Modeling and Solving in Answer Set Programming

Martin Gebser Roland Kaminski Benjamin Kaufmann
Torsten Schaub

University of Potsdam

Outline

1. Motivation
2. Introduction
3. Basic modeling
4. (Language extensions)
5. Solving
6. Advanced modeling
7. Systems
8. Summary

Bibliography

Bias

Focus

- Answer Set Programming as Boolean Constraint Satisfaction Problem
- Answer Set Solving as a Boolean Constraint Solving
- Answer Set Systems at `http://potassco.sourceforge.net`

Further resources

`http://potassco.sourceforge.net/teaching.html`

Bias

Focus

- Answer Set Programming as Boolean Constraint Satisfaction Problem
- Answer Set Solving as a Boolean Constraint Solving
- Answer Set Systems at `http://potassco.sourceforge.net`

Further resources

`http://potassco.sourceforge.net/teaching.html`

Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

4 Rooting ASP

5 Problem solving

6 Use

Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

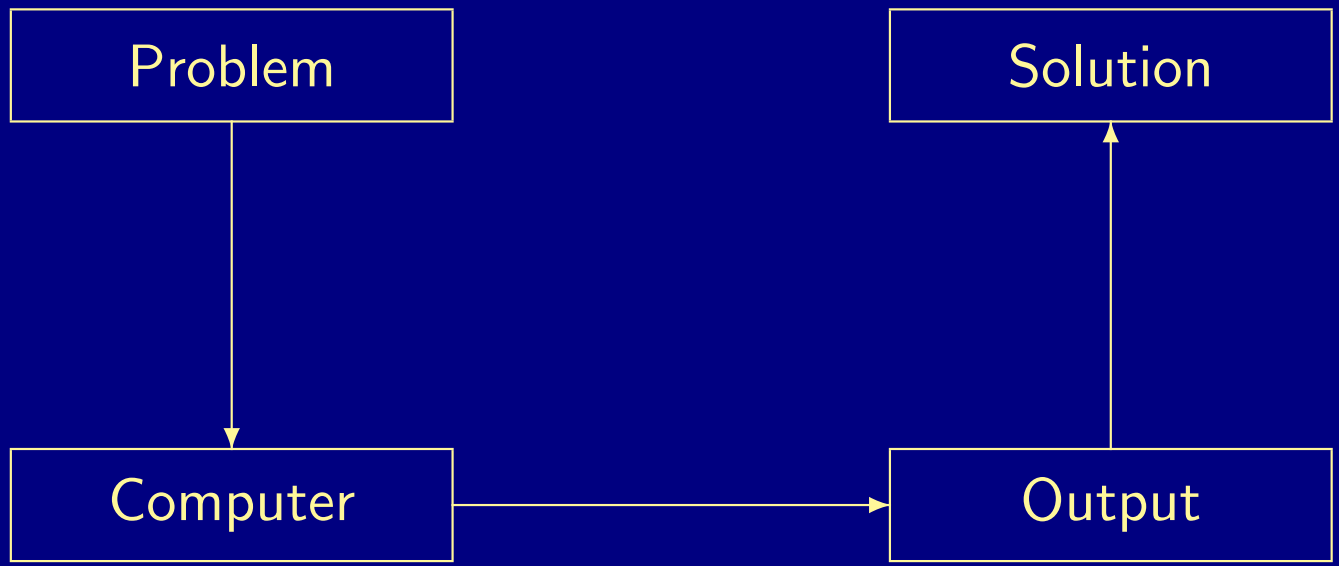
4 Rooting ASP

5 Problem solving

6 Use

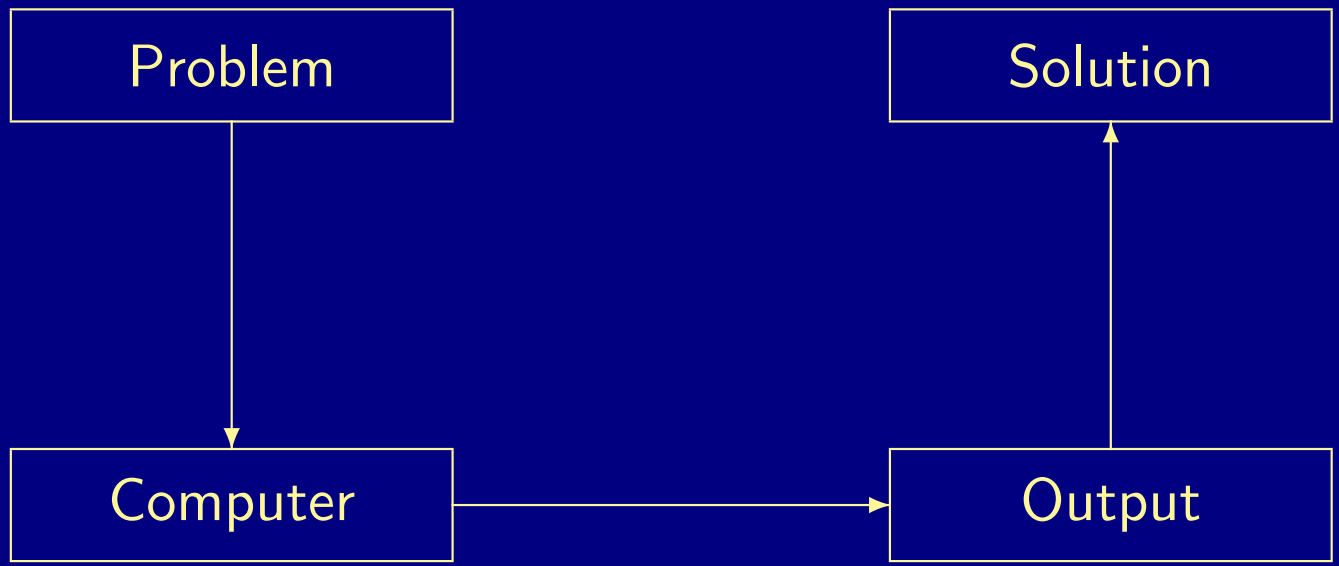
Informatics

"What is the problem?" versus *"How to solve the problem?"*



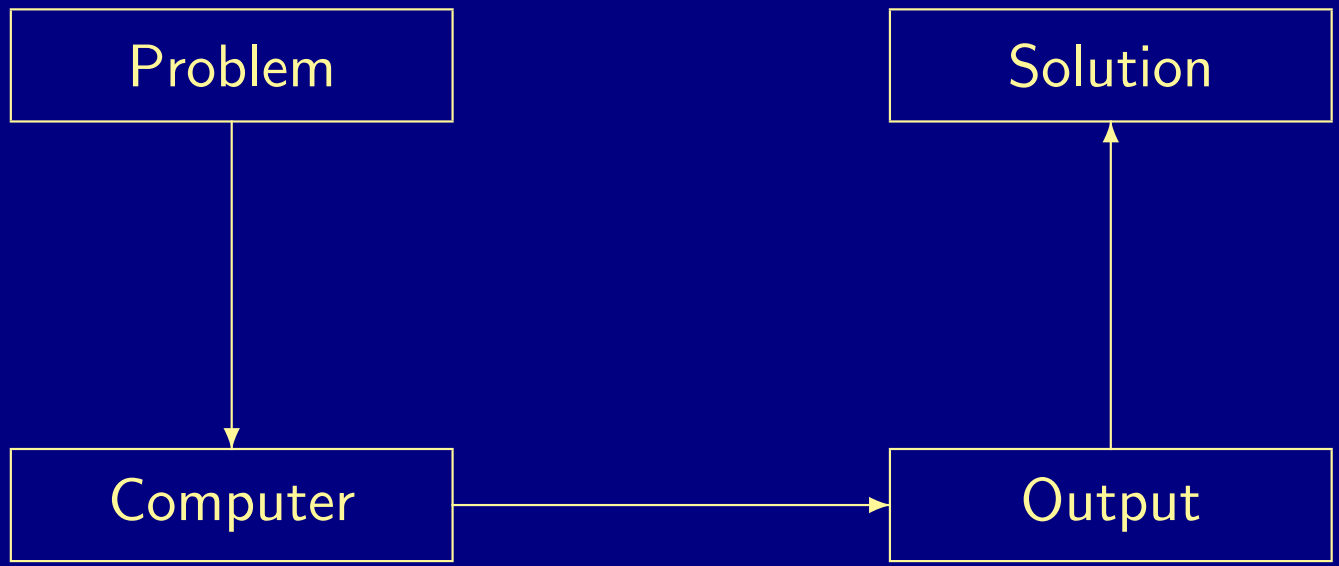
Informatics

"What is the problem?" versus *"How to solve the problem?"*



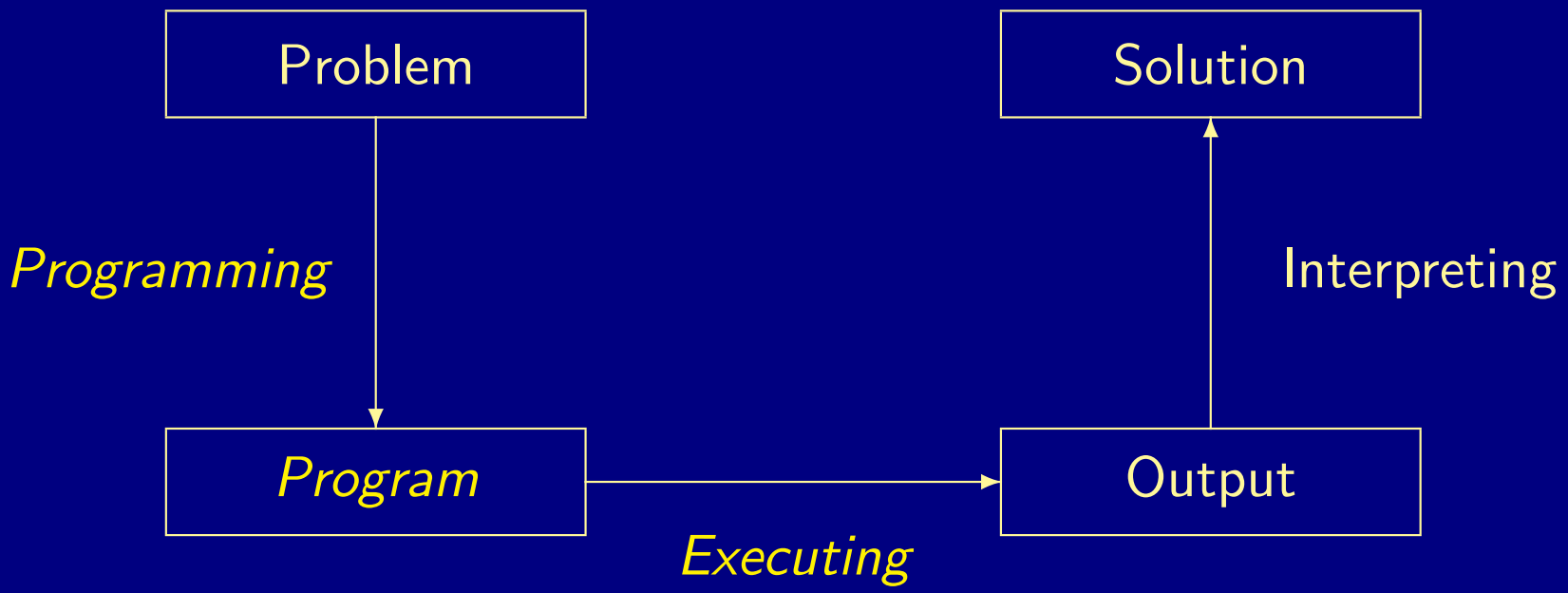
Traditional programming

"What is the problem?" versus *"How to solve the problem?"*



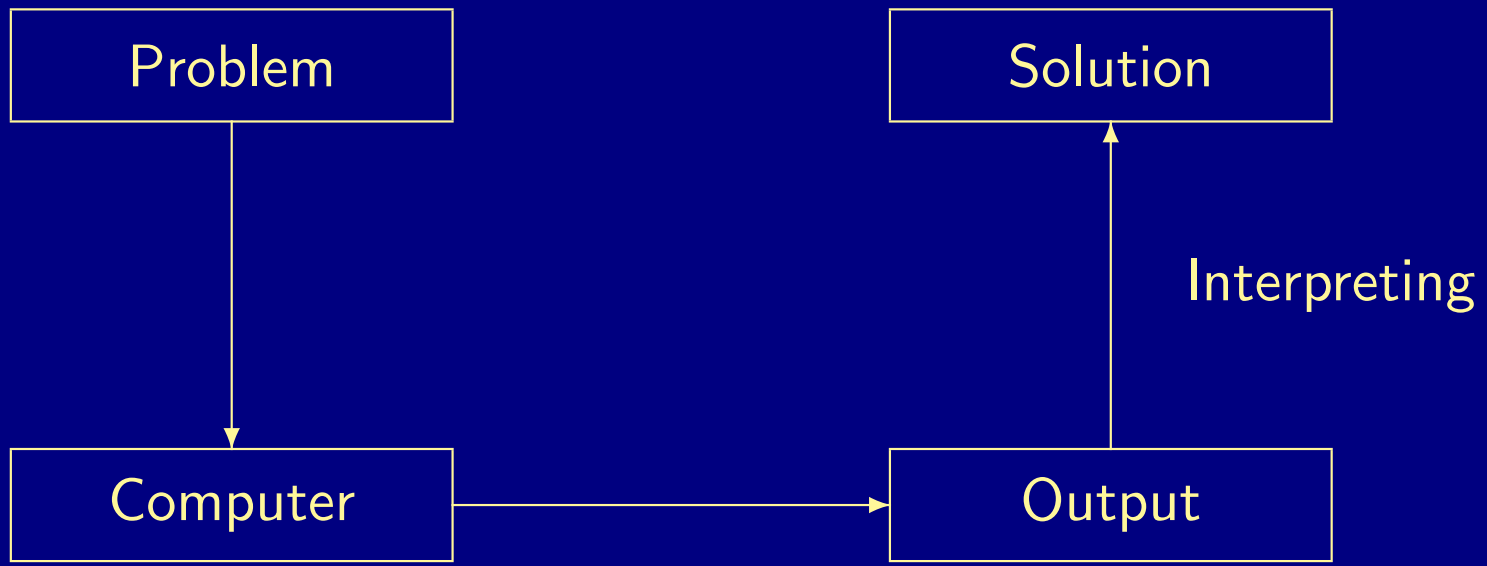
Traditional programming

"What is the problem?" versus *"How to solve the problem?"*



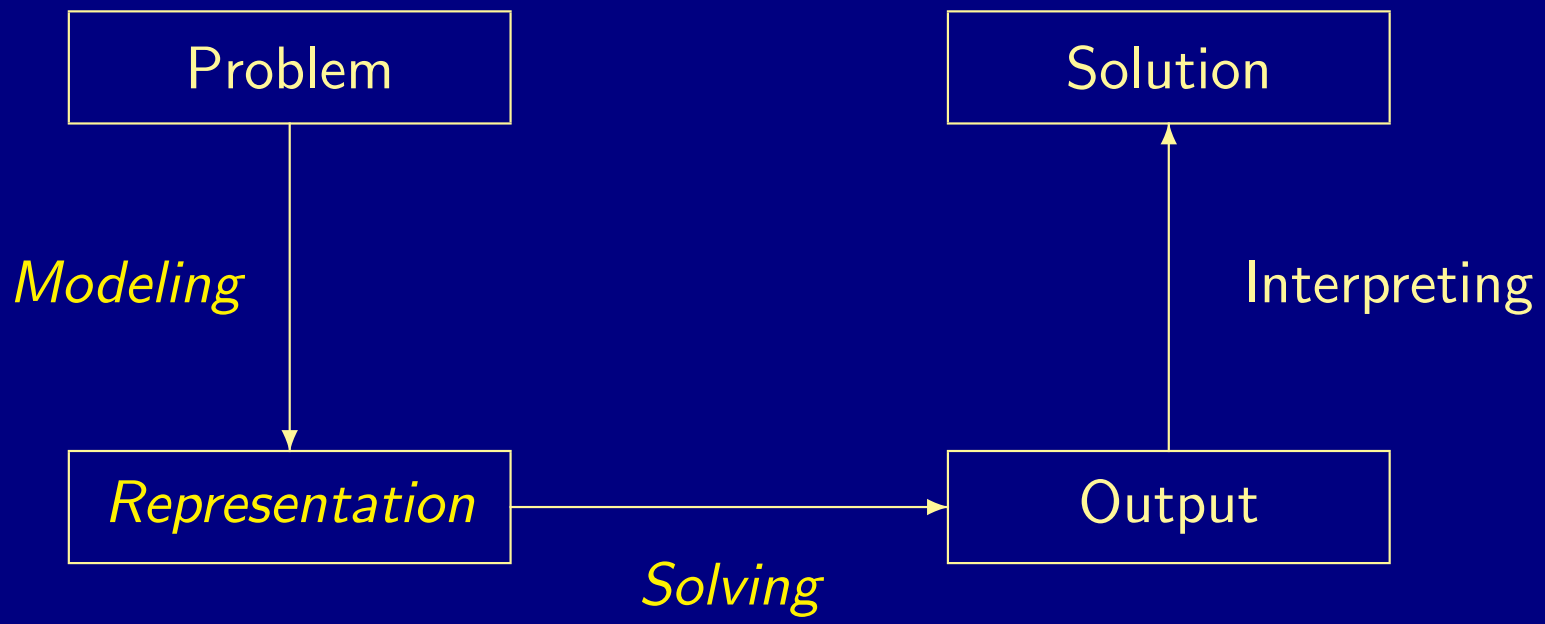
Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



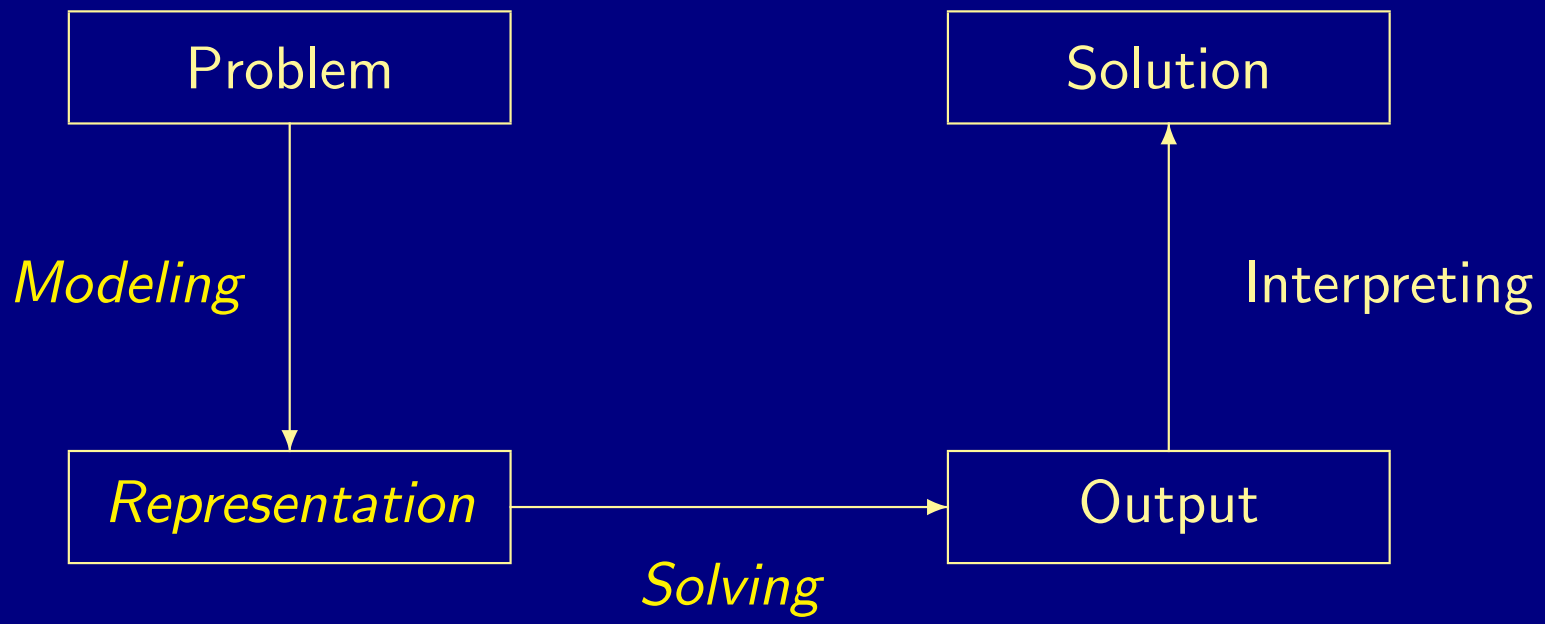
Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

4 Rooting ASP

5 Problem solving

6 Use

Answer Set Programming

in a Nutshell

ASP is an approach to declarative problem solving, combining
- a rich yet simple modeling language
- with high-performance solving capacities

ASP has its roots in
- (logic-based) knowledge representation and (nonmonotonic) reasoning
- (deductive) databases
- constraint solving (in particular, SATisfiability testing)
- logic programming (with negation)

ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way

ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11

ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11
- ASP embraces many emerging application areas

Answer Set Programming

in a Peanutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning

Answer Set Programming

in a Peanutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning

ASP = KR+DB+SAT+LP

Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

4 Rooting ASP

5 Problem solving

6 Use

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Model Generation based Problem Solving

Representation	Solution	
constraint satisfaction problem	assignment	
propositional horn theories	smallest model	
propositional theories	models	**SAT**
propositional theories	minimal models	
propositional theories	stable models	
propositional programs	minimal models	
propositional programs	supported models	
propositional programs	stable models	
first-order theories	models	
first-order theories	minimal models	
first-order theories	stable models	
first-order theories	Herbrand models	
auto-epistemic theories	expansions	
default theories	extensions	
⋮	⋮	

LP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Prolog queries

```
?- above(a,c).
true.

?- above(c,a).
no.
```

LP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Prolog queries

```
?- above(a,c).
true.

?- above(c,a).
no.
```

LP-style playing with blocks

Another Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
```

Prolog queries

```
?- above(a,c).

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Another Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
```

Prolog queries

```
?- above(a,c).

Fatal Error: local stack overflow.
```

SAT-style playing with blocks

Formula

$$
\begin{aligned}
& on(a,b) \\
\wedge \quad & on(b,c) \\
\wedge \quad & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge \quad & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{aligned}
$$

Herbrand model

$$
\begin{aligned}
\{ & on(b,b), on(a,b), on(b,c), on(a,c), \\
& above(b,b), above(c,b), above(a,b), \\
& above(b,c), above(c,c), above(a,c) \ \}
\end{aligned}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model

$$
\begin{array}{l}
\{on(b,b), on(a,b), on(b,c), on(a,c), \\
\ above(b,b), above(c,b), above(a,b), \\
\ above(b,c), above(c,c), above(a,c)\ \}
\end{array}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\begin{array}{l}
\{on(b,b), on(a,b), on(b,c), on(a,c), \\
\ above(b,b), above(c,b), above(a,b), \\
\ above(b,c), above(c,c), above(a,c)\ \}
\end{array}
$$

Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

4 Rooting ASP

5 Problem solving

6 Use

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

➥ Answer Set Programming (ASP)

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *at large*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *commonly*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *in practice*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *in practice*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
first-order programs	stable Herbrand models

ASP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Stable Herbrand model

$$\{\ \texttt{on(a,b)}, \texttt{on(b,c)}, \texttt{above(b,c)}, \texttt{above(a,b)}, \texttt{above(a,c)}\ \}$$

ASP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Stable Herbrand model

$$\{\ \texttt{on(a,b)}, \texttt{on(b,c)}, \texttt{above(b,c)}, \texttt{above(a,b)}, \texttt{above(a,c)}\ \}$$

ASP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Stable Herbrand model (and no others)

$$\{\ \mathtt{on(a,b)}, \mathtt{on(b,c)}, \mathtt{above(b,c)}, \mathtt{above(a,b)}, \mathtt{above(a,c)}\ \}$$

ASP versus LP

ASP	Prolog
Model generation	Query orientation
Bottom-up	Top-down
Modeling language	Programming language
Rule-based format	
Instantiation Flat terms	Unification Nested terms
(Turing +) $NP(^{NP})$	Turing

ASP versus SAT

ASP	SAT
Model generation	
Bottom-up	
Constructive Logic	Classical Logic
Closed (and open) world reasoning	Open world reasoning
Modeling language	—
Complex reasoning modes	Satisfiability testing
Satisfiability	Satisfiability
Enumeration/Projection	—
Optimization	—
Intersection/Union	—
(Turing +) $NP(^{NP})$	NP

Motivation: Overview

1 Motivation

2 Nutshell

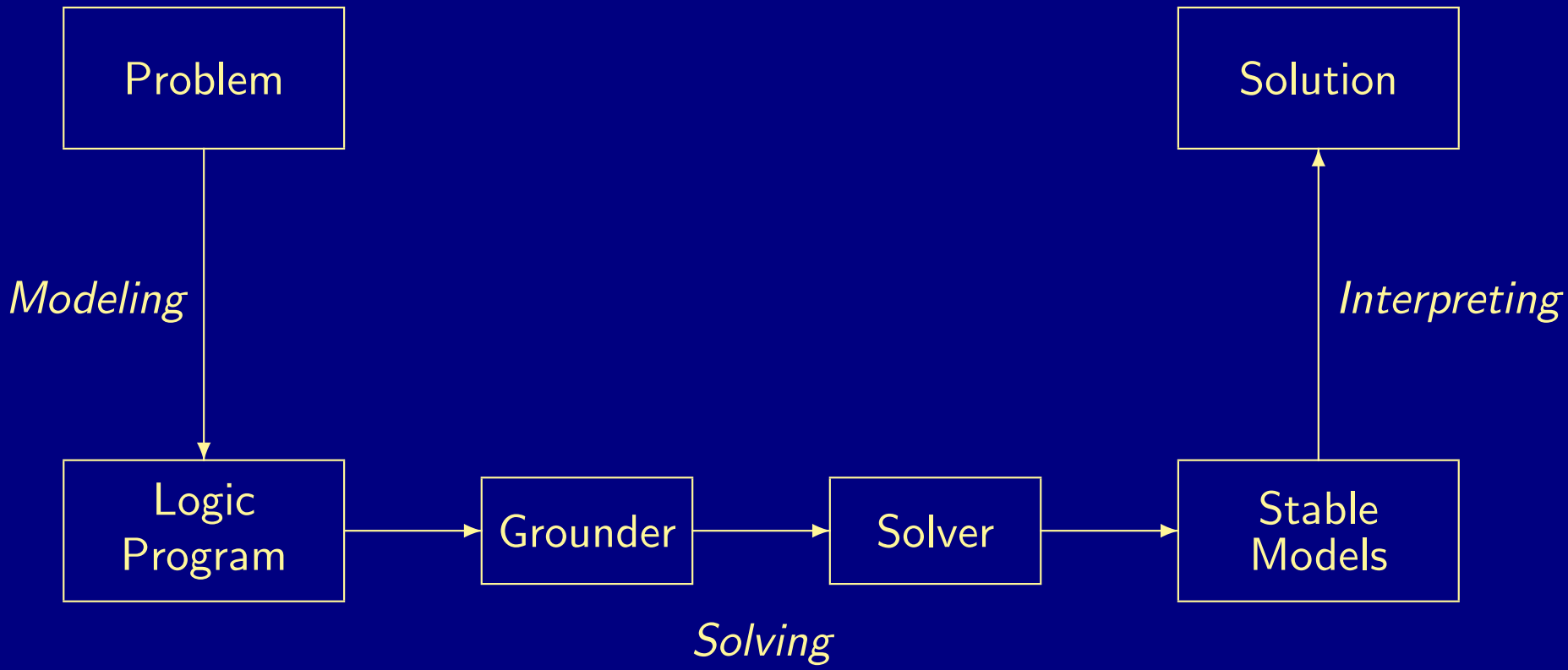
3 Shifting paradigms

4 Rooting ASP

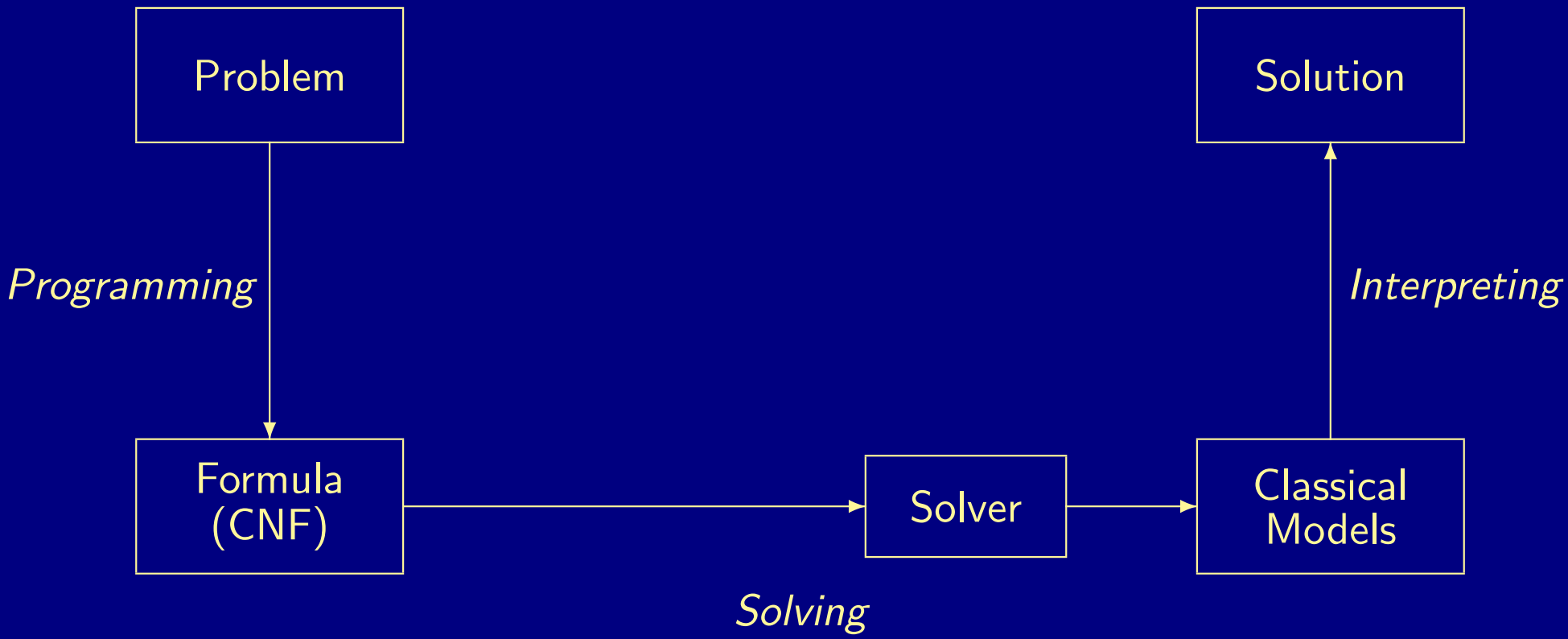
5 Problem solving

6 Use

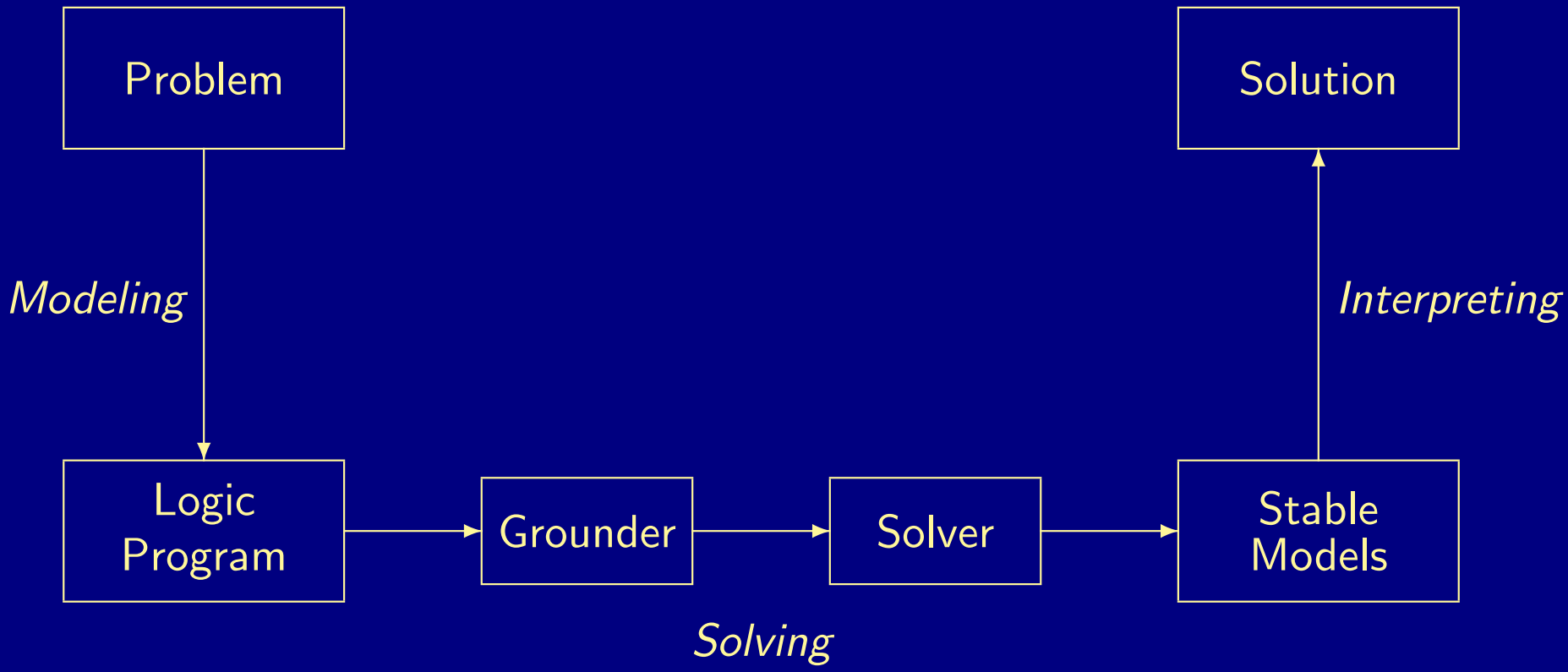
ASP solving


Problem
Solution
Modeling
Interpreting
Logic Program
Grounder
Solver
Stable Models
Solving

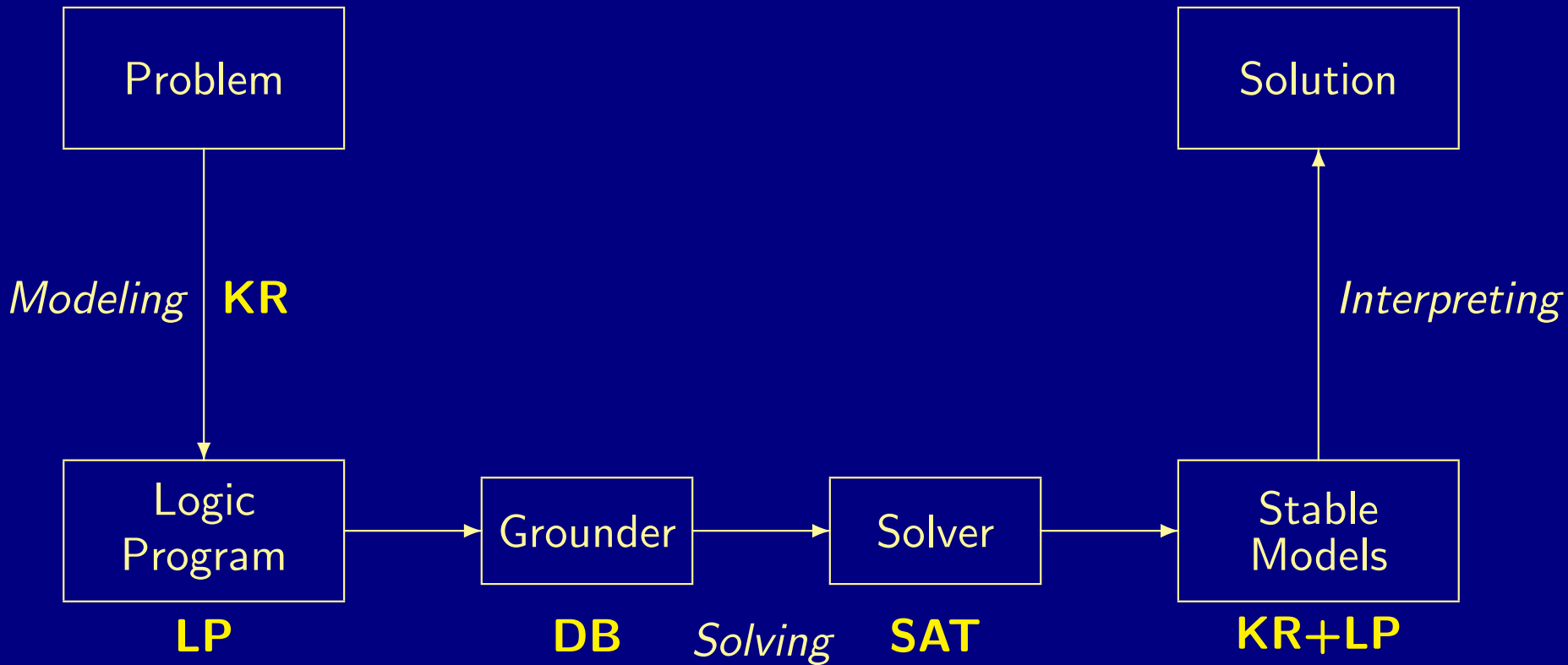
SAT solving


Problem
Solution
Programming
Interpreting
Formula
(CNF)
Solver
Classical
Models
Solving

Rooting ASP solving



Rooting ASP solving



Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

4 Rooting ASP

5 Problem solving

6 Use

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - Automated Planning,
  - Code Optimization,
  - Composition of Renaissance Music,
  - Database Integration,
  - Decision Support for NASA shuttle controllers,
  - Model Checking,
  - Product Configuration,
  - Robotics,
  - System Biology,
  - System Synthesis,
  - (industrial) Team-building,
  - and many many more.

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - Automated Planning,
  - Code Optimization,
  - Composition of Renaissance Music,
  - Database Integration,
  - Decision Support for NASA shuttle controllers,
  - Model Checking,
  - Product Configuration,
  - Robotics,
  - System Biology,
  - System Synthesis,
  - (industrial) Team-building,
  - and many many more.

What does ASP offer?

- Integration of KR, DB, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc.

What does ASP offer?

- Integration of KR, DB, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc.

ASP = KR+DB+SAT+LP

Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

Introduction: Overview

7 Syntax

8 Semantics

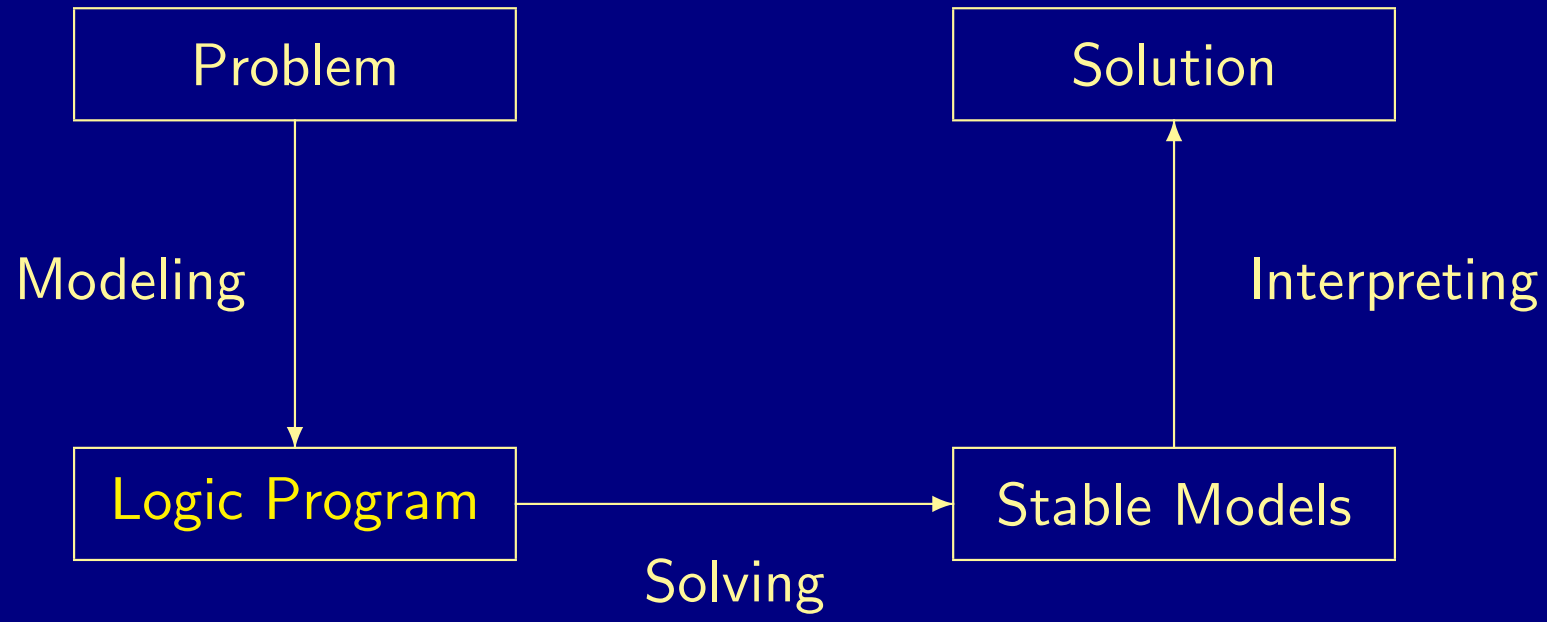
9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

Problem solving in ASP: Syntax



Normal logic programs

- A (normal) rule, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) logic program is a finite set of rules.
- Notation

$$\begin{aligned} \mathit{head}(r) &= A_0 \\ \mathit{body}(r) &= \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}(r)^+ &= \{A_1, \ldots, A_m\} \\ \mathit{body}(r)^- &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called positive if $\mathit{body}(r)^- = \emptyset$ for all its rules.

Normal logic programs

- A (normal) rule, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) logic program is a finite set of rules.
- Notation

$$\begin{aligned} \mathit{head}(r) &= A_0 \\ \mathit{body}(r) &= \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}(r)^+ &= \{A_1, \ldots, A_m\} \\ \mathit{body}(r)^- &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called positive if $\mathit{body}(r)^- = \emptyset$ for all its rules.

Normal logic programs

- A (normal) rule, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) logic program is a finite set of rules.
- Notation

$$\begin{aligned} \mathit{head}(r) &= A_0 \\ \mathit{body}(r) &= \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}(r)^+ &= \{A_1, \ldots, A_m\} \\ \mathit{body}(r)^- &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called positive if $\mathit{body}(r)^- = \emptyset$ for all its rules.

(Rough) notational convention

We sometimes use the following notation interchangeably in order to stress the respective view:

	if	and	or	negation as failure	classical negation
source code	`:-`	`,`	`|`	`not`	`-`
logic program	$\leftarrow$	,	;	$not/\sim$	$\neg$
formula	$\rightarrow$	$\wedge$	$\vee$	$\sim/(\neg)$	$\neg$

Introduction: Overview

7 Syntax

8 Semantics

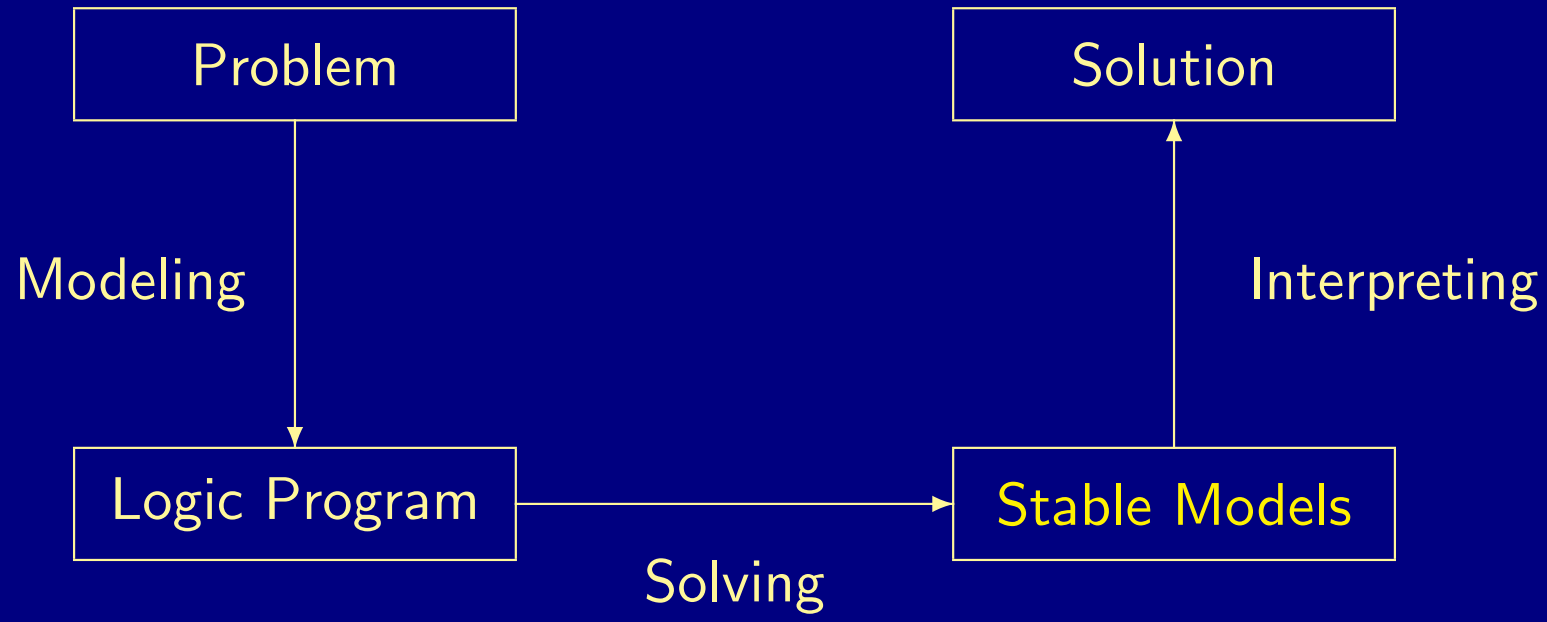
9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

Problem solving in ASP: Semantics



Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $\mathit{head}(r) \in X$ whenever $\mathit{body}(r)^+ \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - Non-definite Horn clauses can be regarded as integrity constraints.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of such set of clauses.
  - Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - Non-definite Horn clauses can be regarded as integrity constraints.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of such set of clauses.
  - Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - Non-definite Horn clauses can be regarded as integrity constraints.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of such set of clauses.
  - Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\quad q \wedge (q \wedge \neg r \rightarrow p)$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

Π_Φ

$q \leftarrow$

$p \leftarrow q,\ not\ r$

$\{p, q\}$

Informally, a set X of atoms is an answer set of a logic program Π

if X is a (classical) model of Π and

if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\quad q \wedge (q \wedge \neg r \rightarrow p)$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

Π_Φ

$$q \leftarrow$$
$$p \leftarrow q, \; not \; r$$

$\{p, q\}$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

$\{p, q\}$

$$\boxed{\begin{array}{lcl} p & \mapsto & 1 \\ q & \mapsto & 1 \\ r & \mapsto & 0 \end{array}}$$

Π_Φ $\boxed{\begin{array}{ll} q & \leftarrow \\ p & \leftarrow \ q, \ not \ r \end{array}}$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\quad q \wedge (q \wedge \neg r \rightarrow p)$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

Π_Φ

$$q \leftarrow$$
$$p \leftarrow q, \; not \; r$$

$\{p, q\}$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$\Pi_\Phi \quad \begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \ not \ r \end{array}$$

$$\{p, q\}$$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$\Pi_\Phi \quad \begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \ not \ r \end{array}$$

$$\{p, q\}$$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\quad q \wedge (q \wedge \neg r \rightarrow p)$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

Π_Φ

$$q \leftarrow$$
$$p \leftarrow q,\ not\ r$$

$\{p, q\}$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\quad q \wedge (q \wedge \neg r \rightarrow p)$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

Π_Φ

$q \leftarrow$

$p \leftarrow q,\ not\ r$

$\{p, q\}$

Informally, a set X of atoms is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Formal Definition

Normal programs

- The reduct, Π^X, of a program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body(r)^+ \mid r \in \Pi \text{ and } body(r)^- \cap X = \emptyset\}.$$

- A set X of atoms is a stable model of a program Π, if $Cn(\Pi^X) = X$.

- Note: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.
- Note: Every atom in X is justified by an *"applying rule from* Π*"*
- Sorry, but: We interchangeably use the terms answer set and stable model.

Answer set: Formal Definition

Normal programs

- The reduct, Π^X, of a program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body(r)^+ \mid r \in \Pi \text{ and } body(r)^- \cap X = \emptyset\}.$$

- A set X of atoms is a stable model of a program Π, if $Cn(\Pi^X) = X$.

- Note: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.
- Note: Every atom in X is justified by an *"applying rule from* Π*"*
- Sorry, but: We interchangeably use the terms answer set and stable model.

Answer set: Formal Definition

Normal programs

- The reduct, Π^X, of a program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body(r)^+ \mid r \in \Pi \text{ and } body(r)^- \cap X = \emptyset\}.$$

- A set X of atoms is a stable model of a program Π, if $Cn(\Pi^X) = X$.

- Note: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.
- Note: Every atom in X is justified by an *"applying rule from* Π*"*
- Sorry, but: We interchangeably use the terms answer set and stable model.

Answer set: Formal Definition

Normal programs

- The reduct, Π^X, of a program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body(r)^+ \mid r \in \Pi \text{ and } body(r)^- \cap X = \emptyset\}.$$

- A set X of atoms is a stable model of a program Π, if $Cn(\Pi^X) = X$.

- Note: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.
- Note: Every atom in X is justified by an *"applying rule from* Π*"*
- Sorry, but: We interchangeably use the terms answer set and stable model.

A closer look at Π^X

- In other words, given a set X of atoms from Π,

  Π^X is obtained from Π by deleting
  1. each rule having a *not* A in its body with $A \in X$
     and then
  2. all negative atoms of the form *not* A
     in the bodies of the remaining rules.

- Note: Only negative body literals are evaluated wrt X

A closer look at Π^X

- In other words, given a set X of atoms from Π,

  Π^X is obtained from Π by deleting
  1. each rule having a *not* A in its body with $A \in X$
     and then
  2. all negative atoms of the form *not* A
     in the bodies of the remaining rules.

- Note: Only negative body literals are evaluated wrt X

Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p\}$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow \mathit{not}\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	✗
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A second example

$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$
$\{p\}$	$p \leftarrow$	$\{p\}$
$\{q\}$	$q \leftarrow$	$\{q\}$
$\{p, q\}$		$\emptyset$

A second example

$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$ ✗
$\{p\}$	$p \leftarrow$	$\{p\}$ ✓
$\{q\}$	$q \leftarrow$	$\{q\}$ ✓
$\{p, q\}$		$\emptyset$ ✗

A second example

$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$\Pi = \{p \leftarrow \mathit{not}\ q,\ \ q \leftarrow \mathit{not}\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A third example

$\Pi = \{p \leftarrow \mathit{not}\ p\}$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow$	$\{p\}$
$\{p\}$		$\emptyset$

A third example

$\Pi = \{p \leftarrow \mathit{not}\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✗
$\{p\}$		$\emptyset$	✗

A third example

$\Pi = \{p \leftarrow \mathit{not}\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

A third example

$\Pi = \{p \leftarrow \mathit{not}\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

Answer set: Some properties

- A logic program may have zero, one, or multiple answer sets!
- If X is an answer set of a logic program Π, then X is a model of Π (seen as a formula).
- If X and Y are answer sets of a *normal* program Π, then $X \not\subset Y$.

Answer set: Some properties

- A logic program may have zero, one, or multiple answer sets!
- If X is an answer set of a logic program Π,
  then X is a model of Π (seen as a formula).
- If X and Y are answer sets of a *normal* program Π,
  then $X \not\subset Y$.

Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

Programs with Variables

Let Π be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.

- Ground Instantiation of Π: $ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$

Programs with Variables

Let Π be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms (also called Herbrand universe)
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$ (also called alphabet or Herbrand base)
  - Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

    $$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}\}$$

    where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.
  - Ground Instantiation of Π: $ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$

Programs with Variables

Let Π be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.

- Ground Instantiation of Π: $ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$

Programs with Variables

Let Π be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.

- Ground Instantiation of Π: $ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ r(b,c) \leftarrow,\ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$\mathcal{T} = \{a, b, c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{l} r(a,b) \leftarrow\ , \\ r(b,c) \leftarrow\ , \\ t(a,a) \leftarrow r(a,a),\ t(b,a) \leftarrow r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b),\ t(b,b) \leftarrow r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c),\ t(b,c) \leftarrow r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

Intelligent Grounding aims at reducing the ground instantiation.

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ r(b,c) \leftarrow,\ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$\mathcal{T} = \{a, b, c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{l} r(a,b) \leftarrow\ , \\ r(b,c) \leftarrow\ , \\ t(a,a) \leftarrow r(a,a),\ t(b,a) \leftarrow r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b),\ t(b,b) \leftarrow r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c),\ t(b,c) \leftarrow r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

- Intelligent Grounding aims at reducing the ground instantiation.

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ r(b,c) \leftarrow,\ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$\mathcal{T} = \{a, b, c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{l} r(a,b) \leftarrow\ , \\ r(b,c) \leftarrow\ , \\ t(a,a) \leftarrow r(a,a),\ t(b,a) \leftarrow r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b),\ t(b,b) \leftarrow r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c),\ t(b,c) \leftarrow r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

- Intelligent Grounding aims at reducing the ground instantiation.

Answer sets of programs with Variables

Let Π be a normal logic program with variables.

- A set X of (ground) atoms as a stable model of Π, if $Cn(ground(\Pi)^X) = X$.

Answer sets of programs with Variables

Let Π be a normal logic program with variables.

- A set X of (ground) atoms as a stable model of Π, if $Cn(ground(\Pi)^X) = X$.

Introduction: Overview

7 Syntax

8 Semantics

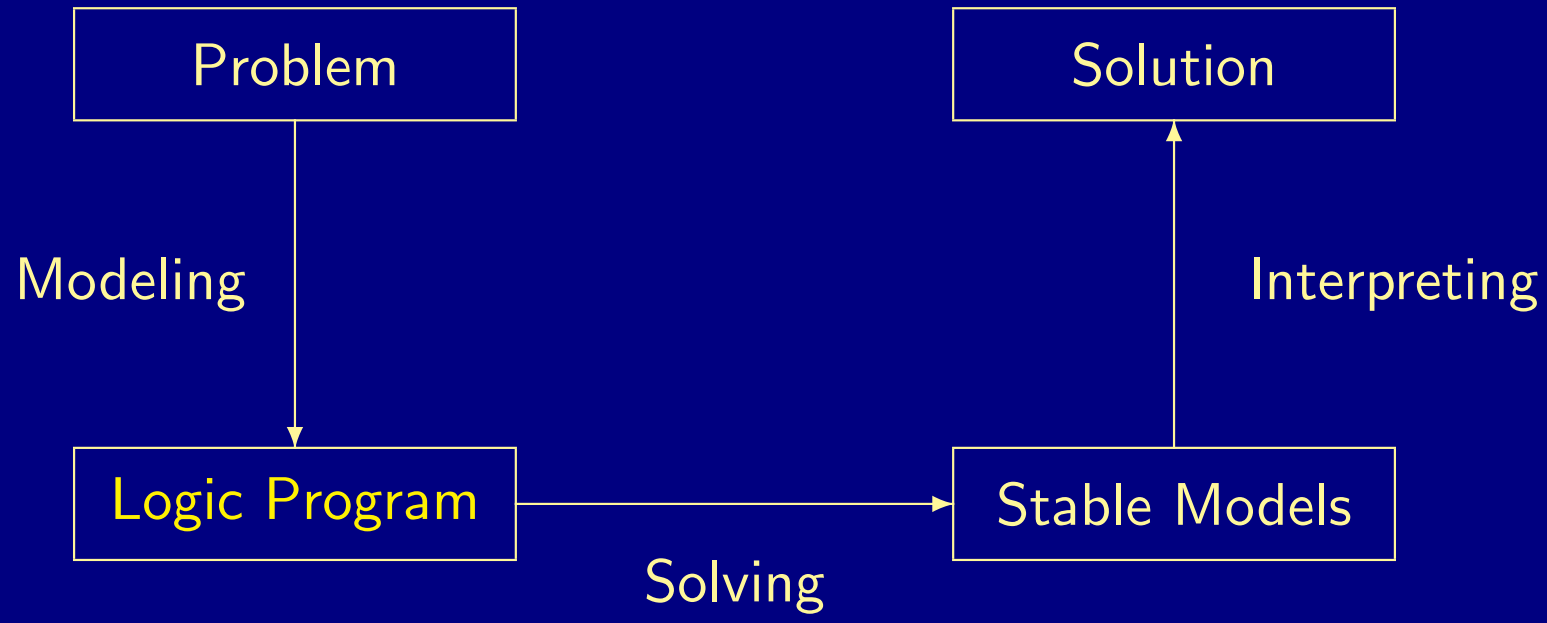
9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

Problem solving in ASP: Extended Syntax



Language Constructs

Variables (over the Herbrand Universe)

`p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for

```
p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)
```

Conditional Literals

`p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for

```
p :- q(a), q(b), q(c)
```

Disjunction

```
p(X) | q(X) :- r(X)
```

Integrity Constraints

```
:- q(X), p(X)
```

Choice

```
2 { p(X,Y) : q(X) } 7 :- r(Y)
```

Aggregates

```
s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7
```

also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a`, `b`, `c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- **Disjunction**
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Introduction: Overview

7 Syntax

8 Semantics

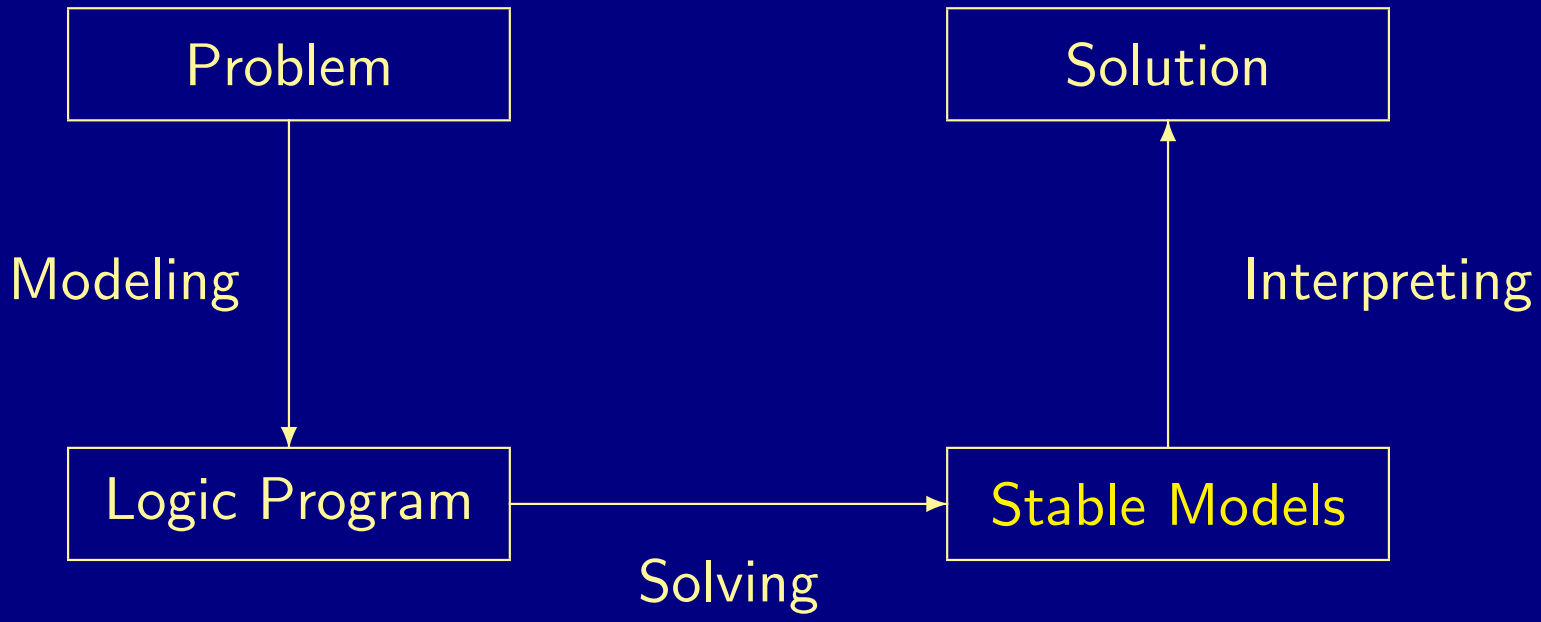
9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes

Problem solving in ASP: Reasoning Modes



Reasoning Modes

- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization
- and combinations of them

† without solution recording

‡ without solution enumeration

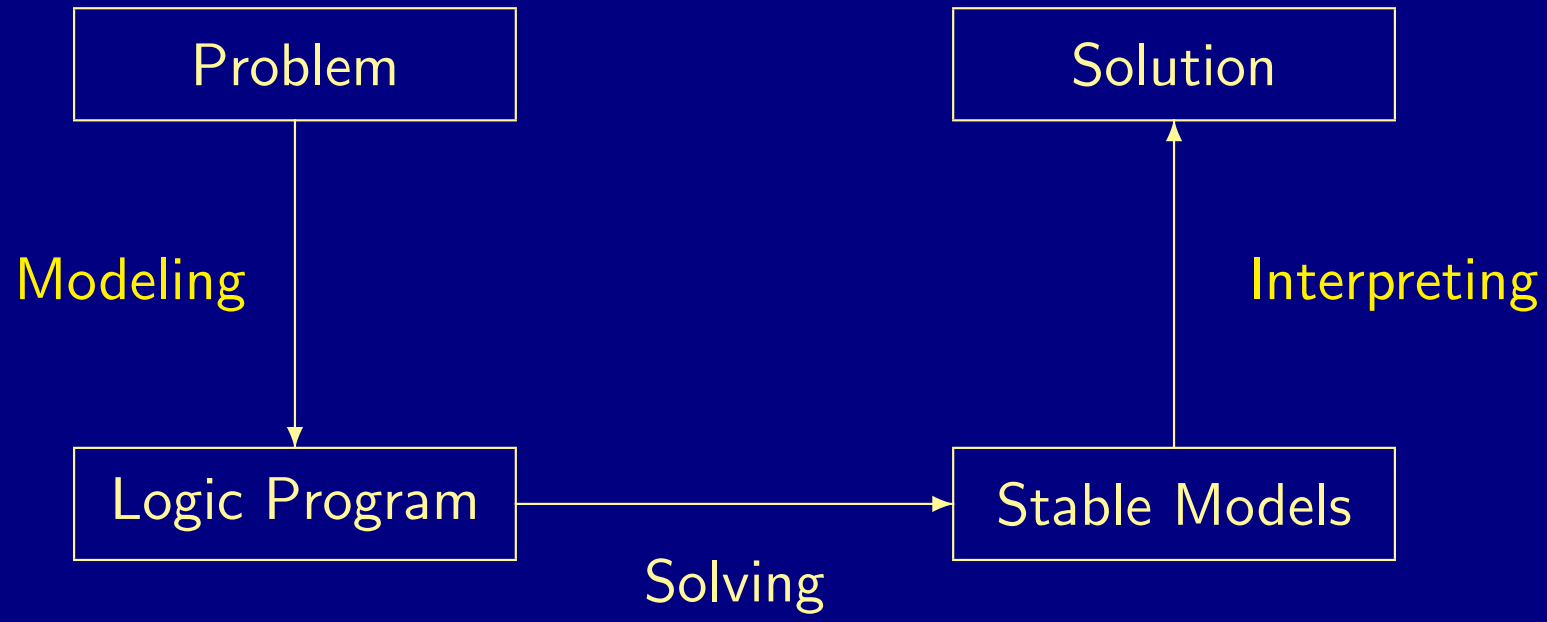
Basic Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
- Graph Coloring

15 Methodology
- Satisfiability
- Queens
- Reviewer Assignment
- Planning

Modeling and Interpreting



Modeling

For solving a problem class P for a problem instance I, encode

1. the problem instance I as a set C(I) of facts and
2. the problem class P as a set C(P) of rules

such that the solutions to P for I can be (polynomially) extracted from the answer sets of $C(I) \cup C(P)$.

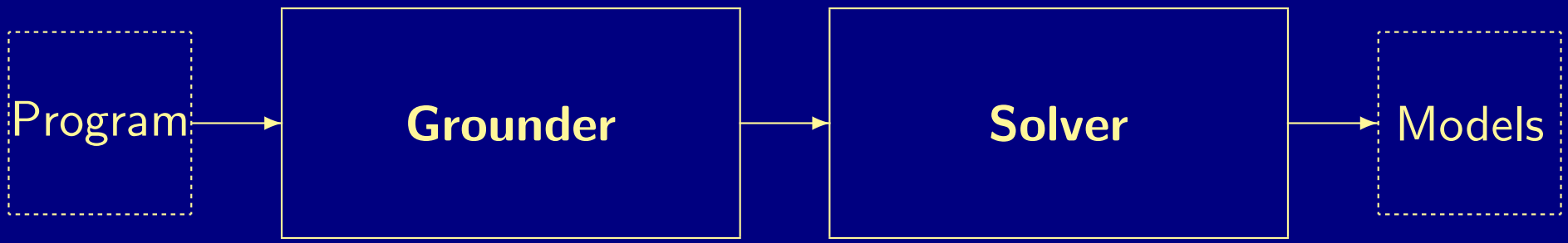
Basic Modeling: Overview

13 ASP Solving Process

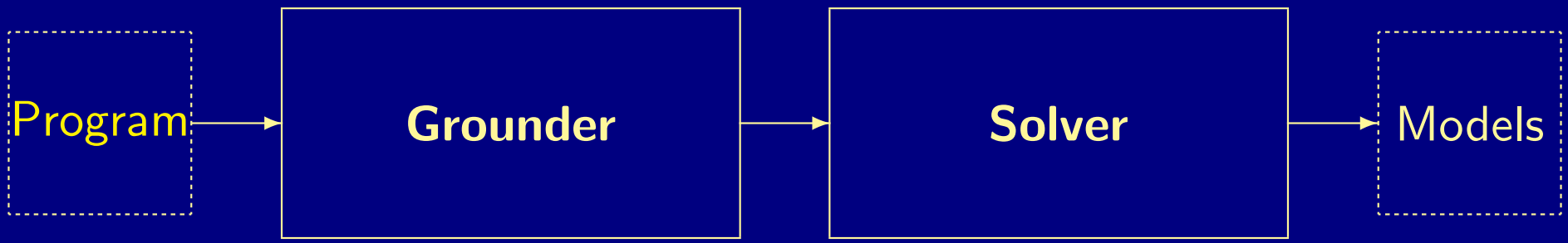
14 Problems as Logic Programs
- Graph Coloring

15 Methodology
- Satisfiability
- Queens
- Reviewer Assignment
- Planning

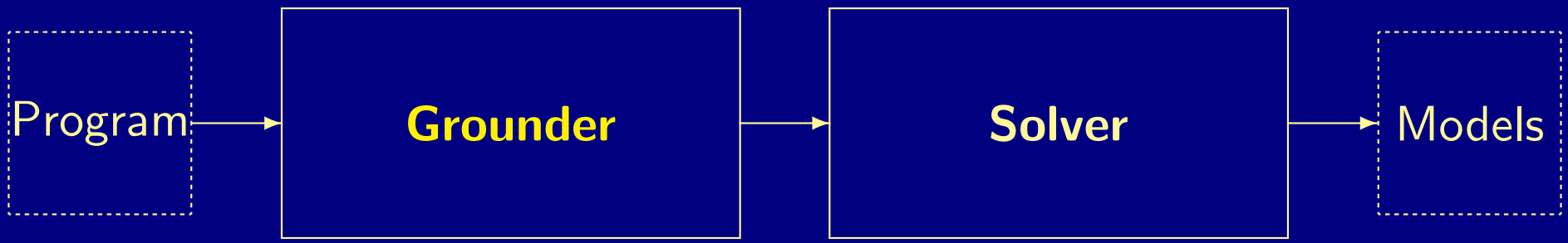
ASP Solving Process



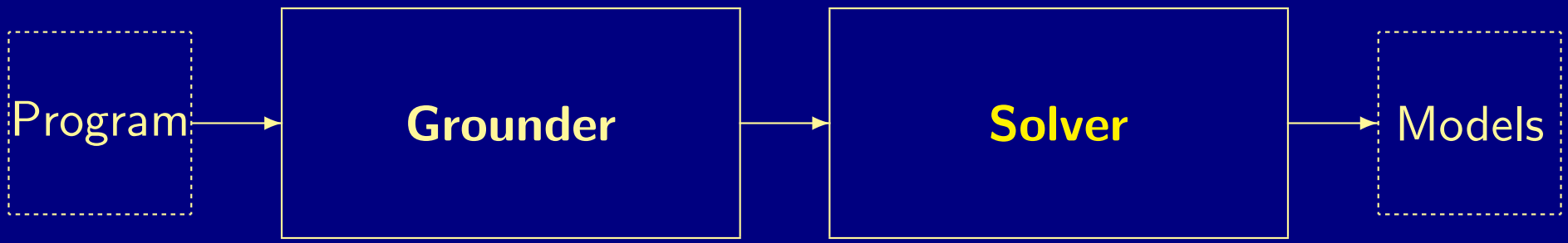
ASP Solving Process



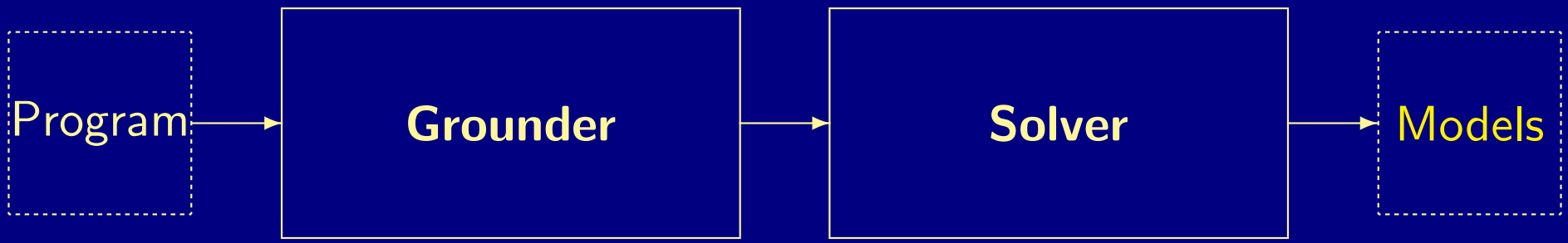
ASP Solving Process



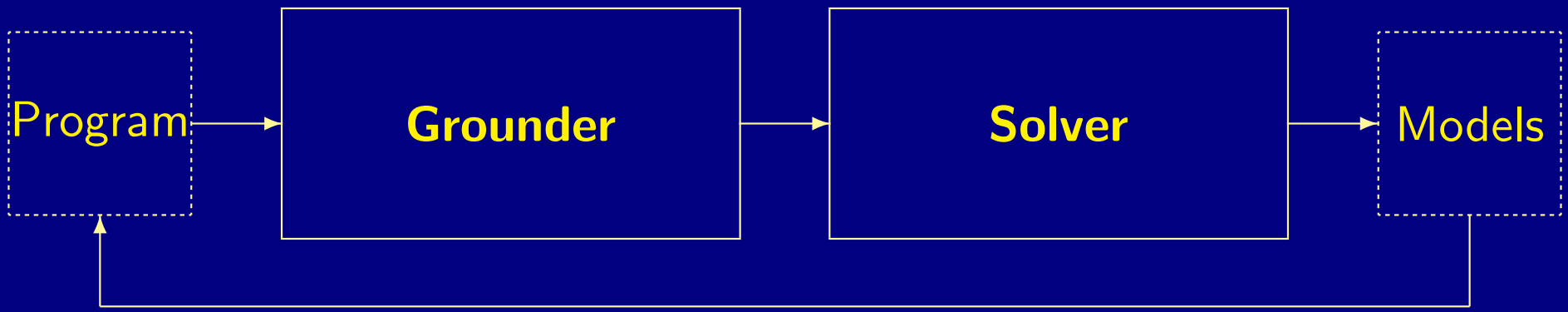
ASP Solving Process



ASP Solving Process



ASP Solving Process



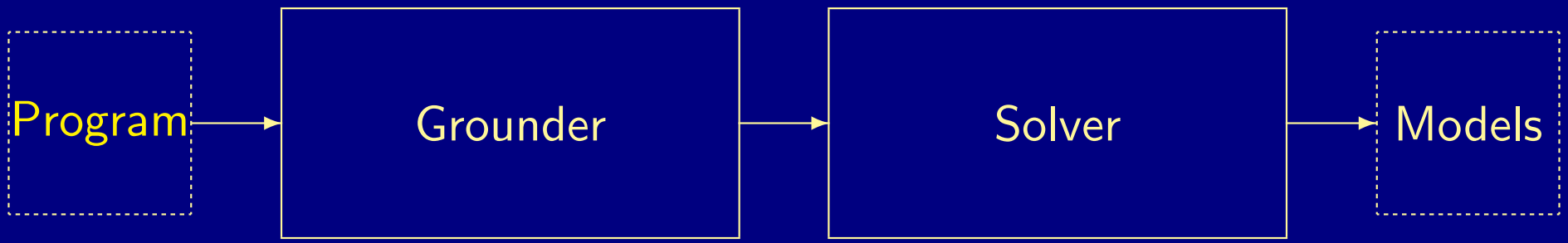
Basic Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
- Graph Coloring

15 Methodology
- Satisfiability
- Queens
- Reviewer Assignment
- Planning

ASP Solving Process



Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).

1 {color(X,C) : col(C)} 1 :- node(X).

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

Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).

1 {color(X,C) : col(C)} 1 :- node(X).

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

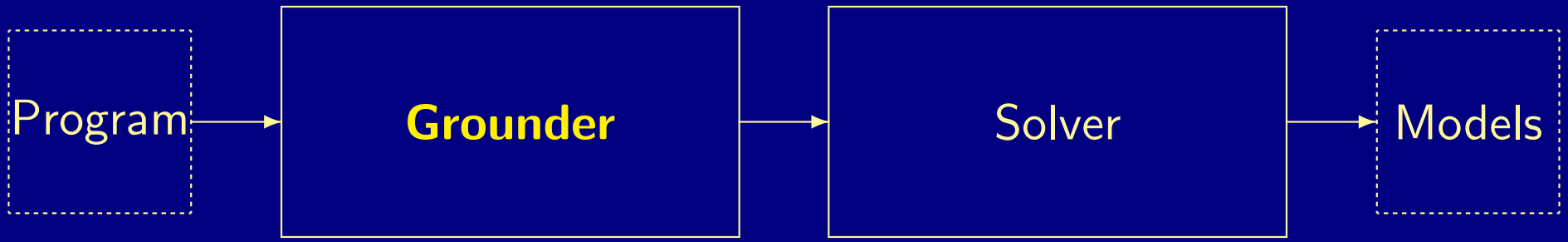
Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).

1 {color(X,C) : col(C)} 1 :- node(X).

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

Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).

1 {color(X,C) : col(C)} 1 :- node(X).

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

ASP Solving Process



Graph Coloring: Grounding

```
$ gringo -t color.lp
```

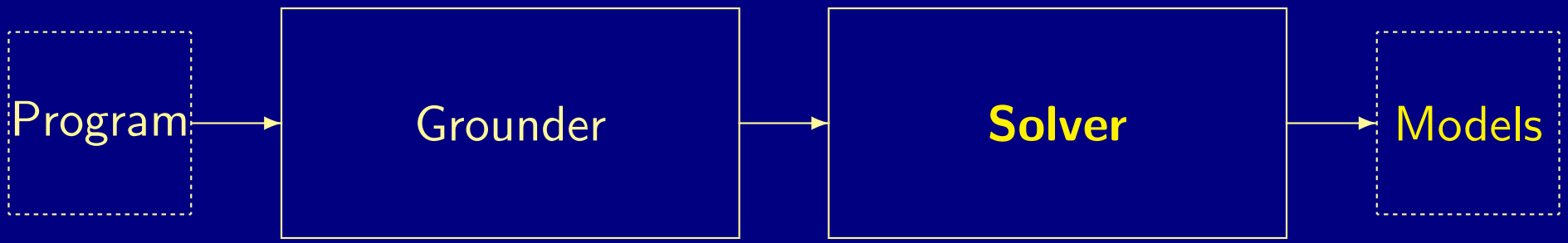
```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g). ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).      :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).      :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).      :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).      :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).      :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).      :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).      :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).      :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

Graph Coloring: Grounding

```
$ gringo -t color.lp
```

```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g). ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).      :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).      :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).      :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).      :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).      :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).      :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).      :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).      :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

ASP Solving Process



Graph Coloring: Solving

```
$ gringo color.lp | clasp 0
```

```
clasp version 1.2.1
Reading from stdin
Reading      : Done(0.000s)
Preprocessing: Done(0.000s)
Solving...
Answer: 1
color(1,b) color(2,r) color(3,r) color(4,g) color(5,b) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 2
color(1,g) color(2,r) color(3,r) color(4,b) color(5,g) color(6,b) node(1) ... edge(1,2) ... col(r) ...
Answer: 3
color(1,b) color(2,g) color(3,g) color(4,r) color(5,b) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 4
color(1,g) color(2,b) color(3,b) color(4,r) color(5,g) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 5
color(1,r) color(2,b) color(3,b) color(4,g) color(5,r) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 6
color(1,r) color(2,g) color(3,g) color(4,b) color(5,r) color(6,b) node(1) ... edge(1,2) ... col(r) ...

Models       : 6
Time         : 0.000  (Solving: 0.000)
```

Graph Coloring: Solving

```
$ gringo color.lp | clasp 0
```

```
clasp version 1.2.1
Reading from stdin
Reading      : Done(0.000s)
Preprocessing: Done(0.000s)
Solving...
Answer: 1
color(1,b) color(2,r) color(3,r) color(4,g) color(5,b) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 2
color(1,g) color(2,r) color(3,r) color(4,b) color(5,g) color(6,b) node(1) ... edge(1,2) ... col(r) ...
Answer: 3
color(1,b) color(2,g) color(3,g) color(4,r) color(5,b) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 4
color(1,g) color(2,b) color(3,b) color(4,r) color(5,g) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 5
color(1,r) color(2,b) color(3,b) color(4,g) color(5,r) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 6
color(1,r) color(2,g) color(3,g) color(4,b) color(5,r) color(6,b) node(1) ... edge(1,2) ... col(r) ...

Models       : 6
Time         : 0.000  (Solving: 0.000)
```

Basic Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
- Graph Coloring

15 Methodology
- Satisfiability
- Queens
- Reviewer Assignment
- Planning

Basic Methodology

Generate and Test (or: Guess and Check) approach

Generator Generate potential answer set candidates
(typically through non-deterministic constructs)

Tester Eliminate invalid candidates
(typically through integrity constraints)

Nutshell

$$\text{Logic program} = \text{Data} + \text{Generator} + \text{Tester}$$
$$(+ \text{ Optimizer})$$

Basic Methodology

Generate and Test (or: Guess and Check) approach

Generator	Generate potential answer set candidates (typically through non-deterministic constructs)
Tester	Eliminate invalid candidates (typically through integrity constraints)

Nutshell

$$\text{Logic program} = \text{Data} + \text{Generator} + \text{Tester} \; (+ \text{Optimizer})$$

Satisfiability

- Problem Instance: A propositional formula ϕ in CNF.
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true.

- Example: Consider formula $(a \vee \neg b) \wedge (\neg a \vee b)$.
- Logic Program:

Generator	**Tester**	**Stable models**
{a,b} $\leftarrow$	$\leftarrow$ not a, b	X_1 = {a,b}
	$\leftarrow$ a, not b	X_2 = {}

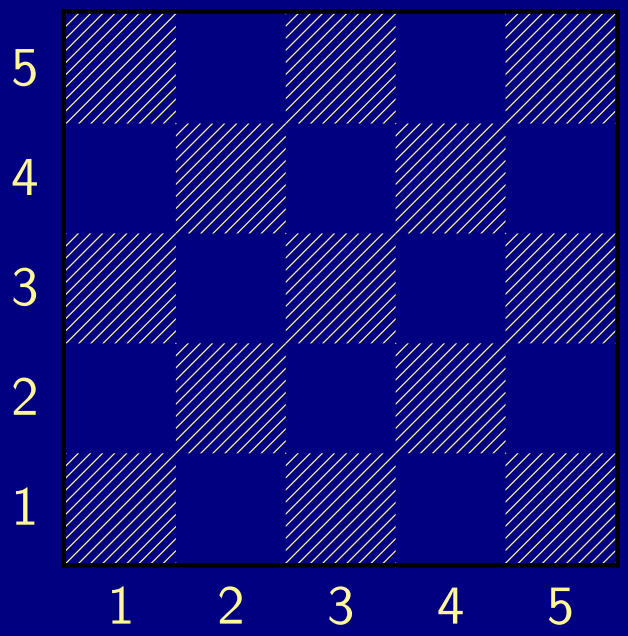
Satisfiability

- Problem Instance: A propositional formula ϕ in CNF.
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true.

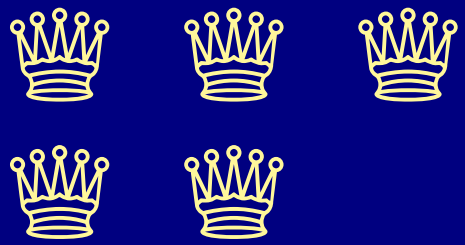
- Example: Consider formula $(a \vee \neg b) \wedge (\neg a \vee b)$.
- Logic Program:

Generator	**Tester**	**Stable models**
{a,b} ←	← not a, b	X_1 = {a,b}
	← a, not b	X_2 = {}

The n-Queens Problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another



Defining the Field

queens.lp

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

- Create file `queens.lp`
- Define the field
  - n rows
  - n columns

Defining the Field

Running ...

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
SATISFIABLE

Models       : 1
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Placing some Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
```

- Guess a solution candidate
- Place some queens on the board

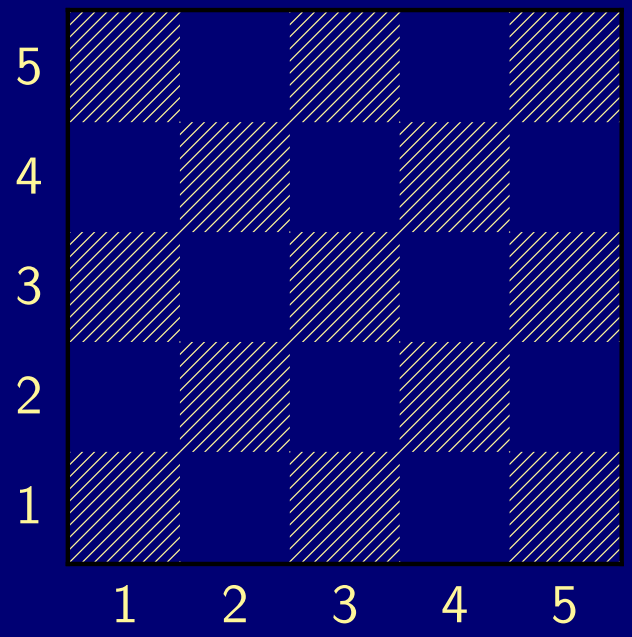
Placing some Queens

Running ...

```
$ clingo queens.lp -c n=5 3
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(1,1)
Answer: 3
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(2,1)
SATISFIABLE

Models       : 3+
```

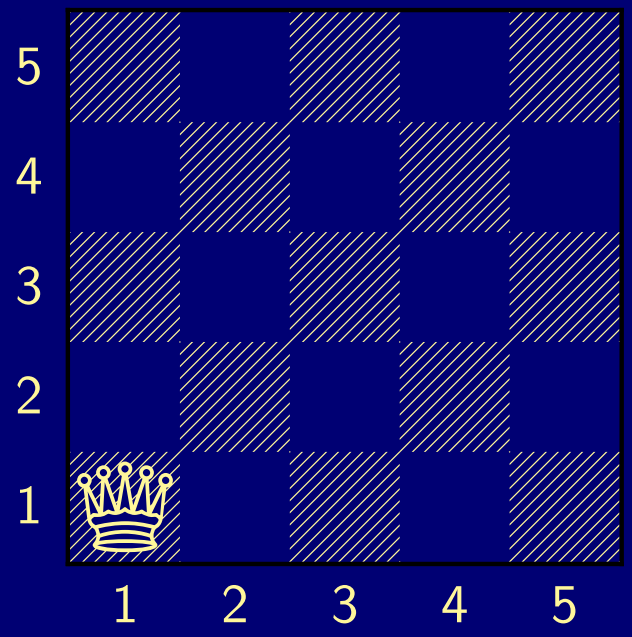
Placing some Queens: Answer 1

Answer 1



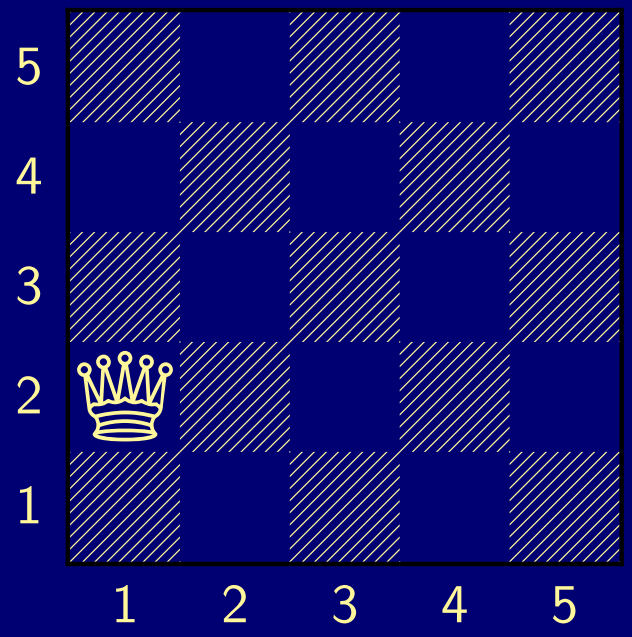
Placing some Queens: Answer 2

Answer 2



Placing some Queens: Answer 3

Answer 3



Placing n Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
```

- Place exactly n queens on the board

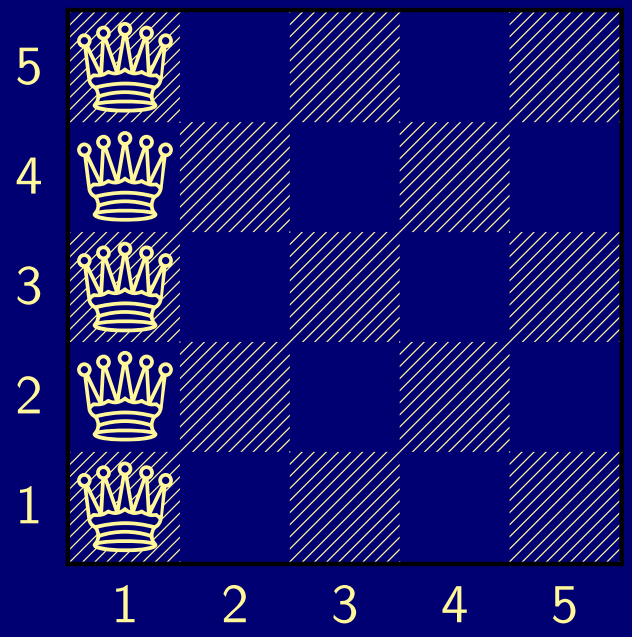
Placing n Queens

Running ...

```
$ clingo queens.lp -c n=5 2
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,1) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,2) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
...
```

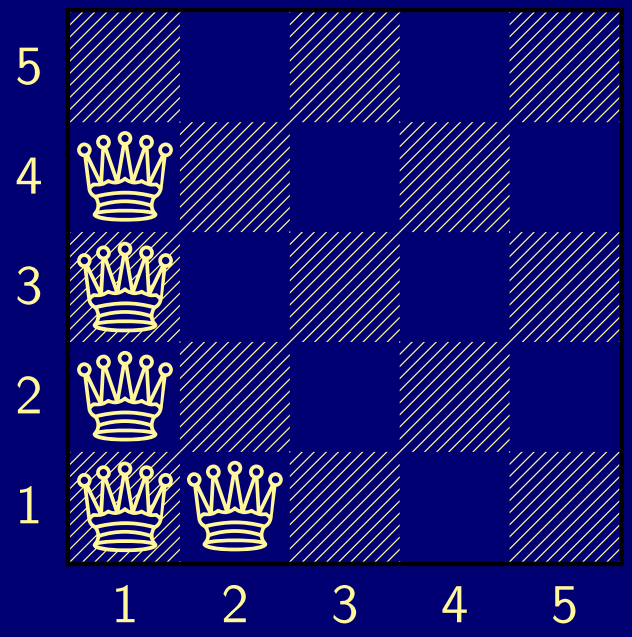
Placing n Queens: Answer 1

Answer 1



Placing n Queens: Answer 2

Answer 2



Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

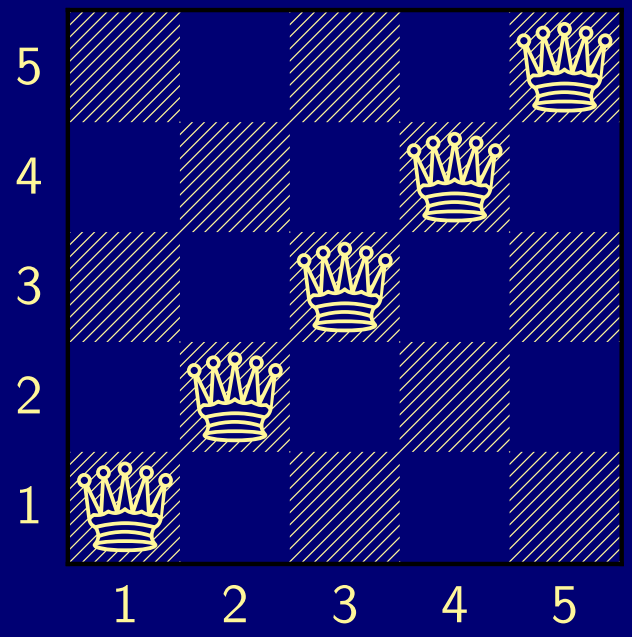
Horizontal and vertical Attack

Running ...

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,5) queen(4,4) queen(3,3) \
queen(2,2) queen(1,1)
...
```

Horizontal and vertical Attack: Answer 1

Answer 1



Diagonal Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ),
   I-J == II-JJ.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ),
   I+J == II+JJ.
```

- Forbid diagonal attacks

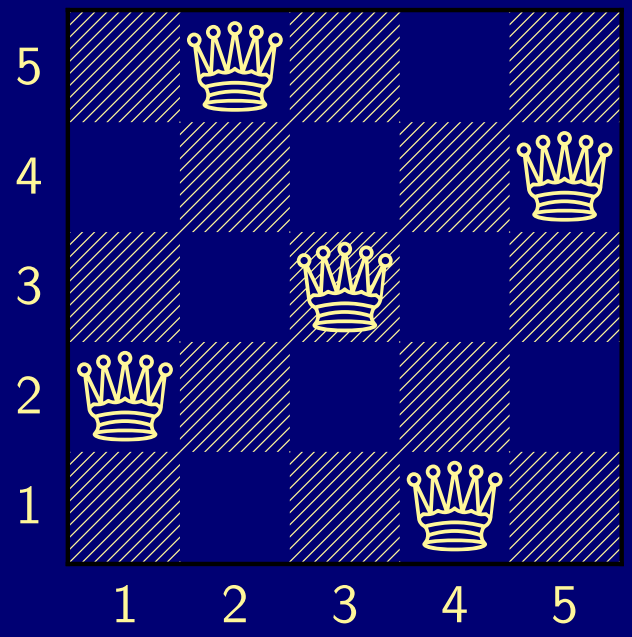
Diagonal Attack

Running ...

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(4,5) queen(1,4) queen(3,3) \
queen(5,2) queen(2,1)
SATISFIABLE

Models       : 1+
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Diagonal Attack: Answer 1

Answer 1



Optimizing

queens-opt.lp

```
1 { queen(I,1..n) } 1 :- I = 1..n.
1 { queen(1..n,J) } 1 :- J = 1..n.
:- { queen(D-J,J) } 2, D =   2..2*n.
:- { queen(D+J,J) } 2, D = 1-n..n-1.
```

- Encoding can be optimized
- Much faster to solve

- See Section *Tweaking N-Queens*

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- 9 { assigned(P,R) : paper(P) }  , reviewer(R).
 :-   { assigned(P,R) : paper(P) } 6, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- 9 { assigned(P,R) : paper(P) }   , reviewer(R).
 :-   { assigned(P,R) : paper(P) } 6, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- 9 { assigned(P,R) : paper(P) }  , reviewer(R).
 :-   { assigned(P,R) : paper(P) } 6, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- 9 { assigned(P,R) : paper(P) }   , reviewer(R).
 :-   { assigned(P,R) : paper(P) } 6, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- 9 { assigned(P,R) : paper(P) }   , reviewer(R).
 :-   { assigned(P,R) : paper(P) } 6, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Simplistic STRIPS Planning

```
fluent(p).    fluent(q).    fluent(r).
action(a).    pre(a,p).     add(a,q).     del(a,p).
action(b).    pre(b,q).     add(b,r).     del(b,q).
init(p).      query(r).

time(1..k).   lasttime(T) :- time(T), not time(T+1).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Simplistic STRIPS Planning

```
fluent(p).    fluent(q).    fluent(r).
action(a).    pre(a,p).     add(a,q).     del(a,p).
action(b).    pre(b,q).     add(b,r).     del(b,q).
init(p).      query(r).

time(1..k).   lasttime(T) :- time(T), not time(T+1).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Simplistic STRIPS Planning

```
fluent(p).     fluent(q).     fluent(r).
action(a).     pre(a,p).      add(a,q).     del(a,p).
action(b).     pre(b,q).      add(b,r).     del(b,q).
init(p).       query(r).

time(1..k).    lasttime(T) :- time(T), not time(T+1).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Language Extensions: Overview

16 Motivation
17 Integrity Constraints
18 Choice Rules
19 Cardinality Constraints
20 Weight Constraints
21 Optimization statements
22 Conditional literals
23 smodels format

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Integrity Constraints

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom
- Example `:- edge(X,Y), color(X,C), color(Y,C).`
- Embedding For a new symbol x, map

$$\begin{array}{rcl} & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto \quad x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \Pi \cup \{\leftarrow p\}$ and $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

Integrity Constraints

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom
- Example `:- edge(X,Y), color(X,C), color(Y,C).`
- Embedding For a new symbol x, map

$$\begin{array}{rcl} & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto \quad x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \Pi \cup \{\leftarrow p\}$ and $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

Integrity Constraints

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom
- Example `:- edge(X,Y), color(X,C), color(Y,C).`
- Embedding For a new symbol x, map

$$\begin{array}{rrcl} & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto & x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \Pi \cup \{\leftarrow p\}$ and $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Choice rules

- Idea Choices over subsets
- Syntax

$$\{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o,$$

- Informal meaning If the body is satisfied in an answer set, then any subset of $\{A_1, \ldots, A_m\}$ can be included in the answer set
- Example `1 {color(X,C) : col(C)} 1 :- node(X).`
- Another Example Program $\Pi = \{\ \{a\} \leftarrow b,\ b \leftarrow\}$ has two answer sets: $\{b\}$ and $\{a, b\}$

Choice rules

- Idea Choices over subsets
- Syntax

$$\{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o,$$

- Informal meaning If the body is satisfied in an answer set, then any subset of $\{A_1, \ldots, A_m\}$ can be included in the answer set
- Example `1 {color(X,C) : col(C)} 1 :- node(X).`
- Another Example Program $\Pi = \{\ \{a\} \leftarrow b,\ b \leftarrow\}$ has two answer sets: $\{b\}$ and $\{a, b\}$

Choice rules

- Idea Choices over subsets
- Syntax

$$\{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o,$$

- Informal meaning If the body is satisfied in an answer set, then any subset of $\{A_1, \ldots, A_m\}$ can be included in the answer set
- Example `1 {color(X,C) : col(C)} 1 :- node(X).`
- Another Example Program $\Pi = \{\ \{a\} \leftarrow b,\ b \leftarrow\}$ has two answer sets: $\{b\}$ and $\{a, b\}$

Embedding in normal logic programs

- A choice rule of form

$$\{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o$$

can be translated into $2m + 1$ rules

$$A \leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o$$

$$A_1 \leftarrow A, \mathit{not}\ \overline{A_1} \quad \ldots \quad A_m \leftarrow A, \mathit{not}\ \overline{A_m}$$

$$\overline{A_1} \leftarrow \mathit{not}\ A_1 \quad \ldots \quad \overline{A_m} \leftarrow \mathit{not}\ A_m$$

by introducing new atoms $A, \overline{A_1}, \ldots, \overline{A_m}$

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Cardinality constraints

- Syntax A (positive) cardinality constraint is of the form

$$l\ \{A_1, \ldots, A_m\}\ u$$

- Informal meaning A cardinality constraint is satisfied in an answer set X, if the number of atoms from $\{A_1, \ldots, A_m\}$ satisfied in X is between l and u (inclusive)
  More formally, if $l \leq |\{A_1, \ldots, A_m\} \cap X| \leq u$
- Example 2 {hd(a),. . . ,hd(m)} 4
- Conditions $l\ \{A_1 : B_1, \ldots, A_m : B_m\}\ u$
  where $B_1, \ldots, B_m$ are used for restricting instantiations of variables occurring in $A_1, \ldots, A_m$

Cardinality constraints

- Syntax A (positive) cardinality constraint is of the form

$$l\ \{A_1, \ldots, A_m\}\ u$$

- Informal meaning A cardinality constraint is satisfied in an answer set X, if the number of atoms from $\{A_1, \ldots, A_m\}$ satisfied in X is between l and u (inclusive)
  More formally, if $l \leq |\{A_1, \ldots, A_m\} \cap X| \leq u$
- Example 2 {hd(a),. . . ,hd(m)} 4

- Conditions $l\ \{A_1 : B_1, \ldots, A_m : B_m\}\ u$
  where $B_1, \ldots, B_m$ are used for restricting instantiations of variables occurring in $A_1, \ldots, A_m$

Cardinality rules

- Syntax

$$A_0 \leftarrow l\ \{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$$

- Informal meaning If at least l elements of the "body" are true in an answer set, then add A_0 to the answer set
  - ➥ l is a lower bound on the "body"
- Example Program $\Pi = \{\ a \leftarrow 1\{b, c\},\ \ b \leftarrow\}$ has answer set $\{a, b\}$

Embedding in normal logic programs

- Replace each cardinality rule

$$A_0 \leftarrow l\ \{A_1, \ldots, A_m\} \qquad \text{by} \qquad A_0 \leftarrow cc(A_1, l)$$

where atom $cc(A_i, j)$ represents the fact that at least j of the atoms in $\{A_i, \ldots, A_m\}$, that is, of the atoms that have an equal or greater index than i, are in a particular answer set

- The definition of $cc(A_i, j)$ is given by the rules

$$\begin{aligned} cc(A_i, j{+}1) &\leftarrow cc(A_{i+1}, j), A_i \\ cc(A_i, j) &\leftarrow cc(A_{i+1}, j) \\ cc(A_{m+1}, 0) &\leftarrow \end{aligned}$$

Embedding in normal logic programs

- Replace each cardinality rule

$$A_0 \leftarrow l\ \{A_1, \ldots, A_m\} \qquad \text{by} \qquad A_0 \leftarrow cc(A_1, l)$$

  where atom $cc(A_i, j)$ represents the fact that at least j of the atoms in $\{A_i, \ldots, A_m\}$, that is, of the atoms that have an equal or greater index than i, are in a particular answer set

- The definition of $cc(A_i, j)$ is given by the rules

$$\begin{aligned} cc(A_i, j{+}1) &\leftarrow cc(A_{i+1}, j), A_i \\ cc(A_i, j) &\leftarrow cc(A_{i+1}, j) \\ cc(A_{m+1}, 0) &\leftarrow \end{aligned}$$

. . . and vice versa

- A normal rule

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

can be represented by the cardinality rule

$$A_0 \leftarrow n\ \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\}$$

Cardinality rules with upper bounds

- A rule of the form

$$A_0 \leftarrow l\ \{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}\ u$$

stands for

$$\begin{aligned} A_0 &\leftarrow B, not\ C \\ B &\leftarrow l\ \{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\} \\ C &\leftarrow u{+}1\ \{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\} \end{aligned}$$

Cardinality constraints as heads

- A rule of the form

$$I\ \{A_1, \ldots, A_m\}\ u \leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o,$$

stands for

$$\begin{aligned} B &\leftarrow A_{m+1}, \ldots, A_n, \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ A_o \\ \{A_1, \ldots, A_m\} &\leftarrow B \\ C &\leftarrow I\ \{A_1, \ldots, A_m\}\ u \\ &\leftarrow B, \mathit{not}\ C \end{aligned}$$

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

stands for $0 \leq i \leq n$

$$\begin{aligned} B_i &\leftarrow l_i\ S_i \\ C_i &\leftarrow u_i{+}1\ S_i \\ A &\leftarrow B_1, \ldots, B_n, \mathit{not}\ C_1, \ldots, \mathit{not}\ C_n \\ &\leftarrow A, \mathit{not}\ B_0 \\ &\leftarrow A, C_0 \\ S_0 \cap \mathcal{A} &\leftarrow A \end{aligned}$$

where $\mathcal{A}$ is the underlying alphabet

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

stands for $0 \leq i \leq n$

$$\begin{aligned} B_i &\leftarrow l_i\ S_i \\ C_i &\leftarrow u_i{+}1\ S_i \\ A &\leftarrow B_1, \ldots, B_n, \mathit{not}\ C_1, \ldots, \mathit{not}\ C_n \\ &\leftarrow A, \mathit{not}\ B_0 \\ &\leftarrow A, C_0 \\ S_0 \cap \mathcal{A} &\leftarrow A \end{aligned}$$

where $\mathcal{A}$ is the underlying alphabet

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Weight constraints

- Syntax $l\ [A_1 = w_1, \ldots, A_m = w_m,$
  $not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n]\ u$
- Informal meaning A weight constraint is satisfied in an answer set X, if

$$l \leq \left(\sum_{1 \leq i \leq m, A_i \in X} w_i + \sum_{m < i \leq n, A_i \notin X} w_i \right) \leq u$$

  ➥ Generalization of cardinality constraints
- Example 80 [hd(a)=50,. . . ,hd(m)=100] 400

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Optimization statements

- Idea Compute optimal answer sets by minimizing (or maximizing) a weighted sum of given elements
- Syntax
  - $\#minimize\ [A_1 = w_1, \ldots, A_m = w_m, not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n]$
  - $\#maximize\ [A_1 = w_1, \ldots, A_m = w_m, not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n]$
- Example
  - #minimize [hd(a)=30,. . . ,hd(m)=50]
  - #minimize [road(X,Y) : length(X,Y,L) = L]

- Multi-criteria optimization can be accomplished by adding priority levels to weighted literals, that is, by replacing $L_i = w_i$ by $L_i = w_i@P_i$

Optimization statements

- Idea Compute optimal answer sets by minimizing (or maximizing) a weighted sum of given elements
- Syntax
  - $\#minimize\ [A_1 = w_1, \ldots, A_m = w_m, not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n]$
  - $\#maximize\ [A_1 = w_1, \ldots, A_m = w_m, not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n]$
- Example
  - #minimize [hd(a)=30,. . . ,hd(m)=50]
  - #minimize [road(X,Y) : length(X,Y,L) = L]
- Multi-criteria optimization can be accomplished by adding priority levels to weighted literals, that is, by replacing $L_i = w_i$ by $L_i = w_i@P_i$

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

Conditional literals

- Idea Encode the contents of a (multi-)set without enumerating its elements
- Syntax $A_0 : A_1 : \ldots : A_m : \mathit{not}\ A_{m+1} : \ldots : \mathit{not}\ A_n$
- Informal meaning List all ground instances of A_0 such that corresponding instances of $A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ are true
- Example Given ‘`p(1). p(2). p(3). q(2).`’ the choice

  ```
  {r(X) : p(X) : not q(X)}.
  ```

  is instantiated to

  ```
  {r(1), r(3)}.
  ```

Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format

smodels format

- Logic programs in *smodels* format consist of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - optimization statements
- Such a format is obtained by grounders *lparse* and `gringo`

Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Motivation

- Goal Approach to computing answer sets of logic programs, based on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Checking (SAT)
- Idea View inferences in ASP as unit propagation on nogoods
- Benefits
  - A uniform constraint-based framework for different kinds of inferences in ASP
  - Advanced techniques from the areas of CP and SAT
  - Highly competitive implementation

Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Assignments

- An assignment A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

of signed literals σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.

  - ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.

- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \quad \text{and} \quad A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}\,.$$

Assignments

- An assignment A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

  $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.
  - ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \text{ and } A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}\,.$$

Assignments

- An assignment A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

  $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.
  - ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \text{ and } A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}\,.$$

Assignments

- An assignment A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

  $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.
  - ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \text{ and } A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}\,.$$

Assignments

- An assignment A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

  $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.
  - ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \text{ and } A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}\,.$$

Assignments

- An assignment A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

  $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.
  - ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \quad \text{and} \quad A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}\ .$$

Nogoods, Solutions, and Unit Propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$.
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$.
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$.
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

Nogoods, Solutions, and Unit Propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$.
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$.
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$.
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

Nogoods, Solutions, and Unit Propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$.
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$.
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$.
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

Nogoods, Solutions, and Unit Propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$.
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$.
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$.
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Nogoods from logic programs

via program completion

The completion of a logic program Π can be defined as follows:

$$\{p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \mid \beta \in body(\Pi), \beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}\}$$

$$\cup \quad \{p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k} \mid p \in atom(\Pi), body(p) = \{\beta_1, \ldots, \beta_k\}\} \ ,$$

where $body(p) = \{body(r) \mid r \in \Pi, head(r) = p\}$.

Nogoods from logic programs

via program completion

Let $\beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ be a body.

The equivalence

$$p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

can be decomposed into two implications.

1. We get

$$p_\beta \rightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

which is equivalent to the conjunction of

$$\neg p_\beta \vee p_1,\ \ldots,\ \neg p_\beta \vee p_m,\ \neg p_\beta \vee \neg p_{m+1},\ \ldots,\ \neg p_\beta \vee \neg p_n$$

This set of clauses expresses the following set of nogoods:

$$\Delta(\beta) = \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}$$

Nogoods from logic programs

via program completion

Let $\beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ be a body.

The equivalence

$$p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

can be decomposed into two implications.

1. We get

   $$p_\beta \rightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

   which is equivalent to the conjunction of

   $$\neg p_\beta \vee p_1,\ \ldots,\ \neg p_\beta \vee p_m,\ \neg p_\beta \vee \neg p_{m+1},\ \ldots,\ \neg p_\beta \vee \neg p_n$$

   This set of clauses expresses the following set of nogoods:

   $$\Delta(\beta) = \{\, \{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\} \,\}$$

Nogoods from logic programs

via program completion

Let $\beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ be a body.

The equivalence

$$p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

can be decomposed into two implications.

2. The converse of the previous implication, viz.

   $$p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \rightarrow p_\beta$$

   gives rise to the nogood

   $$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\}$$

   Intuitively, $\delta(\beta)$ is a constraint enforcing the truth of body β, or the falsity of a contained literal.

Nogoods from logic programs

via program completion

Let $\beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ be a body.

The equivalence

$$p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

can be decomposed into two implications.

2. The converse of the previous implication, viz.

   $$p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \rightarrow p_\beta$$

   gives rise to the nogood

   $$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\}$$

   Intuitively, $\delta(\beta)$ is a constraint enforcing the truth of body β, or the falsity of a contained literal.

Nogoods from logic programs

via program completion

Proceeding analogously with the atom-based equivalences, viz.

$$p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k}$$

we obtain for an atom $p \in \mathit{atom}(\Pi)$ along with its bodies $\mathit{body}(p) = \{\beta_1, \ldots, \beta_k\}$ the nogoods

$$\Delta(p) = \{\, \{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\} \,\} \text{ and } \delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}\,.$$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

$\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and

$\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$not\ z$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

$\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and

$\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{|lcl|} \hline x & \leftarrow & y \\ x & \leftarrow & not\ z \\ \hline \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$not\ z$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$not\ z$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{|lcl|} \hline x & \leftarrow & y \\ x & \leftarrow & not\ z \\ \hline \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{|lcl|} \hline x & \leftarrow & y \\ x & \leftarrow & not\ z \\ \hline \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{|lcl|} \hline x & \leftarrow & y \\ x & \leftarrow & not\ z \\ \hline \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\} \ . \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}$$

$$\begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs

body-oriented nogoods

For a body $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, recall that

$$\begin{aligned} \delta(\beta) &= \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \\ \Delta(\beta) &= \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}\ . \end{aligned}$$

For example, for body $\{x, not\ y\}$, we obtain

$$\begin{array}{l} \ldots \leftarrow x, not\ y \\ \qquad\vdots \\ \ldots \leftarrow x, not\ y \end{array}$$

$$\begin{aligned} \delta(\{x, not\ y\}) &= \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, not\ y\}) &= \{\{\mathbf{T}\{x, not\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, not\ y\}, \mathbf{T}y\}\} \end{aligned}$$

For nogood $\delta(\{x, not\ y\}) = \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, not\ y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, not\ y\}, \mathbf{T}x)$.

Nogoods from logic programs

body-oriented nogoods

For a body $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, recall that

$$\begin{aligned} \delta(\beta) &= \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \\ \Delta(\beta) &= \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}\ . \end{aligned}$$

For example, for body $\{x, not\ y\}$, we obtain

$$\begin{array}{|c|}\hline \ldots \leftarrow x, not\ y \\ \vdots \\ \ldots \leftarrow x, not\ y \\ \hline \end{array}$$

$$\begin{aligned} \delta(\{x, not\ y\}) &= \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, not\ y\}) &= \{\{\mathbf{T}\{x, not\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, not\ y\}, \mathbf{T}y\}\} \end{aligned}$$

For nogood $\delta(\{x, not\ y\}) = \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, not\ y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, not\ y\}, \mathbf{T}x)$.

Nogoods from logic programs

body-oriented nogoods

For a body $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, recall that

$$\begin{aligned} \delta(\beta) &= \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \\ \Delta(\beta) &= \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}\ . \end{aligned}$$

For example, for body $\{x, not\ y\}$, we obtain

$$\begin{array}{|c|}\hline \ldots \leftarrow x, not\ y \\ \vdots \\ \ldots \leftarrow x, not\ y \\ \hline \end{array}$$

$$\begin{aligned} \delta(\{x, not\ y\}) &= \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, not\ y\}) &= \{\{\mathbf{T}\{x, not\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, not\ y\}, \mathbf{T}y\}\} \end{aligned}$$

For nogood $\delta(\{x, not\ y\}) = \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, not\ y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, not\ y\}, \mathbf{T}x)$.

Nogoods from logic programs

body-oriented nogoods

For a body $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, recall that

$$\begin{aligned} \delta(\beta) &= \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \\ \Delta(\beta) &= \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}\ . \end{aligned}$$

For example, for body $\{x, not\ y\}$, we obtain

$$\begin{array}{|c|}\hline \ldots \leftarrow x, not\ y \\ \vdots \\ \ldots \leftarrow x, not\ y \\ \hline \end{array}$$

$$\begin{aligned} \delta(\{x, not\ y\}) &= \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, not\ y\}) &= \{\{\mathbf{T}\{x, not\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, not\ y\}, \mathbf{T}y\}\} \end{aligned}$$

For nogood $\delta(\{x, not\ y\}) = \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, not\ y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, not\ y\}, \mathbf{T}x)$.

Nogoods from logic programs

body-oriented nogoods

For a body $\beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$, recall that

$$\begin{aligned} \delta(\beta) &= \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \\ \Delta(\beta) &= \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}\ . \end{aligned}$$

For example, for body $\{x, \mathit{not}\ y\}$, we obtain

$$\begin{array}{|c|} \hline \ldots \leftarrow x, \mathit{not}\ y \\ \vdots \\ \ldots \leftarrow x, \mathit{not}\ y \\ \hline \end{array}$$

$$\begin{aligned} \delta(\{x, \mathit{not}\ y\}) &= \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, \mathit{not}\ y\}) &= \{\{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T}y\}\} \end{aligned}$$

For nogood $\delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, \mathit{not}\ y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x)$.

Characterization of answer sets

for tight logic programs

Let Π be a logic program and

$$\begin{aligned}\Delta_\Pi &= \{\delta(p) \mid p \in \mathit{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \mathit{atom}(\Pi)\} \\ &\cup \{\delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \mathit{body}(\Pi)\}\ .\end{aligned}$$

Theorem

Let Π be a tight logic program. Then,
$X \subseteq \mathit{atom}(\Pi)$ is an answer set of Π iff
$X = A^{\mathsf{T}} \cap \mathit{atom}(\Pi)$ for a (unique) solution A for Δ_Π.

Characterization of answer sets

for tight logic programs

Let Π be a logic program and

$$\begin{aligned}\Delta_\Pi &= \{\delta(p) \mid p \in \mathit{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \mathit{atom}(\Pi)\} \\ &\cup \{\delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \mathit{body}(\Pi)\}\ .\end{aligned}$$

Theorem

Let Π be a tight logic program. Then,
$X \subseteq \mathit{atom}(\Pi)$ is an answer set of Π iff
$X = A^{\mathsf{T}} \cap \mathit{atom}(\Pi)$ for a (unique) solution A for Δ_Π.

Characterization of answer sets

for tight logic programs, ie. free of positive recursion

Let Π be a logic program and

$$\begin{aligned}\Delta_\Pi &= \{\delta(p) \mid p \in \mathit{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \mathit{atom}(\Pi)\} \\ &\cup \{\delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \mathit{body}(\Pi)\}\ .\end{aligned}$$

Theorem

Let Π be a tight logic program. Then,
$X \subseteq \mathit{atom}(\Pi)$ is an answer set of Π iff
$X = A^{\mathsf{T}} \cap \mathit{atom}(\Pi)$ for a (unique) solution A for Δ_Π.

Nogoods from logic programs

via loop formulas

Let Π be a normal logic program and recall that:

- For $L \subseteq atom(\Pi)$, the external supports of L for Π are
$$ES_\Pi(L) = \{r \in \Pi \mid head(r) \in L, body(r)^+ \cap L = \emptyset\}.$$
- The (disjunctive) loop formula of L for Π is
$$LF_\Pi(L) = (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_\Pi(L)} body(r))$$
$$\equiv (\bigwedge_{r \in ES_\Pi(L)} \neg body(r)) \rightarrow (\bigwedge_{A \in L} \neg A)$$
  - ☞ The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported.
- The external bodies of L for Π are
$$EB(L) = \{body(r) \mid r \in ES_\Pi(L)\}$$

Nogoods from logic programs

via loop formulas

Let Π be a normal logic program and recall that:

- For $L \subseteq atom(\Pi)$, the external supports of L for Π are
$$ES_\Pi(L) \quad = \quad \{r \in \Pi \mid head(r) \in L, body(r)^+ \cap L = \emptyset\}.$$
- The (disjunctive) loop formula of L for Π is
$$\begin{aligned} LF_\Pi(L) &= \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_\Pi(L)} body(r)\right) \\ &\equiv \left(\bigwedge_{r \in ES_\Pi(L)} \neg body(r)\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right) \end{aligned}$$
  - ☞ The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported.
- The external bodies of L for Π are
$$EB(L) \quad = \quad \{body(r) \mid r \in ES_\Pi(L)\}$$

Nogoods from logic programs

via loop formulas

Let Π be a normal logic program and recall that:

- For $L \subseteq atom(\Pi)$, the external supports of L for Π are
$$ES_\Pi(L) \quad = \quad \{r \in \Pi \mid head(r) \in L, body(r)^+ \cap L = \emptyset\}.$$
- The (disjunctive) loop formula of L for Π is
$$\begin{aligned} LF_\Pi(L) &= \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_\Pi(L)} body(r)\right) \\ &\equiv \left(\bigwedge_{r \in ES_\Pi(L)} \neg body(r)\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right) \end{aligned}$$
  - ☞ The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported.
- The external bodies of L for Π are
$$EB(L) \quad = \quad \{body(r) \mid r \in ES_\Pi(L)\}$$

Nogoods from logic programs

loop nogoods

- For a logic program Π and some $\emptyset \subset U \subseteq atom(\Pi)$, define the loop nogood of an atom $p \in U$ as

$$\lambda(p, U) \quad = \quad \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}$$

  where $EB(U) = \{\beta_1, \ldots, \beta_k\}$.
- In all, we get the following set of loop nogoods for Π:

$$\Lambda_\Pi \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(\Pi)} \{\lambda(p, U) \mid p \in U\}$$

- The set Λ_Π of loop nogoods denies cyclic support among *true* atoms.

Nogoods from logic programs

loop nogoods

- For a logic program Π and some $\emptyset \subset U \subseteq atom(\Pi)$, define the loop nogood of an atom $p \in U$ as

$$\lambda(p, U) \quad = \quad \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}$$

  where $EB(U) = \{\beta_1, \ldots, \beta_k\}$.
- In all, we get the following set of loop nogoods for Π:

$$\Lambda_\Pi \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(\Pi)} \{\lambda(p, U) \mid p \in U\}$$

- The set Λ_Π of loop nogoods denies cyclic support among *true* atoms.

Nogoods from logic programs

loop nogoods

- For a logic program Π and some $\emptyset \subset U \subseteq atom(\Pi)$, define the loop nogood of an atom $p \in U$ as

$$\lambda(p, U) \quad = \quad \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}$$

  where $EB(U) = \{\beta_1, \ldots, \beta_k\}$.
- In all, we get the following set of loop nogoods for Π:

$$\Lambda_\Pi \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(\Pi)} \{\lambda(p, U) \mid p \in U\}$$

- The set Λ_Π of loop nogoods denies cyclic support among *true* atoms.

Example

Consider

$$\Pi = \left\{ \begin{array}{ll} x \leftarrow not\ y & u \leftarrow x \\ y \leftarrow not\ x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Example

Consider

$$\Pi = \left\{ \begin{array}{ll} x \leftarrow not\ y & u \leftarrow x \\ y \leftarrow not\ x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Example

Consider

$$\Pi = \left\{ \begin{array}{ll} x \leftarrow not\ y & u \leftarrow x \\ y \leftarrow not\ x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Characterization of answer sets

Theorem

Let Π be a logic program. Then,
$X \subseteq atom(\Pi)$ is an answer set of Π iff
$X = A^{\mathsf{T}} \cap atom(\Pi)$ for a (unique) solution A for $\Delta_\Pi \cup \Lambda_\Pi$.

Some remarks

- Nogoods in Λ_Π augment Δ_Π with conditions checking for unfounded sets, in particular, those being loops.
- While $|\Delta_\Pi|$ is linear in the size of Π, Λ_Π may contain exponentially many (non-redundant) loop nogoods

Characterization of answer sets

Theorem

Let Π *be a logic program. Then,*
$X \subseteq atom(\Pi)$ *is an answer set of* Π *iff*
$X = A^{\mathsf{T}} \cap atom(\Pi)$ *for a (unique) solution* A *for* $\Delta_\Pi \cup \Lambda_\Pi$.

Some remarks

- Nogoods in Λ_Π augment Δ_Π with conditions checking for unfounded sets, in particular, those being loops.
- While $|\Delta_\Pi|$ is linear in the size of Π, Λ_Π may contain exponentially many (non-redundant) loop nogoods

Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

- Traditional DPLL-style approach
  - (Unit) propagation
  - (Chronological) backtracking
- Modern CDCL-style approach
  - (Unit) propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion

DPLL-style solving

loop

propagate // compute deterministic consequences

if no conflict **then**

if all variables assigned **then return** variable assignment

else *decide* // non-deterministically assign some literal

else

if top-level conflict **then return** unsatisfiable

else

backtrack // undo assignments made after last decision

flip // assign complement of last decision literal

CDCL-style solving

loop

propagate // compute deterministic consequences

if no conflict **then**

if all variables assigned **then return** variable assignment

else *decide* // non-deterministically assign some literal

else

if top-level conflict **then return** unsatisfiable

else

analyze // analyze conflict and add a conflict constraint

backjump // undo assignments until conflict constraint is unit

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion $[\Delta_\Pi]$
  - Loop nogoods, determined and recorded on demand $[\Lambda_\Pi]$
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding an answer set (a solution for $\Delta_\Pi \cup \Lambda_\Pi$)
  - Deriving a conflict independently of (heuristic) choices

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion $[\Delta_\Pi]$
  - Loop nogoods, determined and recorded on demand $[\Lambda_\Pi]$
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding an answer set (a solution for $\Delta_\Pi \cup \Lambda_\Pi$)
  - Deriving a conflict independently of (heuristic) choices

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion $[\Delta_\Pi]$
  - Loop nogoods, determined and recorded on demand $[\Lambda_\Pi]$
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding an answer set (a solution for $\Delta_\Pi \cup \Lambda_\Pi$)
  - Deriving a conflict independently of (heuristic) choices

Algorithm 1: CDNL-ASP

Input : A logic program Π.
Output : An answer set of Π or "no answer set".

$A \leftarrow \emptyset$ // *assignment over* $atom(\Pi) \cup body(\Pi)$
$\nabla \leftarrow \emptyset$ // *set of (dynamic) nogoods*
$dl \leftarrow 0$ // *decision level*
loop
  $(A, \nabla) \leftarrow \text{NogoodPropagation}(\Pi, \nabla, A)$
  if $\varepsilon \subseteq A$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ **then**
    if $dl = 0$ **then return** no answer set
    $(\delta, k) \leftarrow \text{ConflictAnalysis}(\varepsilon, \Pi, \nabla, A)$
    $\nabla \leftarrow \nabla \cup \{\delta\}$ // *learning*
    $A \leftarrow (A \setminus \{\sigma \in A \mid k < dl(\sigma)\})$ // *backjumping*
    $dl \leftarrow k$
  else if $A^{\mathbf{T}} \cup A^{\mathbf{F}} = atom(\Pi) \cup body(\Pi)$ **then**
    return $A^{\mathbf{T}} \cap atom(\Pi)$ // *answer set*
  else
    $\sigma_d \leftarrow \text{Select}(\Pi, \nabla, A)$ // *heuristic choice of* $\sigma_d \notin A$
    $dl \leftarrow dl + 1$
    $A \leftarrow A \circ (\sigma_d)$ // $dl(\sigma_d) = dl$

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A.
- For a heuristically chosen literal $\sigma_d = \mathbf{T}p$ or $\sigma_d = \mathbf{F}p$, respectively, we require $p \in (atom(\Pi) \cup body(\Pi)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$.
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned.
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_\Pi \cup \nabla$.
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of answer sets.
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$.
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation.
  - No explicit flipping of heuristically chosen literals !

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A.
- For a heuristically chosen literal $\sigma_d = \mathbf{T}p$ or $\sigma_d = \mathbf{F}p$, respectively, we require $p \in (atom(\Pi) \cup body(\Pi)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$.
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned.
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_\Pi \cup \nabla$.
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of answer sets.
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$.
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation.
  - No explicit flipping of heuristically chosen literals !

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A.
- For a heuristically chosen literal $\sigma_d = \mathbf{T}p$ or $\sigma_d = \mathbf{F}p$, respectively, we require $p \in (atom(\Pi) \cup body(\Pi)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$.
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned.
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_\Pi \cup \nabla$.
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of answer sets.
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$.
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation.
  - No explicit flipping of heuristically chosen literals !

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{not\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{not\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{not\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{not\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_Π and ∇;
  - Unfounded sets $U \subseteq atom(\Pi)$.
- Note that U is unfounded if $EB(U) \subseteq A^{\mathbf{F}}$.
  - ☞ For any $p \in U$, we have $(\lambda(p, U) \setminus \{\mathbf{T}p\}) \subseteq A$.
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}}) \ .$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of Π.
  - ➥ Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $p \in U$, adding $\lambda(p, U)$ to ∇ triggers a conflict or further derivations by unit propagation.
  - ☞ Add loop nogoods atom by atom to eventually falsify all $p \in U$.

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_Π and ∇;
  - Unfounded sets $U \subseteq atom(\Pi)$.
- Note that U is unfounded if $EB(U) \subseteq A^{\mathbf{F}}$.
  - ☞ For any $p \in U$, we have $(\lambda(p, U) \setminus \{\mathbf{T}p\}) \subseteq A$.
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}}) \ .$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of Π.
  - ➥ Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $p \in U$, adding $\lambda(p, U)$ to ∇ triggers a conflict or further derivations by unit propagation.
  - ☞ Add loop nogoods atom by atom to eventually falsify all $p \in U$.

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_Π and ∇;
  - Unfounded sets $U \subseteq atom(\Pi)$.
- Note that U is unfounded if $EB(U) \subseteq A^{\mathbf{F}}$.
  - ☞ For any $p \in U$, we have $(\lambda(p, U) \setminus \{\mathbf{T}p\}) \subseteq A$.
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}}) \ .$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of Π.
  - ➥ Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $p \in U$, adding $\lambda(p, U)$ to ∇ triggers a conflict or further derivations by unit propagation.
  - ☞ Add loop nogoods atom by atom to eventually falsify all $p \in U$.

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_Π and ∇;
  - Unfounded sets $U \subseteq atom(\Pi)$.
- Note that U is unfounded if $EB(U) \subseteq A^{\mathbf{F}}$.
  - ☞ For any $p \in U$, we have $(\lambda(p, U) \setminus \{\mathbf{T}p\}) \subseteq A$.
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}}) \ .$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of Π.
  - ➥ Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $p \in U$, adding $\lambda(p, U)$ to ∇ triggers a conflict or further derivations by unit propagation.
  - ☞ Add loop nogoods atom by atom to eventually falsify all $p \in U$.

Algorithm 2: NogoodPropagation

Input : A logic program Π, a set ∇ of nogoods, and an assignment A.
Output : An extended assignment and set of nogoods.

$U \leftarrow \emptyset$ *// set of unfounded atoms*
loop
  repeat
    if $\delta \subseteq A$ for some $\delta \in \Delta_\Pi \cup \nabla$ **then return** (A, ∇) *// conflict*
    $\Sigma \leftarrow \{\delta \in \Delta_\Pi \cup \nabla \mid (\delta \setminus A) = \{\sigma\}, \overline{\sigma} \notin A\}$ *// unit-resulting nogoods*
    if $\Sigma \neq \emptyset$ **then**
      let $\sigma \in (\delta \setminus A)$ for some $\delta \in \Sigma$ **in**
        $A \leftarrow A \circ (\overline{\sigma})$ *// $dl(\overline{\sigma}) = max(\{dl(\rho) \mid \rho \in (\delta \setminus \{\sigma\})\} \cup \{0\})$*
  until $\Sigma = \emptyset$

  if Π is tight **then return** (A, ∇) *// no unfounded set $\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$*
  else
    $U \leftarrow (U \setminus A^{\mathbf{F}})$
    if $U = \emptyset$ **then** $U \leftarrow$ UnfoundedSet(Π, A)
    if $U = \emptyset$ **then return** (A, ∇)*// no unfounded set $\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$*
    let $p \in U$ **in**
      $\nabla \leftarrow \nabla \cup \{\lambda(p, U)\}$ *// record unit-resulting or violated loop nogood*

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee the following for a result U
  1. $U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$
  2. $EB(U) \subseteq A^{\mathbf{F}}$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(\Pi) \setminus A^{\mathbf{F}})$
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of Π
  - Usually, the latter option is implemented in ASP solvers

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee the following for a result U
  1. $U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$
  2. $EB(U) \subseteq A^{\mathbf{F}}$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(\Pi) \setminus A^{\mathbf{F}})$
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of Π
  - Usually, the latter option is implemented in ASP solvers

Example: NogoodPropagation

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{not\ x, not\ y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$
3	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_\Pi \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$.
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_\Pi \cup \nabla$.
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) \ .$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$.
  - Iterated resolution progresses in inverse order of assignment.
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl.
  - This literal σ is called First Unique Implication Point (First-UIP).
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl.

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_\Pi \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$.
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_\Pi \cup \nabla$.
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) \ .$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$.
  - Iterated resolution progresses in inverse order of assignment.
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl.
  - This literal σ is called First Unique Implication Point (First-UIP).
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl.

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_\Pi \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$.
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_\Pi \cup \nabla$.
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) \ .$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$.
  - Iterated resolution progresses in inverse order of assignment.
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl.
  - This literal σ is called First Unique Implication Point (First-UIP).
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl.

Algorithm 3: ConflictAnalysis

Input : A violated nogood δ, a logic program Π, a set ∇ of nogoods, and an assignment A.

Output : A derived nogood and a decision level.

loop

  let $\sigma \in \delta$ such that $(\delta \setminus A[\sigma]) = \{\sigma\}$ **in**

    $k \leftarrow max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$

    if $k = dl(\sigma)$ **then**

      let $\varepsilon \in \Delta_\Pi \cup \nabla$ such that $(\varepsilon \setminus A[\sigma]) = \{\overline{\sigma}\}$ **in**

        $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ *// resolution*

    else return (δ, k)

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{lll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y \end{array} \quad w \leftarrow not\ x, not\ y \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{lll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y \end{array} \quad w \leftarrow not\ x, not\ y \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Remarks

- There always is a First-UIP at which conflict analysis terminates.
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl.
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$.
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$.
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting.
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates.
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl.
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$.
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$.
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting.
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates.
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl.
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$.
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$.
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting.
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates.
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl.
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$.
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$.
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting.
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before,
  without explicitly flipping any heuristically chosen literal !

Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

29 Encoding Methodology
- Tweaking N-Queens
- Do's and Dont's

30 Hints

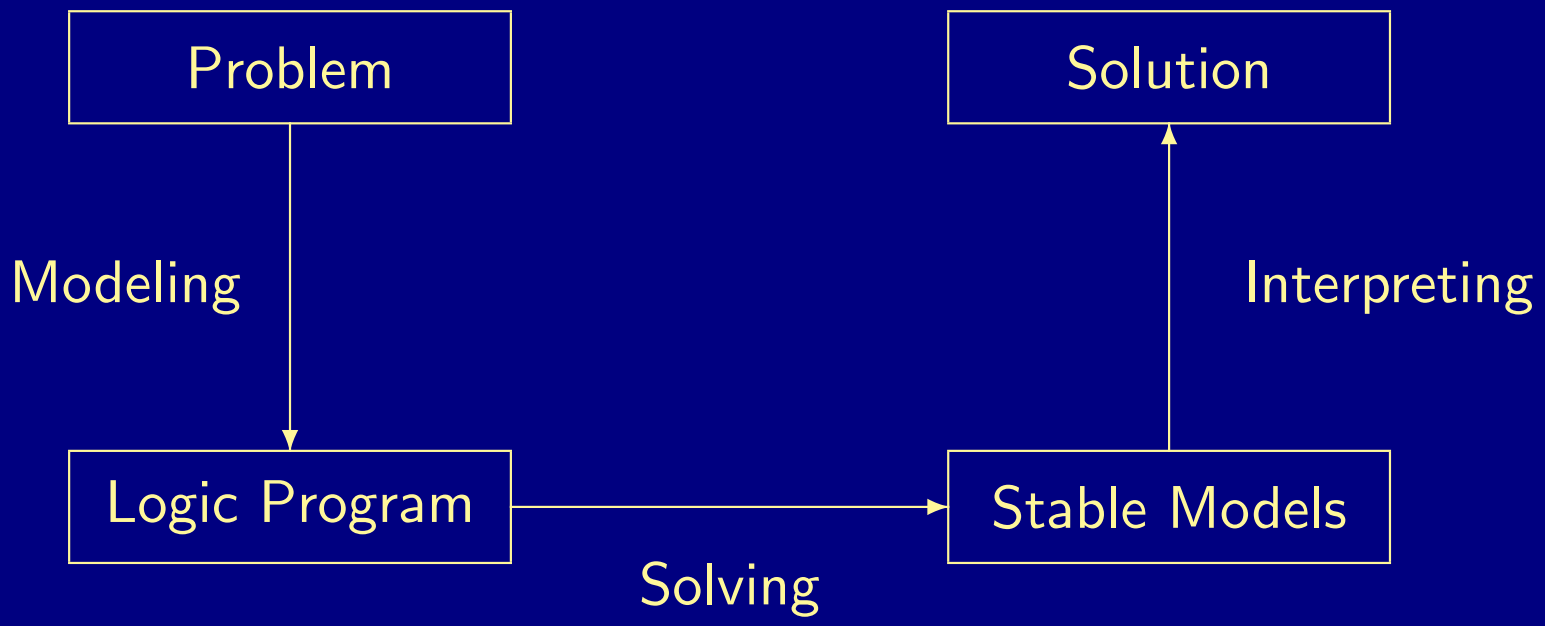
Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

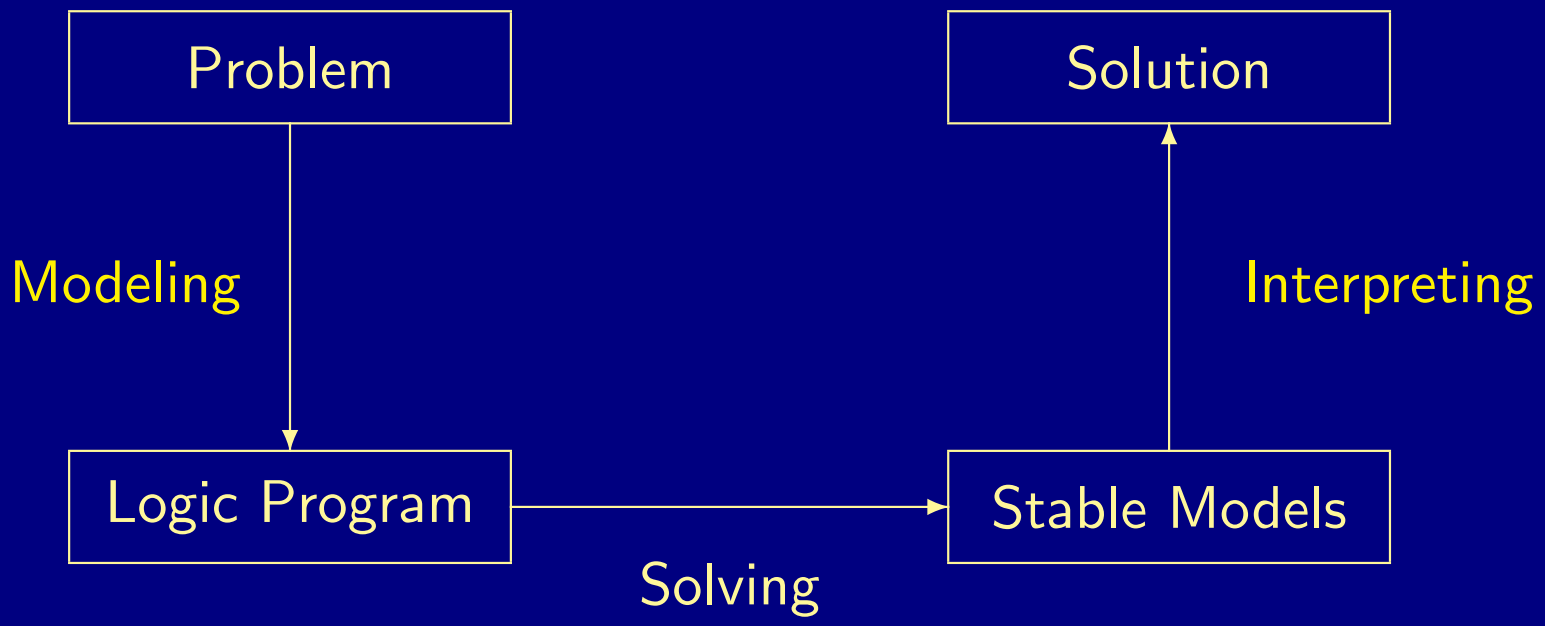
29 Encoding Methodology
- Tweaking N-Queens
- Do's and Dont's

30 Hints

Modeling and Interpreting



Modeling and Interpreting



Problems as Logic Program

For solving a problem class P for a problem instance I,
encode

1. the problem instance I as a set C(I) of facts and
2. the problem class P as a set C(P) of rules

such that the solutions to P for I can be (polynomially) extracted from the answer sets of $C(I) \cup C(P)$.

Uniform encoding

A uniform encoding C(P) is a first-order logic program, encoding the solutions to P for any set C(I) of facts.

Problems as Logic Program

For solving a problem class P for a problem instance I,
encode

1. the problem instance I as a set C(I) of facts and
2. the problem class P as a set C(P) of rules

such that the solutions to P for I can be (polynomially) extracted from the answer sets of $C(I) \cup C(P)$.

Uniform encoding

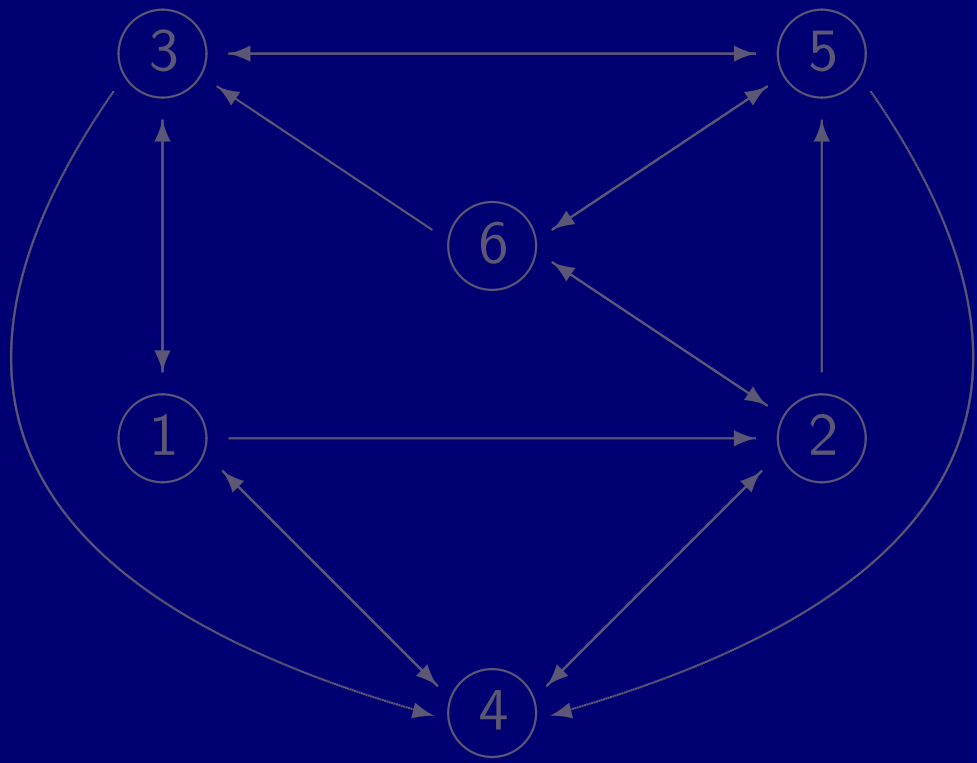
A uniform encoding C(P) is a first-order logic program, encoding the solutions to P for any set C(I) of facts.

N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

$$G = \begin{pmatrix} V = \{1,2,3,4,5,6\}, \\ E = \{(1,2),(1,3),(1,4), \\ \quad (2,4),(2,5),(2,6), \\ \quad (3,1),(3,4),(3,5), \\ \quad (4,1),(4,2), \\ \quad (5,3),(5,4),(5,6), \\ \quad (6,2),(6,3),(6,5)\} \end{pmatrix}$$

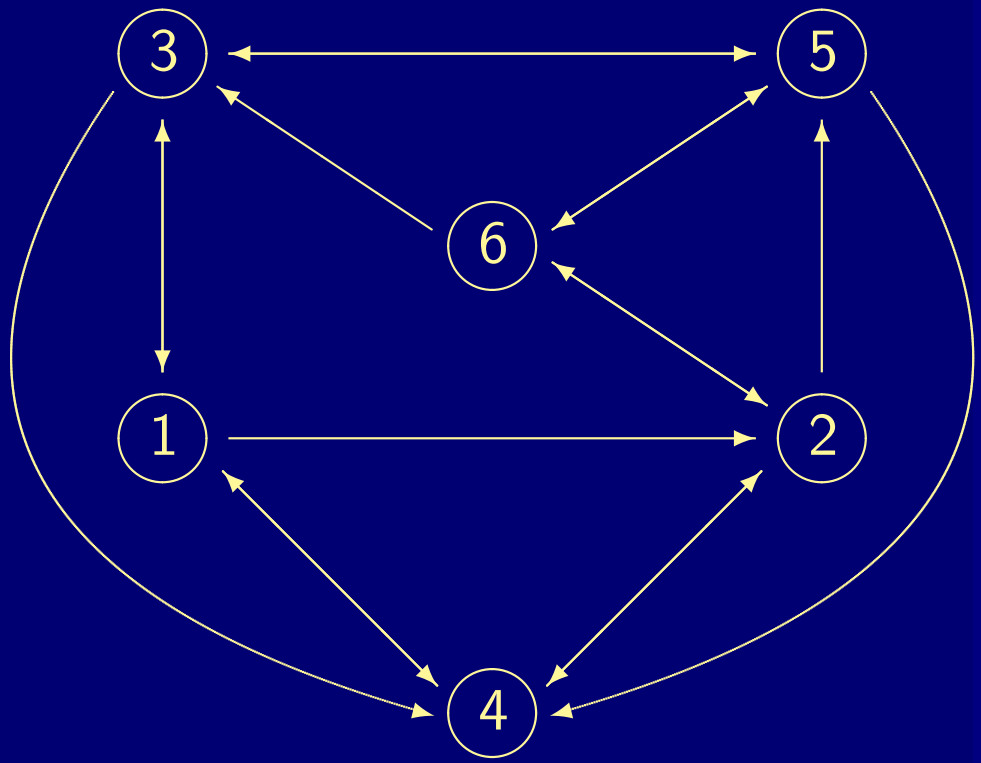


N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

$$G = \begin{pmatrix} V = \{1,2,3,4,5,6\}, \\ E = \{(1,2),(1,3),(1,4), \\ \quad (2,4),(2,5),(2,6), \\ \quad (3,1),(3,4),(3,5), \\ \quad (4,1),(4,2), \\ \quad (5,3),(5,4),(5,6), \\ \quad (6,2),(6,3),(6,5)\} \end{pmatrix}$$

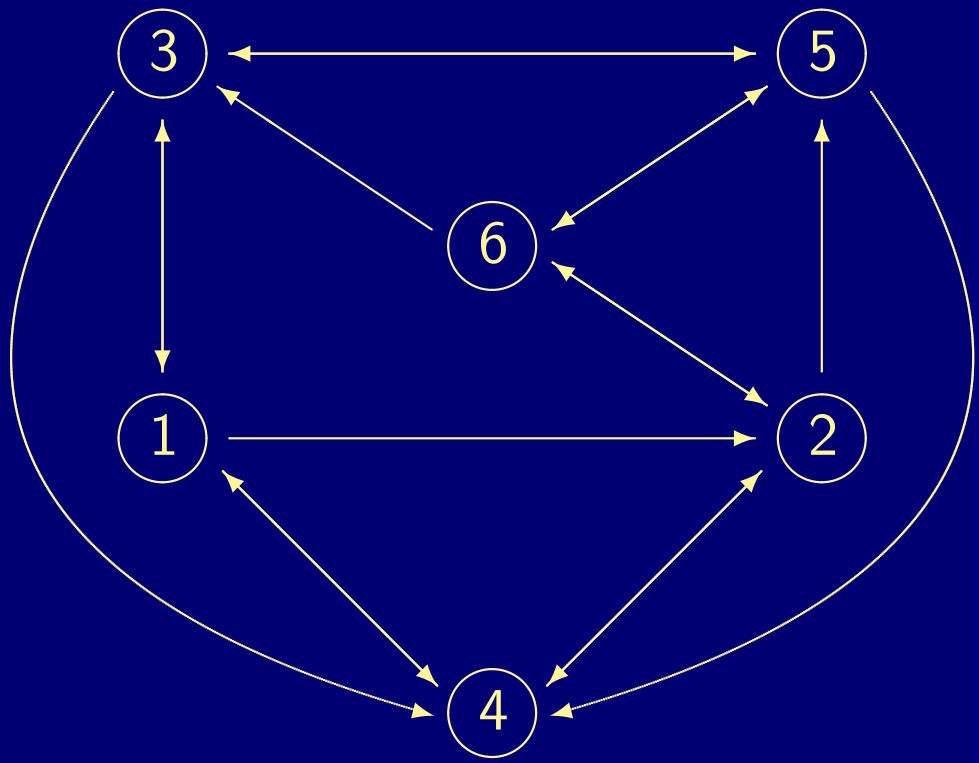


N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2). edge(5,3).
edge(5,4). edge(5,6). edge(6,2).
edge(6,3). edge(6,5).
```

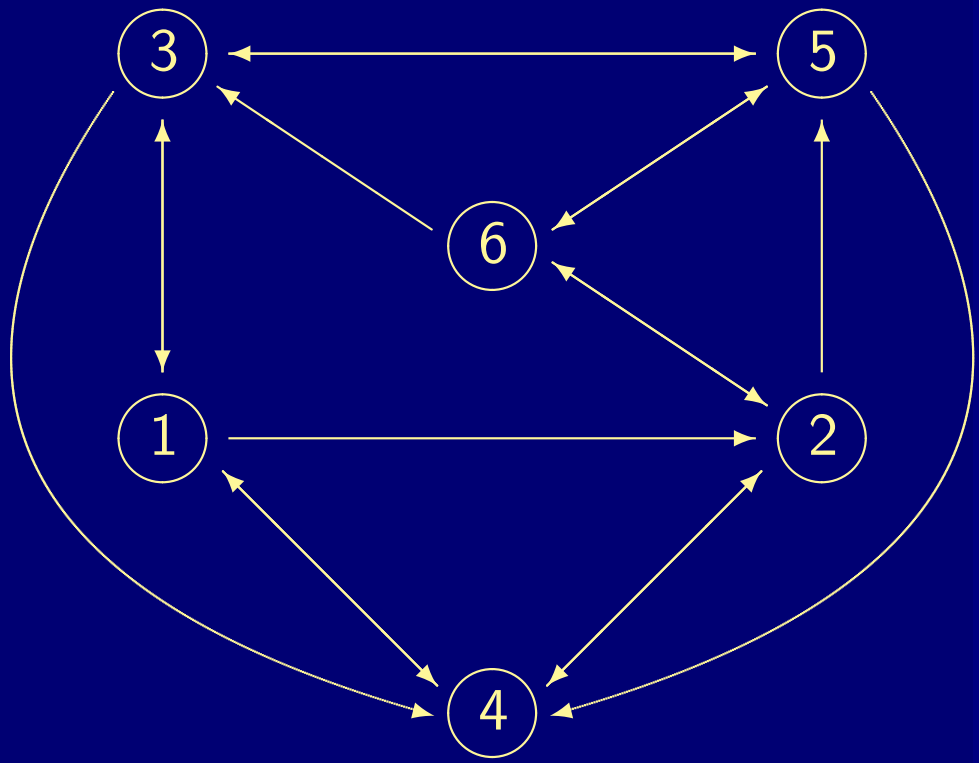


N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2). edge(5,3).
edge(5,4). edge(5,6). edge(6,2).
edge(6,3). edge(6,5).
```



N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C),
   node(X), not other(X,C).

   other(X,C) :- iscol(C),
   color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C),
      edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C), node(X), not other(X,C).

   other(X,C) :- iscol(C), color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C), edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C), node(X), not other(X,C).

   other(X,C) :- iscol(C), color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C), edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C),
   node(X), not other(X,C).

   other(X,C) :- iscol(C),
   color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C),
      edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C),
   node(X), not other(X,C).

   other(X,C) :- iscol(C),
   color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C),
      edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C),
   node(X), not other(X,C).

   other(X,C) :- iscol(C),
   color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C),
      edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   color(X,C) :- iscol(C),
   node(X), not other(X,C).

   other(X,C) :- iscol(C),
   color(X,D), D != C.
   ```
2. ```
   :- color(X,C), color(Y,C),
      edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1. ```
   1 #count{ color(X,C) :
                 iscol(C) } 1
             :- node(X).
   ```
2. ```
   :- color(X,C), color(Y,C),
      edge(X,Y).
   ```
3. ```
   #const n=3.
   iscol(1..n).
   ```
4. ```
   #hide.
   #show color/2.
   ```

N-Colorability

Recapitulation I

Instance as Facts (in `graph.lp`)

```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).
```

N-Colorability

Recapitulation II

Uniform Encoding (in `color.lp`)

```
% DOMAIN
#const n=3. iscol(1..n).

% GENERATE
1 #count{ color(X,C) : iscol(C) } 1 :- node(X).
% color(X,C) :- iscol(C), node(X), not other(X,C).
% other(X,C) :- iscol(C), color(X,D), D != C.

% TEST
:- color(X,C), color(Y,C), edge(X,Y).

% DISPLAY
#hide. #show color/2.
```

N-Colorability

Let's Run it!

```
gringo graph.lp color.lp | clasp 0
```

```
clasp version 2.0.2
Reading from stdin
Solving...
Answer: 1
color(6,2) color(5,3) color(4,2) color(3,1) color(2,1) color(1,3)
Answer: 2
color(6,1) color(5,3) color(4,1) color(3,2) color(2,2) color(1,3)
Answer: 3
color(6,3) color(5,2) color(4,3) color(3,1) color(2,1) color(1,2)
Answer: 4
color(6,1) color(5,2) color(4,1) color(3,3) color(2,3) color(1,2)
Answer: 5
color(6,3) color(5,1) color(4,3) color(3,2) color(2,2) color(1,1)
Answer: 6
color(6,2) color(5,1) color(4,2) color(3,3) color(2,3) color(1,1)
```

N-Colorability

Let's Run it!

```
gringo graph.lp color.lp | clasp 0
```

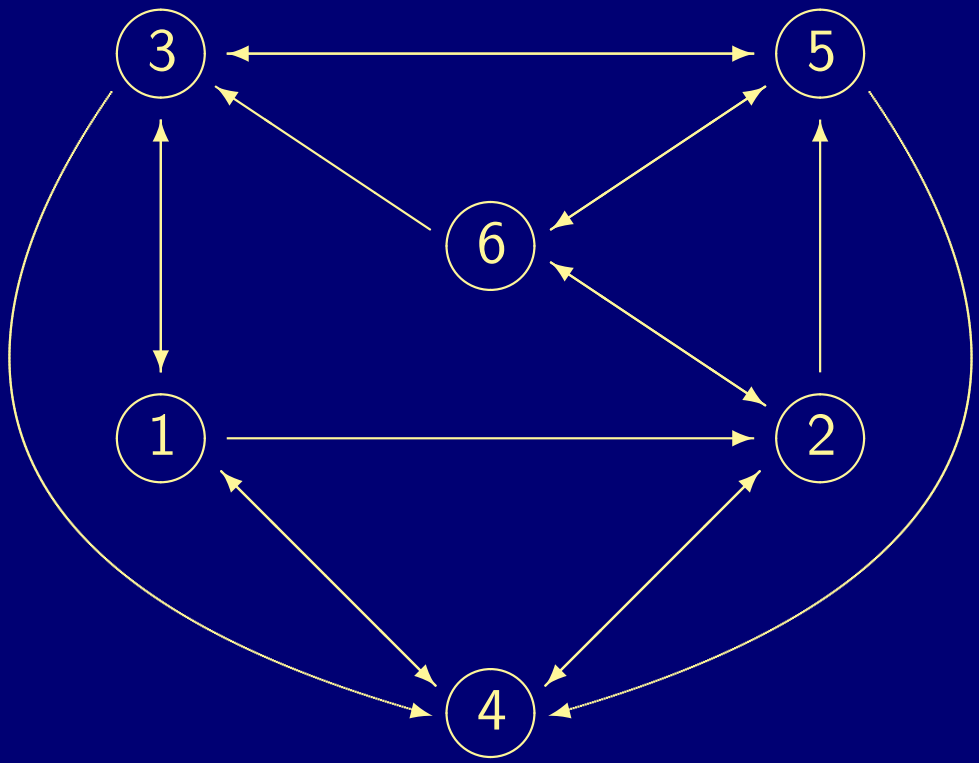
```
clasp version 2.0.2
Reading from stdin
Solving...
Answer: 1
color(6,2) color(5,3) color(4,2) color(3,1) color(2,1) color(1,3)
Answer: 2
color(6,1) color(5,3) color(4,1) color(3,2) color(2,2) color(1,3)
Answer: 3
color(6,3) color(5,2) color(4,3) color(3,1) color(2,1) color(1,2)
Answer: 4
color(6,1) color(5,2) color(4,1) color(3,3) color(2,3) color(1,2)
Answer: 5
color(6,3) color(5,1) color(4,3) color(3,2) color(2,2) color(1,1)
Answer: 6
color(6,2) color(5,1) color(4,2) color(3,3) color(2,3) color(1,1)
```

N-Colorability

Let's Interpret it!

Found: 3-coloring(s)

```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```

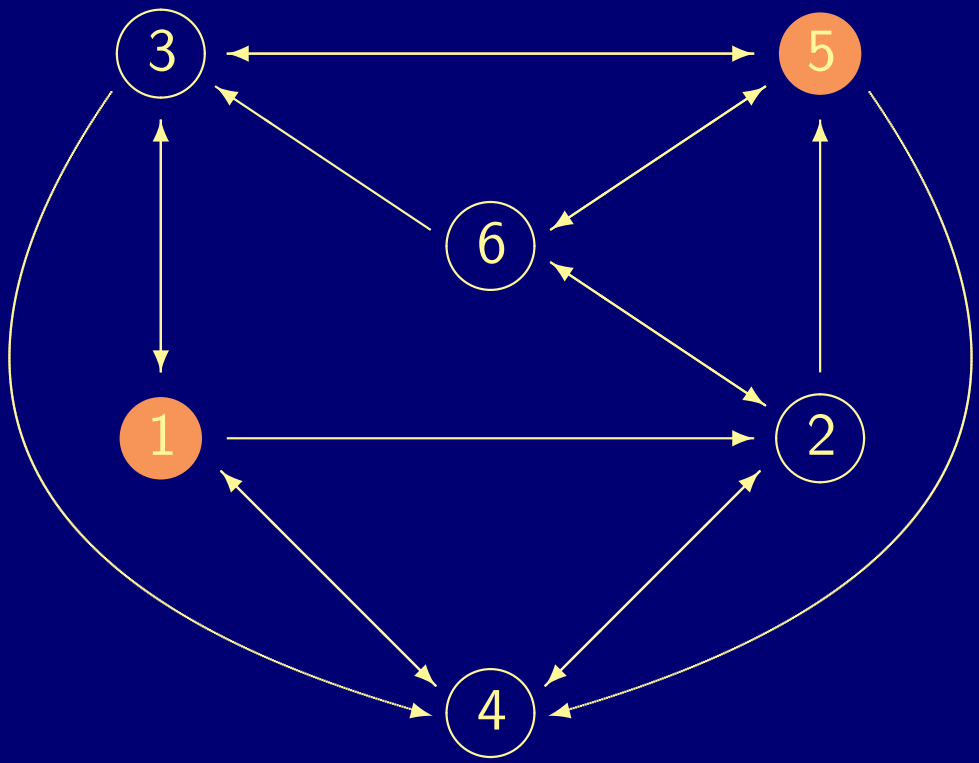


N-Colorability

Let's Interpret it!

Found: 3-coloring(s)

```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```

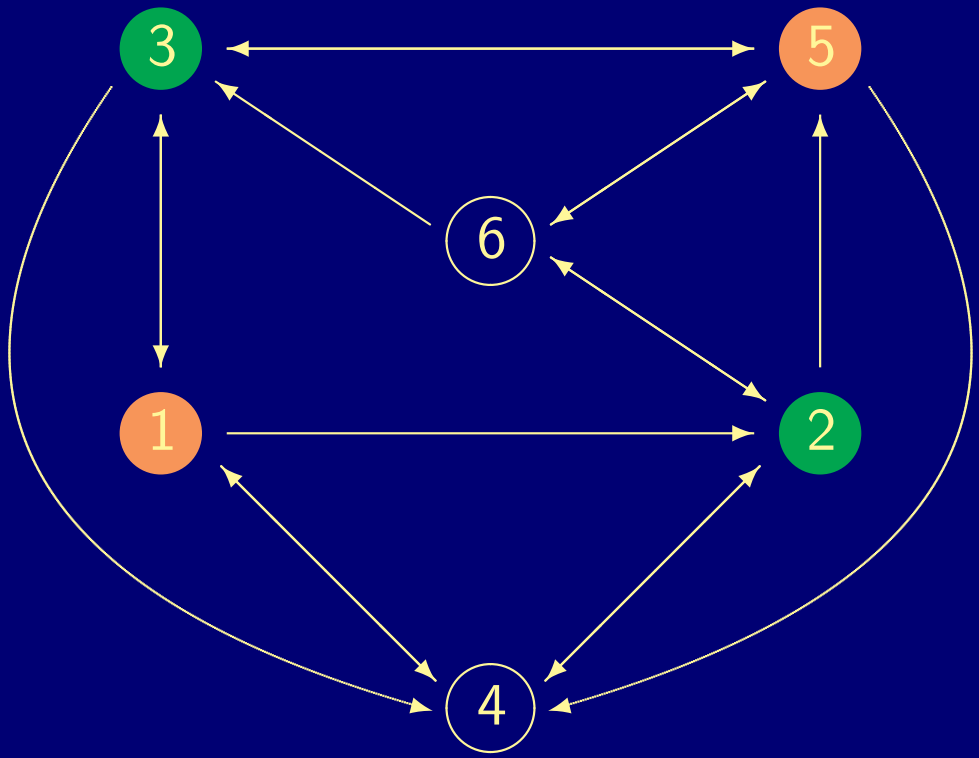


N-Colorability

Let's Interpret it!

Found: 3-coloring(s)

```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```

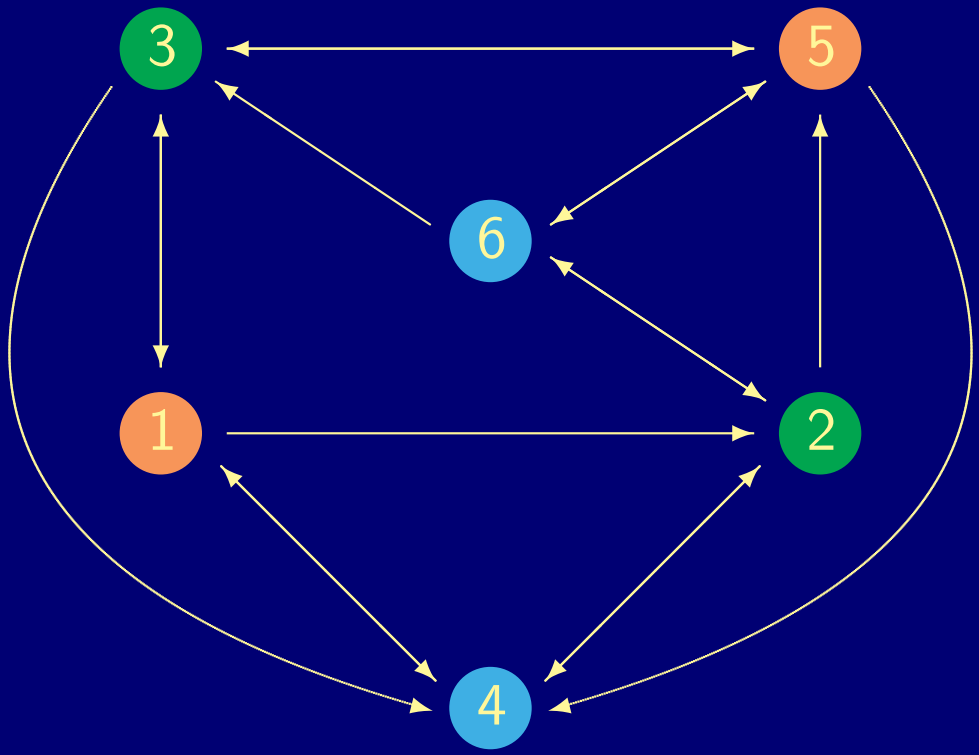


N-Colorability

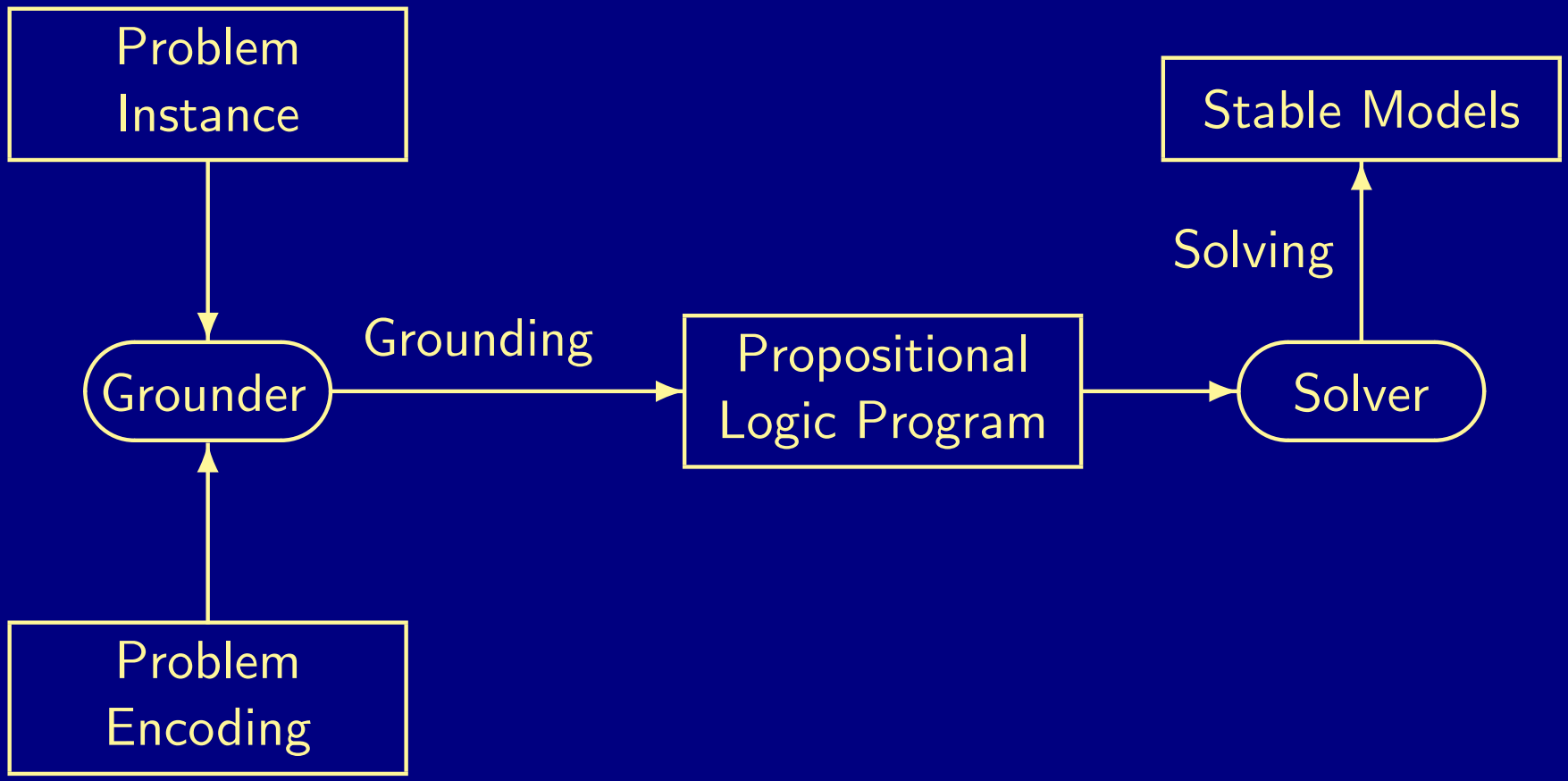
Let's Interpret it!

Found: 3-coloring(s)

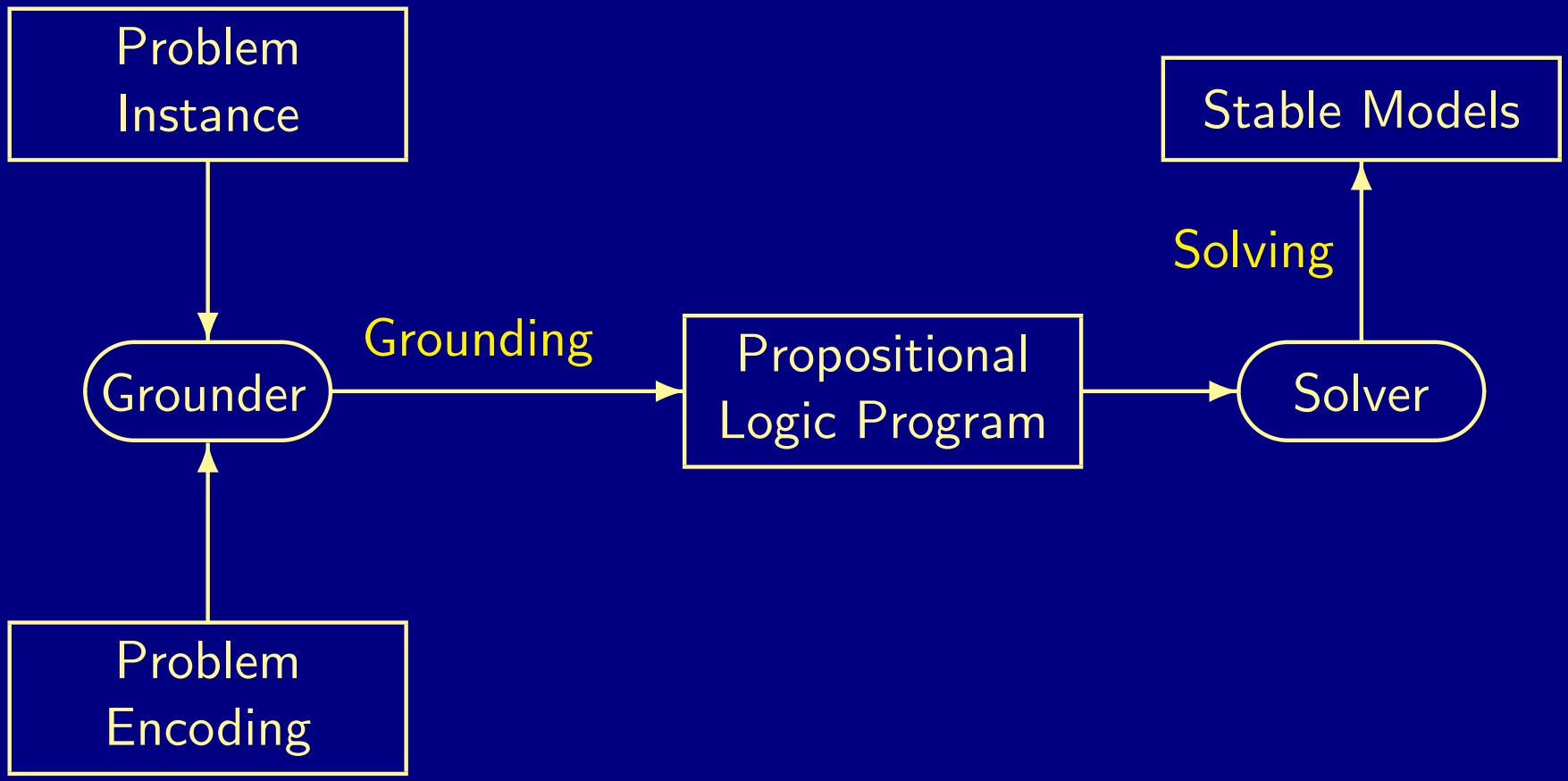
```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```



Interlude: Answer Set(s) Computation



Interlude: Answer Set(s) Computation



N-Colorability

Grounding

```
gringo -t graph.lp color.lp
```

```
node(1).   node(2).   node(3).   node(4).   node(5).   node(6).
edge(1,2). edge(1,3). edge(1,4). edge(2,4). edge(2,5). ...

iscol(1).  iscol(2).  iscol(3).

1 #count{ color(1,1), color(1,2), color(1,3) } 1.
1 #count{ color(2,1), color(2,2), color(2,3) } 1.
1 #count{ color(3,1), color(3,2), color(3,3) } 1.
1 #count{ color(4,1), color(4,2), color(4,3) } 1.
1 #count{ color(5,1), color(5,2), color(5,3) } 1.
1 #count{ color(6,1), color(6,2), color(6,3) } 1.

:- color(1,1), color(2,1).
:- color(1,2), color(2,2).
:- color(1,3), color(2,3). ...
```

N-Colorability

Grounding

```
gringo -t graph.lp color.lp
```

```
node(1).   node(2).   node(3).   node(4).   node(5).   node(6).
edge(1,2). edge(1,3). edge(1,4). edge(2,4). edge(2,5). ...

iscol(1).  iscol(2).  iscol(3).

1 #count{ color(1,1), color(1,2), color(1,3) } 1.
1 #count{ color(2,1), color(2,2), color(2,3) } 1.
1 #count{ color(3,1), color(3,2), color(3,3) } 1.
1 #count{ color(4,1), color(4,2), color(4,3) } 1.
1 #count{ color(5,1), color(5,2), color(5,3) } 1.
1 #count{ color(6,1), color(6,2), color(6,3) } 1.

:- color(1,1), color(2,1).
:- color(1,2), color(2,2).
:- color(1,3), color(2,3). ...
```

N-Colorability

Solving

```
gringo graph.lp color.lp | clasp --stats 0
```

```
...
Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 5
Conflicts    : 0
Restarts     : 0

Atoms        : 63
Rules        : 113    (1: 95 2: 12 3: 6)
Bodies       : 64
Equivalences: 106    (Atom=Atom: 31 Body=Body: 6 Other: 69)
Tight        : Yes

Variables    : 63     (Eliminated: 0 Frozen:   30)
Constraints  : 45     (Binary: 73.3% Ternary:  0.0% Other: 26.7%)
Lemmas       : 0      (Binary:  0.0% Ternary:  0.0% Other:  0.0%)
```

N-Colorability

Solving

```
gringo graph.lp color.lp | clasp --stats 0
```

```
...
Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 5
Conflicts    : 0
Restarts     : 0

Atoms        : 63
Rules        : 113    (1: 95 2: 12 3: 6)
Bodies       : 64
Equivalences: 106    (Atom=Atom: 31 Body=Body: 6 Other: 69)
Tight        : Yes

Variables    : 63     (Eliminated: 0 Frozen:   30)
Constraints  : 45     (Binary: 73.3% Ternary:  0.0% Other: 26.7%)
Lemmas       : 0      (Binary:  0.0% Ternary:  0.0% Other:  0.0%)
```

N-Colorability

Solving

```
gringo graph.lp color.lp | clasp --stats 0
```

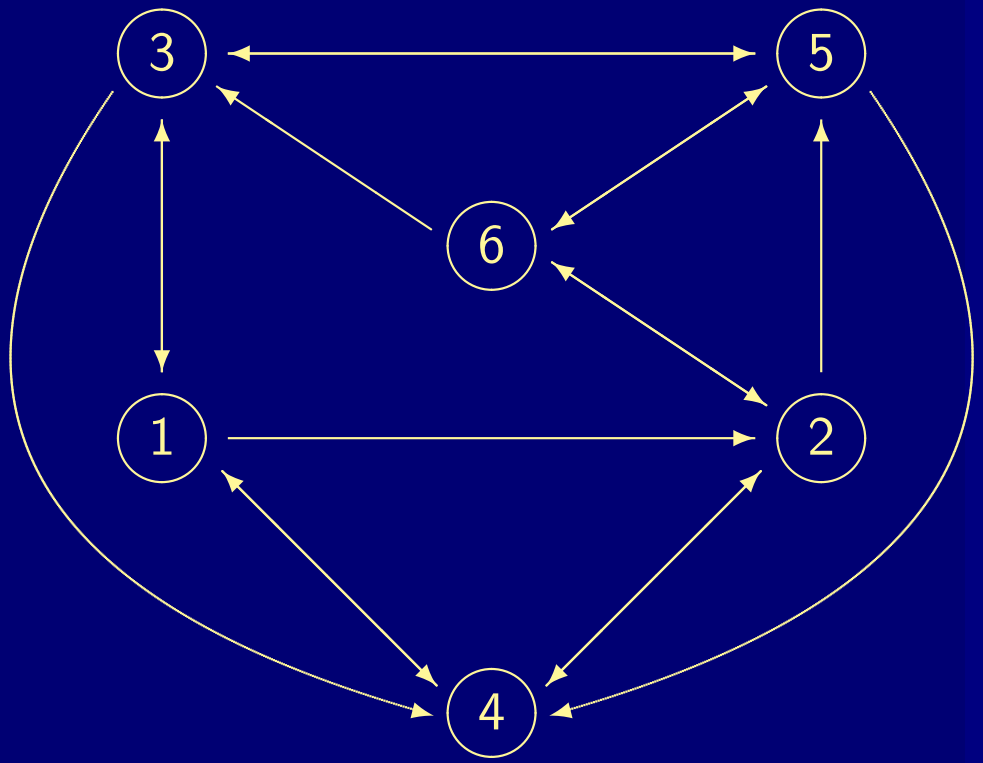
```
...
Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 5
Conflicts    : 0
Restarts     : 0

Atoms        : 63
Rules        : 113    (1: 95 2: 12 3: 6)
Bodies       : 64
Equivalences: 106    (Atom=Atom: 31 Body=Body: 6 Other: 69)
Tight        : Yes

Variables    : 63     (Eliminated: 0 Frozen:   30)
Constraints  : 45     (Binary: 73.3% Ternary:  0.0% Other: 26.7%)
Lemmas       : 0      (Binary:  0.0% Ternary:  0.0% Other:  0.0%)
```

Hamiltonian Cycle

Problem Instance as Facts

Recall: a directed graph G

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2). edge(5,3).
edge(5,4). edge(5,6). edge(6,2).
edge(6,3). edge(6,5).
```



Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).
```

- ☞ The definition of `reach` is recursive!

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).

first(X) :- X = #min[ node(Y) = Y ].
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).

:- node(Y), not reach(Y).
```

Hamiltonian Cycle

The Complete Picture

Uniform Encoding (in `cycle.lp`)

```
% DOMAIN
first(X) :- X = #min[ node(Y) = Y ].

% GENERATE
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

% DEFINE
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).

% TEST
:- node(Y), not reach(Y).

% DISPLAY
#hide. #show cycle/2.
```

Hamiltonian Cycle

Let's Run it!

```
gringo graph.lp cycle.lp | clasp --stats
```

```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
SATISFIABLE

Models       : 1+
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 3
Conflicts    : 0
Restarts     : 0

Atoms        : 84
Rules        : 117    (1: 84 2: 21 3: 12)
Bodies       : 81
Equivalences: 174    (Atom=Atom: 36 Body=Body: 12 Other: 126)
Tight        : No     (SCCs: 1 Nodes: 20)
```

Hamiltonian Cycle

Let's Run it!

```
gringo graph.lp cycle.lp | clasp --stats
```

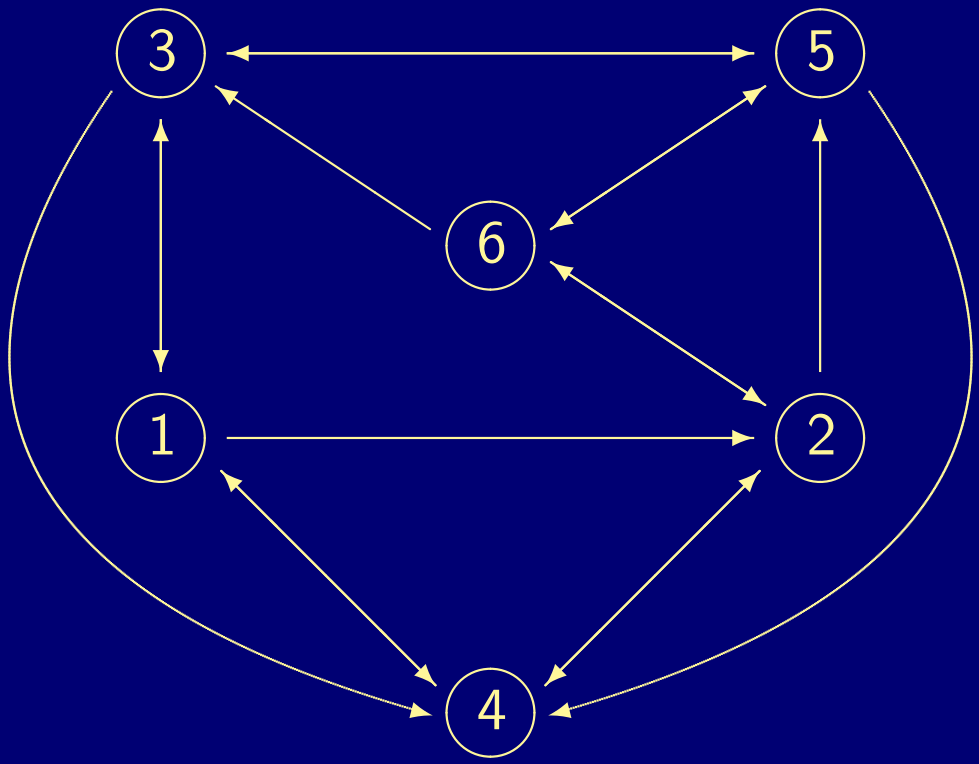
```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
SATISFIABLE

Models       : 1+
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 3
Conflicts    : 0
Restarts     : 0

Atoms        : 84
Rules        : 117    (1: 84 2: 21 3: 12)
Bodies       : 81
Equivalences: 174    (Atom=Atom: 36 Body=Body: 12 Other: 126)
Tight        : No     (SCCs: 1 Nodes: 20)
```

Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

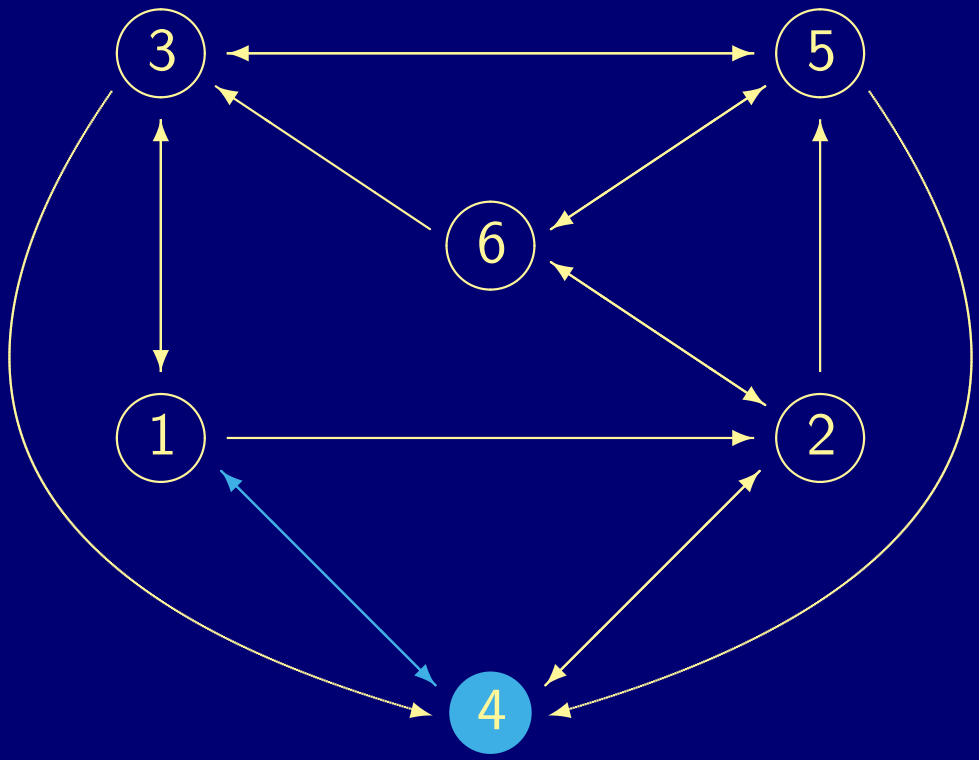


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

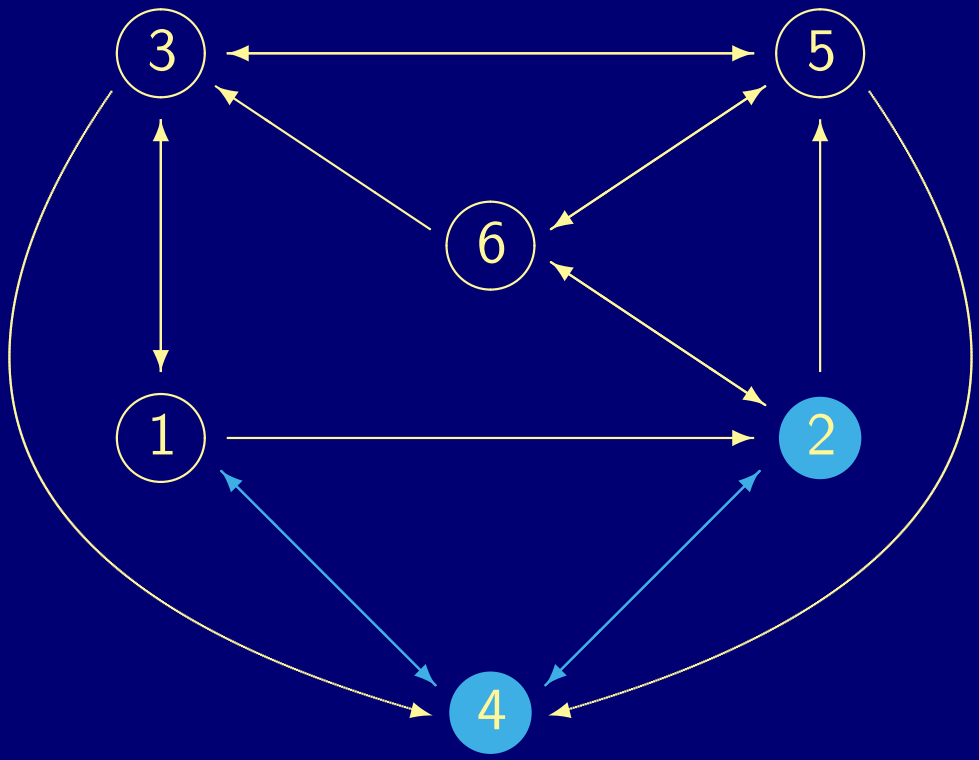


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

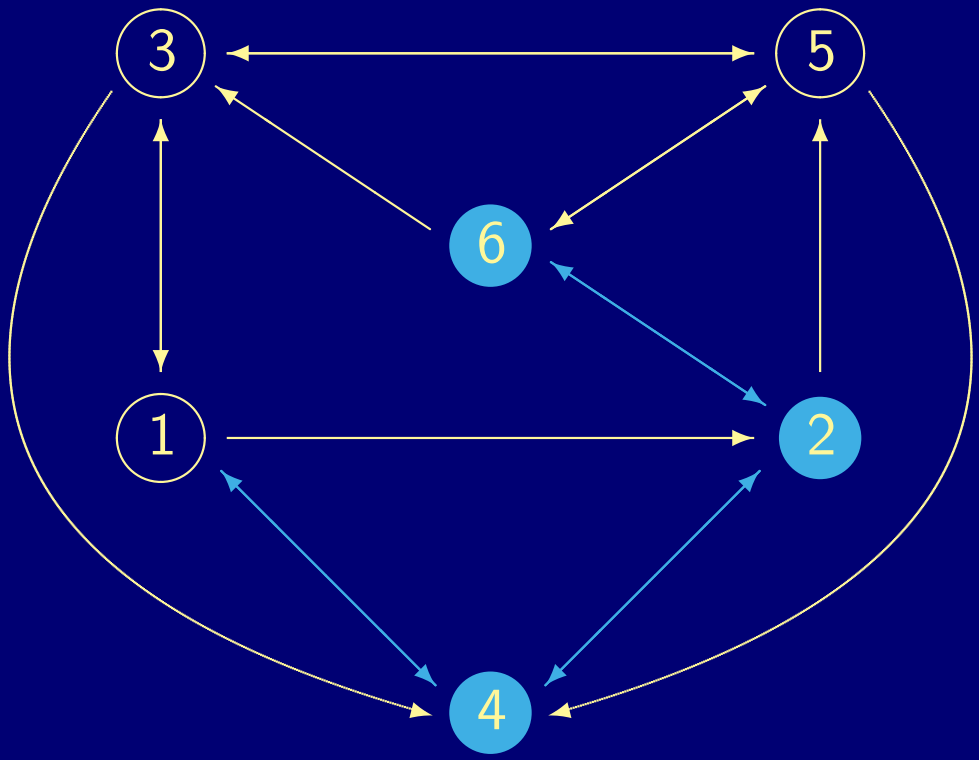


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

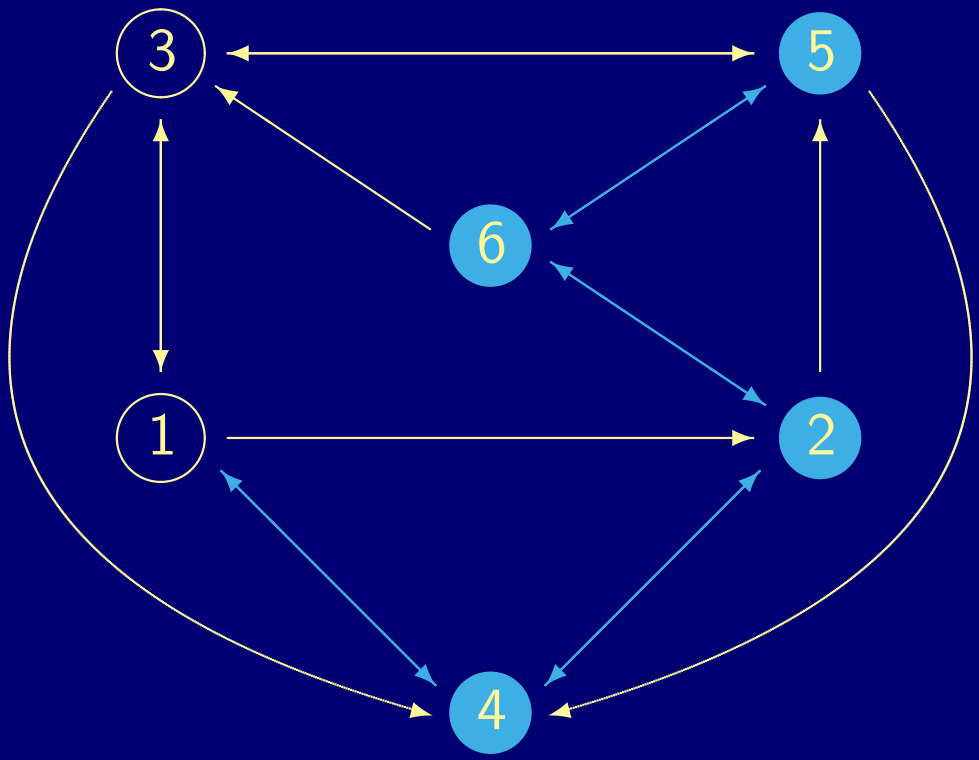


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

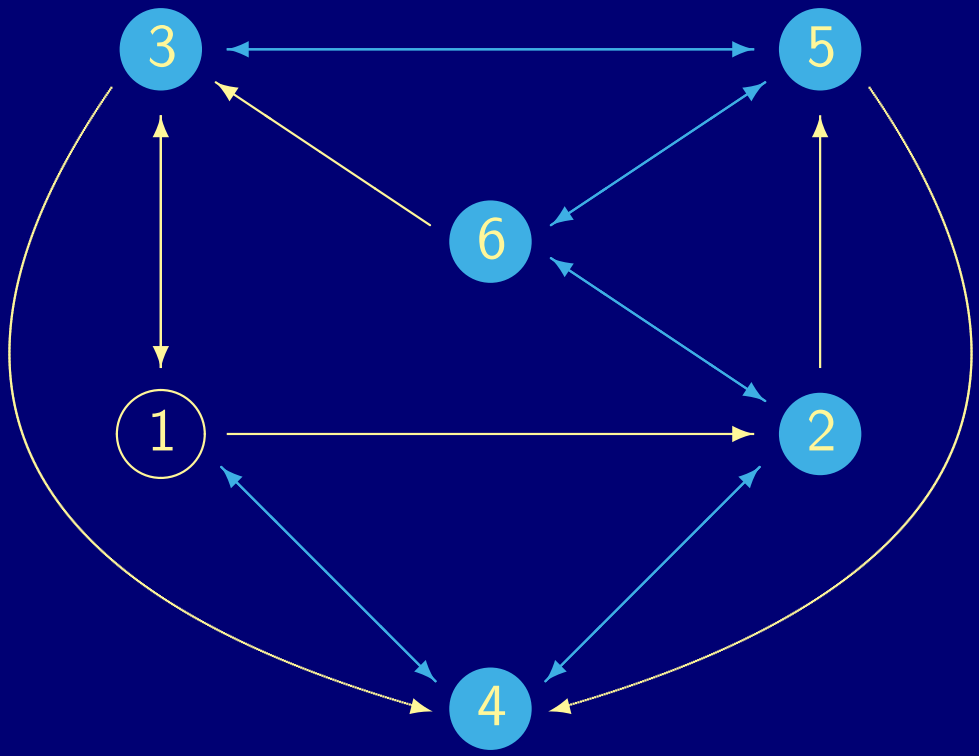


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

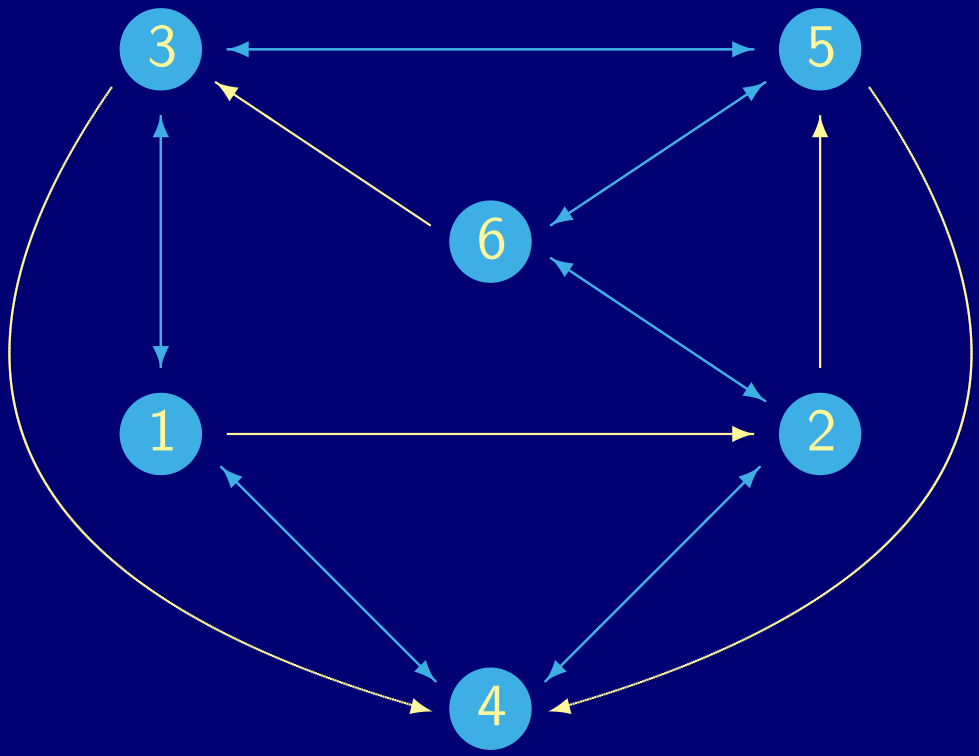


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

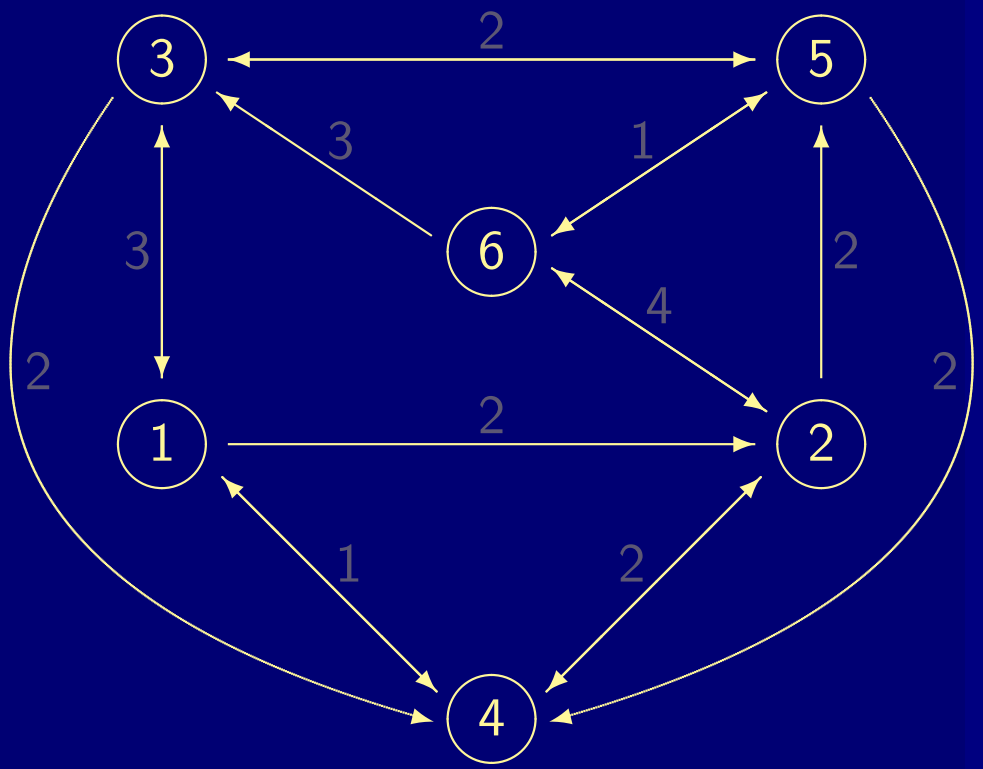


Mr Hamilton as Traveling Salesperson

Problem Instance as Facts

Given: a directed graph G plus edge costs

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2). edge(5,3).
edge(5,4). edge(5,6). edge(6,2).
edge(6,3). edge(6,5).
```

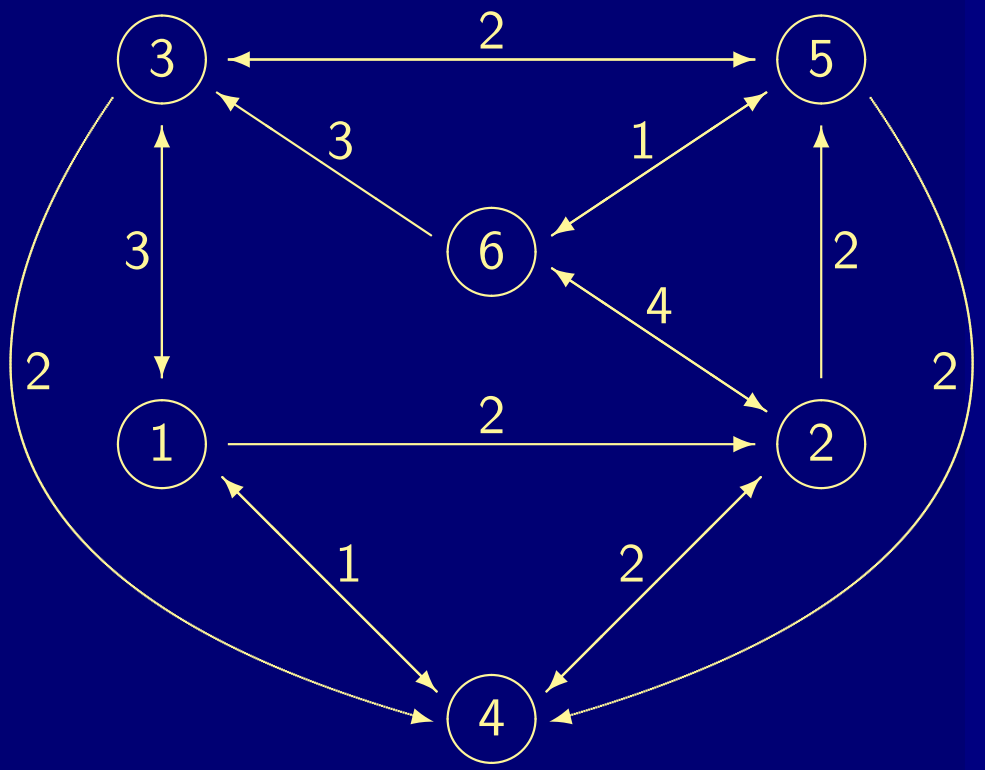


Mr Hamilton as Traveling Salesperson

Problem Instance as Facts

Given: a directed graph G plus edge costs

```
cost(1,2,2).
cost(1,3,3). cost(3,1,3).
cost(1,4,1). cost(4,1,1).
cost(2,4,2). cost(4,2,2).
cost(2,5,2).
cost(2,6,4). cost(6,2,4).
cost(3,4,2).
cost(3,5,2). cost(5,3,2).
cost(5,4,2).
cost(5,6,1). cost(6,5,1).
cost(6,3,3).
```



Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

- ☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

- ☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

- ☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

- ☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

- ☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

- ☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Let's Run it!

```
gringo graph.lp costs.lp cycle.lp price.lp | clasp --stats 0
```

```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
Optimization: 13
Answer: 2
cycle(6,5) cycle(5,3) cycle(4,1) cycle(3,4) cycle(2,6) cycle(1,2)
Optimization: 12
Answer: 3
cycle(6,3) cycle(5,6) cycle(4,1) cycle(3,4) cycle(2,5) cycle(1,2)
Optimization: 11
OPTIMUM FOUND

Models       : 1
  Enumerated: 3
  Optimum    : yes
Optimization: 11
Time         : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Mr Hamilton as Traveling Salesperson

Let's Run it!

```
gringo graph.lp costs.lp cycle.lp price.lp | clasp --stats 0
```

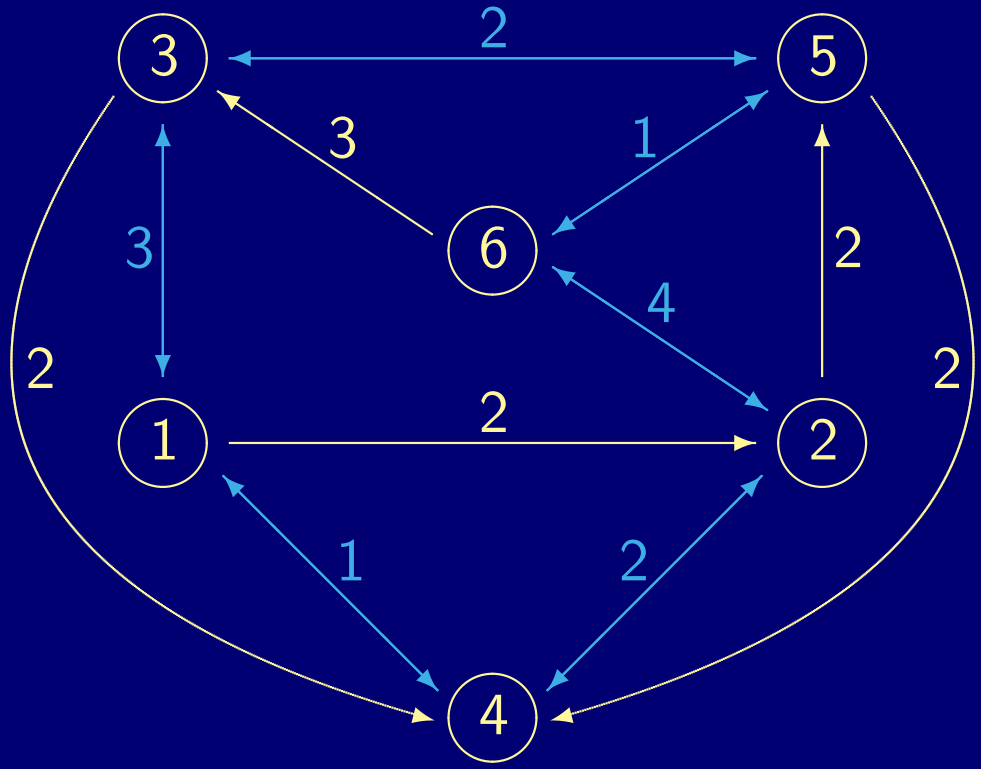
```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
Optimization: 13
Answer: 2
cycle(6,5) cycle(5,3) cycle(4,1) cycle(3,4) cycle(2,6) cycle(1,2)
Optimization: 12
Answer: 3
cycle(6,3) cycle(5,6) cycle(4,1) cycle(3,4) cycle(2,5) cycle(1,2)
Optimization: 11
OPTIMUM FOUND

Models       : 1
  Enumerated: 3
  Optimum    : yes
Optimization: 11
Time         : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Mr Hamilton as Traveling Salesperson

Let's Interpret it!

Found: optimal Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

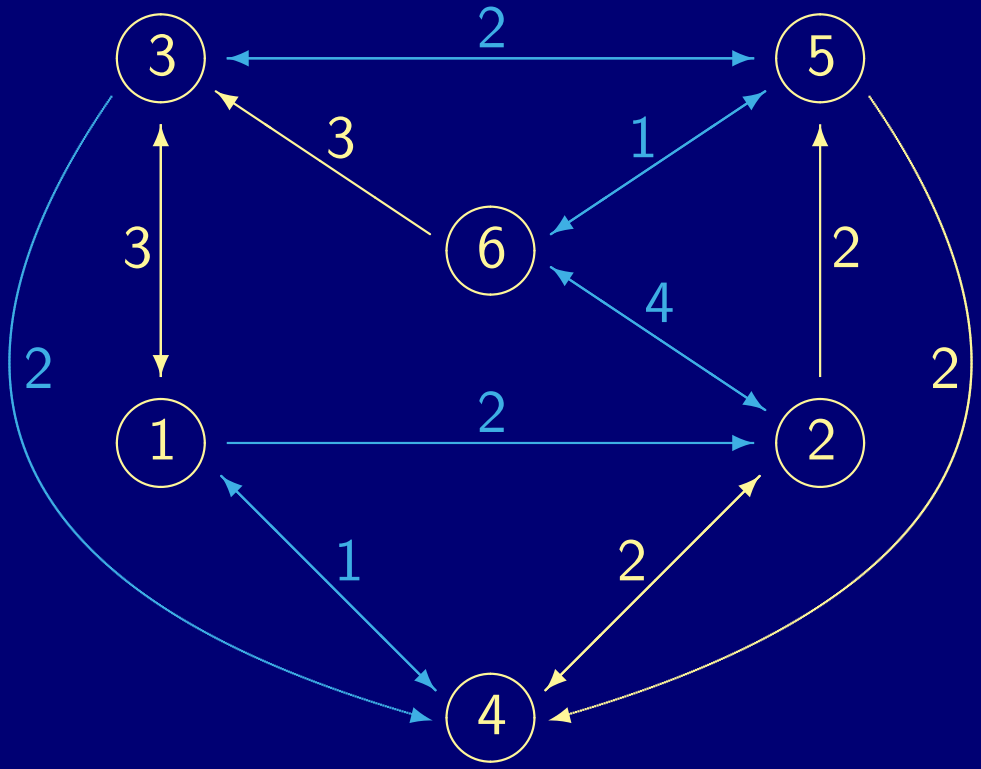


Mr Hamilton as Traveling Salesperson

Let's Interpret it!

Found: optimal Hamiltonian cycle

```
Answer: 2

cycle(1,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,4)
cycle(4,1)
```

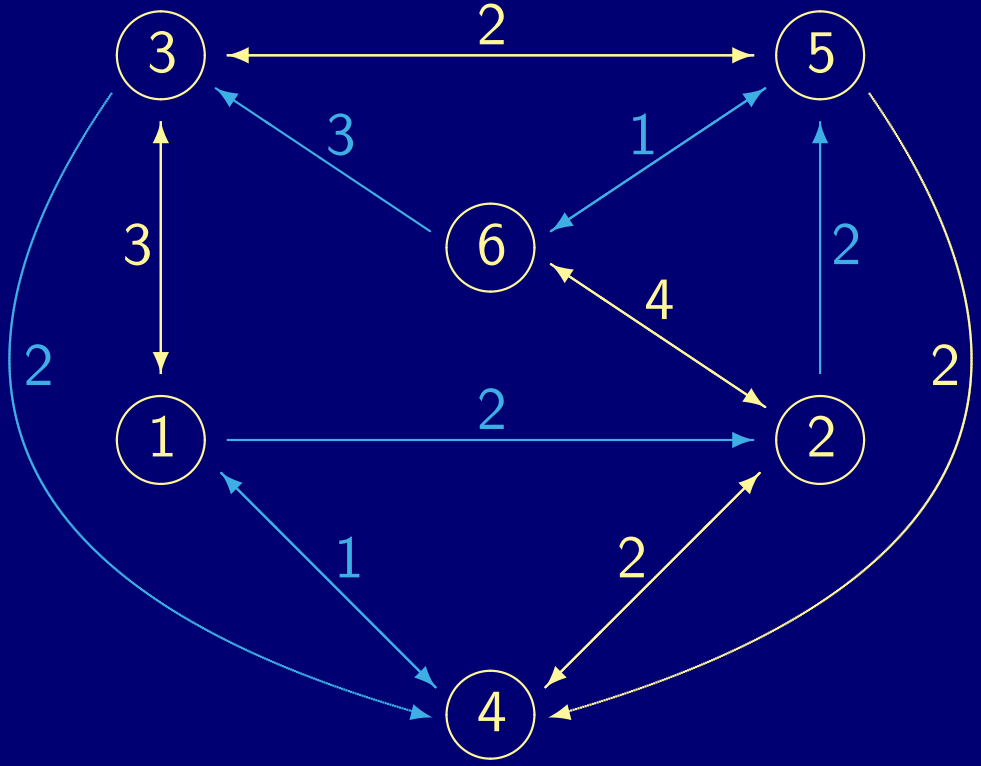


Mr Hamilton as Traveling Salesperson

Let's Interpret it!

Found: optimal Hamiltonian cycle

```
Answer: 3

cycle(1,2)
cycle(2,5)
cycle(5,6)
cycle(6,3)
cycle(3,4)
cycle(4,1)
```



Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

29 Encoding Methodology
- Tweaking *N*-Queens
- Do's and Dont's

30 Hints

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?

Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?
Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?
Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?
Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

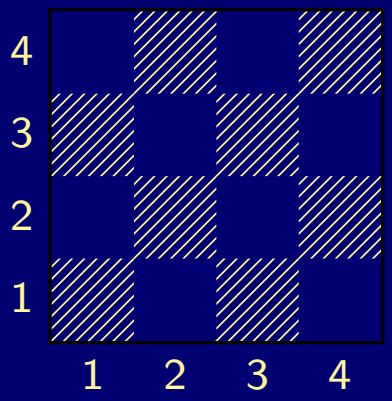
N-Queens Problem

Problem Specification

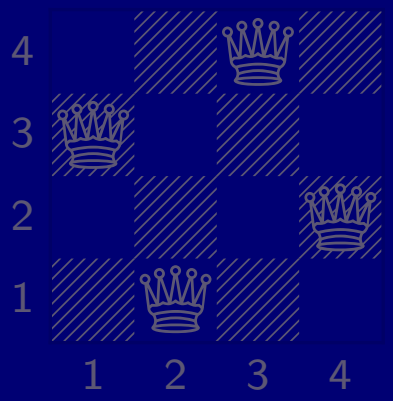
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally).

$N = 4$

Chessboard



Placement



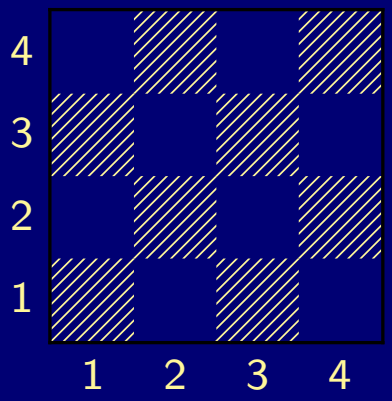
N-Queens Problem

Problem Specification

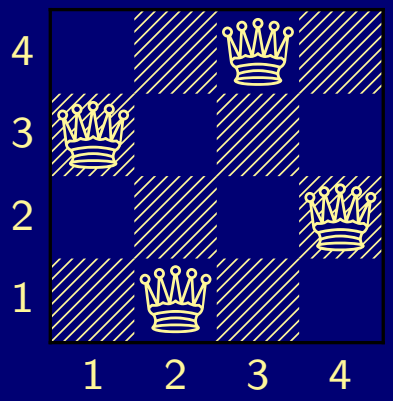
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally).

$N = 4$

Chessboard



Placement



A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#hide. #show queen/2.
```

Anything missing?

A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.
4. We have to place (at least) N queens.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models       : 1+
Time         : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 18
Conflicts    : 13
Restarts     : 0

Variables    : 793
Constraints  : 729
```

A First Encoding

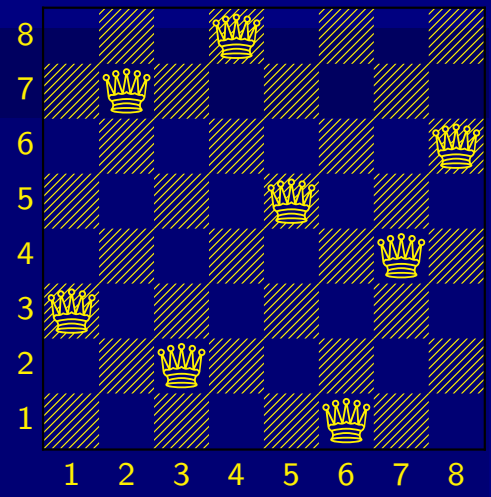
Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models       : 1+
Time         : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 18
Conflicts    : 13
Restarts     : 0

Variables    : 793
Constraints  : 729
```

A First Encoding

Let's Place 8 Queens!

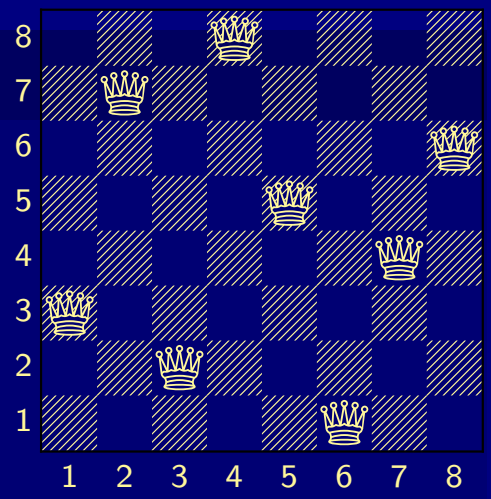



```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models       : 1+
Time         : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 18
Conflicts    : 13
Restarts     : 0

Variables    : 793
Constraints  : 729
```

A First Encoding

Let's Place 8 Queens!




```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models       : 1+
Time         : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 18
Conflicts    : 13
Restarts     : 0

Variables    : 793
Constraints  : 729
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models       : 1+
Time         : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time     : 147.480s
Choices      : 594960
Conflicts    : 574565
Restarts     : 19

Variables    : 17271
Constraints  : 16787
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models       : 1+
Time         : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time     : 147.480s
Choices      : 594960
Conflicts    : 574565
Restarts     : 19

Variables    : 17271
Constraints  : 16787
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least *N* queens?

Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least N queens?

Exactly one queen per row and column!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.


% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models       : 1+
Time         : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.020s
Choices      : 132
Conflicts    : 105
Restarts     : 1

Variables    : 7238
Constraints  : 6710
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models       : 1+
Time         : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.020s
Choices      : 132
Conflicts    : 105
Restarts     : 1

Variables    : 7238
Constraints  : 6710
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models       : 1+
Time         : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time     : 6.930s
Choices      : 1373
Conflicts    : 845
Restarts     : 4

Variables    : 1211338
Constraints  : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models       : 1+
Time         : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time     : 6.930s
Choices      : 1373
Conflicts    : 845
Restarts     : 4

Variables    : 1211338
Constraints  : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models       : 1+
Time         : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time     : 6.930s
Choices      : 1373
Conflicts    : 845
Restarts     : 4

Variables    : 1211338
Constraints  : 1196210
```

A First Refinement

Where Time Has Gone

```
time( gringo -c n=122 queens_1.lp | clasp --stats
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding causes the problem!

A First Refinement

Where Time Has Gone

```
time( gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding causes the problem!

A First Refinement

Where Time Has Gone

```
time( gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

Just kidding :)

☞ Grounding causes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding causes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding causes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding causes the problem!

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                  O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                        O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n²×n²)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                  O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                        O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                          O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n²×n²)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                          O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                  O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                        O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                          O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                          O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n²×n²)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                          O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                  O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                        O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                  O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                        O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

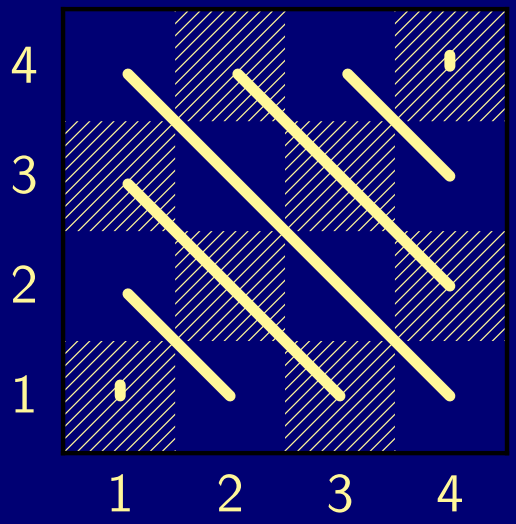
queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                          O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

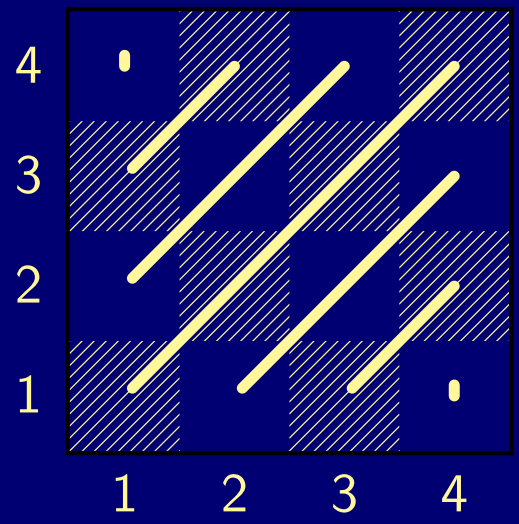
Diagonals cause trouble!

A Nomenclature for Diagonals

$N = 4$



$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

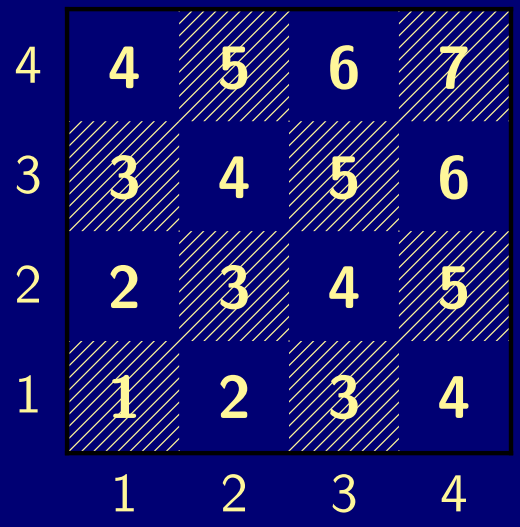


$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

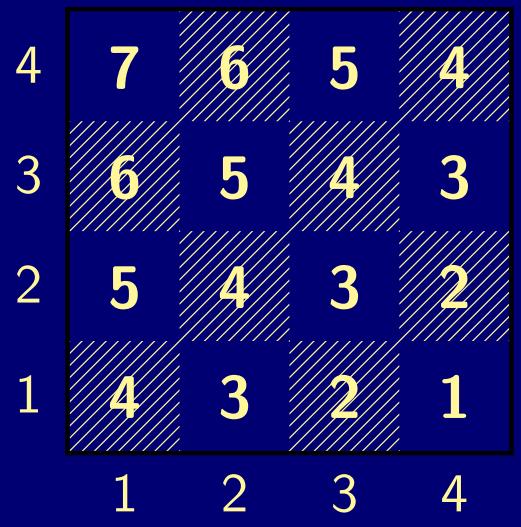
☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Nomenclature for Diagonals

$N = 4$



$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

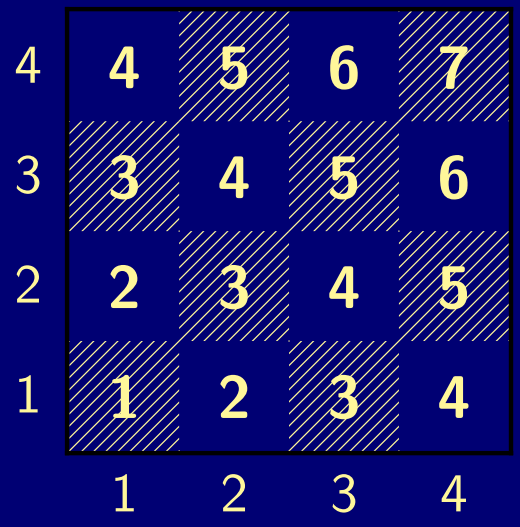


$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

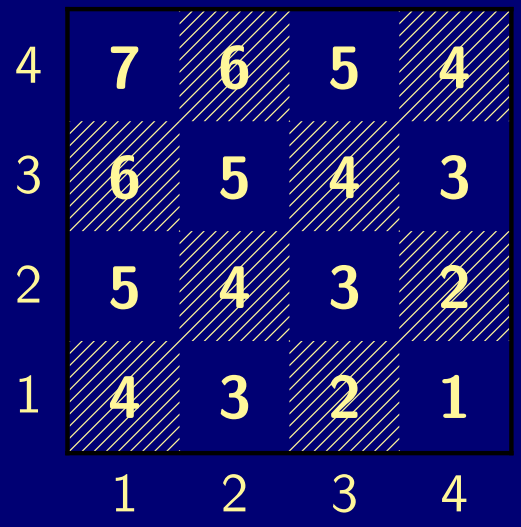
☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Nomenclature for Diagonals

$N = 4$



$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

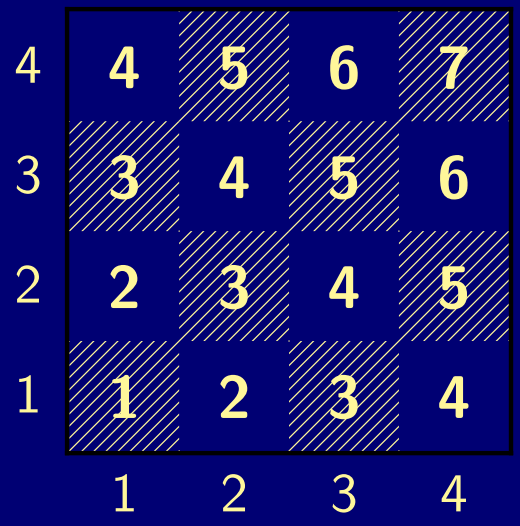


$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

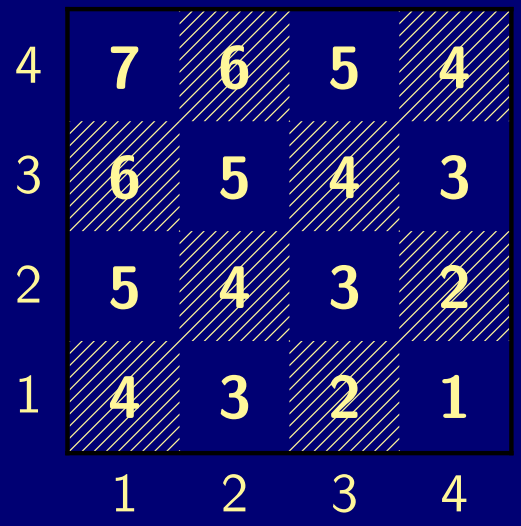
☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Nomenclature for Diagonals

$N = 4$



$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$



$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.


% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1


% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models       : 1+
Time         : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time     : 0.210s
Choices      : 11036
Conflicts    : 499
Restarts     : 3

Variables    : 16098
Constraints  : 970
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models       : 1+
Time         : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time     : 0.210s
Choices      : 11036
Conflicts    : 499
Restarts     : 3

Variables    : 16098
Constraints  : 970
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models       : 1+
Time         : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time     : 0.210s
Choices      : 11036
Conflicts    : 499
Restarts     : 3

Variables    : 16098
Constraints  : 970
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time     : 7.250s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time     : 7.250s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time     : 7.250s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Precompute Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Precompute Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Precompute Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Precompute Diagonals!

queens_3.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models       : 1+
Time         : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time     : 68.620s
Choices      : 869379
Conflicts    : 25746
Restarts     : 12

Variables    : 365994
Constraints  : 4794
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models       : 1+
Time         : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time     : 68.620s
Choices      : 869379
Conflicts    : 25746
Restarts     : 12

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models       : 1+
Time         : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time     : 68.620s
Choices      : 869379
Conflicts    : 25746
Restarts     : 12

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models       : 1+
Time         : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time     : 68.620s
Choices      : 869379
Conflicts    : 25746
Restarts     : 12

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models       : 1+
Time         : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time     : 29.580s
Choices      : 961315
Conflicts    : 3222
Restarts     : 7

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models       : 1+
Time         : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time     : 29.580s
Choices      : 961315
Conflicts    : 3222
Restarts     : 7

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,90) queen(2,452) queen(3,494) queen(4,145) queen(5,84) ...
SATISFIABLE

Models       : 1+
Time         : 22.654s (Solving: 10.53s 1st Model: 10.47s Unsat: 0.00s)
CPU Time     : 15.750s
Choices      : 1058729
Conflicts    : 2128
Restarts     : 6

Variables    : 403123
Constraints  : 49636
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).       veg(cucumber).
pro(asparagus,cheap). pro(cucumber,cheap).
pro(asparagus,fresh). pro(cucumber,fresh).
pro(asparagus,tasty). pro(cucumber,tasty).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).       veg(cucumber).
pro(asparagus,cheap). pro(cucumber,cheap).
pro(asparagus,fresh). pro(cucumber,fresh).
pro(asparagus,tasty). pro(cucumber,tasty).


buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty), pro(X,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).
pro(asparagus,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).    pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).    pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).    pre(tasty).
pro(asparagus,clean).                            pre(clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).    pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).    pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).    pre(tasty).
pro(asparagus,clean).                            pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

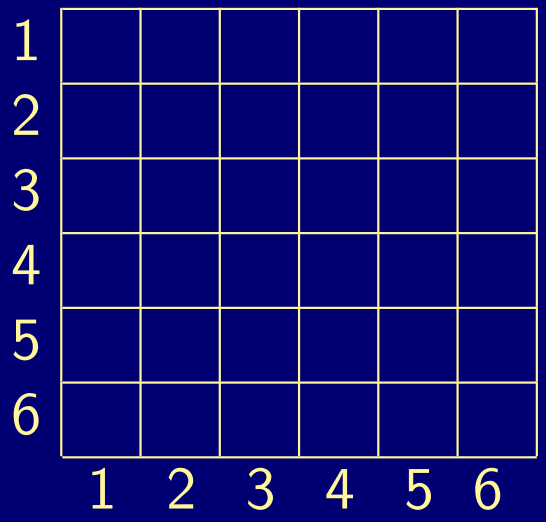
1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).    pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).    pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).    pre(tasty).
pro(asparagus,clean).                            pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Running Example: Latin Square

Given: an $N \times N$ board



represented by facts:

```
square(1,1). ... square(1,6).
square(2,1). ... square(2,6).
square(3,1). ... square(3,6).
square(4,1). ... square(4,6).
square(5,1). ... square(5,6).
square(6,1). ... square(6,6).
```

Wanted: assignment of $1, \ldots, N$

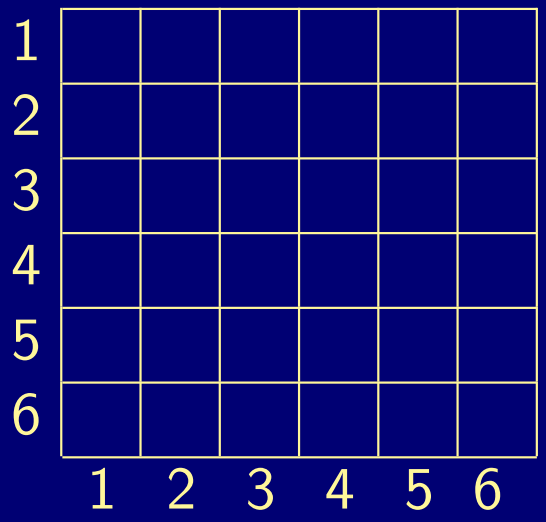
	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Running Example: Latin Square

Given: an $N \times N$ board



represented by facts:

```
square(1,1). ... square(1,6).
square(2,1). ... square(2,6).
square(3,1). ... square(3,6).
square(4,1). ... square(4,6).
square(5,1). ... square(5,6).
square(6,1). ... square(6,6).
```

Wanted: assignment of $1, \ldots, N$

1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5
	1	2	3	4	5	6

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).


% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

☞ unreused "singleton variables"

```
gringo latin_0.lp | wc
105480 2558984 14005258
```

```
gringo latin_1.lp | wc
42056 273672 1690522
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).


% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

☞ unreused "singleton variables"

```
gringo latin_0.lp | wc
105480 2558984 14005258
```

```
gringo latin_1.lp | wc
42056 273672 1690522
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).


% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

☞ unreused "singleton variables"

```
gringo latin_0.lp | wc
```

```
105480 2558984 14005258
```

```
gringo latin_1.lp | wc
```

```
42056 273672 1690522
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X1)   , N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- squareY(Y1)   , N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

☞ unreused "singleton variables"

```
gringo latin_0.lp | wc
```

```
105480 2558984 14005258
```

```
gringo latin_1.lp | wc
```

```
42056 273672 1690522
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X1)   , N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- squareY(Y1)   , N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

☞ unreused "singleton variables"

```
gringo latin_0.lp | wc
```

```
105480 2558984 14005258
```

```
gringo latin_1.lp | wc
```

```
42056 273672 1690522
```

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

☞ duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

```
gringo latin_2.lp | wc
2071560 12389384 40906946
```

```
gringo latin_3.lp | wc
1055752 6294536 21099558
```

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

☞ duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

```
gringo latin_2.lp | wc
2071560 12389384 40906946
```

```
gringo latin_3.lp | wc
1055752 6294536 21099558
```

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

☞ duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

```
gringo latin_2.lp | wc
```

```
2071560 12389384 40906946
```

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 <  Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 <  X2.
```

☞ duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

```
gringo latin_2.lp | wc
```

```
2071560 12389384 40906946
```

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 <  Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 <  X2.
```

☞ duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

```
gringo latin_2.lp | wc
```

```
2071560 12389384 40906946
```

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

☞ uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

☞ uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc                gringo latin_4.lp | wc
1055752 6294536 21099558              228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

☞ uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
```

```
228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

☞ uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
```

```
228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

☞ uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
```

```
228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

☞ uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
```

```
228360 1205256 4780744
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc                gringo latin_6.lp | wc
204136 5778440 30252505                48136 373768 2185042
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

gringo latin_5.lp | wc gringo latin_6.lp | wc

204136 5778440 30252505 48136 373768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.   :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

gringo latin_5.lp | wc gringo latin_6.lp | wc

204136 5778440 30252505 48136 272768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

gringo latin_5.lp | wc gringo latin_6.lp | wc

204136 5778440 30252505 48136 373768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C #count{ num(X,Y,N) } C, C = 0..n.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C #count{ num(X,Y,N) } C, C = 0..n.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

☞ internal transformation by gringo

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.        ✗
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.        ✗
:- occX(X,N,C), C != 1.   :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

gringo latin_5.lp | wc gringo latin_6.lp | wc

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

```
48136 373768 2185042
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.   :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
304136 5778440 30252505
```

```
gringo latin_6.lp | wc
```

```
48136 373768 2185042
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
304136 5778440 30252505
```

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
304136 5778440 30252505
```

```
gringo latin_6.lp | wc
```

```
48136 373768 2185042
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

☞ many symmetric solutions (mirroring, rotation, value permutation)

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

☞ easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

☞ easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

☞ Let's compare enumeration speed!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

```
Models : 161280    Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 161280    Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 1344      Time : 0.024s
```

Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

29 Encoding Methodology
- Tweaking N-Queens
- Do's and Dont's

30 Hints

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

develop and test incrementally
- prepare toy instances with "interesting features"
- build the encoding bottom-up and verify additions (eg. new predicates)

compare the encoded to the intended meaning
- check whether the grounding fits (use `gringo -t`)
- if answer sets are unintended, investigate conditions that fail to hold
- if answer sets are missing, examine integrity constraints (add heads)

ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo -t`)
  - if answer sets are unintended, investigate conditions that fail to hold
  - if answer sets are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo -t`)
  - if answer sets are unintended, investigate conditions that fail to hold
  - if answer sets are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

  if great search efforts (`Conflicts`/`Choices`/`Restarts`), then

  try auto-configuration (offered by `claspfolio`)
  try manual fine-tuning (requires expert knowledge!)
  if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  - try auto-configuration (offered by `claspfolio`)
  - try manual fine-tuning (requires expert knowledge!)
  - if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

☞ if great search efforts (`Conflicts`/`Choices`/`Restarts`), then

- try auto-configuration (offered by `claspfolio`)
- try manual fine-tuning (requires expert knowledge!)
- if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

☞ if great search efforts (`Conflicts`/`Choices`/`Restarts`), then

  1. try auto-configuration (offered by `claspfolio`)
  2. try manual fine-tuning (requires expert knowledge!)
  3. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

☞ if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try auto-configuration (offered by `claspfolio`)
  2. try manual fine-tuning (requires expert knowledge!)
  3. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

☞ if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try auto-configuration (offered by `claspfolio`)
  2. try manual fine-tuning (requires expert knowledge!)
  3. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Systems: Overview

31 Potassco

32 gringo

33 clasp

34 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

35 Book

Systems: Overview

31 Potassco

32 gringo

33 clasp

34 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

35 Book

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, pyngo
- *Solver*: clasp, claspd, claspar
- *Grounder+Solver*: clingo, iclingo, oclingo, clingcon
- *Further Tools*: claspfolio, coala, inca, plasp, sbass, xorro

Benchmark repository: http://asparagus.cs.uni-potsdam.de

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, pyngo
- *Solver*: clasp, claspd, claspar
- *Grounder+Solver*: clingo, iclingo, oclingo, clingcon
- *Further Tools*: claspfolio, coala, inca, plasp, sbass, xorro
- *Benchmark repository*: http://asparagus.cs.uni-potsdam.de

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, pyngo
- *Solver*: clasp, claspd, claspar
- *Grounder+Solver*: clingo, iclingo, oclingo, clingcon
- *Further Tools*: claspfolio, coala, inca, plasp, sbass, xorro
- *Benchmark repository*: `http://asparagus.cs.uni-potsdam.de`

Systems: Overview

31 Potassco

32 gringo

33 clasp

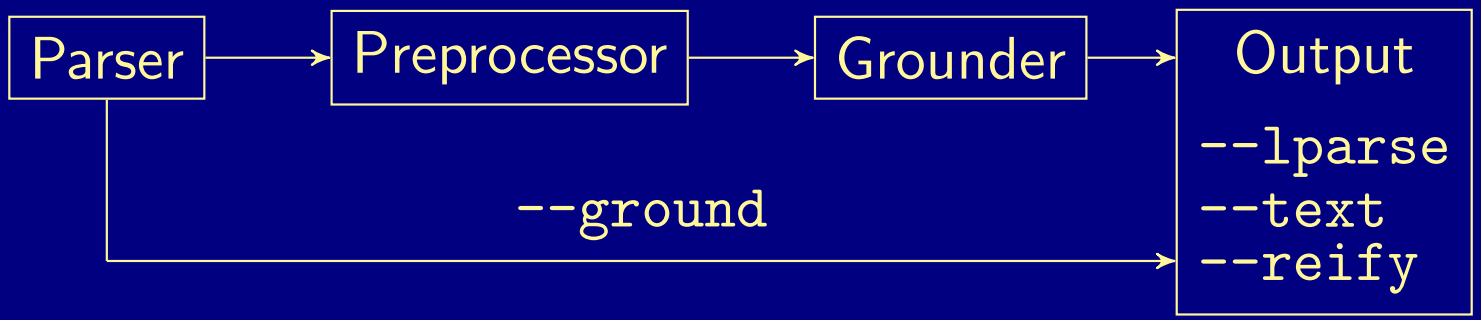
34 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

35 Book

gringo

- Accepts safe programs with aggregates
- Tolerates unrestricted use of function symbols
  (as long as it yields a finite ground instantiation :)
- Expressive power of a Turing machine

- Basic architecture of *gringo*:



Systems: Overview

31 Potassco

32 gringo

33 clasp

34 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

35 Book

clasp

- Native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - Advanced preprocessing including, e.g., equivalence reasoning
  - Lookback-based decision heuristics
  - Restart policies
  - Nogood deletion
  - Progress saving
  - Dedicated data structures for binary and ternary nogoods
  - Lazy data structures (watched literals) for long nogoods
  - Dedicated data structures for cardinality and weight constraints
  - Lazy unfounded set checking based on "source pointers"
  - Tight integration of unit propagation and unfounded set checking
  - Reasoning modes
  - Multi-threaded search
  - ...

Reasoning modes of *clasp*

Beyond deciding answer set existence, *clasp* allows for:

- Optimization
- Enumeration [without solution recording]
- Projective Enumeration [without solution recording]
- Brave and Cautious Reasoning determining the
  - union or
  - intersection

  of all answer sets by computing only linearly many of them
- and combinations thereof

clasp also allows for solving

- propositional CNF formulas (extended *dimacs*)
- pseudo-Boolean formulas (*opb* and *wbo*)

Reasoning modes of *clasp*

Beyond deciding answer set existence, *clasp* allows for:

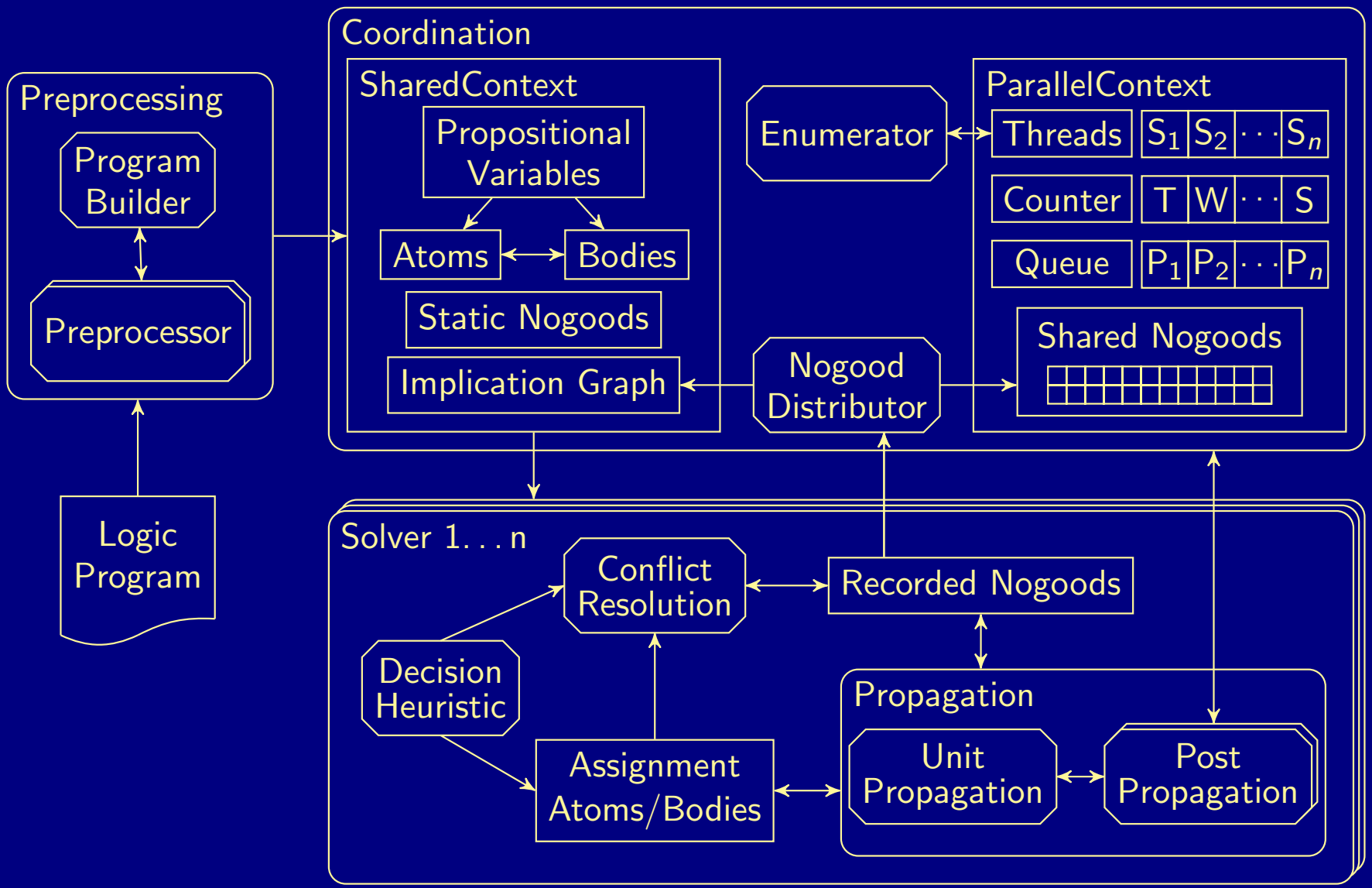
- Optimization
- Enumeration [without solution recording]
- Projective Enumeration [without solution recording]
- Brave and Cautious Reasoning determining the
  - union or
  - intersection

  of all answer sets by computing only linearly many of them
- and combinations thereof

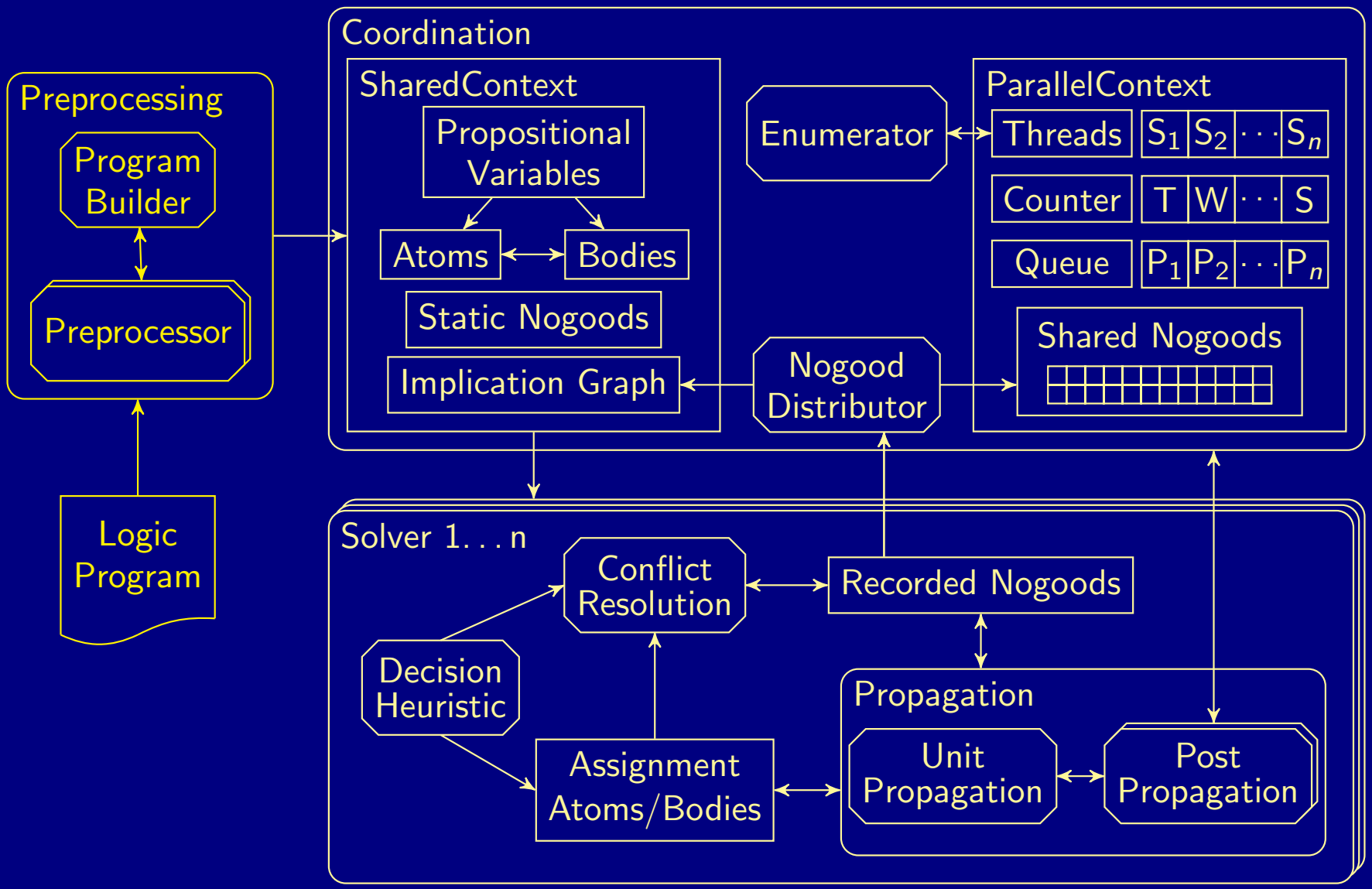
clasp also allows for solving

- propositional CNF formulas (extended *dimacs*)
- pseudo-Boolean formulas (*opb* and *wbo*)

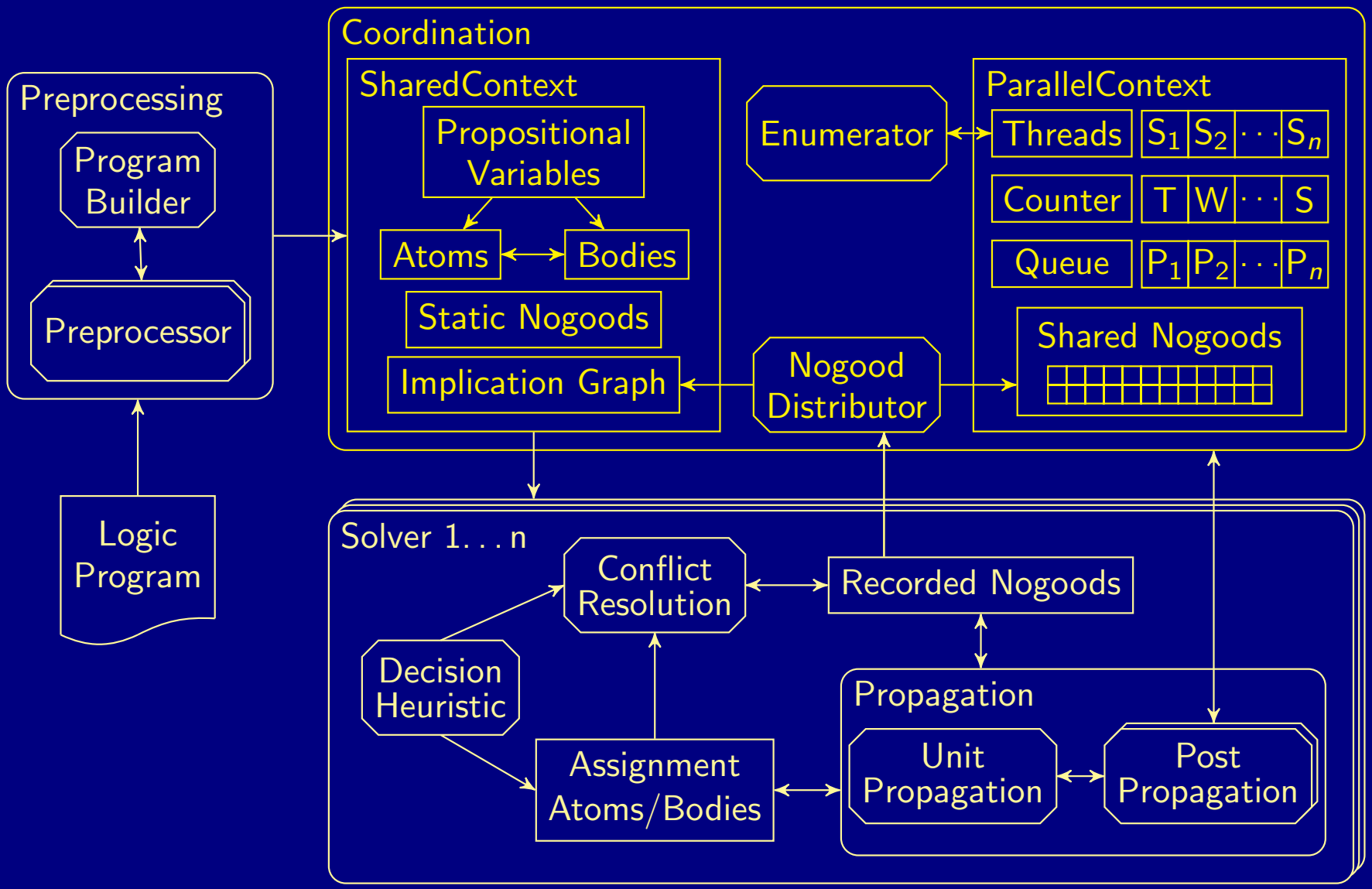
Multi-threaded architecture of *clasp*



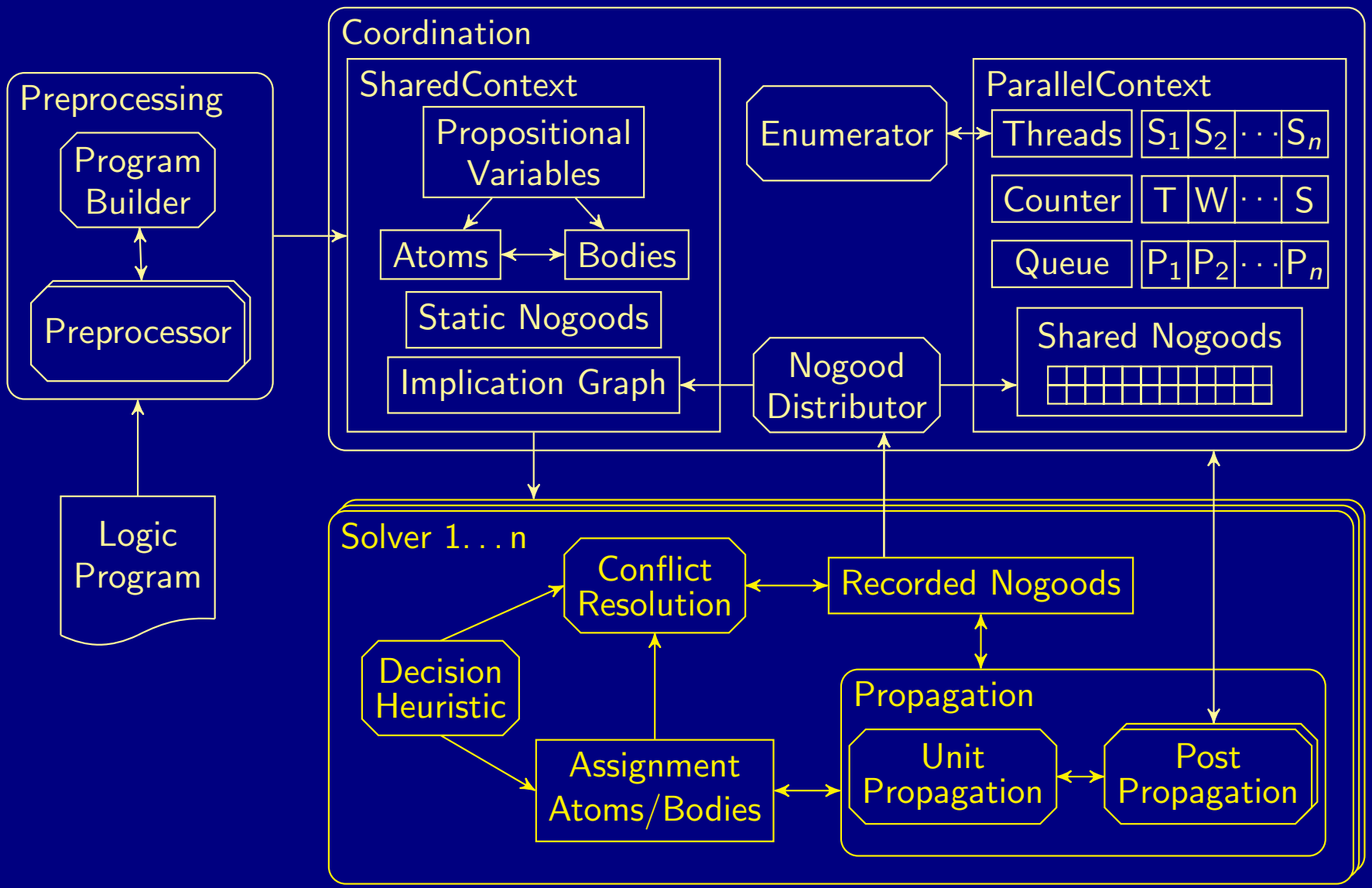
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Systems: Overview

31 Potassco

32 gringo

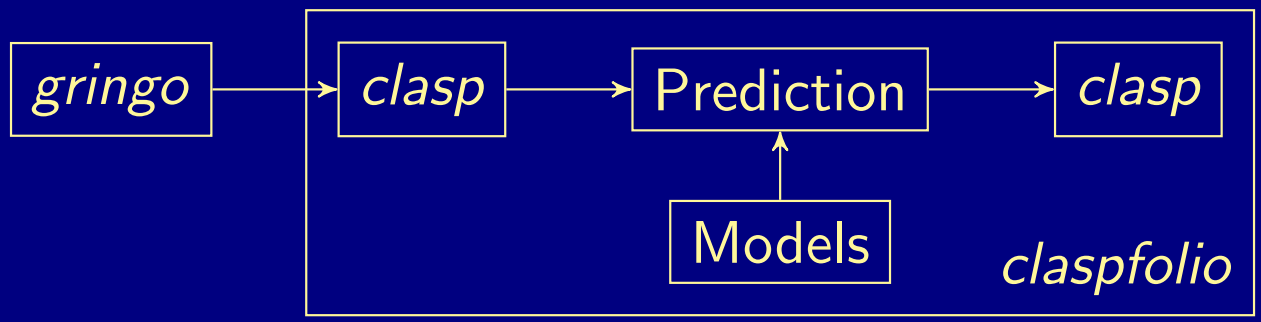
33 clasp

34 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

35 Book

claspfolio

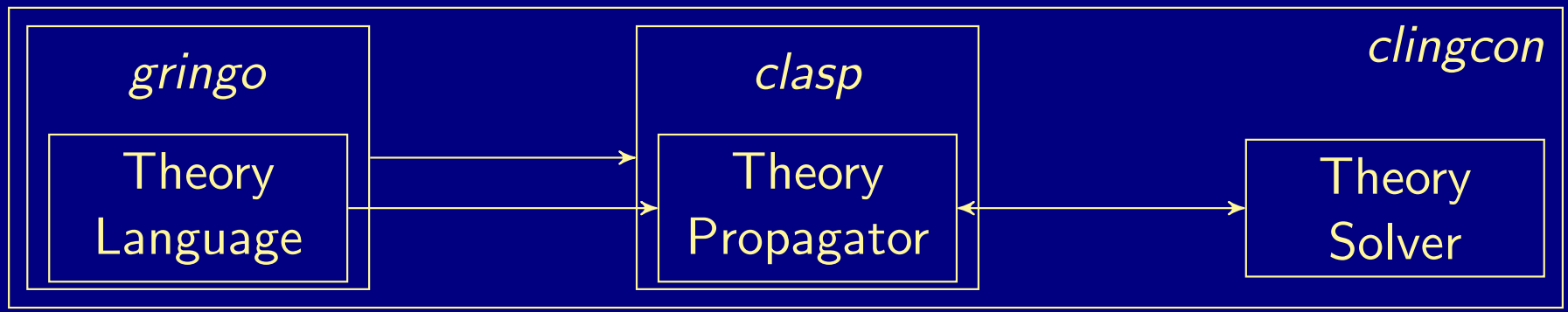
- Automatic selection of *clasp* configuration among 22 configuration via (learned) classifiers
- Basic architecture of *claspfolio*:



clingcon

- Hybrid grounding and solving
- Solving in hybrid domains, like Bio-Informatics

- Basic architecture of *clingcon*:



Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $== 0           :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) + amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

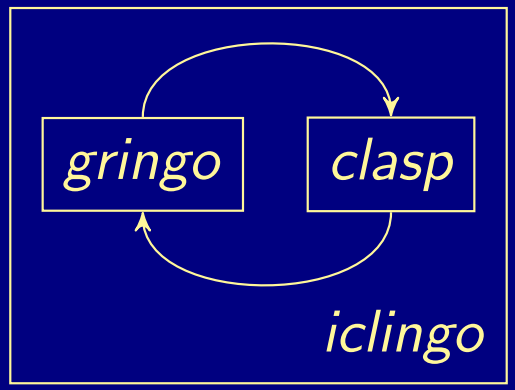
Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $== 0           :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) + amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

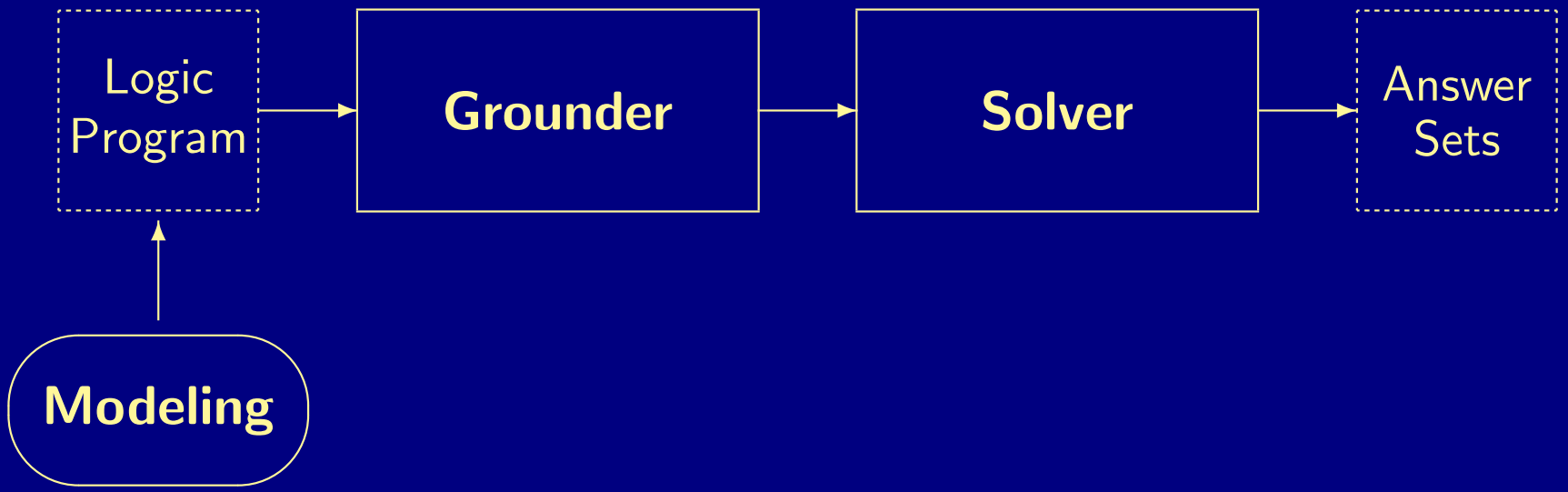
iclingo

- Incremental grounding and solving
- Offline solving in dynamic domains, like Automated Planning

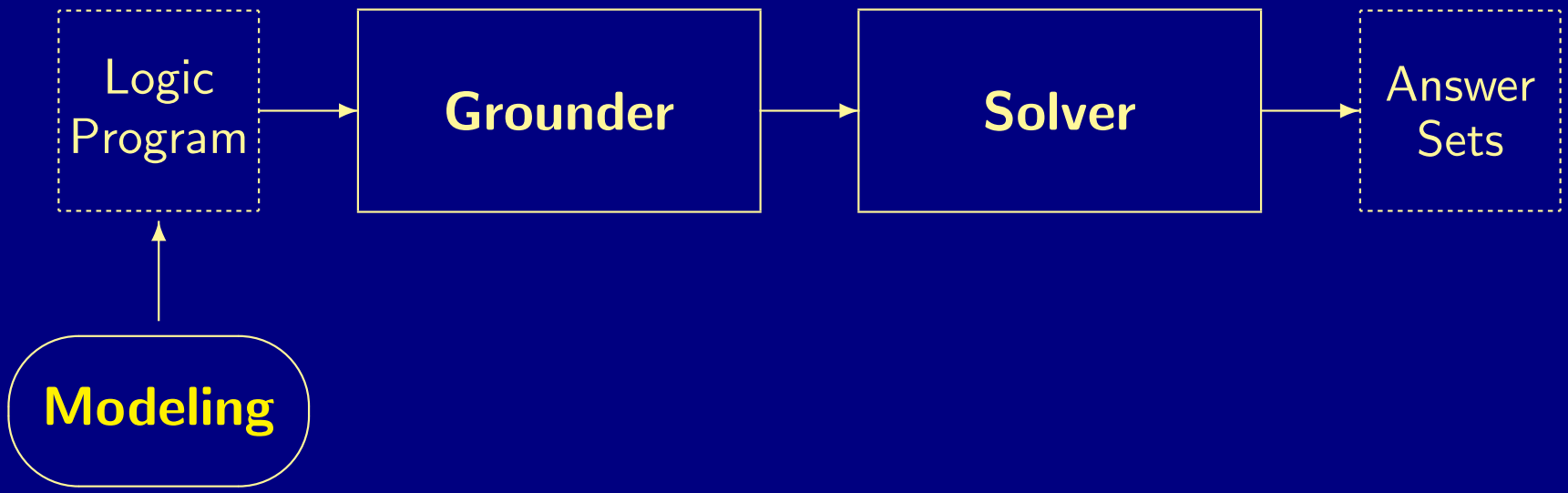
- Basic architecture of *iclingo*:



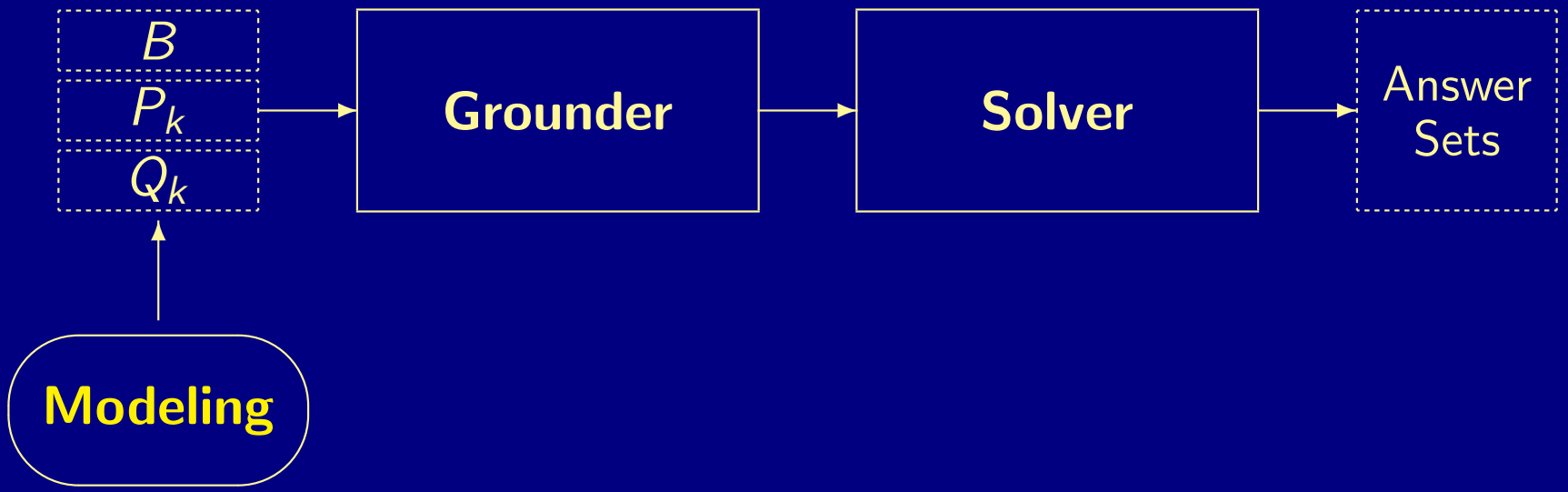
Incremental ASP Solving Process



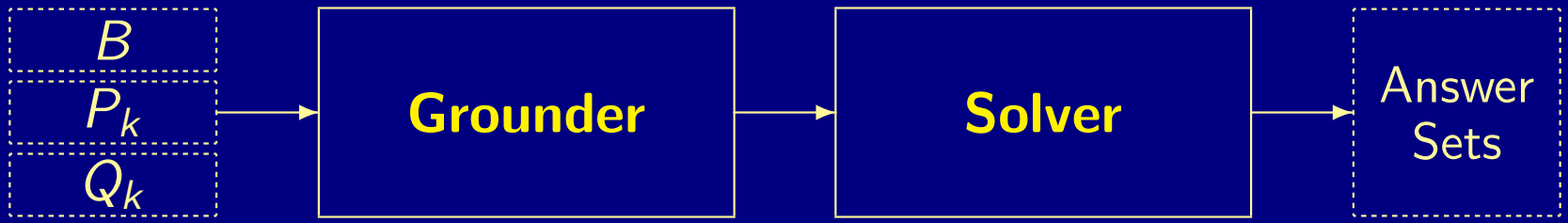
Incremental ASP Solving Process



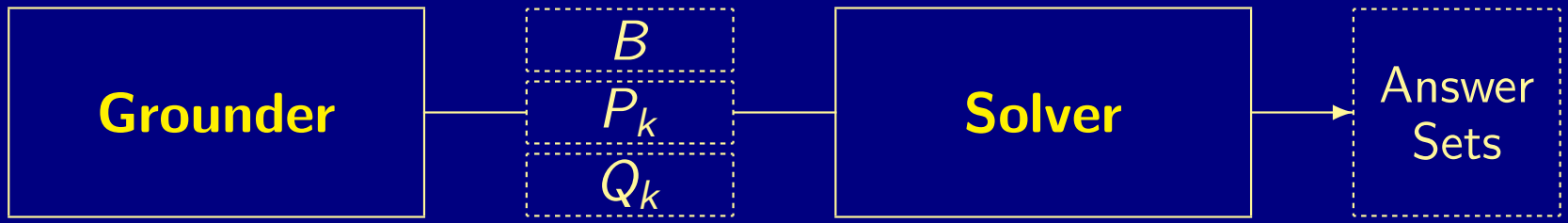
Incremental ASP Solving Process



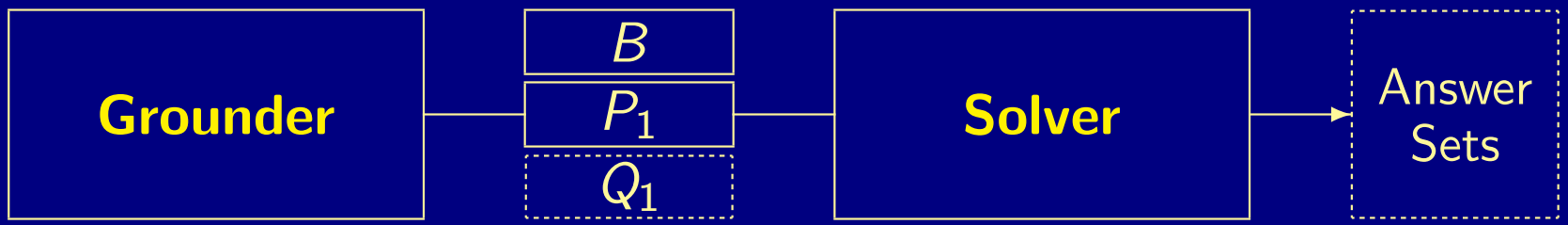
Incremental ASP Solving Process



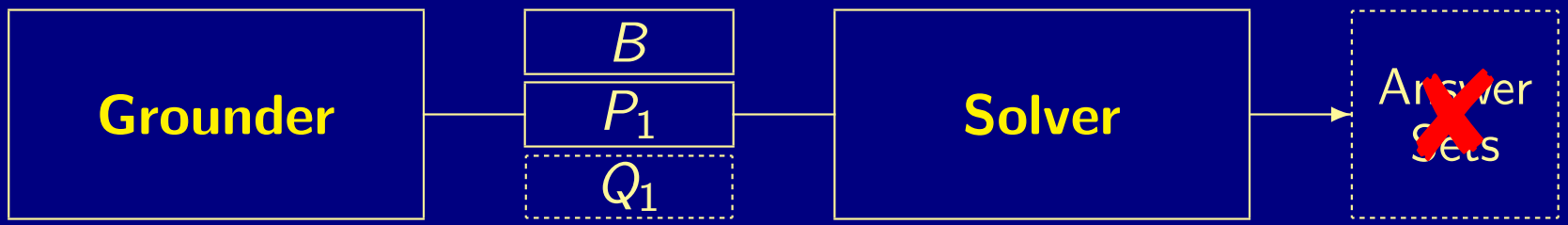
Incremental ASP Solving Process



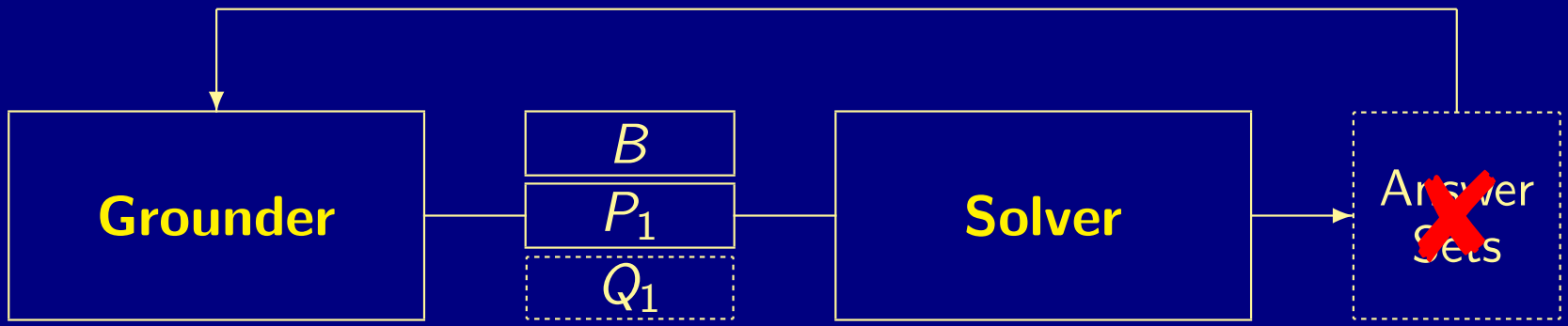
Incremental ASP Solving Process



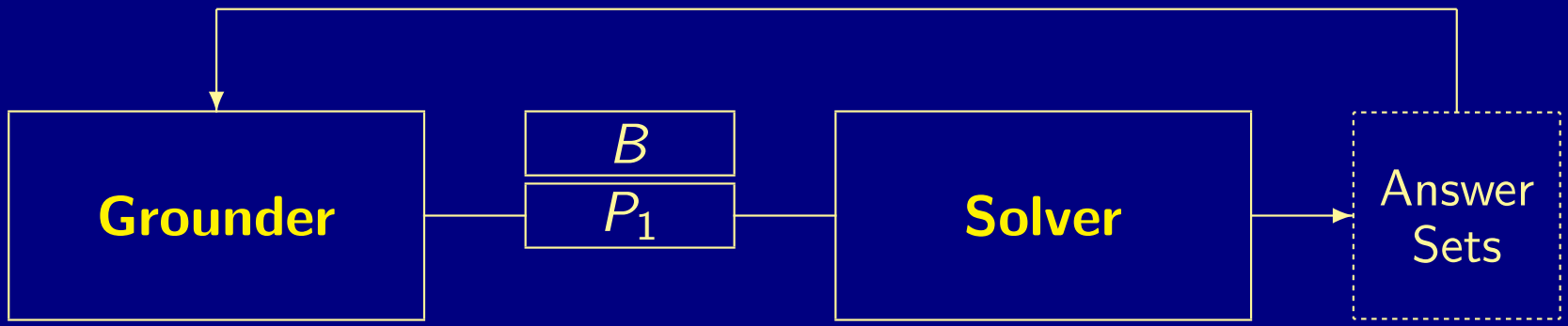
Incremental ASP Solving Process



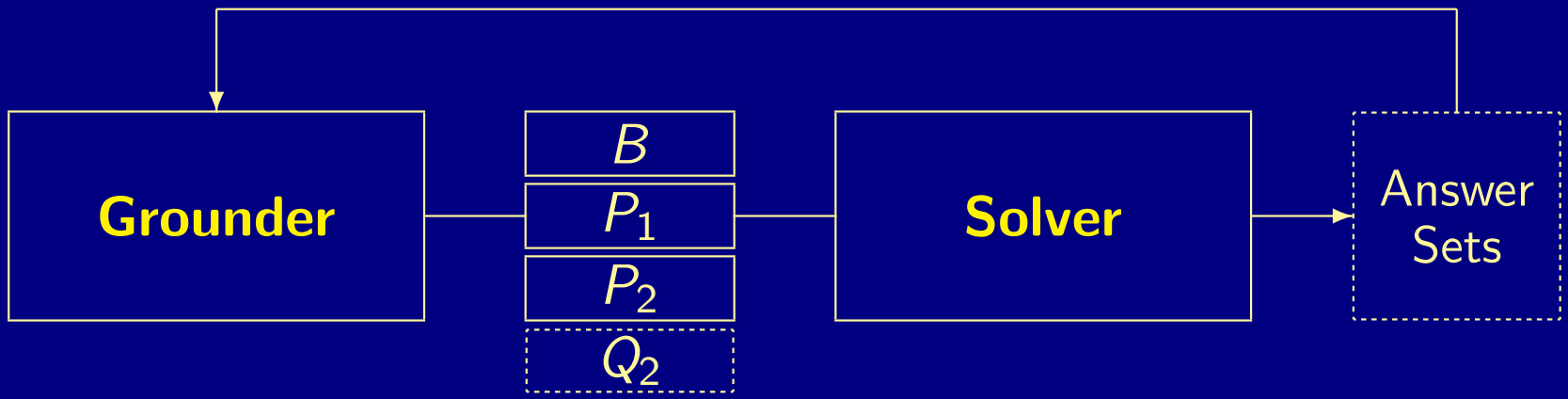
Incremental ASP Solving Process



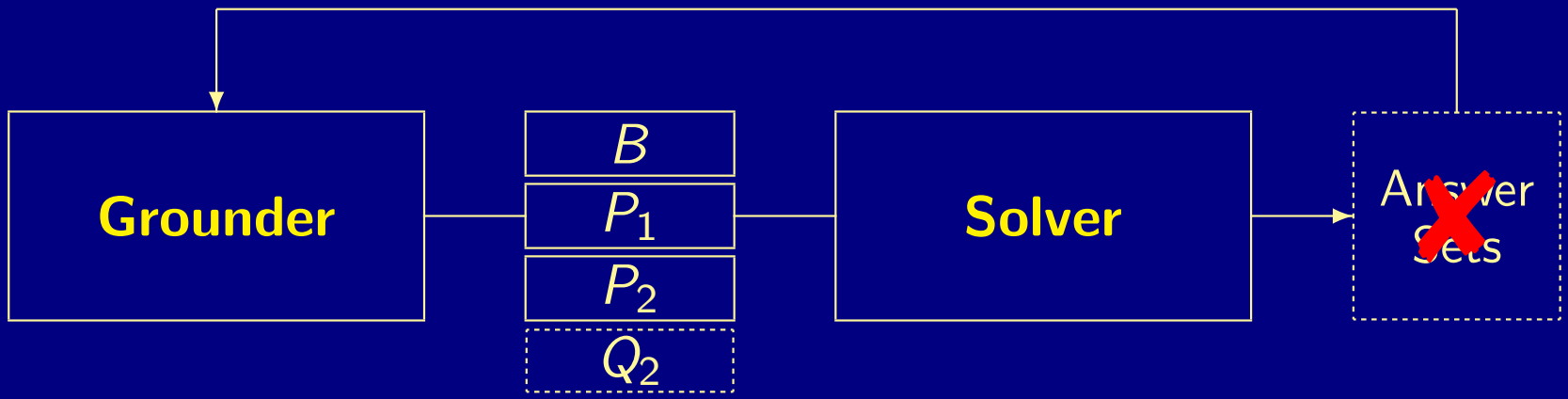
Incremental ASP Solving Process



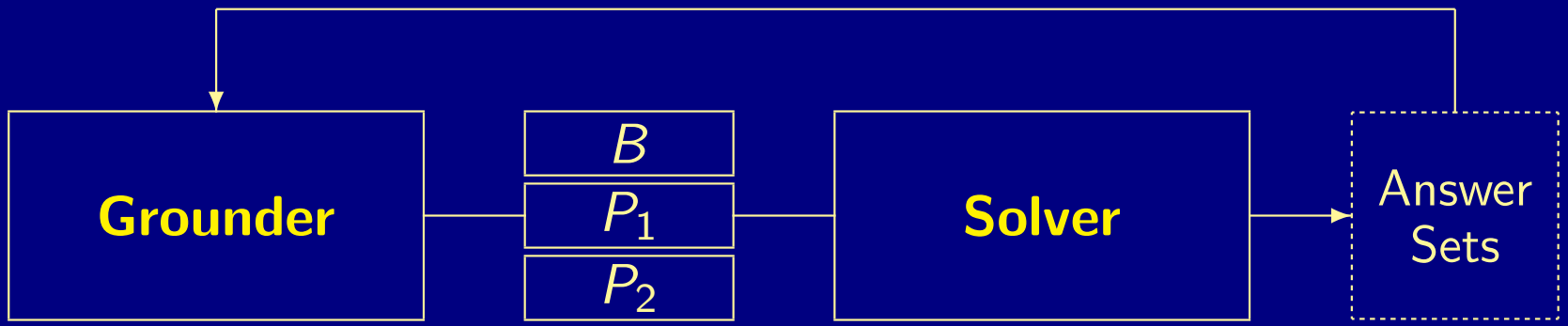
Incremental ASP Solving Process



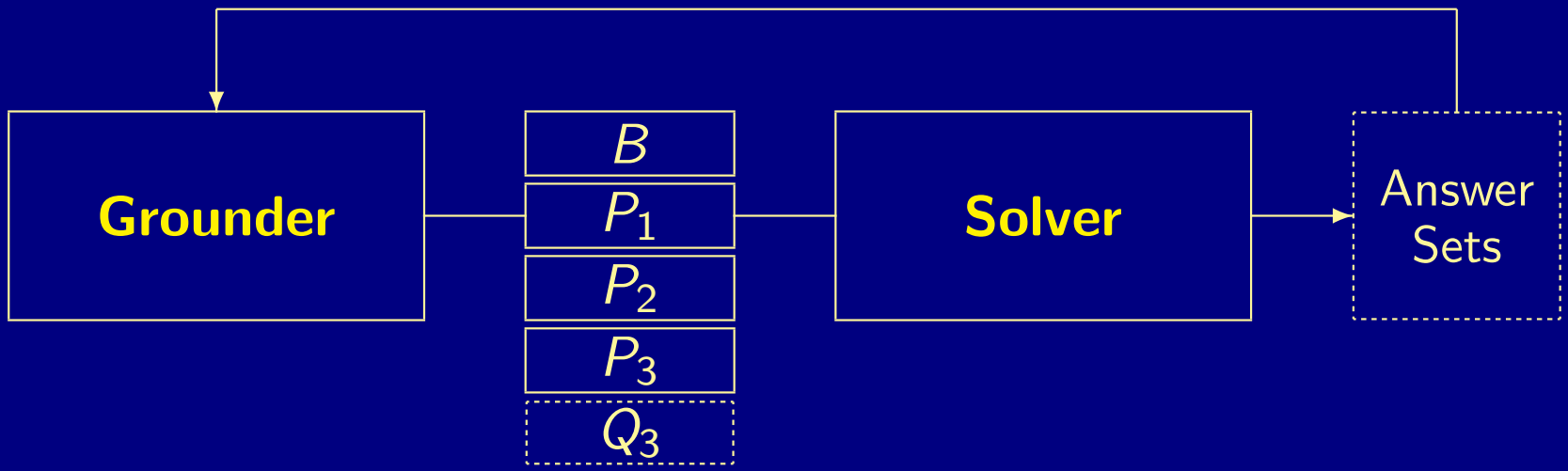
Incremental ASP Solving Process



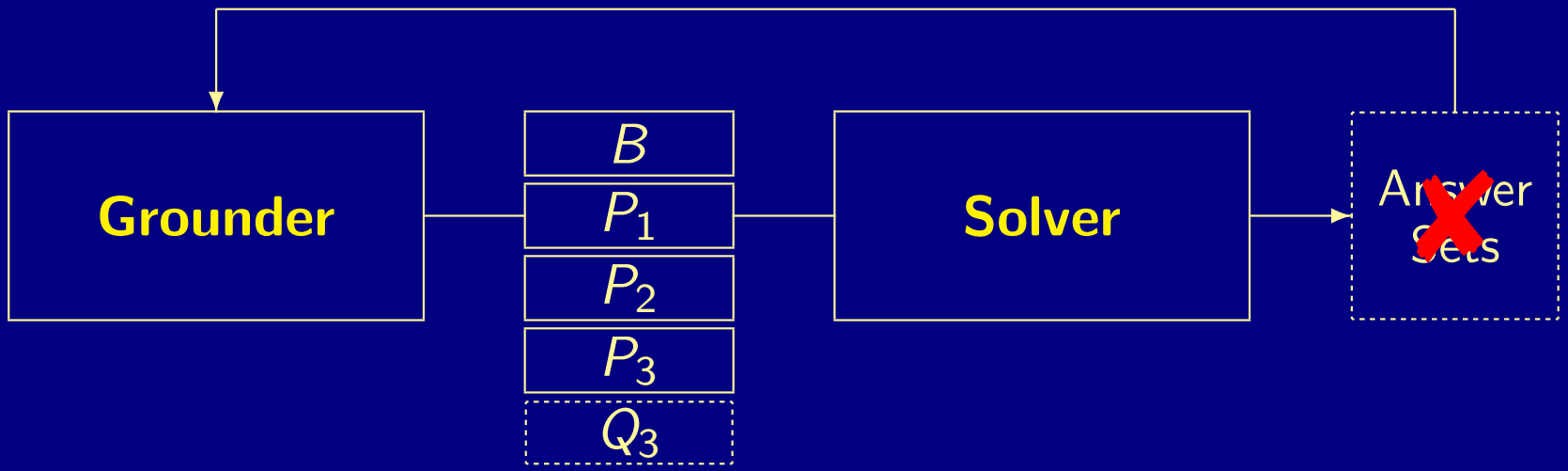
Incremental ASP Solving Process



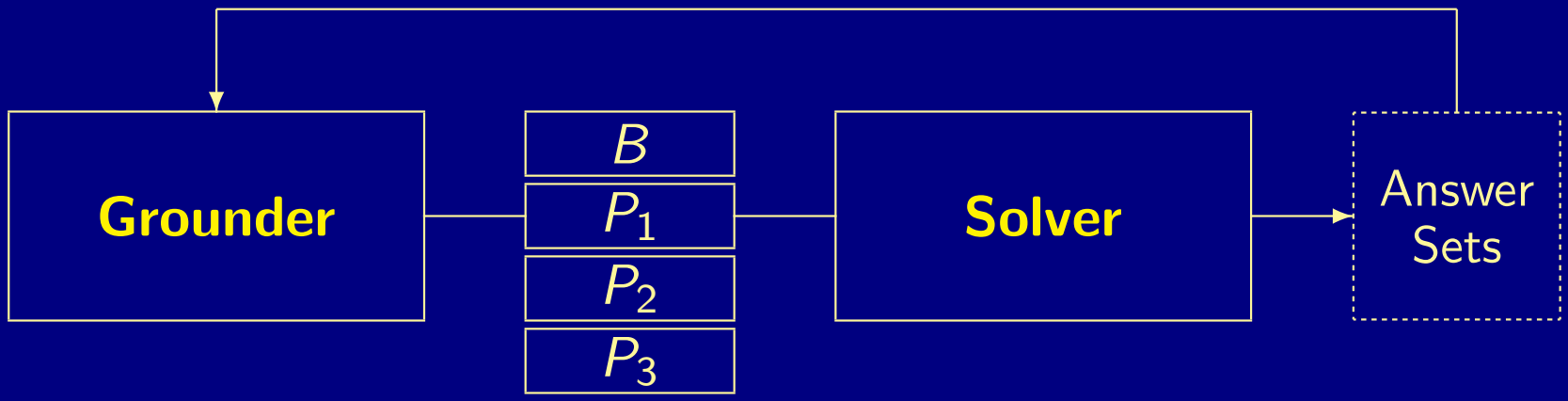
Incremental ASP Solving Process



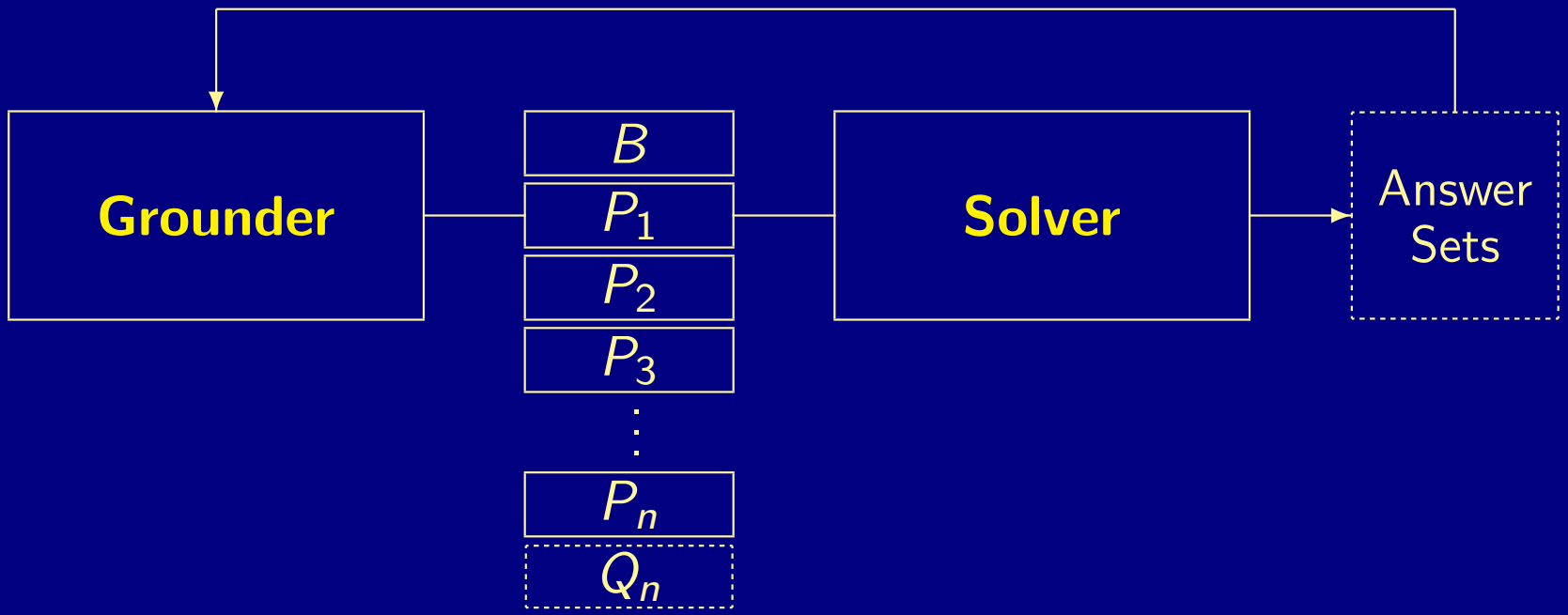
Incremental ASP Solving Process



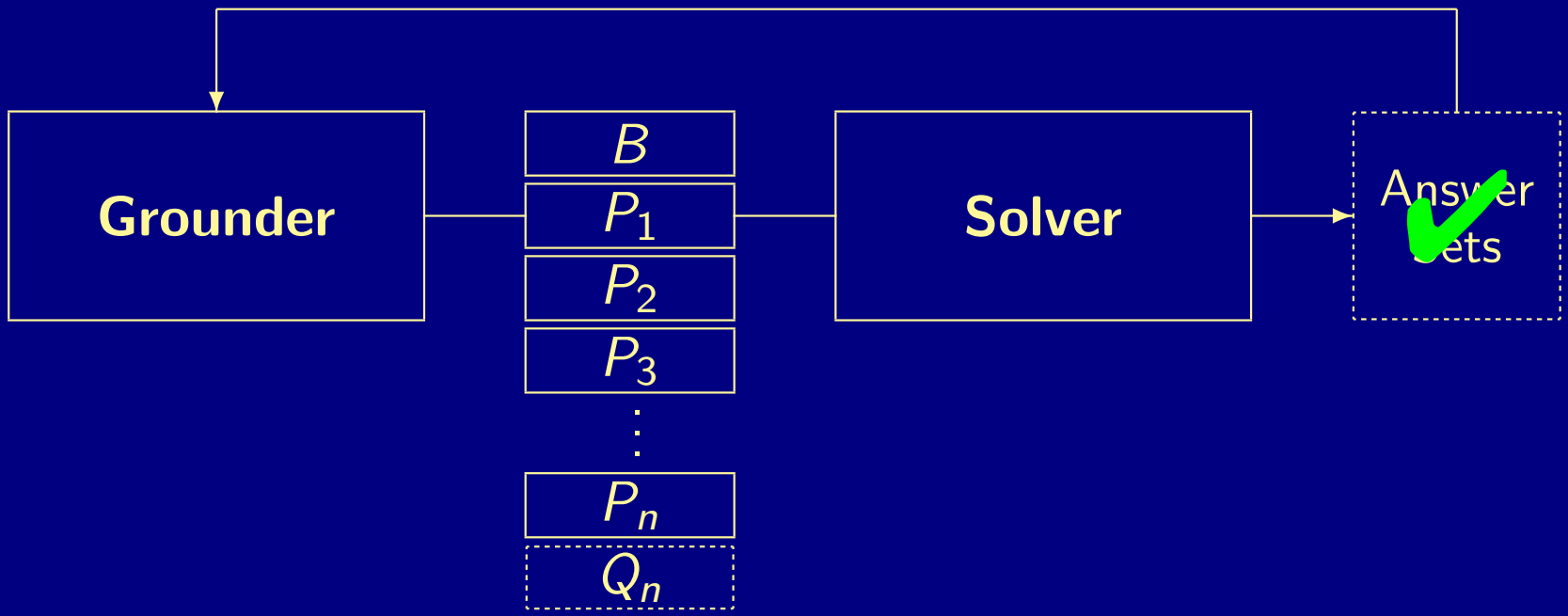
Incremental ASP Solving Process



Incremental ASP Solving Process



Incremental ASP Solving Process



Simplistic STRIPS Planning

```
#base.

fluent(p).     fluent(q).    fluent(r).
action(a).     pre(a,p).     add(a,q).     del(a,p).
action(b).     pre(b,q).     add(b,r).     del(b,q).
init(p).       query(r).

holds(P,0) :- init(P).

#cumulative t.

1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.

 :- query(F), not holds(F,t).
```

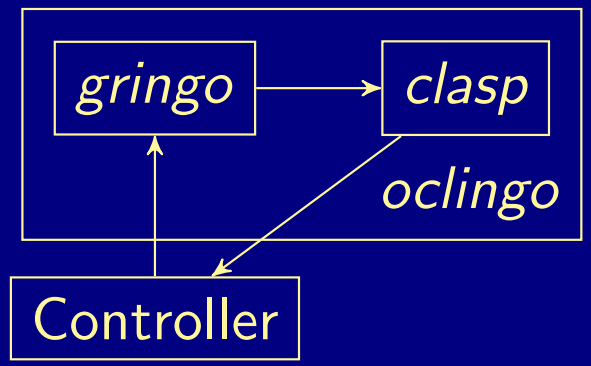
Simplistic STRIPS Planning

```
#base.

fluent(p).     fluent(q).     fluent(r).
action(a).     pre(a,p).      add(a,q).     del(a,p).
action(b).     pre(b,q).      add(b,r).     del(b,q).
init(p).       query(r).

holds(P,0) :- init(P).

#cumulative t.

1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.

 :- query(F), not holds(F,t).
```

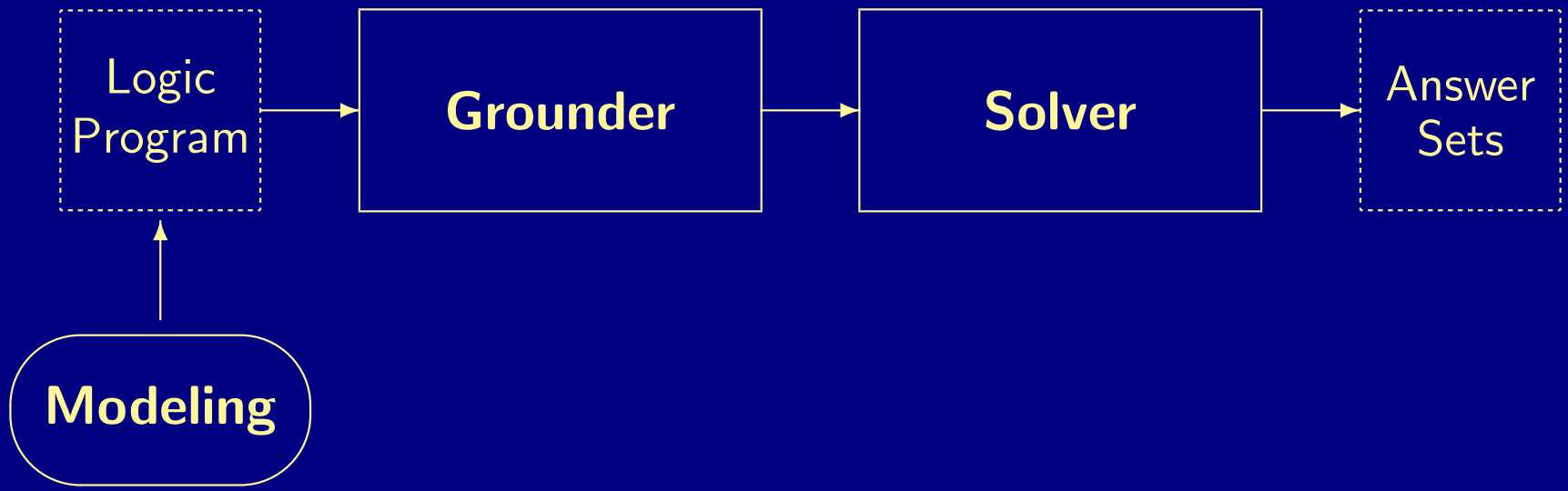
oclingo

- Reactive grounding and solving
- Online solving in dynamic domains, like Robotics

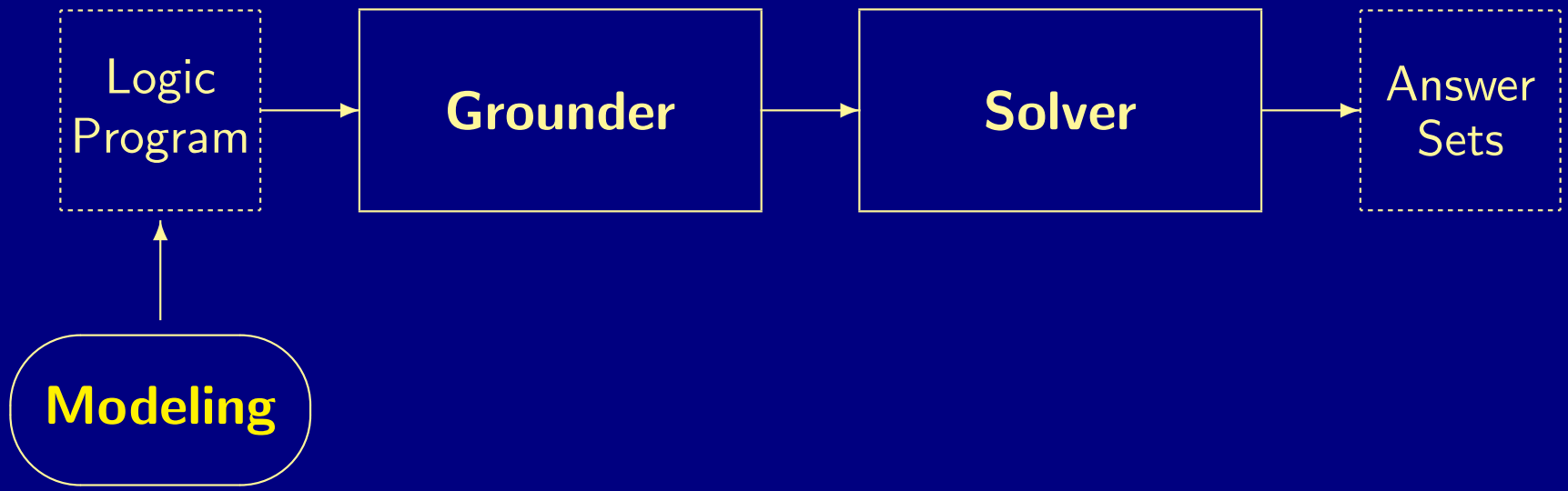
- Basic architecture of *oclingo*:



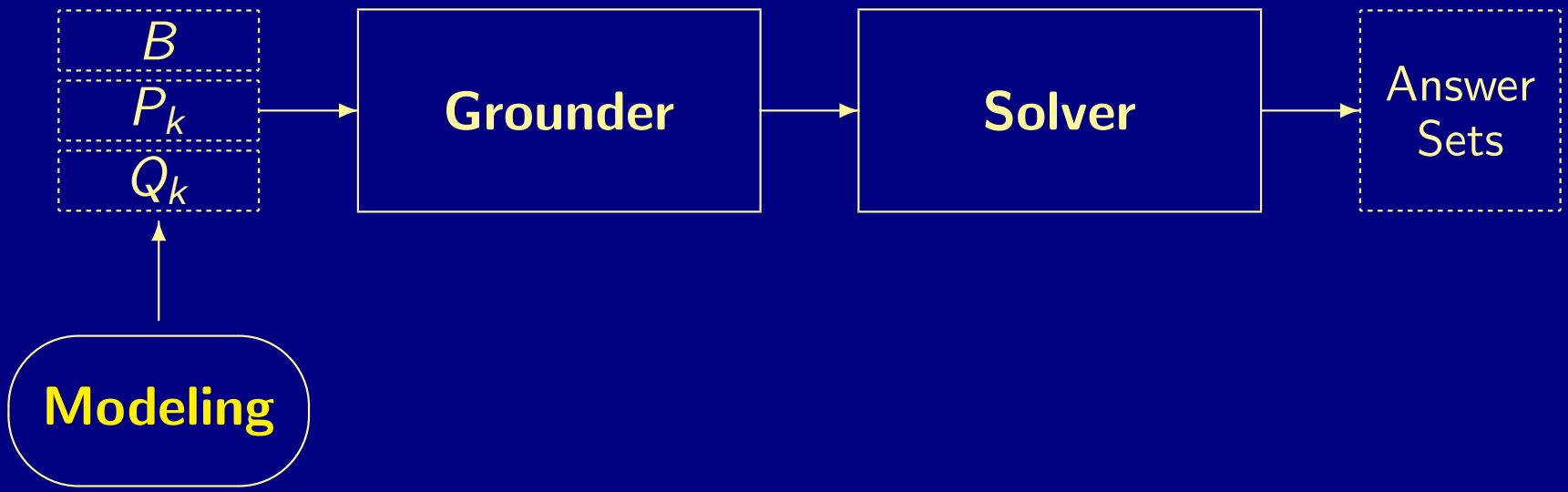
Reactive ASP Solving Process



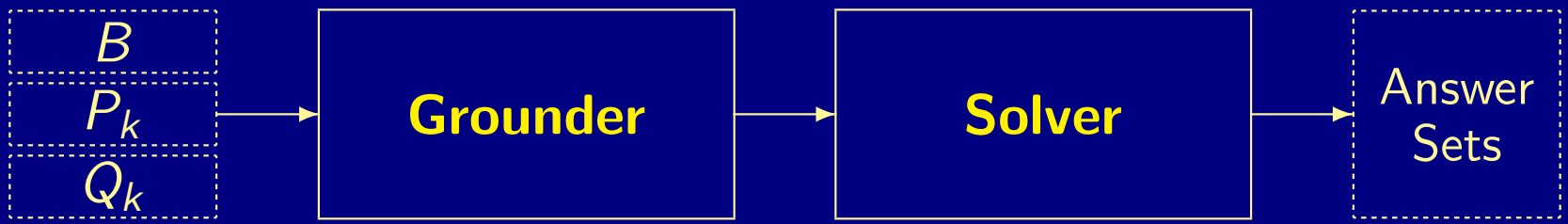
Reactive ASP Solving Process



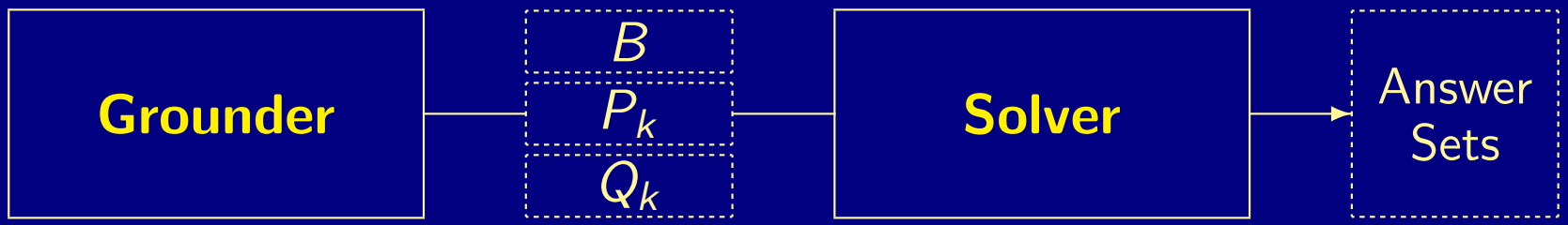
Reactive ASP Solving Process



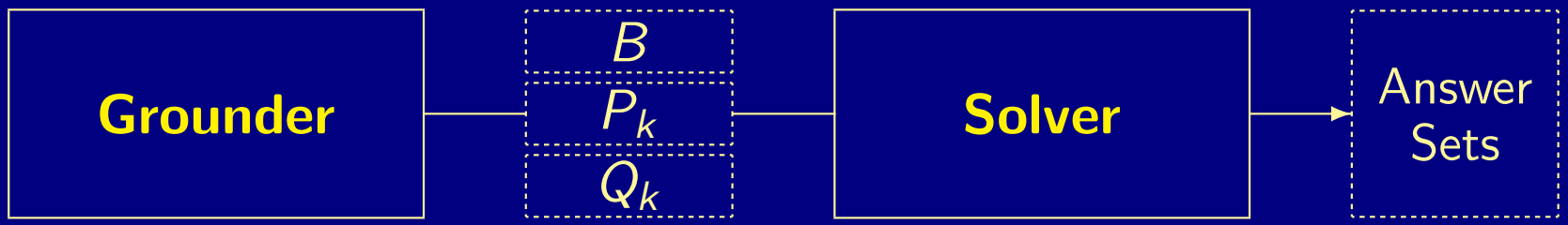
Reactive ASP Solving Process



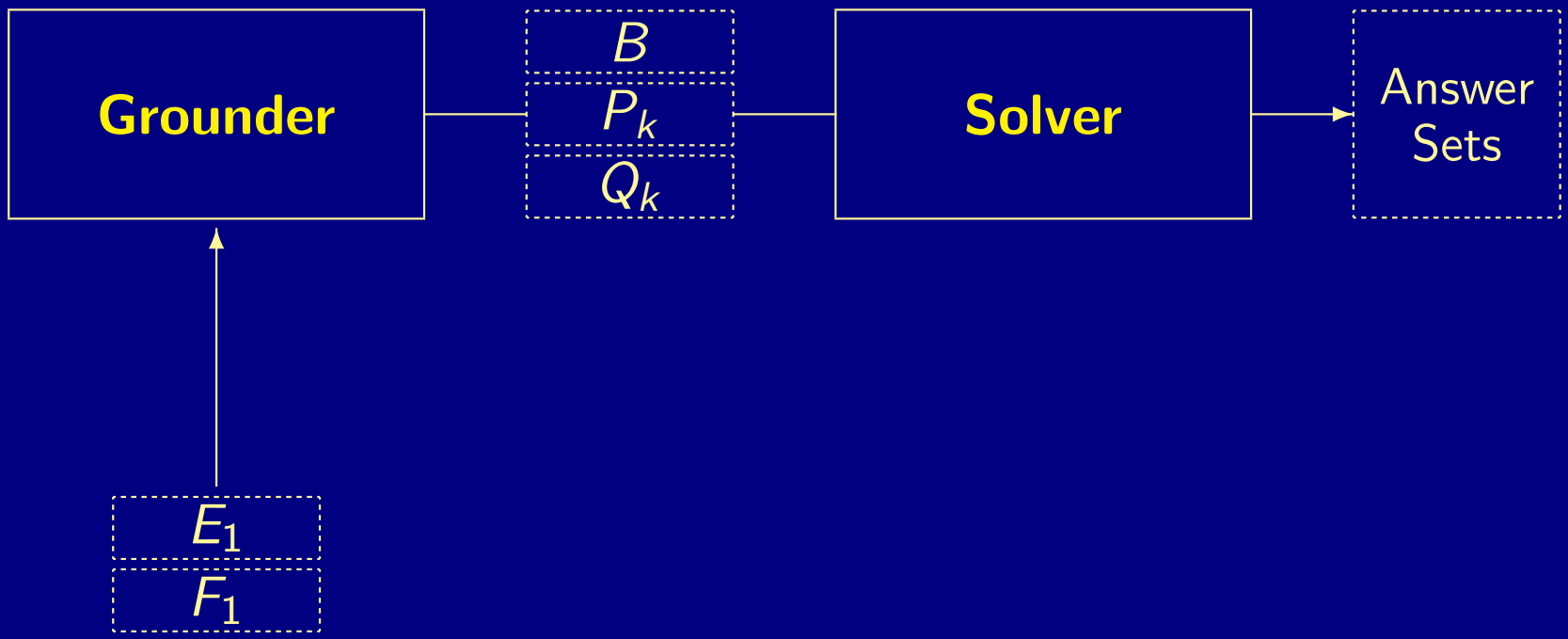
Reactive ASP Solving Process



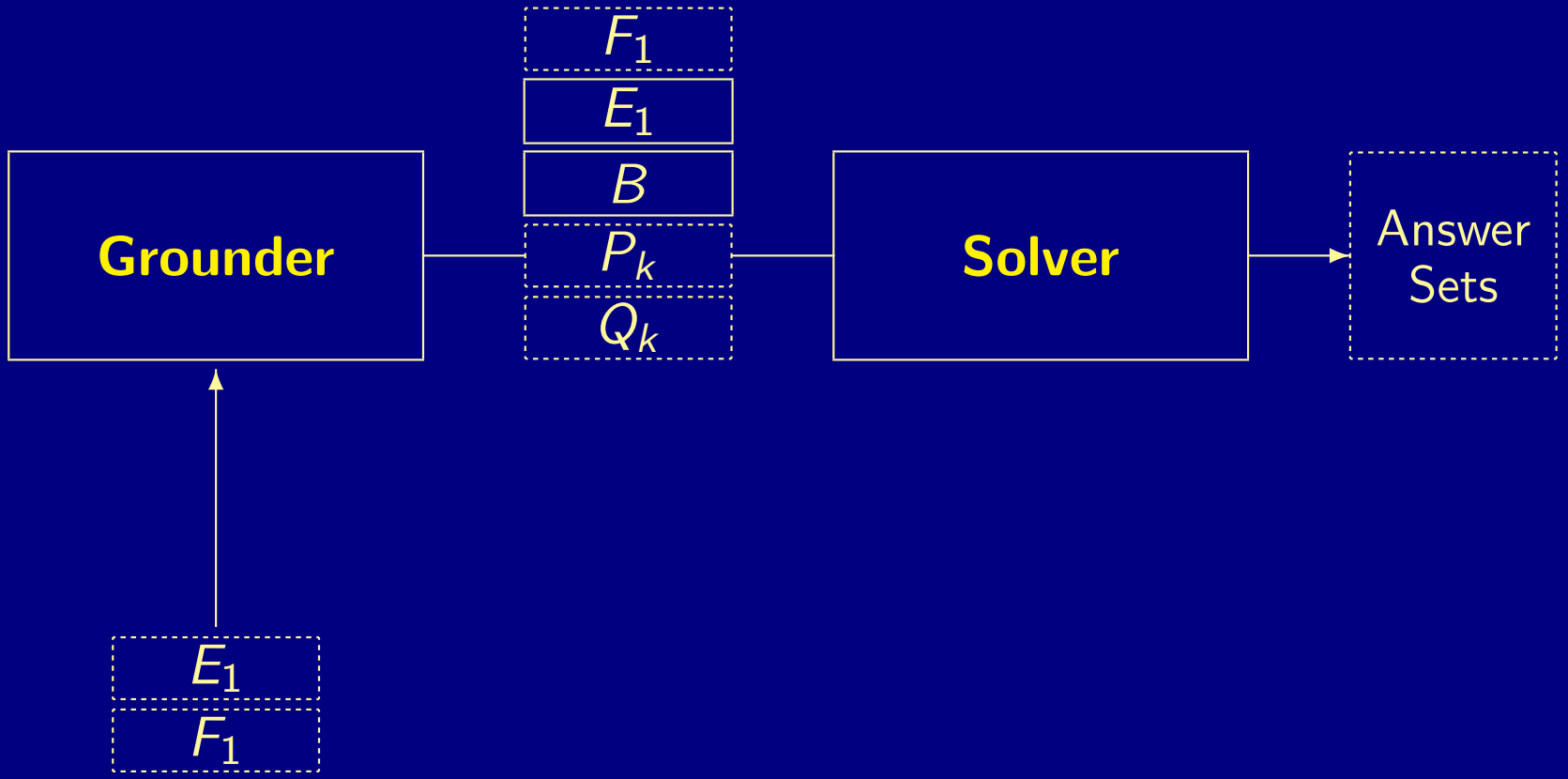
Reactive ASP Solving Process



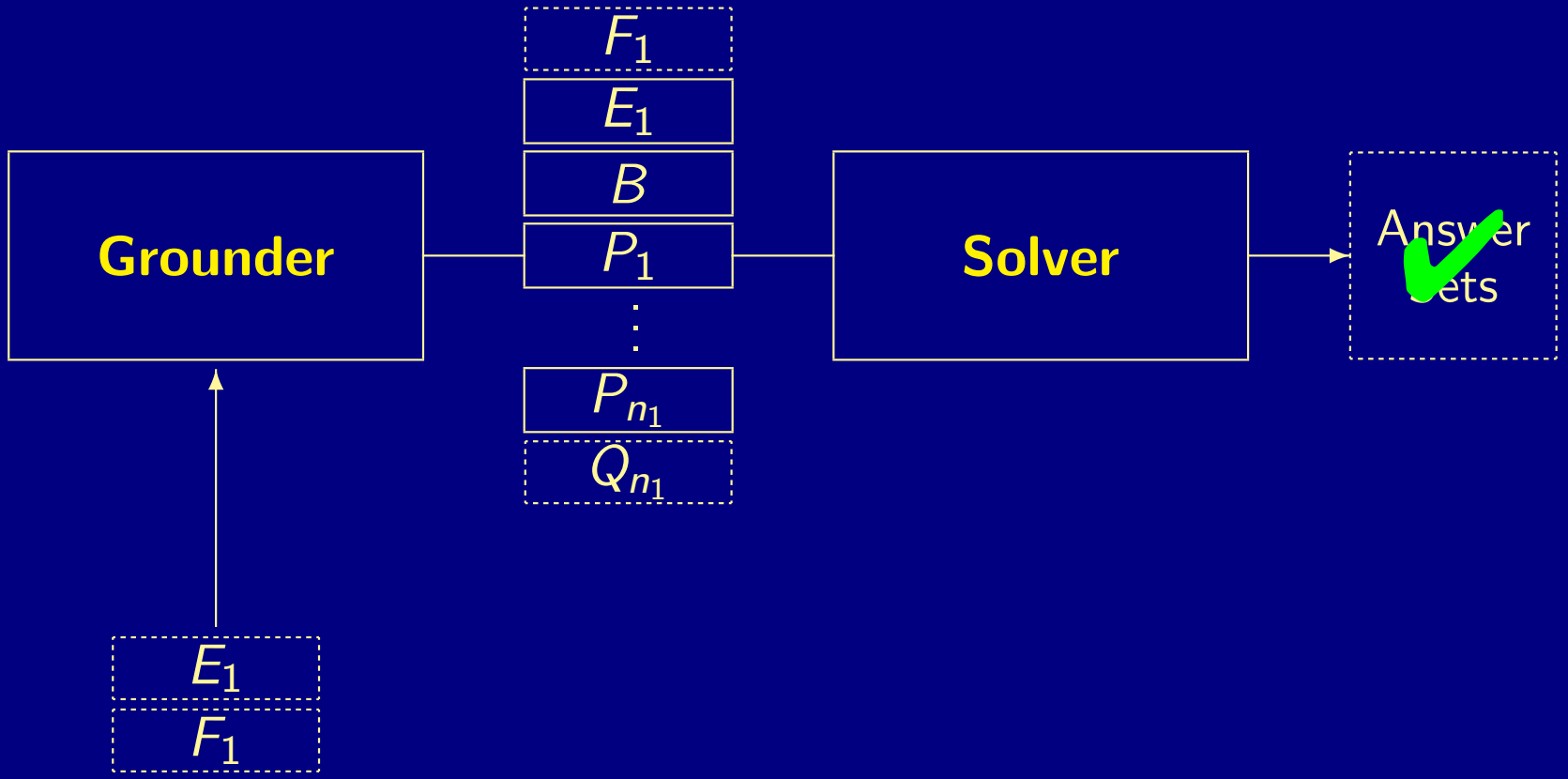
Reactive ASP Solving Process



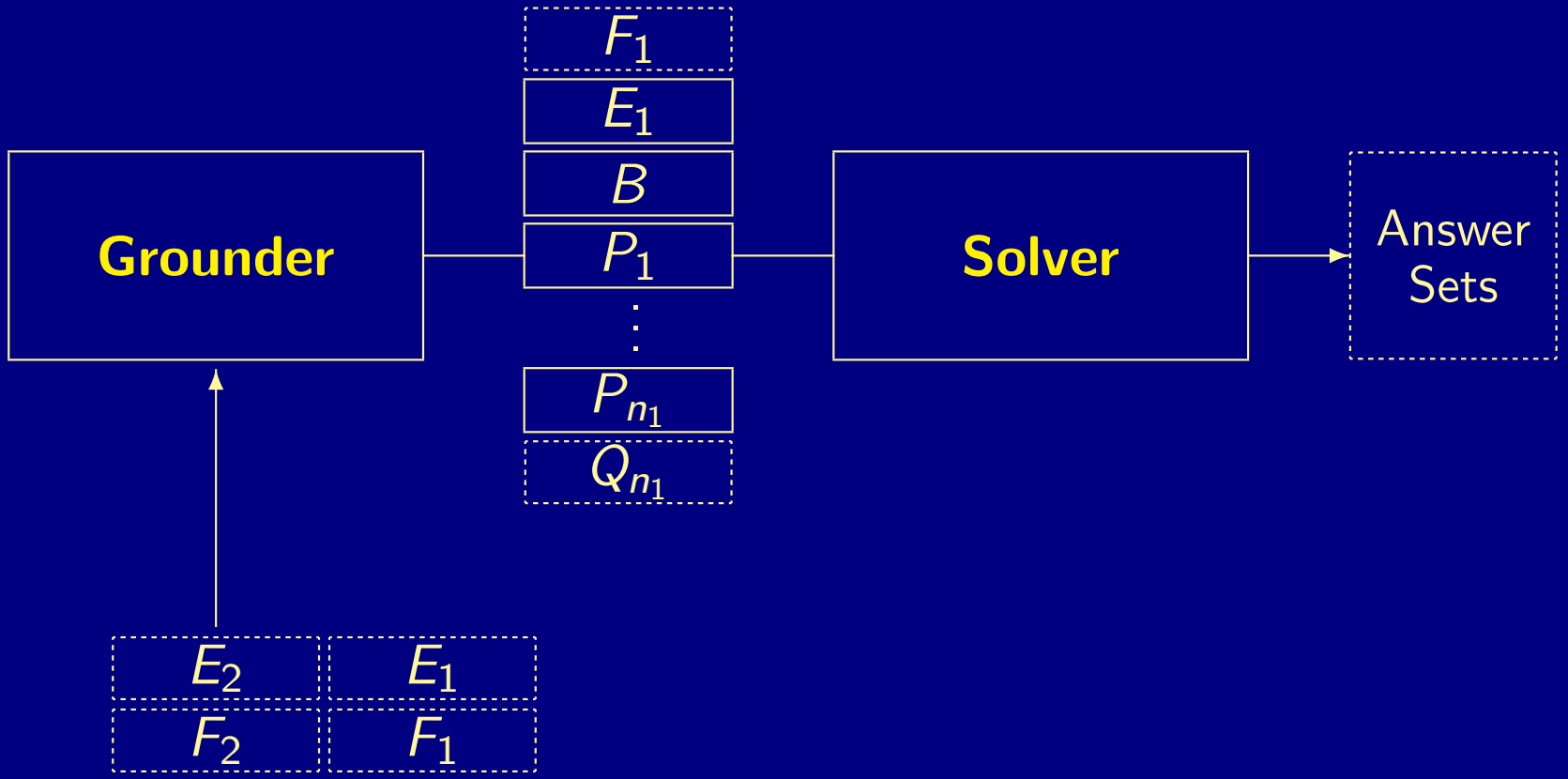
Reactive ASP Solving Process



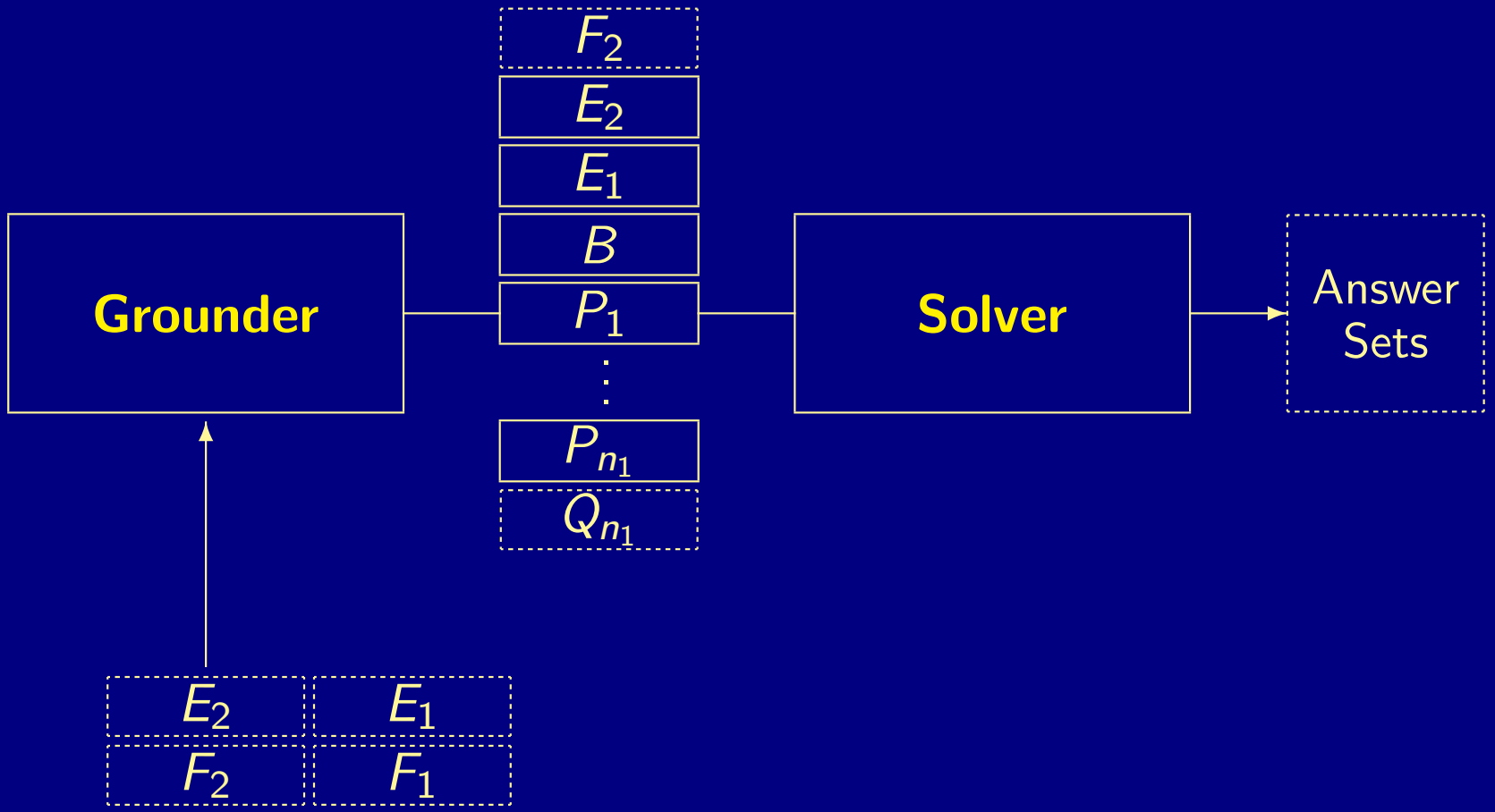
Reactive ASP Solving Process



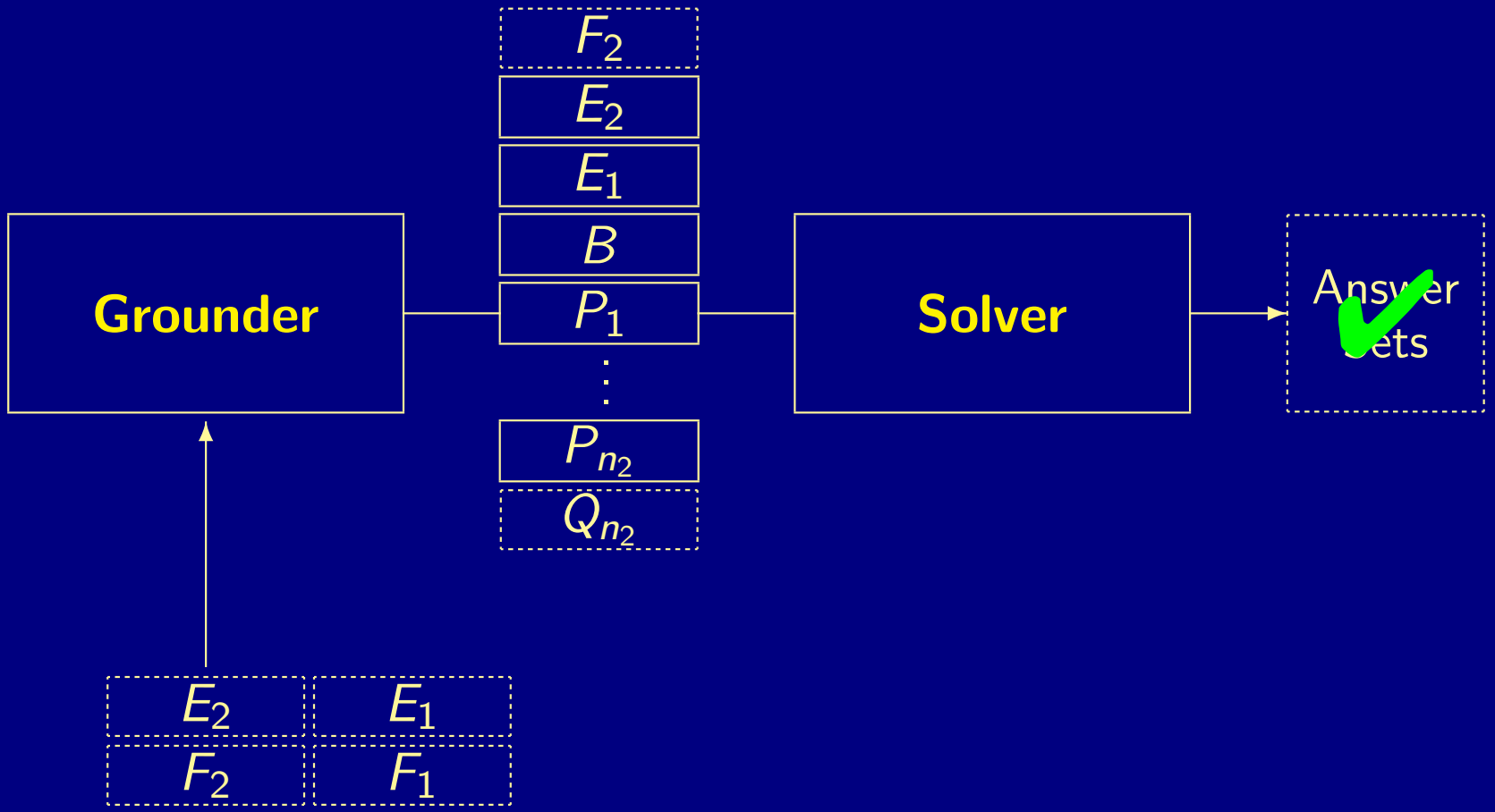
Reactive ASP Solving Process



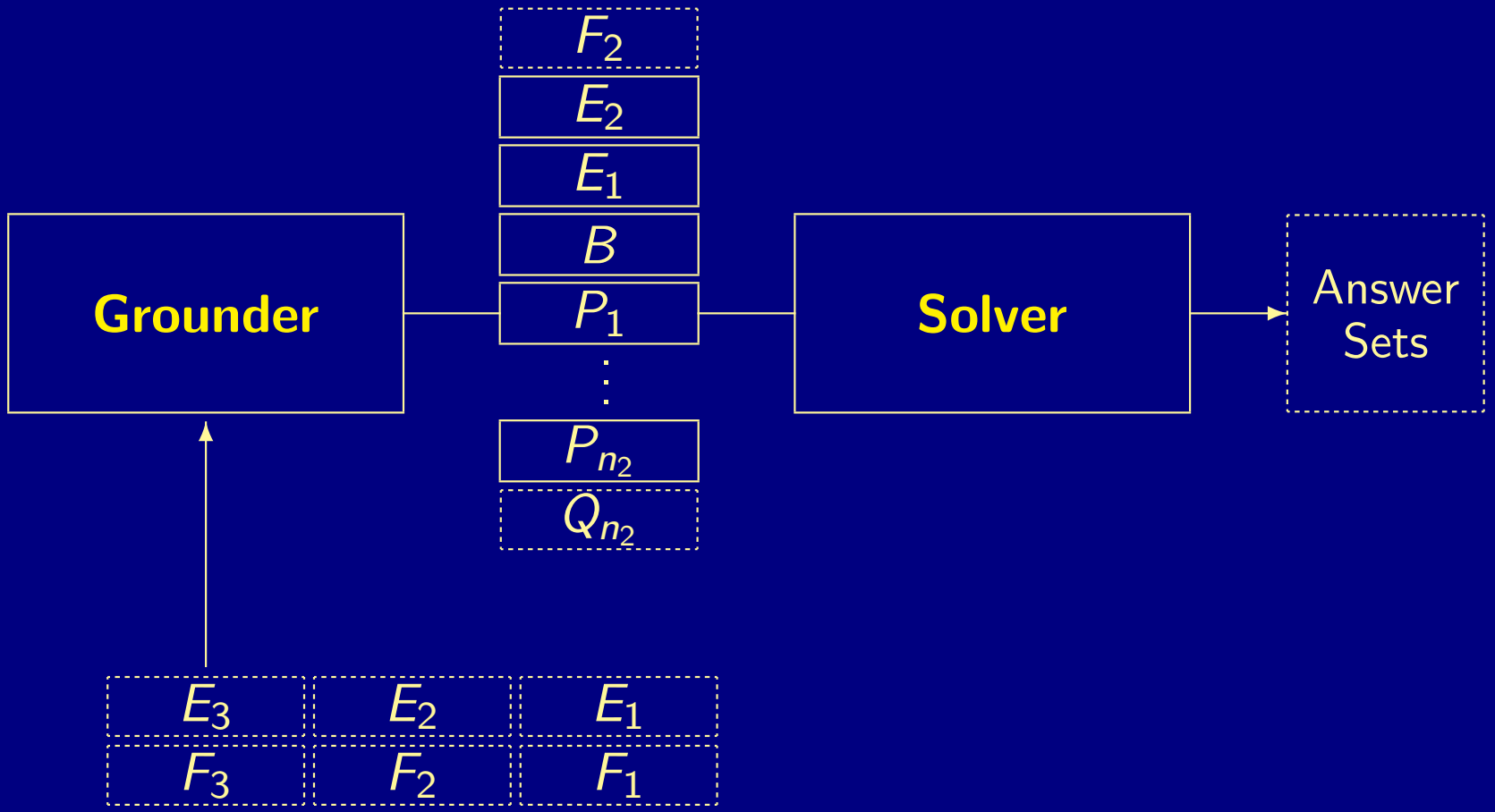
Reactive ASP Solving Process



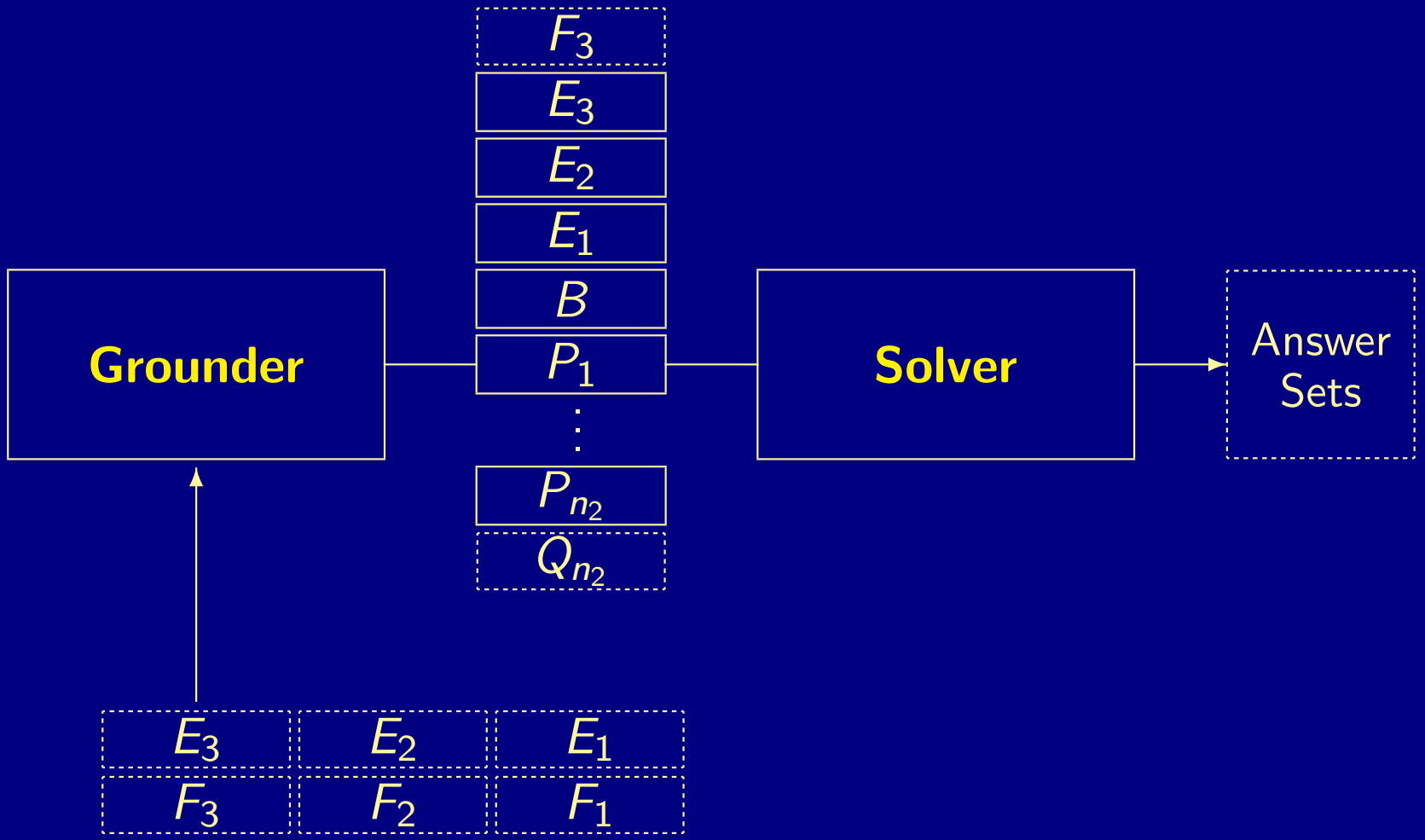
Reactive ASP Solving Process



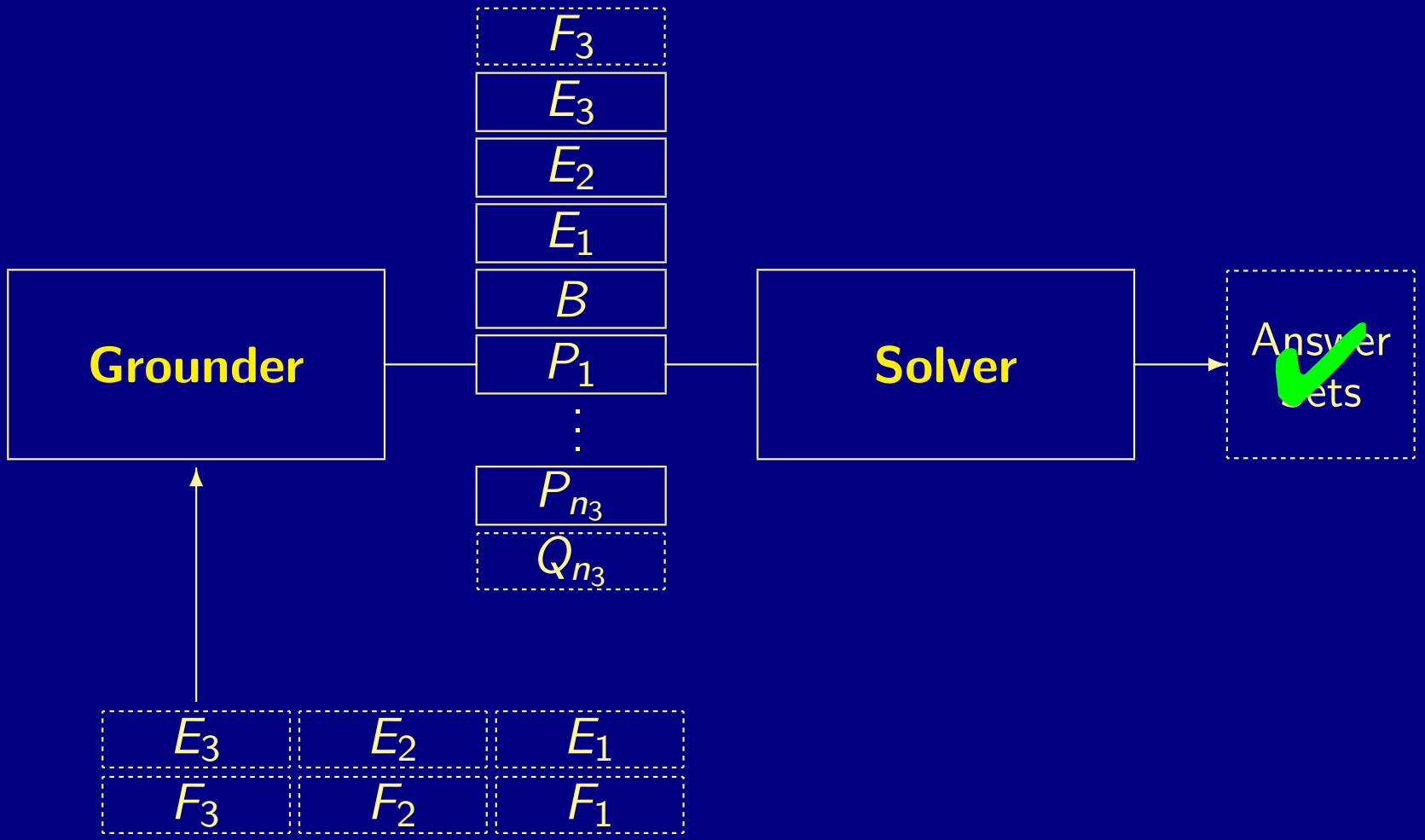
Reactive ASP Solving Process



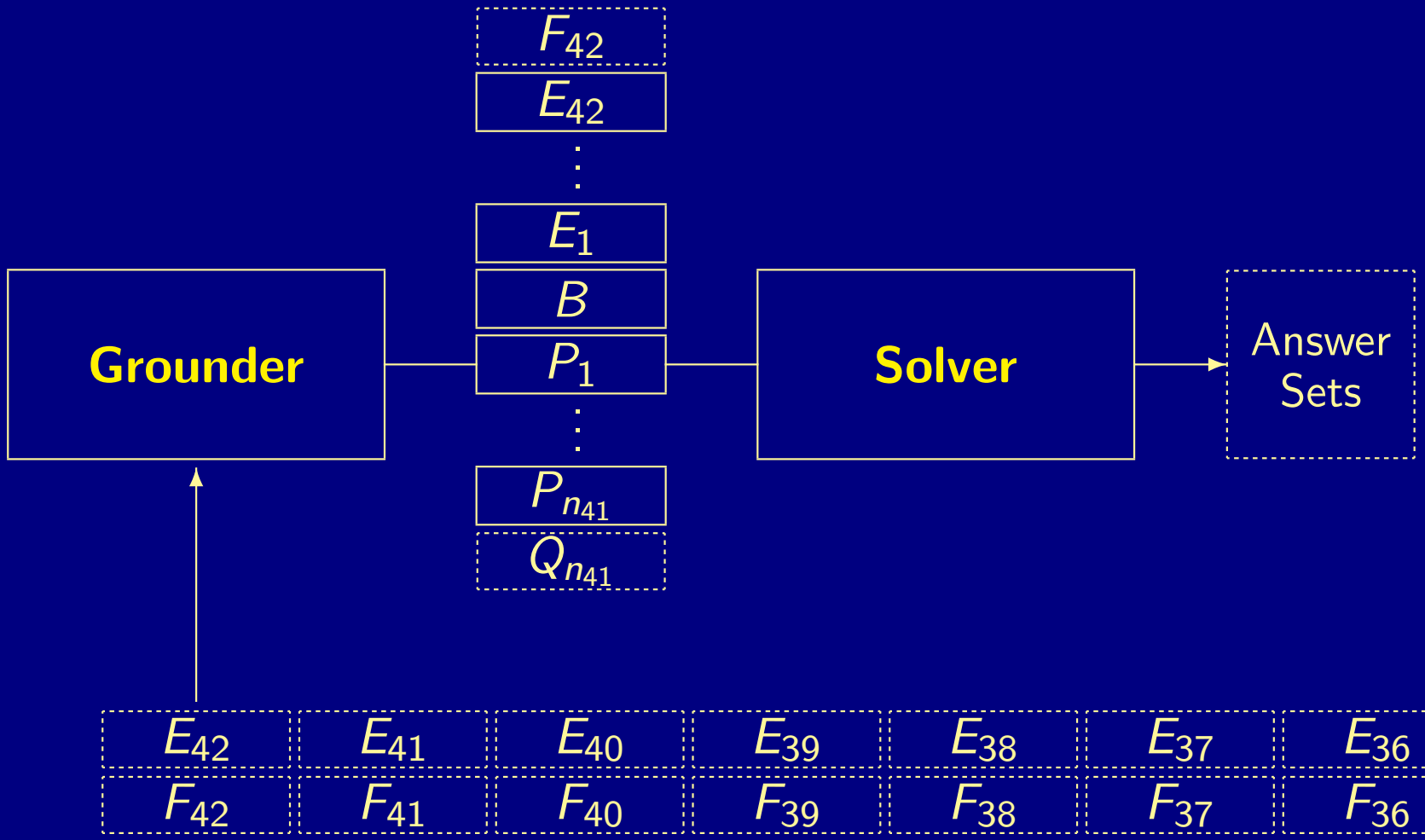
Reactive ASP Solving Process



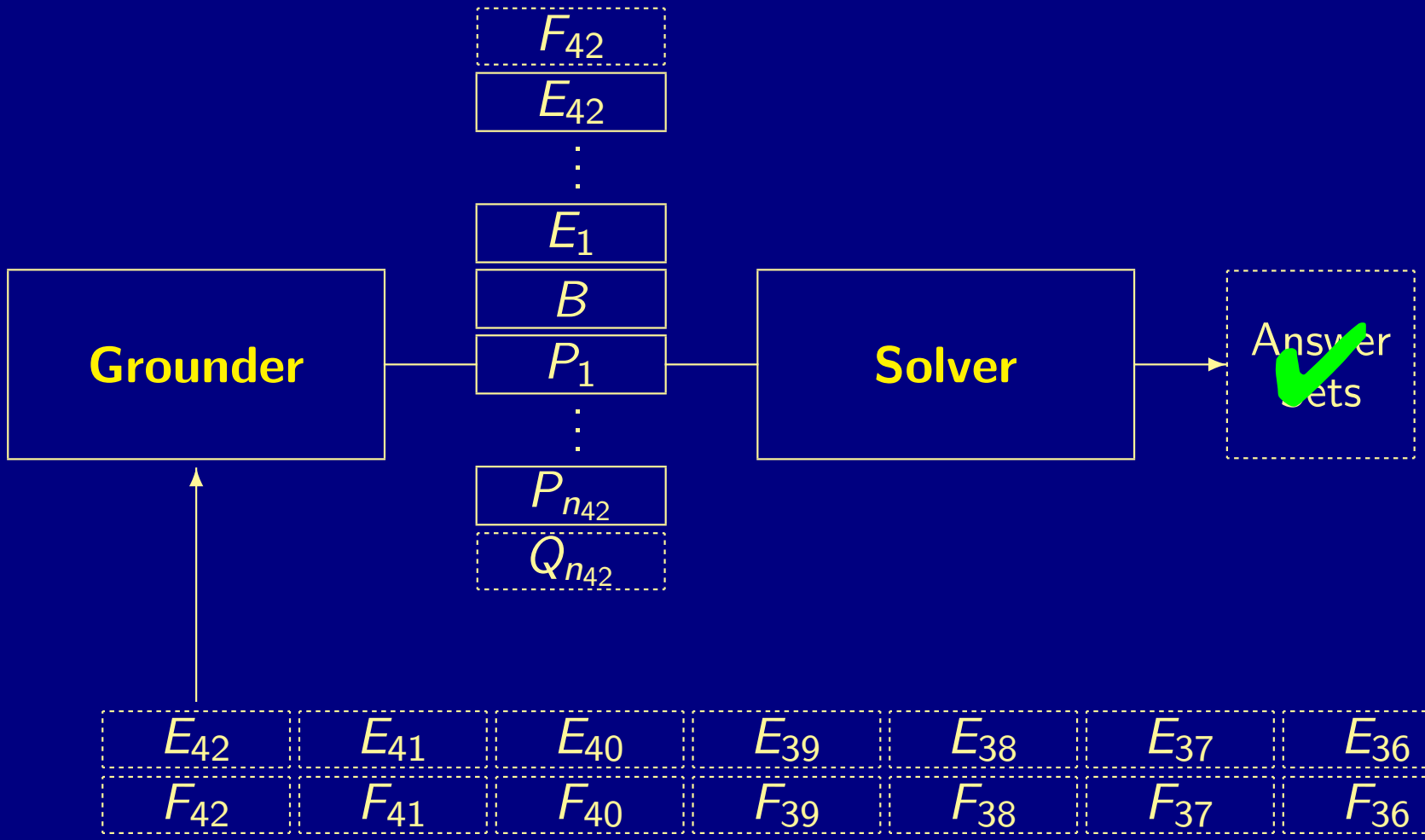
Reactive ASP Solving Process



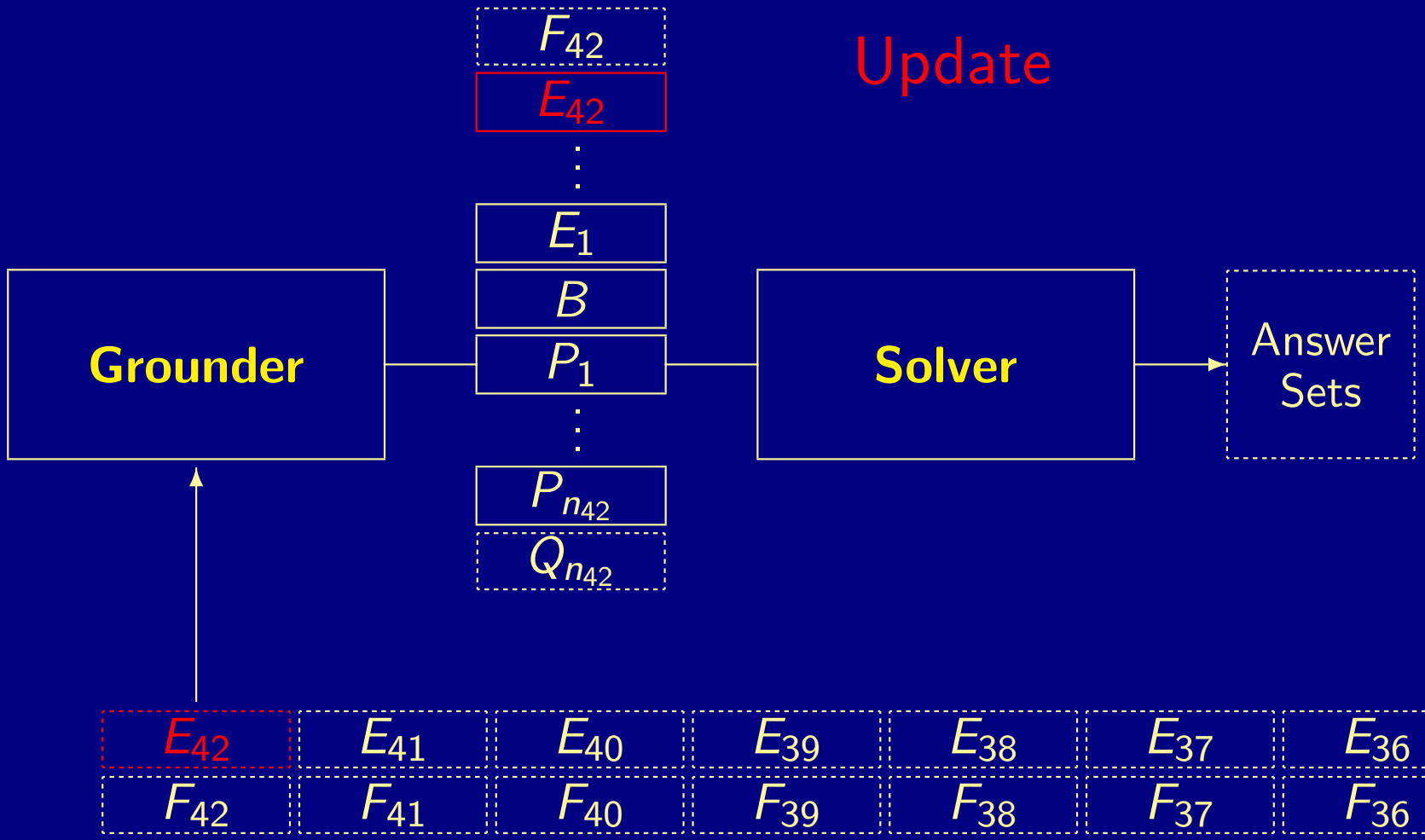
Reactive ASP Solving Process



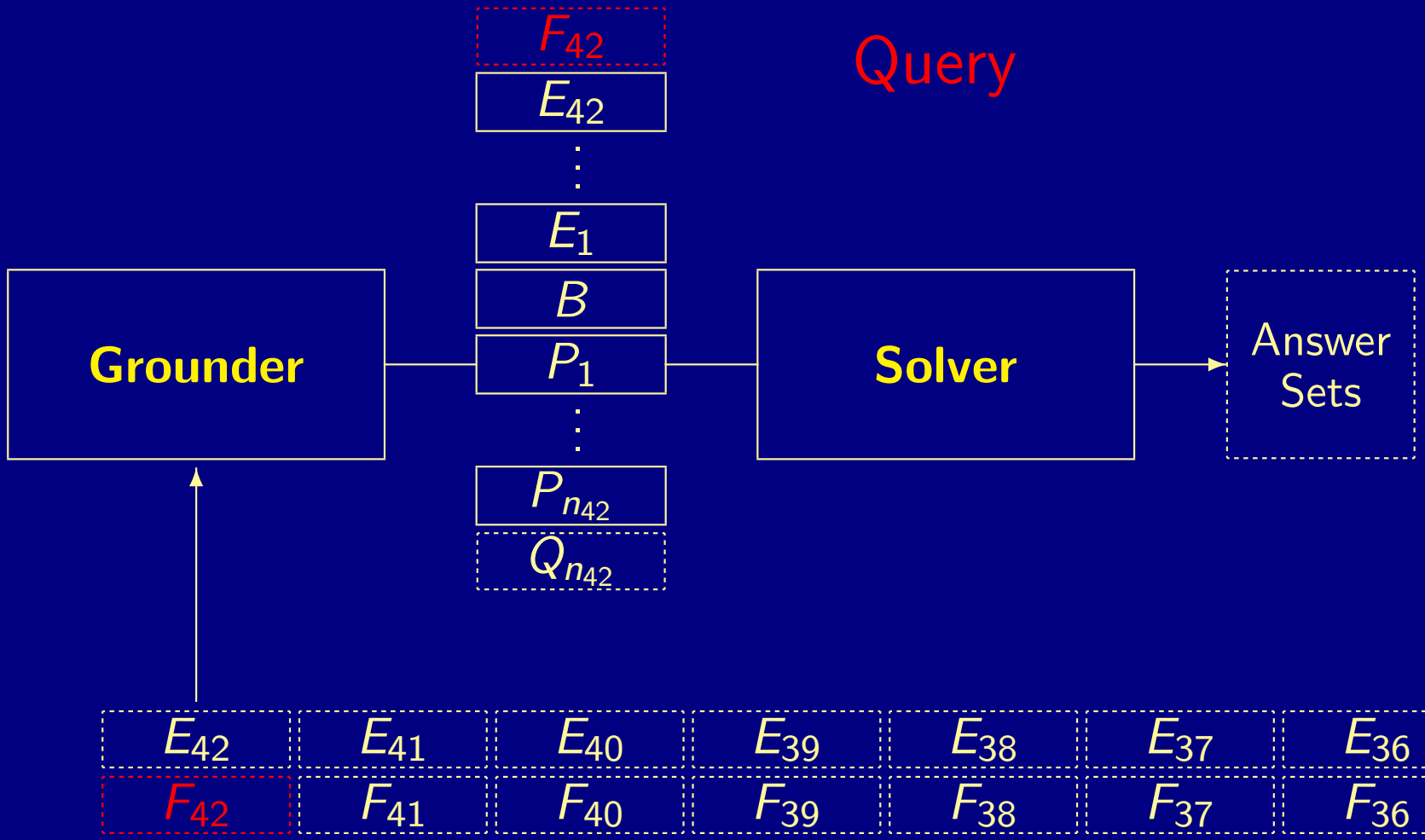
Reactive ASP Solving Process



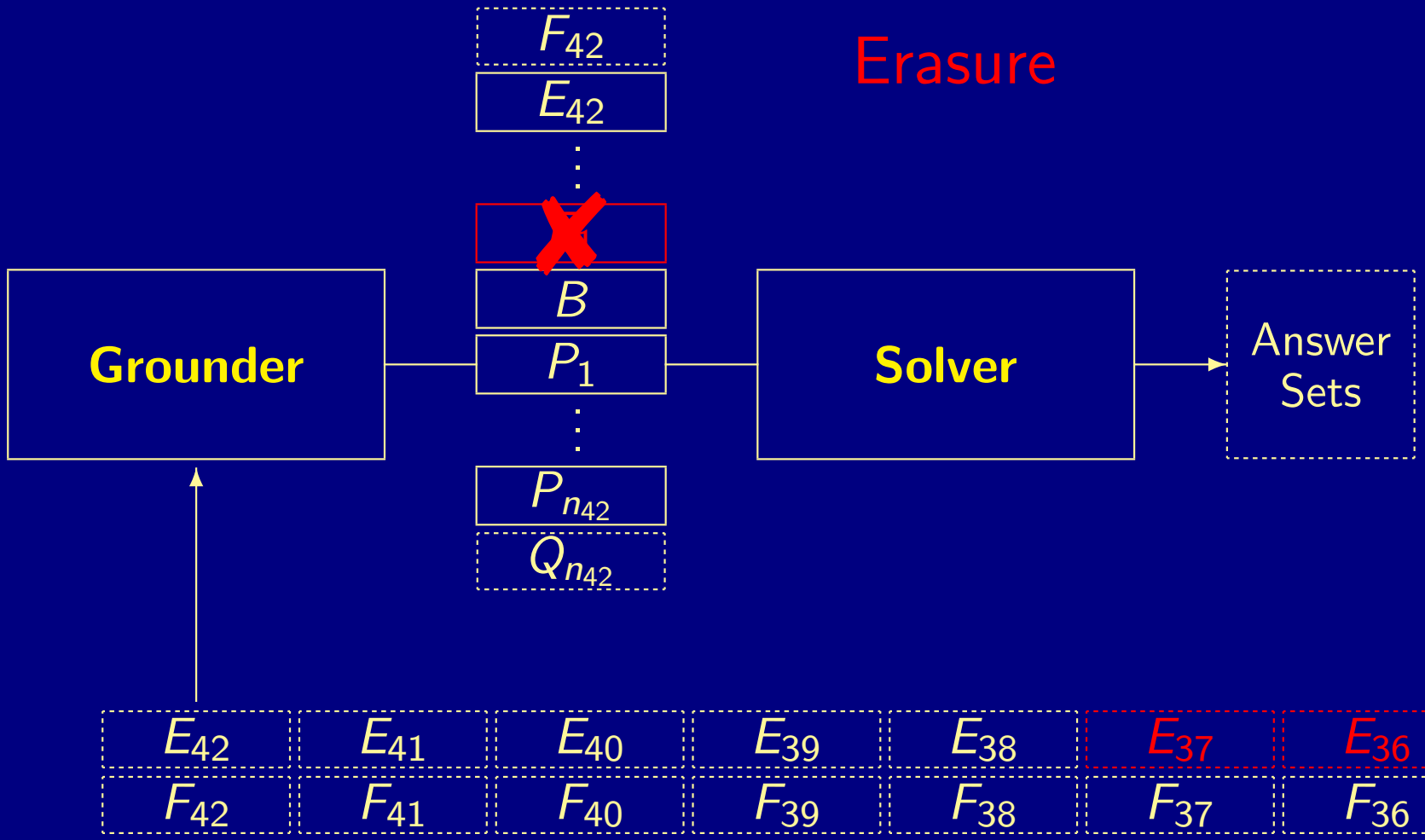
Reactive ASP Solving Process



Reactive ASP Solving Process



Reactive ASP Solving Process



Elevator Control

```
#base.

floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) :  floor(F).

1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t),     floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.

:- not goal(t).
```

Elevator Control

```
#base.

floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) :  floor(F).

1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t),     floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.

:- not goal(t).
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Systems: Overview

31 Potassco

32 gringo

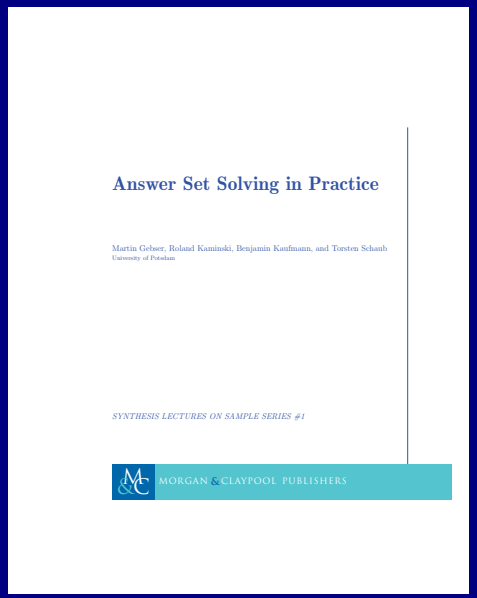
33 clasp

34 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

35 Book

The (forthcoming) Potassco Book

1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions



`http://potassco.sourceforge.net/teaching.html`

Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP'07/09/11, CASC'11, MISC'11, PB'09/11, and SAT'09/11
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

ASP = KR+DB+SAT+LP


C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub.
The `nomore++` approach to answer set solving.
In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 95–109. Springer-Verlag, 2005.


Y. Babovich and V. Lifschitz.
Computing answer sets using program completion.
Unpublished draft; available at `http://www.cs.utexas.edu/users/tag/cmodels.html`, 2003.


C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.
Cambridge University Press, 2003.


C. Baral, G. Brewka, and J. Schlipf, editors.

Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.


S. Baselice, P. Bonatti, and M. Gelfond.
Towards an integration of answer set and constraint solving.
In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.


A. Biere.
Adaptive restart strategies for conflict driven SAT solvers.
In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer-Verlag, 2008.


A. Biere.
PicoSAT essentials.

Journal on Satisfiability, Boolean Modeling and Computation, 4:75–97, 2008.


A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*.
IOS Press, 2009.


M. Brain, O. Cliffe, and M. de Vos.
A pragmatic programmer's guide to answer set programming.
In M. de Vos and T. Schaub, editors, *Proceedings of the Second Workshop on Software Engineering for Answer Set Programming (SEA'09)*, volume 546, pages 49–63. CEUR Workshop Proceedings, 2009.


M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran.
Debugging ASP programs by means of ASP.
In Baral et al. , pages 31–43.


M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran.
That is illogical captain! — the debugging support tool spock for answer-set programs: System description.
In M. de Vos and T. Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, volume 281, pages 71–85. CEUR Workshop Proceedings, 2007.


K. Clark.
Negation as failure.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.


O. Cliffe, M. de Vos, M. Brain, and J. Padget.
ASPVIZ: Declarative visualisation and animation using answer set programming.
In Garcia de la Banda and Pontelli , pages 724–728.


M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.
Handbook of Tableau Methods.

Kluwer Academic Publishers, 1999.


E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov.
Complexity and expressive power of logic programming.
In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity (CCC'97)*, pages 82–101. IEEE Computer Society Press, 1997.


M. Davis, G. Logemann, and D. Loveland.
A machine program for theorem-proving.
Communications of the ACM, 5:394–397, 1962.


M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.


C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub.
Conflict-driven disjunctive answer set solving.

In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008.


C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub.
Heuristics in conflict resolution.
In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, number UNSW-CSE-TR-0819 in School of Computer Science and Engineering, The University of New South Wales, Technical Report Series, pages 141–149, 2008.


N. Eén and N. Sörensson.
An extensible SAT-solver.
In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004.


T. Eiter and G. Gottlob.

On the computational cost of disjunctive logic programming: Propositional case.
Annals of Mathematics and Artificial Intelligence, 15(3-4):289–323, 1995.


F. Fages.
Consistency of Clark's completion and the existence of stable models.
Journal of Methods of Logic in Computer Science, 1:51–60, 1994.


P. Ferraris.
Answer sets for propositional theories.
In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 119–131. Springer-Verlag, 2005.


M. Fitting.
A Kripke-Kleene semantics for logic programs.
Journal of Logic Programming, 2(4):295–312, 1985.


M. Garcia de la Banda and E. Pontelli, editors.

Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08), volume 5366 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.


M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber.
Repair and prediction (under inconsistency) in large biological networks with answer set programming.
In F. Lin and U. Sattler, editors, *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 497–507. AAAI Press, 2010.


M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.


M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
Engineering an incremental ASP solver.
In Garcia de la Banda and Pontelli , pages 190–205.


M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
On the implementation of weight constraint rules in conflict-driven ASP solvers.
In Hill and Warren , pages 250–264.


M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Multi-criteria optimization in answer set programming.
In J. Gallagher and M. Gelfond, editors, *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, volume 11, pages 1–10. Leibniz International Proceedings in Informatics (LIPIcs), 2011.


M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Multi-criteria optimization in ASP and its application to Linux package configuration.
Unpublished draft, 2011.
Available at
`http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc11b.pdf`.


M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller.
A portfolio solver for answer set programming: Preliminary report.
In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 352–357. Springer-Verlag, 2011.


M. Gebser, R. Kaminski, and T. Schaub.
Complex optimization in answer set programming.
Theory and Practice of Logic Programming, 11(4-5):821–839, 2011.


M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
clasp: A conflict-driven answer set solver.
In Baral et al. , pages 260–265.


M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. , pages 136–148.


M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso , pages 386–392.


M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Advanced preprocessing for answer set solving.
In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.


M. Gebser, B. Kaufmann, and T. Schaub.
The conflict-driven answer set solver clasp: Progress report.
In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 509–514. Springer-Verlag, 2009.


M. Gebser, B. Kaufmann, and T. Schaub.
Solution enumeration for projected Boolean search problems.

In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.


M. Gebser, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren , pages 235–249.


M. Gebser, J. Pührer, T. Schaub, and H. Tompits.
A meta-programming technique for debugging answer-set programs.
In D. Fox and C. Gomes, editors, *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 448–453. AAAI Press, 2008.


M. Gebser and T. Schaub.
Tableau calculi for answer set programming.
In S. Etalle and M. Truszczyński, editors, *Proceedings of the Twenty-second International Conference on Logic Programming*

(ICLP'06), volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 2006.

M. Gebser and T. Schaub.
Generic tableaux for answer set programming.
In V. Dahl and I. Niemelä, editors, *Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2007.

M. Gelfond.
Answer sets.
In V. Lifschitz, F. van Harmelen, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier Science, 2008.

M. Gelfond and N. Leone.
Logic programming and knowledge representation — the A-Prolog perspective.
Artificial Intelligence, 138(1-2):3–38, 2002.


M. Gelfond and V. Lifschitz.
The stable model semantics for logic programming.
In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.


M. Gelfond and V. Lifschitz.
Logic programs with classical negation.
In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.


E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.
Journal of Automated Reasoning, 36(4):345–377, 2006.


P. Hill and D. Warren, editors.
Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.


J. Huang.
The effect of restarts on the efficiency of clause learning.
In Veloso , pages 2318–2323.


H. Kautz and B. Selman.
Planning as satisfiability.
In B. Neumann, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363. John Wiley & sons, 1992.


K. Konczak, T. Linke, and T. Schaub.
Graphs and colorings for answer set programming.
Theory and Practice of Logic Programming, 6(1-2):61–106, 2006.


R. Kowalski.
Logic for data description.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 77–103. Plenum Press, 1978.


J. Lee.

A model-theoretic counterpart of loop formulas.
In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.


N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello.
The DLV system for knowledge representation and reasoning.
ACM Transactions on Computational Logic, 7(3):499–562, 2006.


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


V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
ACM Transactions on Computational Logic, 7(2):261–268, 2006.


V. Lifschitz, L. Tang, and H. Turner.
Nested expressions in logic programs.

Annals of Mathematics and Artificial Intelligence, 25(3-4):369–389, 1999.


F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.
Artificial Intelligence, 157(1-2):115–137, 2004.


J. Lloyd.
Foundations of Logic Programming.
Symbolic Computation. Springer-Verlag, 1987.


V. Marek and M. Truszczyński.
Stable models and an alternative logic programming paradigm.
In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.


J. Marques-Silva, I. Lynce, and S. Malik.
Conflict-driven clause learning SAT solvers.
In Biere et al. , chapter 4, pages 131–153.


J. Marques-Silva and K. Sakallah.
GRASP: A search algorithm for propositional satisfiability.
IEEE Transactions on Computers, 48(5):506–521, 1999.


V. Mellarkod and M. Gelfond.
Integrating answer set reasoning with constraint solving techniques.
In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.


V. Mellarkod, M. Gelfond, and Y. Zhang.
Integrating answer set programming and constraint logic programming.
Annals of Mathematics and Artificial Intelligence, 53(1-4):251–287, 2008.


D. Mitchell.
A SAT solver primer.

Bulletin of the European Association for Theoretical Computer Science, 85:112–133, 2005.


M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.


I. Niemelä.
Logic programs with stable model semantics as a constraint programming paradigm.
Annals of Mathematics and Artificial Intelligence, 25(3-4):241–273, 1999.


R. Nieuwenhuis, A. Oliveras, and C. Tinelli.
Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T).
Journal of the ACM, 53(6):937–977, 2006.


J. Oetsch, J. Pührer, and H. Tompits.

Catching the ouroboros: On debugging non-ground answer-set programs.
In *Theory and Practice of Logic Programming. Twenty-sixth International Conference on Logic Programming (ICLP'10) Special Issue*, volume 10(4-6), pages 513–529. Cambridge University Press, 2010.


K. Pipatsrisawat and A. Darwiche.
A lightweight component caching scheme for satisfiability solvers.
In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.


L. Ryan.
Efficient algorithms for clause-learning SAT solvers.
Master's thesis, Simon Fraser University, 2004.


J. Schlipf.
The expressive powers of the logic programming semantics.

Journal of Computer and System Sciences, 51:64–86, 1995.


P. Simons, I. Niemelä, and T. Soininen.
Extending and implementing the stable model semantics.
Artificial Intelligence, 138(1-2):181–234, 2002.


T. Syrjänen.
Lparse 1.0 user's manual.


A. van Gelder, K. Ross, and J. Schlipf.
The well-founded semantics for general logic programs.
Journal of the ACM, 38(3):620–650, 1991.


M. Veloso, editor.
Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI Press/The MIT Press, 2007.


L. Zhang, C. Madigan, M. Moskewicz, and S. Malik.
Efficient conflict driven learning in a Boolean satisfiability solver.
In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.