Deep Reasoning in AI with Answer Set Programming

ASP Solving and Modeling beyond NP

Francesco Ricca and Mario Alviano

Department of Mathematics and Computer Science

University of Calabria

ESSAI 2024 - Athens

Outline

1. Beyond NP with ASP
2. ASP solving overview
    - Instantiation
    - Model Generation & Checking
    - Grounding-less ASP
    - Implementation of ASP(Q)
3. Programming Hints

Just a quick mention

Complexity notions

- What is a decision problem?
- The P class
- The NP class
- The co-NP class
- Beyond NP: the Polynomial Hierarchy
  - The Σ_k^P class
  - The Π_k^P class

Context (1)

Answer Set Programming (ASP) [BET11]

- Declarative programming paradigm
- Non-monotonic reasoning and logic programming
- Roots in Datalog and Nonmonotonic Logic
- Stable model semantics [GL91]
- Robust and efficient systems [GLM$^+$18]
  - DLV [AAC$^+$18], Clingo [GKK$^+$16], ...
- Effective in practical industrial-grade applications [EGL16]

Context (2)

Expressive KR Language

- Basic ASP models up to Σ_2^P [DEGV01]

  → i.e., problems not (polynomially) translatable to SAT or CSP

Well-known facts about ASP

- Uniform and compact encodings

  → Fixed encoding, instances as facts, inductive definitions
- Modular solutions

  → Generate-Define-Test/Guess&Check methodology [Lif02, EFLP00]
- Compact and elegant modeling of problem in NP

Context (2)

Expressive KR Language

- Basic ASP models up to Σ_2^P [DEGV01]

  → i.e., problems not (polynomially) translatable to SAT or CSP

Well-known facts about ASP

- Uniform and compact encodings

  → Fixed encoding, instances as facts, inductive definitions

- Modular solutions

  → Generate-Define-Test/Guess&Check methodology [Lif02, EFLP00]

- Compact and elegant modeling of problem in NP

Context (2)

Expressive KR Language

- Basic ASP models up to Σ_2^P [DEGV01]

  → i.e., problems not (polynomially) translatable to SAT or CSP

Well-known facts about ASP

- Uniform and compact encodings

  → Fixed encoding, instances as facts, inductive definitions

- Modular solutions

  → Generate-Define-Test/Guess&Check methodology [Lif02, EFLP00]

- Compact and elegant modeling of problem in NP

The usual example

Example (3-col)

Problem: Given a graph, assign one color out of 3 colors to each node such that two adjacent nodes have always different colors.

Input: a Graph is represented by *node*(_) and *edge*(_, _).

% guess a coloring for the nodes

$(r) \quad col(X, red) \mid col(X, yellow) \mid col(X, green) \ {:}{-} \ node(X).$

% discard colorings where adjacent nodes have the same color

$(c) \quad {:}{-} \ edge(X, Y), col(X, C), \ col(Y, C).$

% NB: answer sets are subset minimal → only one color per node

Motivation (1)

What about modeling beyond NP with ASP?

- It is possible...

Motivation (1)

What about modeling beyond NP with ASP?

- It is possible... with unrestricted disjunction [DEGV01]

  → Stable model checking in co-NP

Motivation (1)

What about modeling beyond NP with ASP?

- It is possible... with unrestricted disjunction [DEGV01]

  → Stable model checking in co-NP

- Rarely elegant and compact

  → Unless one can find a positive encoding

A rare example...

Example (Strategic Companies is Σ_2^P-complete)

Problem: There are various products, each one is produced by several companies. We now have to sell some companies. What are the minimal sets of strategic companies, such that all products can still be produced? A company also belong to the set, if all its controlling companies belong to it.

Input: *produced_by*(_, _, _) and *controlled_by*(_, _, _, _)

% Guess strategic companies

$strategic(Y) \mid strategic(Z) \text{ :- } produced\_by(X, Y, Z).$

% Ensure they are strategic

$strategic(W) \text{ :- } controlled\_by(W, X, Y, Z),$
$\quad strategic(X), strategic(Y), strategic(Z).$

Motivation (1)

What about modeling beyond NP with ASP?

- It is possible... to some extent
- Rarely elegant and compact

  → Unless one can find a positive encoding
  → Well-known strategic companies example

- Generate-define-test approach is no longer sufficient
- Saturation technique [EG95]
  - Exploits the minimality to check "for all" conditions
  - Difficult to use, not intuitive

    → Introduces constraints with no direct relation with the problem

Motivation (1)

What about modeling beyond NP with ASP?

- It is possible... to some extent
- Rarely elegant and compact

  → Unless one can find a positive encoding
  → Well-known strategic companies example
- Generate-define-test approach is no longer sufficient
- Saturation technique [EG95]
  - Exploits the minimality to check “for all” conditions
  - Difficult to use, not intuitive

    → Introduces constraints with no direct relation with the problem

Beyond NP (Saturation)

Example (Quantified Boolean Formulas by [EG95])

Problem: Given a QBF formula $\Phi = \exists X \forall Y \phi(X, Y)$, where ϕ is in 3-DNF form, determine an assignment for X that makes Φ satisfiable.

Input: $conj(X_1, S_{X_1}, X_2, S_{X_2}, X_3, S_{X_3})$ and $exist(X)$, $forall(Y)$

% Guess assignment for X
$asgn(X, true) \lor asgn(X, false) \leftarrow exist(X).$

% Guess assignment for Y
$asgn(Y, true) \lor asgn(Y, false) \leftarrow forall(Y).$

% Saturate Y
$asgn(Y, true) \leftarrow sat, forall(Y).$
$asgn(Y, false) \leftarrow sat, forall(Y).$

% check satisfiability Y
$sat \leftarrow conj(X_1, S_1, X_2, S_2, X_3, S_3), asgn(X_1, S_1), asgn(X_2, S_2), asgn(X_3, S_3).$
$\leftarrow \text{not } sat.$

Motivation and Goals

> *"Unlike the ease of common ASP modeling, [...] these techniques are rather involved and hardly usable by ASP laymen." [GKS11]*

Goals

- Address the shortcomings of ASP beyond NP
- Make modeling natural as for NP

Motivation and Goals

> *"Unlike the ease of common ASP modeling, [...] these techniques are rather involved and hardly usable by ASP laymen." [GKS11]*

Goals

- Address the shortcomings of ASP beyond NP
- Make modeling natural as for NP

ASP with Quantifiers: Syntax and Semantics [ART19]

Definition (ASP with Quantifiers)

An ASP with Quantifiers (ASP(Q)) program Π is of the form:

$$\Box_1 P_1 \; \Box_2 P_2 \; \cdots \; \Box_n P_n : C, \tag{1}$$

$\Box_i \in \{\exists^{st}, \forall^{st}\}$; P_i a program; C a stratified normal program.

Intuitive semantics

Program $\Pi = \exists^{st} P_1 \forall^{st} P_2 \cdots \exists^{st} P_{n-1} \forall^{st} P_n : C$ is coherent if:

"There is an answer set M_1 of P_1 s.t. for each answer set M_2 of $P_2 \cup fix(M_1)$ there is an answer set M_3 of $P_3 \cup fix(M_2)$ such that ... for each answer set M_n of $P_n \cup fix(M_{n-1})$ there is an answer set of $C \cup fix(M_n)$"

where $fix_P(I) = \{a \mid a \in I\} \cup \{\leftarrow a \mid a \in B_P \setminus I\}$. M_1 quantified answer set of Π

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- $P'_2 = P_2 \cup fix_{P_1}(\{a(1)\})$, and $fix_{P_1}(\{a(1)\}) = \{a(1); \leftarrow a(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\; b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\; not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- $P_2' = \{b(1) \vee b(2) \leftarrow a(1);\; b(2) \leftarrow a(2); a(1); \leftarrow a(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- $P_2' = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2); a(1); \leftarrow a(2)\}$
- P_2' has two answer sets $\{a(1), b(1)\}$ and $\{a(1), b(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- $P'_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2); a(1); \leftarrow a(2)\}$
- P'_2 has two answer sets $\{a(1), b(1)\}$ and $\{a(1), b(2)\}$
- **But** $C \cup fix_{P'_2}(\{a(1), b(1)\})$ is not coherent!

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- $P_2' = P_2 \cup fix_{P_1}(\{a(2)\})$, and $fix_{P_1}(\{a(2)\}) = \{a(2); \leftarrow a(1)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- P'_2 has one answer set $\{a(2), b(2)\}$

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- P_2' has one answer set $\{a(2), b(2)\}$
- Finally, $\{a(2), b(2)\}$ satisfies C!

Basic Example

Example (Quantified ASP Program)

Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$

- $P_1 = \{a(1) \vee a(2)\}$
- $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$
- $C = \{\leftarrow b(1),\ not\ b(2)\}$

Π is coherent, and $\{a(2)\}$ is a quantified answer set of Π

- P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$
- P_2' has one answer set $\{a(2), b(2)\}$
- Finally, $\{a(2), b(2)\}$ satisfies C!

Beyond NP (Saturation vs ASP(Q)) (1)

Example (Quantified Boolean Formulas)

Problem: Given a QBF formula $\Phi = \exists X \forall Y \phi(X, Y)$, where ϕ is in 3-DNF form, determine an assignment for X that makes Φ satisfiable.

Input: $conj(X_1, S_{X_1}, X_2, S_{X_2}, X_3, S_{X_3})$ and $exist(X)$, $forall(Y)$

% Guess assignment for X
$asgn(X, true) \lor asgn(X, false) \leftarrow exist(X).$

% Guess assignment for Y
$asgn(Y, true) \lor asgn(Y, false) \leftarrow forall(Y).$

% Saturate Y
$asgn(Y, true) \leftarrow sat, forall(Y).$
$asgn(Y, false) \leftarrow sat, forall(Y).$

% Check satisfiability Y
$sat \leftarrow conj(X_1, S_1, X_2, S_2, X_3, S_3), asgn(X_1, S_1), asgn(X_2, S_2), asgn(X_3, S_3).$
$\leftarrow \text{not } sat.$

Beyond NP (Saturation vs ASP(Q)) (2)

Example (Quantified Boolean Formulas)

Problem: Given a QBF formula $\Phi = \exists X \forall Y \phi(X, Y)$, where ϕ is in 3-DNF form, determine an assignment for X that makes Φ satisfiable.

Input: $conj(X_1, S_{X_1}, X_2, S_{X_2}, X_3, S_{X_3})$ and $exist(X)$, $forall(Y)$

Solution: $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$ such that:

% Guess assignment for X
$P_1 = \{\ asgn(X, true)\ \vee\ asgn(X, false) \leftarrow exist(X).\ \}$

% Guess assignment for Y
$P_2 = \{\ asgn(Y, true)\ \vee\ asgn(Y, false) \leftarrow forall(Y).\ \}$

% Check satisfiability Y
$C = \{$
$sat \leftarrow conj(X_1, S_1, X_2, S_2, X_3, S_3), asgn(X_1, S_1), asgn(X_2, S_2), asgn(X_3, S_3).$
$\leftarrow \text{not } sat.$
$\}$

Beyond NP (Π_2^P-complete)

Example (Quantified Boolean Formulas)

Problem: Given a QBF formula $\Psi = \forall X \exists Y \psi(X, Y)$, where ψ is in 3-CNF form, determine an assignment for X that makes Ψ satisfiable.

Input: $disj(X_1, S_{X_1}, X_2, S_{X_2}, X_3, S_{X_3})$ and $exist(X)$, $forall(Y)$

Solution: $\Pi = \forall^{st} P_1 \exists^{st} P_2 : C$ such that:

% Guess assignment for X
$P_1 = \{\ asgn(X, true)\ \vee\ asgn(X, false) \leftarrow forall(X).\ \}$

% Guess assignment for Y
$P_2 = \{\ asgn(Y, true)\ \vee\ asgn(Y, false) \leftarrow exist(Y).\ \}$

% Check satisfiability Y
$C = \{$
$\leftarrow disj(X_1, S_1, X_2, S_2, X_3, S_3), iasgn(X_1, S_1), iasgn(X_2, S_2), iasgn(X_3, S_3).$
$iasgn(X, false)$:- $asgn(X, true).$
$iasgn(X, true)$:- $asgn(X, false).$
$\}$

Theoretical Results

Theorem (ASP(Q) is a straightforward generalization of ASP)

Let P be an ASP program, and Π *the ASP(Q) program* $\exists^{st} P : C$*, with* $C = \emptyset$*. Then,*

$$AS(P) = QAS(\Pi).$$

COHERENCE problem: Given Π, decide whether Π is coherent.

Theorem (Complexity)

The COHERENCE *problem is*

(*i*) *PSPACE-complete, even restricted to normal ASP(Q) programs;*

(*ii*) Σ_n^P*-complete for n-normal existential ASP(Q) programs;*

(*iii*) Π_n^P*-complete for n-normal universal ASP(Q) programs.*

Modeling Examples from [ART19]

Min-Max Clique [Ko95]

- Example of Π_2^P-complete problem
- Key role in game theory, optimization and complexity [CDG$^+$95]
- Approach can be adapted to model other minmax problems

Pebbling Number [MC06]

- Mathematical game
- Example of Π_2^P-complete problem

Vapnik-Chervonenkis Dimension (VC-Dimension) [BEHW89]

- Relevant problem in machine learning
- Measures the capacity of a space of functions that can be learned by a statistical classification algorithm
- Example of Σ_3^P-complete problem

Modeling Examples from [ART19]

Min-Max Clique [Ko95]

- Example of Π_2^P-complete problem
- Key role in game theory, optimization and complexity [CDG$^+$95]
- Approach can be adapted to model other minmax problems

Pebbling Number [MC06]

- Mathematical game
- Example of Π_2^P-complete problem

Vapnik-Chervonenkis Dimension (VC-Dimension) [BEHW89]

- Relevant problem in machine learning
- Measures the capacity of a space of functions that can be learned by a statistical classification algorithm
- Example of Σ_3^P-complete problem

Minmax Clique: The Problem

Definition (Minmax Clique)

Given a graph G, sets of indices I and J, a partition $(A_{i,j})_{i \in I, j \in J}$, and an integer k, decide whether

$$\min_{f \in J^I} \max\{|Q| : Q \text{ is a clique of } G_f\} \geq k.$$

J^I is the set of all total functions from I to J, and G_f is the subgraph of G induced by $\bigcup_{i \in I} A_{i,f(i)}$.

In simpler words:

"For each total function $f \in J^I$, there exists a clique c in G_f, such that the size of c is larger than k"

Solution: An ASP(Q) program $\Pi = \forall^{st} P_1 \exists^{st} P_2 : C$.

Minmax Clique: The Solution

"For each total function $f \in J^I$"

$$P_1 = \left\{ \begin{array}{rcll} edge(a,b) & & & \forall (a,b) \in E \\ node(a) & & & \forall a \in N \\ v(i,j,a) & & & \forall i \in I,\ j \in J,\ a \in A_{i,j} \\ setI(X) & \leftarrow & v(X,\_,\_) & \\ setJ(X) & \leftarrow & v(\_,X,\_) & \\ 1\{f(X,Y) : setJ(Y)\}1 & \leftarrow & setI(X) & \end{array} \right\}$$

"There exists a clique c in G_f"

$$P_2 = \left\{ \begin{array}{rcl} inInduced(Z) & \leftarrow & v(X,Y,Z),\ f(X,Y) \\ edgeP(X,Y) & \leftarrow & edge(X,Y),\ inInduced(X), \\ & & inInduced(Y) \\ \{inClique(X)\ :\ inInduced(X)\} & & \\ & \leftarrow & inClique(X),\ inClique(Y), \\ & & not\ edgeP(X,Y), X < Y. \end{array} \right\}$$

"Such that the size of c is larger than k"

$$C = \{\ \leftarrow \quad \#\text{count}\{X : inClique(X)\} < k\ \ \}$$

Pebbling Number: The Problem

Definition (Pebbling Number)

Given a digraph $G = \langle N, E \rangle$ with pebbles placed on (some of) its nodes.

- A pebbling move along (a, b) removes 2 pebbles from a and adds 1 to b
- The Pebbling number $\pi(G)$ is the smallest number of pebbles s.t. for each assignment of k pebbles and for each node w (the target), some sequence of pebbling moves results in a pebble on w

Problem: Is $\pi(G) \leq k$?

In simpler words:

"For each assignment of k pebbles to the nodes of G, and for each target node $t \in N$, there exists a sequence of pebble moves (at most $k - 1$ moves), such that some pebble is on w"

Solution: An ASP(Q) program $\Pi = \forall^{st} P_1 \exists^{st} P_2 : C$.

Pebbling Number: The Solution (1)

"For each assignment of k pebbles to the nodes of G, and for each target node $w \in N$"

$$P_1 =$$

$$\left\{ \begin{array}{rcl} edge(a, b) & & \forall (a, b) \in E \\ node(a) & & \forall a \in N \\ pebble(i) & & \forall i = 0, 1, \ldots, k \\ 1\{onNode(X, N) : pebble(N)\}1 & \leftarrow & node(X) \\ \leftarrow \#\mathrm{sum}\{N, X : onNode(X, N)\} \neq k & & \\ 1\{target(X) : node(X)\}1 & & \end{array} \right\}$$

Pebbling Number: The Solution (2)

“There exists a sequence of pebble moves”

$$P_2 =$$

$$\left\{ \begin{array}{rcl} step(i) & & \forall i = 0, 1, \ldots, k-1 \\ 1\{endstep(S) : step(S)\}1 & & \\ onNode(X, N, 0) & \leftarrow & onNode(X, N) \\ 1\{move(X, Y, S) : edge(X, Y)\}1 & \leftarrow & step(S), endstep(T), 1 \leq S,\ S \leq T \\ & \leftarrow & move(X, Y, S),\ onNode(X, N, S), N < 2 \\ affected(X, S) & \leftarrow & move(X, Y, S) \\ affected(Y, S) & \leftarrow & move(X, Y, S) \\ onNode(X, N-2, S) & \leftarrow & onNode(X, N, S-1), move(X, Y, S) \\ onNode(Y, M+1, S) & \leftarrow & onNode(Y, M, S-1), move(X, Y, S) \\ onNode(X, N, S) & \leftarrow & onNode(X, N, S-1),\ not\ affected(X, S) \end{array} \right\}$$

“Such that some pebble is on *w*”

$$C = \{ \leftarrow \quad target(W), onNode(W, 0, T), endstep(T) \ \}$$

Vapnik-Chervonenkis Dimension: The Problem

Definition (VC Dimension)

Let k be an integer, U a finite set, $\mathcal{C} = \{S_1, \ldots, S_n\} \subseteq 2^U$ a collection of subsets of U represented by a program $P_{\mathcal{C}}$.

Problem: Is there $X \subseteq U$ of size at least k, s.t. for each $S \subseteq X$, there is S_i s.t. $S = S_i \cap X$?
(VC dimension of $\mathcal{C}$, $VC(\mathcal{C})$ is the maximum size of such a set X.)

Solution: An ASP(Q) program $\Pi = \exists^{st} P_1 \forall^{st} P_2 1 \exists^{st} P_3 : C$.

Vapnik-Chervonenkis Dimension: The Solution

"There is $X \subseteq U$ of size at least k"

$$P_1 = \left\{ \begin{array}{rl} inU(x) & \forall x \in U \\ k\{inX(X) : inU(X)\} & \end{array} \right\}$$

"Such that for each $S \subseteq X$"

$$P_2 = \left\{ \ \{inS(X) : inX(X)\} \ \right\}$$

"There is S_i"

$$P_3 = P_{\mathcal{C}}$$

"Such that $S = S_i \cap X$"

$$C = \left\{ \begin{array}{rcl} inIntersection(Z) & \leftarrow & true(Z),\ inX(Z) \\ & \leftarrow & inIntersection(Z),\ not\ inS(Z) \\ & \leftarrow & not\ inIntersection(Z),\ inS(Z) \end{array} \right\}$$

ASP(Q) vs ASP vs QBF

ASP vs ASP(Q)

- ASP(Q) is a natural extension of ASP
- Natural in Σ_2^P with disjunctive positive encodings
- Normal program sufficient to model PH

QBF vs ASP(Q)

- Both extend base language with some form of quantifier

  → variable assignments vs answer sets
- Same computational properties
- ASP(Q) supports variables and inductive definitions
- ASP(Q) inherits aggregates, choice rules, strong negation, and disjunction

ASP Solving

ASP Implementation
(overview)

Answer Set Programming (ASP) (1)

Idea:

1. Represent a computational problem by a Logic program
2. Answer sets correspond to problem solutions
3. Use an ASP implementation to find these solutions

Answer Set Programming (ASP) (1)

Idea:

1. Represent a computational problem by a Logic program
2. Answer sets correspond to problem solutions
3. Use an ASP implementation to find these solutions

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
    1. Write a program representing a computational problem
       → i.e., such that answer sets correspond to solutions
    2. Use a solver to find solutions

- **Why is the knowledge of ASP Solving important?**
    - Knowledge of programming methodology
      → you can write programs
    - Knowledge of the evaluation process
      →you can write programs more efficiently
    - Knowledge of an ASP System
      → you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     → i.e., such that answer sets correspond to solutions
  2. Use a solver to find solutions

- **Why is the knowledge of ASP Solving important?**
  - Knowledge of programming methodology
    → you can write programs
  - Knowledge of the evaluation process
    →you can write programs more efficiently
  - Knowledge of an ASP System
    → you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     → i.e., such that answer sets correspond to solutions
  2. Use a solver to find solutions

- **Why is the knowledge of ASP Solving important?**
  - Knowledge of programming methodology
    → you can write programs
  - Knowledge of the evaluation process
    →you can write programs more efficiently
  - Knowledge of an ASP System
    → you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     → i.e., such that answer sets correspond to solutions
  2. Use a solver to find solutions

- **Why is the knowledge of ASP Solving important?**
  - Knowledge of programming methodology
    → you can write programs
  - Knowledge of the evaluation process
    →you can write programs more efficiently
  - Knowledge of an ASP System
    → you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     → i.e., such that answer sets correspond to solutions
  2. Use a solver to find solutions

- **Why is the knowledge of ASP Solving important?**
  - Knowledge of programming methodology
    → you can write programs
  - Knowledge of the evaluation process
    →you can write programs more efficiently
  - Knowledge of an ASP System
    → you can actually implement applications

Evaluation of ASP Programs (1)

Traditionally a two-step process:

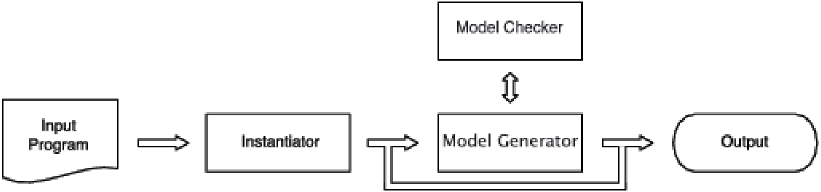
1. **Instantiation (or grounding)**

   → Variable elimination

2. **Propositional search (or ASP Solving)**

   → Model Generation: "generate models"

   → (Stable) Model Checking: "verify that models are answer sets"



Outline

About the Instantiation

Some facts:

- Exponential in the worst case
- Input of a subsequent exponential procedure
- Significantly affects the performance of the overall process

Full instantiation: *i.e., apply every possible substitution*

→ Not viable in practice

Intelligent instantiation

→ Keep the size of the instantiation as small as possible

→ Equivalent to the full one

→ Intelligent Instantiators can solve problems in P

→ Deductive Databases as a subcase!

About the Instantiation

Some facts:

- Exponential in the worst case
- Input of a subsequent exponential procedure
- Significantly affects the performance of the overall process

Full instantiation: *i.e., apply every possible substitution*

→ Not viable in practice

Intelligent instantiation

→ Keep the size of the instantiation as small as possible

→ Equivalent to the full one

→ Intelligent Instantiators can solve problems in P

→ Deductive Databases as a subcase!

About the Instantiation

Some facts:

- Exponential in the worst case
- Input of a subsequent exponential procedure
- Significantly affects the performance of the overall process

Full instantiation: *i.e., apply every possible substitution*

→ Not viable in practice

Intelligent instantiation

→ Keep the size of the instantiation as small as possible

→ Equivalent to the full one

→ Intelligent Instantiators can solve problems in P

→ Deductive Databases as a subcase!

Instantiation Example: 3-Colorability

% guess a coloring for the nodes
$(r) \quad col(X, red) \mid col(X, yellow) \mid col(X, green) \ {:}\!-\ node(X).$

% discard colorings where adjacent nodes have the same color
$(c) \quad {:}\!-\ edge(X, Y), col(X, C),\ col(Y, C).$

Instance: $node(1).\ node(2).\ node(3).\ edge(1, 2).\ edge(2, 3).$

Instantiation Example: 3-Colorability

% guess a coloring for the nodes

$(r) \quad col(X, red) \mid col(X, yellow) \mid col(X, green) \text{ :- } node(X).$

% discard colorings where adjacent nodes have the same color

$(c) \quad \text{:- } edge(X, Y), col(X, C), \; col(Y, C).$

Instance: $node(1).\ node(2).\ node(3).\ edge(1, 2).\ edge(2, 3).$

Full Theoretical Instantiation:

$col(red, red) \mid col(red, yellow) \mid col(red, green) \text{ :- } node(red).$
$col(yellow, red) \mid col(yellow, yellow) \mid col(yellow, green) \text{ :- } node(yellow).$
$col(green, red) \mid col(green, yellow) \mid col(green, green) \text{ :- } node(green).$
...
$col(1, red) \mid col(1, yellow) \mid col(1, green) \text{ :- } node(1).$
...
$\text{:- } edge(1, 2), col(1, 1), \; col(2, 1).$
...
$\text{:- } edge(1, 2), col(1, red), \; col(2, red).$
...

Instantiation Example: 3-Colorability

% guess a coloring for the nodes

$(r)\quad col(X, red) \mid col(X, yellow) \mid col(X, green) \ {:\!-}\ node(X).$

% discard colorings where adjacent nodes have the same color

$(c)\quad {:\!-}\ edge(X, Y), col(X, C),\ col(Y, C).$

Instance: $node(1).\ node(2).\ node(3).\ edge(1, 2).\ edge(2, 3).$

Full Theoretical Instantiation: → is huge (2916 rules) and redundant!

$col(red, red) \mid col(red, yellow) \mid col(red, green) \ {:\!-}\ node(red).$
$col(yellow, red) \mid col(yellow, yellow) \mid col(yellow, green) \ {:\!-}\ node(yellow).$
$col(green, red) \mid col(green, yellow) \mid col(green, green) \ {:\!-}\ node(green).$
...
$col(1, red) \mid col(1, yellow) \mid col(1, green) \ {:\!-}\ node(1).$ ← OK!
...
${:\!-}\ edge(1, 2), col(1, 1),\ col(2, 1).$ ← redundant!
...
${:\!-}\ edge(1, 2), col(1, red),\ col(2, red).$ ← OK!
...

Instantiation Example: 3-Colorability

% guess a coloring for the nodes

$(r) \quad col(X, red) \mid col(X, yellow) \mid col(X, green) \text{ :- } node(X).$

% discard colorings where adjacent nodes have the same color

$(c) \quad \text{ :- } edge(X, Y), col(X, C), \; col(Y, C).$

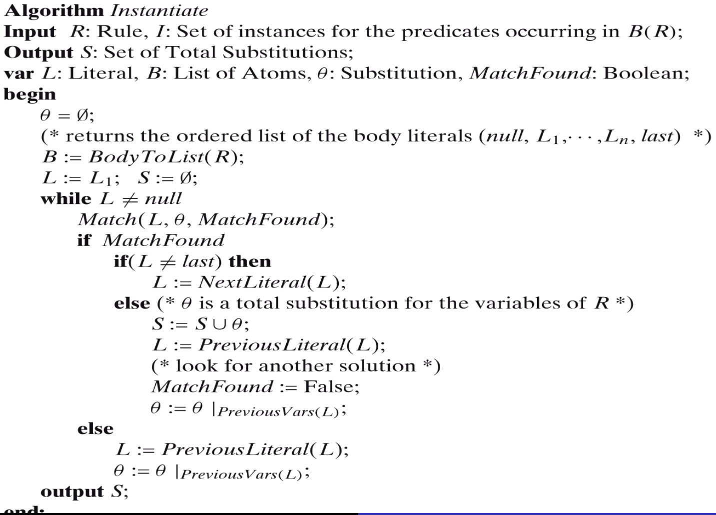
Instance: $node(1).\ node(2).\ node(3).\ edge(1, 2).\ edge(2, 3).$

Intelligent Instantiation: → equivalent but much smaller (9 rules)!

$$
\begin{array}{l}
col(1, red) \mid col(1, yellow) \mid col(1, green).\\
col(2, red) \mid col(2, yellow) \mid col(2, green).\\
col(3, red) \mid col(3, yellow) \mid col(3, green).
\end{array}
$$

$$
\begin{array}{l}
\text{:- } col(1, red), \; col(2, red).\\
\text{:- } col(1, green), \; col(2, green).\\
\text{:- } col(1, yellow), \; col(2, yellow).\\
\text{:- } col(2, red), \; col(3, red).\\
\text{:- } col(2, green), \; col(3, green).\\
\text{:- } col(2, yellow), \; col(3, yellow).
\end{array}
$$

Instantiation of a Rule: like a join in a DB



```
**Algorithm** *Instantiate*
**Input** $R$: Rule, $I$: Set of instances for the predicates occurring in $B(R)$;
**Output** $S$: Set of Total Substitutions;
**var** $L$: Literal, $B$: List of Atoms, $\theta$: Substitution, $MatchFound$: Boolean;
**begin**
    $\theta = \emptyset$;
    (* returns the ordered list of the body literals ($null$, $L_1, \cdots, L_n$, $last$) *)
    $B := BodyToList(R)$;
    $L := L_1$;   $S := \emptyset$;
    **while** $L \neq null$
        $Match(L, \theta, MatchFound)$;
        **if** $MatchFound$
            **if**($L \neq last$) **then**
                $L := NextLiteral(L)$;
            **else** (* $\theta$ is a total substitution for the variables of $R$ *)
                $S := S \cup \theta$;
                $L := PreviousLiteral(L)$;
                (* look for another solution *)
                $MatchFound :=$ False;
                $\theta := \theta \mid_{PreviousVars(L)}$;
        **else**
            $L := PreviousLiteral(L)$;
            $\theta := \theta \mid_{PreviousVars(L)}$;
    **output** $S$;
**end**;
```

Intelligent Instantiator

The instantiator (or grounder)

- outputs a ground program equivalent to the input
- ...often much smaller than full theoretical instantiation
- Performs “deterministic” inferences
- Computes the unique answer set if the input is stratified and non disjunctive

Intelligent Instantiator

The instantiator (or grounder)

- outputs a ground program equivalent to the input
- ...often much smaller than full theoretical instantiation
- Performs “deterministic” inferences
- Computes the unique answer set if the input is stratified and non disjunctive

Outline

Model Generation & Checking

Model Generation → *produces candidate models*

- Similar to a SAT solver
- Davis-Putnam-Logeman-Loveland (DPLL) - (1st gen.)
  - Propagate Deterministic Consequences
    → Unit Propagation
    → Support Propagation
    → Well-founded Negation
  - Assume a literal *l* (heuristically) until a model is generated
  - Upon inconsistency Backtrack (assume not *l*)

Model Checker → *checks if candidates are Answer Sets*

- Polynomial time computable check
- Translation to UNSAT for hard (non-HCF) instances

Model Generation & Checking

Model Generation → *produces candidate models*

- Similar to a SAT solver
- Davis-Putnam-Logeman-Loveland (DPLL) - (1st gen.)
  - Propagate Deterministic Consequences
    → Unit Propagation
    → Support Propagation
    → Well-founded Negation
  - Assume a literal *l* (heuristically) until a model is generated
  - Upon inconsistency Backtrack (assume not *l*)

Model Checker → *checks if candidates are Answer Sets*

- Polynomial time computable check
- Translation to UNSAT for hard (non-HCF) instances

Model Generator (DPLL)

```
Input   : An interpretation $I$ for a program $\Pi$
Output: True if $\Pi$ admits answer set, false otherwise
begin
    if ! Propagate($I$) then
        return false ;
    end
    if $I$ is total then
        return CheckModel($I$)
    end
    $\ell$ := ChooseUndefinedLiteral();
    if ComputeAnswerSet($I \cup \{\ell\}$) then
        return true;
    end
    if ComputeAnswerSet($I \cup \{\text{not } \ell\}$) then
        return true;
    end
    else
        return false;
    end
end
```

Unit Propagation

- Infer a literal if it is the only one which can satisfy a rule
- Forward Inference + Contraposition
- Same as unit propagation in SAT

Example (Unit propagation)

$a \mid b \coloneq c.$

If *b* is false and *c* is true infer *a* to be true.

Support Propagation

- Based on the supportedness property
- “Each atom in an answer set has to be supported”

Example (Support propagation)

$a \mid b :- c.$
$a \mid d :- \mathrm{not}\ b.$

If *b* and *c* are false and *d* is true infer *a* false.

Aggregates Propagation

- Similar to propagation in pseudo-boolean solvers
- Basically needed only for #count and #sum
- “Apply the semantics of the aggregate”

Example (Aggregate propagation)

```
:– #sum{                              (from :– #sum{X,Y : a(X,Y)}>=2)
          1,2 :a(1,2);
          1,3 :a(1,3);
          1,4 :a(1,4)
     } >= 2
```

If $a(1,2)$ is true then $a(1,3)$ and $a(1,4)$ must be false

Well-founded Propagation

- Self-supporting truth is not admitted in answer sets
- Unfounded sets are sets of atoms violating this property

Definition (Unfounded set)

A set U is an unfounded set for program Π w.r.t. I if, for each $a \in$ U, for each rule $r \in \Pi$ such that $a \in H(r)$ at least one of these holds:

$$\text{(i) } B(r) \cap \neg I \neq \emptyset \quad \text{(ii) } B^+(r) \cap U \neq \emptyset \quad \text{(iii) } H(r) \setminus U \cap I \neq \emptyset$$

- Detected unfounded sets are propagated as false

Model Generation Example: 3-Colorability

Model Generation step:

$col(1, red) \mid col(1, yellow) \mid col(1, green).$
$col(2, red) \mid col(2, yellow) \mid col(2, green).$
$col(3, red) \mid col(3, yellow) \mid col(3, green).$

$:\!-\ col(1, red),\ col(2, red).$
$:\!-\ col(1, green),\ col(2, green).$
$:\!-\ col(1, yellow),\ col(2, yellow).$
$:\!-\ col(2, red),\ col(3, red).$
$:\!-\ col(2, green),\ col(3, green).$
$:\!-\ col(2, yellow),\ col(3, yellow).$

True: {}
False: {}

Model Generation Example: 3-Colorability

Model Generation step: **Chose literal**

$$col(1, red) \mid col(1, yellow) \mid col(1, green).$$
$$col(2, red) \mid col(2, yellow) \mid col(2, green).$$
$$col(3, red) \mid col(3, yellow) \mid col(3, green).$$

$$:\!-\ col(1, red),\ col(2, red).$$
$$:\!-\ col(1, green),\ col(2, green).$$
$$:\!-\ col(1, yellow),\ col(2, yellow).$$
$$:\!-\ col(2, red),\ col(3, red).$$
$$:\!-\ col(2, green),\ col(3, green).$$
$$:\!-\ col(2, yellow),\ col(3, yellow).$$

True: $\{\} \leftarrow col(1, red)$
False: $\{\}$

Model Generation Example: 3-Colorability

Model Generation step: **Propagate Deterministic Consequences**

$col(1, red) \mid col(1, yellow) \mid col(1, green).$ ← 1-support propagation
$col(2, red) \mid col(2, yellow) \mid col(2, green).$
$col(3, red) \mid col(3, yellow) \mid col(3, green).$

$:-\ col(1, red),\ col(2, red).$ ← 2-unit propagation
$:-\ col(1, green),\ col(2, green).$
$:-\ col(1, yellow),\ col(2, yellow).$
$:-\ col(2, red),\ col(3, red).$
$:-\ col(2, green),\ col(3, green).$
$:-\ col(2, yellow),\ col(3, yellow).$

True: $\{col(1, red)\}$
False: $\{\ col(1, yellow),\ col(1, green),\ col(2, red)\ \}$

Model Generation Example: 3-Colorability

Model Generation step: **Chose literal**

$$col(1, red) \mid col(1, yellow) \mid col(1, green).$$
$$col(2, red) \mid col(2, yellow) \mid col(2, green).$$
$$col(3, red) \mid col(3, yellow) \mid col(3, green).$$

$$:-\ col(1, red),\ col(2, red).$$
$$:-\ col(1, green),\ col(2, green).$$
$$:-\ col(1, yellow),\ col(2, yellow).$$
$$:-\ col(2, red),\ col(3, red).$$
$$:-\ col(2, green),\ col(3, green).$$
$$:-\ col(2, yellow),\ col(3, yellow).$$

True: $\{col(1, red) \leftarrow col(2, yellow)\}$
False: $\{col(1, yellow),\ col(1, green),\ col(2, red)\}$

Model Generation Example: 3-Colorability

Model Generation step: **Propagate Deterministic Consequences**

$col(1, red) \mid col(1, yellow) \mid col(1, green).$
$col(2, red) \mid col(2, yellow) \mid col(2, green).$ ←1-support propagation
$col(3, red) \mid col(3, yellow) \mid col(3, green).$

$:- \ col(1, red),\ col(2, red).$
$:- \ col(1, green),\ col(2, green).$
$:- \ col(1, yellow),\ col(2, yellow).$
$:- \ col(2, red),\ col(3, red).$
$:- \ col(2, green),\ col(3, green).$
$:- \ col(2, yellow),\ col(3, yellow).$ ← 2-unit propagation

True: $\{col(1, red), col(2, yellow)\}$
False: $\{col(1, yellow), col(1, green), col(2, red), col(2, green), col(3, yellow)\}$

Model Generation Example: 3-Colorability

Model Generation step: Chose literal

$$col(1, red) \mid col(1, yellow) \mid col(1, green).$$
$$col(2, red) \mid col(2, yellow) \mid col(2, green).$$
$$col(3, red) \mid col(3, yellow) \mid col(3, green).$$

$$:\!-\ col(1, red),\ col(2, red).$$
$$:\!-\ col(1, green),\ col(2, green).$$
$$:\!-\ col(1, yellow),\ col(2, yellow).$$
$$:\!-\ col(2, red),\ col(3, red).$$
$$:\!-\ col(2, green),\ col(3, green).$$
$$:\!-\ col(2, yellow),\ col(3, yellow).$$

True: $\{col(1, red), col(2, yellow)\} \leftarrow col(3, red)$
False: $\{\ col(1, yellow),\ col(1, green),\ col(2, red),\ col(2, green),$ $col(3, yellow)\ \}$

Model Generation Example: 3-Colorability

Model Generation step: **Propagate Deterministic Consequences**

$col(1, red) \mid col(1, yellow) \mid col(1, green).$
$col(2, red) \mid col(2, yellow) \mid col(2, green).$
$col(3, red) \mid col(3, yellow) \mid col(3, green).$ ←support propagation

$:\!-\ col(1, red),\ col(2, red).$
$:\!-\ col(1, green),\ col(2, green).$
$:\!-\ col(1, yellow),\ col(2, yellow).$
$:\!-\ col(2, red),\ col(3, red).$
$:\!-\ col(2, green),\ col(3, green).$
$:\!-\ col(2, yellow),\ col(3, yellow).$

True: $\{col(1, red), col(2, yellow), col(3, red)\}$
False: $\{\ col(1, yellow),\ col(1, green),\ col(2, red),\ col(2, green),$
$col(3, yellow)\ col(3, green)\ \}$

Model Generation Example: 3-Colorability

Model Generation step: Answer set found!

$$col(1, red) \mid col(1, yellow) \mid col(1, green).$$
$$col(2, red) \mid col(2, yellow) \mid col(2, green).$$
$$col(3, red) \mid col(3, yellow) \mid col(3, green).$$

$$:\!-\ col(1, red),\ col(2, red).$$
$$:\!-\ col(1, green),\ col(2, green).$$
$$:\!-\ col(1, yellow),\ col(2, yellow).$$
$$:\!-\ col(2, red),\ col(3, red).$$
$$:\!-\ col(2, green),\ col(3, green).$$
$$:\!-\ col(2, yellow),\ col(3, yellow).$$

Answer Set: $\{col(1, red), col(2, yellow), col(3, red)\}$

Model Checking

Model Checker → *checks if candidates are Answer Sets*

- Polynomial time computable check
- Translation to UNSAT for hard (non-HCF) instances

Implementation

- Generate SAT formula
- Call SAT solver
- ...do it only if necessary!

Model Checking: build SAT Formula

```
Input: A ground DLP program $\mathcal{P}$ and a model $M$ for $\mathcal{P}$.
Output: A propositional CNF formula $\Gamma_M(\mathcal{P})$ over $M$.
var $\mathcal{P}'$: DLP Program; $S$: Set of Clauses;
begin
1.   Delete from $\mathcal{P}$ each rule whose body is false w.r.t. $M$;
2.   Remove all negative literals from the (bodies of the) remaining rules;
3.   Remove all false atoms (w.r.t. $M$) from the heads of the resulting rules;
4.   $S := \emptyset$;
5.   Let $\mathcal{P}'$ be the program resulting from steps 1–3;
6.   for each rule $a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_m$ in $\mathcal{P}'$ do
7.           $S := S \cup \{ b_1 \vee \cdots \vee b_m \leftarrow a_1 \wedge \cdots \wedge a_n \}$;
8.   end for;
9.   $\Gamma_M(\mathcal{P}) := \bigwedge_{c \in S} c \;\wedge\; (\bigvee_{x \in M} x)$;
10.  output $\Gamma_M(\mathcal{P})$
end.
```

Example: build SAT Formula

Consider: $M = \{a, b\}$

$a \mid b \mid c.$
$a \,{:}\!-\, b.$
$b \,{:}\!-\, a, \text{not } c..$
$a \,{:}\!-\, c.$

Example: build SAT Formula

Step:(1)

Consider: $M = \{a, b\}$

$a \mid b \mid c.$

$a :- b.$

$b :- a, \text{not } c..$

~~$a :- c.$~~

Example: build SAT Formula

Step:(2)
Consider: $M = \{a, b\}$

$a \mid b \mid c.$
$a \text{ :- } b.$
$b \text{ :- } a$~~, not c~~.

Example: build SAT Formula

Step:(3)
Consider: $M = \{a, b\}$

$$a \mid b \cancel{\mid c}.$$
$$a \mathrel{:-} b.$$
$$b \mathrel{:-} a.$$

Example: build SAT Formula

Consider: $M = \{a, b\}$

$a \mid b.$
$a \,{:}\!-\, b.$
$b \,{:}\!-\, a.$

Example: build SAT Formula

Step: (6-9)
Consider: $M = \{a, b\}$

$a \mid b. \implies \leftarrow a \wedge b$
$a :\!- b.$
$b :\!- a.$

Example: build SAT Formula

Step: (6-9)

Consider: $M = \{a, b\}$

$a \mid b. \Longrightarrow \leftarrow a \wedge b$

$a \mathrel{:-} b. \Longrightarrow \; b \leftarrow a$

$b \mathrel{:-} a.$

Example: build SAT Formula

Step: (6-9)

Consider: $M = \{a, b\}$

$$a \mid b. \implies \leftarrow a \wedge b$$
$$a \coloneq b. \implies b \leftarrow a$$
$$b \coloneq a. \implies a \leftarrow b$$

Example: build SAT Formula

Step: (6-9)

Consider: $M = \{a, b\} \implies a \lor b \leftarrow$

$a \mid b. \implies \leftarrow a \land b$

$a \text{ :- } b. \implies b \leftarrow a$

$b \text{ :- } a. \implies a \leftarrow b$

Example: build SAT Formula

Step: (6-9)

Consider: $M = \{a, b\} \implies a \vee b$

$$a \mid b. \implies \leftarrow a \wedge b \implies \neg a \vee \neg b$$
$$a \,{:}\!-\, b. \implies b \leftarrow a \implies \neg a \vee b$$
$$b \,{:}\!-\, a. \implies a \leftarrow b \implies \neg b \vee a$$

Example: build SAT Formula

Consider: $M = \{a, b\} \Longrightarrow \; a \vee b$

$$\Longrightarrow \; \neg a \vee \neg b$$

$$\Longrightarrow \; \neg a \vee b$$

$$\Longrightarrow \; \neg b \vee a$$

Unsatisfiable → Answer Set!

Example: build SAT Formula

Consider: $M = \{a, c\}$

$a \mid b \mid c$
$a :\!-\, b.$
$b :\!-\, a, \mathrm{not}\ c.$
$a :\!-\, c.$

Example: build SAT Formula

Consider: $M = \{a, c\} \implies a \lor c \leftarrow$

$a \mid b \mid c \implies a \lor c$

$a \text{:-} b. \implies$

$b \text{:-} a, \text{not } c. \implies$

$a \text{:-} c. \implies c \leftarrow a.$

Example: build SAT Formula

Consider: $M = \{a, c\} \implies a \vee c \leftarrow$

$a \vee c$

$c \leftarrow a.$

Satisfied by $\{c\} \rightarrow$ not an answer set!

Modern ASP Solvers - (2nd gen)

Pre-processing

- Program Rewriting (mostly completion)
- Program Simplification

Model Generator

- CDCL-based [MLM21] ASP solver

  → Unit+Well-founded Negation+Aggregate Propagation

  → New Heuristics & Learning

Model Checker

- Build the “formula” once (solving under assumption)

  → Reduct-based check (Wasp)

  → Encoding of (Hard) Unfounded-free check (Clasp)

Rewriting and Simplification (1)

Program Completion [Cla77, EL03]

- Simulate support propagation
- Sufficient for *tight* programs (i.e., no positive recursion)

Example (completion)

$a :- b, c$ $\qquad$ $a \leftrightarrow ((b \wedge c) \vee (e \wedge f))$
$a :- e, f$

% transformed to “simulate” support

$a :- a_{bc}$ $\qquad$ $a_{bc} :- b, c$ $\qquad$ $a_{ef} :- e, f$
$a :- a_{ef}$ $\qquad$ $:- \ a_{bc}, \text{not } b$ $\qquad$ $:- \ a_{ef}, \text{not } e$
$:- a, \text{not } a_{bc}, \text{not } a_{ef}$ $\qquad$ $:- \ a_{bc}, \text{not } c$ $\qquad$ $:- \ a_{ef}, \text{not } f$

Rewriting and Simplification (2)

Simplifications (intuition, from `SatELite` [EB05, HJL^{+}15])

- Remove useless statements (subsumption-check)
- Reduce search space (variable-elimination)
- ... and more

Example (some simplifications)

$b \mid c.$ $\quad f \mid g.$		$b \mid c.$ $\quad f \mid g.$	
$a.$		$a.$	
$a \text{ :- not } b, c$	$\rightarrow$	~~$a \text{ :- not } b, c$~~	% subsumption
$\text{:- not } e, f$	$\rightarrow$	~~$\text{:- not } e, f$~~	% pure literal elim.
$\text{:- not } e, g$	$\rightarrow$	~~$\text{:- not } e, g$~~	% pure literal elim.
	$\rightarrow$		
$\text{:- not } f, \text{not } c, \text{not } b$	$\rightarrow$	$\text{:- not } c, \text{not } b$	% self-subsum.
$\text{:- } f, \text{not } c$	$\rightarrow$		

Rewriting and Simplification (2)

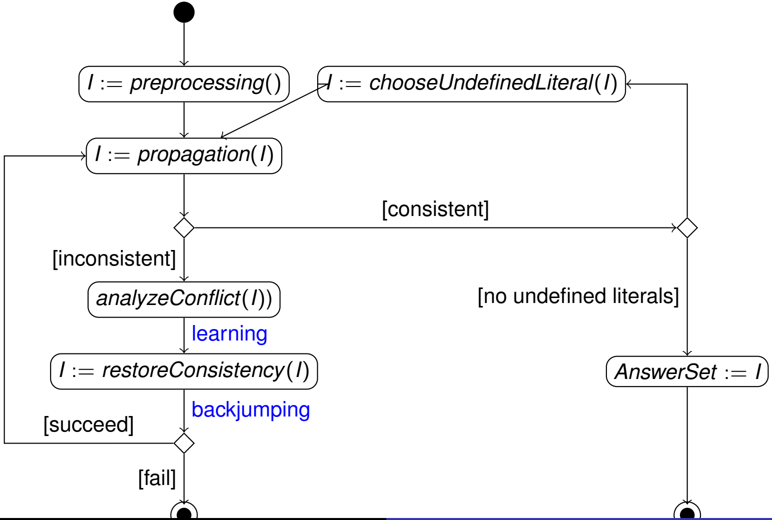
Simplifications (intuition, from `SatELite` [EB05, HJL$^+$15])

- Remove useless statements (subsumption-check)
- Reduce search space (variable-elimination)
- ... and more

Example (some simplifications)

$b \mid c. \quad f \mid g.$		$b \mid c. \quad f \mid g.$	
$a.$		$a.$	
$a :- \text{not } b, c$	$\rightarrow$	$\cancel{a :- \text{not } b, c}$	% subsumption
$:- \text{not } e, f$	$\rightarrow$	$\cancel{:- \text{not } e, f}$	% pure literal elim.
$:- \text{not } e, g$	$\rightarrow$	$\cancel{:- \text{not } e, g}$	% pure literal elim.
	$\rightarrow$		
$:- \text{not } f, \text{not } c, \text{not } b$	$\rightarrow$	$:- \text{not } c, \text{not } b$	% self-subsum.
$:- f, \text{not } c$	$\rightarrow$		

Model Generator: CDCL Algorithm



Heuristics and learning [MMZ$^+$01, ES03, MLM21]

Learning

- Detect the reason of a conflict
- Learn constraints using 1-UIP schema

Deletion Policy

- Exponentially many constraints → forget something
- Less “useful” constraints are removed

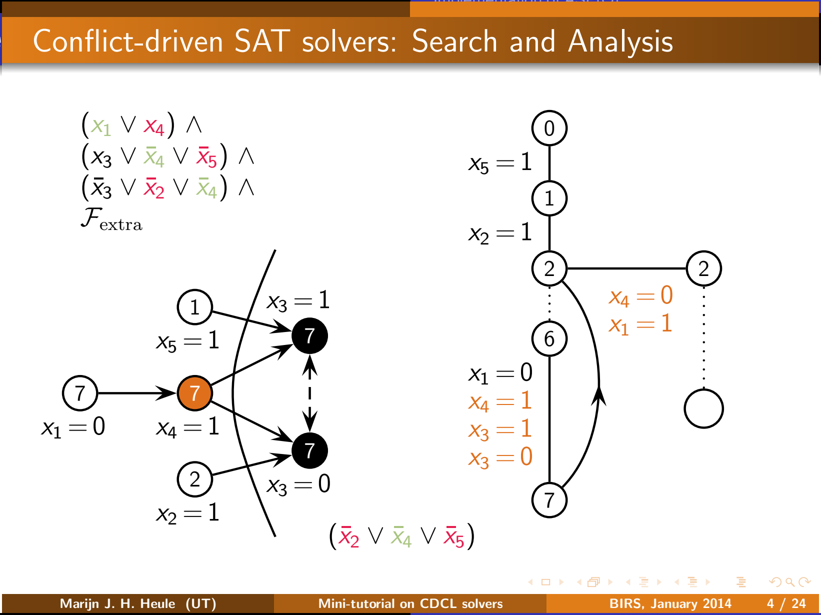
Search Restarts

- Escape “local minimum” by restarting the search
- Based on some heuristic sequence

Branching Heuristics

- VSIDS-Based heuristic

Le

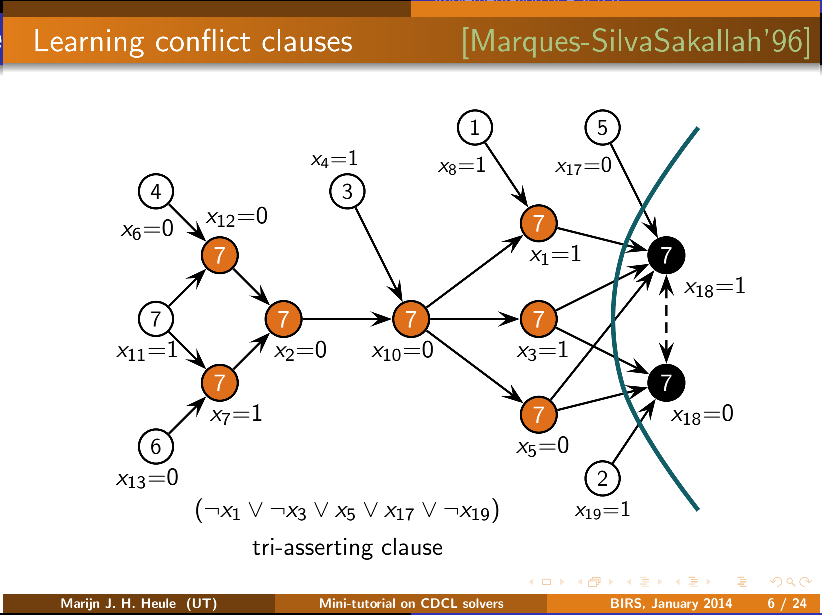


Conflict-driven SAT solvers: Search and Analysis

$$(x_1 \vee x_4) \wedge$$
$$(x_3 \vee \bar{x}_4 \vee \bar{x}_5) \wedge$$
$$(\bar{x}_3 \vee \bar{x}_2 \vee \bar{x}_4) \wedge$$
$$\mathcal{F}_{\text{extra}}$$

$(\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5)$

Le



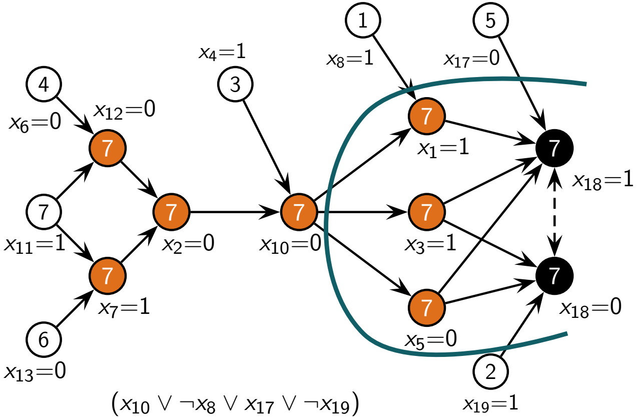
Learning conflict clauses [Marques-SilvaSakallah'96]

$(\neg x_1 \lor \neg x_3 \lor x_5 \lor x_{17} \lor \neg x_{19})$

tri-asserting clause

Le

Learning conflict clauses [Marques-SilvaSakallah'96]

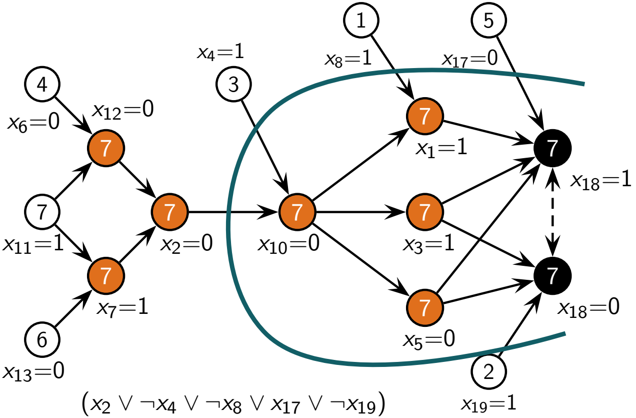


$(x_{10} \vee \neg x_8 \vee x_{17} \vee \neg x_{19})$

first unique implication point

Le

Learning conflict clauses [Marques-SilvaSakallah'96]



$(x_2 \vee \neg x_4 \vee \neg x_8 \vee x_{17} \vee \neg x_{19})$

second unique implication point

Outline

Grounding bottleneck

When traditional ASP solving is effective?

- When you can keep the grounding “small”!

The truth about grounding...

- Exponential process
- Might fill the entire memory
- Might be problematic also if only quadratic

The grounding bottleneck problem

ASP Systems Architectures

The “Grounding-less” architecture

- Instantiate rules if required during the search
- Interleave G&S to avoid the “grounding bottleneck”

→ ASPERIX [LefeN09], GASP [PaluDPR09]

→ OMIGA [DaoTrEFWW12], ALPHA [Weinzierl17,BJW19]

The “Compilation-based” architecture

- Simulate rules to avoid the “grounding bottleneck”
- Extend the solver with adhoc propagator procedures

→ Partial Compilation: WaspProp [MRD22, CDRS20]

→ Full Compilation: ProASP [DMR23, MDR24]

Compilation of ASP Programs

Compilation

“Translation of a program in a high-level language into another (lower-level) programming language”

Ideally by “compilation of ASP programs” we mean

1. Given a (non-ground) ASP program Π
2. Transform it in a C++ program Π_C
   → Optimized ad-hoc implementation
3. Run C++ building pipeline on Π_C
   → Obtain a specific binary executable
4. Run binary on different instances

Compilation of ASP Programs

Compilation

“Translation of a program in a high-level language into another (lower-level) programming language”

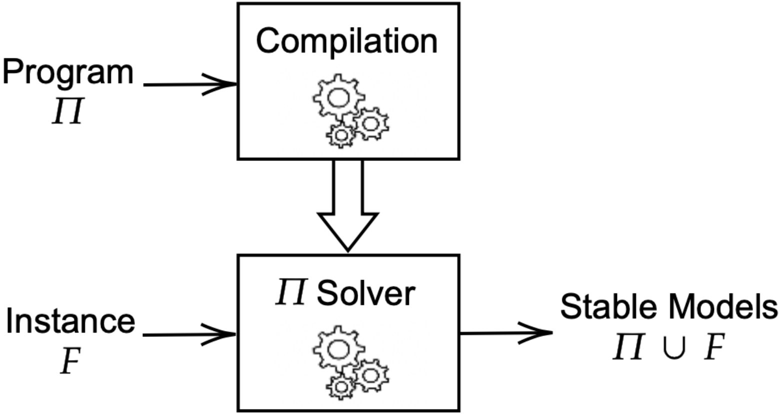
Ideally by “compilation of ASP programs” we mean

1. Given a (non-ground) ASP program Π
2. Transform it in a C++ program Π_C

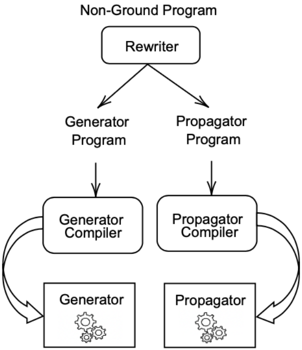
   → Optimized ad-hoc implementation → Runtime advantages!
3. Run C++ building pipeline on Π_C

   → Obtain a specific binary executable
4. Run binary on different instances → Fits ASP idea!!

Idea: Compilation of ASP Programs



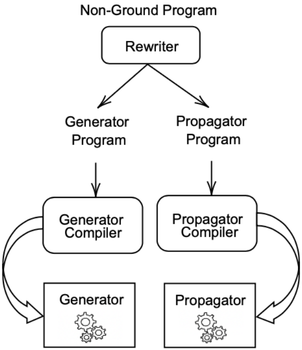
The ProASP System: Compilation Phase



Given a non-ground program Π:

1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}
    - Π^{Prop} simulates the propagation of Π
    - Π^{Gen} defines the domain of predicates in Π^{Gen}
2. Π^{Gen} is compiled into custom bottom-up evaluation procedures
3. Π^{Prop} is compiled into custom propagators

The ProASP System: Compilation Phase



Given a non-ground program Π:

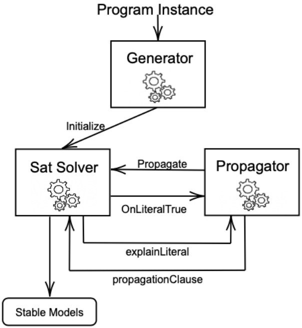
1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}
   - Π^{Prop} simulates the propagation of Π
   - Π^{Gen} defines the domain of predicates in Π^{Gen}
2. Π^{Gen} is compiled into custom bottom-up evaluation procedures
3. Π^{Prop} is compiled into custom propagators

Example: Compiled Propagator Module

```
Input  : A literal $l$, an interpretation $M$
Output: A set of literals $M_l$
begin
    $M_l := \emptyset$;
    if $\underline{pred(l) = \text{``}asgn\text{''} \textbf{ and } l \in M^+}$
      then
        $x := l[0]$;    $c := l[1]$;
        for $\underline{l_2 \in \{edge(x, y) \in M^+\}}$
          do
            $y := l_2[2]$;
            $M_l := M_l \cup \{\overline{asgn(y, c)}\}$
        end
        $y := l[0]$;    $c := l[1]$;
        for $\underline{l_2 \in \{edge(x, y) \in M^+\}}$
          do
            $x := l_2[2]$;
            $M_l := M_l \cup \{\overline{asgn(x, c)}\}$
        end
    end
    return $\underline{M_l}$
end
```

```
Input  : A literal $l$, an interpretation $M$
Output: A set of literals $M_l$
begin
    $M_l := \emptyset$;
    if $\underline{pred(l) = \text{``}asgn\text{''} \textbf{ and } l \in M^+}$
      then
        $x := l[0]$;    $c := l[1]$;
        $M_l := M_l \cup \{\overline{nAsgn(x, c)}\}$
    end
    if
      $\underline{pred(l) = \text{``}nAsgn\text{''} \textbf{ and } l \in M^+}$
      then
        $x := l[0]$;    $c := l[1]$;
        $M_l := M_l \cup \{\overline{asgn(x, c)}\}$
    end
    return $\underline{M_l}$
end
```

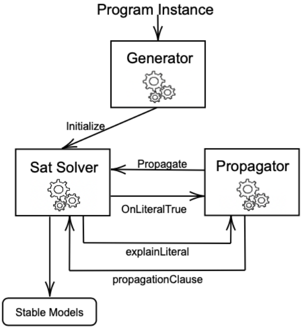
The ProASP System: Solving Phase



Given a program instance *F*:

1. The Generator module computes the domain of each predicate
2. Generated atoms are fed into the Sat Solver and CDCL starts
3. Each assigned literal activates the Propagator module, and rule inferences are propagated
4. Conflicts are analyzed in the Sat Solver asking the Propagator to reconstruct propagation clauses

The ProASP System: Solving Phase



Given a program instance *F*:

1. The Generator module computes the domain of each predicate
2. Generated atoms are fed into the Sat Solver and CDCL starts
3. Each assigned literal activates the Propagator module, and rule inferences are propagated
4. Conflicts are analyzed in the Sat Solver asking the Propagator to reconstruct propagation clauses

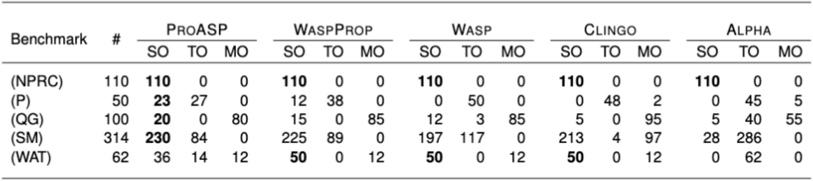
The ProASP System performance

Synthetic benchmark:

k	WASP-PROP		WASP		CLINGO	
	t	mem	t	mem	t	mem
1000	**0**	0	1.02	59.6	0.66	40.2
2000	**0.1**	18.6	4.36	286.5	3.23	303.5
3000	**0.24**	38.5	9.98	696.3	7.68	709.5
4000	**0.53**	53.7	18.47	1168.1	13.62	1216.8
5000	**0.93**	74.6	28.8	2215.4	22.23	1933.3
6000	**1.19**	109.8	42.47	2807.2	32.73	2871.1
7000	**1.44**	142.3	58.31	3402	42.88	3576.7
8000	**1.96**	152	-	-	-	-
9000	**2.89**	191.6	-	-	-	-
10000	**3.87**	254.7	-	-	-	-
20000	**12.52**	898.5	-	-	-	-
30000	**26.43**	1888.3	-	-	-	-
40000	**58.88**	3319.6	-	-	-	-
50000	-	-	-	-	-	-

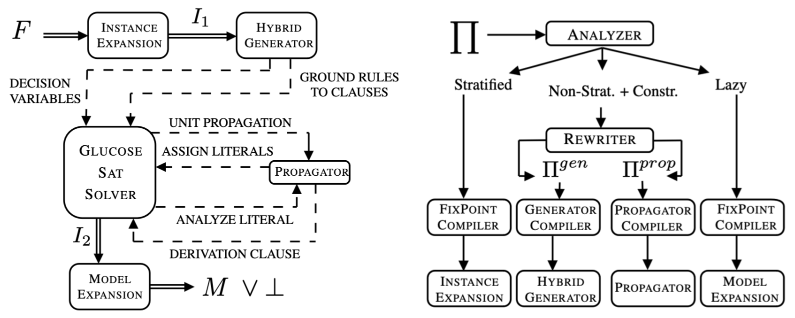
The ProASP System: Real instances

Considered benchmarks: Non Partition Removal Colouring **(NPRC)**, Packing Problem **(P)**, Quasi Group **(QG),** Stable Marriage **(SM)**, Weight Assignment Tree **(WAT)**



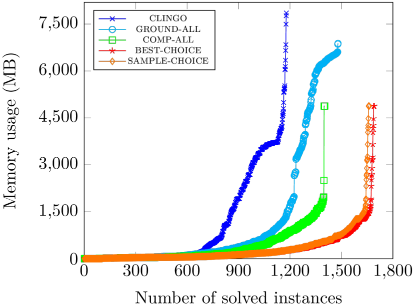
Benchmark	#	ProASP			WaspProp			Wasp			Clingo			Alpha		
		SO	TO	MO	SO	TO	MO	SO	TO	MO	SO	TO	MO	SO	TO	MO
(NPRC)	110	**110**	0	0	**110**	0	0	**110**	0	0	**110**	0	0	**110**	0	0
(P)	50	**23**	27	0	12	38	0	0	50	0	0	48	2	0	45	5
(QG)	100	**20**	0	80	15	0	85	12	3	85	5	0	95	5	40	55
(SM)	314	**230**	84	0	225	89	0	197	117	0	213	4	97	28	286	0
(WAT)	62	36	14	12	**50**	0	12	**50**	0	12	**50**	0	12	0	62	0

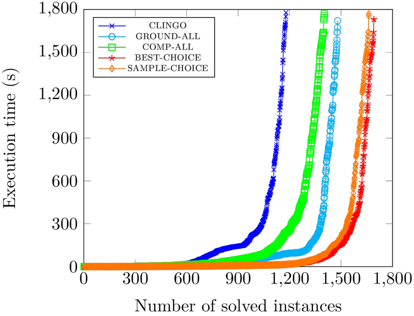
The ProASP System: Blending grounding and solving



- Also grounding can be compiled!
- *Fine-grained blending grounding and solving* [MDR24]

The ProASP System: Blending performance





Outline

1. Beyond NP with ASP
2. ASP solving overview
   - Instantiation
   - Model Generation & Checking
   - Grounding-less ASP
   - Implementation of ASP(Q)
3. Programming Hints

A solver for ASP(Q)

ASP(Q):

1. A natural solution for modeling beyond NP with ASP
    - ASP(Q) extends ASP via quantifiers over stable models
2. Clear computational properties of the language
3. Examples to show the modeling capabilities

Can be used in practice?

- QASP: Implementation by rewriting in QBF (LPNMR22) [ACRT22]
- PYQASP implementation in python featuring program optimizations and automatic selection of the backend (ICLP23) [FMR23]

A solver for ASP(Q)

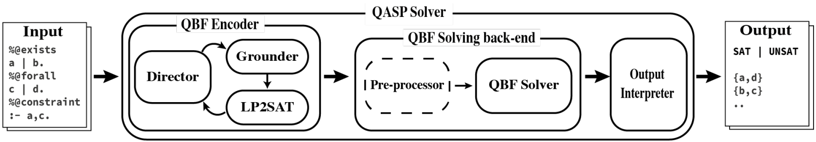
ASP(Q):

1. A natural solution for modeling beyond NP with ASP
   - ASP(Q) extends ASP via quantifiers over stable models
2. Clear computational properties of the language
3. Examples to show the modeling capabilities

Can be used in practice?

- QASP: Implementation by rewriting in QBF (LPNMR22) [ACRT22]
- PYQASP implementation in python featuring program optimizations and automatic selection of the backend (ICLP23) [FMR23]

System Architecture and Implementation



- **Input Language**
  - → Basically ASP
  - → Quantifiers annotated comments
- **Implementation**
  - → Existing ASP tools (gringo, lp2*)
  - → A choice of QBF pre-processors and solvers

QBF Encoder core (1)

Intermediate Groundings G_i

Let Π be an ASP(Q) program of the form (1), and let $G(P)$ denote the grounding of P:

$$G_i = \begin{cases} G(P_1) & i = 1 \\ G(P_i \cup CH(G_{i-1}, P_i)) & i \in [2..n] \\ G(C \cup CH(G_n, C)) & i = n+1 \end{cases}$$

where $CH(P, P') = ch(\bigcup_{p \in Int(P,P')} at(G(P), p))$.

Intuitively

- Each subprogram P_i is grounded separately
- A choice rule for the "interface" between P_i and P_{i+1}, $i > 1$

  → i.e., atoms that are passed from i-th to i+1-th program

QBF Encoder core (2)

QBF encoding $\Phi(\Pi)$

Let Π be an ASP(Q) program of the form (1).

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left(\bigwedge_{i=1}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c,$$

where $CNF(P)$ is a CNF formula encoding P; $\phi_1, \ldots, \phi_{n+1}$ are fresh prop. variables; $\boxplus_i = \exists x_i$ if $\Box_i = \exists^{st}$ or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ for $i = 1, \cdots, n+1$, and ϕ_c is the formula

$$\phi_c = \phi'_1 \odot_1 (\phi'_2 \odot_2 (\cdots \phi'_n \odot_n (\phi_{n+1}) \cdots))$$

where $\odot_i = \vee$ if $\Box_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi'_i = \neg\phi_i$ if $\Box_i = \forall^{st}$, and $\phi'_i = \phi_i$ otherwise.

Intuitively

- ϕ_i is satisfied iff P_i is coherent
- ϕ_c imposes semantics of ASP(Q) quantifiers

QBF Encoder core (2)

QBF encoding $\Phi(\Pi)$

Let Π be an ASP(Q) program of the form (1).

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left(\bigwedge_{i=1}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c,$$

where $CNF(P)$ is a CNF formula encoding P; $\phi_1, \ldots, \phi_{n+1}$ are fresh prop. variables; $\boxplus_i = \exists x_i$ if $\Box_i = \exists^{st}$ or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ for $i = 1, \cdots, n+1$, and ϕ_c is the formula

$$\phi_c = \phi_1' \odot_1 (\phi_2' \odot_2 (\cdots \phi_n' \odot_n (\phi_{n+1}) \cdots))$$

where $\odot_i = \vee$ if $\Box_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi_i' = \neg\phi_i$ if $\Box_i = \forall^{st}$, and $\phi_i' = \phi_i$ otherwise.

Theorem

Let Π be a quantified program. Then $\Phi(\Pi)$ is true iff Π is coherent.

Actual implementation: QBF Encoding (2)

Example

Let Π be the following ASP(Q) program:

$$
\begin{array}{l}
@exists\ \%\ P1 \\
a \leftarrow a,\ not\ b \\
c \leftarrow not\ a \\
\{b\} \leftarrow \\
@forall\ \%\ P2 \\
\{d(1..2)\} \leftarrow c \\
\{d(3..100)\} \leftarrow a \\
@constraint \\
\%\ C\ is\ empty
\end{array}
$$

Example

The resulting encoding is obtained as follows

$$
\begin{cases}
G_1 = P_1 \\
G_2 = P_2 \cup \{\ \{a; c\} \leftarrow \} \\
G_3 = C \cup \{\}
\end{cases}
$$

$$
\begin{array}{rl}
\Phi(\Pi) = & \exists^{st} a, b, c \\
& \forall^{st} d(1),\ \ldots,\ d(100) \\
& (\phi_1 \leftrightarrow CNF(G_1))\wedge \\
& (\phi_2 \leftrightarrow CNF(G_2))\wedge \\
& (\phi_3 \leftrightarrow CNF(G_3))\wedge \\
& \phi_1 \wedge (\neg\phi_2 \vee \phi_3)
\end{array}
$$

Actual implementation: QBF Encoding (2)

Example

Let Π be the following ASP(Q) program:

$$\begin{array}{l} @\mathit{exists}\ \%\ P1 \\ a \leftarrow a,\ \mathit{not}\ b \\ c \leftarrow \mathit{not}\ a \\ \{b\} \leftarrow \\ @\mathit{forall}\ \%\ P2 \\ \{d(1..2)\} \leftarrow c \\ \{d(3..100)\} \leftarrow a \\ @\mathit{constraint} \\ \%\ C\ \mathit{is\ empty} \end{array}$$

Example

The resulting encoding is obtained as follows

$$\begin{cases} G_1 = P_1 \\ G_2 = P_2 \cup \{\ \{a; c\} \leftarrow\} \\ G_3 = C \cup \{\} \end{cases}$$

$$\begin{array}{rl} \Phi(\Pi) = & \exists^{st} a, b, c \\ & \forall^{st} d(1),\ \ldots,\ d(100) \\ & (\phi_1 \leftrightarrow CNF(G_1)) \wedge \\ & (\phi_2 \leftrightarrow CNF(G_2)) \wedge \\ & (\phi_3 \leftrightarrow CNF(G_3)) \wedge \\ & \phi_1 \wedge (\neg \phi_2 \vee \phi_3) \end{array}$$

Benchmarks

- **Quantified Boolean Formulas (QBF)**
  - 933 Hard instances from QBF Lib
  - 2049 Easy and hard random instances
- **Argumentation Coherence (AC)**
  - 326 instances of ICCMA 2019
  - No dedicated solvers
- **Minmax Clique (MMC) [Ko95]**
  - 45 graphs from ASP Competitions
  - No dedicated solvers
- **Paracoherent ASP (PAR)**
  - 73 Existing benchmark [ADFR21]
  - 441 Random 3-SAT around the phase transition

Some experimental results

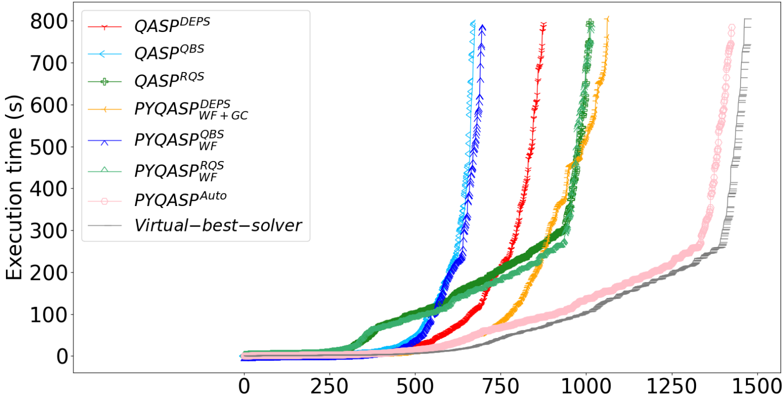


Figure: Comparison qasp vs pyqasp.

There is still space for improvement and new applications!

The PYQASP system: Download & Setup

1. PYQASP is available at:
   `https://github.com/MazzottaG/PyQASP`
2. Required python packages
   - joblib
   - scikit-learn
   - pyinstaller
3. From repository root run:
   - *./clean-install.bash pyqasp*
4. Use -h option to get available options:
   - *./dist/pyqasp -h*

Programming for performance

Programming for Performance

(hints)

Programming for performance: basic idea

Example (Maximal Clique)

Problem: Given an indirected Graph compute a clique of maximal size
Input: *node*(_) and *edge*(_, _).

Programming for performance: basic idea

Example (Maximal Clique)

Problem: Given an indirected Graph compute a clique of maximal size
Input: *node*(_) and *edge*(_, _).

Natural Encoding:

$inClique(X) \mid outClique(X) \text{ :- } node(X).$ % Guess
$\text{:- } inClique(X), inClique(Y), \text{not } edge(X,Y), X <> Y.$ % Check
$:\sim outClique(X).[1, X]$ % Optimize

Programming for performance: basic idea

Example (Maximal Clique)

Problem: Given an indirected Graph compute a clique of maximal size
Input: *node*(_) and *edge*(_, _).

Natural Encoding:

$inClique(X) \mid outClique(X) \text{ :- } node(X).$ % Guess
$\text{:- } inClique(X), inClique(Y), \mathrm{not}\ edge(X, Y), X <> Y.$ % Check
$:\sim outClique(X).[1, X]$ % Optimize

Optimized Encoding:

$inClique(X) \mid outClique(X) \text{ :- } node(X).$
$\text{:- } inClique(X), inClique(Y), \mathrm{not}\ edge(X, Y), X < Y.$ ← less constraints!
$:\sim outClique(X).[1, X]$

Programming for performance: basic idea (2)

Example (3-col- encoding 1)

% guess a coloring for the nodes

$col(X, red) \mid col(X, yellow) \mid col(X, green) \text{ :- } node(X).$

% check condition $\text{ :- } edge(X, Y), col(X, C), col(Y, C).$

Example (3-col- encoding 2)

% guess a coloring for the nodes

$col(X, red) \mid ncol(X, red) \text{ :- } node(X).$
$col(X, yellow) \mid ncol(X, yellow) \text{ :- } node(X).$
$col(X, green) \mid ncol(X, green) \text{ :- } node(X).$

% check condition

$\text{ :- } edge(X, Y), col(X, C), col(Y, C).$

$\text{ :- } col(X, C1), col(Y, C2), C1 <> C2.$

Programming for performance: basic idea (2)

Example (3-col- encoding 1)

% guess a coloring for the nodes

$col(X, red) \mid col(X, yellow) \mid col(X, green) \text{ :- } node(X).$

% check condition $\text{:- } edge(X, Y), col(X, C), col(Y, C).$

% NB: answer sets are subset minimal → only one color per node

Example (3-col- encoding 2)

% guess a coloring for the nodes

$col(X, red) \mid ncol(X, red) \text{ :- } node(X).$
$col(X, yellow) \mid ncol(X, yellow) \text{ :- } node(X).$
$col(X, green) \mid ncol(X, green) \text{ :- } node(X).$

% check condition

$\text{:- } edge(X, Y), col(X, C), col(Y, C).$

$\text{:- } col(X, C1), col(Y, C2), C1 <> C2.$ ← additional constraint

Programming for performance: basic idea (2)

Example (3-col- encoding 1)

% guess a coloring for the nodes

$col(X, red) \mid col(X, yellow) \mid col(X, green) \;{:}\!-\; node(X).$

% check condition $\;{:}\!-\; edge(X, Y), col(X, C), \; col(Y, C).$

Example (3-col- encoding 2 - Larger grounding!)

% guess a coloring for the nodes

$col(X, red) \mid ncol(X, red) \;{:}\!-\; node(X).$ ← three times

$col(X, yellow) \mid ncol(X, yellow) \;{:}\!-\; node(X).$ ← more

$col(X, green) \mid ncol(X, green) \;{:}\!-\; node(X).$ ← ground rules

% check condition

${:}\!-\; edge(X, Y), col(X, C), \; col(Y, C).$

${:}\!-\; col(X, C1), \; col(Y, C2), C1 <> C2.$ ← additional ground constraints

Programming for performance: basic idea (2)

Example (3-col- encoding 1)

% guess a coloring for the nodes

$col(X, red) \mid col(X, yellow) \mid col(X, green) \text{ :- } node(X).$

% check condition $\text{ :- } edge(X, Y), col(X, C), \ col(Y, C).$

Example (3-col- encoding 2 - Larger Search Space!)

% guess a coloring for the nodes

$col(X, red) \quad \mid ncol(X, red) \text{ :- } node(X).$ ← additional
$col(X, yellow) \mid ncol(X, yellow) \text{ :- } node(X).$ ← ground
$col(X, green) \mid ncol(X, green) \text{ :- } node(X).$ ← atoms

% check condition

$\text{:- } edge(X, Y), col(X, C), \ col(Y, C).$

$\text{:- } col(X, C1), \ col(Y, C2), C1 <> C2.$

Programming for performance: lesson learned

Prefer an encoding if:

- Easier to ground

  → precomputes as much as possible
- Smaller instantiation

  → use e.g., minimality, aggregates, ...
- Produces less ground disjunctive rules and less “guessed atoms”

  → smaller search space

  → exponential gain

Stable Marriage

Definition

Given *n* men and *n* women, where each person has ranked all members of the opposite sex with a unique number between 1 and *n* in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners.

M	W
john	mary
luca	anna

P1	P2	Pref
john	mary	1
john	anna	2
luca	mary	2
luca	anna	1

P1	P2	Pref
mary	john	1
anna	john	2
mary	luca	2
anna	luca	1

Stable Marriage: Natural Encoding

```
% guess matching
match(M,W) | nMatch(M,W) :– man(M), woman(W).

% no polygamy
:– match(M1,W), match(M,W), M <> M1.
:– match(M,W), match(M,W1), W <> W1.

% no singles
married(M) :- match(M,W).
:- man(M), not married(M).

% strong stability condition
:- match(M,W1), match(M1,W), W1 <> W,
   pref(M,W,Smw), pref(M,W1,Smw1), Smw > Smw1,
   pref(W,M,Swm), pref(W,M1,Swm1), Swm >= Swm1.
```

Stable Marriage: First Optimization

```
% guess matching
{match(M,W)} :- man(M), woman(W).

% no polygamy
 :- match(M1,W), match(M,W), M <> M1.
 :- match(M,W), match(M,W1), W <> W1.

% no singles
married(M) :- match(M,W).
 :- man(M), not married(M).

% strong stability condition
 :- match(M,W1), match(M1,W), W1 <> W,
  pref(M,W,Smw), pref(M,W1,Smw1), Smw > Smw1,
  pref(W,M,Swm), pref(W,M1,Swm1), Swm >= Swm1.
```

Stable Marriage: Second Optimization

```
% guess matching
{match(M,W) : woman(W)} = 1  :-  man(M).

% no singles
married(M) :- match(M,W).
 :-  woman(M), not married(M).

% strong stability condition
 :-  match(M,W1), match(M1,W), W1 <> W,
     pref(M,W,Smw), pref(M,W1,Smw1), Smw > Smw1,
     pref(W,M,Swm), pref(W,M1,Swm1), Swm >= Swm1.
```

Stable Marriage: Third Optimization

```
% guess matching
{match(M,W) : woman(W)} = 1  :–  man(M).

% no singles
married(M) :- match(M,W).
 :–  woman(M), not married(M).

% strong stability condition
  matched(m,M,S)          match(M,W), pref(M,W,S).
  matched(w,W,S-1)        match(M,W), pref(W,M,S), S > 1.
  matched(T,P,S-1)        matched(T,P,S), S > 1.
 :–  pref(M,W,R), pref(W,M,S), not matched(m,M,R), not
matched(w,W,S).
```

Stable Marriage: Impact

In practice (Tested on one instance from 3rd ASP Competition)

Encoding	**Time**	**Number of rules**
Natural encoding	25 seconds	approx. 6 millions
First optimization	25 seconds	approx. 6 millions
Second optimization	22 seconds	approx. 6 millions
Third optimization	0.3 seconds	approx. 40 thousands

Acknowledgments

Thanks for your attention!

Questions?

References

[AAC$^+$18] Weronika T. Adrian, Mario Alviano, Francesco Calimeri, Bernardo Cuteri, Carmine Dodaro, Wolfgang Faber, Davide Fuscà, Nicola Leone, Marco Manna, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV: advancements and applications. KI, 32(2-3):177–179, 2018.

[ACRT22] Giovanni Amendola, Bernardo Cuteri, Francesco Ricca, and Mirek Truszczynski. Solving problems in the polynomial hierarchy with ASP(Q). In LPNMR, volume 13416 of Lecture Notes in Computer Science, pages 373–386. Springer, 2022.

[ADFR21] Giovanni Amendola, Carmine Dodaro, Wolfgang Faber, and Francesco Ricca. Paracoherent answer set computation. Artif. Intell., 299:103519, 2021.

[ART19] Giovanni Amendola, Francesco Ricca, and Miroslaw Truszczynski. Beyond NP: quantifying over answer sets. Theory Pract. Log. Program., 19(5-6):705–721, 2019.

References (cont.)

[BEHW89] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. J. ACM, 36(4):929–965, 1989.

[BET11] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. Commun. ACM, 54(12):92–103, 2011.

[CDG$^+$95] Feng Cao, Ding-Zhu Du, Biao Gao, Peng-Jun Wan, and Panos M. Pardalos. Minimax Problems in Combinatorial Optimization, pages 269–292. Springer US, Boston, MA, 1995.

[CDRS20] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In IJCAI, pages 1688–1694. ijcai.org, 2020.

References (cont.)

[Cla77] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. ACM Comput. Surv., 33(3):374–425, 2001.

[DMR23] Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Compilation of tight ASP programs. In ECAI, volume 372 of Frontiers in Artificial Intelligence and Applications, pages 557–564. IOS Press, 2023.

References (cont.)

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings, volume 3569 of Lecture Notes in Computer Science, pages 61–75. Springer, 2005.

[EFLP00] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the dlv system. In Logic-based Artificial Intelligence, pages 79–103. 2000.

[EG95] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. Ann. Math. Artif. Intell., 15(3-4):289–323, 1995.

[EGL16] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. AI Magazine, 37(3):53–68, 2016.

References (cont.)

[EL03] Esra Erdem and Vladimir Lifschitz. Tight logic programs. Theory Pract. Log. Program., 3(4-5):499–518, 2003.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, volume 2919 of Lecture Notes in Computer Science, pages 502–518. Springer, 2003.

[FMR23] Wolfgang Faber, Giuseppe Mazzotta, and Francesco Ricca. An efficient solver for ASP(Q). Theory Pract. Log. Program., (to appear), 2023.

[GKK$^+$16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In ICLP (Technical Communications), volume 52 of OASICS, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

References (cont.)

[GKS11] Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. TPLP, 11(4-5):821–839, 2011.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. New Generation Comput., 9(3/4):365–386, 1991.

[GLM+18] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In IJCAI, pages 5450–5456. ijcai.org, 2018.

[HJL+15] Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. J. Artif. Intell. Res., 53:127–168, 2015.

References (cont.)

[Ko95] Chih-Long Ko, Ker-Iand Lin. On the Complexity of Min-Max Optimization Problems and their Approximation, pages 219–239. Springer US, Boston, MA, 1995.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. Artif. Intell., 138(1-2):39–54, 2002.

[MC06] Kevin Milans and Bryan Clark. The complexity of graph pebbling. SIAM J. Discret. Math., 20(3):769–798, March 2006.

[MDR24] Giuseppe Mazzotta, Carmine Dodaro, and Francesco Ricca. Blending grounding and compilation for efficient asp solving. In KR, page to appear, 2024.

References (cont.)

[MLM21] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, Handbook of Satisfiability - Second Edition, volume 336 of Frontiers in Artificial Intelligence and Applications, pages 133–182. IOS Press, 2021.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001, pages 530–535. ACM, 2001.

[MRD22] Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. Compilation of aggregates in ASP systems. In AAAI, pages 5834–5841. AAAI Press, 2022.