Processing of Declarative Knowledge –Evaluation of ASP Programs–

Francesco Ricca

Computational Intelligence Curriculum
Institute of Information Systems

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     $\rightarrow$ 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
    $\rightarrow$ you can write programs
  - Knowledge of the evaluation process
    $\rightarrow$you can write programs more efficiently
  - Knowledge of the ASP system
    $\rightarrow$ you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     $\rightarrow$ 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
    $\rightarrow$ you can write programs
  - Knowledge of the evaluation process
    $\rightarrow$you can write programs more efficiently
  - Knowledge of an ASP System
    $\rightarrow$ you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     $\rightarrow$ 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
    $\rightarrow$ you can write programs
  - Knowledge of the evaluation process
    $\rightarrow$you can write programs more efficiently
  - Knowledge of an ASP System
    $\rightarrow$ you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     $\rightarrow$ 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
    $\rightarrow$ you can write programs
  - Knowledge of the evaluation process
    $\rightarrow$you can write programs more efficiently
  - Knowledge of an ASP System
    $\rightarrow$ you can actually implement applications

Introduction: Evaluation of ASP Programs

- **The idea of ASP**:
  1. Write a program representing a computational problem
     $\rightarrow$ 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
    $\rightarrow$ you can write programs
  - Knowledge of the evaluation process
    $\rightarrow$you can write programs more efficiently
  - Knowledge of an ASP System
    $\rightarrow$ you can actually implement applications

Evaluation of ASP Programs (1)

Computationally expensive
Traditionally a two-step process:

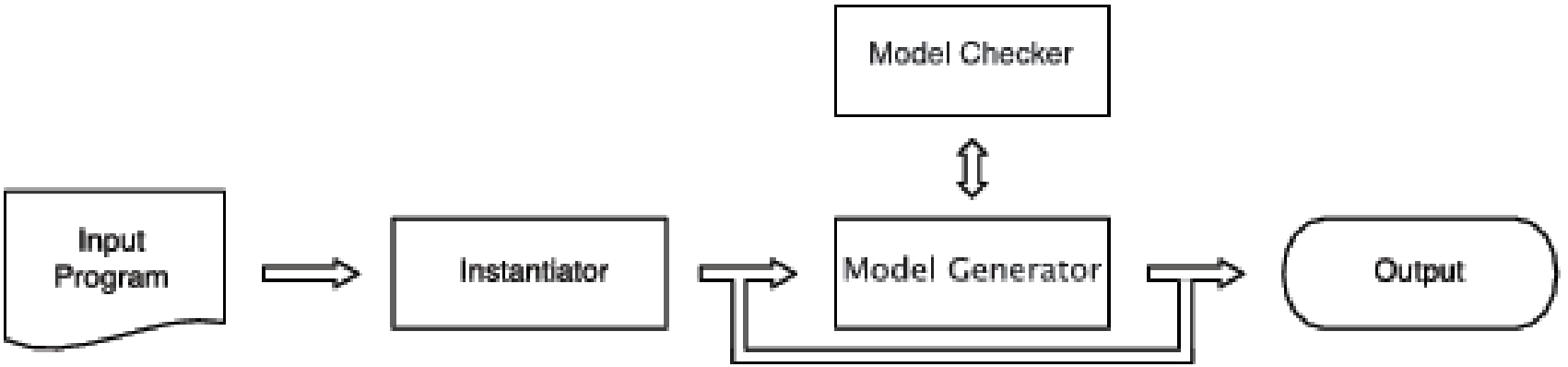
1. **Instantiation (or grounding)**

   → Variable elimination

2. **Propositional search**

   → Model Generation: “generate models”

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



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

% discard colorings where adjacent nodes have the same color

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

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

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

% 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: $\rightarrow$ is huge (2916 rules) and redundant!

$col(red, red) \mid col(red, yellow) \mid col(red, green) \text{ :- } node(red).$
$col(yellow, red) \mid col(yellow, yellow) \mid col(yellow, green) \text{ :- } node(yellow).$
$col(green, red) \mid col(green, yellow) \mid col(green, green) \text{ :- } node(green).$
...
$col(1, red) \mid col(1, yellow) \mid col(1, green) \text{ :- } node(1).$ $\leftarrow$ OK!
...
$\text{:- } edge(1, 2), col(1, 1), col(2, 1).$ $\leftarrow$ redundant!
...
$\text{:- } edge(1, 2), col(1, red), col(2, red).$ $\leftarrow$ 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: $\rightarrow$ equivalent but much smaller (9 rules)!

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

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

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, θ: 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 (* θ is a total substitution for the variables of R *)
        $S := S \cup \theta$;
        $L := PreviousLiteral(L)$;
        (* look for another solution *)
        $MatchFound :=$ False;
        $\theta := \theta\ |_{PreviousVars(L)}$;
    else
      $L := PreviousLiteral(L)$;
      $\theta := \theta\ |_{PreviousVars(L)}$;
  output S;
end;

Instantiation of a Program

Substitutions:

- generate rules
- derive knowledge

Advanced Techniques:

- Join ordering
- Backjumping

Instantiating a Program

- Handle recursion
- Handle negation

Instantiation of a Program

Substitutions:

- generate rules
- derive knowledge

Advanced Techniques:

- Join ordering
- Backjumping

Instantiating a Program

- Handle recursion
- Handle negation

Instantiation of a Program

Substitutions:

- generate rules
- derive knowledge

Advanced Techniques:

- Join ordering
- Backjumping

Instantiating a Program

- Handle recursion
- Handle negation

Dependency & Component Graphs

$a(1).\ t(X, Y) \text{ :- } p(X, Y), a(Y).$

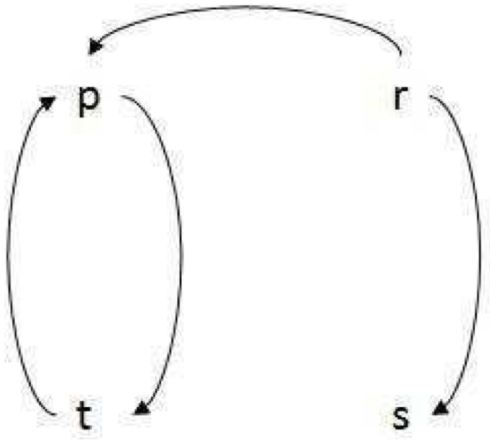
$p(X, Y) | s(Y) \text{ :- } r(X), r(Y).$

$p(X, Y) \text{ :- } r(X), t(X, Y).$

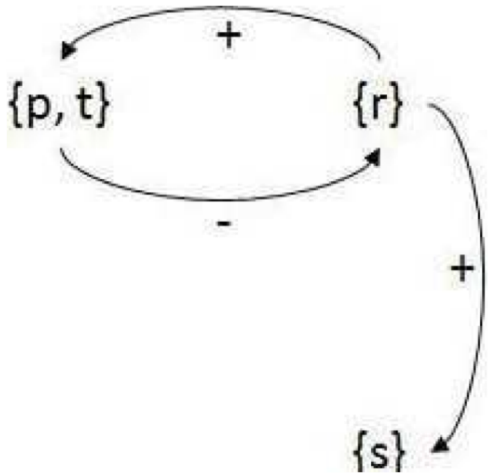
$r(X) \text{ :- } a(X), \text{not } t(X, X).$

Dependency & Component Graphs

$a(1).\ t(X, Y) \text{ :- } p(X, Y), a(Y).$
$p(X, Y) | s(Y) \text{ :- } r(X), r(Y).$
$p(X, Y) \text{ :- } r(X), t(X, Y).$
$r(X) \text{ :- } a(X), \text{not } t(X, X).$

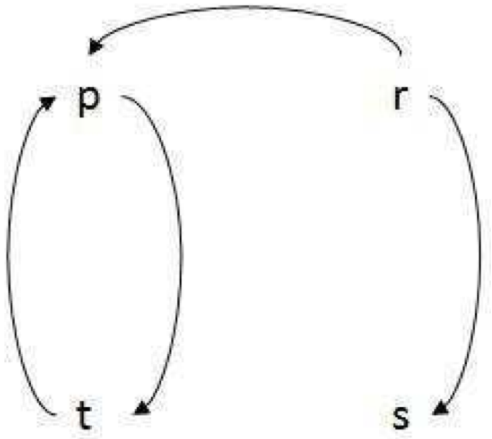


$G_{\mathcal{P}}$

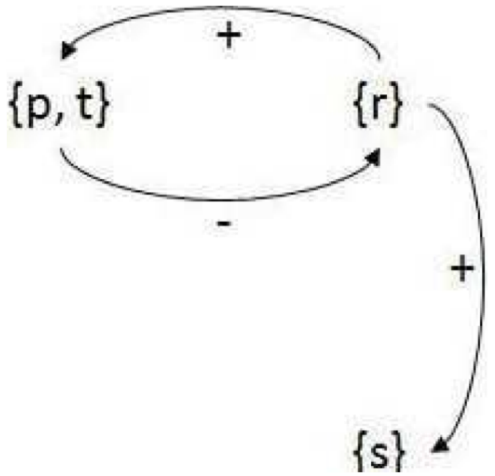


$G^c_{\mathcal{P}}$

Subprograms



$G_{\mathcal{P}}$



$G^c_{\mathcal{P}}$

$$P_{\{p,t\}} = \{p(X,Y) \mid s(Y) \text{ :- } r(X), r(Y).$$
$$p(X,Y) \text{ :- } r(X), t(X,Y).\}$$
$$t(X,Y) \text{ :- } p(X,Y), a(Y).\}$$
$$P_{\{s\}} = \{p(X,Y) \mid s(Y) \text{ :- } r(X), r(Y).\}$$
$$P_{\{r\}} = \{r(X) \text{ :- } a(X), \text{not } t(X,X).\}$$

Component Ordering

Exit and Recursive Rules

Given a component C ,a rule r in P_C

- is recursive if there is a predicate $p \in C$ s.t. p occurs in the positive body of r
- otherwise, r is said to be an exit rule.

$$P_{\{p,t\}} = \{p(X,Y)|s(Y) :- \ r(X), r(Y). \Leftarrow \text{exit}$$
$$p(X,Y) :- \ r(X), t(X,Y).\} \Leftarrow \text{recursive}$$
$$t(X,Y) :- \ p(X,Y), a(Y).\} \Leftarrow \text{recursive}$$
$$P_{\{s\}} = \{p(X,Y)|s(Y) :- \ r(X), r(Y).\} \Leftarrow \text{exit}$$
$$P_{\{r\}} = \{r(X) :- \ a(X), \text{not } t(X,X).\} \Leftarrow \text{exit}$$

Component Ordering

Component Ordering:

$A \prec_+ B$ If there is a path in G_P^c from A to B in which all arcs are labeled with "+"

$A \prec_- B$ If there is a path in G_P^c from A to B in which at least one arc is labeled with "-"

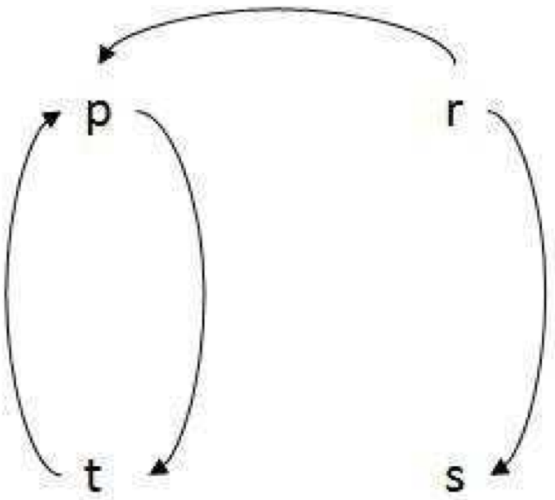
Admissible Component Sequence

Sequence $C_1, \ldots, C_n$ is admissible if for each $i < j$:

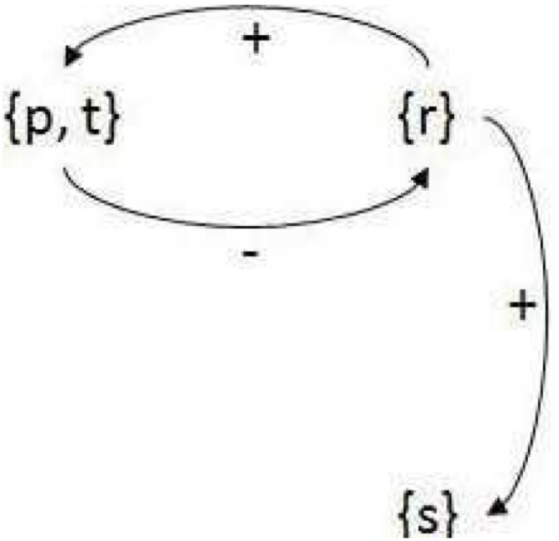
(*i*) $C_i \not\prec_+ C_j$, and

(*ii*) if $C_i \not\prec_- C_j$ there is a cycle in G_P^c.

Admissible Sequence: Example



$G_{\mathcal{P}}$



$G^{c}_{\mathcal{P}}$

Admissible Component Sequence: $\{r\}, \{p, t\}, \{s\}$

Instantiation of a Program: follow dependencies

Procedure *Instantiate*($\mathcal{P}$: Program; $G^c_{\mathcal{P}}$: ComponentGraph; **var** Π: GroundProgram)

var S: SetOfAtoms, $(C_1, \ldots, C_n)$: List of nodes of $G^c_{\mathcal{P}}$;

$S = EDB(\mathcal{P})$; $\Pi := \emptyset$;

$(C_1, \ldots, C_n) := OrderedNodes(G^c_{\mathcal{P}})$; /* *admissible component sequence* */

for i = 1 ... n **do** $InstantiateModule(\mathcal{P}, C_i, S, \Pi)$;

Instantiation of a Program: semi-naïve

Procedure *InstantiateModule* ($\mathcal{P}$: Program; C: SetOfPredicates;
var S: SetOfAtoms; **var** Π: GroundProgram)

var $\mathcal{N}S$: SetOfAtoms, ΔS: SetOfAtoms;
$\mathcal{N}S := \emptyset$; $\Delta S := \emptyset$;
for each $r \in Exit(C, \mathcal{P})$ **do** $InstantiateRule(r, S, \Delta S, \mathcal{N}S, \Pi)$;
do
$\Delta S := \mathcal{N}S$; $\mathcal{N}S = \emptyset$;
for each $r \in Recursive(C, \mathcal{P})$ **do** $InstantiateRule(r, S, \Delta S, \mathcal{N}S, \Pi)$;
$S := S \cup \Delta S$;
while $\mathcal{N}S \neq \emptyset$

Procedure *InstantiateRule*(r: rule; S: SetOfAtoms; ΔS: SetOfAtoms;
var $\mathcal{N}S$: SetOfAtoms; **var** Π: GroundProgram)

/* *Given S and ΔS builds the ground instances of r, simplifies them (see Sec. 4.3), adds them to Π, and add to $\mathcal{N}S$ the head atoms of the generated ground rules.* */

Program Simplification (intuition)

Remove redundant literals/rules

- If a positive body literal Q is in $B(r)$ and $Q \in S$, then delete Q from $B(r)$.
- If a solved negative body literal $\text{not } Q$ is in $B(r)$ and $Q \notin S$, then delete $\text{not } Q$ from $B(r)$.
- If a negative body literal $\text{not } Q$ is in $B(r)$ and $Q \notin S$, then remove the ground instance of r.

Intelligent Instantiator

The instantiation process

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

Intelligent Instantiator

The instantiation process

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

Intelligent Instantiator

The instantiation process

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

Model Generation & Checking

Model Generation $\rightarrow$ *produces candidate models*

- Similar to a SAT solver
- Davis-Putnam-Logeman-Loveland method
  - Propagate Deterministic Consequences
    $\rightarrow$ Unit Propagation
    $\rightarrow$ Support Propagation
    $\rightarrow$ Well-founded Negation
  - Assume a literal *l* (heuristically) until a model is generated
  - Upon inconsistency Backtrack (assume not *l*)

Model Checker $\rightarrow$ *checks if candidates are Answer Sets*

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

Model Generation & Checking

Model Generation $\rightarrow$ *produces candidate models*

- Similar to a SAT solver
- Davis-Putnam-Logeman-Loveland method
  - Propagate Deterministic Consequences
    $\rightarrow$ Unit Propagation
    $\rightarrow$ Support Propagation
    $\rightarrow$ Well-founded Negation
  - Assume a literal *l* (heuristically) until a model is generated
  - Upon inconsistency Backtrack (assume not *l*)

Model Checker $\rightarrow$ *checks if candidates are Answer Sets*

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

Model Generator

Algorithm 1: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

Model Generator

Algorithm 2: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

Model Generator

Algorithm 3: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

Model Generator

Algorithm 4: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

Model Generator

Algorithm 5: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

Model Generator

Algorithm 6: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

Model Generator

Algorithm 7: Compute Answer Set

Input : An interpretation I for a program Π
Output: True if Π admits answer set, false otherwise

```
1  begin
2      if ! Propagate(I) then
3          return false ;
4      if I is total then
5          return CheckModel(I)
6      ℓ := ChooseUndefinedLiteral();
7      if ComputeAnswerSet(I ∪ {ℓ}) then
8          return true;
9      if ComputeAnswerSet(I ∪ {not ℓ}) then
10         return true;
11     else
12         return false;
```

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)

$a \mid b :\!\!- \; c.$

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)

$a \mid b \text{ :- } c.$
$a \mid d \text{ :- not } b.$
If b and c are false and d is true infer a 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:

(i) $B(r) \cap \neg I \neq \emptyset$ (ii) $B^+(r) \cap U \neq \emptyset$ (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 $\rightarrow$ *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. $\quad S := S \cup \{\ b_1 \vee \cdots \vee b_m \leftarrow a_1 \wedge \cdots \wedge a_n\ \}$;
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: $M = \{a, b\}$

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

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

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

Example: build SAT Formula

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

$a \mid b \mid \sout{c}.$
$a :\!- b.$
$b :\!- a.$

Example: build SAT Formula

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

$a \mid b.$
$a :\!- b.$
$b :\!- a.$

Example: build SAT Formula

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

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

Example: build SAT Formula

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

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

Example: build SAT Formula

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

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

Example: build SAT Formula

Step: (6-9)

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

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

a :- $b. \Longrightarrow\ b \leftarrow a$

b :- $a. \Longrightarrow\ a \leftarrow b$

Example: build SAT Formula

Step: (6-9)

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

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

$a :\!- b. \Longrightarrow\ b \leftarrow a \Longrightarrow\ \neg a \vee b$

$b :\!- a. \Longrightarrow\ a \leftarrow b \Longrightarrow\ \neg b \vee a$

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 $\rightarrow$ Answer Set!

Example: build SAT Formula

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

$a \mid b \mid c$

a :– b.

b :– a, not c.

a :– c.

Example: build SAT Formula

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

$a \mid b \mid c \implies a \vee c$

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

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

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

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!

Programming for performance: basic idea

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:

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

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

Optimized Encoding:

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

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 $\rightarrow$ 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

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

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

Programming for performance: basic idea (2)

Example (3-col- encoding 1)

% guess a coloring for the nodes

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

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

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

% guess a coloring for the nodes

$col(X, red) \mid ncol(X, red) \text{ :- } node(X).$ ← three times
$col(X, yellow) \mid ncol(X, yellow) \text{ :- } node(X).$ ← more
$col(X, green) \mid ncol(X, green) \text{ :- } node(X).$ ← ground rules

% check condition

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

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

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

% guess a coloring for the nodes

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

  $\rightarrow$ precomputes as much as possible
- Smaller instantiation

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

  $\rightarrow$ smaller search space

  $\rightarrow$ exponential gain

Exercises

Minumim Spanning Tree

Given a weighted graph by means of edge(Node1, Node2, Cost), and node(N), compute a tree that starts at a root node, spans that graph, and has minimum cost.

Seating

A gala dinner has to be organized and table composition must satisfy a number of requirements:

- *Each table has nc chairs.*
- *Each guest must be assigned one and only one table.*
- *People liking each other should sit at the same table.*
- *People disliking each other should not sit at the same table.*

Solution

Example (Minimum Spanning Tree)

Problem: Given a Weighted Graph compute a tree that starts at a root node, spans that graph, and has minimum cost
Input: *node*(_) and *edge*(_, _, _), and *root*(_).

```
% Guess the edges that are part of the tree:                          | Guess
inTree(X, Y) | outTree(X, Y) :- edge(X, Y).
% Check that this is a tree!
:- root(X), inTree(_, X). % root in-degree is 0                       | Check
:- inTree(X, Y), inTree(X1, Y), X <> X1. % nodes in-degree is 1       |
:- node(X), not reached(X). % a tree is connected                     |
reached(X) :- reached(Y), inTree(Y, X).                               | Aux.
reached(X) :- root(X).                                                | Rules
% Minimize the sum of distances
:~ inTree(X, Y), edge(X, Y, C). [C, X, Y, C]                          | Opt.
```

Solution

Example (Minimum Spanning Tree)

Problem: Given a Weighted Graph compute a tree that starts at a root node, spans that graph, and has minimum cost
Input: *node*(_) and *edge*(_, _, _), and *root*(_).

```
% Guess the edges that are part of the tree:                          | Guess
inTree(X, Y) | outTree(X, Y) :- edge(X, Y).
% Check that this is a tree!
:- root(X), inTree(_, X). % root in-degree is 0                       | Check
:- inTree(X, Y), inTree(X1, Y), X <> X1. % nodes in-degree is 1       |
:- node(X), not reached(X). % a tree is connected                     |

reached(X) :- reached(Y), inTree(Y, X).                               | Aux.
reached(X) :- root(X).                                                | Rules

% Minimize the sum of distances
:~ inTree(X, Y), edge(X, Y, C). [C, X, Y, C]                          | Opt.
```

Solution

Example (Minimum Spanning Tree)

Problem: Given a Weighted Graph compute a tree that starts at a root node, spans that graph, and has minimum cost
Input: *node*(_) and *edge*(_, _, _), and *root*(_).

```
% Guess the edges that are part of the tree:                          | Guess
inTree(X, Y) | outTree(X, Y) :- edge(X, Y).
% Check that this is a tree!
:- root(X), inTree(_, X). % root in-degree is 0                       | Check
:- inTree(X, Y), inTree(X1, Y), X <> X1. % nodes in-degree is 1       |
:- node(X), not reached(X). % a tree is connected                     |
reached(X) :- reached(Y), inTree(Y, X).                               | Aux.
reached(X) :- root(X).                                                | Rules
% Minimize the sum of distances
:~ inTree(X, Y), edge(X, Y, C). [C, X, Y, C]                          | Opt.
```

Solution

Example (Seating Problem)

Problem: Organize table composition such that:
(*i*) Each table has nc chairs; (*ii*) Only one table per guest;
(*iii*) People liking each other should sit at the same table;
(*iv*) People disliking each other should not sit at the same table.
Input: $guest(P)$, $table(T)$, $like(P1, P2)$, $dislike(P1, P2)$, $guestPerTable(N)$

```
% Generate a sitting arrangement for guests.                          | Guess
  at(P,T)|nat(P,T) :- guest(P), table(T).

% Each table must not host more than nc guests.                       | Check
  :- table(T), not #count{P : at(P,T)} <= N, guestPerTable(N).        |
% Each guest must be assigned one and only one table.                 |
  :- guest(P), not #count{T : at(P,T)} = 1.                           |
% People liking each other should sit at the same table.             |
  :- like(P1,P2), at(P1,T), not at(P2,T).                             |
% People disliking each other should not sit at the same table.      |
  :- dislike(P1,P2), at(P1,T), at(P2,T).                              |
```

Exercises

Minumim Node Cover

Given a graph G modeled by edge$(Node1, Node2)$*, and node*(N)*, find a minimum node cover, that is, a subset MNC of nodes of minimum cardinality such that for each edge* (u, v) *in G at least one of u and v belongs to MNC.*