Algorithms for Computational Logic

Answer Set Programming

IST, ULisboa

Outline

Definition

- ASP for short
- Declarative problem solving paradigm
- Similar to SAT and CSP: difference in modelling language and semantics involved
- Problems specified using logic programming like rules + convenient extensions
  - Inspiration in Prolog
  - More compact and readable problem descriptions
- Roots in knowledge representation, in particular non-monotonic reasoning

ASP Syntax Basics

"The head must be true whenever the body is true:"

```
a :- b_1, ..., b_n, not c_1, ..., not c_m.
```

ASP Syntax Basics

"The head must be true whenever the body is true:"

```
a :- b_1, ..., b_n, not c_1, ..., not c_m.
```

"Derive for any X satisfying P and Q that it satisfies O."

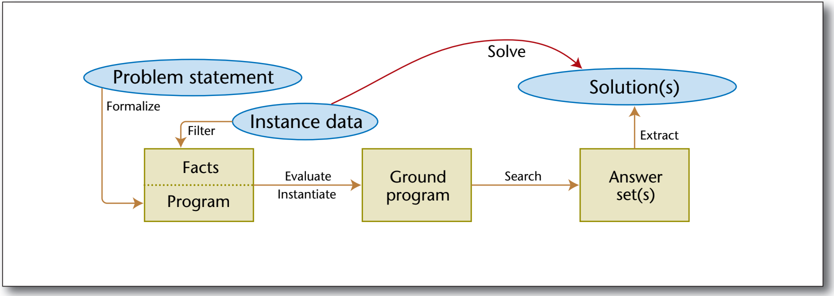
```
O(X) :- P(X), Q(X).
```

Bibliography

- Main:
  - "Answer Set Programming"
    AI Magazine, Volume 37, Number 3, Fall 2016

- Additional:
  - "Answer Set Solving in Practice" by Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub
    Synthesis Lectures on Artificial Intelligence and Machine Learning
    Morgan & Claypool Publishers 2012
  - "Answer Set Solving in Practice: tutorial slides"
    Torsten Schaub
    `http://www.cs.uni-potsdam.de/~torsten/Potassco/Slides/motivation.pdf`
  - "Efficient Solving Techniques for Answer Set Programming"
    Carmine Dodaro
    `https://blog.tewi.uni-klu.ac.at/wp-content/uploads/2016/04/slides.pdf`

ASP Conceptual Model



Outline

Prolog (I)

6 facts, 1 rule

```
p(1).  p(2).  p(3).
q(2).  q(3).  q(4).
r(X) :- p(X), q(X).
```

Query:

```
?- r(X).
X=2 ; X=3.
```

Answer Sets (I)

6 facts, 1 rule

```
p(1).  p(2).  p(3).
q(2).  q(3).  q(4).
r(X) :- p(X), q(X).
```

No need to query. The only answer set:

```
p(1).  p(2).  p(3).
q(2).  q(3).  q(4).
r(2).  r(3).
```

Prolog (II)

Negation as failure, symbol $\backslash+$

```
p(1).  p(2).  p(3).
q(2).  q(3).  q(4).
r(X) :- p(X), \+ q(X).
```

Query:

```
?- r(X).
X=1
```

Answer Sets (II)

Negation denoted by **not**

```
p(1).  p(2).  p(3).
q(2).  q(3).  q(4).
r(X) :- p(X), not q(X).
```

Answer set:

```
p(1).  p(2).  p(3).
q(2).  q(3).  q(4).
r(1).
```

Negation as failure is not declarative

```
p(1).  p(2).  p(3).
q(3) :- \+ r(3).
r(X) :- p(X), \+ q(X).
```

Prolog (III)

Negation as failure is not declarative

```
p(1).  p(2).  p(3).
q(3) :- \+ r(3).
r(X) :- p(X), \+ q(X).
```

```
?- q(X).
Error: out of local stack
```

Prolog (III)

Negation as failure is not declarative

```
p(1).  p(2).  p(3).
q(3) :- \+ r(3).
r(X) :- p(X), \+ q(X).
```

```
?- q(X).
Error: out of local stack
```

```
?- r(X).
Error: out of local stack
```

Answer sets (III)

```
p(1).  p(2).  p(3).
q(3) :- not r(3).
r(X) :- p(X), not q(X).
```

Answer sets (III)

```
p(1).  p(2).  p(3).
q(3) :- not r(3).
r(X) :- p(X), not q(X).
```

Answer set 1

```
p(1) p(2)  p(3)
r(1) r(2)
q(3)
```

Answer sets (III)

```
p(1).  p(2).  p(3).
q(3) :- not r(3).
r(X) :- p(X), not q(X).
```

Answer set 1

```
p(1) p(2)  p(3)
r(1) r(2)
q(3)
```

Answer set 2

```
p(1) p(2)  p(3)
r(1) r(2)
r(3)
```

Answer sets (III)

```
p(1).  p(2).  p(3).
q(3) :- not r(3).
r(X) :- p(X), not q(X).
```

Answer set 1

```
p(1) p(2)  p(3)
r(1) r(2)
q(3)
```

Answer set 2

```
p(1) p(2)  p(3)
r(1) r(2)
r(3)
```

Programs with several answer sets are “bad” Prolog programs

Non-Monotonic Reasoning

- Reasoning is non-monotonic, if adding more premises may reduce the set of inferable facts.
- Reasoners draw conclusions tentatively, reserving the right to retract them in the light of further information

Tweety Example

```
flies(X) :- bird(X), not abnormal(X).
bird(X) :- penguin(X).
bird(X) :- parrot(X).
abnormal(X) :- penguin(X).
penguin(tweety).
parrot(cracker).
```

Tweety Example

```
flies(X) :- bird(X), not abnormal(X).
bird(X) :- penguin(X).
bird(X) :- parrot(X).
abnormal(X) :- penguin(X).
penguin(tweety).
parrot(cracker).
```

Answer set:

```
penguin(tweety).
bird(tweety).
abnormal(tweety).
parrot(cracker).
bird(cracker).
flies(cracker).
```

Tweety Example

```
flies(X) :- bird(X), not abnormal(X).
bird(X) :- penguin(X).
bird(X) :- parrot(X).
abnormal(X) :- penguin(X).
penguin(tweety).
parrot(cracker).
```

Answer set:

```
penguin(tweety).
bird(tweety).
abnormal(tweety).
parrot(cracker).
bird(cracker).
flies(cracker).
```

Behaviour non-monotonic, adding the fact that a bird is abnormal, no longer enables deriving it flies.

Outline

How to compute answer sets?

- First, ground the program, i.e., substitute specific values for variables in the rules
- In positive programs - simpler case: no negation
- In programs with negation - consider a guess S - compute the reduct which has no negations
  - Intuition: rules of the program that “fire” assuming S the set of atoms that can be generated by the rules
  - If the answer set of the reduct (positive program) is exactly S then S is an answer set

Positive Programs (Without Negation)

```
#show invited/1.
relative(X, Y) :- relative(Y, X).
relative(X, Y) :- relative(X, Z), relative(Z, Y).

relative(X, Y) :- child(X, Y).
relative(X, Y) :- sibling(X, Y).

child(ana, bruno).
sibling(bruno, carlos).
child(ricardo, pedro).

invited(X) :- relative(X, ana).
```

Positive Programs (Without Negation)

```
#show invited/1.
relative(X, Y) :- relative(Y, X).
relative(X, Y) :- relative(X, Z), relative(Z, Y).

relative(X, Y) :- child(X, Y).
relative(X, Y) :- sibling(X, Y).

child(ana, bruno).
sibling(bruno, carlos).
child(ricardo, pedro).

invited(X) :- relative(X, ana).
```

```
invited(bruno).
invited(carlos).
invited(ana).
```

Same semantics as Prolog

Answer set: programs with negation - example

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(X) :- p(X), not q(X).
```

Grounding:

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1), not q(1).
r(2) :- p(2), not q(2).
r(3) :- p(3), not q(3).
r(4) :- p(4), not q(4).
```

Answer set: programs with negation - algorithm

- Goal: Decide whether a set S of ground atoms is an answer set

1. Form the *reduct* of the grounded program with respect to S
   - Consider a general rule
     A_0 `:-` $A_1, \cdots, A_m,$ `not` $A_{m+1}, \cdots,$ `not` A_n
   - If S does not contain atoms $A_{m+1}, \cdots, A_n$, drop the negated atoms `not` $A_{m+1}, \cdots,$ `not` A_n from the rule
   - All other rules with negations are dropped
2. Compute the answer set of the reduct (positive program)
3. If the answer set of the reduct is exactly S then S is an answer set of the program with negation

Answer set: programs with negation - example

- Program after grounding:

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1), not q(1).
r(2) :- p(2), not q(2).
r(3) :- p(3), not q(3).
r(4) :- p(4), not q(4).
```

- Answer set? {`p(1), p(2), p(3), q(2), q(3), q(4), r(1)`}
- Reduct:

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1).
```

- Conclusion: {`p(1), p(2), p(3), q(2), q(3), q(4), r(1)`} is an answer set

Answer set: programs with negation - another example

```
p(1).  p(2).  p(3).
q(3) :- not r(3).
r(X) :- p(X), not q(X).
```

Grounding? Answer sets?

Exercise: What are the answer sets?

```
a :- a.
```

Exercise: What are the answer sets?

```
a :- a.
```

1 Answer Set, which is empty ("Cannot derive a from a.")

```
....
```

Exercise: What are the answer sets?

```
a :- a.
```

1 Answer Set, which is empty ("Cannot derive a from a.")

```
....
```

```
a :- not a.
```

Exercise: What are the answer sets?

```
    a :- a.
```

1 Answer Set, which is empty ("Cannot derive a from a.")

```
....
```

```
    a :- not a.
```

No Answer Set (UNSAT) ("If a not derived, derive it, which breaks the premise of the derivation.")

Answer set: programs with negation - more examples

- Program: {`p :- p.  q :- not p.`}
- Check all possible answer sets...

AnswerSet?	Reduct(R)	AnswerSet(R)	Outcome
{ }	`p :- p.  q.`	{q}	NO
{p}	`p :- p.`	{ }	NO
{q}	`p :- p.  q.`	{q}	YES
{p, q}	`p :- p.`	{ }	NO

Outline

Extensions of the basic language

- Arithmetic
- Disjunctive rules
- Choice rules
- Constraints
- Classical negation

Definition of answer set has to be extended

Arithmetic

Rules may contain symbols for arithmetic operations and comparisons

Program

```
p(1). p(2).
q(1). q(2).
r(X+Y) :- p(X), q(Y), X<Y.
```

Answer set

```
p(1) p(2) q(1) q(2) r(3)
```

Disjunctive rules

The head of a rule may be a disjunction of several atoms (often separated by bars or semicolons), rather than a single atom

```
p(1) | p(2).
```

```
Answer: 1
p(1)
Answer: 2
p(2)
```

Choice Rules

Enclosing the list of atoms in the head in curly braces represents the "choice" construct: choose in all possible ways which atoms from the list will be included in the answer set

```
{ p(1) ; p(2) }.
```

```
Answer: 1

Answer: 2
p(1)
Answer: 3
p(2)
Answer: 4
p(1)  p(2)
```

Choice Rules (cont.)

One may specify bounds on the number of atoms that are included: the lower bound is shown to the left of the expression in braces, and the upper bound to the right

```
1 { p(1) ; p(2) }.
```

Answer sets: 2-4

```
{ p(1) ; p(2) } 1.
```

Answer sets: 1-3

Constraints

Disjunctive rule that has 0 disjuncts in the head, so that it starts with the symbol `:-`

```
{ p(1) ; p(2) }.
:- p(1), not p(2).
```

Answer sets: 1, 3 and 4

Eliminates the answer sets that satisfy the body of the constraint

Classical negation

The “classical negation” sign (−) should be distinguished from the negation as failure symbol (not)

- Example:
  answer set `p(a) p(b) -p(c) q(a) -q(c)`
  (whether b has property q we do not know)
- Closed world assumption: rule `-A :- not A`
- Frame default: rule `p(T+1) :- p(T), not -p(T+1)`

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans
- eric is a sporting fan (bruno doesn't care about sports)

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans
- eric is a sporting fan (bruno doesn't care about sports)
- carlos passionately uses acordo ortográfico, 1990

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans
- eric is a sporting fan (bruno doesn't care about sports)
- carlos passionately uses acordo ortográfico, 1990
- bruno is strictly pre-acordo ortográfico, 1990

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans
- eric is a sporting fan (bruno doesn't care about sports)
- carlos passionately uses acordo ortográfico, 1990
- bruno is strictly pre-acordo ortográfico, 1990
- fans of different teams are enemies

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans
- eric is a sporting fan (bruno doesn't care about sports)
- carlos passionately uses acordo ortográfico, 1990
- bruno is strictly pre-acordo ortográfico, 1990
- fans of different teams are enemies
- people writing differently are also enemies

Model the Following Problem

- andy, bruno, carlos, daniel, and eric are deciding on their careers whether to become lawful or gangsters.
- andy, carlos, and daniel are benfica fans
- eric is a sporting fan (bruno doesn't care about sports)
- carlos passionately uses acordo ortográfico, 1990
- bruno is strictly pre-acordo ortográfico, 1990
- fans of different teams are enemies
- people writing differently are also enemies
- enemies should not both be lawful nor be both gangsters

Basic facts

```
person(andy).
person(bruno).
person(carlos).
person(daniel).
person(eric).

fan(andy, benfica).
fan(carlos, benfica).
fan(daniel, benfica).

fan(eric, sporting).

writes(carlos, ao90).
writes(bruno, preao90).
```

Enemy

```
enemy(X, Y) :- fan(X, T1), fan(Y, T2), T1 != T2.
enemy(X, Y) :- writes(X, T1), writes(Y, T2), T1 != T2.
```

Choice between lawful and gangsters

```
lawful(X) :- not gangster(X), person(X).
gangster(X) :- not lawful(X), person(X).
```

Remark: `person(X)` is needed otherwise `X` is over undefined range.

Split enemies

```
:- gangster(X), gangster(Y), enemy(X, Y).
:- lawful(X), lawful(Y), enemy(X, Y).
```

Observations - Remember

- Rules with empty head (false) are called constraints
- The previous example follows generate & test pattern
- A set of rules generates possible solutions and constraints check their validity (giving only the intended ones)

Choice Syntax

Choose one and only one of lawful, gangster for any person.

```
1 { lawful(X) ; gangster(X) } 1 :- person(X).
```

Choice Syntax

Choose one and only one of lawful, gangster for any person.

```
1 { lawful(X) ; gangster(X) } 1 :- person(X).
```

Choose between `LowerBound` and `UpperBound` facts of `p(X)` for which `q(X)` and `z(X)` holds:

```
LowerBound { p(X) : q(X), z(X) } UpperBound
```

Another Example, N-Queens

```
% board size
#const n = 8.

% board
row(1..n).
col(1..n).

% generate
n { queen(I,J) : row(I), col(J) } n.

% test
:- queen(I,J1), queen(I,J2), J1 != J2.
:- queen(I1,J), queen(I2,J), I1 != I2.
:- queen(I1,J1), queen(I2,J2),
   (I1,J1) != (I2,J2), I1+J1 == I2+J2.
:- queen(I1,J1), queen(I2,J2),
   (I1,J1) != (I2,J2), I1-J1 == I2-J2.
```

N-Queens, Improvement?

```
% board size
#const n = 8.

% board
row(1..n).
col(1..n).

% generate
1 { queen(I,J) : row(J) } 1 :- col(I).

% test
:- queen(I,J1), queen(I,J2), J1 != J2.
:- queen(I1,J), queen(I2,J), I1 != I2.
:- queen(I1,J1), queen(I2,J2),
   (I1,J1) != (I2,J2), I1+J1 == I2+J2.
:- queen(I1,J1), queen(I2,J2),
   (I1,J1) != (I2,J2), I1-J1 == I2-J2.
```

Outline

The seating problem

- Persons $p_1, \cdots, p_n$, are invited for dinner.
- There are tables $t_1, \cdots, t_k$ with the respective capacities $c_1, \cdots, c_k$ available for seating such that $c_1 + \cdots + c_k \geq n$.
- There are both friends and enemies among the invitees. Friends should be seated with other friends. Enemies cannot be seated in the same table.
- A solution to this problem is a mapping $s(p_i) = t_j$ of persons p_i to tables t_j so that the mutual relationships are respected.

The seating problem: instance

- Group of 20 people: Alice, Bob, John, and others
- Four tables, seating 7, 6, 5, and 4 people, respectively
- Alice likes Bob, Bob likes John, Alice dislikes John, John dislikes Alice, and so on
- The goal is (1) to find at least one solution that fulfills the criteria set in the problem statement, or (2) to show that no solution exists.

The seating problem: encoding the instance

```
% Instance
person(alice). person(bob). person(john).
likes(alice,bob). likes(bob,john). ...
dislikes(alice,john). dislikes(john,alice). ...
tbl(1,7). tbl(2,6). tbl(3,5). tbl(4,4).
```

The seating problem: encoding the problem

```
% Rules and Constraints
1 { seat(P,T): tbl(T,_) } 1 :- person(P).
:- #count{seat(P,T): person(P)}>C, tbl(T,C).
:- likes(P1,P2), seat(P1,T1), seat(P2,T2),
   person(P1), person(P2),
   tbl(T1,_), tbl(T2,_), T1 != T2.
:- dislikes(P1,P2), seat(P1,T), seat(P2,T),
   person(P1), person(P2), tbl(T,_).
```

First rule to be further explained later

$\texttt{\#count}\{\texttt{atom}_1\texttt{;}\ldots\texttt{;atom}_k\}\texttt{>C}$ is the same as $\texttt{C+1}\{\texttt{atom}_1\texttt{;}\ldots\texttt{;atom}_k\}$

The seating problem: complete ASP encoding

```
% Instance
person(alice). person(bob). person(john).
likes(alice,bob). likes(bob,john). ...
dislikes(alice,john). dislikes(john,alice). ...
tbl(1,7). tbl(2,6). tbl(3,5). tbl(4,4).

% Rules and Constraints
1 { seat(P,T): tbl(T,_) } 1 :- person(P).
:- #count{seat(P,T): person(P)}>C, tbl(T,C).
:- likes(P1,P2), seat(P1,T1), seat(P2,T2),
   person(P1), person(P2),
   tbl(T1,_), tbl(T2,_), T1 != T2.
:- dislikes(P1,P2), seat(P1,T), seat(P2,T),
   person(P1), person(P2), tbl(T,_).
```

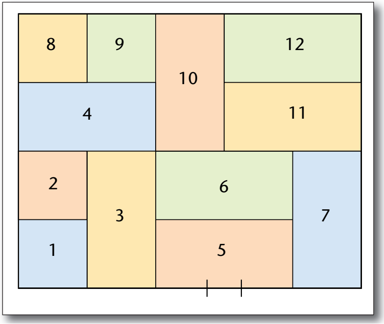
The seating problem: more details

- `1 {seat(P,T) : tbl(T,_)} 1 :- person(P).`
  is instantiated when `P` and `T` are replaced
- For example, `P = alice` and `T = [1..4]` yields an instance

```
1 {seat(alice,1); seat(alice,2); seat(alice,3);
    seat(alice,4)} 1 :- person(alice).
```

Locking Design

Goal: design a locking scheme for a building while ensuring accessibility



Single floor with 12 rooms

Locking design: objectives

- Describe the domain with adequate predicates
- Design goal: minimize the number of evacuation connections
- Safety requirement: floor must be effectively evacuated

Locking design: domain rules

- `room/1` defines rooms
- `adj/2` adjacency relation for rooms ($X < Y$)
- `pot/2` potential of installing doors between rooms
- `exit/1` rooms having exits

Locking design: domain rules (cont.)

```
room(R1) :- adj(R1, R2).
room(R2) :- adj(R1, R2).
pot(R1, R2) :- adj(R1, R2).
pot(R1, R2) :- adj(R2, R1).
```

Example:

`adj(1,2).adj(1,3).adj(2,3).adj(2,4). . . .adj(11,12).exit(5).`

Locking design: evacuation plan

`evac/2` existence of evacuation route between rooms
`reach/2` reachability of rooms
`ok/1` room ok if some exit is reachable

Example: at least 11 connections (22020 solutions)
Problem: length of the evacuation routes!

Locking design: evacuation plan (cont.)

```
{evac(R1,R2);evac(R2,R1)}1 :- pot(R1,R2).

reach(R, R) :- room(R).
reach(R1, R2) :- reach(R1, R3), evac(R3, R2).

ok(R) :- room(R), reach(R, X), exit(X).
:- not ok(R), room(R).

#minimize{1,R1,R2 : evac(R1, R2) }.
```

Locking design: evacuation plan II

`step/1` characterizes the (number of) existing steps
`reach/3` number of steps required to connect two rooms

Example with `s=2`: no plan is found
Example with `s=3`: plan with 11 connections is found (152 plans)

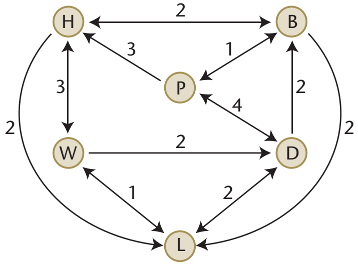
Locking design: evacuation plan II (cont.)

```
{evac(R1,R2);evac(R2,R1)}1 :- pot(R1,R2).

step(0..s).

reach(R, R, 0) :- room(R).
reach(R1, R2, S+1) :- reach(R1, R3, S), evac(R3, R2),
                      step(S), step(S+1).

ok(R) :- room(R), reach(R, X, S), exit(X), step(S).
:- not ok(R), room(R).

#minimize{1,R1,R2 : evac(R1, R2) }.
```

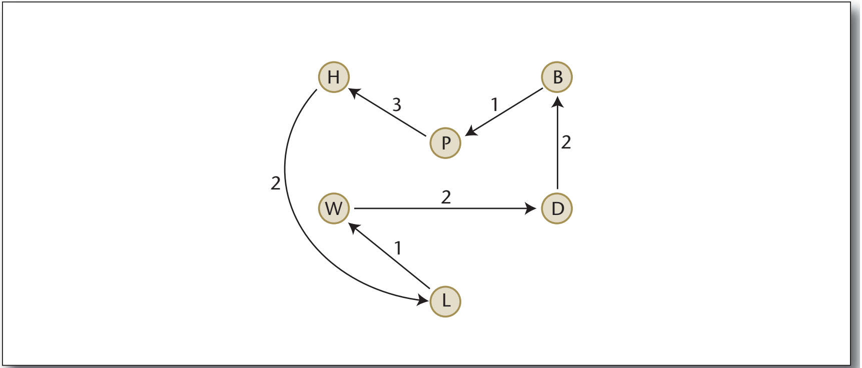
Traveling Salesman Problem (TSP): definition

- Number of places
- Links between two places associated with a cost / distance
- Find smaller cost / shortest round trip

TSP: example instance



TSP: example solution



11 hours: B $\xrightarrow{1}$ P $\xrightarrow{3}$ H $\xrightarrow{2}$ L $\xrightarrow{1}$ W $\xrightarrow{2}$ D $\xrightarrow{2}$ B

TSP: instance encoding

`place/1` places in the problem instance
`link/3` links between two places with a given cost

TSP: example instance

```
% Instance
place(b). % Berlin
place(d). % Dresden
place(h). % Hamburg
place(l). % Leipzig
place(p). % Postdam
place(w). % Wolfsburg

link(b, h, 2). link(b, p, 2). link(b, p, 1).
link(d, b, 2). link(d, l, 2). link(d, p, 4).
link(h, b, 2). link(h, l, 2). link(h, w, 3).
link(l, d, 2). link(l, w, 1).
link(p, b, 1). link(p, d, 4). link(p, h, 3).
link(w, d, 2). link(w, h, 3). link(w, l, 1).
```

TSP: problem encoding (natural language)

- Every place is linked to exactly one successor in a trip.
- Starting from an arbitrary place, a trip visits all places and then returns to its starting point.
  - Every place is linked to exactly one predecessor in a trip.
- The sum of costs associated with the links in a trip ought to be minimal.

TSP: problem encoding (structure)

Generate-and-test pattern

- DOMAIN: auxiliary concepts
- GENERATE: provide solution candidates
- DEFINE: relevant properties of solution candidates
- TEST: elimination of invalid candidates
- OPTIMIZE: optimization statement or weak constraints
- DISPLAY: predicates restricting answer set output

TSP: problem encoding (code)

```
% DOMAIN
start(X) :- X = #min{ Y : place(Y) }.

% GENERATE
{travel(X,Y) : link(X,Y,C)} = 1 :- place(X).

% DEFINE
visit(X) :- start(X).
visit(Y) :- visit(X), travel(X,Y).

% TEST
:- place(Y), not visit(Y).
:- start(Y), #count{X : travel(X,Y)} < 1.
:- place(Y), #count{X : travel(X,Y)} > 1.

% OPTIMIZE
:~ travel(X,Y), link(X,Y,C). [C,X]

% DISPLAY
#show travel/2.
```

TSP: code explained (I)

```
start(X) :- X = #min{Y: place(Y)}.
```

Meaning of the rule: determine the lexicographically smallest identifier among places in an instance as (arbitrary) starting point for the construction of a round trip.

Instantiation:

```
start(b) :- b = #min{ b :  place(b); d :  place(d);
                      h :  place(h); p :  place(p);
                      l :  place(l); w :  place(w)}.
```

Simplified to:

```
start(b) :- b = #min{b; d; h; p; l; w}.
```

TSP: code explained (II)

Instantiation of choice rule

```
{travel(X,Y) : link(X,Y,C)} = 1 :- place (X).
```

considering X = Potsdam

```
{travel(p, b) :  link(p, b, 1);
 travel(p, d) :  link(p, d, 4);
 travel(p, h) :  link(p, h, 3)} = 1 :- place(p).
```

After simplifications:

```
{travel(p, b); travel(p, d); travel(p, h)} = 1.
```

TSP: code explained (III)

(partial) Instantiation of rules in lines 8-9

```
visit(b) :- start(b).
visit(p) :- visit(b), travel(b, p).
visit(h) :- visit(p), travel(p, h).
visit(l) :- visit(h), travel(h, l).
visit(w) :- visit(l), travel(l, w).
visit(d) :- visit(w), travel(w, d).
```

TSP: code explained (IV)

Instantiations of constraints in test

Line 13: a trip returns to its starting point

```
:- start(b), #count{d : travel(d, b); h : travel(h, b);
                    p : travel(p, b)} < 1.
```

Line 14: a place cannot be linked to several predecessors

```
:- place(b), #count{d : travel(d, b); h : travel(h, b);
                    p : travel(p, b)} > 1.
```

TSP: code explained (V)

OPTIMIZE: weak constraints - an answer set is optimal if the obtained cost is minimal among all answer sets of the given program

```
:~ travel(b, h), link(b, h, 2).  [2, b]
:~ travel(b, l), link(b, l, 2).  [2, b]
:~ travel(b, p), link(b, p, 1).  [1, b]
```

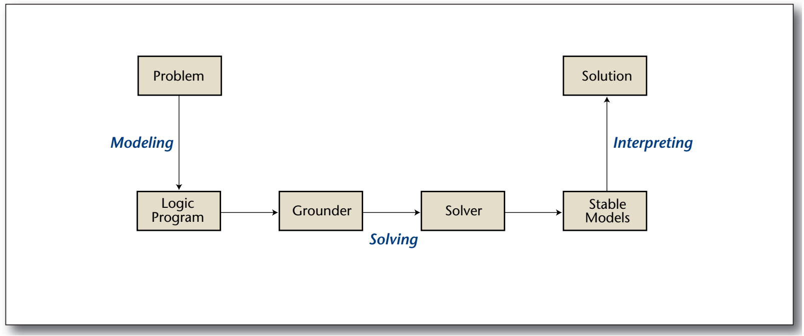
the weak constraint in line 17 associates every place with the cost of the link to its successor in a round trip.

TSP: solver run (clingo)

```
$ clingo tsp-ins.lp tsp-enc.lp

Answer: 1
    travel(b,l) travel(l,w) travel(w,d)
    travel(d,p) travel(p,h) travel(h,b)
Optimization: 14

Answer: 2
    travel(b,p) travel(p,h) travel(h,w)
    travel(w,l) travel(l,d) travel(d,b)
Optimization: 12

Answer: 3
    travel(b,p) travel(p,h) travel(h,l)
    travel(l,w) travel(w,d) travel(d,b)
Optimization: 11

OPTIMUM FOUND
```

Outline

The workflow of ASP



(Naive) Grounding

- Ground program does not contain any variable
- Grounding (a.k.a. instantiation) can be exponential
- Example with 2^n ground rules:

```
obj(0).
obj(1).
tuple(X1,...,Xn) :- obj(X1), ..., obj(Xn).
```

Grounding: useless rules

- Program:
```
c(1,2).
a(X) | b(Y) :- c(X,Y).
```
- Full instantiation:
```
a(1) | b(1) :- c(1,1).
a(2) | b(1) :- c(2,1).
a(2) | b(2) :- c(2,2).
a(1) | b(2) :- c(1,2).
```
- First three ground rules are useless!

The Instantiation Procedure

- Rule instantiation: Given a rule r and the set of ground atoms S, which represent the extention of the predicates, generate the ground instances of r
- In practice: iterate on the body literals looking for possible substitutions for their variables

The Instantiation Procedure: example

- Non-ground rule r:

```
a(X) | b(Y) :- p(X,Z), q(Z,Y).
```

- $S = \{$ `p(1,2), q(2,1), q(2,3)` $\}$
- Ground atom in S matching body of r, i.e. `p(X,Z)` first and then `q(Z,Y)` (after propagation)?
  - Performing backtracking if needed
- Only two ground rules:

```
a(1) | b(1) :- p(1,2), q(2,1).
a(1) | b(3) :- p(1,2), q(2,3).
```

The Instantiation Procedure: computation of S

- Initially $S = \mathit{Facts}$, i.e. ground atoms of the program
- S is expanded with the ground atoms in the *head* of the newly generated ground rules
- Previous example: `a(1)`, `b(1)` and `b(3)` added to S and possibly used to extend S
- The evaluation order is important!
  - To guarantee that the reduced ground program has the same answer sets of the full instantiation, while being possibly smaller

The Instantiation Procedure: evaluation order

- *Dependency Graph*: directed graph describing how predicates depend on each other
- Graph induces a partition of the input program into subprograms, associated with the strongly connected components, and a topological ordering over them
- Subprograms are instantiated one at a time starting from the ones associated with the lowest components in the topological ordering

The Instantiation Procedure: recursive rules

- *Reachability* as an example: given a finite directed graph, compute all pairs of nodes (a, b) such that b is reachable from a through a nonempty sequence of arcs.

```
reach(X,Y) :- arc(X,Y).
reach(X,Y) :- arc(X,U), reach(U,Y).
```

- Assume $S = \{$`arc(1,2), arc(2,3), arc(3,4)`$\}$

The Instantiation Procedure: recursive rules (cont.)

- Three ground instances:
```
reach(1,2) :- arc(1,2).
reach(2,3) :- arc(2,3).
reach(3,4) :- arc(3,4).
```
- $S = S \cup \{\texttt{reach(1,2), reach(2,3), reach(3,4)}\}$

The Instantiation Procedure: recursive rules (cont.)

- More two ground instances:
  ```
  reach(1,3) :- arc(1,2), reach(2,3).
  reach(2,4) :- arc(2,3), reach(3,4).
  ```
- $S = S \cup \{\texttt{reach(1,3), reach(2,4)}\}$

The Instantiation Procedure: recursive rules (cont.)

- One more ground instance:
  `reach(1,4) :- arc(1,2), reach(2,4).`
- $S = S \cup \{\texttt{reach(1,4)}\}$
- New iteration: fix point reached!
- Additional optimizations exist...
- Function symbols bring new challenges...

Solving: general outline

- Modern ASP solvers rely upon advanced conflict-driven search procedures, pioneered in the area of satisfiability testing (SAT)
- Conflicts are analyzed and recorded, decisions are taken in view of conflict scores, and back-jumps are directed to the origin of a conflict

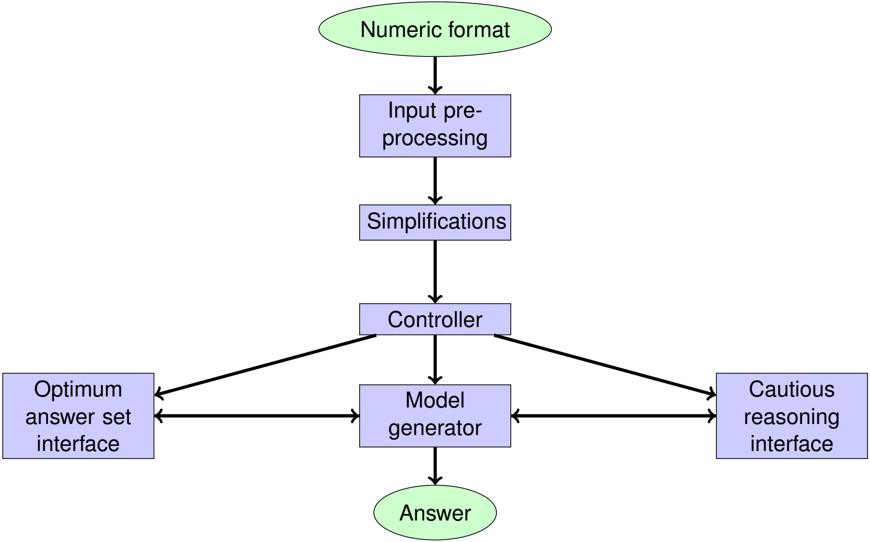
Solving: ASP vs SAT

1. Semantics enforces that atoms are not merely true but provably true
2. Rich modeling language of ASP comes with complex language constructs
   - Disjunction in rule heads and nonmonotone aggregates lead to an elevated level of computational complexity
3. ASP with various computational tasks: model generation, optimum answer set search, cautious reasoning, etc.

Computational tasks: more details

- Model generation: given a ground ASP program P, find an answer set of P
- Optimum answer set search: given a ground ASP program P, find an answer set of P with the minimum cost
- Cautious reasoning: given a ground ASP program P and a ground atom a, check whether a is true in all answer sets of P

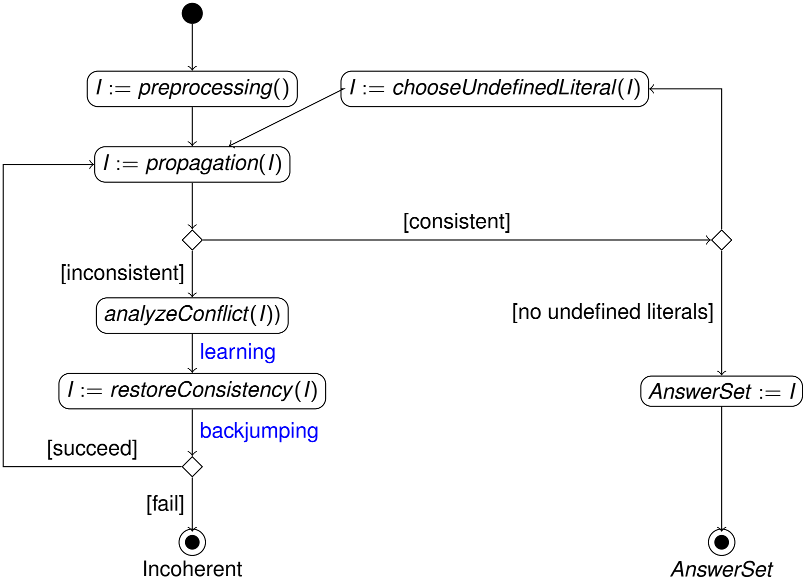
Arquitecture of an ASP solver



Input preprocessing and simplifications

- Preprocessing of the input program
  - Deletion of duplicate rules: even more than 80% in some benchmarks
  - Deterministic inferences: deletion of satisfied rules
  - Clark's completion constraints for discarding unsupported models
- Simplifications
  - In the style of SATELITE [Eén and Biere, SAT 2005]

Model Generator



Model Generator: propagation

- Unit propagation (from SAT)
- Aggregates propagation (from Pseudo-Boolean)
- Unfounded-free propagation (ASP specific)

Solving: inference main idea

- Map inferences in ASP onto unit propagation on nogoods, which traces back to a characterization of answer sets in propositional logic

Solving: inference example

Program P

$$P = \left\{ \begin{array}{ll} \texttt{a :- not b.} & \texttt{b :- not a.} \\ \texttt{x :- a, not c.} & \texttt{x :- y.} \\ \texttt{y :- x, b.} & \end{array} \right\}$$

Interpreting P in propositional logic (default negation becomes classical)

$$RF(P) = \left\{ \begin{array}{ll} \texttt{a} \leftarrow \neg\texttt{b.} & \texttt{b} \leftarrow \neg\texttt{a.} \\ \texttt{x} \leftarrow (\texttt{a} \wedge \neg\texttt{c}) \vee \texttt{y.} & \\ \texttt{y} \leftarrow \texttt{x} \wedge \texttt{b.} & \end{array} \right\}$$

Obs: c is not supported by any rule, as is b whenever a is true as well

Solving: inference example (cont.)

Models with unsupported atoms are eliminated by turning implications onto equivalences

$$CF(P) = \left\{ \begin{array}{ll} \texttt{a} \leftrightarrow \neg\texttt{b.} & \texttt{b} \leftrightarrow \neg\texttt{a.} \\ \texttt{x} \leftrightarrow (\texttt{a} \wedge \neg\texttt{c}) \vee \texttt{y.} & \\ \texttt{y} \leftrightarrow \texttt{x} \wedge \texttt{b.} & \texttt{c} \leftrightarrow \bot. \end{array} \right\}$$

3 models:

1. b true
2. b, x, y true
3. a, x true

Solving: inference example (cont.)

Obs: x and y support each other in a circular way
Need to learn a new rule / nogood:

$$LF(P) = \{ \texttt{(x} \lor \texttt{y)} \rightarrow \texttt{(a} \land \neg\texttt{c)} \}$$

Consequences: model 2 no longer holds

Outline

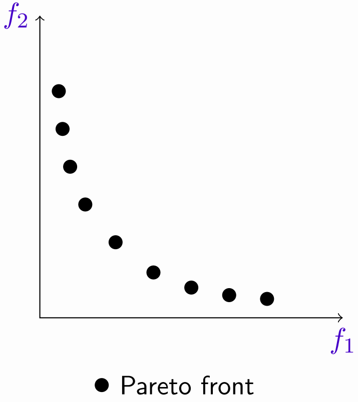
Multi-Objective Problems - Examples

- Users define multiple conflicting objectives
- Package Upgradability
  - Maximize the number of installed packages
  - Maximize the number of up-to-date packages
  - Minimize the number of packages to be removed from the current installation

Multi-Objective Problems - Examples

- Users define multiple conflicting objectives
- Package Upgradability
  - Maximize the number of installed packages
  - Maximize the number of up-to-date packages
  - Minimize the number of packages to be removed from the current installation
- Virtual Machine Consolidation
  - Minimize energy consumption
  - Minimize migration of virtual machines
  - Minimize resource wastage

Multi-Objective Combinatorial Optimization (MOCO)



$$\text{min} \quad f_1 : \sum l_j$$

$$\text{min} \quad f_2 : \sum l_j$$

$$l_j \in \{x_j, \neg x_j\}$$

$$\phi_h = \left\{ \sum a_{ij} x_j \geq b_i : i \in 1..m \right\}$$

Multi-Objective Algorithms

Main approaches

- Specialized algorithms
- Branch and Bound
- Stochastic approaches

Enhance multi-objective combinatorial optimization algorithms through SAT solving

Multi-Objective Algorithms

SAT-based main approaches

- Iterative MaxSAT [CP'17]
- MCS Enumeration [SAT'17, IJCAI'18, AAAI'18]
- Core-Guided and Hitting-Set based Algorithms [TACAS'23]
- Slide & Drill [CP'24]

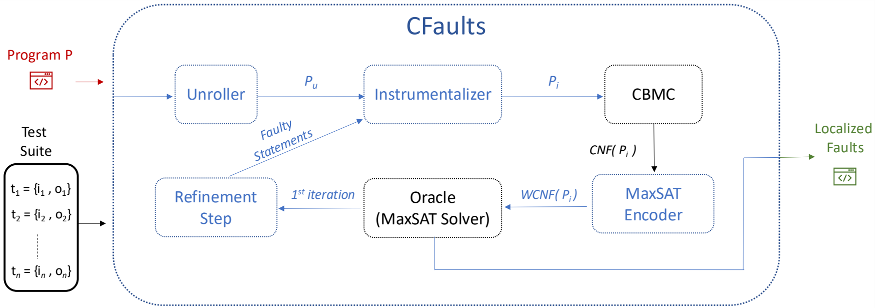
Next Steps:

- Approximation Algorithms
- Proofs of correctness in MOCO

CFaults

[FM'24]

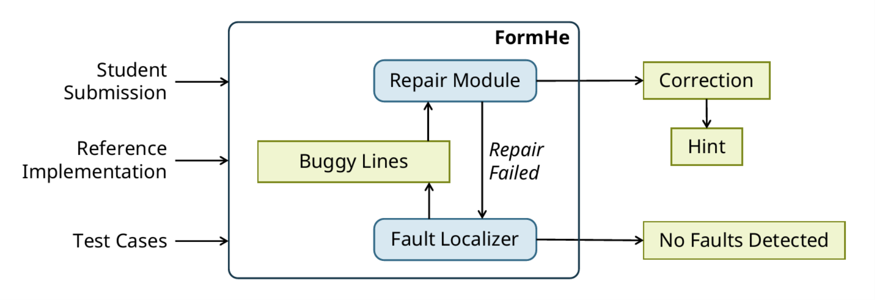
- Leverages MaxSAT to find the minimal number of faulty lines
- Deployed in Introduction to Algorithms and Data Structures course



Fault-Localization and Repair in ASP Programs - FormHe

FormHe

- Uses MCSs to find a minimal set of faulty lines
- Repair using Large Language Models (LLMs)
- To be deployed in this course!!



Conclusion

- Hope you enjoyed the course!!!
- Lots of new applications:
  - Model revision and repair
  - Multi-Objective Optimization
  - Software Engineering
    - Fault Localization
    - Program Repair