur*

$$\sum_{a_j \in \mathcal{A}} A^i_j = 1$$ *Single action at a time*

Positive Fluent F

$$\begin{array}{lcl} \mathrm{PosFired}^i_f = 1 & \Leftrightarrow & \mathrm{PosDyn}^i_f = 1 \vee \mathrm{PosStat}^{i+1}_f = 1 \\ \mathrm{PosDyn}^i_f = 1 & \Leftrightarrow & \bigvee_{j=1}^{m} (\hat{\alpha}^i_j \wedge A^i_{t_j} = 1) \\ \mathrm{PosStat}^i_f = 1 & \Leftrightarrow & \bigvee_{j=1}^{h} \hat{\delta}^i_j \end{array}$$

Some constraints

Negative Fluent F

$$
\begin{array}{lcl}
\text{NegFired}_f^i = 1 & \Leftrightarrow & \text{NegDyn}_f^i = 1 \vee \text{NegStat}_f^{i+1} = 1 \\
\text{NegDyn}_f^i = 1 & \Leftrightarrow & \bigvee_{j=1}^{p} (\hat{\beta}_j^i \wedge A_{z_j}^i = 1) \\
\text{NegStat}_f^i = 1 & \Leftrightarrow & \bigvee_{j=1}^{k} \hat{\gamma}_j^i
\end{array}
$$

Some constraints

Negative Fluent F

$$\begin{aligned}
\text{NegFired}_f^i = 1 &\Leftrightarrow \text{NegDyn}_f^i = 1 \vee \text{NegStat}_f^{i+1} = 1 \\
\text{NegDyn}_f^i = 1 &\Leftrightarrow \bigvee_{j=1}^{p} (\hat{\beta}_j^i \wedge A_{z_j}^i = 1) \\
\text{NegStat}_f^i = 1 &\Leftrightarrow \bigvee_{j=1}^{k} \hat{\gamma}_j^i
\end{aligned}$$

Target for F

$$\text{PosFired}_f^i = 0 \vee \text{NegFired}_f^i = 0$$

$$F^{i+1} = 1 \Leftrightarrow \text{PosFired}_f^i = 1 \vee (\text{NegFired}_f^i = 0 \wedge F^i = 1)$$

Some constraints

Negative Fluent F

$$
\begin{aligned}
\text{NegFired}_f^i = 1 &\Leftrightarrow \text{NegDyn}_f^i = 1 \vee \text{NegStat}_f^{i+1} = 1 \\
\text{NegDyn}_f^i = 1 &\Leftrightarrow \bigvee_{j=1}^{p} (\hat{\beta}_j^i \wedge A_{z_j}^i = 1) \\
\text{NegStat}_f^i = 1 &\Leftrightarrow \bigvee_{j=1}^{k} \hat{\gamma}_j^i
\end{aligned}
$$

Target for F

$$
\begin{aligned}
&\text{PosFired}_f^i = 0 \vee \text{NegFired}_f^i = 0 \\
&F^{i+1} = 1 \Leftrightarrow \text{PosFired}_f^i = 1 \vee (\text{NegFired}_f^i = 0 \wedge F^i = 1)
\end{aligned}
$$

Constraints for State

- C_F^i conjunction of all constraints for F and i
- C_F^i conjunction of all C_F^i for each $F \in F$ set of fluents in the language

Some Theoretical Results

- State u represented by variable assignment σ_u
- Action occurrence A in state i represented by variable assignment σ^i_A

Theorem

Given an action theory D, if the transition $\langle s_i, A, s_{i+1}\rangle$ is possible in the transition system of D, then $\sigma_{s_i} \cup \sigma_{s_{i+1}} \cup \sigma_A$ is a solution of $C_F^{s_i}$.

Reverse is not as trivial

Some Theoretical Results

- Action theory with $F = \{f, g, h\}$ and action $A = \{a\}$ such that a is always executable and

$$\mathtt{causes}(a, f, true) \quad \mathtt{caused}(g, h) \quad \mathtt{caused}(h, g)$$

- Transition: $s_0 = \{\neg f, \neg g, \neg h\} \xrightarrow{a} s_1 = \{f, \neg h, \neg g\}$
- Constraints:

$$\begin{array}{c} F^1 = 1 \Leftrightarrow F^0 = 1 \vee A^0 = 1 \\ G^1 = 1 \Leftrightarrow G^0 = 1 \vee H^1 = 1 \\ H^1 = 1 \Leftrightarrow H^0 = 1 \vee G^1 = 1 \\ F^i \in \{0,1\} \wedge G^i \in \{0,1\} \wedge H^i \in \{0,1\} \end{array}$$

- If we set $F^0 = 0, G^0 = 0, H^0 = 0$ and $A^0 = 1$, there are two solutions:

  1. $F^1 = 1, G^1 = 0, H^1 = 0$

Some Theoretical Results

- Action theory with $F = \{f, g, h\}$ and action $A = \{a\}$ such that a is always executable and

$$\texttt{causes}(a, f, true) \quad \texttt{caused}(g, h) \quad \texttt{caused}(h, g)$$

- Transition: $s_0 = \{\neg f, \neg g, \neg h\} \xrightarrow{a} s_1 = \{f, \neg h, \neg g\}$
- Constraints:

$$\begin{array}{c} F^1 = 1 \Leftrightarrow F^0 = 1 \vee A^0 = 1 \\ G^1 = 1 \Leftrightarrow G^0 = 1 \vee H^1 = 1 \\ H^1 = 1 \Leftrightarrow H^0 = 1 \vee G^1 = 1 \\ F^i \in \{0,1\} \wedge G^i \in \{0,1\} \wedge H^i \in \{0,1\} \end{array}$$

- If we set $F^0 = 0$, $G^0 = 0$, $H^0 = 0$ and $A^0 = 1$, there are two solutions:

  1. $F^1 = 1, G^1 = 0, H^1 = 0$
  2. $\boxed{F^1 = 1, G^1 = 1, H^1 = 1.}$

Some Theoretical Results

- Problem: cyclic dependencies generated by static causal laws
- Dependency Graph of static causal laws: caused$([\ell_1, \ldots, \ell_k], \ell)$ creates edges (ℓ_i, ℓ).
- Construct a set of constraints $CONS(\ell_i)$ to defeat self-sustaining loop: loop $\ell_1, \ell_2, \ldots, \ell_m$
  - for each causes(a_j, ℓ_i, α) add to constraint $A^u_j = 0$ or $F^u_\ell = 0$ for some $\ell \in \alpha$
  - for each caused(γ, ℓ_i) add to constraint $F^u_\ell = 0$ for some $\ell \in \gamma$
  - add to constraint $F^u_{\ell_i} = 0$ or $F^{u+1}_{\ell_i} = 0$

  Generate constraints $c_1 \wedge \cdots \wedge c_m \Rightarrow F^{u+1}_{\ell_1} = 0 \wedge \cdots \wedge F^{u+1}_{\ell_m} = 0$

The language $\mathcal{B}^{MV}$

(Multi-)Fluents: introduced through *domain declarations*:

$$\mathtt{fluent}(\mathtt{f}, \mathtt{v}_1, \mathtt{v}_2), \quad \mathtt{fluent}(\mathtt{f}, \mathtt{Set})$$

Annotated Fluents: modeling backward references:

$$\mathtt{f}^{-\mathtt{a}} \qquad \text{with } a \in \mathbb{N}$$

Fluent Expressions:

$$\mathtt{FE} ::= \mathtt{n} \mid \mathtt{AF} \mid \mathtt{abs}(\mathtt{FE}) \mid \mathtt{FE}_1 \oplus \mathtt{FE}_2 \mid \mathtt{rei}(\mathtt{FC})$$

with $\oplus \in \{+, -, *, /, \mathtt{mod}\}$

Fluent Constraints:

$$\mathtt{FC} ::= \mathtt{FE}_1 \ \mathtt{op} \ \mathtt{FE}_2$$

with $\mathtt{op} \in \{=, \neq, \geq, \leq, <, >\}$

Action description specification

- Dynamic causal laws

$$\texttt{causes(a, C}_1\texttt{, C)}$$

- Static causal laws

$$\texttt{caused(C, C}_1\texttt{)}$$

- Executability conditions

$$\texttt{executable(a, C)}$$

where a is an action, $\texttt{C}_1$ is a fluent constraint, and C a conjunction of fluent constraints.

Main advantages

- It allows the compact representation of numerical domains.
- An encoding in **B** is still possible, but the number of fluents explodes
- Consider
$$\texttt{causes}(a, f = f^{-1} + 1, [])$$
with $\text{dom}(f) = [1..100]$.
  - In B:
$$\texttt{causes}(a, f_2, [f_1]). \quad \ldots \quad \texttt{causes}(a, f_{100}, [f_{99}])$$
  - Alternative encodings (e.g., bit-based) incur analogous problems
- The CLP(FD) mapping requires minor changes.

The Languager $\mathcal{B}^{MV}$

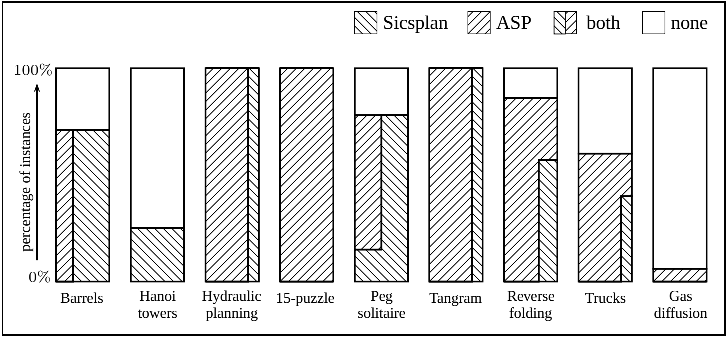
Sketch of encoding (no static causal laws); for fluent f:

F_f^i, F_f^{i+1}	$\in$	$dom(f)$	*Fluent domains*
$A_a^i = 1$	$\Rightarrow$	$\bigvee_{\texttt{executable}(a,\delta)} \delta^i$	*Only executable actions*
$A_a^i = 1 \wedge \alpha^i$	$\Leftrightarrow$	$Dyn_a^i = 1$	*a has f in consequence;* $\texttt{causes}(a, C, \alpha)$
$Dyn_a^i = 1$	$\Rightarrow$	C^{i+1}	*a has f in consequence;* $\texttt{causes}(a, C, \alpha)$
$\sum_{a_f \text{ affects } f} Dyn_{a_f}^i = 0$	$\Rightarrow$	$F_f^i = F_f^{i+1}$	*Inertia*

Experimental comparison: Benchmarks used

- *Hydraulic planning* (by Michael Gelfond et al., ASP 2009)
- *Peg Solitaire* (ASP 2009)
- *Sam Lloyd's 15 puzzle* (ASP 2009)
- *Towers of Hanoi* (ASP 2009)
- The *trucks* domain from the (IPC5)
- A generalized version of the classical *3-barrels* problem
- The *Gas Diffusion problem*
- The *reverse folding problem*.
- The *Tangram* puzzle.

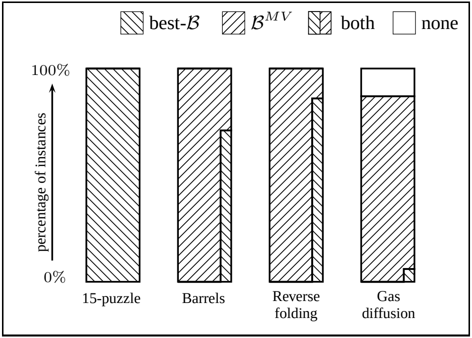
ASP (clasp) vs CLP (SICStus)



Experimental comparison

Instance	Length	ASP	CLP	B^{MV}
Barrel-20-11-9	18	185+43.71	3+102	0.65
Barrel-20-11-9	21	189+4.39	2+80	0.53
Community C_5	6	MEM	1946	888
Community D_5	6	MEM	1519	1802
8-tile Puzzle I_4	25	55+437	2+79	57
Wolf-Goat-Cabbage	35	0.2+1.39	0.15+0.54	0.4
Wolf-Goat-Cabbage	36	0.2+4.24	0.13+1.87	1.9

$\mathcal{B}$ best vs $\mathcal{B}^{MV}$(CLP)



Planning with Preferences and Domain Knowledge

Preferences

- Common feature in several research areas
  - Mathematics and physics: minimum and maximum (e.g., greatest and least fix-point solutions)
  - Economics: the best decision
  - AI: preferred solutions
  - ...
- Property of preferences
  - Soft constraint on solutions, i.e., not a "must be satisfied" property of a solution

Issues

- Representation
  - languages for preferences representation
  - types of preferences
- Semantics
  - combining preferences
  - incomplete and/or inconsistent preferences
- Computation (algorithms, complexity for dealing with preferences)
- Applications

Example

- Planning problem: Going to the airport
- Several possible solutions
  - Drive (get into the car; drive to the airport)
  - Taxi (call a taxi; ask to be at the airport)
  - Train (go to the train station; take the train to the airport)
  - Walk (walk to the airport)
- Question: which plan?

Motivation

Motivation

- Planning with Preferences
  - Several plans achieve the goal
  - Finding a plan is easy
  - Users have preference over the plans that will be executed
- Preference vs. Goal
  - Goals — hard constraints
  - Preferences — soft constraints
- Question: How to compute 'preferred plans' using ASP? If so, how?

PP: A language for Planning with Preferences

- Preferences are described through formulae of different complexity
  - Basic desires
  - Atomic preferences
  - General preferences
- Preferences formulae will be evaluated with respect to trajectories (s_i: state, a_i: action)

$$\alpha = s_1 a_1 s_2 a_2 \dots a_n s_{n+1}$$

- A more preferred relationship between trajectories is defined
- Most preferred trajectories will be computed

Basic Desires

- State preference (Point-wise preference)
  - *occ*(a): action a should occur in the current state
  - ψ: fluent formula ψ should hold in the current state
  - *goal*(ψ): fluent formula ψ should hold in the final state
- Basic desires (Trajectory preference)
  - Temporal formulae over state preferences constructed using
    - propositional operators ($\wedge, \vee, \neg$)
    - temporal operators (**next** $\bigcirc$, **eventually** $\Diamond$, **always** $\Box$, **until** $\bigcup$)

Preference Relation over Basic Desires

- Given $\alpha = s_1 a_1 s_2 a_2 \ldots a_n s_{n+1}$ and basic desire φ. We say α satisfies φ, denoted by $\alpha \models \varphi$, if
  - $\varphi = occ(a)$ and $a_1 = a$; or
  - $\varphi = \psi$ where ψ is a fluent formula and ψ holds in s_1; or
  - $\varphi = goal(\psi)$ and ψ holds in s_{n+1}; or
  - $\varphi = \varphi_1 \wedge \varphi_2$ and $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$ (similarly for $\vee$, $\neg$); or
  - $\varphi = \mathbf{next}(\varphi_1)$ and $\alpha[1] \models \varphi_1$ where $\alpha[1] = s_2 a_2 \ldots a_n s_{n+1}$ (similarly for **eventually**, **always**, **until**)
- α preferred to β with respect to φ if $\alpha \models \varphi$ and $\beta \not\models \varphi$;
- α indistinguishable to β with respect to φ if $\alpha \models \varphi$ iff $\beta \models \varphi$ holds.

Atomic Preferences and General Preferences

- Idea:
  - Users often have a set of basic desires
  - Basic desires might be in conflict (time vs. cost, comfort vs. cost, safety vs. time)

  Need to have ways to combine these preferences
- Atomic preference: $\varphi_1 \lhd \varphi_2 \lhd \ldots \lhd \varphi_n$ where each φ_i is a basic desire;
- General preference: formulae over atomic preference, constructed using propositional (&, |, !, $\lhd$)
- Semantics:
  - Defining $\alpha \models \varphi$ where $\varphi = \varphi_1 \lhd \varphi_2 \lhd \ldots \lhd \varphi_n, \varphi_1 \& \varphi_2, \varphi_1 \mid \varphi_2, or ! \varphi_1$
  - Most preferred trajectories with respect to $\lhd$:
    - those satisfying φ_1,
    - if none exists then trajectories satisfying φ_2, etc.

Implementation in Answer Set Planning

Requirements

- Encoding of preference formulae (basic desire, atomic preference, or general preference)
- Checking satisfiability of preference formula given a trajectory $\alpha = s_1 a_1 \ldots a_n s_{n+1}$ which corresponds to an answer set of $P(A, I, G)$ (notation: $\alpha[t] = s_t a_t s_{t+1} a_{t+1} \ldots a_n s_{n+1}$)

Implementation in Answer Set Planning

Idea

- Defining *satisy*(φ, T) — $\alpha[T]$ satisfies basic desire φ
- Associating weight to preference formulae such that if α is more preferred than β with respect to φ then $w(\alpha) < w(\beta)$
- Use ASP construct **minimize** to find most preferred trajectories

Realization by logic programming rules for

- defining *satisy*(φ, T)
- computing an admissible weight function

Encoding a Preference Formula φ by Π_φ

- Standard encoding of a fluent formula using ASP rules: each formula φ is encoded by a set of rules r_φ
- Assigned a unique name n_φ

Π_φ: encoding of φ

- if $\varphi = goal(\varphi_1)$ then $\Pi_\varphi = \{desire(n_\varphi), goal(n_\varphi)\} \cup r_{\varphi_1}$; or
- if $\varphi = occ(a)$ then $\Pi_\varphi = \{desire(n_\varphi), happen(n_\varphi, a)\}$; or
- if φ is a fluent formula φ_1 then $\Pi_\varphi = \{desire(n_\varphi)\} \cup r_{\varphi_1}$
- if $\varphi = \varphi_1 \wedge \varphi_2$ then $\Pi_\varphi = \{desire(n_\varphi), and(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\} \cup \Pi_{\varphi_1} \cup \Pi_{\varphi_2}$; or
- if $\varphi = \varphi_1 \vee \varphi_2$ then $\Pi_\varphi = \{desire(n_\varphi), or(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\} \cup \Pi_{\varphi_1} \cup \Pi_{\varphi_2}$; or
- if $\varphi = \neg\varphi_1$ then $\Pi_\varphi = \{desire(n_\varphi), negation(n_\varphi, n_{\varphi_1})\} \cup \Pi_{\varphi_1}$; or
- if $\varphi = \textbf{next}(\varphi_1)$ then $\Pi_\varphi = \{desire(n_\varphi), next(n_\varphi, n_{\varphi_1})\} \cup \Pi_{\varphi_1}$; or
- if $\varphi = \textbf{always}(\varphi_1)$ then $\Pi_\varphi = \{desire(n_\varphi), always(n_\varphi, n_{\varphi_1})\} \cup \Pi_{\varphi_1}$;

Rules for $satify(\varphi, T)$

$$
\begin{array}{rcl}
satisfy(F,T) & \leftarrow & desire(F), goal(F), satisfy(F, length). \\
satisfy(F,T) & \leftarrow & desire(F), happen(F,A), occ(A,T). \\
satisfy(F,T) & \leftarrow & desire(F), formula(F,G), h(G,T). \\
satisfy(F,T) & \leftarrow & desire(F), and(F,F1,F2), \\
 & & satisfy(F1,T), satisfy(F2,T). \\
satisfy(F,T) & \leftarrow & desire(F), negation(F,F1), notsatisfy(F1,T). \\
satisfy(F,T) & \leftarrow & desire(F), until(F,F1,F2), during(F1,T,T1), \\
 & & satisfy(F2,T1). \\
satisfy(F,T) & \leftarrow & desire(F), always(F,F1), during(F1,T,length+1). \\
satisfy(F,T) & \leftarrow & desire(F), next(F,F1), satisfy(F1,T+1). \\
during(F,T,T1) & \leftarrow & T < T1-1, desire(F), satisfy(F,T), \\
 & & during(F,T+1,T1). \\
during(F,T,T1) & \leftarrow & T = T1-1, desire(F), satisfy(F,T).
\end{array}
$$

Correctness of the Implementation

- Given:
  - Planning problem (A, I, G)
  - Basic desire formula φ

$$\Pi_{pref} = P(A, I, G) \cup \Pi_{\varphi} \cup \Pi_{satisfy}$$

- Property of Π_{pref}:
  - Each answer set of Π_{pref} corresponds to a most preferred trajectory with respect to φ; and
  - Each preferred trajectory with respect to φ corresponds to an answer set of Π_{pref}.

Implementation of General Preferences

Idea

- Develop an admissible weight function: $w_\varphi(\alpha)$ such that if $w_\varphi(\alpha) \geq w_\varphi(\beta)$ then α is more preferred than β with respect to φ
- Compute $w_\varphi(\alpha)$ and find answer set with maximal $w_\varphi(\alpha)$

An Admissible Weight Function

- Basic desires
  - $w_\varphi(\alpha) = 1$ if $\alpha \models \varphi$ and $w_\varphi(\alpha) = 0$ if $\alpha \not\models \varphi$
  - $\max(\varphi, \alpha) = 2$
- Atomic preferences $\varphi = \varphi_1 \lhd \varphi_2 \lhd \ldots \lhd \varphi_n$
  $w_\varphi(\alpha) = 2^{n-1} w_{\varphi_1}(\alpha) + \ldots + 2^0 w_{\varphi_n}(\alpha)$
- General preferences
  - if $\varphi = \varphi_1 \& \varphi_2$ then $w_\varphi(\alpha) = w_{\varphi_1}(\alpha) + w_{\varphi_2}(\alpha)$
  - if $\varphi = \varphi_1 \mid \varphi_2$ then $w_\varphi(\alpha) = w_{\varphi_1}(\alpha) + w_{\varphi_2}(\alpha)$
  - if $\varphi = !\varphi_1$ then $w_\varphi(\alpha) = \max(\varphi_1, \alpha) - w_{\varphi_1}(\alpha)$

Domain Knowledge

- Temporal knowledge: in order to get on the airplane, you first need to be at the airport
  (if the goal is to board the airplane, being at the airport must be true at some point) i.e., $\diamond at\_airport$ must be true
- Procedural knowledge: in order to repair a photocopy machine, one needs to follow a procedure
  (a procedure might have if-then statement, sensing actions, etc.)
  e.g., **if** $light_1$ is on **then do** xxx; **else do** yyy.
- Hierarchical knowledge: assembling a car consisting of several tasks (e.g., attaching the doors to the frame, putting the wheels on, etc.), some might have to be done before another (e.g., the electrical system needs to be completed before the wheels)

Advantage of using domain knowledge

- Efficiency
- Scalability

Planning with Domain Knowledge [Son et al. (2006)]

Representation in ASP

- Temporal knowledge: temporal formula
- Procedural knowledge: procedural formula
- Hierarchical knowledge: ordering formula

Implementation in ASP: similar to planning with preferences

- For each type of formulas, define a set of rules that determine when a formula is true.
  $\mathtt{satisfy}(\diamond f, T) \leftarrow \mathtt{holds}(f, T_1), T_1 \leq n.$
  $\mathtt{not\_satisfy}(\Box f, T) \leftarrow \textbf{not}\ \mathtt{holds}(f, T_1), T_1 \leq n.$
  $\mathtt{satisfy}(\Box f, n) \leftarrow \textbf{not}\ \mathtt{not\_holds}(f, n).$
- Add constraints that force the formula to be true:
  $\leftarrow \textbf{not}\ \mathtt{satisfy}(\diamond f, n).$
  etc.

Planning with Incomplete Information

Conformant Planning and Complexity

Definition (Conformant Planning Problem)

- Given: an $\mathcal{AL}$-action theory (D, δ), where δ is a partial state, and a set of fluent literals G.
- Determine: a sequence of actions α such that $(D, \delta) \models G$ **after** α

From [Baral et al. (2000); Liberatore (1997); Turner (2002)]:

Theorem (Complexity)

- Conformant Planning: *(D, δ) is deterministic: Σ_P^2-hard even for plans of length 1, Σ_P^2-complete for polynomial-bounded length plans.*
- Conformant Planning: *(D, δ) is non-deterministic: Σ_P^3-hard even for plans of length 1, Σ_P^3-complete for polynomial-bounded length plans.*

Epistemic planning

Need for reasoning about knowledge (or beliefs) of agents in planning

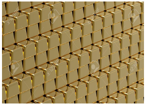
Example: Open the correct door and you get the gold; the wrong one and meet a tiger!

Epistemic planning

Need for reasoning about knowledge (or beliefs) of agents in planning

Example: Open the correct door and you get the gold; the wrong one and meet a tiger!

Real state of the world



Epistemic planning

Need for reasoning about knowledge (or beliefs) of agents in planning

Example: Open the correct door and you get the gold; the wrong one and meet a tiger!

What is a plan? Open a door (left or right)? This does not guarantee success.

Epistemic planning

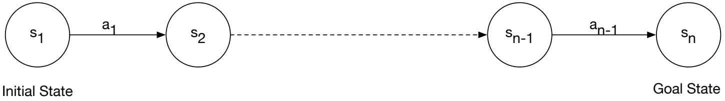
Need for reasoning about knowledge (or beliefs) of agents in planning

Example: Open the correct door and you get the gold; the wrong one and meet a tiger!

What is a plan? Open a door (left or right)? This does not guarantee success.
A reasonable plan: determine where the tiger is (e.g., smell, or make noise then listen, etc.) and open the other door.

Rough classification

- Conformant planning: initial state is incomplete, no sensing action, actions might be non-deterministic; solution is a sequence of actions (s_i is a belief state, a_i is an action).



- Conditional planning: initial state is incomplete, sensing action, actions might be non-deterministic (probabilistic); plan is often a policy or a conditional plan (with **if-then** constructs).

State of the art

- Several approaches to planning with incomplete information and sensing actions in single agent environment.
- Available systems: generation of *plan* satisfying
  $(D, \delta) \models \varphi$ **after** *plan*

Approaches to Reasoning with Incomplete Information

Incomplete Information: initial state is not fully specified (e.g. δ in (D, δ) might not be a state)

- Possible world approach (PSW): Extension of the transition function to a transition function over belief states.
- Approximation: Modifying the transition function to a transition function over approximation states.

Notation

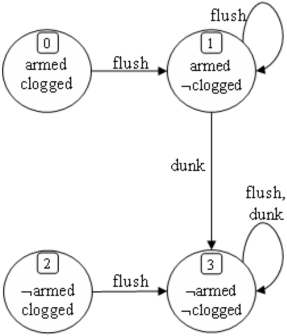
	Belief states (S and Σ)	Approximation states (δ and Δ)	
S	a set of states	a set of fluent literals	δ
Σ	a set of belief states	a set of approximation states	Δ

Example (Bomb-In-The-Toilet Revisited)

There may be a bomb in a package. Dunking the package into a toilet disarms the bomb. ...

- Fluents: *armed*, *clogged*
- Actions: *dunk*, *flush*
- Action domain:

$$\mathcal{D}_b = \begin{cases} \texttt{causes}(dunk, \neg armed, [armed]) \\ \texttt{causes}(flush, \neg clogged, []) \\ \texttt{executable}(dunk, [\neg clogged]) \end{cases}$$



- Initially, we know nothing about the value of *armed* and *clogged*.
- PWS: the initial belief state $S_0 = \{0, 1, 2, 3\}$.
- Approximation: the initial approximation state $\delta_0 = \emptyset$.

Definitions I

Approximation state/Partial state: a set of fluent literals which is a part of some state.

Belief state: a set of states

Note

Not every set of fluent literals is a partial state:

- In the airport example, $\{at(john, home)\}$ is a partial state and $\{at(john, home), at(john, airport)\}$ is not;
- In the dominoes example, $\emptyset$ is a partial state and $\{down(1), \neg down(2)\}$ is not;
- In a domain with the static causal law $\texttt{caused}(l, [\varphi])$, any set of fluent literals δ satisfying $\delta \models \varphi$ and $\delta \models \neg l$ is not a partial state.

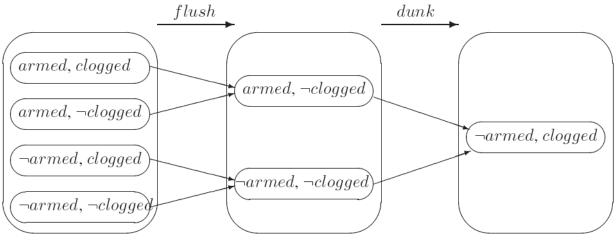
Definitions II

For an action theory (D, δ_0):

- Initial approximation state: δ_0 — a partial state
- Initial belief state:
$$S_0 = \mathit{bef}(\delta_0)$$
where
$$\mathit{bef}(\delta) = \{s \mid \delta \subseteq s,\ s \text{ is a state}\}$$
- A fluent formula φ true (false) in a belief state S if it true (false) in every state $s \in S$; it is unknown if it is neither true nor false in S.
- A fluent literal l is true (false) in an approximation state δ if $l \in \delta$ ($\neg l \in \delta$); unknown, otherwise. The truth value of a fluent formula φ is defined in the usual way.

Possible World Approach

- $S_0 = bef(\delta_0)$
- $$\Phi^c(a, S) = \begin{cases} \emptyset & \text{if } a \text{ is not executable in some } s \in S \\ \bigcup_{s \in S} \Phi(a, s) & \text{otherwise} \end{cases}$$
- Φ^c extended to $\hat{\Phi}^c$ in the usual way
- $(D, \delta_0) \models^P \varphi$ **after** α if φ is true in the final belief state
- Size of search space: n fluents $\rightarrow 2^{2^n}$ belief states



Planning Systems for Incomplete Domains

	$\mathtt{DLV}^{\mathcal{K}}$	MBP	CMBP	SGP	POND	CFF	KACMBP
Language	K	AR	AR	PDDL	PDDL	PDDL	SMV
Sequential	yes	yes	yes	no	yes	yes	yes
Concurrent	yes	no	no	yes	no	no	no
Conformant	yes	yes	yes	yes	yes	yes	yes

Table: Features of Planning Systems

Planning Systems for Incomplete Domains

- Heuristic search based planners (search in the space of belief states)
  - CFF: A belief state S is represented by the initial belief state (a CNF formula) and the action sequence leading to S. To check whether a fluent literal l is true is S, a call to a SAT-solver is made. (subset of) PDDL as input.
  - POND: Graph plan based conformant planner. (subset of) PDDL as input.
- Translation into model checking: KACMBP (CMBP) – Input is a finite state automaton. Employing BDD (Binary Decision Diagram) techniques to represent and search the automaton. Consider nondeterministic domains with uncertainty in both the initial state and action effects.
- Translation into logic programming: $\texttt{DLV}^{\mathcal{K}}$ is a declarative, logic-based planning system built on top of the DLV system (an answer set solver).

General Considerations and Properties

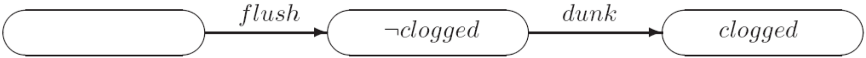
- Address the complexity problem of the possible world approach: give up completeness for efficiency in reasoning/planning
- Sound with respect to possible world semantics (formal proof is provided in some work)
- Representation languages and approaches are different
  - Situation calculus: [Etzioni et al. (1996); Goldman and Boddy (1994); Petrick and Bacchus (2004)]
  - Action languages: [Son and Baral (2001); Son and Tu (2006); Son et al. (2005b)]
  - Logic programming: [Son et al. (2005a); Tu et al. (2006, 2011)]

0-Approximation Approach [Son and Baral (2001)]

- Initial partial state: δ_0
- Transition function is defined as

$$\Phi^0(a, \delta) = (\delta \cup de(a, \delta)) \setminus \neg pe(a, \delta)$$

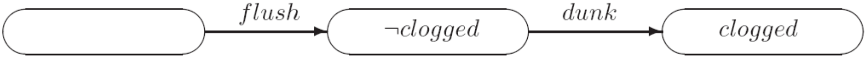
  where
  - $de(a, \delta)$ is the set of "direct effects" of a in δ
  - $pe(a, \delta)$ is the set of "possible effects" of a in δ
- $(D, \delta_0) \models^0 \varphi$ **after** α if φ is true in the final partial state
- n fluents $\rightarrow 3^n$ partial states
- Incomplete
- No static causal laws



0-Approximation Approach – Example

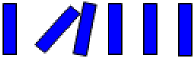
$$D_b = \begin{cases} \texttt{causes}(\mathit{dunk}, \neg \mathit{armed}, [\mathit{armed}]) \\ \texttt{causes}(\mathit{flush}, \neg \mathit{clogged}, []) \\ \texttt{executable}(\mathit{dunk}, [\neg \mathit{clogged}]) \end{cases}$$

- $\delta_0 = \emptyset$
  - *dunk* is not executable in δ_0
  - *flush* is executable in δ_0, $de(\mathit{flush}, \delta_0) = pe(\mathit{flush}, \delta_0) = \{\neg \mathit{clogged}\}$
  - $\Phi^0(\mathit{flush}, \delta_0) = \{\neg \mathit{clogged}\}$
- $\delta_1 = \{\neg \mathit{clogged}\}$
  - *dunk*, *flush* are executable in δ_1
  - $de(\mathit{dunk}, \delta_1) = \emptyset$ and $pe(\mathit{dunk}, \delta_1) = \{\neg \mathit{armed}\}$
  - $\Phi^0(\mathit{dunk}, \delta_1) = \{\mathit{clogged}\}$



Dealing with Static Causal Laws

How will the 0-approximation fare in the dominoes example?



$$D_d = \begin{cases} \texttt{caused}(down(n+1), [down(n)]) \\ \texttt{causes}(touch(i), down(i), []) \end{cases}$$

$\delta_0 = \emptyset$

- $touch(i)$ is executable for every i
- $de(touch(i), \delta_0) = \{down(i)\}$ and $pe(touch(i), \delta_i) = \{down(i)\}$
- $\Phi^0(touch(i), \delta_0) = \{down(i)\}$

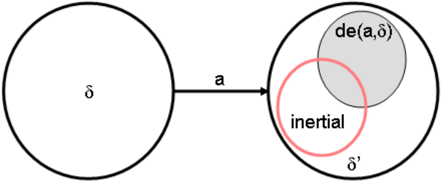
Intuitive result

$\{down(j) \mid i \leq j \leq n\} \subseteq \Phi^0(touch(i), \delta_0)$

Not good!

Dealing with Static Causal Laws

$$\delta' = Cn_D(de(a, \delta) \cup (\delta \cap \delta'))$$



The next state has three parts: (i) the direct effect $de(a, \delta)$; (ii) the inertial; (iii) the indirect effects (the closure of Cn_D).

Dealing with Static Causal Laws

Question

What will be the inertial part?

Ideas

A literal does not change its value if it belongs to δ and

- either its negation cannot possibly hold in δ';
  $\Rightarrow$ possible holds approximation
- or it cannot possibly change in δ'
  $\Rightarrow$ possible change approximation

Φ^{ph} Approximation – Idea

A literal l possibly holds in the next state if

- it possibly holds in the current state (i.e., $l \notin \neg\delta$)
- it does not belong to the negation of the direct effect of the action (i.e., $l \notin \neg Cl_D(de(a, \delta))$
- there is some static causal law whose body possibly holds (i.e., there exists some static causal law l **if** φ such that φ possibly holds)

Φ^{ph} Approximation – Definition

$$E(a,\delta) = Cl_D(e(a,\delta)) \quad [\text{always belongs to } \delta']$$

$$ph(a,\delta) = \bigcup_{i=0}^{\infty} ph^i(a,\delta) \quad [\text{possiblly holds in } \delta']$$

$$ph^0(a,\delta) = (pe(a,\delta) \cup \{l \mid \neg l \notin \delta\}) \setminus \neg E(a,\delta)$$

OBS: if l **if** φ in D and φ possibly holds then l possibly holds.

$$ph^{i+1}(a,\delta) = ph^i(a,\delta) \cup \left\{ l \;\middle|\; \begin{array}{l} \exists[\ l \textbf{ if } \psi\ ] \text{ in } D \text{ s.t. } l \notin \neg E(a,\delta), \\ \psi \subseteq ph^i(a,\delta), \neg\psi \cap E(a,\delta) = \emptyset \end{array} \right\}$$

Definition

- if a is not executable in δ then
$$\Phi^{ph}(a,\delta) = \emptyset$$
- otherwise,
$$\Phi^{ph}(a,\delta) = Cl_D(\{l \mid l \notin \neg ph(a,\delta)\})$$

Φ^{ph} Approximation – Example

$$D_d = \begin{cases} down(i+1) \textbf{ if } down(i) \\ touch(i) \texttt{ causes} down(i) \end{cases}$$

Computation for $\delta_0 = \emptyset$

- $de(touch(i), \delta_0) = \{down(i)\}$ and $pe(touch(i), \delta_0) = \{down(i)\}$
- $E(touch(i), \delta_0) = \{down(j) \mid i \leq j \leq n\}$
- $ph^0(touch(i), \delta_0) = \{down(j) \mid 1 \leq j \leq n\} \cup \{\neg down(j) \mid 1 \leq j < i\}$
- $ph^k(touch(i), \delta_0) = \{down(j) \mid 1 \leq j \leq n\} \cup \{\neg down(j) \mid 1 \leq j < i\}$
- $\Phi^{ph}(touch(i), \delta_0) = \{down(j) \mid i \leq j \leq n\}$

Φ^{pc} Approximation – Idea

A literal l possibly changes if

- it is not in δ
- it is a possible effect a (i.e., there exists a dynamic law $\texttt{causes}(a, l, [\varphi])$ and φ is not false in δ)
- it is a possibly indirect effect of a (i.e., there exists a static causal law $\texttt{caused}(l, [\varphi])$ and φ possibly changes)

Φ^{pc} Approximation

$$pc(a,\delta) = \bigcup_{i=0}^{\infty} pc^i(a,\delta)$$

$$pc^0(a,\delta) = pe(a,\delta) \setminus \delta$$

$$pc^{i+1}(a,\delta) = pc^i(a,\delta) \cup \left\{ l \;\middle|\; \begin{array}{l} \exists [\, l \textbf{ if } \psi \,] \in D \text{ s.t. } , l \notin \delta \\ \psi \cap pc^i(a,\delta) \neq \emptyset, \text{ and } \neg\psi \cap E(a,\delta) = \emptyset \end{array} \right\}$$

Definition

- if a is not executable in δ then
$$\Phi^{pc}(a,\delta) = \emptyset$$
- otherwise,
$$\Phi^{pc}(a,\delta) = Cl_D(E(a,\delta) \cup (\delta \setminus \neg pc(a,\delta)))$$

Φ^{pc} Approximation – Example

$$D_d = \begin{cases} down(i+1) \textbf{ if } down(i) \\ touch(i) \text{ causes} down(i) \end{cases}$$

Computation for $\delta_0 = \emptyset$

- $de(touch(i), \delta_0) = \{down(i)\}$ and $pe(touch(i), \delta_0) = \{down(i)\}$
- $E(touch(i), \delta_0) = \{down(j) \mid i \leq j \leq n\}$
- $pc^0(touch(i), \delta_0) = \{down(i)\}$
- $pc^1(touch(i), \delta_0) = \{down(i), down(i+1)\}$
- $pc(touch(i), \delta_0) = \{down(j) \mid i \leq j \leq n\}$
- $\Phi^{pc}(touch(i), \delta_0) = \{down(j) \mid i \leq j \leq n\}$

Properties of Φ^{ph} and Φ^{pc} Approximations

- Behave exactly as 0-approximation in action theories without static causal laws
- Sound but incomplete (proofs in [Tu (2007)])
- Support parallel execution of actions (formal proofs available)
- Incompatibility between Φ^{ph} and Φ^{pc} $\Rightarrow$ could union the two to create a better approximation
- Deterministic: $\Phi^{A}(a, \delta)$ can be computed in polynomial-time
- Polynomial-length planning problem w.r.t Φ^{A} is NP-complete
- Could improve the approximations

ASP Implementation

Replacing rules for computing effects of actions with the following:

- Dynamic causal law: $\texttt{causes}(A, L, \varphi)$
  $\texttt{de}(L, T+1) \leftarrow \texttt{occurs}(A, T), \texttt{holds}(\varphi, T)$
  $\texttt{ph}(L, T) \leftarrow \texttt{occurs}(A, T-1), \textbf{not } \texttt{holds}(\neg\varphi, T), \textbf{not } \texttt{de}(\neg L, T+1)$
- Static causal law: $\texttt{caused}(L, \varphi)$
  $\texttt{ph}(L, T) \leftarrow \texttt{ph}(\varphi, T)$
- Additional rule:
  $\texttt{ph}(L, T) \leftarrow \textbf{not } \texttt{holds}(\neg L, T-1), \textbf{not } \texttt{de}(\neg L, T)$
- Inertial rule:
  $\texttt{holds}(L, T) \leftarrow \textbf{not } \texttt{ph}(\neg L, T), \texttt{holds}(L, T-1)$

What is good about the approximation?

Theorem (Complexity)

Conformant Planning: *(D, δ) is deterministic: NP-complete for polynomial-bounded length plans.*

Consequence

If (D, δ) is complete, planners can use the 0-approximation (lower complexity) instead of the possible world semantics. In fact, classical planners can be used to solve conformant planning (change in the computation of the next state.)

Approximation Based Conformant Planners

- Earlier systems [Etzioni et al. (1996); Goldman and Boddy (1994)]: approximation is used in dealing with sensing actions (context-dependent actions and non-deterministic outcomes)
- PKS [Petrick and Bacchus (2004)] is very efficient (plus: use of domain knowledge in finding plans)
- CpA and CPASP [Son et al. (2005b,a); Tu et al. (2007, 2006, 2011)] are competitive with others such as CFF, POND, and KACMBP in several benchmarks
- Incompleteness

Application in Conformant Planning

- CPASP:
  - Logic programming based
  - Uses Φ^{ph} approximation
  - Can generate both concurrent plans and sequential plans
  - Can handle disjunctive information about the initial state
  - Competitive with concurrent conformant planners and with others in problems with short solutions
- CPA:
  - Forward, best-first search with simple heuristic function (number of fulfilled subgoals)
  - Provides users with an option to select the approximation
  - Generates sequential plans only
  - Can handle disjunctive information about the initial state
  - Competitive with other state-of-the-art conformant planners

$\mathcal{B}$ vs. PDDL — Revisited

1. PDDL domains can be translated into $\mathcal{B}$ domains — 1-to-1
2. $\mathcal{AL}$ domains can be translated into PDDL — might need to introduce additional actions (only polynomial number of actions)

Consequence

Planners using PDDL as their representation language can make use of the approximations in dealing with unrestricted defined fluents.

Why sensing actions?

- Some properties of the domain can be observed after some sensing actions are executed
  - Cannot decide whether a package contains a bomb until we use a special device to detect it
  - A robot cannot determine an obstacle until it uses a sensor to detect it

Why sensing actions?

- Some properties of the domain can be observed after some sensing actions are executed
  - Cannot decide whether a package contains a bomb until we use a special device to detect it
  - A robot cannot determine an obstacle until it uses a sensor to detect it
- Two important questions:
  - What is a plan?

Why sensing actions?

- Some properties of the domain can be observed after some sensing actions are executed
  - Cannot decide whether a package contains a bomb until we use a special device to detect it
  - A robot cannot determine an obstacle until it uses a sensor to detect it
- Two important questions:
  - What is a plan?
  - How to reason about sensing actions?

Extending $\mathcal{B}$ to handle sensing actions

- Allow knowledge-producing laws of the form

$$\texttt{determines}(a, \theta)$$

"if sensing action a is executed, then the values of $l \in \theta$ will be known"

- New language is called $\mathcal{B}_k$

Why sensing actions? — Example

- One bomb, two packages; exactly one package contains the bomb
- Initially, the toilet is not clogged. No *flush* action.
- Bomb can be detected by only by *X-ray*.

$$D_2 = \left\{ \begin{array}{l} \textbf{oneof } \{armed(1), armed(2)\} \\ \texttt{causes}(dunk(P), \neg armed(P), []) \\ \texttt{causes}(dunk(P), clogged, []) \\ \texttt{executable}(dunk(P), [\neg clogged) \\ \texttt{determines}(x{-}ray, \{armed(1), armed(2)\}) \end{array} \right\}$$

- No conformant plan for

$$P_1 = (D_2, \{\neg clogged\}, \{\neg armed(1), \neg armed(2)\})$$

What is a plan in the presence of sensing actions?

- Conditional Plans: take into account contingencies that may arise
  - If a is a non-sensing action and $\langle\beta\rangle$ is a conditional plan then $\langle a, \beta\rangle$ is a conditional plan
  - If a is a sensing action that senses literals $l_1, \ldots, l_n$, and $\langle\beta_i\rangle$ is a conditional plan then

$$\left\langle a, \textbf{cases}\begin{pmatrix} l_1 \rightarrow \beta_1 \\ \ldots \\ l_n \rightarrow \beta_n \end{pmatrix}\right\rangle$$

is a conditional plan

Example of Conditional Plan

$$\left\langle x{-}ray, \textbf{cases}\begin{pmatrix} armed(1) \rightarrow dunk(1) \\ armed(2) \rightarrow dunk(2) \end{pmatrix} \right\rangle$$

is a solution of

$$P_1 = (D_2, \{\neg clogged\}, \{\neg armed(1), \neg armed(2)\})$$

How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: $\mathit{Actions} \times \mathit{Partial\ States} \rightarrow 2^{\mathit{Partial\ States}}$

How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: $Actions \times Partial\ States \rightarrow\ 2^{Partial\ States}$
- For each $A \in \{ph, pc\}$, we define a transition function Φ_S^A as follows
  - for a non-sensing action a, Φ_S^A is the same as Φ^A
  - for a sensing action a, each partial state in Φ_S^A corresponds to a literal that is sensed by a
- Result in four different approximations of $\mathcal{B}_k$ domain descriptions
- Entailment $\models_S^A$

$$(D, \delta_0) \models_S^A \varphi \textbf{ after } \alpha$$

if φ is true in every final partial state of the execution of α

How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: $Actions \times Partial\ States \rightarrow 2^{Partial\ States}$
- For each $A \in \{ph, pc\}$, we define a transition function Φ_S^A as follows
  - for a non-sensing action a, Φ_S^A is the same as Φ^A
  - for a sensing action a, each partial state in Φ_S^A corresponds to a literal that is sensed by a
- Result in four different approximations of $\mathcal{B}_k$ domain descriptions
- Entailment $\models_S^A$

$$(D, \delta_0) \models_S^A \varphi \textbf{ after } \alpha$$

  if φ is true in every final partial state of the execution of α
- Properties
  - Φ_S^A can be computed in polynomial time

How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: $Actions \times Partial\ States \rightarrow\ 2^{Partial\ States}$
- For each $A \in \{ph, pc\}$, we define a transition function Φ^A_S as follows
  - for a non-sensing action a, Φ^A_S is the same as Φ^A
  - for a sensing action a, each partial state in Φ^A_S corresponds to a literal that is sensed by a
- Result in four different approximations of $\mathcal{B}_k$ domain descriptions
- Entailment $\models^A_S$

$$(D, \delta_0) \models^A_S \varphi\ \textbf{after}\ \ \alpha$$

  if φ is true in every final partial state of the execution of α
- Properties
  - Φ^A_S can be computed in polynomial time
  - the polynomial-length conditional planning: NP-complete

$\mathcal{B}_k$ Approximations

Definition

- If a is not executable in δ then
$$\Phi_S^A(a,\delta) = \emptyset$$
- If a is a non-sensing action then
$$\Phi_S^A(a,\delta) = \begin{cases} \emptyset & \text{if } \Phi^A(a,\delta) \text{ is consistent} \\ \{\Phi^A(a,\delta)\} & \text{otherwise} \end{cases}$$
- If a is a sensing action associated with
$$a \ \texttt{determines} \ \theta$$
then
$$\Phi_S^A(a,\delta) = \{Cl_D(\delta \cup \{g\}) \mid g \in \theta \text{ and } Cl_D(\delta \cup \{g\}) \text{ is consistent}\}$$

Application in Conditional Planning

- Conditional Planning Problem: $P = (D, \delta_0, \mathcal{G})$
  A solution of P is a conditional plan α such that
  $$(D, \delta_0) \models^P \mathcal{G} \textbf{ after } \alpha$$
- ASCP:
  - Implemented in logic programming (Rules similar to approximation)
  - Approximation: Φ_S^{pc}
  - Can generate both concurrent plans and sequential plans
  - Soundness and completeness of ASCP are proved
  - Competitive with some other conditional planners

Analysis of Experimental Results — Possible Improvements

1. Dealing directly with static causal laws (defined fluents) is helpful.
2. CPA (CPA+) is good in domains with high degree of uncertainty and the search does not require the exploration of a large number of states.
3. CPA (CPA+) is not so good in domains with high degree of uncertainty and the search requires the exploration of a large number of states.
4. Other heuristics can be used in CPA as well (preliminary results on a new version of a CPA+ plus sum/max heuristics are very good)
5. Performance can be improved by running on parallel machine as well (preliminary results on a parallel version of CPA+ and a parallel version of FF show that parallel planning can solve larger instances [Tu et al. (2009)]).

Towards More Complex Domains

Transition functions have been defined for domains with

1. actions with durations, delayed effects
2. resources
3. processes
4. time and deadlines

Problems for planning systems in complex domains:

1. Representation: possibility of infinitely many fluents (e.g. resources and time) $\Rightarrow$ compact representation of state?
2. Search:
    1. possibility of infinitely many successor states
    2. concurrent actions

   $\Rightarrow$ new type of heuristic?

Outline

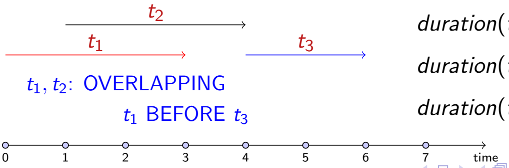
Scheduling

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).



$duration(t_1, 3)$

$duration(t_2, 3)$

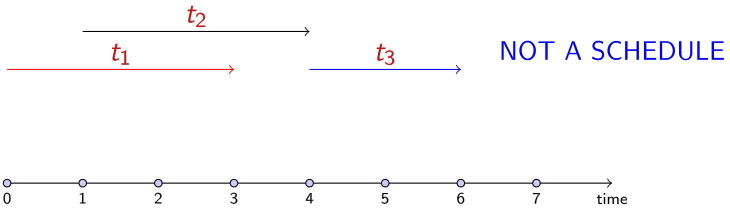
$duration(t_3, 2)$

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 $\{t_1, t_2, t_3\}$;

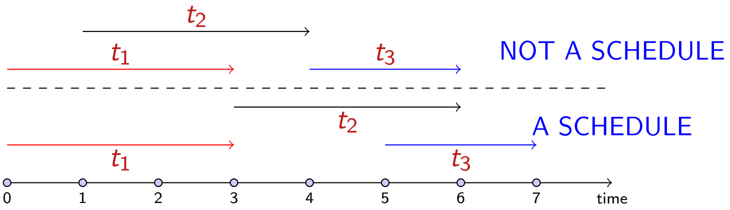


A Schedule for a Set of Tasks: Definition

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.

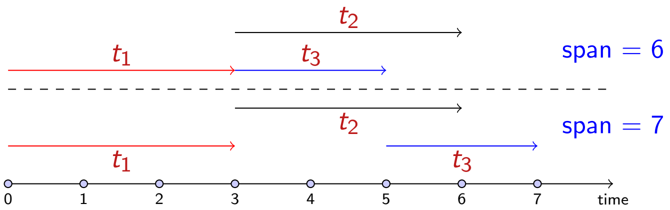


Span of a Schedule: Definition

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

Code

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

ASP Encoding for Scheduling

Input: $task(t_1), \ldots,, duration(t_1, d_1), \ldots, prec(t_i, t_j), \ldots, non\_overlap(t_i, t_j), \ldots,$

Code

```
% 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: $task(t_1), \ldots,, duration(t_1, d_1), \ldots, prec(t_i, t_j), \ldots, non\_overlap(t_i, t_j), \ldots,$

Code

```
% 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+D2, S1≥ S2.
```

ASP Encoding for Scheduling

Input: $task(t_1), \ldots,, duration(t_1, d_1), \ldots, prec(t_i, t_j), \ldots, non\_overlap(t_i, t_j), \ldots,$

Code

```
% 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+D2, 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)}.
```

Scalability: An Experiment

- 5 tasks
- Maximum task duration μ: increasing from 100 to 200
- Task duration randomly generated between $\frac{\mu}{2}$ and μ
- $prec(t_i, t_j)$: set for 30% of $\langle t_i, t_j \rangle$ pairs, randomly selected
- Simplifications: (1) no overlap constraints; (2) no span minimization

Execution times:

Maximum task duration										
100	**110**	**120**	**130**	**140**	**150**	**160**	**170**	**180**	**190**	**200**
11.176	9.640	10.300	14.350	12.424	16.114	14.732	22.156	21.940	27.172	24.156

Times in seconds, averaged over 5 *randomly generated trials for every configuration*

- Execution time increases with task duration
- 11+ sec for 5 tasks and no overlap constraints: too much?

Can we do better?

Scheduling with CASP

CSP features constructs for increased efficiency of scheduling...

Idea

- Use CASP rather than ASP
- Encode the qualitative parts of the problem in ASP
- Encode the numerical parts using CSP, embedded in ASP

CASP 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

CASP 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

```
% generating start time
var(st(T),0,length) :- task(T).
required(cumulative([st/1],[duration/2])).
% List notation
%   [var/k] expands to [var(term_1,... ,term_k), ...]
%      e.g., [st/1] is expanded to [st(t_1), st(t_2), ..., st(t_n) ]
%   [pred/k] expands to list of last arguments of all pred(term_1,... ,term_k)
%      e.g., [duration/2] is expanded to [d_1, d_2, ..., d_n ]
% result: required(cumulative([st(t_1), st(t_2), ..., st(t_n) ],[d_1, d_2, ..., d_n ])).
```

CASP 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

```
% generating start time
var(st(T),0,length) :- task(T).
required(cumulative([st/1],[duration/2])).
% checking prec
required(st(T2) ≥ st(T1)+D1) :- prec(T1,T2), duration(T1,D1).
```

CASP 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

```
% generating start time
var(st(T),0,length) :- task(T).
required(cumulative([st/1],[duration/2])).
% checking prec
required(st(T2) ≥ st(T1)+D1) :- prec(T1,T2), duration(T1,D1).
% non-overlap
required(st(T2) ≥ st(T1)+D1 ∨ st(T1) ≥ st(T2)+D2) :-
        non_overlap(T1,T2), duration(T1, D1), duration(T2, D2).
```

Scalability: ASP vs CASP

- 5 tasks
- Maximum task duration μ: increasing from 100 to 200
- Task duration randomly generated between $\frac{\mu}{2}$ and μ
- $prec(t_i, t_j)$: set for 30% of $\langle t_i, t_j \rangle$ pairs, randomly selected
- Simplifications: (1) no overlap constraints; (2) no span minimization

Execution times:

	Maximum task duration										
Tasks	**100**	**110**	**120**	**130**	**140**	**150**	**160**	**170**	**180**	**190**	**200**
ASP	11.176	9.640	10.300	14.350	12.424	16.114	14.732	22.156	21.940	27.172	24.156
CASP	0.039	0.041	0.040	0.042	0.041	0.040	0.048	0.050	0.042	0.060	0.045

Times in seconds, averaged over 5 *randomly generated trials for every configuration*

With CASP:

- Execution time virtually independent of maximum task duration
- Neglibile time for 5 tasks

But why CASP rather than CSP?

Scheduling in CASP in Practice

Print Shop Scheduling Problem

Constraints:

- Multiple job phases (print, cut, bind, ...)
- Multiple devices available for each phase
- Different device capabilities and configurations
- Various types of consumables (paper, ink, ...)
- Constraints on which consumables can be used on which devices (size, quality, . . .)
- Ability to incrementally:
  - Add new jobs
  - Handle device failures
- Include heuristics from shop operators

Print Shop Scheduling Encoding

```
% Start time of job J on device type D
var(st(D,J),0,MT) :- job(J), job_device(J,D), max_time(MT).
% assign start times; only up to N overlapping jobs for a device with N instances
required(cumulative([st(D)/2],[len_by_dev(D)/3],N)) :- n_instances(D,N)
% length of a job is the same on any suitable device
len_by_dev(D,J,L) :- job(J), job_device(J,D), job_len(J,L).
```

Print Shop Scheduling Encoding

```
% Start time of job J on device type D
var(st(D,J),0,MT) :- job(J), job_device(J,D), max_time(MT).
% assign start times; only up to N overlapping jobs for a device with N
instances
required(cumulative([st(D)/2],[len_by_dev(D)/3],N)) :- n_instances(D,N)
% length of a job is the same on any suitable device
len_by_dev(D,J,L) :- job(J), job_device(J,D), job_len(J,L).
% checking prec
required(st(D2,J2) ≥ st(D1,J1)+Len1) :-
        job_device(J1,D1), job_device(J2,D2),
        prec(J1,J2), job_len(J1,Len1).
```

Introducing Incremental Updates

Problem:

- A schedule has already been computed
- We need to add new jobs
- Jobs that are currently running must not be affected
- Jobs that have already run should be disregarded

Input:
curr_start(J,T): current schedule
curr_device(J,D): current job-device assignment
curr_time(CT): current (wall-clock) time

```
% identify jobs that have already started
already_started(J) :- curr_start(J,T), curr_time(CT), CT > T.
```

Introducing Incremental Updates

Problem:

- A schedule has already been computed
- We need to add new jobs
- Jobs that are currently running must not be affected
- Jobs that have already run should be disregarded

Input:
curr_start(J,T): current schedule
curr_device(J,D): current job-device assignment
curr_time(CT): current (wall-clock) time

```
% identify jobs that have already started
already_started(J) :- curr_start(J,T), curr_time(CT), CT > T.
% determine which jobs must not be affected
must_not_schedule(J) :-
        already_started(J), not ab(must_not_schedule(J)).
```

Introducing Incremental Updates

Input:
curr_start(J,T): current schedule
curr_device(J,D): current job-device assignment
curr_time(CT): current (wall-clock) time

```
% identify jobs that have already started
already_started(J) :- curr_start(J,T), curr_time(CT), CT > T.
% determine which jobs must not be affected
must_not_schedule(J) :-
        already_started(J), not ab(must_not_schedule(J)).
% the start time of those jobs remains the same
required(st(D,J)=T) :-
        curr_device(J,D), curr_start(J,T), must_not_schedule(J).
```

Introducing Incremental Updates

Input:
curr_start(J,T): current schedule
curr_device(J,D): current job-device assignment
curr_time(CT): current (wall-clock) time

```
% identify jobs that have already started
already_started(J) :- curr_start(J,T), curr_time(CT), CT > T.
% determine which jobs must not be affected
must_not_schedule(J) :-
        already_started(J), not ab(must_not_schedule(J)).
% the start time of those jobs remains the same
required(st(D,J)=T) :-
        curr_device(J,D), curr_start(J,T), must_not_schedule(J).
% the start time of all future jobs must occur in the future
required(st(D,J)≥T) :-
        curr_device(J,D), curr_start(J,T),
```

What About Production Failures?

If a production failure occurs at runtime, each failed job must be rescheduled, even though it is currently running.

Input:
production_failed(J): production of J has failed

```
% determine which jobs must not be affected
must_not_schedule(J) :-
        already_started(J), not ab(must_not_schedule(J)).
```

What About Production Failures?

If a production failure occurs at runtime, each failed job must be rescheduled, even though it is currently running.

Input:
production_failed(J): production of J has failed

```
% determine which jobs must not be affected
must_not_schedule(J) :-
        already_started(J), not ab(must_not_schedule(J)).
% a failed job is abnormal w.r.t. non-rescheduling
ab(must_not_schedule(J)) :-
        already_started(J), production_failed(J).
% the start time of all future jobs must occur in the future
required(st(D,J)≥T) :-
        curr_device(J,D), curr_start(J,T),
        not must_not_schedule(J).
```

What About Production Failures?

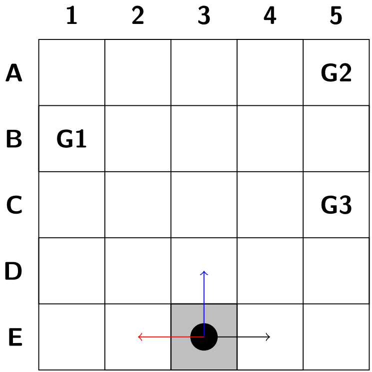
```
% determine which jobs must not be affected
must_not_schedule(J) :-
        already_started(J), not ab(must_not_schedule(J)).
% a failed job is abnormal w.r.t. non-rescheduling
ab(must_not_schedule(J)) :-
        already_started(J), production_failed(J).
% the start time of all future jobs must occur in the future
required(st(D,J)≥T) :-
        curr_device(J,D), curr_start(J,T),
        not must_not_schedule(J).
```

Summarizing:

- The parts in green leverage non-monotonicity of ASP to make decisions about production failures
- The parts in blue define the CSP accordingly

Outline

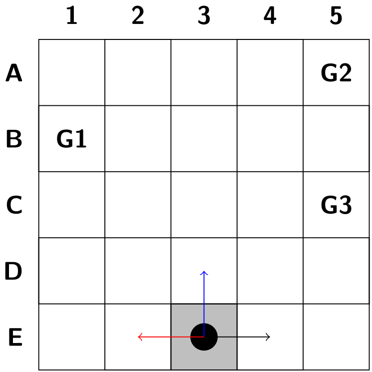
Goal Recognition



-

- Special form of plan recognition
- Several applications (security, computer games, NLP, etc.)

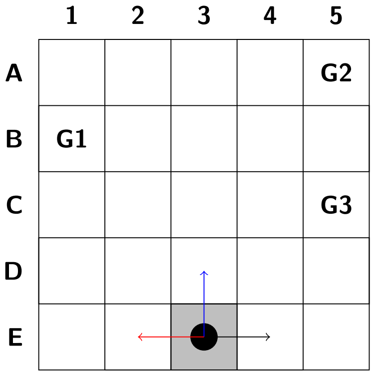
Goal Recognition



- Question: What is the goal of the agent?
- Assumption: agents act optimally.
-

- Special form of plan recognition
- Several applications (security, computer games, NLP, etc.)

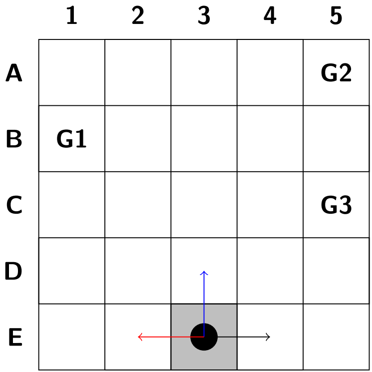
Goal Recognition



- Question: What is the goal of the agent?
- Assumption: agents act optimally.
- Left: G1
-

- Special form of plan recognition
- Several applications (security, computer games, NLP, etc.)

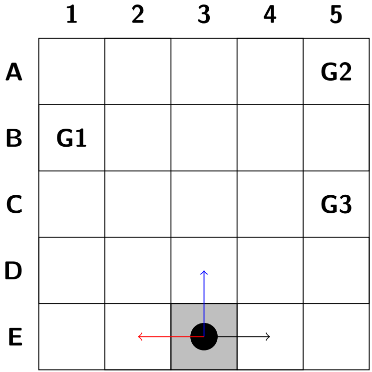
Goal Recognition



- Question: What is the goal of the agent?
- Assumption: agents act optimally.
- Left: G1
- Up: G1 or G2 or G3
-

- Special form of plan recognition
- Several applications (security, computer games, NLP, etc.)

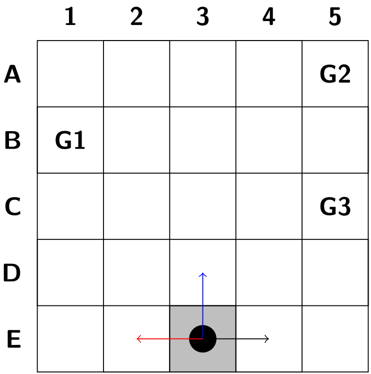
Goal Recognition



- Question: What is the goal of the agent?
- Assumption: agents act optimally.
- Left: G1
- Up: G1 or G2 or G3
- Right: G2 or G3
-

- Special form of plan recognition
- Several applications (security, computer games, NLP, etc.)

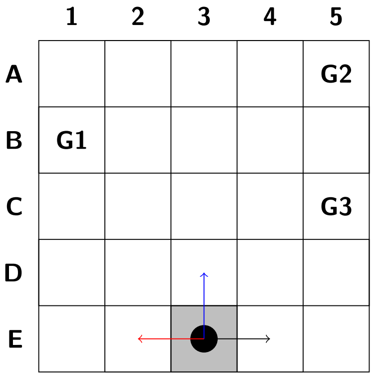
Goal Recognition Design*



- Question: What is the goal of the agent?
- Assumption: agents act optimally.
- Idea: Modify the planning problem so we can recognize the goal as early as possible!

* *Goal recognition design* by Keren, S., Gal, A., and Karpas, E., ICAPS 2014.

Goal Recognition Design*



- Question: What is the goal of the agent?
- Assumption: agents act optimally.
- Idea: Modify the planning problem so we can recognize the goal as early as possible!
  How? Blocking actions.

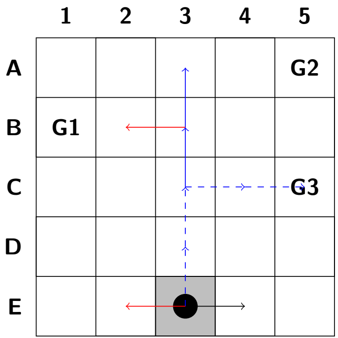
* *Goal recognition design* by Keren, S., Gal, A., and Karpas, E., ICAPS 2014.

Goal Recognition Design*

- Question: What is the goal of the agent?
- Assumption: agents act optimally.
- Idea: Modify the planning problem so we can recognize the goal as early as possible!
  How? Blocking actions.
  Which modification is the best?
  Reducing Worst-Case Distinctiveness (WCD)

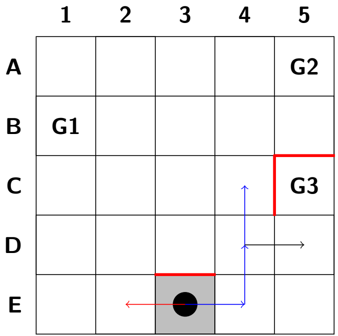
* *Goal recognition design* by Keren, S., Gal, A., and Karpas, E., ICAPS 2014.

Worst-Case Distinctiveness



- WCD = The maximal number of actions that the agent can execute before revealing the goal.
- WCD = 1? no
- WCD = 2? no
- WCD = 3? no
- WCD = 4? yes (dashed path)

Worst-Case Distinctiveness



- WCD = The maximal number of actions that the agent can execute before revealing the goal.
- Original: WCD=4
- Blocking up(E3-D3), up(C5-B5), right(C4-C5) reduces WCD to 2.

Problems

- Computing WCD?
- Reducing WCD: given an integer k, identify a set of at most k actions so that the WCD of the problem without these k actions is smallest?

Approach

Goals

- Computing WCD?
- Reducing WCD: given an integer k, identify a set of at most k actions so that the WCD of the problem without these k actions is smallest?

What's new?

- Previous approach: imperative language
  - use state-of-the-art off-the-shelf planning systems (Fast Downward) for computing optimal plans
  - develop and implement algorithms for computing WCD and reducing WCD
- Our approach: declarative language
  - represent both problems as logic programs
  - use state-of-the-art off-the-shelf answer set solver (CLASP)
  - two different encodings - perform significantly better

General Idea

- Computing WCD: two different approaches
  - solving QBF problems using ASP (saturation based meta encoding).
  - extending traditional answer set planning to deal with multiple goals (multi-shot encoding)
- Reducing WCD:
  - Guess a set of actions (that will be removed) by using choice atoms: $\{blocked(A) : action(A)\}k$.
  - Calculate WCD for **new** problem.
  - Identify the **best** answer (with smallest WCD).

Computing WCD: Saturation Based Meta Encoding

- Saturation based method in ASP: for solving QBF-problems
- Computing WCD can be represented as a QBF problem

WCD

maximal number of actions before revealing the goal same as longest common prefix between optimal plans of two different goals

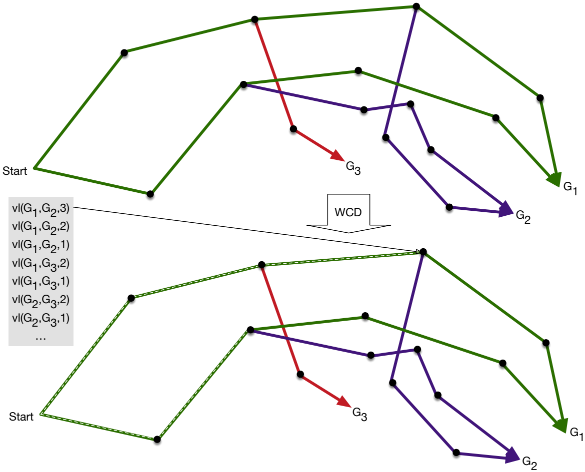
WCD as QBF

$vl(x, y, c)$: c is common prefix of two minimal cost plans π_x^* and π_y^* (π_g^* is an optimal plan for g).

$$\exists x, y, c[vl(x, y, c) \wedge [\forall x', y', c'[vl(x', y', c') \rightarrow |c| \geq |c'|]]$$

$x, y, x', y' \in \mathbf{G}$

Saturation Based Meta Encoding: Illustration



Two different implementations: one needs one call to the solver; the other uses two calls (computing the set of potential helpful actions in the first pass).

Multi-Shot Encoding

- Multi-shot ASP (**clingo**): new feature, allows for dealing with continuous changes, Python+ASP.
- Computing WCD:
  - Compute optimal cost of plan for each goal.
  - Compute answer sets containing minimal plans for all goals (one plan per goal).
  - Calculate WCD (given an answer set).
  - Use optimization feature to identify WCD (answer set with the maximal WCD).
- Reducing WCD:
  - Guess a set of actions (that will be removed).
  - Calculate WCD for **new** problem.
  - Identify the **best** answer (with smallest WCD).

Experimental Results

- Use benchmarks from original package: four domains (Grid Navigation, IPC+Grid, BlockWords, Logistics).
- Timeout: 5 hours.
- Parameters: $k = 1$ or $k = 2$ (suggested in original package).
- Outcome: both methods perform very well.

Experimental Results I ($k = 1$)

Domain	Instances	WCD reduction	Runtime (s) PR	SAT-1	SAT-2	MS
GRID-NAVIGATION	5-14	$9 \rightarrow 9$	12	26	1	1
	19-10	$17 \rightarrow 17$	12	18	1	1
	20-9	$39 \rightarrow 39$	23	406	3	3
	16-11	$4 \rightarrow 4$	11	12	1	1
	16-11	$4 \rightarrow 4$	12	10	1	1
IPC-GRID$^+$	5-5-5	$4 \rightarrow 3$	14	9	1	1
	5-10-10	$11 \rightarrow 11$	194	475	14	11
	10-5-5	$12 \rightarrow 10$	46	36	2	1
	10-10-10	$19 \rightarrow 19$	2,661	1,257	33	30
BLOCK-WORDS	8-20	$10 \rightarrow 10$	946	timeout	64	48
	8-20	$14 \rightarrow 14$	809	timeout	121	71
LOGISTICS	1-2-6-2-2-6	$18 \rightarrow 18$	3,506	timeout	151	228
	1-2-6-2-2-6	$18 \rightarrow 18$	2,499	timeout	135	140
	2-2-6-2-2-6	$17 \rightarrow 17$	3,173	timeout	352	756
	2-2-6-2-4-6	$17 \rightarrow 17$	timeout	timeout	5,377	1,943
	2-2-6-2-6-6	$16 \rightarrow 16$	timeout	timeout	5,166	timeout

Experimental Results II ($k = 2$)

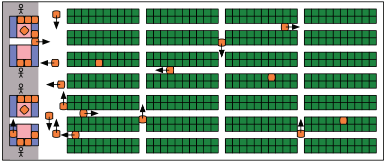
(b) $k = 2$

Domain	Instances	WCD reduction	Runtime (s) PR	SAT-1	SAT-2	MS
GRID-NAVIGATION	5-14	$9 \rightarrow 8$	50	811	2	12
	19-10	$7 \rightarrow 17$	12	488	1	18
	20-9	$39 \rightarrow 39$	23	2,980	3	74
	16-11	$4 \rightarrow 3$	24	147	1	8
	16-11	$4 \rightarrow 3$	24	63	1	5
IPC-GRID$^+$	5-5-5	$4 \rightarrow 3$	33	62	1	5
	5-10-10	$11 \rightarrow 11$	194	10,092	14	362
	10-5-5	$12 \rightarrow 10$	92	1,022	2	36
	10-10-10	$19 \rightarrow 19$	2,665	timeout	33	2,208
BLOCK-WORDS	8-20	$10 \rightarrow 10$	3,927	timeout	178	938
	8-20	$14 \rightarrow 14$	3,482	timeout	218	1,015
LOGISTICS	1-2-6-2-2-6	$18 \rightarrow 18$	3,527	timeout	155	639
	1-2-6-2-2-6	$18 \rightarrow 18$	2,496	timeout	137	483
	2-2-6-2-2-6	$17 \rightarrow 17$	timeout	timeout	594	1,943
	2-2-6-2-4-6	$17 \rightarrow 7$	timeout	timeout	6,752	6,065
	2-2-6-2-6-6	$16 \rightarrow 16$	timeout	timeout	5,215	timeout

- Two ASP-based encodings that exploit saturation based meta encoding methodology and advanced features of answer set solver for goal recognition design problems.
- Proof of correctness of the implementations.
- Demonstrate that answer set programming technologies are competitive with others.

Outline

- Grid map, some locations might be blocked
- A number of agents
- Agents can move to connected locations, one step at a time
- Agents need to visit sequences of locations (checkpoints)
- Constraints: not swapping, collision free, deadlines, group completion



(7 by 4 with block of 10)

Movement constraints

Overall Goal

- Paths for agents
- Optimal: minimal span, minimal total cost

Encoding GTAPF in ASP

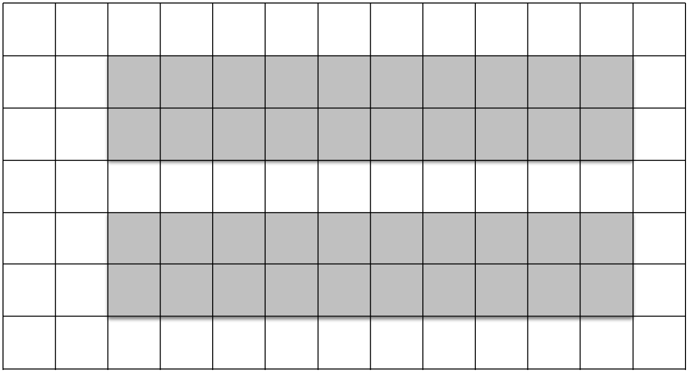
- Basic encoding: ASP encoding for planning problem, extended to multi agent domains:
  - holds($at(L), T$) becomes holds($at(R, L), T$):
    "Agent R is at L at time step T."
  - occ(A, T) becomes occ(R, A, T):
    "Agent R executes A at time step T."
- Adding rules to deal with constraints:
  - Enforcing movement constraints
    :- occ($R1, move(L), T$), occ($R2, move(L1), T$), edge($L, L1$), holds($at(R1, L1), T$), holds($at(R2, L), T$).
  - Enforcing visiting sequence: all checkpoints must be visited in the order.

Scalability and Efficiency

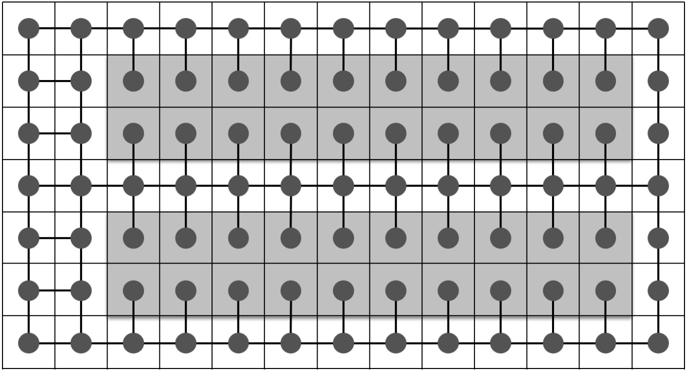
Computing solutions

- Optimal solution: compute solution when horizon is 0, 1, . . ., until solution is found.
- Scalability and efficiency:
  - not so good for TAPF problems Ma and Koenig (2016).
  - improvement if non-optimal solution is considered
- Kiva setting: abstraction can be employed (exploiting domain knowledge)

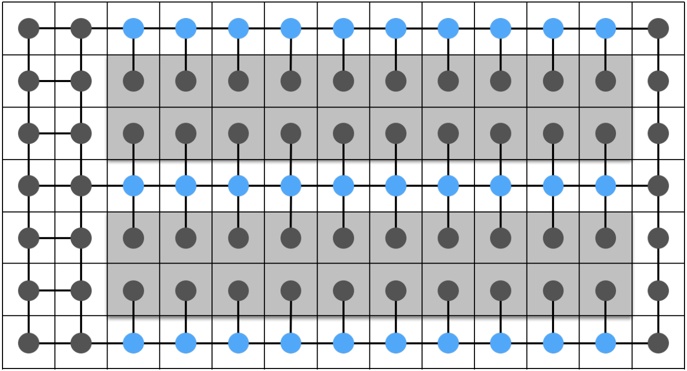
Idea of Abstraction



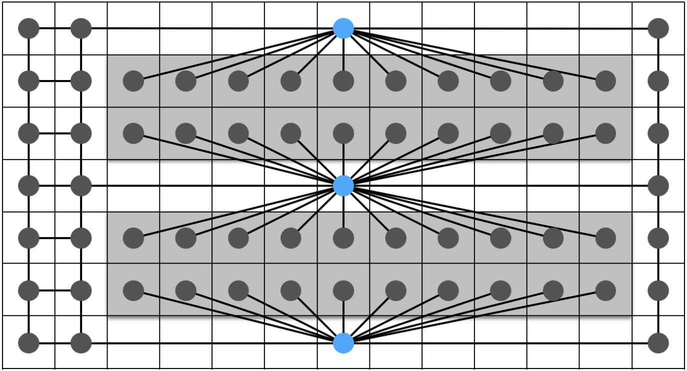
Idea of Abstraction



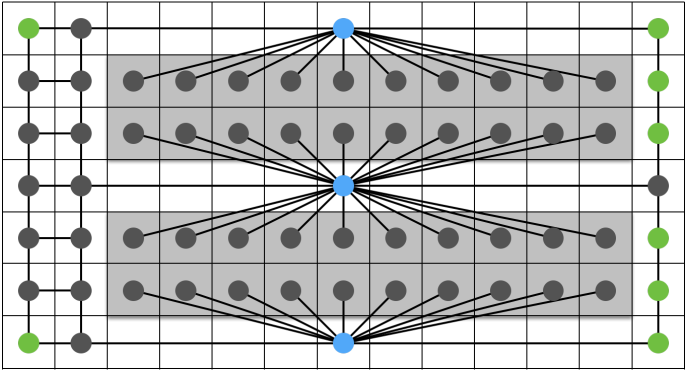
Idea of Abstraction



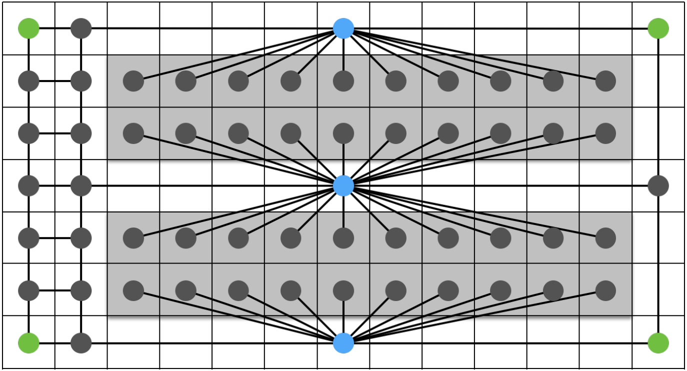
Idea of Abstraction



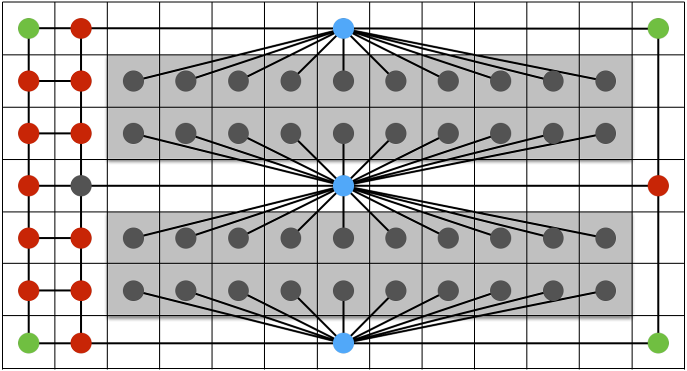
Idea of Abstraction



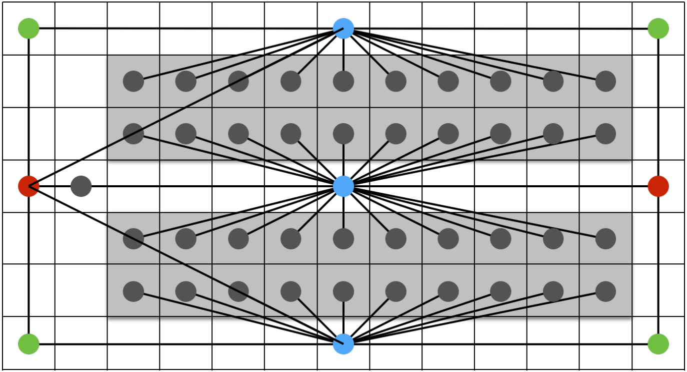
Idea of Abstraction



Idea of Abstraction



Idea of Abstraction



Simplified Map

Abstraction in Kiva

Non-optimal solution only

- Planing in simplified map
- Reassemble paths

Gain: from a map of 2 by 2 with block of 7 by 4 with block of 10.

Outline

Overview

Motivations

- CLP and Constraint Optimization used for modeling and solving planning problems
- Challenges in modeling multi-agent planning problems (e.g., use of extra-logical features like Linda)
- Distributed Constraint Optimization Problems (DCOP) as a potential novel paradigm for distributed planning

Goal

Modeling and Solving DCOPs in Logic Programming

DCOP

DCOP $\langle X, D, F, A, \alpha \rangle$

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of (decision) *variables*;
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite *domains*, where D_i is the domain of the variable $x_i \in \mathcal{X}$, for $1 \leq i \leq n$;
- $\mathcal{F} = \{f_1, \ldots, f_m\}$ is a finite set of *constraints*, where f_j is a k_j-ary function $f_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_{k_j}} \mapsto \mathbb{R} \cup \{-\infty\}$ that specifies the utility of each combination of values of variables in its *scope*; the scope is denoted by $scp(f_j) = \{x_{j_1}, \ldots, x_{j_{k_j}}\}$;[a]
- $\mathcal{A} = \{a_1, \ldots, a_p\}$ is a finite set of *agents*; and
- $\alpha : \mathcal{X} \mapsto \mathcal{A}$ maps each variable to an agent.

[a] For the sake of simplicity, we assume a given ordering of variables.

DCOP

$M = \langle X, D, F, A, \alpha \rangle$

- Assignment: $f : X \mapsto \bigcup_{D \in D} D$ such that $f(x) \in D_x$
- $C(M)$ set of all assignments
- Optimal Solution:

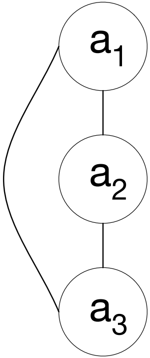
$$\mathbf{x} = \operatorname*{argmax}_{x \in \mathcal{C}(\mathcal{M})} \sum_{j=1}^{m} f_j(x_{f_j})$$

- Graph Representation: $G_{\mathcal{M}} = (V, E)$ where $V = \mathcal{A}$ and

$$E = \{\{a, a'\} \mid \{a, a'\} \subseteq \mathcal{A}, \exists f \in \mathcal{F} \textit{ such that } \{x_i, x_j\} \subseteq scp(f), \textit{ and } \alpha(x_i) = a, \alpha(x_j) = a'\}$$

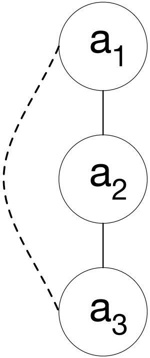
- Pseudo-tree: subgraph of $G_{\mathcal{M}}$ with all nodes of $G_{\mathcal{M}}$ and *(i)* the included edges form a tree, and *(ii)* two nodes connected in $G_{\mathcal{M}}$ are in the same branch of the tree.

DCOP



for $i < j$

x_i	x_j	Utilities
0	0	5
0	1	8
1	0	20
1	1	3



Distributed Pseudo-Tree Optimization Procedure (DPOP)

Separator sep_i of a_i

- Variables owned by ancestors of a_i
- Related via constraints to variables owned within the subtree rooted at a_i

DPOP:

1. Phase 1: Construction of Pseudo-Tree
2. Phase 2: Upward propagation of UTIL messages
3. Phase 3: Downward propagation of VALUE messages

Distributed Pseudo-Tree Optimization Procedure (DPOP)

UTIL

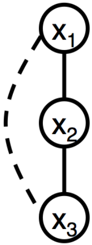
- $UTIL_{a_i}^{a_j}$ message from a_i to a_j; optimal utility for each combination of values to variables in sep_i
- $U = UTIL_{a_k}^{a_i} \oplus UTIL_{a_l}^{a_i}$ is the join of two UTIL matrices. $scp(U) = scp(UTIL_{a_k}^{a_i}) \cup scp(UTIL_{a_l}^{a_i})$. For each possible combination x of values of variables in $scp(U)$, $U(x) = UTIL_{a_k}^{a_i}(x_{UTIL_{a_k}^{a_i}}) + UTIL_{a_l}^{a_i}(x_{UTIL_{a_l}^{a_i}})$,
- Let $\alpha_i \subseteq scp(JOIN_{a_i}^{P_i})$. Let X_i be the set of all possible value combinations of variables in α_i.
  $U = JOIN_{a_i}^{P_i} \perp_{\alpha_i}$ is defined as:
  1. $scp(U) = scp(JOIN_{a_i}^{P_i}) \setminus \alpha_i$
  2. for each possible value combination x of variables in $scp(U)$, $U(x) = \max_{x' \in X_i} JOIN_{a_i}^{P_i}(x, x')$.

Distributed Pseudo-Tree Optimization Procedure (DPOP)

Each agent a_i does:

begin
- $JOIN_{a_i}^{P_i} = null$
- **foreach** $a_c \in C_i$ **do**
  - wait for $UTIL_{a_c}^{a_i}$; `/* message to arrive from` a_c `*/`
  - $JOIN_{a_i}^{P_i} = JOIN_{a_i}^{P_i} \oplus UTIL_{a_c}^{a_i}$; `/* join UTIL from children */`
- `/* join constraints with parent/pseudo-parents */`
- $JOIN_{a_i}^{P_i} = JOIN_{a_i}^{P_i} \oplus \left(\oplus_{f \in R_{a_i}} f \right)$
- `/* projection to eliminate owned variables */`
- $UTIL_{a_i}^{P_i} = JOIN_{a_i}^{P_i} \perp_{\alpha_i}$
- Send $UTIL_{a_i}^{P_i}$ message to its parent agent P_i

Distributed Pseudo-Tree Optimization Procedure (DPOP)



In x_3

x_1	x_2	Utilities	
		$x_3 = 0$	$x_3 = 1$
0	0	5+5=10	8+8=16
0	1	5+20=25	8+3=11
1	0	20+5=25	3+8=11
1	1	20+20=40	3+3=6

In x_2

x_1	Utilities	
	$x_2 = 0$	$x_2 = 1$
0	5+16=21	8+25=23
1	20+25=45	3+40=43

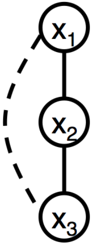
VALUE

- $VALUE_{P_i}^{a_i}$ message from parent P_i to child a_i
- $VALUE_{P_i}^{a_i}$ contains optimal value for variables of P_i and pseudo-parents

Each agent a_i does:

```
begin
    wait for VALUE_{P_i}^{a_i}(sep_i^*) message from parent P_i
    /* Determine optimal value for variables of a_i                */
    α_i^* ← argmax_{α_i ∈ X_i} JOIN_{a_i}^{P_i}(sep_i^*, α_i)
    foreach a_c ∈ C_i do
        let sep_i^** be partial optimal value assignment for variables in sep_c
        from sep_i^*
        send VALUE(sep_i^**, α_i^*) as VALUE_{a_i}^{a_c} message to its child agent a_c
```

Distributed Pseudo-Tree Optimization Procedure (DPOP)



In x_3

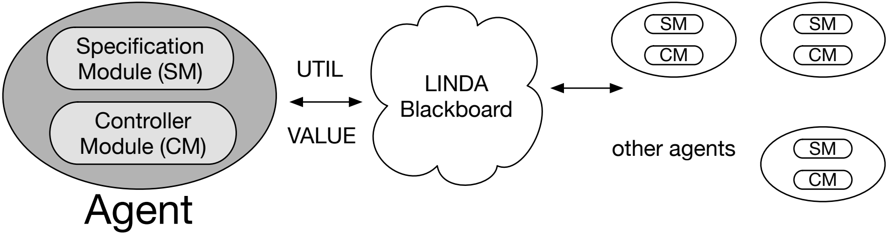
x_1	x_2	Utilities	
		$X_3 = 0$	$X_3 = 1$
0	0	5+5=10	8+8=16
0	1	5+20=25	8+3=11
1	0	20+5=25	3+8=11
1	1	20+20=40	3+3=6

In x_2

x_1	Utilities	
	$X_2 = 0$	$X_2 = 1$
0	5+16=21	8+25=23
1	20+25=45	3+40=43

ASP-DPOP

Agent Architecture



1. *Specification Module* (ASP)
2. *Controller Module* (Prolog)

SM

- For $x_j \in X$ with $D_j = \{\ell, \ldots, u\}$:

$$variable(x_j).$$
$$value(x_j, \ell..u).$$

- For $f \in F$ with $scp(f) = \{x_1, \ldots, x_k\}$:

$$constraint(f).$$
$$scope(f, x_1). \quad \ldots \quad scope(f, x_k).$$
$$f(u, v_1, \ldots, v_k).$$

SM

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{U}_{1,2}$
0	0	0
0	1	1
1	0	1
1	1	2

$$
\begin{aligned}
&x1\_cons\_x2(0,0,0).\\
&x1\_cons\_x2(1,0,1).\\
&x1\_cons\_x2(1,1,0).\\
&x1\_cons\_x2(2,1,1).
\end{aligned}
$$

SM

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{U}_{1,2}$
0	0	0
0	1	1
1	0	1
1	1	2

$$x1\_cons\_x2(0,0,0).$$
$$x1\_cons\_x2(1,0,1).$$
$$x1\_cons\_x2(1,1,0).$$
$$x1\_cons\_x2(2,1,1).$$

$$x1\_cons\_x2(U_1+U_2,U_1,U_2) :- value(x_1,U_1), value(x_2,U_2).$$

SM

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{U}_{1,2}$
0	0	0
0	1	1
1	0	1
1	1	2

$x1\_cons\_x2(0,0,0).$
$x1\_cons\_x2(1,0,1).$
$x1\_cons\_x2(1,1,0).$
$x1\_cons\_x2(2,1,1).$

$$x1\_cons\_x2(U_1+U_2, U_1, U_2) :- value(x_1, U_1), value(x_2, U_2).$$

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{U}_{1,2}$
0	0	0
0	1	$-\infty$
1	0	$-\infty$
1	1	$-\infty$

$x1\_cons\_x2(0,0,0).$

SM

Agent a_i information:

- Identification:
  $agent(a_i)$.
- For each $x \in X$ such that $\alpha(x) = a_i$:
  $owner(a_i, x)$.
- For each neighbor agent a_j:
  $neighbor(a_j)$.
- For each x' owned by a neighbor agent a_j:
  $owner(a_j, x')$.

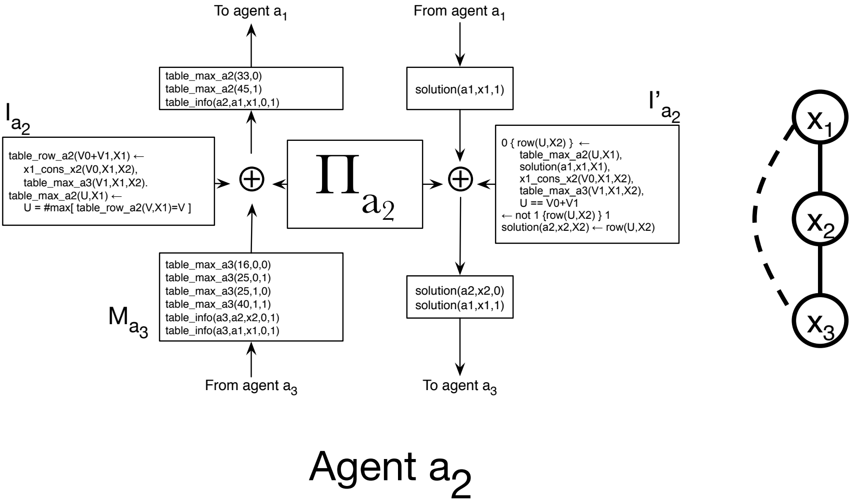
SM

- $UTIL_{a_i}^{P_i}$ with $sep_i = \langle x_1, \ldots, x_k \rangle$:

$$\begin{array}{l} table\_info(a_i, a_{i_1}, x_1, \ell_1, u_1). \\ \ldots \\ table\_info(a_i, a_{i_k}, x_k, \ell_k, u_k). \\ table\_max\_a_i(u, v_1, \ldots, v_k). \end{array}$$

- $VALUE_{P_i}^{a_i}$:
  $solution(a, x, v).$

CM



To agent a1
From agent a1
table_max_a2(33,0)
table_max_a2(45,1)
table_info(a2,a1,x1,0,1)
solution(a1,x1,1)
I'a2
Ia2
table_row_a2(V0+V1,X1) ←
x1_cons_x2(V0,X1,X2),
table_max_a3(V1,X1,X2).
table_max_a2(U,X1) ←
U = #max[table_row_a2(V,X1)=V]
Πa2
0 { row(U,X2) } ←
table_max_a2(U,X1),
solution(a1,x1,X1),
x1_cons_x2(V0,X1,X2),
table_max_a3(V1,X1,X2),
U == V0+V1
← not 1 {row(U,X2) } 1
solution(a2,x2,X2) ← row(U,X2)
Ma3
table_max_a3(16,0,0)
table_max_a3(25,0,1)
table_max_a3(25,1,0)
table_max_a3(40,1,1)
table_info(a3,a2,x2,0,1)
table_info(a3,a1,x1,0,1)
solution(a2,x2,0)
solution(a1,x1,1)
From agent a3
To agent a3
x1
x2
x3
Agent a2

CM

```
perform_Phase_2(ReceivedUTILMessages):-
  compute_separator(ReceivedUTILMessages, Separator),
  assert(separatorlist(Separator)),
  compute_related_constraints(ConstraintList),
  assert(constraintlist(ConstraintList)),
  generate_UTIL_ASP(Separator, ConstraintList),
  solve_answer_set1(ReceivedUTILMessages, Answer),
  store(Answer),
  send_message(a_i, a_p, util, Answer).
```

CM

```
perform_Phase_3(ReceivedVALUEMessage):-
  separatorlist(Separator),
  constraintlist(ConstraintList),
  generate_VALUE_ASP(Separator,ConstraintList),
  solve_answer_set2(ReceivedVALUEMessage, Answer),
  send_message_to_children(a_i, value, Answer).
```

Some Analysis

- ASP-DPOP is sound and complete in solving DCOPs
- Experimental comparison against
  - DPOP
  - AFP (Asynchronous Forward-Bounding): complete, search-based
  - Hard-Constraints DPOP (H-DPOP): DPOP with hard constraints and PH-DPOP
  - Open-DPOP

Experiments:

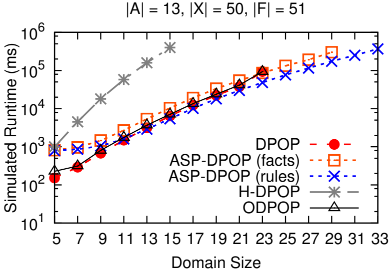
- Random Graphs; variation on number of nodes, density, and tightness
- Power Networks

Random Graphs

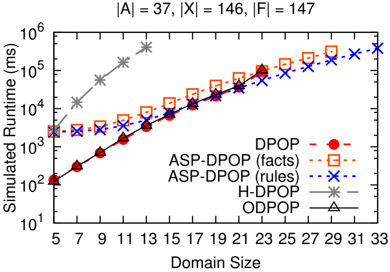
$\|X\|$	DPOP		H-DPOP		PH-DPOP		AFB		ASP-DPOP	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
15	86%	39,701	100%	148	98%	67,161	100%	53	100%	1,450
20	0%	-	100%	188	0%	-	100%	73	100%	1,777
25	0%	-	100%	295	0%	-	100%	119	100%	1,608
150	0%	-	0%	-	0%	-	100%	31,156	100%	37,862
200	0%	-	0%	-	0%	-	100%	117,913	100%	115,966
250	0%	-	0%	-	0%	-	0%	-	100%	298,361

Tight	DPOP		H-DPOP		PH-DPOP		AFB		ASP-DPOP	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
0.5	94%	38,043	100%	161	96%	71,181	100%	57	92%	4,722
0.6	90%	31,513	100%	144	98%	68,307	100%	52	100%	1,410
0.7	90%	39,352	100%	128	100%	49,377	100%	48	100%	1,059
0.8	92%	40,526	100%	112	100%	62,651	100%	57	100%	1,026

Power Network



13-Bus Configuration



37-Bus Configuration

$\lvert D_i \rvert$	13-Bus				37-Bus			
	5	7	9	11	5	7	9	11
H-DPOP	6,742	30,604	97,284	248,270	6,742	30,604	97,284	248,270
DPOP	3,125	16,807	59,049	161,051	3,125	16,807	59,049	161,051
ASP-DPOP	10	14	18	22	10	14	18	22

Conclusions

- ASP-DPOP is competitive
- Potential for many extensions:
    - Declarative encoding of constraints
    - Dedicated propagation algorithms within each agent
    - Intensional representation of UTIL tables

Outline

Summary

- Introduction of ASP, CLP, and CASP.
- Planning using ASP and CLP.
- Applications of ASP, CLP, and CASP.

References I

Baral, C., Kreinovich, V., and Trejo, R. (2000). Computational complexity of planning and approximate planning in the presence of incompleteness. *Artificial Intelligence*, 122:241–267.

Dimopoulos, Y., Nebel, B., and Koehler, J. (1997). Encoding planning problems in non-monotonic logic programs. In *Proceedings of European conference on Planning*, pages 169–181.

Dovier, A., Formisano, A., and Pontelli, E. (2010). Multivalued Action Languages with Constraints in CLP(FD). *Theory and Practice of Logic Programming*, 10(2):167–235.

Etzioni, O., Golden, K., and Weld, D. (1996). Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89:113–148.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–387.

References II

Goldman, R. and Boddy, M. (1994). Representing uncertainty in simple planners. In *KR 94*, pages 238–245.

Liberatore, P. (1997). The Complexity of the Language *A*. *Electronic Transactions on Artificial Intelligence*, 1(1–3):13–38.

Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54.

Ma, H. and Koenig, S. (2016). Optimal target assignment and path finding for teams of agents. pages 1144–1152.

Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Automated Planning and Scheduling, 2004*, pages 2–11.

Son, T. C. and Baral, C. (2001). Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91.

References III

Son, T. C., Baral, C., Tran, N., and McIlraith, S. (2006). Domain-Dependent Knowledge in Answer Set Planning. *ACM Transactions on Computational Logic*, 7(4).

Son, T. C. and Tu, P. H. (2006). On the Completeness of Approximation Based Reasoning and Planning in Action Theories with Incomplete Information. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 481–491.

Son, T. C., Tu, P. H., Gelfond, M., and Morales, R. (2005a). An Approximation of Action Theories of *AL* and its Application to Conformant Planning. In *Proceedings of the the 7th International Conference on Logic Programming and NonMonotonic Reasoning*, pages 172–184.

References IV

Son, T. C., Tu, P. H., Gelfond, M., and Morales, R. (2005b). Conformant Planning for Domains with Constraints — A New Approach. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1211–1216.

Subrahmanian, V. and Zaniolo, C. (1995). Relating stable models and ai planning domains. In *Proceedings of the International Conference on Logic Programming*, pages 233–247.

Thiebaux, S., Hoffmann, J., and Nebel, B. (2003). In Defense of PDDL Axioms. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*.

Tu, P., Pontelli, E., Son, T. C., and To, S. (2009). Applications of Parallel Processing Technologies in Heuristic Search Planning. *Concurrency and Computation*, 21(15):1928–1960.

References V

Tu, P., Son, T., and Baral, C. (2006). Reasoning and planning with sensing actions, incomplete information, and static causal laws using logic programming. *Theory and Practice of Logic Programming*, 7:1–74.

Tu, P., Son, T., Gelfond, M., and Morales, R. (2011). Approximation of action theories and its application to conformant planning. *Artificial Intelligence Journal*, 175(1):79–119.

Tu, P. H. (2007). *Reasoning AND Planning With Incomplete Information In The Presence OF Static Causal Laws*. PhD thesis, New Mexico State University.

Tu, P. H., Son, T. C., and Pontelli, E. (2007). Cpp: A constraint logic programming based planner with preferences. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 290–296. Springer.

References VI

Turner, H. (2002). Polynomial-length planning spans the polynomial hierarchy. In *Proc. of Eighth European Conf. on Logics in Artificial Intelligence (JELIA'02)*, pages 111–124.