Logic Programming
Foundations and Applications

Tran Cao Son¹, Enrico Pontelli¹, Marcello Balduccini²

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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
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. $\Sigma^2_P$, $\Pi^2_P$, etc.)

Answer set programming has been applied in several areas: reasoning about actions and changes, planning, configuration, wire routing, phylogenetic inference, semantic web, information integration, etc.
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)
- \( \text{not } a \): a negation-as-failure atom (naf-atom).
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)
- \( \text{not } a \): 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

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

- $a_i$, $b_j$: atom of a language $L$ ($L$ can either be propositional or first order)
- $\text{not } a$: 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

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

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

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 from function symbols and constants occurring in the program. \((U_\pi)\)
- **Herbrand base**: the set of ground atoms constructed from predicate symbols and ground terms from the Herbrand universe. \((B_\pi)\)
- **Rule with variables**: shorthand for the collection of its ground instances. \((\text{ground}(r))\)
- **Program with variables**: collection of ground instances of its rules. \((\text{ground}(\pi))\)
$L$ is a propositional language

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

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

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

$P_3 = \begin{cases} a \leftarrow \\ b \leftarrow c \end{cases}$
$L$ is a first order language

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

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

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

$$Q_3 = \{ p(f(X)) \leftarrow p(f(f(f(X)))) \leftarrow p(X) \}$$
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 \( T_P(X) \subseteq T_P(Y) \)
- \( \emptyset \subseteq T_P(\emptyset) \subseteq T_P(T_P(\emptyset)) \subseteq \ldots \subseteq T^n_P(\emptyset) \subseteq \text{lfp}(T_P) \) for \( n \to \infty \)
Computing $T_P$: Example 1

$L$: set of propositions such as $p, q, r, a, b \ldots$

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

\[
T_{P_1}(\emptyset) = \{a, c\} \\
T^2_{P_1}(\emptyset) = T_{P_1}(T_{P_1}(\emptyset)) = T_{P_1}(\{a, c\}) = \{a, c, b\} \\
T^3_{P_1}(\emptyset) = T_{P_1}(T^2_{P_1}(\emptyset)) = T_{P_1}(\{a, c, b\}) = \{a, c, b\} = \text{lfp}(T_{P_1})
\]
Computing $T_P$: Example 2

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

\[ P_2 = \begin{cases} 
  a & \leftarrow b \\
  b & \leftarrow a, c \\
  p & \leftarrow a, p \\
  c & \leftarrow 
\end{cases} \]

\[ T_{P_2}(\emptyset) = \{c\} \]
\[ T_{P_2}^2(\emptyset) = T_{P_2}(T_{P_2}(\emptyset)) = T_{P_2}(\{c\}) = \{c\} = \text{lfp}(T_{P_2}) \]
Computing $T_P$: Example 3

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

\[
T_{P_3}(\emptyset) = \{a\} \\
T_{P_3}^2(\emptyset) = T_{P_3}(T_{P_3}(\emptyset)) = T_{P_3}(\{a\}) = \{a\} = \text{lfp}(T_{P_3})
\]
Computing $T_P$: Example 4 and 5

$$P_4 = \{ a \leftarrow b, b \leftarrow a \}$$

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

$$P_5 = \{ a \leftarrow b \leftarrow a, b \}$$

$$T_{P_5}(\emptyset) = \{a\}$$

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

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

The language \( L \) of a program \( \Pi \) is often given \textit{implicitly}.

Rules with Variables

\( \text{ground}(r, L) \): 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 \( \text{ground}(r, L_1) \) will consist of the following rules:

\[
\begin{align*}
p(f(a)) & \leftarrow p(a). \\
p(f(b)) & \leftarrow p(b). \\
p(f(f(a))) & \leftarrow p(f(a)). \\
p(f(f(b))) & \leftarrow p(f(b)). \\
\vdots
\end{align*}
\]
Main Definitions

- \( \text{ground}(r, L) \): 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 \( \Pi \):
  - \( \text{ground}(\Pi, L) = \bigcup_{r \in \Pi} \text{ground}(r, L) \)
  - \( L_\Pi \): The language of a program \( \Pi \) is the language consists of the constants, variables, function and predicate symbols (with their corresponding arities) occurring in \( \Pi \). In addition, it contains a constant \( a \) if no constant occurs in \( \Pi \).
  - \( \text{ground}(\Pi) = \bigcup_{r \in \Pi} \text{ground}(r, L_\Pi) \).
Example 2

\[
\Pi:
\begin{align*}
p(a) & .
p(b) & .
p(c) & .
p(f(X)) & \leftarrow p(X).
\end{align*}
\]

\[
ground(\Pi):
\begin{align*}
p(a) & \leftarrow .
p(b) & \leftarrow .
p(c) & \leftarrow .
p(f(a)) & \leftarrow p(a).
p(f(b)) & \leftarrow p(b).
p(f(c)) & \leftarrow p(c).
p(f(f(a))) & \leftarrow p(f(a)).
p(f(f(b))) & \leftarrow p(f(b)).
p(f(f(c))) & \leftarrow p(f(c)).
\end{align*}
\]

\[
p(f^{k+1}(x)) \leftarrow p(f^k(x)). \text{ for } x \in \{a, b, c\}
\]
Herbrand Interpretation I

Definition
The Herbrand universe (resp. Herbrand base) of $\Pi$, denoted by $U_\Pi$ (resp. $B_\Pi$), is the Herbrand universe (resp. Herbrand base) of $L_\Pi$.

Example
For $\Pi = \{ p(X) \leftarrow q(f(X), g(X)). \quad r(Y) \leftarrow \}$
the language of $\Pi$ 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 $a$.

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

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

A Herbrand interpretation of a program $\Pi$ is a set of atoms from its Herbrand base.
Let $\Pi$ be a positive program and $I$ be a Herbrand interpretation of $\Pi$. $I$ is called a Herbrand model of $\Pi$ if for every rule “$a_0 \leftarrow a_1, \ldots, a_n$” in $\text{ground}(\Pi)$, $a_1, \ldots, a_n$ 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 $\Pi$ is called the *minimal model* of $\Pi$ and is denoted by $M_\Pi$.

**Computing** $M_\Pi$. Let $\Pi$ be a program. We define a fixpoint operator $T_\Pi$ that maps a set of atoms (of program $\Pi$) to another set of atoms as follows.

$$T_\Pi(X) = \{ a \mid a \in B_\Pi, \text{there exists a rule } a \leftarrow a_1, \ldots, a_n \text{ in } \Pi \text{ s. t. } a_i \in X \}$$ (1)
Note: By $a_0 \leftarrow a_1, \ldots, a_n$ in $\text{ground}(\Pi)$ we mean there exists a rule $b_0 \leftarrow b_1, \ldots, b_n$ in $\Pi$ (that might contain variables) and a ground substitution $\sigma$ such that $a_0 = b_0\sigma$ and $a_i = b_i\sigma$.

Remark

The operator $T_\Pi$ 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 $T_\Pi$

- $T_\Pi$ is monotonic: $T_\Pi(X) \subseteq T_\Pi(Y)$ if $X \subseteq Y$.
- $T_\Pi$ 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 $\Pi$ with finite $B_\Pi$.
We denote the least fix point of $T_\Pi$ with $T_\Pi^\infty(\emptyset)$ or $\text{lfp}(T_\Pi)$.

**Theorem**

$M_\Pi = \text{lfp}(T_\Pi)$.

**Theorem**

*For every positive program $\Pi$ without constraint, $M_\Pi$ is unique.*
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 \( \Pi \) be a program and \( X \) be a set of atoms, by \( \Pi^X \) we denote the program obtained from \( \text{ground}(\Pi) \) by

1. Deleting from \( \text{ground}(\Pi) \) 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.
**Remark**

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

**Remark**

$\Pi^X$ is a positive program.

**Definition**

A set of atoms $X$ is called an *answer set* of a program $\Pi$ if $X$ is the minimal model of the program $\Pi^X$.

**Theorem**

*For every positive program $\Pi$, the minimal model of $\Pi$, $M_\Pi$, is also the unique answer set of $\Pi$.*
Consider $\Pi_2 = \{ a \leftarrow \text{not } b. \quad b \leftarrow \text{not } a. \}$. We will show that its has two answer sets $\{a\}$ and $\{b\}$.

<table>
<thead>
<tr>
<th>$S_1 = \emptyset$</th>
<th>$S_2 = {a}$</th>
<th>$S_3 = {b}$</th>
<th>$S_4 = {a, b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi_{S_1}^{\Pi_2}$ :</td>
<td>$\Pi_{S_2}^{\Pi_2}$ :</td>
<td>$\Pi_{S_3}^{\Pi_2}$ :</td>
<td>$\Pi_{S_4}^{\Pi_2}$ :</td>
</tr>
<tr>
<td>$a \leftarrow$</td>
<td>$a \leftarrow$</td>
<td>$b \leftarrow$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b \leftarrow$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$M_{\Pi_{S_1}^{\Pi_2}} = {a, b}$</td>
<td>$M_{\Pi_{S_2}^{\Pi_2}} = {a}$</td>
<td>$M_{\Pi_{S_3}^{\Pi_2}} = {b}$</td>
<td>$M_{\Pi_{S_4}^{\Pi_2}} = \emptyset$</td>
</tr>
<tr>
<td>$M_{\Pi_{S_1}^{\Pi_2}} \neq S_1$</td>
<td>$M_{\Pi_{S_2}^{\Pi_2}} = S_2$</td>
<td>$M_{\Pi_{S_3}^{\Pi_2}} = S_3$</td>
<td>$M_{\Pi_{S_4}^{\Pi_2}} \neq S_4$</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

**Remark**

A program may have zero, one, or more than one answer sets.
A set of atoms \( S \) is **closed under** a program \( \Pi \) if for all rules of the form \( a_0 \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n \) in \( \Pi \), \( \{a_1, \ldots, a_m\} \subseteq S \) 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** \( \Pi \) if for all \( p \in S \) there is a rule of the form \( p \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n \) in \( \Pi \), 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 \( \Pi \) iff (i) \( S \) is closed under \( \Pi \) and (ii) there exists a level mapping function \( \lambda \) (that maps atoms in \( S \) to a number) such that for each \( p \in S \) there is a rule in \( \Pi \) of the form \( p \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n \) 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 \( \Pi \).

It is known that if \( S \) is an answer set of \( \Pi \) then...
Further intuitions behind the semantics II

1. \( S \) must be closed under \( \Pi \) and
2. \( S \) must be supported by \( \Pi \).

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 \( \Pi_3 \). As such, \( X_0 \) cannot be an answer set of \( \Pi_3 \).

2. For \( X_7 = \{ p, a, b \} \). Take \( a \in X_7 \): the only rule in \( \Pi_3 \) 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 \( \Pi_3 \) and hence \( X_7 \) cannot be an answer set of \( \Pi_3 \).
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 $\Pi$ be a program with constraints. Let $\Pi_O = \{r \mid r \in \Pi, \text{ } r \text{ has non-empty head}\}$ and $\Pi_C = \Pi \setminus \Pi_O$ ($\Pi_O \text{ and } \Pi_C$: set of normal logic program rules and constraints in $\Pi$, respectively).

**Definition**

A set of atoms $S$ is an answer sets of a program $\Pi$ if it is an answer set of $\Pi_O$ and satisfies all the constraints in $\text{ground}(\Pi_C)$. 
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
...```

Son, Pontelli, Balduccini (NMSU & Drexel)
Answer Set Programming
General Methodology

Answer Set Programming

Traditional Approach

Problem → Solutions

Modeling

Logic Program → Solver → Answer Sets

Interpreting

Son, Pontelli, Balduccini (NMSU & Drexel)
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 $\Pi_G$ such that each answer set of $\Pi_G$ gives us a solution to the problem. Furthermore, each solution of the problem corresponds to an answer set.

The program $\Pi_G$ 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 $\text{color}(X, Y)$ - node $X$ is assigned the color $Y$.
- Generating the solutions: Each node is assigned one color. The three rules

\[
\begin{align*}
\text{color}(X, \text{red}) \leftarrow & \text{not color}(X, \text{blue}), \text{not color}(X, \text{yellow}). \quad (2) \\
\text{color}(X, \text{blue}) \leftarrow & \text{not color}(X, \text{red}), \text{not color}(X, \text{yellow}). \quad (3) \\
\text{color}(X, \text{yellow}) \leftarrow & \text{not color}(X, \text{blue}), \text{not color}(X, \text{red}). \quad (4)
\end{align*}
\]

- 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:

\[
\leftarrow \text{edge}(X, Y), \text{color}(X, C), \text{color}(Y, C). \quad (5)
\]
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 $\Pi_G$ be a program constructed from $G$. Each solution of $G$ corresponds to an answer set of $\Pi_G$ 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

\[
1 \{\text{color}(X, \ C) : \text{is\_color}(\ C)\} 1 :- \text{node}(X).
\]

and a set of atoms

\[
is\_\text{color}(\text{yellow}). \quad is\_\text{color}(\text{red}). \quad is\_\text{color}(\text{blue}).
\]

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

\[
\{p_1, p_2, \ldots, p_k\} \leq u
\]

where \(0 \leq l \leq u\) 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 \(l \leq |\{p_i \mid p_i \in S\}| \leq u\).
Syntactic Extensions of Logic Programming

Weighted Atoms

\[ l_0 = w_0, \ldots, l_k = w_k, \text{ not } l_{k+1} = w_{k+1}, \ldots, \text{ not } l_{k+n} = w_{k+n} \] \[ \{ \text{u} \} \text{ where } l_i \text{'s are atoms, } w_i \text{ are integers, and } l \leq u \text{ are integers. This atom is true with respect to a set of literals } S \text{ if } l \leq \sum_{0 \leq j \leq k} w_j + \sum_{k+1 \leq j \leq k+n} w_j \leq u. \]

Special case: choice atom – \( w_i = 1 \) for every \( i \).

Aggregates

\( Sum(\Omega), Count(\Omega), Average(\Omega), Min(\Omega), Max(\Omega) \) where \( \Omega \) 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.
**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

\[
\text{col}(1..n). \\
\text{row}(1..n).
\]

// $n$ columns

// $n$ 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

\[
1\{\text{cell}(I, J) : \text{row}(J)\}1 \leftarrow \text{col}(I).
\]

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
  \[
  \leftarrow \text{cell}(I, J1), \text{cell}(I, J2), J1 \neq J2.
  \]

- No two queens on the same column (not really needed)
  \[
  \leftarrow \text{cell}(I1, J), \text{cell}(I2, J), I1 \neq I2.
  \]

- No two queens on the same diagonal
  \[
  \leftarrow \text{cell}(I1, J1), \text{cell}(I2, J2), |I1 - I2| = |J1 - J2|
  \]
% 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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
Explore alternative approaches to encode action languages (e.g., $B$) 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., $B^{MV}$)
Comparative experimental performance testing
Intuition

- **Traditional Logic Programming:**
  - Terms are uninterpreted
    - \( p(X) :- X=\text{square}(2). \)
    - \(?- p(X). \)
    - \(?- 3=2+1. \)
    - \(?- 3=X+1. \)
  - \( X=\text{square}(2) \) no no
  - **Prolog: extra-logical predicates**
    - \( p(X,Y) :- X \text{ is } Y*2. \)
    - \(?- p(6,Y). \)
    - ERROR: is/2: Arguments not instantiated
  - **Prolog: forces a generate & test style**
    - \( p(X,Y) :- \text{domain}(X), \text{domain}(Y), X > Y. \)

- **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
  ⇒ many unuseful branches explored

• CLP paradigm
  test then generate
  ⇒ cuts branches soon, avoiding exploration
Constraint Logic Programming

SEND
+MORE
---
MONEY
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*S + 1000*O + 100*N + 10*E + Y.

Declarative
1,393,690 Backtrackings
Constraint Logic Programming

Less Declarative
More Efficient
Requires Order of Execution

```prolog
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.
```
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

\[
X \#< Y \text{ :- }
\]

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X #&lt; Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>✗</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>✔</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>✔</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>✗</td>
</tr>
</tbody>
</table>

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)

\[ \Sigma_X = (F_X, \Pi_X) \text{ where} \]
\[ F_X = \{+, -\} \]
\[ \Pi_X = \{=, \neq, \leq\} \cup \{\in_n^m | n \leq m\} \]

\[ D = \mathbb{Z} \]

- 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

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sort</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLP(FD)</td>
<td>Finite Domains</td>
<td>Local Consistency</td>
</tr>
<tr>
<td>CLP($R$)</td>
<td>Real, Linear Constraints</td>
<td>Simplex</td>
</tr>
<tr>
<td>CLP($Q$)</td>
<td>Rationals, Linear Constraints</td>
<td>Simplex</td>
</tr>
<tr>
<td>CLP($SET$)</td>
<td>Hereditarily Finite Sets</td>
<td>Ad-Hoc, ACI</td>
</tr>
<tr>
<td>Prolog-II</td>
<td>Finite Trees</td>
<td>Ad-Hoc</td>
</tr>
</tbody>
</table>
• 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 $\Pi_P$ atom or $B_i \in L$

• Goal:
  $$? :- B_1, \ldots, B_k$$
  where $B_i$ is a $\Pi_P$ 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 $\text{consistent}(C)$
  - $\langle G, C \rangle$ failed if $\neg \text{consistent}(C)$

- 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 $\Pi_P$ atom and there is a rule $p(t_1, \ldots, t_n) : \neg \vec{B}$ 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
    - $C' = C \land \{B_i\}$
    - $G' = G \setminus \{B_i\}$ if $\text{consistent}(G')$ or
      - $G' = \emptyset$ if $\neg \text{consistent}(G')$

- Derivation: $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$ where
  - $S_0 = \langle G, \text{true} \rangle$
  - $S_i \Rightarrow S_{i+1}$ for $0 \leq i < n$
CLP(FD)

- $\Pi_X = \{=,\neq,\lt,\gt,\leq,\geq\} \cup \{\in_m^n \mid n \leq m\}$
- $F_X = \{+,-,\ast,\div\}$
- SWI-Prolog
  - $X \text{ in } 1..10 \quad Y \text{ in } 1..4 \lor 9..12$
  - $X+1 \neq Y \quad X-2\leq Y+1$
  - $X \text{ in } 1..10, X \lt 2 \lor X \gt 9$

- Global Constraints:
  - $\text{sum([X,Y,Z], \gt, 10)}$
  - $\text{all_different([X,Y,Z])}$

- During resolution:
  - Check consistency
  - Possibly simplify constraints:
    - $?- X \text{ in } 1..10, X\gt8.$
    - $X \text{ in } 9..10$
CLP(FD)

- Search:
  - `labeling(+Options,+Variables)`
  - Variable Selection Strategies:
    - `ff`
    - `ffc`
    - `leftmost`
  - Branching Strategy:
    - `step`: $X#=V$ or $X#\not=V$
    - `enum`: $X=V1$ or $X=V2$ or ...
    - `bisect`: $X #< M$ or $X #>= M$

- Alternating Labeling and propagation

- Optimization: `labeling([max(Expression)], Variables)`
Outline

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
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 $\sigma$ is a list of variables and $\rho$ 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 $all\_different(x_1, \ldots, x_k)$. 

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.

\[ P = \langle \{x, y, z\}, \{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}, \{\text{sum}(x, y, z) = 23, x \% 2 = 1, y \% 2 = 1, z \% 2 = 1\} \rangle \]
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 \( \sigma \). 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 \( C \).
Find three integers between 1 and 10 whose sum is 23.

Additional constraint: only odd integers are allowed.

\[ P = \langle \{x, y, z\}, \{\{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}, \{1, 2, \ldots, 10\}\}, \{\text{sum}(x, y, z) = 23, x \mod 2 = 1, y \mod 2 = 1, z \mod 2 = 1\}\rangle \]

- \( x := 0 \) does not satisfy the constraint \( x \mod 2 = 1 \).
- \( x := 1 \) satisfies the constraint \( x \mod 2 = 1 \).
- \( \{x := 9, y := 9, z := 5\} \) 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 $[1, 10]$

\[
\text{sum}(x, y, z) = 23.
\]
\[
x \mod 2 = 1.
\]
\[
y \mod 2 = 1.
\]
\[
z \mod 2 = 1.
\]
- Answer set: \{\text{sum}(x, y, z) = 23, x \mod 2 = 1, y \mod 2 = 1, z \mod 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 $[1,10]$

$hard \lor \neg hard$.

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

- 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 \texttt{EZCSP}

Syntax: extends ASP by pre-interpreted atoms (\textit{CSP atoms}) that encode CSP constraints:

1. \texttt{domain(d):} \textit{domain declaration}, e.g.
   - \texttt{domain(fd)} for numerical constraints over finite domains
   - \texttt{domain(nlp)} for non-linear constraints

2. \texttt{var(x):} \textit{x} is a CSP variable
   Variant: \texttt{var(x, l, u):} \textit{x} is a CSP variable with range \([l, u]\).

3. \texttt{required(\gamma):} \textit{\gamma} is required to be in the CSP.

Possible, but out-of-scope: using \texttt{EZCSP} beyond CASP
Example of Syntax

% Resources
resource(1). resource(2). resource(3).
available(I) ← resource(I), not ¬available(I).
¬available(2).

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

var(x(I)) ← available(I).

required(x(I) > 0) ← available(I).
required(x(I) < 5) ← available(I).
EzCSP Semantics: Translation Function

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

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

Example:

\[
\tau^{-1}(\text{required}(gt(x, "2.4"))) \text{ is } x > 2.4
\]
Answer Sets under the \textbf{EZCSP} Semantics

Given program $\Pi$:

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

Definition

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

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

\[\text{resource}(1). \text{resource}(2). \text{resource}(3).\]
\[\text{available}(I) \leftarrow \text{resource}(I), \text{not} \; \neg \text{available}(I).\]
\[\neg \text{available}(2).\]
\[\text{domain}(fd).\]
\[\text{var}(x(I)) \leftarrow \text{available}(I).\]
\[\text{required}(x(I) > 0) \leftarrow \text{available}(I).\]
\[\text{required}(x(I) < 5) \leftarrow \text{available}(I).\]

Answer set of $\Pi$ under ASP semantics:
\[A = \{\text{resource}(1), \text{resource}(2), \text{resource}(3), \]
\[\text{available}(1), \neg \text{available}(2), \text{available}(3), \]
\[\text{domain}(fd), \text{var}(x(1), \text{var}(3), \]
\[\text{required}(x(1) > 0), \text{required}(x(1) < 5), \text{required}(x(3) > 0), \ldots\}\]
Example of Semantics

Answer set of $\Pi$ under ASP semantics:

$A = \{resource(1), resource(2), resource(3),$

$\quad available(1), \neg available(2), available(3),$

$\quad domain(fd), var(x(1), var(3),$

$\quad required(x(1) > 0), required(x(1) < 5), required(x(3) > 0), \ldots\}$

- $V(A) = \{x(1), x(3)\}$
- $\tau^{-1}(A) = \{x(1) > 1, x(1) < 5, x(3) > 1, x(3) < 5\}$
- 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 $\Pi$ under the EZCSP semantics:

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

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

$\ldots$
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

\textit{domain(nlp)}. 

Advanced Example

If the object is held, it remains idle.

\[
\text{\texttt{var}(x). \hspace{0.5em} \text{\texttt{var}(y).}}
\]
\[
\text{\texttt{\neg in\_motion} \leftarrow \text{\texttt{held}.}}
\]
\[
\text{\texttt{required}(x = 0) \leftarrow \text{\texttt{\neg in\_motion}.}}
\]
\[
\text{\texttt{required}(y = 0) \leftarrow \text{\texttt{\neg in\_motion}.}}
\]
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.

\[
\text{var}(a). \ \text{var}(t). \\
\text{in\_motion} \leftarrow \text{pushed}, \text{not } \neg \text{in\_motion}. \\
\text{required}(x = \cos(a \cdot \pi/180) \cdot t) \leftarrow \text{in\_motion}. \\
\text{required}(y = \sin(a \cdot \pi/180) \cdot t) \leftarrow \text{in\_motion}. 
\]
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?

\[ \text{required}(a = 30). \]
\[ \text{required}(t = 2). \]
\[ \text{pushed}. \]
\[ \neg \text{stuck}. \]
Advanced Example

Consider $\Pi_1$ as above and $A_1 = Q_1 \cup P_1$ where:

$Q_1 = \{pushed, \neg stuck, in\_motion\}$

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

Clearly:

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

Hence, $\langle A_1, \{t = 2, a = 30, x = 1.7305081, y = 1\}\rangle$ is the answer set of $\Pi_1$ 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 $\rightarrow$ EZCSP, preserving $T$-solutions
- **CASP2T Extractor**: answer set $\rightarrow$ candidate $T$-solution
- **T-checker**: verifies candidate $T$-solution
- **T2CASP Expansion**: rejected candidate $T$-solution $\rightarrow$ EZCSP rules
**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

<table>
<thead>
<tr>
<th>Domains</th>
<th>EZCSP</th>
<th>CASP solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLASP+BP</td>
<td>CLASP+MZN</td>
</tr>
<tr>
<td>RF</td>
<td>(11) 10,093.93</td>
<td>N/A</td>
</tr>
<tr>
<td>WS</td>
<td>(30) 18,000.00</td>
<td>(30) 18,000.00</td>
</tr>
<tr>
<td>IS</td>
<td>(16) 11,176.68</td>
<td>N/A</td>
</tr>
<tr>
<td>IS*</td>
<td>(21) 14,372.70</td>
<td>(19) 12,466.03</td>
</tr>
</tbody>
</table>

* ASP/CASP competition problems

<table>
<thead>
<tr>
<th>Domain</th>
<th>EZCSP</th>
<th>CASP solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>CLASP+BP</td>
<td>CMODELS+BP</td>
</tr>
<tr>
<td>Generator</td>
<td>(0) 0.42</td>
<td>(0) 0.43</td>
</tr>
<tr>
<td></td>
<td>(2) 1,430.95</td>
<td>(4) 2,400.97</td>
</tr>
</tbody>
</table>

* PDDL+ planning, linear domains
Outline

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. **Action Description Languages**
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
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 $\ell$ be a Boolean literal. We have:

- **Executability conditions:**
  
  $\text{executable}(a, \text{[list-of-preconditions]})$
  
  asserting that the given preconditions have to be satisfied in the current state for the action $a$ to be executable.

- **Dynamic causal laws:**
  
  $\text{causes}(a, \ell, \text{[list-of-preconditions]})$
  
  describes the effect (the fluent literal $\ell$) of the execution of action $a$ in a state satisfying the given preconditions.

- **Static causal laws:**
  
  $\text{caused}([\text{list-of-preconditions}], \ell)$
  
  describes the fact that the fluent literal $\ell$ is true in a state satisfying the given preconditions.
The language $\mathcal{B}$: Initial State and Goal

- **Initial state**
  
  \[ \text{initially}(\ell) \]

  asserts that $\ell$ holds in the initial state.

- **Goal**
  
  \[ \text{goal}(\ell) \]

  asserts that $\ell$ 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
  \( \text{initially}(\neg \text{dead}) \) and
  \( \text{initially}(\text{walking}) \)
- Initially, the gun is loaded:
  \( \text{initially}(\text{loaded}) \)
- Shooting causes the turkey to be dead if the gun is loaded can be expressed by
  \( \text{causes}(\text{shoot}, \text{dead}, [\text{loaded}]) \) and
  \( \text{causes}(\text{shoot}, \neg \text{loaded}, [\text{loaded}]) \)
- Un/Loading the gun causes the gun to be un/loaded
  \( \text{causes}(\text{load}, \text{loaded}, []) \) and
  \( \text{causes}(\text{unload}, \neg \text{loaded}, []) \)
- Dead turkeys cannot walk
  \( \text{caused}(\neg \text{walking}, [\text{dead}]) \)
Action description: Example II

A gun can be loaded only when it is not loaded
executable\((load, \neg loaded)\)

So, an action theory for the Yale Shooting problem is

\[ I_y = \{ \text{initially}(\neg dead), \text{initially}(walking), \text{initially}(loaded) \} \]

and

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

<table>
<thead>
<tr>
<th>$\mathcal{B}$</th>
<th>PDDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td><strong>√</strong></td>
</tr>
<tr>
<td>Fluent</td>
<td><strong>√</strong></td>
</tr>
<tr>
<td>Conditional Effect</td>
<td><strong>√</strong></td>
</tr>
<tr>
<td>Executability condition</td>
<td><strong>√</strong></td>
</tr>
<tr>
<td>Static causal law (<em>allow cyclic</em>)</td>
<td><strong>√</strong></td>
</tr>
<tr>
<td>Ground Instantiations (Variables: shorthand)</td>
<td>Defined fluent or axiom (<em>no cyclic</em>)</td>
</tr>
<tr>
<td></td>
<td>Typed Variables</td>
</tr>
</tbody>
</table>

#### Notes

1. Dealing directly with static causal laws is advantageous Thiebaux et al. (2003).
2. Not many planners deal with static causal laws directly.
Given an action theory \((D, \delta)\), 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 \(\langle s_1, shoot, s_2 \rangle\):
Action language $\mathcal{B}$, Complete Information (Semantics)

Given an action domain $D$, a fluent literal $l$, sets of fluent literals $\sigma$ and $\psi$

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

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

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

  (* — present unless otherwise stated)

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

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

It can be shown that

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

for every $\delta_d$ 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:** $\text{flame}$, $\text{opened}(V)$, $\text{pressured}(P)$, $0 \leq V \leq n$, $0 \leq P \leq n + 1$,
- **Actions:** $\text{open}(V)$, $\text{close}(V)$
- **Action domain:**

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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
ASP & CLP in Classical Planning
Classical Planning Complexity

Definition (Planning Problem)

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

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

Theorem (Complexity)

- $(D, \delta)$ is deterministic: NP-hard even for plans of length 1, NP-complete for polynomial-bounded length plans (Classical Planning).
- $(D, \delta)$ is non-deterministic: $\Sigma^P_2$-hard even for plans of length 1, $\Sigma^P_2$-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 $\Pi(P, N)$ 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 $\Pi(P, N)$ 1-to-1 correspond to solutions of length $\leq N$ 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 $\Pi_D(N, O)$. 

```prolog
% time(0 .. N) fluent(f). action(a). literal(F): ~ fluent(F). literal(neg(F)): ~ fluent(F).
complement(F, neg(F)). complement(neg(F), F).
```
ASP Encoding of $B$

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 $\Pi_D(N, O)$.

\[
time(0..N)
\]
ASP Encoding of $B$

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 $\Pi_D(N, O)$.

\[
\text{time}(0..N)
\]

\[
\text{fluent}(f). \quad \text{action}(a).
\]
ASP Encoding of \( B \)

Ideas from [Gelfond and Lifschitz 92]

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

\[
\text{time}(0..N)
\]

\[
\text{fluent}(f). \quad \text{action}(a).
\]

\[
\text{literal}(F): \neg \text{fluent}(F). \quad \text{literal}($\neg F$): \neg \text{fluent}(F).
\]

\[
\text{complement}(F, $\neg F$). \quad \text{complement}(\neg F, F).
\]
ASP Encoding of $\mathcal{B}$

The predicate $\text{holds(FluentLiteral,Time)}$ is defined using the axioms:

- $\text{executable}(a, [\ell_1^1, \ldots, \ell_{r_1}^1]), \ldots, \text{executable}(a, [\ell_1^h, \ldots, \ell_{r_h}^h])$:
  
  $\text{possible}(a, T): \neg \text{time}(T), \text{holds}(\ell_1^1, T), \ldots, \text{holds}(\ell_{r_1}^1, T)$.
  
  $\ldots$

  $\text{possible}(a, T): \neg \text{time}(T), \text{holds}(\ell_1^h, T), \ldots, \text{holds}(\ell_{r_h}^h, T)$.
ASP Encoding of $\mathcal{B}$

The predicate \texttt{holds(FluentLiteral,Time)} is defined using the axioms:

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

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

  \[
  \vdots
  \]

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

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

  \[
  \text{holds}(\ell, T): -\text{time}(T), \text{holds}(\ell_1, T), \ldots, \text{holds}(\ell_r, T).
  \]
ASP Encoding of $\mathcal{B}$

The predicate \texttt{holds(FluentLiteral,Time)} is defined using the axioms:

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

\begin{align*}
\texttt{possible}(a, T) & : \neg \texttt{time}(T), \texttt{holds}(\ell^1_1, T), \ldots, \texttt{holds}(\ell^1_{r_1}, T). \\
\vdots \\
\texttt{possible}(a, T) & : \neg \texttt{time}(T), \texttt{holds}(\ell^h_1, T), \ldots, \texttt{holds}(\ell^h_{r_h}, T).
\end{align*}

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

\begin{align*}
\texttt{holds}(\ell, T) & : \neg \texttt{time}(T), \texttt{holds}(\ell_1, T), \ldots, \texttt{holds}(\ell_r, T).
\end{align*}

- \textbf{Dynamic causal laws}: \texttt{causes}(a, \ell, [\ell^1_1, \ldots, \ell^1_{r_1}]):

\begin{align*}
\texttt{holds}(\ell, T + 1) & : \neg \texttt{time}(T), \texttt{occ}(a, T), \\
& \texttt{holds}(\ell^1_1, T), \ldots, \texttt{holds}(\ell^1_r, T).
\end{align*}
ASP Encoding of $\mathcal{B}$

- State consistency:

  \[
  \neg \text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\neg F, T).
  \]
ASP Encoding of \( B \)

- **State consistency:**
  \[ -\text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\neg F, T). \]

- **Frame problem:**
  \[ \text{holds}(L, T + 1) : - \text{time}(T), \text{literal}(L), \text{holds}(L, T), \]
  \[ \text{complement}(L, L_1), \textbf{not} \text{ holds}(L_1, T + 1). \]
ASP Encoding of $B$

- **State consistency:**
  \[-\text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\neg(F), T).\]

- **Frame problem:**
  \[\text{holds}(L, T + 1) : - \text{time}(T), \text{literal}(L), \text{holds}(L, T), \text{complement}(L, L_1), \textbf{not} \text{ holds}(L_1, T + 1).\]

- **Initial state and goal:**
  \[\text{holds}(L, 0): - \text{initially}(L). \quad : - \text{goal}(L), \textbf{not} \text{ holds}(L, N).\]
ASP Encoding of $\mathcal{B}$

- **State consistency:**
  
  \[ \neg \text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\neg(F), T). \]

- **Frame problem:**

  \[ \text{holds}(L, T + 1) : \neg \text{time}(T), \text{literal}(L), \text{holds}(L, T), \]
  \[ \text{complement}(L, L_1), \textbf{not} \ \text{holds}(L_1, T + 1). \]

- **Initial state and goal:**

  \[ \text{holds}(L, 0): \neg \text{initially}(L). \]

  \[ : \neg \text{goal}(L), \textbf{not} \ \text{holds}(L, N). \]

- **One action per time:**

  \[ 1\{\text{occ}(A, T) : \text{action}(A)\}1: \neg \text{time}(T), T < N. \]

  \[ : \neg \text{action}(A), \text{time}(T), \text{occ}(A, T), \textbf{not} \ \text{possible}(A, T). \]
ASP Encoding of \( B \)

- **State consistency:**
  \[-\text{time}(T), \text{fluent}(F), \text{holds}(F, T), \text{holds}(\neg(F), T).\]

- **Frame problem:**
  \[
  \text{holds}(L, T + 1) : - \text{time}(T), \text{literal}(L), \text{holds}(L, T), \\
  \text{complement}(L, L_1), \textbf{not} \text{ holds}(L_1, T + 1).
  \]

- **Initial state and goal:**
  \[
  \text{holds}(L, 0): -\text{initially}(L). \\
  : -\text{goal}(L), \textbf{not} \text{ holds}(L, N).
  \]

- **One action per time:**
  \[
  1\{\text{occ}(A, T) : \text{action}(A)\}1: -\text{time}(T), T < N. \\
  : -\text{action}(A), \text{time}(T), \text{occ}(A, T), \textbf{not} \text{ possible}(A, T).
  \]

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

% 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 \( s_i \) (\( 0 \leq i \leq N \)):

- Every fluent \( F \) is represented by Boolean variable \( F^i \):
  \[ F^i \text{ is the value of fluent } F \text{ in state } s_i \]

- Every action \( A \) is represented as a Boolean variable \( A^i \) (\( i < N \)):
  \[ A^i = 1 \text{ iff action } A \text{ is executed in state } s_i \]
Main idea

\[ \sum_{j=1}^{p} A^i_j = 1 \]

Action \( a \) sets \( F \) to true (\( A^i = 1 \)), or no action that sets \( F \) to false is fired and \( F^i = 1 \).
Main idea

\[ \sum_{j=1}^{p} A^i_j = 1 \]

\( F^{i+1} = 1 \iff \) 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

$\varphi^i$: formula $\varphi$ projected to state $s_i$

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

- $\hat{\alpha}_j$ 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**

\[ A_j^i = 1 \quad \rightarrow \quad \bigvee_{j=1}^{q} \hat{r}_j^i \quad \text{Only executable actions can occur} \]

\[ \sum_{a_j \in A} A_j^i = 1 \quad \text{Single action at a time} \]
Some constraints

General Conditions

\[ A_j^i = 1 \rightarrow \bigvee_{j=1}^q \hat{r}_j^i \]
\[ \sum_{a_j \in A} A_j^i = 1 \]

Only executable actions can occur

Single action at a time

Positive Fluent \( F \)

\( \text{PosFired}_f^i = 1 \iff \text{PosDyn}_f^i = 1 \lor \text{PosStat}_{f+1}^i = 1 \)
\( \text{PosDyn}_f^i = 1 \iff \bigvee_{j=1}^m (\hat{a}_j^i \land A_{t_j}^i = 1) \)
\( \text{PosStat}_f^i = 1 \iff \bigvee_{j=1}^h \delta_j^i \)
Some constraints

**Negative Fluent \( F \)**

\[
\begin{align*}
\text{NegFired}_f^i &= 1 \iff \text{NegDyn}_f^i = 1 \lor \text{NegStat}^i_{f+1} = 1 \\
\text{NegDyn}_f^i &= 1 \iff \bigvee_{j=1}^p (\hat{\beta}_j^i \land A_{z_j}^i = 1) \\
\text{NegStat}_f^i &= 1 \iff \bigvee_{j=1}^k \hat{\gamma}_j^i
\end{align*}
\]
Some constraints

Negative Fluent $F$

\[
\begin{align*}
\text{NegFired}_f^i & = 1 \iff \text{NegDyn}_f^i = 1 \lor \text{NegStat}_f^{i+1} = 1 \\
\text{NegDyn}_f^i & = 1 \iff \bigvee_{j=1}^p (\hat{\beta}_j^i \land A_{z_j}^i = 1) \\
\text{NegStat}_f^i & = 1 \iff \bigvee_{j=1}^k \hat{\gamma}_j^i
\end{align*}
\]

Target for $F$

\[
\begin{align*}
\text{PosFired}_f^i & = 0 \lor \text{NegFired}_f^i = 0 \\
F^{i+1} & = 1 \iff \text{PosFired}_f^i = 1 \lor (\text{NegFired}_f^i = 0 \land F^i = 1)
\end{align*}
\]
Some constraints

Negative Fluent $F$

\[
\begin{align*}
\text{NegFired}_i^f &= 1 \iff \text{NegDyn}_i^f = 1 \lor \text{NegStat}^{i+1}_i^f = 1 \\
\text{NegDyn}_i^f &= 1 \iff \bigvee_{j=1}^p (\hat{\beta}_j^i \land A_{z_j}^i = 1) \\
\text{NegStat}_i^f &= 1 \iff \bigvee_{j=1}^k \gamma_j^i
\end{align*}
\]

Target for $F$

\[
\begin{align*}
\text{PosFired}_i^f &= 0 \lor \text{NegFired}_i^f = 0 \\
F^{i+1} &= 1 \iff \text{PosFired}_i^f = 1 \lor (\text{NegFired}_i^f = 0 \land F^i = 1)
\end{align*}
\]

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 $\sigma_u$
- Action occurrence $A$ in state $i$ represented by variable assignment $\sigma^i_A$

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 $C^s_F$.*

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

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

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

- Constraints:

$$
F^1 = 1 \iff F^0 = 1 \lor A^0 = 1 \\
G^1 = 1 \iff G^0 = 1 \lor H^1 = 1 \\
H^1 = 1 \iff H^0 = 1 \lor G^1 = 1 \\
F^i \in \{0, 1\} \land G^i \in \{0, 1\} \land H^i \in \{0, 1\}
$$

- 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

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

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

- Constraints:

$$F^1 = 1 \iff F^0 = 1 \lor A^0 = 1 \quad \text{G}$$
$$G^1 = 1 \iff G^0 = 1 \lor H^1 = 1 \quad \text{H}$$
$$H^1 = 1 \iff H^0 = 1 \lor G^1 = 1 \quad \text{H}$$

$$F^i \in \{0, 1\} \land G^i \in \{0, 1\} \land H^i \in \{0, 1\}$$

- 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. $[F^1 = 1, G^1 = 1, H^1 = 1]$. 
Some Theoretical Results

- Problem: cyclic dependencies generated by static causal laws
- Dependency Graph of static causal laws: caused([ℓ₁, ..., ℓₖ], ℓ) creates edges (ℓᵢ, ℓ).
- Construct a set of constraints CONS(ℓᵢ) to defeat self-sustaining loop: loop ℓ₁, ℓ₂, ..., ℓₘ
  - for each causes(aⱼ, ℓᵢ, α) add to constraint $A_j^u = 0$ or $F_ℓ^u = 0$ for some $ℓ ∈ α$
  - for each caused(γ, ℓᵢ) add to constraint $F_ℓ^u = 0$ for some $ℓ ∈ γ$
  - add to constraint $F_{ℓᵢ}^u = 0$ or $F_{ℓᵢ}^{u+1} = 0$
- Generate constraints $c_1 ∧ ⋯ ∧ c_m ⇒ F_{ℓ₁}^{u+1} = 0 ∧ ⋯ ∧ F_{ℓₘ}^{u+1} = 0$
The language $B^{MV}$

(Multi-)Fluents: introduced through domain declarations:
- $\text{fluent}(f, v_1, v_2)$, $\text{fluent}(f, \text{Set})$

Annotated Fluents: modeling backward references:
- $f^{-a}$ with $a \in \mathbb{N}$

Fluent Expressions:

$$FE ::= n \mid AF \mid \text{abs}(FE) \mid FE_1 \oplus FE_2 \mid \text{rei}(FC)$$

with $\oplus \in \{+,-,\times,/,\text{mod}\}$

Fluent Constraints:

$$FC ::= FE_1 \text{ op } FE_2$$

with $\text{op} \in \{=, \neq, \geq, \leq, <, >\}$
Action description specification

- **Dynamic causal laws**
  \[ \text{causes}(a, C_1, C) \]

- **Static causal laws**
  \[ \text{caused}(C, C_1) \]

- **Executability conditions**
  \[ \text{executable}(a, C) \]

where \( a \) is an action, \( C_1 \) 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
  \[
  \text{causes}(a, f = f^{-1} + 1, [])
  \]
  with $\text{dom}(f) = [1..100]$.
  - In $B$:
    \[
    \text{causes}(a, f_2, [f_1]). \ldots \text{causes}(a, f_{100}, [f_{99}])
    \]
  - Alternative encodings (e.g., bit-based) incur analogous problems.
- The CLP(FD) mapping requires minor changes.
The Languager $B^{MV}$

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

\[ F_f^i, F_f^{i+1} \quad \in \quad \text{dom}(f) \]
\[ A_a^i = 1 \quad \Rightarrow \quad \bigvee_{\text{executable}(a, \delta)} \delta^i \]
\[ A_a^i = 1 \land \alpha^i \quad \iff \quad \text{Dyn}_a^i = 1 \]
\[ \text{Dyn}_a^i = 1 \quad \Rightarrow \quad C_{i+1}^i \]
\[ \sum_{a_f \text{ affects } f} \text{Dyn}_{a_f}^i = 0 \quad \Rightarrow \quad F_f^i = F_f^{i+1} \]

- **Fluent domains**
- **Only executable actions**
- **a has f in consequence; causes(a, C, \alpha)**
- **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)

Barrels
Hanoi towers
Hydraulic planning
15-puzzle
Peg solitaire
Tangram
Reverse folding
Trucks
Gas diffusion

% Sicsplan
% ASP
% both
% none

100%
0%
percentage of instances
### Experimental comparison

<table>
<thead>
<tr>
<th>Instance</th>
<th>Length</th>
<th>ASP</th>
<th>CLP</th>
<th>$B^{MV}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrel-20-11-9</td>
<td>18</td>
<td>185+43.71</td>
<td>3+102</td>
<td>0.65</td>
</tr>
<tr>
<td>Barrel-20-11-9</td>
<td>21</td>
<td>189+4.39</td>
<td>2+80</td>
<td>0.53</td>
</tr>
<tr>
<td>Community $C_5$</td>
<td>6</td>
<td>MEM</td>
<td>1946</td>
<td>888</td>
</tr>
<tr>
<td>Community $D_5$</td>
<td>6</td>
<td>MEM</td>
<td>1519</td>
<td>1802</td>
</tr>
<tr>
<td>8-tile Puzzle $l_4$</td>
<td>25</td>
<td>55+437</td>
<td>2+79</td>
<td>57</td>
</tr>
<tr>
<td>Wolf-Goat-Cabbage</td>
<td>35</td>
<td>0.2+1.39</td>
<td>0.15+0.54</td>
<td>0.4</td>
</tr>
<tr>
<td>Wolf-Goat-Cabbage</td>
<td>36</td>
<td>0.2+4.24</td>
<td>0.13+1.87</td>
<td>1.9</td>
</tr>
</tbody>
</table>
\( \mathcal{B} \) best vs \( \mathcal{B}^{MV} \) (CLP)

<table>
<thead>
<tr>
<th>Problem</th>
<th>best-( \mathcal{B} )</th>
<th>( \mathcal{B}^{MV} )</th>
<th>both</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-puzzle</td>
<td>100%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barrels</td>
<td></td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse folding</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gas diffusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Son, Pontelli, Balduccini (NMSU & Drexel)
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

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 \((s_i: \text{state}, a_i: \text{action})\)

  \[ \alpha = s_1 a_1 s_2 a_2 \cdots a_n s_{n+1} \]

- 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
  - $\psi$: fluent formula $\psi$ should hold in the current state
  - $goal(\psi)$: fluent formula $\psi$ should hold in the final state

- Basic desires (Trajectory preference)
  - Temporal formulae over state preferences constructed using
    - propositional operators ($\land$, $\lor$, $\neg$)
    - temporal operators (next $\bigcirc$, eventually $\Diamond$, always $\Box$, until $\bigcup$)
Preference Relation over Basic Desires

- Given $\alpha = s_1a_1s_2a_2 \ldots a_ns_{n+1}$ and basic desire $\varphi$. We say $\alpha$ satisfies $\varphi$, denoted by $\alpha \models \varphi$, if
  - $\varphi = occ(a)$ and $a_1 = a$; or
  - $\varphi = \psi$ where $\psi$ is a fluent formula and $\psi$ holds in $s_1$; or
  - $\varphi = \text{goal}(\psi)$ and $\psi$ holds in $s_{n+1}$; or
  - $\varphi = \varphi_1 \land \varphi_2$ and $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$ (similarly for $\lor$, $\neg$); or
  - $\varphi = \text{next}(\varphi_1)$ and $\alpha[1] \models \varphi_1$ where $\alpha[1] = s_2a_2 \ldots a_ns_{n+1}$ (similarly for eventually, always, until)

- $\alpha$ preferred to $\beta$ with respect to $\varphi$ if $\alpha \models \varphi$ and $\beta \not\models \varphi$;

- $\alpha$ indistinguishable to $\beta$ with respect to $\varphi$ 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:** $\varphi_1 \prec \varphi_2 \prec \ldots \prec \varphi_n$ where each $\varphi_i$ is a basic desire;
- **General preference:** formulae over atomic preference, constructed using propositional ($\&, |, !, \prec$)

**Semantics:**

- Defining $\alpha \models \varphi$ where $\varphi = \varphi_1 \prec \varphi_2 \prec \ldots \prec \varphi_n, \varphi_1 \& \varphi_2, \varphi_1 | \varphi_2, or! \varphi_1$
- Most preferred trajectories with respect to $\prec$:
  - those satisfying $\varphi_1$,
  - if none exists then trajectories satisfying $\varphi_2$, 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 $\alpha = s_1a_1 \ldots a_ns_{n+1}$ which corresponds to an answer set of $P(A, I, G)$ (notation: $\alpha[t] = s_t\ a_{t+1}s_{n+1}$)
Implementation in Answer Set Planning

Idea
- Defining $satisfy(\varphi, T) \rightarrow \alpha[T]$ satisfies basic desire $\varphi$
- Associating weight to preference formulae such that if $\alpha$ is more preferred than $\beta$ with respect to $\varphi$ then $w(\alpha) < w(\beta)$
- Use ASP construct `minimize` to find most preferred trajectories

Realization by logic programming rules for
- defining $satisfy(\varphi, T)$
- computing an admissible weight function
Encoding a Preference Formula $\varphi$ by $\Pi_\varphi$

- Standard encoding of a fluent formula using ASP rules: each formula $\varphi$ is encoded by a set of rules $r_\varphi$
- Assigned a unique name $n_\varphi$

$\Pi_\varphi$: encoding of $\varphi$

- if $\varphi = \text{goal}(\varphi_1)$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{goal}(n_\varphi)\} \cup r_{\varphi_1}$; or
- if $\varphi = \text{occ}(a)$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{happen}(n_\varphi, a)\}$; or
- if $\varphi$ is a fluent formula $\varphi_1$ then $\Pi_\varphi = \{\text{desire}(n_\varphi)\} \cup r_{\varphi_1}$
- if $\varphi = \varphi_1 \land \varphi_2$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{and}(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\} \cup \Pi_{\varphi_1} \cup \Pi_{\varphi_2}$; or
- if $\varphi = \varphi_1 \lor \varphi_2$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{or}(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\} \cup \Pi_{\varphi_1} \cup \Pi_{\varphi_2}$; or
- if $\varphi = \neg \varphi_1$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{negation}(n_\varphi, n_{\varphi_1})\} \cup \Pi_{\varphi_1}$; or
- if $\varphi = \text{next}(\varphi_1)$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{next}(n_\varphi, n_{\varphi_1})\} \cup \Pi_{\varphi_1}$; or
- if $\varphi = \text{always}(\varphi_1)$ then $\Pi_\varphi = \{\text{desire}(n_\varphi), \text{always}(n_\varphi, n_{\varphi_1})\} \cup \Pi_{\varphi_1}$;
Rules for $\text{satify}(\varphi, T)$

\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{goal}(F), \text{satify}(F, \text{length}).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{happen}(F, A), \text{oc}(A, T).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{form}(F, G), h(G, T).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{and}(F, F1, F2),
\]
\[
\text{satify}(F1, T), \text{satify}(F2, T).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{negation}(F, F1), \text{notsatisfy}(F1, T).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{until}(F, F1, F2), \text{during}(F1, T, T1),
\]
\[
\text{satify}(F2, T1).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{always}(F, F1), \text{during}(F1, T, \text{length} + 1).
\]
\[
\text{satify}(F, T) \leftarrow \text{desire}(F), \text{next}(F, F1), \text{satify}(F1, T + 1).
\]
\[
\text{during}(F, T, T1) \leftarrow T < T1 - 1, \text{desire}(F), \text{satify}(F, T),
\]
\[
\text{during}(F, T + 1, T1).
\]
\[
\text{during}(F, T, T1) \leftarrow T = T1 - 1, \text{desire}(F), \text{satify}(F, T).
\]
Correctness of the Implementation

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

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

- Property of \(\Pi_{\text{pref}}\):
  - Each answer set of \(\Pi_{\text{pref}}\) corresponds to a most preferred trajectory with respect to \(\varphi\); and
  - Each preferred trajectory with respect to \(\varphi\) corresponds to an answer set of \(\Pi_{\text{pref}}\).
Implementation of General Preferences

Idea

- Develop an admissible weight function: $w_\varphi(\alpha)$ such that if $w_\varphi(\alpha) \geq w_\varphi(\beta)$ then $\alpha$ is more preferred than $\beta$ with respect to $\varphi$
- 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 $\varphi = \varphi_1 \vartriangleleft \varphi_2 \vartriangleleft \ldots \vartriangleleft \varphi_n$
  - $w_\varphi(\alpha) = 2^{n-1}w_{\varphi_1}(\alpha) + \ldots + 2^{0}w_{\varphi_n}(\alpha)$

- 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 $\varphi = \lnot \varphi_1$ 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.
  
  \[
  \text{\texttt{satisfy}(} \diamond f, T) \leftarrow \text{\texttt{holds}}(f, T_1), T_1 \leq n. \\
  \text{\texttt{not\_satisfy}(} \Box f, T) \leftarrow \text{\texttt{not\_holds}}(f, T_1), T_1 \leq n. \\
  \text{\texttt{satisfy}(} \Box f, n) \leftarrow \text{\texttt{not\_not\_holds}}(f, n). \\
  \]

- Add constraints that force the formula to be true:
  
  \[
  \leftarrow \text{\texttt{not\_satisfy}(} \diamond f, n). \\
  \]

  etc.
Planning with Incomplete Information
Conformant Planning and Complexity

Definition (Conformant Planning Problem)

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

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

Theorem (Complexity)

- **Conformant Planning**: $(D, \delta)$ is deterministic: $\Sigma_2^P$-hard even for plans of length 1, $\Sigma_2^P$-complete for polynomial-bounded length plans.
- **Conformant Planning**: $(D, \delta)$ is non-deterministic: $\Sigma_3^P$-hard even for plans of length 1, $\Sigma_3^P$-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. $\delta$ in $(D, \delta)$ 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**

<table>
<thead>
<tr>
<th>Belief states ($S$ and $\Sigma$)</th>
<th>Approximation states ($\delta$ and $\Delta$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$ a set of states</td>
<td>$\delta$ a set of fluent literals</td>
</tr>
<tr>
<td>$\Sigma$ a set of belief states</td>
<td>$\Delta$ a set of approximation states</td>
</tr>
</tbody>
</table>
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:

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

- 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 \text{caused}(l, [\varphi]), any set of fluent literals \delta satisfying \delta \models \varphi \text{ and } \delta \models \neg l \text{ is not a partial state.}
For an action theory \((D, \delta_0)\):

- Initial approximation state: \(\delta_0\) — a partial state
- Initial belief state:

\[
S_0 = \text{bef}(\delta_0)
\]

where

\[
\text{bef}(\delta) = \{ s \mid \delta \subseteq s, \ s \text{ is a state} \}
\]

- A fluent formula \(\varphi\) 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 \(\delta\) if \(l \in \delta\) \((\neg l \in \delta)\); unknown, otherwise. The truth value of a fluent formula \(\varphi\) is defined in the usual way.
Possible World Approach

- $S_0 = bef(\delta_0)$

\[
\Phi^c(a, S) = \begin{cases} 
\emptyset & \text{if } a \text{ is not executable in some } s \in S \\
\bigcup_{s \in S} \Phi(a, s) & \text{otherwise}
\end{cases}
\]

- $\Phi^c$ extended to $\hat{\Phi}^c$ in the usual way
- $(D, \delta_0) \models^P \varphi \textbf{ after } \alpha$ if $\varphi$ is true in the final belief state
- Size of search space: $n$ fluents $\rightarrow 2^n$ belief states
Planning Systems for Incomplete Domains

<table>
<thead>
<tr>
<th>Language</th>
<th>Sequential</th>
<th>Concurrent</th>
<th>Conformant</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLV$^C$</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MBP</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>CMBP</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SGP</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>POND</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>CFF</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>KACMBP</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

**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: 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: $\delta_0$
- 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 $\delta$
- $pe(a, \delta)$ is the set of “possible effects” of $a$ in $\delta$

- $(D, \delta_0) \models^0 \varphi$ after $\alpha$ if $\varphi$ is true in the final partial state
- $n$ fluents $\rightarrow 3^n$ partial states
- Incomplete
- No static causal laws

![Diagram](image-url)
0-Approximation Approach – Example

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

\[ \delta_0 = \emptyset \]

- \textit{dunk} is not executable in \( \delta_0 \)
- \textit{flush} is executable in \( \delta_0 \), \( \text{de}(\text{flush}, \delta_0) = \text{pe}(\text{flush}, \delta_0) = \{\neg \text{clogged}\} \)
- \( \Phi^0(\text{flush}, \delta_0) = \{\neg \text{clogged}\} \)

\[ \delta_1 = \{\neg \text{clogged}\} \]

- \textit{dunk}, \textit{flush} are executable in \( \delta_1 \)
- \( \text{de}(\text{dunk}, \delta_1) = \emptyset \) and \( \text{pe}(\text{dunk}, \delta_1) = \{\neg \text{armed}\} \)
- \( \Phi^0(\text{dunk}, \delta_1) = \{\text{clogged}\} \)
Dealing with Static Causal Laws

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

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

\[\delta_0 = \emptyset\]

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

Intuitive result

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

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 $\delta$ and

- either its negation cannot possibly hold in $\delta'$;
  \[ \Rightarrow \text{possible holds approximation} \]
- or it cannot possibly change in $\delta'$
  \[ \Rightarrow \text{possible change approximation} \]
$\Phi^{ph}$ 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., $l \notin \neg \text{Cl}_D(de(a, \delta))$)
- there is some static causal law whose body possibly holds (i.e., there exists some static causal law $l$ if $\varphi$ such that $\varphi$ possibly holds)
**$\Phi^{ph}$ Approximation – Definition**

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

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

\[ ph^0(a, \delta) = (pe(a, \delta) \cup \{ l \mid \neg l \not\in \delta \}) \setminus \neg E(a, \delta) \]

OBS: if $l$ if $\varphi$ in $D$ and $\varphi$ possibly holds then $l$ possibly holds.

\[ ph^{i+1}(a, \delta) = ph^i(a, \delta) \cup \left\{ l \: \exists [ l \text{ if } \psi ] \text{ in } D \text{ s.t. } l \not\in \neg E(a, \delta), \psi \subseteq ph^i(a, \delta), \neg \psi \cap E(a, \delta) = \emptyset \right\} \]

**Definition**

- if $a$ is not executable in $\delta$ then
  \[ \Phi^{ph}(a, \delta) = \emptyset \]

- otherwise,
  \[ \Phi^{ph}(a, \delta) = Cl_D(\{ l \mid l \not\in \neg ph(a, \delta) \}) \]
**\( \Phi^{ph} \) Approximation – Example**

\[
D_d = \begin{cases} 
  \text{down}(i + 1) & \text{if} \quad \text{down}(i) \\
  \text{touch}(i) \text{ causes } \text{down}(i) 
\end{cases}
\]

**Computation for \( \delta_0 = \emptyset \)**

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

A literal \(l\) possibly changes if

- it is not in \(\delta\)
- it is a possible effect \(a\) (i.e., there exists a dynamic law \(\text{causes}(a, l, [\varphi])\) and \(\varphi\) is not false in \(\delta\))
- it is a possibly indirect effect of \(a\) (i.e., there exists a static causal law \(\text{caused}(l, [\varphi])\) and \(\varphi\) possibly changes)
\( \Phi^{pc} \) Approximation

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

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

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

**Definition**

- If \( a \) is not executable in \( \delta \) then
  \[
  \Phi^{pc}(a, \delta) = \emptyset
  \]

- Otherwise,
  \[
  \Phi^{pc}(a, \delta) = Cl_D(E(a, \delta) \cup (\delta \setminus \neg pc(a, \delta)))
  \]
\( \Phi^{pc} \) Approximation – Example

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

Computation for \( \delta_0 = \emptyset \)
- \( de(touch(i), \delta_0) = \{ \text{down}(i) \} \) and \( pe(touch(i), \delta_0) = \{ \text{down}(i) \} \)
- \( E(touch(i), \delta_0) = \{ \text{down}(j) \mid i \leq j \leq n \} \)
- \( pc^0(touch(i), \delta_0) = \{ \text{down}(i) \} \)
- \( pc^1(touch(i), \delta_0) = \{ \text{down}(i), \text{down}(i + 1) \} \)
- \( pc(touch(i), \delta_0) = \{ \text{down}(j) \mid i \leq j \leq n \} \)
- \( \Phi^{pc}(touch(i), \delta_0) = \{ \text{down}(j) \mid i \leq j \leq n \} \)
Properties of $\Phi^{ph}$ and $\Phi^{pc}$ 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 $\Phi^{ph}$ and $\Phi^{pc}$ ⇒ 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 $\Phi^A$ is NP-complete
- Could improve the approximations
Reverting rules for computing effects of actions with the following:

- **Dynamic causal law:** `causes(A, L, ϕ)`
  
  \[\text{de}(L, T + 1) \leftarrow \text{occurs}(A, T), \text{holds}(ϕ, T)\]
  
  \[\text{ph}(L, T) \leftarrow \text{occurs}(A, T - 1), \text{not holds}(¬ϕ, T), \text{not de}(¬L, T + 1)\]

- **Static causal law:** `caused(L, ϕ)`
  
  \[\text{ph}(L, T) \leftarrow \text{ph}(ϕ, T)\]

- **Additional rule:**
  
  \[\text{ph}(L, T) \leftarrow \text{not holds}(¬L, T - 1), \text{not de}(¬L, T)\]

- **Inertial rule:**
  
  \[\text{holds}(L, T) \leftarrow \text{not ph}(¬L, T), \text{holds}(L, T - 1)\]
What is good about the approximation?

**Theorem (Complexity)**

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

**Consequence**

If \((D, \delta)\) 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 $\Phi^{ph}$ 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
**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 $B$ to handle sensing actions

- Allow knowledge-producing laws of the form
  \[
  \text{determines}(a, \theta)
  \]
  “if sensing action $a$ is executed, then the values of $l \in \theta$ will be known”

- New language is called $B_k$
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.

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

- No conformant plan for
  $$P_1 = (D_2, \{\neg\text{clogged}\}, \{\neg\text{armed}(1), \neg\text{armed}(2)\})$$
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

\[
\langle a, \text{cases} \left( \begin{array}{c}
l_1 \rightarrow \beta_1 \\
\ldots \\
l_n \rightarrow \beta_n
\end{array} \right) \rangle
\]

is a conditional plan
Example of Conditional Plan

\[ \langle \neg \text{ray}, \text{cases} \left( \begin{array}{l}
\text{armed}(1) \rightarrow \text{dunk}(1) \\
\text{armed}(2) \rightarrow \text{dunk}(2)
\end{array} \right) \rangle \]

is a solution of

\[ P_1 = (D_2, \neg \text{clogged}, \neg \text{armed}(1), \neg \text{armed}(2)) \]
How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: $\text{Actions} \times \text{Partial States} \rightarrow 2^{\text{Partial States}}$
How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: $\text{Actions} \times \text{Partial States} \rightarrow 2^{\text{Partial States}}$
  - For each $A \in \{\text{ph, pc}\}$, we define a transition function $\Phi^A_S$ as follows
    - for a non-sensing action $a$, $\Phi^A_S$ is the same as $\Phi^A$
    - for a sensing action $a$, each partial state in $\Phi^A_S$ corresponds to a literal that is sensed by $a$

- Result in four different approximations of $B_k$ domain descriptions

- Entailment $\models^A_S$

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

if $\varphi$ is true in every final partial state of the execution of $\alpha$
How to reason about sensing actions?

Must take into account different outcomes of sensing actions!

- Extending the function
  - Transition function: \( \text{Actions} \times \text{Partial States} \rightarrow 2^{\text{Partial States}} \)
  - For each \( A \in \{\text{ph}, \text{pc}\} \), we define a transition function \( \Phi^A_S \) as follows
    - for a non-sensing action \( a \), \( \Phi^A_S \) is the same as \( \Phi^A \)
    - for a sensing action \( a \), each partial state in \( \Phi^A_S \) corresponds to a literal that is sensed by \( a \)

- Result in four different approximations of \( B_k \) domain descriptions

- Entailment \( \models^A_S \)

\[
(D, \delta_0) \models^A_S \varphi \text{ after } \alpha
\]

if \( \varphi \) is true in every final partial state of the execution of \( \alpha \)

- Properties
  - \( \Phi^A_S \) 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: $\text{Actions} \times \text{Partial States} \rightarrow 2^{\text{Partial States}}$
  - For each $A \in \{ph, pc\}$, we define a transition function $\Phi^A_S$ as follows
    - for a non-sensing action $a$, $\Phi^A_S$ is the same as $\Phi^A$
    - for a sensing action $a$, each partial state in $\Phi^A_S$ corresponds to a literal that is sensed by $a$

- Result in four different approximations of $B_k$ domain descriptions

- Entailment $\models^A_S$

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

  if $\varphi$ is true in every final partial state of the execution of $\alpha$

- Properties
  - $\Phi^A_S$ can be computed in polynomial time
  - the polynomial-length conditional planning: NP-complete
\[ B_k \text{ Approximations} \]

**Definition**

- If \( a \) is not executable in \( \delta \) 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 \( a \) 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:** \( P = (D, \delta_0, G) \)
  A solution of \( P \) is a conditional plan \( \alpha \) such that
  \[(D, \delta_0) \models^P G \text{ after } \alpha\]

- **ASCP:**
  - Implemented in logic programming (Rules similar to approximation)
  - Approximation: \( \Phi^p_S \)
  - 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) ⇒ compact representation of state?
2. **Search**:
   1. possibility of infinitely many successor states
   2. concurrent actions

⇒ new type of heuristic?
Outline

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
### Scheduling

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

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

Goal: find a schedule to complete the $t_1, \ldots, t_n$ with minimal span (total time).
A Schedule for a Set of Tasks: Definition

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

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

Given three tasks $t_1, t_2, t_3$ with $\text{duration}(t_1, 3)$, $\text{duration}(t_2, 3)$, $\text{duration}(t_3, 2)$, and the constraints $\text{prec}(t_1, t_3)$, $\text{non\_overlap}(t_1, t_2)$ 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 \( \mathbb{N} \), denoted by \( \text{start} : T \rightarrow \mathbb{N} \), such that

- if \( \text{prec}(t_i, t_j) \) is true then \( \text{start}(t_i) + d_i \leq \text{start}(t_j) \) (\( t_i \) completed before \( t_j \))
- if \( \text{non_overlap}(t_i, t_j) \) is true then \( \text{start}(t_i) + d_i \leq \text{start}(t_j) \) or \( \text{start}(t_j) + d_j \leq \text{start}(t_i) \)

Given three tasks \( t_1, t_2, t_3 \) with \( \text{duration}(t_1, 3), \text{duration}(t_2, 3), \text{duration}(t_3, 2) \), and the constraints \( \text{prec}(t_1, t_3), \text{non_overlap}(t_1, t_2) \) 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 $\mathbb{N}$, denoted by $\text{start} : T \longrightarrow \mathbb{N}$, such that

- if $\text{prec}(t_i, t_j)$ is true then $\text{start}(t_i) + d_i \leq \text{start}(t_j)$ ($t_i$ completed before $t_j$)
- if $\text{non_overlap}(t_i, t_j)$ is true then $\text{start}(t_i) + d_i \leq \text{start}(t_j)$ or $\text{start}(t_j) + d_j \leq \text{start}(t_i)$

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

Input: assume that the problem is given ...

\[ \text{task}(t_1), \ldots, \text{task}(t_n), \text{duration}(t_1, d_1), \ldots, \text{duration}(t_n, d_n) \]
\[ \text{prec}(t_i, t_j), \ldots, \text{non_overlap}(t_i, t_j), \ldots, \]

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

\[
\begin{align*}
\text{\% generating start time} \\
1 \{ \ starts(T, S) : \ time(S) \} 1 :- \ task(T).
\end{align*}
\]
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:** \( task(t_1), \ldots, \text{duration}(t_1, d_1), \ldots, \text{prec}(t_i, t_j), \ldots, \text{non_overlap}(t_i, t_j), \ldots, \)

**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 \geq S1.

:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T2, D2),
   S1 < S2+D2, S1 \geq 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 $\mu$: increasing from 100 to 200
- Task duration randomly generated between $\frac{\mu}{2}$ and $\mu$
- $prec(t_i, t_j)$: set for 30% of $\langle t_i, t_j \rangle$ pairs, randomly selected
- Simplifications: (1) no overlap constraints; (2) no span minimization

Execution times:

<table>
<thead>
<tr>
<th>Maximum task duration</th>
<th>100</th>
<th>110</th>
<th>120</th>
<th>130</th>
<th>140</th>
<th>150</th>
<th>160</th>
<th>170</th>
<th>180</th>
<th>190</th>
<th>200</th>
</tr>
</thead>
</table>

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 ...

\[\text{task}(t_1), \ldots, \text{task}(t_n), \text{duration}(t_1, d_1), \ldots, \text{duration}(t_n, d_n)\]
\[\text{prec}(t_i, t_j), \ldots, \text{non_overlap}(t_i, t_j), \ldots,\]

Code

\[
\begin{align*}
\% \text{ generating start time} \\
\text{var}(\text{st}(T),0,\text{length}) \leftarrow \text{task}(T). \\
\text{required}(\text{cumulative}([\text{st}/1],[\text{duration}/2])).
\end{align*}
\]

\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\% List notation
\%
\%
\% [var/k] expands to [var(term1,...,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(term1,...,term_k)
\% e.g., [duration/2] is expanded to [d_1, d_2, ..., d_n]
\%
\% result: \text{required}(\text{cumulative}([\text{st}(t_1), \text{st}(t_2), \ldots, \text{st}(t_n)],[d_1, d_2, \ldots, d_n])).
CASP Encoding for Scheduling

**Input:** assume that the problem is given ...

\[\text{task}(t_1), \ldots, \text{task}(t_n), \text{duration}(t_1, d_1), \ldots, \text{duration}(t_n, d_n)\]

\[\text{prec}(t_i, t_j), \ldots, \text{non_overlap}(t_i, t_j), \ldots,\]

**Code**

\[
\text{% generating start time}
\text{var(st(T),0,length) :- task(T).}
\text{required(cumulative([st/1],[duration/2])).}
\text{% checking prec}
\text{required(st(T2) \geq st(T1)+D1) :- prec(T1,T2), duration(T1,D1).}
\]
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
\[\text{var(st(T),0,length) :- task(T).}\]
\[\text{required(cumulative([st/1],[duration/2])).}\]
\% checking prec
\[\text{required(st(T2) \geq st(T1)+D1) :- prec(T1,T2), duration(T1,D1).}\]
\% non-overlap
\[\text{required(st(T2) \geq st(T1)+D1 \lor st(T1) \geq st(T2)+D2) :-}
\]
\[\text{non\_overlap(T1,T2), duration(T1, D1), duration(T2, D2).}\]
Scalability: ASP vs CASP

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

Execution times:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>100</th>
<th>110</th>
<th>120</th>
<th>130</th>
<th>140</th>
<th>150</th>
<th>160</th>
<th>170</th>
<th>180</th>
<th>190</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASP</td>
<td>0.039</td>
<td>0.041</td>
<td>0.040</td>
<td>0.042</td>
<td>0.041</td>
<td>0.040</td>
<td>0.048</td>
<td>0.050</td>
<td>0.042</td>
<td>0.060</td>
<td>0.045</td>
</tr>
</tbody>
</table>

*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
\[ \text{var}(\text{st}(D,J),0,\text{MT}) \] :- \text{job}(J), \text{job\_device}(J,D), \text{max\_time}(	ext{MT}). \]

% assign start times; only up to N overlapping jobs for a device with N instances
\[ \text{required}(\text{cumulative}([\text{st}(D)/2],[\text{len\_by\_dev}(D)/3],N)) \] :- \text{n\_instances}(D,N)

% length of a job is the same on any suitable device
\[ \text{len\_by\_dev}(D,J,L) \] :- \text{job}(J), \text{job\_device}(J,D), \text{job\_len}(J,L). \]
Print Shop Scheduling Encoding

% Start time of job J on device type D
\( \text{var(st(D,J),0,MT)} \) :- \( \text{job(J), job_device(J,D), max_time(MT)} \).

% assign start times; only up to N overlapping jobs for a device with N instances
\( \text{required(cumulative([st(D)/2],[len_by_dev(D)/3],N))} \) :- \( \text{n_instances(D,N)} \)

% length of a job is the same on any suitable device
\( \text{len_by_dev(D,J,L)} \) :- \( \text{job(J), job_device(J,D), job_len(J,L)} \).

% checking prec
\( \text{required(st(D2,J2) \geq st(D1,J1)+Len1)} \) :-
  \( \text{job_device(J1,D1), job_device(J2,D2), prec(J1,J2), job_len(J1,Len1)} \).
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:
- \texttt{curr\_start}(J,T): current schedule
- \texttt{curr\_device}(J,D): current job-device assignment
- \texttt{curr\_time}(CT): current (wall-clock) time

% identify jobs that have already started
\texttt{already\_started}(J) :- \texttt{curr\_start}(J,T), \texttt{curr\_time}(CT), CT > T.

% determine which jobs must not be affected
\texttt{must\_not\_schedule}(J) :-
\quad \texttt{already\_started}(J), \texttt{not ab}(\texttt{must\_not\_schedule}(J)).
Introducing Incremental Updates

Input:
\[
\text{curr\_start}(J,T): \text{current schedule} \\
\text{curr\_device}(J,D): \text{current job-device assignment} \\
\text{curr\_time}(CT): \text{current (wall-clock) time}
\]

% identify jobs that have already started
\[
\text{already\_started}(J) :- \text{curr\_start}(J,T), \text{curr\_time}(CT), CT > T.
\]

% determine which jobs must not be affected
\[
\text{must\_not\_schedule}(J) :- \\
\phantom{\text{must\_not\_schedule}(J)} \text{already\_started}(J), \text{not ab}(\text{must\_not\_schedule}(J)).
\]

% the start time of those jobs remains the same
\[
\text{required}(st(D,J)=T) :- \\
\phantom{\text{required}(st(D,J)=T)} \text{curr\_device}(J,D), \text{curr\_start}(J,T), \text{must\_not\_schedule}(J).
\]
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)\geq 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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
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**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>G2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>G1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>G3</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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.)
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

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*

- 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)

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 $\text{WCD}$?
- Reducing $\text{WCD}$: given an integer $k$, identify a set of at most $k$ actions so that the $\text{WCD}$ 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 $\text{WCD}$ and reducing $\text{WCD}$
- Our approach: declarative language
  - represent both problems as logic programs
  - use state-of-the-art off-the-shelf answer set solver ($\text{Clasp}$)
  - two different encodings - perform significantly better
**General Idea**

- **Computing $\text{WCD}$**: 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 $\text{WCD}$**:
  - Guess a set of actions (that will be removed) by using choice atoms: $\{\text{blocked}(A) : \text{action}(A)\}^k$.
  - Calculate $\text{WCD}$ for **new** problem.
  - Identify the **best** answer (with smallest $\text{WCD}$).
Computing $\text{WCD}$: Saturation Based Meta Encoding

- Saturation based method in ASP: for solving QBF-problems
- Computing $\text{WCD}$ can be represented as a QBF problem

$\text{WCD}$
maximal number of actions before revealing the goal same as longest common prefix between optimal plans of two different goals

$\text{WCD}$ as QBF

$\nu l(x, y, c)$: $c$ is common prefix of two minimal cost plans $\pi^*_x$ and $\pi^*_y$ ($\pi^*_g$ is an optimal plan for $g$).

$$
\exists x, y, c [\nu l(x, y, c) \land [\forall x', y', c' [\nu l(x', y', c') \rightarrow |c| \geq |c'|]]]
$$

$x, y, x', y' \in G$
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 ($k = 1$)

<table>
<thead>
<tr>
<th>Domain</th>
<th>Instances</th>
<th>WCD reduction</th>
<th>PR</th>
<th>Runtime (s)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>grid-navigation</td>
<td>5-14</td>
<td>9 $\rightarrow$ 9</td>
<td>12</td>
<td>26</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19-10</td>
<td>17 $\rightarrow$ 17</td>
<td>12</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20-9</td>
<td>39 $\rightarrow$ 39</td>
<td>23</td>
<td>406</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16-11</td>
<td>4 $\rightarrow$ 4</td>
<td>11</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16-11</td>
<td>4 $\rightarrow$ 4</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ipc-grid+</td>
<td>5-5-5</td>
<td>4 $\rightarrow$ 3</td>
<td>14</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5-10-10</td>
<td>11 $\rightarrow$ 11</td>
<td>194</td>
<td>475</td>
<td>14</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10-5-5</td>
<td>12 $\rightarrow$ 10</td>
<td>46</td>
<td>36</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10-10-10</td>
<td>19 $\rightarrow$ 19</td>
<td>2,661</td>
<td>1,257</td>
<td>33</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>block-words</td>
<td>8-20</td>
<td>10 $\rightarrow$ 10</td>
<td>946</td>
<td>timeout</td>
<td>64</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8-20</td>
<td>14 $\rightarrow$ 14</td>
<td>809</td>
<td>timeout</td>
<td>121</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>logistics</td>
<td>1-2-6-2-2-6</td>
<td>18 $\rightarrow$ 18</td>
<td>3,506</td>
<td>timeout</td>
<td>151</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1-2-6-2-2-6</td>
<td>18 $\rightarrow$ 18</td>
<td>2,499</td>
<td>timeout</td>
<td>135</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2-2-6-2-2-6</td>
<td>17 $\rightarrow$ 17</td>
<td>3,173</td>
<td>timeout</td>
<td>352</td>
<td>756</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2-2-6-2-4-6</td>
<td>17 $\rightarrow$ 17</td>
<td>timeout</td>
<td>timeout</td>
<td>5,377</td>
<td>1,943</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2-2-6-2-6-6</td>
<td>16 $\rightarrow$ 16</td>
<td>timeout</td>
<td>timeout</td>
<td>5,166</td>
<td>timeout</td>
<td></td>
</tr>
</tbody>
</table>
## Experimental Results II ($k = 2$)

### (b) $k = 2$

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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
Overall Goal

- 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

Movement constraints

![Diagram showing movement constraints](image)

(7 by 4 with block of 10)

**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:
  - $\text{holds}(at(L), T)$ becomes $\text{holds}(at(R, L), T)$:
    "Agent $R$ is at $L$ at time step $T$.”
  - $\text{occ}(A, T)$ becomes $\text{occ}(R, A, T)$:
    "Agent $R$ executes $A$ at time step $T$.”

- Adding rules to deal with constraints:
  - Enforcing movement constraints
    $\text{enforcing movement constraints}:\quad \text{occ}(R1, \text{move}(L), T), \text{occ}(R2, \text{move}(L1), T), \text{edge}(L, L1), \text{holds}(at(R1, L1), T), \text{holds}(at(R2, L), T)$.
  - 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, \ldots, 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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
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 $\langle X, D, F, A, \alpha \rangle$

- $X = \{x_1, \ldots, x_n\}$ is a finite set of (decision) variables;
- $D = \{D_1, \ldots, D_n\}$ is a set of finite domains, where $D_i$ is the domain of the variable $x_i \in X$, for $1 \leq i \leq n$;
- $F = \{f_1, \ldots, f_m\}$ 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 $scp(f_j) = \{x_{j_1}, \ldots, x_{j_{k_j}}\}$;
- $A = \{a_1, \ldots, a_p\}$ is a finite set of agents; and
- $\alpha : X \mapsto A$ maps each variable to an agent.

---

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 \( f(x) \in D_x \)
- **Assignment:** \( f : X \mapsto \bigcup_{D \in D} D \) such that \( f(x) \in D_x \)
- **C(M)** set of all assignments
- **Optimal Solution:**
  \[ x = \arg\max_{x \in C(M)} \sum_{j=1}^{m} f_j(x_{f_j}) \]
- **Graph Representation:** \( G_M = (V, E) \) where \( V = A \) and
  \[ E = \{ \{a, a'\} \mid \{a, a'\} \subseteq A, \exists f \in F \text{ such that } \{x_i, x_j\} \subseteq \text{scp}(f), \text{ and } \alpha(x_i) = a, \alpha(x_j) = a' \} \]
- **Pseudo-tree:** subgraph of \( G_M \) with all nodes of \( G_M \) and (i) the included edges form a tree, and (ii) two nodes connected in \( G_M \) are in the same branch of the tree.
DCOP

\[
\begin{array}{ccc}
\text{Utilities} & x_i & x_j \\
5 & 0 & 0 \\
8 & 0 & 1 \\
20 & 1 & 0 \\
3 & 1 & 1 \\
\end{array}
\]

for \( i < j \)
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**

- $\text{UTIL}_{a_j}^{a_i}$ message from $a_i$ to $a_j$; optimal utility for each combination of values to variables in $\text{sep}_i$

- $U = \text{UTIL}_{a_i}^{a_i} \oplus \text{UTIL}_{a_k}^{a_i}$ is the join of two UTIL matrices.

$$\text{scp}(U) = \text{scp}(\text{UTIL}_{a_k}^{a_i}) \cup \text{scp}(\text{UTIL}_{a_i}^{a_i})$$

For each possible combination $x$ of values of variables in $\text{scp}(U)$,

$$U(x) = \text{UTIL}_{a_k}^{a_i}(x_{\text{UTIL}_{a_k}^{a_i}}) + \text{UTIL}_{a_i}^{a_i}(x_{\text{UTIL}_{a_i}^{a_i}}),$$

- Let $\alpha_i \subseteq \text{scp}(\text{JOIN}_{a_i}^{P_i})$. Let $X_i$ be the set of all possible value combinations of variables in $\alpha_i$.

$U = \text{JOIN}_{a_i}^{P_i} \perp_{\alpha_i}$ is defined as:

1. $\text{scp}(U) = \text{scp}(\text{JOIN}_{a_i}^{P_i}) \setminus \alpha_i$
2. for each possible value combination $x$ of variables in $\text{scp}(U)$,

$$U(x) = \max_{x' \in X_i} \text{JOIN}_{a_i}^{P_i}(x, x').$$
Each agent $a_i$ does:

begin

$JOIN_{a_i}^P = \text{null}$

foreach $a_c \in C_i$ do

wait for $UTIL_{a_c}^a$ ; /* message to arrive from $a_c$ */

$JOIN_{a_i}^P = JOIN_{a_i}^P \oplus UTIL_{a_c}^a$ ; /* join $UTIL$ from children */

/* join constraints with parent/pseudo-parents */

$JOIN_{a_i}^P = JOIN_{a_i}^P \oplus (\oplus f \in R_{a_i} f)$

/* projection to eliminate owned variables */

$UTIL_{a_i}^P = JOIN_{a_i}^P \perp \alpha_i$

Send $UTIL_{a_i}^P$ message to its parent agent $P_i$

end
Distributed Constraint Optimization Problems

Distributed Pseudo-Tree Optimization Procedure (DPOP)

In $x_3$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$5+5=10$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$5+20=25$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$20+5=25$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$20+20=40$</td>
</tr>
</tbody>
</table>

In $x_2$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$5+16=21$</td>
</tr>
<tr>
<td>0</td>
<td>$5+16=21$</td>
</tr>
<tr>
<td>1</td>
<td>$20+25=45$</td>
</tr>
</tbody>
</table>
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 $VALUE^{a_i}_{P_i}(sep_{i}^*)$ message from parent $P_i$

/* Determine optimal value for variables of $a_i$ */

$\alpha_{i}^* \leftarrow \text{argmax}_{\alpha_{i} \in X_{i}} \text{JOIN}^{P_i}_{a_i}(sep_{i}^*, \alpha_{i})$

foreach $a_c \in C_i$ do

let $sep_{i}^{**}$ be partial optimal value assignment for variables in $sep_c$

from $sep_{i}^*$

send $VALUE(sep_{i}^{**}, \alpha_{i}^*)$ as $VALUE^{a_c}_{a_i}$ message to its child agent $a_c$

end
Distributed Pseudo-Tree Optimization Procedure (DPOP)

In $x_3$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5+5=10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5+20=25</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>20+5=25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20+20=40</td>
</tr>
</tbody>
</table>

In $x_2$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5+16=21</td>
</tr>
<tr>
<td>0</td>
<td>8+25=23</td>
</tr>
<tr>
<td>1</td>
<td>20+25=45</td>
</tr>
<tr>
<td>3+40=43</td>
<td></td>
</tr>
</tbody>
</table>
Agent Architecture

1. **Specification Module (ASP)**
2. **Controller Module (Prolog)**
For $x_j \in X$ with $D_j = \{\ell, \ldots, u\}$:

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

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

- $constraint(f)$.
- $scope(f, x_1)$.
- $scope(f, x_k)$.
- $f(u, v_1, \ldots, v_k)$. 
Distributed Constraint Optimization Problems

ASP-DPOP

SM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(U_{1,2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ x_1 \text{ cons } x_2(0, 0, 0). \]
\[ x_1 \text{ cons } x_2(1, 0, 1). \]
\[ x_1 \text{ cons } x_2(1, 1, 0). \]
\[ x_1 \text{ cons } x_2(2, 1, 1). \]
### SM

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$U_{1,2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

$x_{1\_cons\_x2}(0, 0, 0)$.
$x_{1\_cons\_x2}(1, 0, 1)$.
$x_{1\_cons\_x2}(1, 1, 0)$.
$x_{1\_cons\_x2}(2, 1, 1)$.

$x_{1\_cons\_x2}(U_1 + U_2, U_1, U_2) : \neg value(x_1, U_1), value(x_2, U_2)$. 
\[ x_1 \text{ \_cons\_} x_2(0, 0, 0). \]
\[ x_1 \text{ \_cons\_} x_2(1, 0, 1). \]
\[ x_1 \text{ \_cons\_} x_2(1, 1, 0). \]
\[ x_1 \text{ \_cons\_} x_2(2, 1, 1). \]

\[ x_1 \text{ \_cons\_} x_2(U_1 + U_2, U_1, U_2) : - \text{value}(x_1, U_1), \text{value}(x_2, U_2). \]
Agent $a_i$ information:

- Identification:
  \[ \text{agent}(a_i). \]

- For each $x \in X$ such that $\alpha(x) = a_i$:
  \[ \text{owner}(a_i, x). \]

- For each neighbor agent $a_j$:
  \[ \text{neighbor}(a_j). \]

- For each $x'$ owned by a neighbor agent $a_j$:
  \[ \text{owner}(a_j, x'). \]
\( UTIL_{P_i} \) with \( sep_i = \langle x_1, \ldots, x_k \rangle \):

\[
\text{table}_\text{info}(a_i, a_{i_1}, x_1, \ell_1, u_1).
\]

\[
\text{...}
\]

\[
\text{table}_\text{info}(a_i, a_{i_k}, x_k, \ell_k, u_k).
\]

\[
\text{table}_\text{max}_a(i)(u, v_1, \ldots, v_k).
\]

\( VALUE_{P_i} \):

\[
\text{solution}(a, x, v).
\]
Distributed Constraint Optimization Problems

ASP-DPOP

To agent $a_1$

From agent $a_1$

From agent $a_3$

To agent $a_3$

Solution

$I'_{a_2}$

Son, Pontelli, Balduccini (NMSU & Drexel)  LP: Foundations & Applications ICAPS 2017 227 / 241
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).
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-Constrained 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

<table>
<thead>
<tr>
<th>X</th>
<th>DPOP</th>
<th></th>
<th>H-DPOP</th>
<th></th>
<th>PH-DPOP</th>
<th></th>
<th>AFB</th>
<th></th>
<th>ASP-DPOP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
</tr>
<tr>
<td>15</td>
<td>86%</td>
<td>39,701</td>
<td>100%</td>
<td>148</td>
<td>98%</td>
<td>67,161</td>
<td>100%</td>
<td>53</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>188</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>73</td>
<td>100%</td>
</tr>
<tr>
<td>25</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>295</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>119</td>
<td>100%</td>
</tr>
<tr>
<td>150</td>
<td>0%</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>31,156</td>
<td>100%</td>
</tr>
<tr>
<td>200</td>
<td>0%</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>117,913</td>
<td>100%</td>
</tr>
<tr>
<td>250</td>
<td>0%</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>100%</td>
<td>-</td>
<td>100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tight</th>
<th>DPOP</th>
<th></th>
<th>H-DPOP</th>
<th></th>
<th>PH-DPOP</th>
<th></th>
<th>AFB</th>
<th></th>
<th>ASP-DPOP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
</tr>
<tr>
<td>0.5</td>
<td>94%</td>
<td>38,043</td>
<td>100%</td>
<td>161</td>
<td>96%</td>
<td>71,181</td>
<td>100%</td>
<td>57</td>
<td>92%</td>
</tr>
<tr>
<td>0.6</td>
<td>90%</td>
<td>31,513</td>
<td>100%</td>
<td>144</td>
<td>98%</td>
<td>68,307</td>
<td>100%</td>
<td>52</td>
<td>100%</td>
</tr>
<tr>
<td>0.7</td>
<td>90%</td>
<td>39,352</td>
<td>100%</td>
<td>128</td>
<td>100%</td>
<td>49,377</td>
<td>100%</td>
<td>48</td>
<td>100%</td>
</tr>
<tr>
<td>0.8</td>
<td>92%</td>
<td>40,526</td>
<td>100%</td>
<td>112</td>
<td>100%</td>
<td>62,651</td>
<td>100%</td>
<td>57</td>
<td>100%</td>
</tr>
</tbody>
</table>
Power Network

13-Bus Configuration

<table>
<thead>
<tr>
<th>Domain Size</th>
<th>Simulated Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>DPOP 3,125</td>
</tr>
<tr>
<td>7</td>
<td>ASP-DPOP (facts) 10</td>
</tr>
<tr>
<td>9</td>
<td>ASP-DPOP (rules) 14</td>
</tr>
<tr>
<td>11</td>
<td>H-DPOP 18</td>
</tr>
<tr>
<td>13</td>
<td>ODPOP 22</td>
</tr>
</tbody>
</table>

37-Bus Configuration

<table>
<thead>
<tr>
<th>Domain Size</th>
<th>Simulated Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>DPOP 6,742</td>
</tr>
<tr>
<td>7</td>
<td>ASP-DPOP (facts) 30,604</td>
</tr>
<tr>
<td>9</td>
<td>ASP-DPOP (rules) 59,049</td>
</tr>
<tr>
<td>11</td>
<td>H-DPOP 161,051</td>
</tr>
<tr>
<td>13</td>
<td>ODPOP 18</td>
</tr>
</tbody>
</table>

| | | | | | | | | |
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

1. Answer Set Programming
2. Constraint Logic Programming
3. Constraint Answer Set Programming
4. Action Description Languages
5. Answer Set Planning and CLP Planning
6. Scheduling
7. Goal Recognition Design
8. Generalized Target Assignment and Path Finding
9. Distributed Constraint Optimization Problems
10. Conclusions
Summary

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


References II


