# Efficient ASP Techniques for Solving Hard Problems

Carmine Dodaro
DIBRIS, University of Genoa

Arcavacata di Rende, 2-5-6-7 February 2018

**Answer Set Programming (ASP)**

- Declarative programming paradigm
- Based on the stable model (answer set) semantics

**Idea:**

1. Logic programs represent computational problems
2. Answer sets correspond to solutions
3. Use a solver to find solutions

## Applications in several fields

# ASP Syntax

**Rule: ($r$)** $\underbrace{a_1 | \ldots | a_n}_{head} \text{ :- } \underbrace{b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m.}_{body}$

# ASP Syntax

**Rule: ($r$)** $\underbrace{a_1 | \ldots | a_n}_{head} \text{ :- } \underbrace{b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m.}_{body}$

**Atoms, and Literals:** $a_i$ , $b_i$, $not\ b_i$
**Head of $r$:** $H(r) = \{a_1, \ldots, a_n\}$
**Body of $r$:** $B(r) = B^+(r) \cup B^-(r)$
**Positive Body:** $B^+(r) = \{b_1, \ldots, b_k\}$
**Negative Body:** $B^-(r) = \{not\ b_{k+1}, \ldots, not\ b_m.\}$

**Variables:** Begin with uppercase letter
**Safety:** Variables must occur in the positive body

**Fact**: Rule with empty body
**Constraint**: Rule with empty head

# Examples of Rules

## Example (Disjunction, Negation, Constraints)

% Disjunctive knowledge: “A parent $P$ is either a father
% or a mother”
$mother(P, S) \mid father(P, S) \text{ :- } parent(P, S).$

% Default Negation: “Check if an undirected graph”
% is not connected”
$disconnected \text{ :- } node(X), node(Y), not\ reachable(X, Y).$

% Constraints: “Admit only connected graphs.”
$\text{ :- } disconnected.$

% Facts: Often used to define the input
$node(1) \text{ :- } .$

# Examples of Rules

## Example (Disjunction, Negation, Constraints)

% Disjunctive knowledge: “A parent $P$ is either a father
% or a mother”
$mother(P, S) \mid father(P, S) \text{ :- } parent(P, S).$

% Default Negation: “Check if an undirected graph”
% is not connected”
$disconnected \text{ :- } node(X), node(Y), not\ reachable(X, Y).$

% Constraints: “Admit only connected graphs.”
$\text{ :- } disconnected.$

% Facts: Often used to define the input
$node(1) \text{ :- } .$

In case of facts symbol :- is omitted.

# ASP Computation

- Grounder
  - Eliminates variables
  - Produces an equivalent propositional theory
- Solver
  - Works on propositional theory
  - Produces answer sets

# Example Grounding

**Input Program $\mathcal{P}$**

$c(1).$
$c(2).$
$a(X) \mid b(X) :- c(X).$

# Example Grounding

## Input Program $\mathcal{P}$

$c(1).$
$c(2).$
$a(X) \mid b(X) :- c(X).$

## Ground Program $\Pi$

$c(1).$
$c(2).$
$a(1) \mid b(1) :- c(1).$
$a(2) \mid b(2) :- c(2).$

# Arithmetic Expressions and Builtins

**Arithmetic and comparison operators**

- $<, >, <=, >=, =$
- $+, -, *, /$

**Example (Fibonacci numbers)**

$fib(0, 0).$
$fib(1, 1).$
$fib(N + 2, Y1 + Y2) :- fib(N, Y1), fib(N + 1, Y2), N < 19.$

# Arithmetic Expressions and Builtins

**Arithmetic and comparison operators**

- $<, >, <=, >=, =$
- $+, -, *, /$

**Example (Fibonacci numbers)**

$fib(0, 0).$
$fib(1, 1).$
$fib(N + 2, Y1 + Y2) :- fib(N, Y1), fib(N + 1, Y2), N < 19.$
N < 19 guarantees the termination!

# Informal Semantics (1)

**Disjunctive Rule:**

$$a_1 \mid \ldots \mid a_n \text{ :- } b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m.$$

**Informal Semantics:**

"If all $b_1, \ldots, b_k$ are true and all $b_{k+1}, \ldots, b_m$ are not true, then at least one among $a_1, \ldots, a_n$ is true".

# Informal Semantics (1)

**Disjunctive Rule:**

$$a_1 \mid \ldots \mid a_n \text{ :- } b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m.$$

**Informal Semantics:**

"If all $b_1, \ldots, b_k$ are true and all $b_{k+1}, \ldots, b_m$ are not true, then at least one among $a_1, \ldots, a_n$ is true".

## Example

$\mathit{isInterestedinASP}(\mathit{john}) \mid \mathit{isCurious}(\mathit{john})$ :- $\mathit{attendsASP}(\mathit{john})$.

$\mathit{attendsASP}(\mathit{john})$.

Three models encoding three plausible scenarios:

- $M_1$: $\{\mathit{isInterestedinASP}(\mathit{john}), \mathit{attendsASP}(\mathit{john})\}$
- $M_2$: $\{\mathit{isCurious}(\mathit{john}), \mathit{attendsASP}(\mathit{john})\}$
- $M_3$: $\{\mathit{isCurious}(\mathit{john}), \mathit{isInterestedinASP}(\mathit{john}), \mathit{attendsASP}(\mathit{john})\}$

# Informal Semantics (1)

**Disjunctive Rule:**

$$a_1 \mid \ldots \mid a_n \text{ :- } b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m.$$

**Informal Semantics:**

"If all $b_1, \ldots, b_k$ are true and all $b_{k+1}, \ldots, b_m$ are not true, then at least one among $a_1, \ldots, a_n$ is true".

## Example

$\mathit{isInterestedinASP}(\mathit{john}) \mid \mathit{isCurious}(\mathit{john})$ :- $\mathit{attendsASP}(\mathit{john})$.

$\mathit{attendsASP}(\mathit{john})$.

Three models encoding three plausible scenarios:

- $M_1$: $\{\mathit{isInterestedinASP}(\mathit{john}), \mathit{attendsASP}(\mathit{john})\}$
- $M_2$: $\{\mathit{isCurious}(\mathit{john}), \mathit{attendsASP}(\mathit{john})\}$
- $M_3$: $\{\mathit{isCurious}(\mathit{john}), \mathit{isInterestedinASP}(\mathit{john}), \mathit{attendsASP}(\mathit{john})\}$

$M_1$ and $M_2$ are answer sets, while $M_3$ is not an answer set!

# Informal Semantics (2)

**Constraint:**

$$:- \; b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m.$$

**Informal Semantics:**

"It is not possible that all $b_1, \ldots, b_k$ are true, and all $b_{k+1}, \ldots, b_m$ are false".

# Informal Semantics (2)

**Constraint:**

$$:\!- \; b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m.$$

**Informal Semantics:**

"It is not possible that all $b_1, \ldots, b_k$ are true, and all $b_{k+1}, \ldots, b_m$ are false".

## Example

$\mathit{isInterestedinASP}(\mathit{john}) \mid \mathit{isCurious}(\mathit{john}) :\!- \mathit{attendsASP}(\mathit{john}).$

$:\!- \mathit{hatesASP}(\mathit{john}), \mathit{isInterestedinASP}(\mathit{john}).$

$\mathit{attendsASP}(\mathit{john}). \quad \mathit{hatesASP}(\mathit{john}).$

Only one plausible scenario:

- ~~$M_1$:{$\mathit{isInterestedinASP}(\mathit{john}), \mathit{attendsASP}(\mathit{john}).$}~~
- $M_2$:{$\mathit{isCurious}(\mathit{john}), \mathit{attendsASP}(\mathit{john}), \mathit{hatesASP}(\mathit{john}).$}

# Informal Semantics (3)

**Semantics of disjunction is:**

- Minimal

  $a \mid b \mid c. \Rightarrow \{a\}, \{b\}, \{c\}$

# Informal Semantics (3)

**Semantics of disjunction is:**

- Minimal

  $a \mid b \mid c. \Rightarrow \{a\}, \{b\}, \{c\}$

- Actually subset minimal

  $a \mid b.$

  $a \mid c. \Rightarrow \{a\}, \{b, c\}$

# Informal Semantics (3)

**Semantics of disjunction is:**

- Minimal

  $a \mid b \mid c. \Rightarrow \{a\}, \{b\}, \{c\}$

- Actually subset minimal

  $a \mid b.$

  $a \mid c. \Rightarrow \{a\}, \{b, c\}$

- ...but *not exclusive*

  $a \mid b.$

  $a \mid c.$

  $b \mid c. \Rightarrow \{a, b\}, \{a, c\}, \{b, c\}$

# Informal Semantics (4)

- Disjunctive rules “generate models”

  $a \mid b.$

  $a \mid c.$

  $b \mid c.$

  $\Rightarrow \{a, b\}, \{a, c\}, \{b, c\}$

- Integrity constraints “discard” unwanted models:

  % Add:

  $:\!- a, not\ \ b.$

  $\Rightarrow \{a, b\}, \{b, c\}$

# Informal Semantics (5)

## Ground Program Π

$c(1).$
$c(2).$
$a(1) \mid b(1) :- c(1).$
$a(2) \mid b(2) :- c(2).$

## Answer sets

# Informal Semantics (5)

## Ground Program Π

$c(1).$
$c(2).$
$a(1) \mid b(1) :- c(1).$
$a(2) \mid b(2) :- c(2).$

## Answer sets

$\{a(1), a(2), c(1), c(2)\}$

# Informal Semantics (5)

## Ground Program Π

$c(1).$
$c(2).$
$a(1) \mid b(1) :- c(1).$
$a(2) \mid b(2) :- c(2).$

## Answer sets

$\{a(1), a(2), c(1), c(2)\}$
$\{a(1), b(2), c(1), c(2)\}$

# Informal Semantics (5)

## Ground Program Π

$c(1).$
$c(2).$
$a(1) \mid b(1) :- c(1).$
$a(2) \mid b(2) :- c(2).$

## Answer sets

$\{a(1), a(2), c(1), c(2)\}$
$\{a(1), b(2), c(1), c(2)\}$
$\{b(1), a(2), c(1), c(2)\}$

# Informal Semantics (5)

## Ground Program Π

$c(1).$
$c(2).$
$a(1) \mid b(1) :- c(1).$
$a(2) \mid b(2) :- c(2).$

## Answer sets

$\{a(1), a(2), c(1), c(2)\}$
$\{a(1), b(2), c(1), c(2)\}$
$\{b(1), a(2), c(1), c(2)\}$
$\{b(1), b(2), c(1), c(2)\}$

# Reduct of a Program

- The formal semantics is based on the concept of reduct
- A model $M$ is an answer set (stable model) of a program $\Pi$ if $M$ is a subset minimal model of $\Pi^M$, which is the reduct of $\Pi$ w.r.t. $M$.

**Program $\Pi$**

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow not\ d$
$d \leftarrow not\ c$

**Model $M_1$**

$a, b, c$

**Model $M_2$**

$a, b, d$

# Reduct of a Program

- The formal semantics is based on the concept of reduct
- A model $M$ is an answer set (stable model) of a program $\Pi$ if $M$ is a subset minimal model of $\Pi^M$, which is the reduct of $\Pi$ w.r.t. $M$.

**Program $\Pi$**

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow not\ d$
$d \leftarrow not\ c$

**Model $M_1$**

$a, b, c$

Reduct $\Pi^{M_1}$

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow \sout{not\ d}$
$\sout{d \leftarrow not\ c}$

**Model $M_2$**

$a, b, d$

# Reduct of a Program

- The formal semantics is based on the concept of reduct
- A model $M$ is an answer set (stable model) of a program $\Pi$ if $M$ is a subset minimal model of $\Pi^M$, which is the reduct of $\Pi$ w.r.t. $M$.

**Program $\Pi$**

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow \mathit{not}\ d$
$d \leftarrow \mathit{not}\ c$

**Model $M_1$**

$a, b, c$

Reduct $\Pi^{M_1}$

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow \sout{\mathit{not}\ d}$
$\sout{d \leftarrow \mathit{not}\ c}$

✓ $M_1$: an answer set

**Model $M_2$**

$a, b, d$

# Reduct of a Program

- The formal semantics is based on the concept of reduct
- A model $M$ is an answer set (stable model) of a program $\Pi$ if $M$ is a subset minimal model of $\Pi^M$, which is the reduct of $\Pi$ w.r.t. $M$.

**Program $\Pi$**

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow not\ d$
$d \leftarrow not\ c$

**Model $M_1$**

$a, b, c$

Reduct $\Pi^{M_1}$

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow \sout{not\ d}$
$\sout{d \leftarrow not\ c}$

✓ $M_1$: an answer set

**Model $M_2$**

$a, b, d$

Reduct $\Pi^{M_2}$

$\sout{a \mid b \leftarrow c}$
$a \leftarrow b$
$b \leftarrow a$
$\sout{c \leftarrow not\ d}$
$d \leftarrow \sout{not\ c}$

# Reduct of a Program

- The formal semantics is based on the concept of reduct
- A model $M$ is an answer set (stable model) of a program $\Pi$ if $M$ is a subset minimal model of $\Pi^M$, which is the reduct of $\Pi$ w.r.t. $M$.

**Program $\Pi$**

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow not\ d$
$d \leftarrow not\ c$

**Model $M_1$**

$a, b, c$

Reduct $\Pi^{M_1}$

$a \mid b \leftarrow c$
$a \leftarrow b$
$b \leftarrow a$
$c \leftarrow \sout{not\ d}$
$\sout{d \leftarrow not\ c}$

✓ $M_1$: an answer set

**Model $M_2$**

$a, b, d$

Reduct $\Pi^{M_2}$

$\sout{a \mid b \leftarrow c}$
$a \leftarrow b$
$b \leftarrow a$
$\sout{c \leftarrow not\ d}$
$d \leftarrow \sout{not\ c}$

✗ $M_2$: not an answer set

# Complex Constructs

**Extension of the plain language:**

- Choice rules
- Aggregates
- Weak constraints

# Choice Rules (1)

**A choice rule**

- Allows to perform a "free" choice on some atoms
- The choice rule $\{a\}.$ can be viewed as a shortcut for $a \mid na$ where $na$ is a fresh symbol which does not appear in the answer sets

**Program Π**

$c(1).$
$c(2).$
$\{a(X)\} :- c(X).$
$\{b(X)\} :- c(X).$

**Answer sets**

{c(1), c(2)}
{c(1), c(2), b(1)}
{c(1), c(2), a(2), b(1)}
{c(1), c(2), a(2)}
{c(1), c(2), a(1), a(2)}
{c(1), c(2), a(1), a(2), b(1)}
{c(1), c(2), a(1), b(1)}
{c(1), c(2), a(1)}
{c(1), c(2), b(2), a(1)}
{c(1), c(2), b(2), a(1), b(1)}
{c(1), c(2), b(2), b(1)}
{c(1), c(2), b(2)}
{c(1), c(2), b(2), a(2)}
{c(1), c(2), b(2), a(2), b(1)}
{c(1), c(2), b(2), a(2), a(1), b(1)}
{c(1), c(2), b(2), a(2), a(1)}

# Choice Rules (2)

## Program $\Pi_1$

$c(1).$
$c(2).$
$\{a(X) : c(X)\}.$
$\{b(X) : c(X)\}.$

## Program $\Pi_2$

$c(1).$
$c(2).$
$\{a(X) : c(X); b(X) : c(X)\}.$

## Ground instantiation of $\Pi_1$

$c(1).$
$c(2).$
$\{a(1); a(2)\}.$
$\{b(1); b(2)\}.$

## Ground instantiation of $\Pi_2$

$c(1).$
$c(2).$
$\{a(1); a(2); b(1); b(2)\}.$

# Choice Rules (3)

## Program $\Pi_1$

$c(1).$
$c(2).$
$1 \leq \{a(X) : c(X)\} \leq 1.$
$1 \leq \{b(X) : c(X)\} \leq 1.$

## Program $\Pi_2$

$c(1).$
$c(2).$
$1 \leq \{a(X) : c(X); b(X) : c(X)\} \leq 1.$

## Answer sets of $\Pi_1$

$\{c(1), c(2), b(1), a(1)\}$
$\{c(1), c(2), b(1), a(2)\}$
$\{c(1), c(2), b(2), a(1)\}$
$\{c(1), c(2), b(2), a(2)\}$

## Answer sets of $\Pi_2$

$\{c(1), c(2), a(1)\}$
$\{c(1), c(2), a(2)\}$
$\{c(1), c(2), b(1)\}$
$\{c(1), c(2), b(2)\}$

## Program $\Pi_2$

$c(1).$ $\quad$ $c(2).$

$1 \leq \{a(X) : c(X); b(X) : c(X)\} \leq 1.$

## Program $\Pi_2'$

$c(1).$ $\quad$ $c(2).$

$\{a(X) : c(X); b(X) : c(X)\}.$

$:- \#count\{X, a : a(X), c(X); X, b : b(X), c(X)\} \neq 1.$

## Grounding of $\Pi_2'$

$c(1).$ $\quad$ $c(2).$

$\{a(1); b(1); a(2); b(2)\}.$

$:- \#count\{1, a : a(1), 1, b : b(1), 2, a : a(2), 2, b : b(2)\} \neq 1.$

# Aggregates

**Aggregates**

- Concise modeling of properties over sets of data
- Type of aggregates are #count, #sum, #min, #max

# Aggregates: Example

## Choice of products

$product(pizza, 7).\ product(pasta, 1).\ product(hamburger, 8).$
$product(water, 1).\ budget(8).$
$\{buy(X) : product(X, Price)\}.$
$:- budget(B), \#sum\{Price, X : buy(X), product(X, Price)\} > B.$

## Answer sets (showing only buy)

$\{\}$
$\{buy(hamburger)\}$
$\{buy(pasta)\}$
$\{buy(water)\}$
$\{buy(pizza)\}$
$\{buy(pizza), buy(pasta)\}$
$\{buy(water), buy(pizza)\}$
$\{buy(water), buy(pasta)\}$

# Weak Constraints (1)

**Weak constraints**

- Used to model optimization problems
- The idea is to associate a cost to each answer set
- Answer sets with the lowest cost are *optimum*

# Weak Constraints (2)

## Program Π

$c(1).$
$c(2).$
$1 \leq \{a(X) : c(X)\} \leq 1.$
$1 \leq \{b(X) : c(X)\} \leq 1.$
$:\sim a(X), b(X). \quad [X@1, X]$

## Answer sets

$\{c(1), c(2), b(1), a(2)\} \rightarrow COST = 0$
$\{c(1), c(2), b(2), a(1)\} \rightarrow COST = 0$
$\{c(1), c(2), b(1), a(1)\} \rightarrow COST = 1$
$\{c(1), c(2), b(2), a(2)\} \rightarrow COST = 2$

# Weak Constraints: Example

## Choice of products

$product(pizza, 7).\quad product(pasta, 1).\quad budget(8).$
$product(hamburger, 8).\qquad product(water, 1).$
$\{buy(X) : product(X, Price)\}.$
$pay(S) :- S = \#sum\{Price, X : buy(X),\ product(X, Price)\}.$
$:- \quad budget(B), pay(S), S > B.$
$:\sim \quad budget(B), pay(S), B > S.\qquad [B - S@1, S]$

## Answer sets (showing only buy)

$\{buy(hamburger)\} \rightarrow COST = 0$
$\{buy(pizza), buy(pasta)\} \rightarrow COST = 0$
$\{buy(water), buy(pizza)\} \rightarrow COST = 0$
$\{buy(pizza)\} \rightarrow COST = 1$
$\{buy(water), buy(pasta)\} \rightarrow COST = 6$
$\{buy(pasta)\} \rightarrow COST = 7$
$\{buy(water)\} \rightarrow COST = 7$
$\{\} \rightarrow COST = 8$

# Problem solving in ASP

**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

# Problem solving in ASP

**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

**Programming Steps:**

# Problem solving in ASP

**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

**Programming Steps:**

1. Model your domain

   $\rightarrow$ Single out input/output predicates

# 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, 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
$reachable(X, Y) :- reachable(X, Z), edge(Z, Y).$

# Direct Encodings when...

**Use a "Direct" Encoding with Datalog rules for**

- Polynomial Problems, 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).
```

The method in often unfeasible for search problems from NP and beyond: need for a programming methodology

**Guess & Check & Optimize (GCO)**

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

**Guess & Check & Optimize (GCO)**

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

*Optimization problem?*

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

**Guess & Check & Optimize (GCO)**

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

*Optimization problem?*

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

**In other words...**

1. disjunctive (choice) 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*(_, _).

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

# 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
$group(P, 1) \mid group(P, 2) :- person(P).$
% a constraint to discard unwanted solutions
% i.e., father and children cannot belong to the same group
$:- group(P1, G), group(P2, G), father(P1, P2).$

# 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
$group(P, 1) \mid group(P, 2) :- person(P).$
% a constraint to discard unwanted solutions
% i.e., father and children cannot belong to the same group
$:- group(P1, G), group(P2, G), father(P1, P2).$ **...so how does it work really?**

# Guessing part explained

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

# Guessing part explained

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

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

# 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:

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

# Checking part explained

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

Now add: $:- group(P1, G), group(P2, G), father(P1, P2).$

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

# Checking part explained

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

Now add: $:- group(P1, G), group(P2, G), father(P1, P2).$

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

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 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).
```

# 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).
```

# 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).
```

# 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 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).

% A node can be reached only once
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.
:- inPath(X, Y), inPath(X1, Y), X <> X1.
```

# 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).

% A node can be reached only once
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.
:- inPath(X, Y), inPath(X1, Y), X <> X1.
% All nodes must be reached
:- node(X), not reached(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
inPath(X, Y) | outPath(X, Y) :- edge(X, Y).

% A node can be reached only once
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.
:- 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).
```

# 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, 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, 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, _).
% Ensure that it is Hamiltonian (as before)
:- inPath(X, Y), inPath(X, Y1), Y <> Y1.
:- 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).
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                                                    | 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@1, X, Y, C]                   | Optimize
```

# ASP at Work: Systems

**Grounders**

- Gringo - https://potassco.org/
- i-DLV - https://github.com/DeMaCS-UNICAL/I-DLV/wiki

**Solvers**

- clasp - https://potassco.org/
- wasp - github.com/alviano/wasp

**Full systems**

- clingo (gringo + clasp) - https://potassco.org/clingo/
- DLV2 (i-DLV + wasp) - https://www.mat.unical.it/DLV2/

# Exercise 1

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

- Each table has n chairs.
- Each guest must be assigned exactly one table.
- People liking each other must sit at the same table.
- People disliking each other must not sit at the same table.

## Input

```
% table(number,capacity)
table(1,5).       table(2,5).
% guest(name)
guest(luca).      guest(francesco).      guest(john).      guest(mary).
guest(marco).     guest(andrea).      guest(giovanna).      guest(laura).
% like(X,Y), X likes Y
like(luca,francesco).      like(luca,mary).      like(john,mary).
like(andrea,giovanna).      like(giovanna,marco).      like(laura,luca).
% dislike(X,Y), X doesn't like Y
dislike(luca,marco).      dislike(luca,andrea).      dislike(laura,giovanna).
dislike(laura,andrea).      dislike(marco,luca).      dislike(marco,francesco).
```

# Exercise 2

Given an undirected graph $G = (V, E)$, a clique is a subset of the vertices all adjacent to each other. A clique $C$ is said to be maximal if for each other clique $C'$ in $G$, the number of nodes in $C$ are larger than or equal to the number of nodes in $C'$. Write an ASP encoding to find maximal cliques of an input graph.

**Input**

```
node(1).  node(2).  node(3).  node(4).  node(5).
edge(1,2).  edge(3,4).  edge(4,5).  edge(3,4).
```