Processing of Declarative Knowledge
–Programming in ASP–

Francesco Ricca

Computational Intelligence Curriculum
Institute of Information Systems

ASP Basics

ASP:

Datalog $\leftarrow$ done!

+ Default negation $\leftarrow$ done!
+ Disjunction $\leftarrow$ done!
+ Integrity Constraints $\leftarrow$ done!
+ Weak Constraints $\leftarrow$ done!
+ Aggregate atoms $\leftarrow$ done!

How to program in ASP?

- Programming methodology

ASP Basics

ASP:

Datalog $\leftarrow$ done!

+ Default negation $\leftarrow$ done!
+ Disjunction $\leftarrow$ done!
+ Integrity Constraints $\leftarrow$ done!
+ Weak Constraints $\leftarrow$ done!
+ Aggregate atoms $\leftarrow$ done!

How to program in ASP?

- Programming methodology

Problem solving in ASP

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

Programming Steps:

1. Model your domain

   → Single out input/output predicates

2. Write a logic program modeling your problem

   → Use predicates representing relevant entities

   → **Hint:** take input data separated from derived ones

Problem solving in ASP

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

Programming Steps:

1. Model your domain

   → Single out input/output predicates

2. Write a logic program modeling your problem

   → Use predicates representing relevant entities

   → **Hint:** take input data separated from derived ones

Problem solving in ASP

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

Programming Steps:

1. Model your domain

   → Single out input/output predicates

2. Write a logic program modeling your problem

   → Use predicates representing relevant entities

   → **Hint:** take input data separated from derived ones

Problem solving in ASP

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

Programming Steps:

1. Model your domain

   → Single out input/output predicates

2. Write a logic program modeling your problem

   → Use predicates representing relevant entities

   → **Hint:** take input data separated from derived ones

Direct Encodings when...

Use a "Direct" Encoding with Datalog rules for

- Polynomial Problems, Deductive Database, etc.

Example (Reachability)

Problem: Find all nodes reachable from the others.
Input: *edge*(_, _).

```
% X is reachable from Y if an edge (X,Y) exists
reachable(X, Y) :- edge(X, Y).

% Reachability is transitive
reachable(X, Y) :- reachable(X, Z), edge(Z, Y).
```

Unfeasible for search problems from NP and beyond: need for a programming methodology

Programming Methodology

Guess & Check & Optimize (GCO)

1. Guess solutions $\rightarrow$ using disjunctive rules
2. Check admissible ones $\rightarrow$ using strong constraints

Optimization problem?

3. Specify Preference criteria $\rightarrow$ using weak constraints

In other words...

1. disjunctive rules $\rightarrow$ generate candidate solutions
2. constraints $\rightarrow$ test solutions discarding unwanted ones
3. weak constraints $\rightarrow$ single out optimal solutions

Programming Methodology

Guess & Check & Optimize (GCO)

1. Guess solutions $\rightarrow$ using disjunctive rules
2. Check admissible ones $\rightarrow$ using strong constraints

Optimization problem?

3. Specify Preference criteria $\rightarrow$ using weak constraints

In other words...

1. disjunctive rules $\rightarrow$ generate candidate solutions
2. constraints $\rightarrow$ test solutions discarding unwanted ones
3. weak constraints $\rightarrow$ single out optimal solutions

Programming Methodology

Guess & Check & Optimize (GCO)

1. Guess solutions $\rightarrow$ using disjunctive rules
2. Check admissible ones $\rightarrow$ using strong constraints

Optimization problem?

3. Specify Preference criteria $\rightarrow$ using weak constraints

In other words...

1. disjunctive rules $\rightarrow$ generate candidate solutions
2. constraints $\rightarrow$ test solutions discarding unwanted ones
3. weak constraints $\rightarrow$ single out optimal solutions

Guess and Check (Example 1)

Example (Group Assignments)

Problem: We want to partition a set of persons in two groups, while avoiding that father and children belong to the same group.
Input: persons and fathers are represented by *person*(_) and *father*(_, _).

% a disjunctive rule to "guess" all the possible assignments

$$group(P, 1) \mid group(P, 2) \text{ :- } person(P).$$

% a constraint to discard unwanted solutions
% i.e., father and children cannot belong to the same group

$$\text{:- } group(P1, G), group(P2, G), father(P1, P2).$$

...so how does it work really?

Guess and Check (Example 1)

Example (Group Assignments)

Problem: We want to partition a set of persons in two groups, while avoiding that father and children belong to the same group.
Input: persons and fathers are represented by *person*(_) and *father*(_, _).

% a disjunctive rule to "guess" all the possible assignments

$$group(P, 1) \mid group(P, 2) \text{ :- } person(P).$$

% a constraint to discard unwanted solutions
% i.e., father and children cannot belong to the same group

$$\text{:- } group(P1, G), group(P2, G), father(P1, P2).$$

...so how does it work really?

Guess and Check (Example 1)

Example (Group Assignments)

Problem: We want to partition a set of persons in two groups, while avoiding that father and children belong to the same group.
Input: persons and fathers are represented by *person*(_) and *father*(_, _).

% a disjunctive rule to "guess" all the possible assignments

$$group(P, 1) \mid group(P, 2) \text{ :- } person(P).$$

% a constraint to discard unwanted solutions
% i.e., father and children cannot belong to the same group

$$\text{:- } group(P1, G), group(P2, G), father(P1, P2).$$

...so how does it work really?

Guessing part explained

Consider: $group(P, 1) \mid group(P, 2) \text{ :- } person(P).$

If the input is: $person(john).\ person(joe).\ father(john, joe).$

Then, the answer set of this single-rule program are:

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 2)\}$

i.e., one a.s. for each assignment of 3 pers. to 2 groups!

Guessing part explained

Consider: $group(P, 1) \mid group(P, 2) \text{ :- } person(P).$

If the input is: $person(john).\ \ person(joe).\ \ father(john, joe).$

Then, the answer set of this single-rule program are:

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 2)\}$

i.e., one a.s. for each assignment of 3 pers. to 2 groups!

Guessing part explained

Consider: $group(P, 1) \mid group(P, 2) :\!- person(P).$

If the input is: $person(john).\ \ person(joe).\ \ father(john, joe).$

Then, the answer set of this single-rule program are:

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 2)\}$

i.e., one a.s. for each assignment of 3 pers. to 2 groups!

Guessing part explained

Consider: $group(P, 1) \mid group(P, 2) :\!- person(P).$

If the input is: $person(john).\ \ person(joe).\ \ father(john, joe).$

Then, the answer set of this single-rule program are:

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 2)\}$

i.e., one a.s. for each assignment of 3 pers. to 2 groups!

Checking part explained

Consider: $group(P, 1) \mid group(P, 2) \text{ :- } person(P).$

Now add: $\text{:- } group(P1, G), group(P2, G), father(P1, P2).$

If the input is: $person(john).\ \ person(joe).\ \ father(john, joe).$

The constraint "discards" two non admissible answers:

~~$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 1)\}$~~

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$

$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$

~~$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 2)\}$~~

Checking part explained

Consider: $group(P, 1) \mid group(P, 2) \text{ :- } person(P).$

Now add: $\text{:- } group(P1, G), group(P2, G), father(P1, P2).$

If the input is: $person(john).\ \ person(joe).\ \ father(john, joe).$

The constraint “discards” two non admissible answers:

~~$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 1)\}$~~

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$

$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$

~~$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 2)\}$~~

Guess & Check explained

Consider: $group(P, 1) \mid group(P, 2) \text{ :- } person(P).$
$\text{ :- } group(P1, G), group(P2, G), father(P1, P2).$

If the input is: $person(john).\ \ person(joe).\ \ father(john, joe).$

The answer sets are:

$\{person(john), person(joe), father(john, joe), group(john, 1), group(joe, 2)\}$
$\{person(john), person(joe), father(john, joe), group(john, 2), group(joe, 1)\}$

G&C = Define search space + specify desired solutions

Guess and Check (Example 2)

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)  col(X, red) | col(X, yellow) | col(X, green) :- node(X).

% discard colorings where adjacent nodes have the same color
(c)  :- edge(X, Y), col(X, C), col(Y, C).

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

Guess and Check (Example 2)

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)  col(X, red) | col(X, yellow) | col(X, green) :- node(X).

% discard colorings where adjacent nodes have the same color
(c)  :- edge(X, Y), col(X, C), col(Y, C).

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

Guess and Check (Example 2)

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

% discard colorings where adjacent nodes have the same color

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

% NB: answer sets are subset minimal $\rightarrow$ only one color per node

Guess and Check (Example 2)

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

% discard colorings where adjacent nodes have the same color

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

% NB: answer sets are subset minimal $\rightarrow$ only one color per node

Guess and Check (Example 3)

Example (Hamiltonian Path)

Problem: Find a path in a Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _), and *start*(_).

```
% Guess a path
inPath(X, Y) | outPath(X, Y) :- edge(X, Y).                    | Guess

% A node can be reached only once                               |
 :- inPath(X, Y), inPath(X, Y1), Y <> Y1.                       | Check
 :- inPath(X, Y), inPath(X1, Y), X <> X1.                       |
% All nodes must be reached                                     |
 :- node(X), not reached(X).                                    |
% The path is not cyclic                                        |
 :- inPath(X, Y), start(Y).                                     |

reached(X) :- reached(Y), inPath(Y, X).                         | Aux. Rules
reached(X) :- start(X).                                         |
```

Guess and Check (Example 3)

Example (Hamiltonian Path)

Problem: Find a path in a Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _), and *start*(_).

```
% Guess a path                                                    | Guess
inPath(X, Y) | outPath(X, Y) :- edge(X, Y).

% A node can be reached only once                                 |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                          | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                          |
% All nodes must be reached                                       |
:- node(X), not reached(X).                                       |
% The path is not cyclic                                          |
:- inPath(X, Y), start(Y).

reached(X) :- reached(Y), inPath(Y, X).                           | Aux. Rules
reached(X) :- start(X).                                           |
```

Guess and Check (Example 3)

Example (Hamiltonian Path)

Problem: Find a path in a Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _), and *start*(_).

```
% Guess a path                                                    | Guess
inPath(X, Y) | outPath(X, Y) :- edge(X, Y).

% A node can be reached only once                                 |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                          | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                          |
% All nodes must be reached                                       |
:- node(X), not reached(X).                                       |
% The path is not cyclic                                          |
:- inPath(X, Y), start(Y).

reached(X) :- reached(Y), inPath(Y, X).                           | Aux. Rules
reached(X) :- start(X).                                           |
```

Guess and Check (Example 3)

Example (Hamiltonian Path)

Problem: Find a path in a Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _), and *start*(_).

```
% Guess a path                                                    | Guess
inPath(X, Y) | outPath(X, Y) :- edge(X, Y).

% A node can be reached only once                                 |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                          | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                          |
% All nodes must be reached                                       |
:- node(X), not reached(X).                                       |
% The path is not cyclic                                          |
:- inPath(X, Y), start(Y).

reached(X) :- reached(Y), inPath(Y, X).                           | Aux. Rules
reached(X) :- start(X).                                           |
```

Guess and Check (Example 3)

Example (Hamiltonian Path)

Problem: Find a path in a Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _), and *start*(_).

```
% Guess a path                                                    | Guess
inPath(X, Y) | outPath(X, Y) :- edge(X, Y).

% A node can be reached only once                                 |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                          | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                          |
% All nodes must be reached                                       |
:- node(X), not reached(X).                                       |
% The path is not cyclic                                          |
:- inPath(X, Y), start(Y).

reached(X) :- reached(Y), inPath(Y, X).                           | Aux. Rules
reached(X) :- start(X).                                           |
```

Guess, Check and Optimize (Example 4)

Example (Traveling Salesman Person)

Problem: Find a path of minimum length in a Weighted Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _, _), and *start*(_).

```
% Guess a path
inPath(X, Y) | outPath(X, Y) :- edge(X, Y, _).                    | Guess

% Ensure that it is Hamiltonian (as before)                       |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                          | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                          |
:- node(X), not reached(X). :- inPath(X, Y), start(Y).            |
reached(X) :- reached(Y), inPath(Y, X).                           | Aux. Rules
reached(X) :- start(X).                                           |

% Minimize the sum of distances
:~ inPath(X, Y), edge(X, Y, C). [C@0, X, Y, C]                    | Optimize
```

Guess, Check and Optimize (Example 4)

Example (Traveling Salesman Person)

Problem: Find a path of minimum length in a Weighted Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _, _), and *start*(_).

```
% Guess a path                                                          | Guess
inPath(X, Y) | outPath(X, Y) :- edge(X, Y, _).
% Ensure that it is Hamiltonian (as before)                            |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                               | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                               |
:- node(X), not reached(X). :- inPath(X, Y), start(Y).                 |
reached(X) :- reached(Y), inPath(Y, X).                                | Aux. Rules
reached(X) :- start(X).                                                |

% Minimize the sum of distances                                        |
:~ inPath(X, Y), edge(X, Y, C). [C@0, X, Y, C]                         | Optimize
```

Guess, Check and Optimize (Example 4)

Example (Traveling Salesman Person)

Problem: Find a path of minimum length in a Weighted Graph beginning at the starting node which contains all nodes of the graph.
Input: *node*(_) and *edge*(_, _, _), and *start*(_).

```
% Guess a path                                                          | Guess
inPath(X, Y) | outPath(X, Y) :- edge(X, Y, _).
% Ensure that it is Hamiltonian (as before)                            |
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.                               | Check
:- inPath(X, Y), inPath(X1, Y), X <> X1.                               |
:- node(X), not reached(X). :- inPath(X, Y), start(Y).                 |
reached(X) :- reached(Y), inPath(Y, X).                                | Aux. Rules
reached(X) :- start(X).                                                |

% Minimize the sum of distances
:~ inPath(X, Y), edge(X, Y, C). [C@0, X, Y, C]                         | Optimize
```

Guess and Check (Example 5)

Example (Strategic Companies)

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

Exploits minimality... but Checking and guessing interfere!
→ non-HCF encoding → higher computational complexity?!
→ Indeed, checking Strategic Companies is Σ_2^P-complete

Guess and Check (Example 5)

Example (Strategic Companies)

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

Exploits minimality... but Checking and guessing interfere!
→ non-HCF encoding → higher computational complexity?!
→ Indeed, checking Strategic Companies is Σ_2^P-complete

Guess and Check (Example 5)

Example (Strategic Companies)

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

Exploits minimality... but Checking and guessing interfere!
$\rightarrow$ non-HCF encoding $\rightarrow$ higher computational complexity?!
$\rightarrow$ Indeed, checking Strategic Companies is Σ_2^P-complete

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.*