Systems and Solving Techniques for Knowledge Representation

– Grounding –

Marco Maratea
University of Genoa, Italy

066 011 Double degree programme Computational Logic (Erasmus-Mundus)
066 931 Computational Intelligence
066 937 Software Engineering & Internet Computing
Institute of Information Systems

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
     $\rightarrow$ 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
     → 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

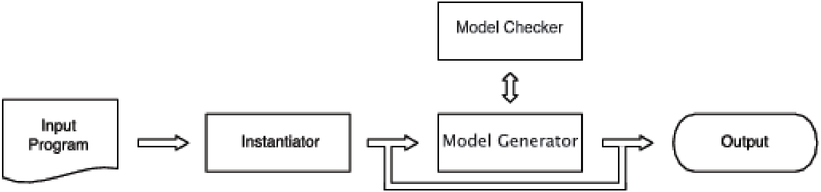
Evaluation of ASP Programs (1)

Computationally expensive
Traditionally a two-step process:

1. **Instantiation (or grounding)**
   $\rightarrow$ Variable elimination
2. **Propositional search** (depends on complexity, details later)

   $\rightarrow$ Model Generation: "generate models"

   $\rightarrow$ (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*

$\rightarrow$ Not viable in practice

Intelligent instantiation

$\rightarrow$ Keep the size of the instantiation as small as possible

$\rightarrow$ Equivalent to the full one

$\rightarrow$ Intelligent Instantiators can solve problems in P

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*

$\rightarrow$ Not viable in practice

Intelligent instantiation

$\rightarrow$ Keep the size of the instantiation as small as possible

$\rightarrow$ Equivalent to the full one

$\rightarrow$ Intelligent Instantiators can solve problems in P

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*

$\rightarrow$ Not viable in practice

Intelligent instantiation

$\rightarrow$ Keep the size of the instantiation as small as possible

$\rightarrow$ Equivalent to the full one

$\rightarrow$ Intelligent Instantiators can solve problems in P

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

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!
...
$\text{:-}\; edge(1, 2), col(1, 1),\; col(2, 1).$
...
$\text{:-}\; edge(1, 2), col(1, red),\; col(2, red).$ $\leftarrow$ 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;
                $\theta := \theta \mid_{PreviousVars(L)}$;
        else
            $L := PreviousLiteral(L)$;
            $\theta := \theta \mid_{PreviousVars(L)}$;
    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

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

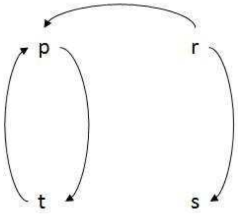
Dependency & Component Graphs

$a(1).\ t(X,Y) \coloneq p(X,Y), a(Y).$

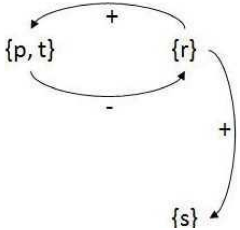
$p(X,Y)|s(Y) \coloneq r(X), r(Y).$

$p(X,Y) \coloneq r(X), t(X,Y).$

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

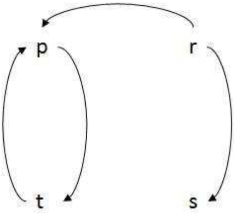


$G_{\mathcal{P}}$

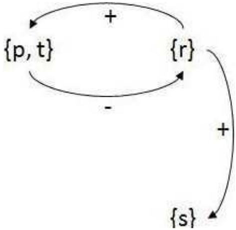


$G^c_{\mathcal{P}}$

Subprograms



$G_{\mathcal{P}}$



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

$$
\begin{aligned}
P_{\{p,t\}} = \{ & p(X,Y) | s(Y) \coloneq r(X), r(Y). \\
& p(X,Y) \coloneq r(X), t(X,Y).\} \\
& t(X,Y) \coloneq p(X,Y), a(Y).\} \\
P_{\{s\}} = \{ & p(X,Y) | s(Y) \coloneq r(X), r(Y).\} \\
P_{\{r\}} = \{ & r(X) \coloneq a(X), \mathrm{not}\ t(X,X).\}
\end{aligned}
$$

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.

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

Component Ordering

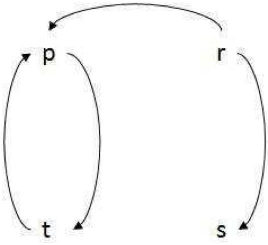
Component Ordering:

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

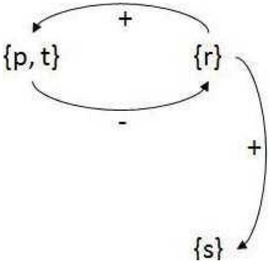
Admissible Component Sequence

Sequence $C_1, \ldots, C_n$ is admissible if $i < j$ whenever $C_i \prec_+ C_j$.

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

Procedure *InstantiateRule*(r: rule; S: SetOfAtoms; ΔS: SetOfAtoms;
var $\mathcal{NS}$: 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 $\mathcal{NS}$ 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 negative body literal $\mathrm{not}\ Q$ is in $B(r)$ and $Q \notin S$, then delete $\mathrm{not}\ Q$ from $B(r)$.
- If a negative body literal $\mathrm{not}\ Q$ is in $B(r)$ and $Q \in S$, 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

Thanks to Francesco Ricca for a preliminary version of these slides

EXERCISES

Exercise (VII) and (VIII)

Consider the solution(s) you have devised for exercise (V) and/or (VI), try to figure out what specific simplifications are made during grounding, by also checking what is the output of doing grounding with gringo.

What you are requested to do

What you are requested to do is:

1. sending by email at mmaratea@dbai.tuwien.ac.at before 24:00 (resp. 12:00) of the day before (resp. same day) if lecture is done in the morning (resp. in the afternoon), solutions related to exercise (V) and/or (VI),
2. “check” your solution using a grounder,
3. coming to the black-board! (if time/space allow :)

This lecture is dedicated to . . .