Answer Set Programs: Modeling and Solving

Martin Gebser

University of Potsdam

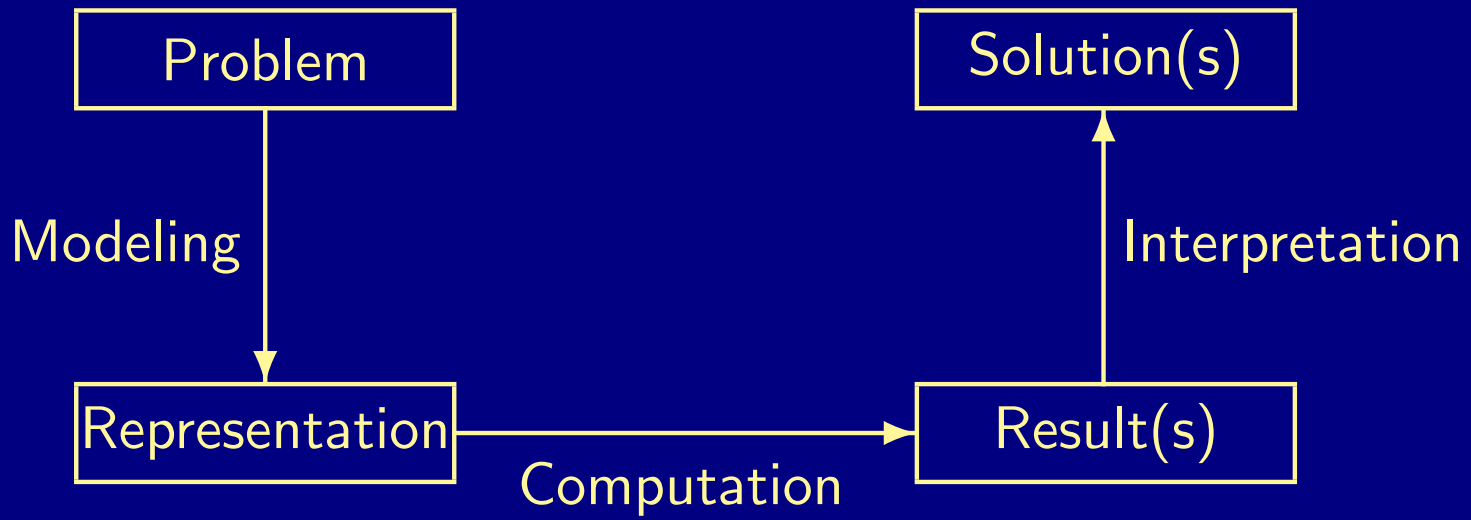
13 January 2012

Outline

Outline

Declarative Programming

What is the problem? But not how to solve it!

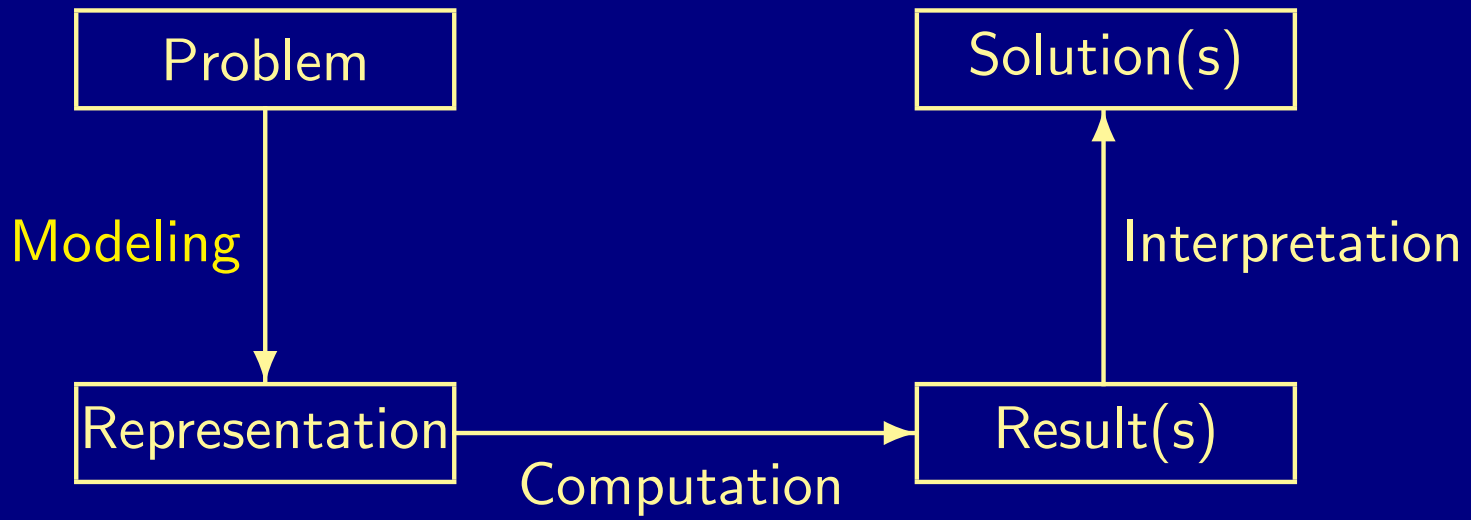


Desiderata

- simplicity
- uniformity
- domain independence
- average-case efficiency

Declarative Programming

What is the problem? But not how to solve it!

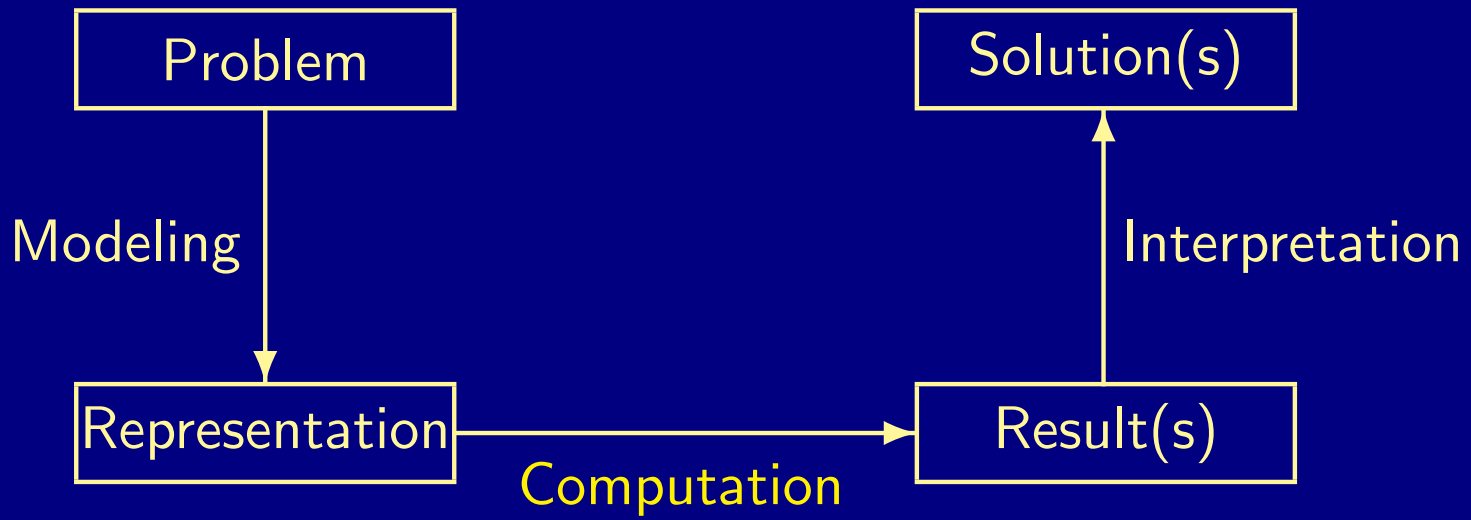


Desiderata

- simplicity
- uniformity
- domain independence
- average-case efficiency

Declarative Programming

What is the problem? But not how to solve it!

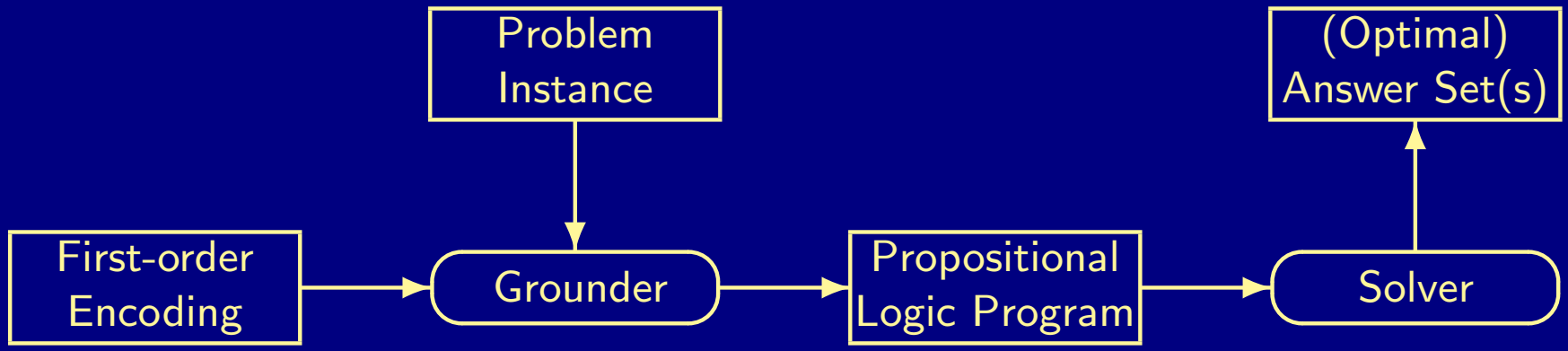


Desiderata

- simplicity
- uniformity
- domain independence
- average-case efficiency

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

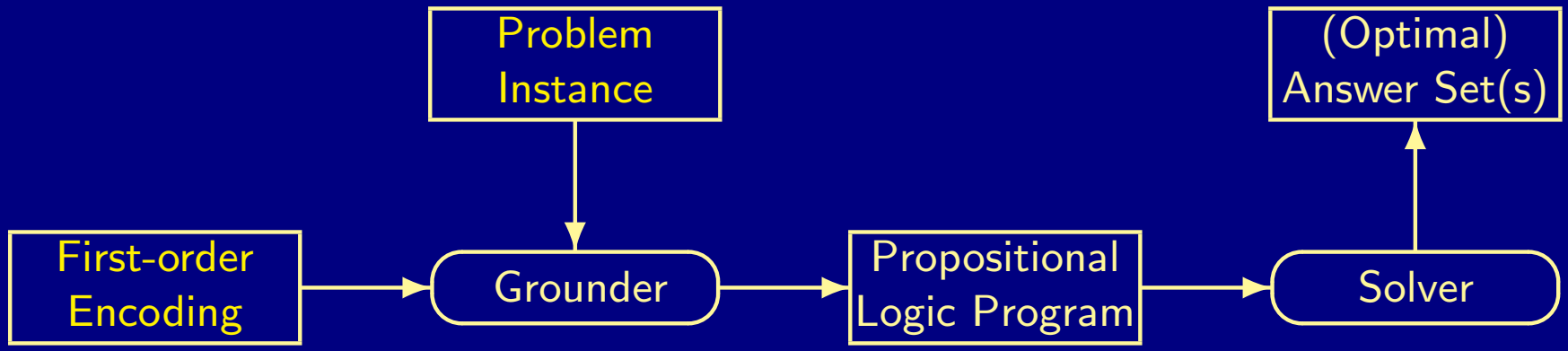
☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

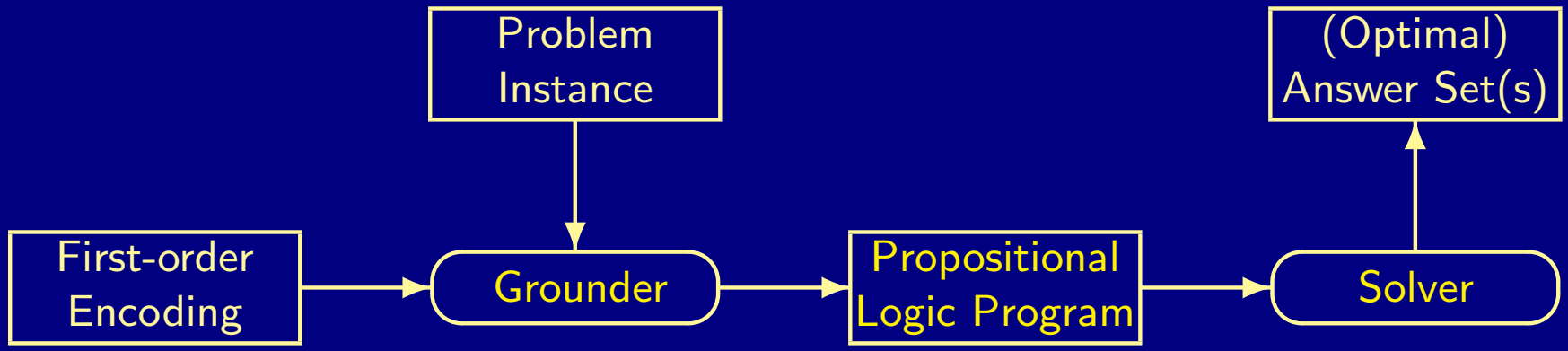
☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

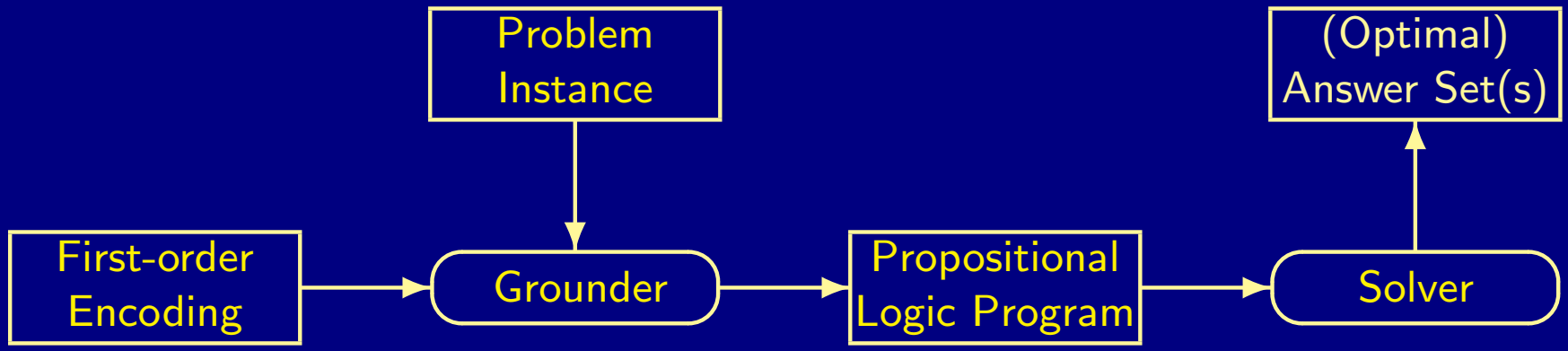
☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

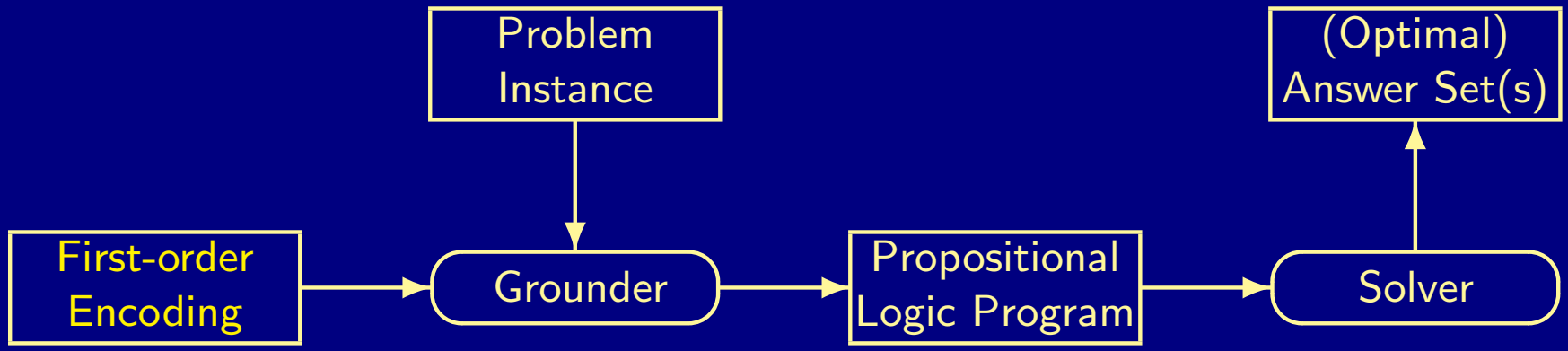
☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

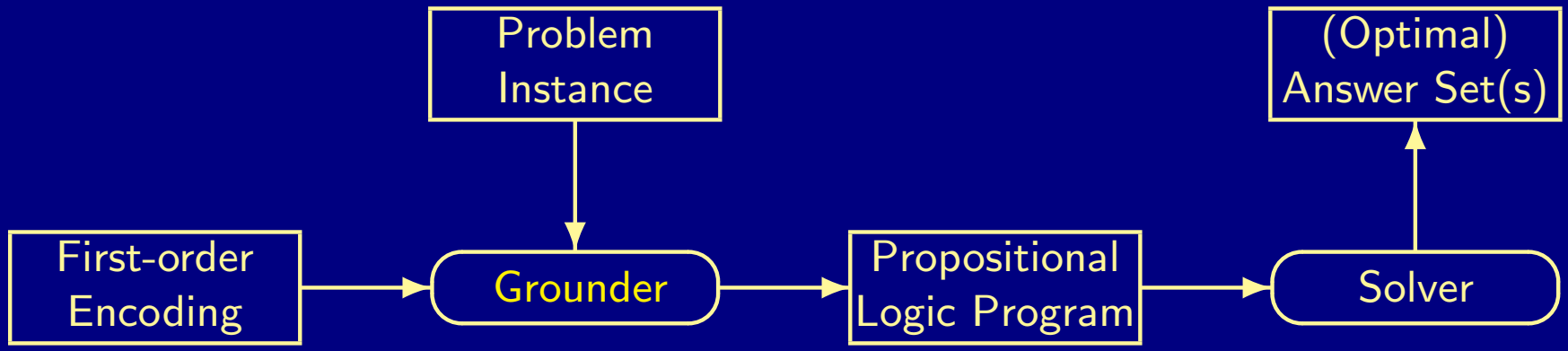
☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

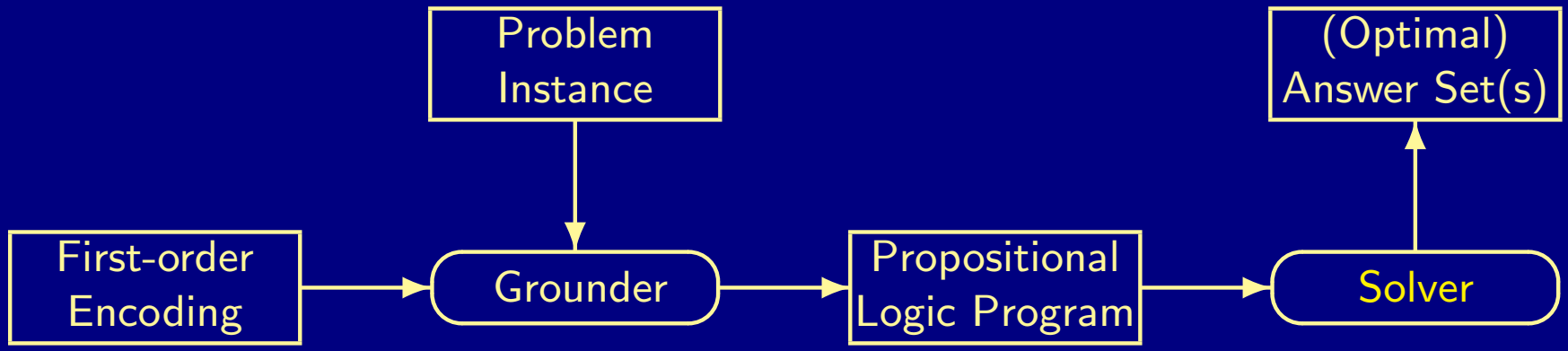
☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Answer Set Programming (ASP)

for Automated Declarative Problem Solving



Advantages of ASP

- easy modeling
- powerful reasoning

☞ All problems in NP (and NP^{NP}) can be uniformly modeled and solved.

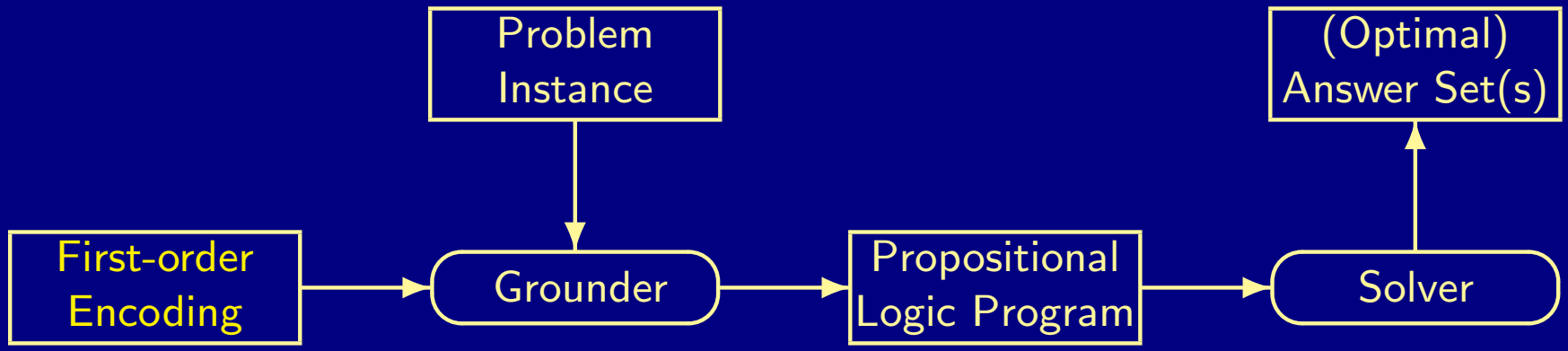
In a Nutshell

ASP = Knowledge Representation + Database Techniques + Search

Outline

Uniform Problem Representation

Problem + Instance $\longmapsto$ Rules + Facts



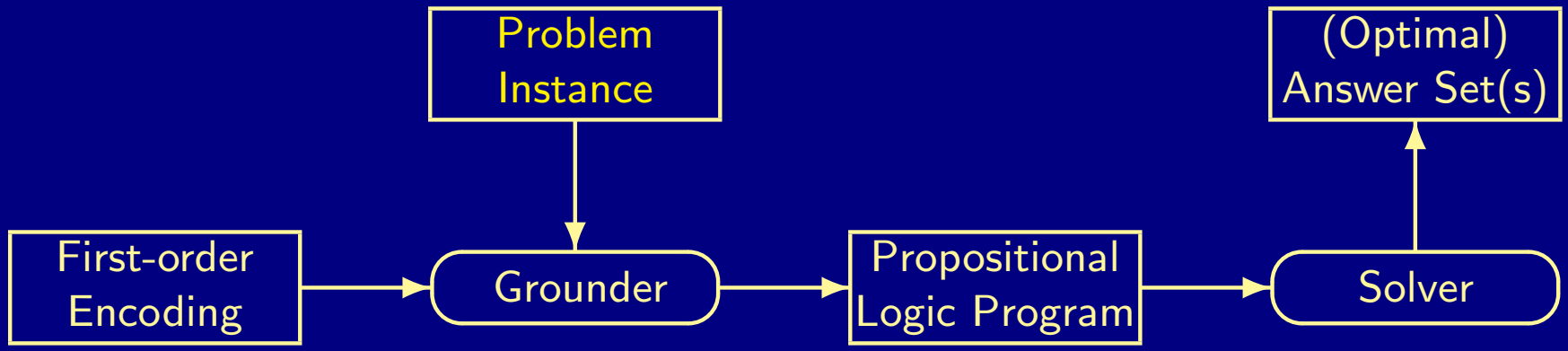
Solving a problem class P for a problem instance I

1. encode the problem class P as a set C(P) of rules and
2. the problem instance I as a set C(I) of facts such that
3. (optimal) answer sets of C(P) $\cup$ C(I) give (optimal) solutions to P for I

☞ A uniform encoding C(P) specifies (optimal) solutions to P w.r.t. any set C(I) of facts for an instance I.

Uniform Problem Representation

Problem + Instance $\longmapsto$ Rules + Facts



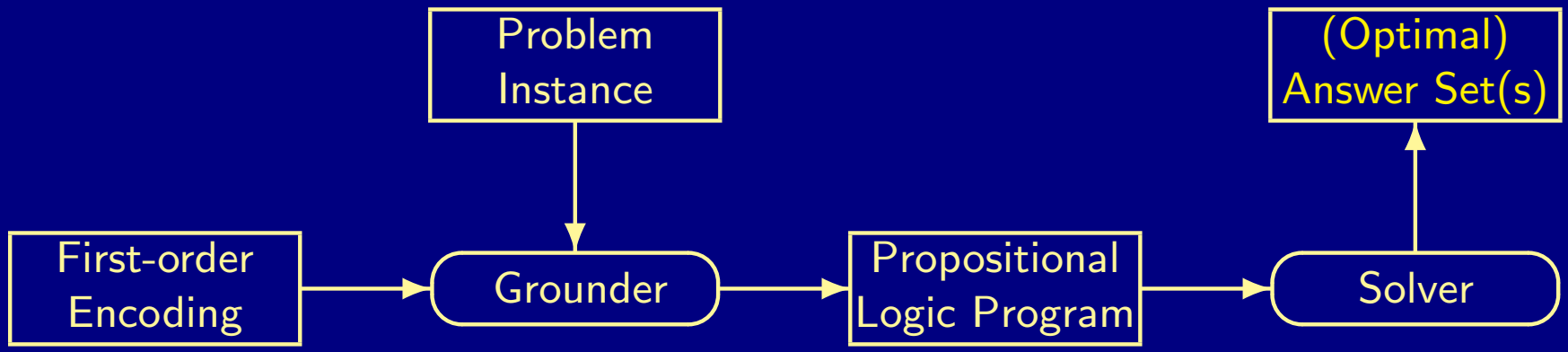
Solving a problem class P for a problem instance I

1. encode the problem class P as a set C(P) of rules and
2. the problem instance I as a set C(I) of facts such that
3. (optimal) answer sets of $C(P) \cup C(I)$ give (optimal) solutions to P for I

☞ A uniform encoding C(P) specifies (optimal) solutions to P w.r.t. any set C(I) of facts for an instance I.

Uniform Problem Representation

Problem + Instance $\longmapsto$ Rules + Facts



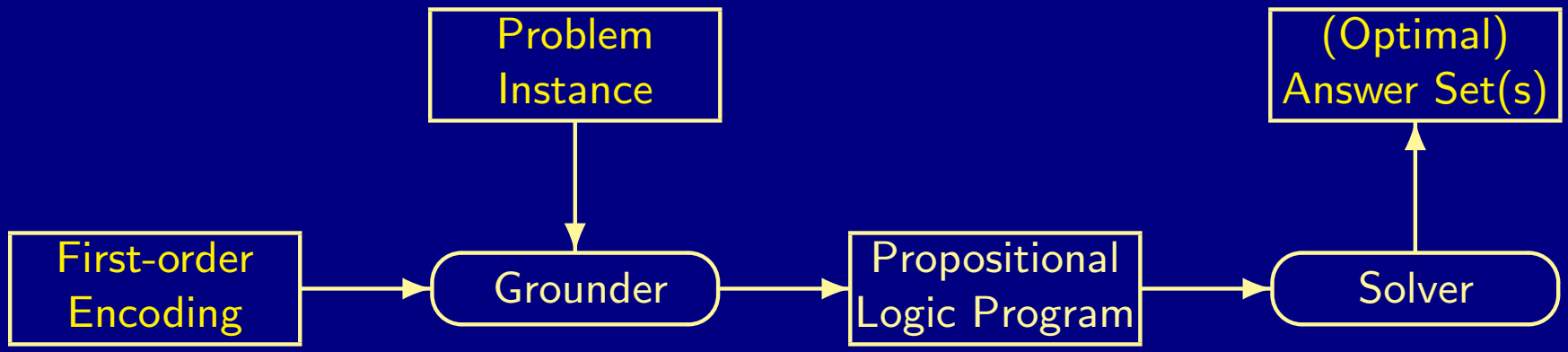
Solving a problem class P for a problem instance I

1. encode the problem class P as a set C(P) of rules and
2. the problem instance I as a set C(I) of facts such that
3. (optimal) answer sets of $C(P) \cup C(I)$ give (optimal) solutions to P for I

☞ A uniform encoding C(P) specifies (optimal) solutions to P w.r.t. any set C(I) of facts for an instance I.

Uniform Problem Representation

Problem + Instance $\longmapsto$ Rules + Facts



Solving a problem class P for a problem instance I

1. encode the problem class P as a set C(P) of rules and
2. the problem instance I as a set C(I) of facts such that
3. (optimal) answer sets of $C(P) \cup C(I)$ give (optimal) solutions to P for I

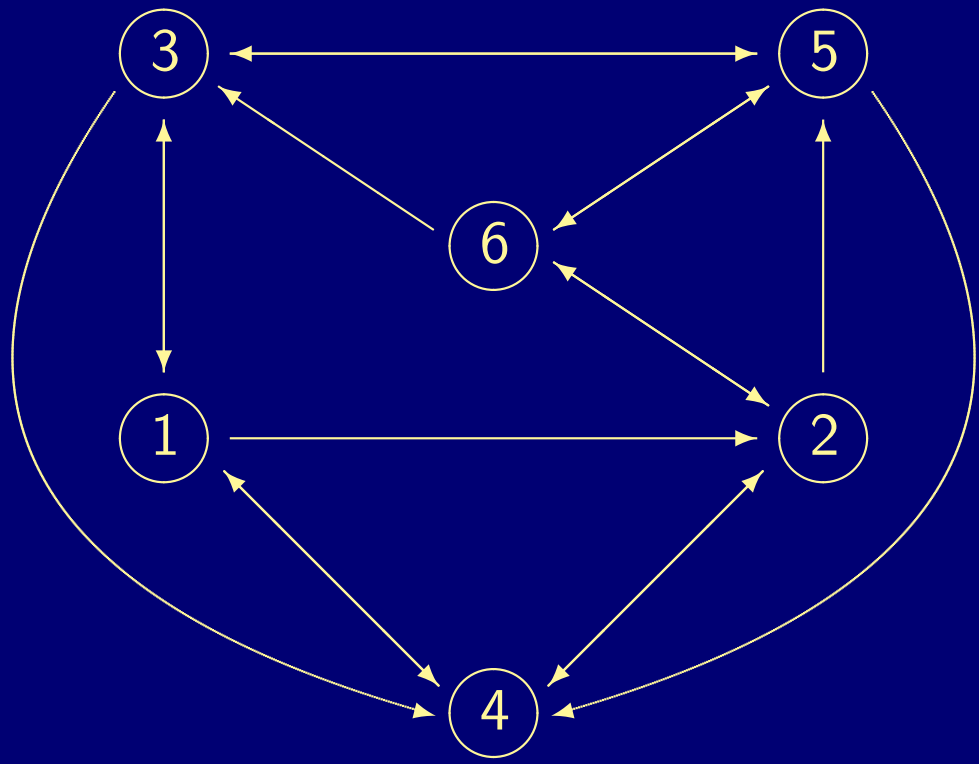
☞ A uniform encoding C(P) specifies (optimal) solutions to P w.r.t. any set C(I) of facts for an instance I.

Hamiltonian Cycle

Problem Instance as Facts

A directed graph (graph.lp)

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).
```

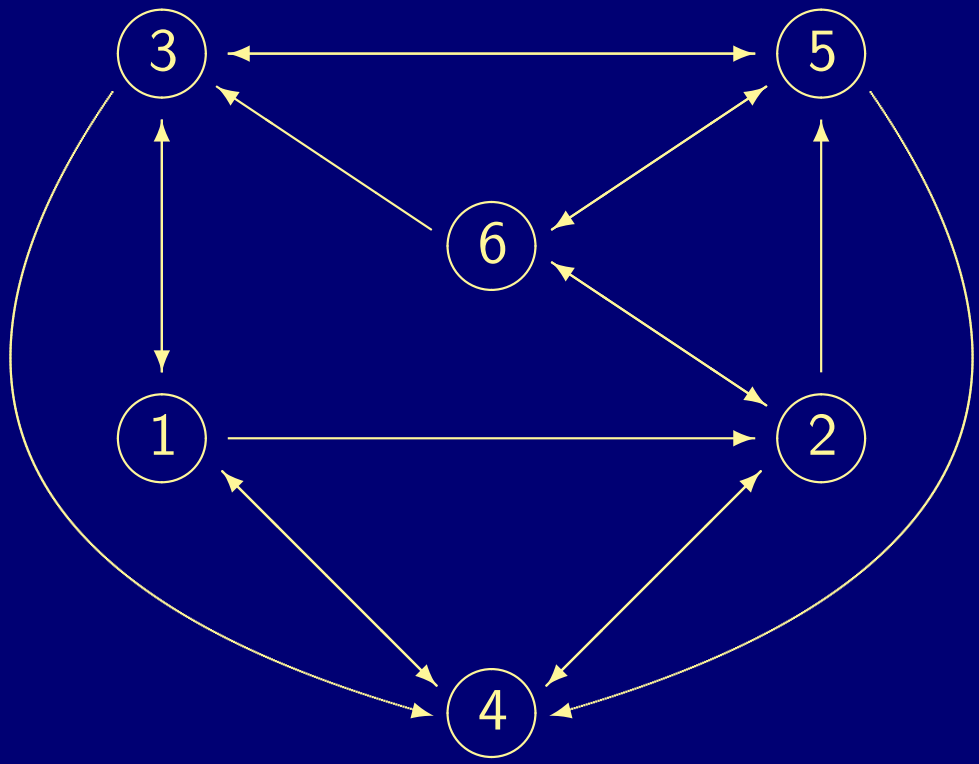


Hamiltonian Cycle

Problem Instance as Facts

A directed graph (`graph.lp`)

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).
```



Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (cycle.lp)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (`cycle.lp`)

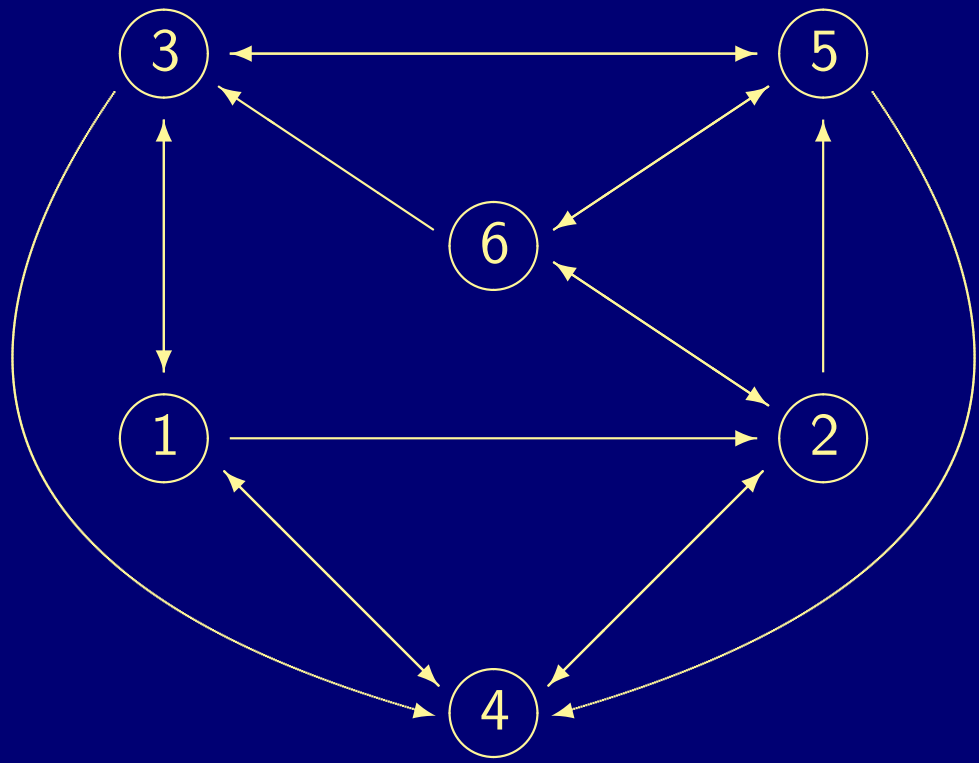
```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Hamiltonian Cycle

Automated (Intelligent) Grounding

```
gringo --text graph.lp cycle.lp
```

```
node(1).  ...  node(6).
edge(1,2). edge(1,3). edge(1,4). ...
edge(6,2). edge(6,3). edge(6,5).

1 #count{ cycle(3,1), cycle(4,1) } 1.
1 #count{ cycle(1,2), cycle(1,3),
                      cycle(1,4) } 1.
...
1 #count{ cycle(2,6), cycle(5,6) } 1.
1 #count{ cycle(6,2), cycle(6,3),
                      cycle(6,5) } 1.

reach(1).
reach(2) :- cycle(1,2).
reach(2) :- reach(4), cycle(4,2).
reach(2) :- reach(6), cycle(6,2).
...
:- not reach(2).  ...  :- not reach(6).
```

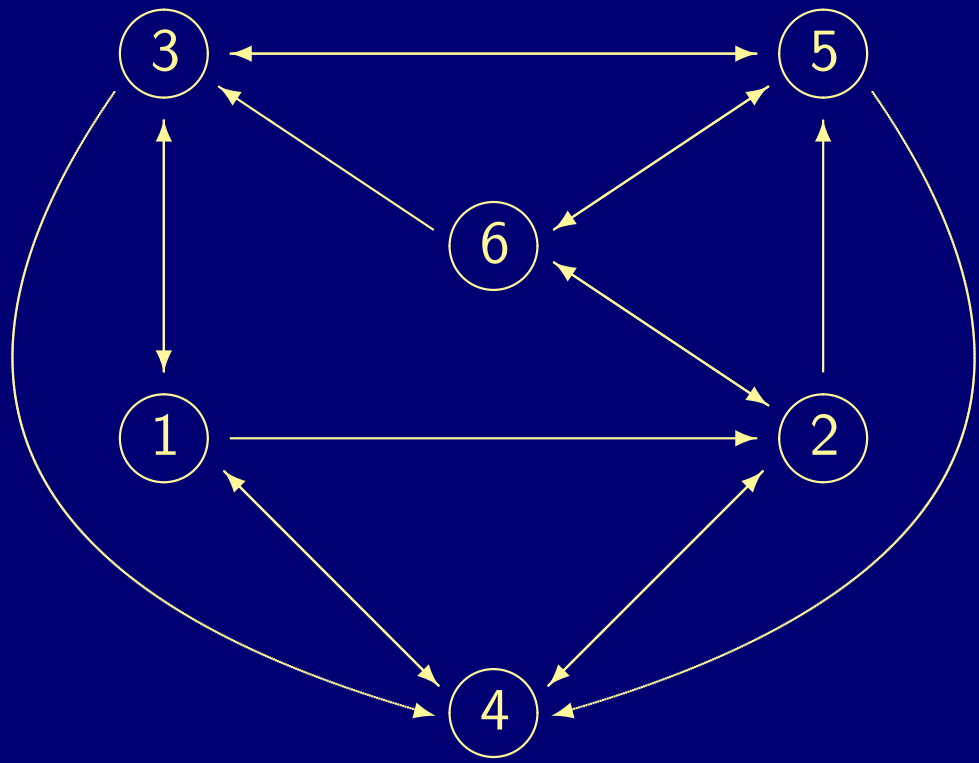


Hamiltonian Cycle

Automated (Intelligent) Grounding

```
gringo --text graph.lp cycle.lp
```

```
node(1).  ...  node(6).
edge(1,2). edge(1,3). edge(1,4). ...
edge(6,2). edge(6,3). edge(6,5).

1 #count{ cycle(3,1), cycle(4,1) } 1.
1 #count{ cycle(1,2), cycle(1,3),
                      cycle(1,4) } 1.
...
1 #count{ cycle(2,6), cycle(5,6) } 1.
1 #count{ cycle(6,2), cycle(6,3),
                      cycle(6,5) } 1.

reach(1).
reach(2) :- cycle(1,2).
reach(2) :- reach(4), cycle(4,2).
reach(2) :- reach(6), cycle(6,2).
...
:- not reach(2).  ...  :- not reach(6).
```

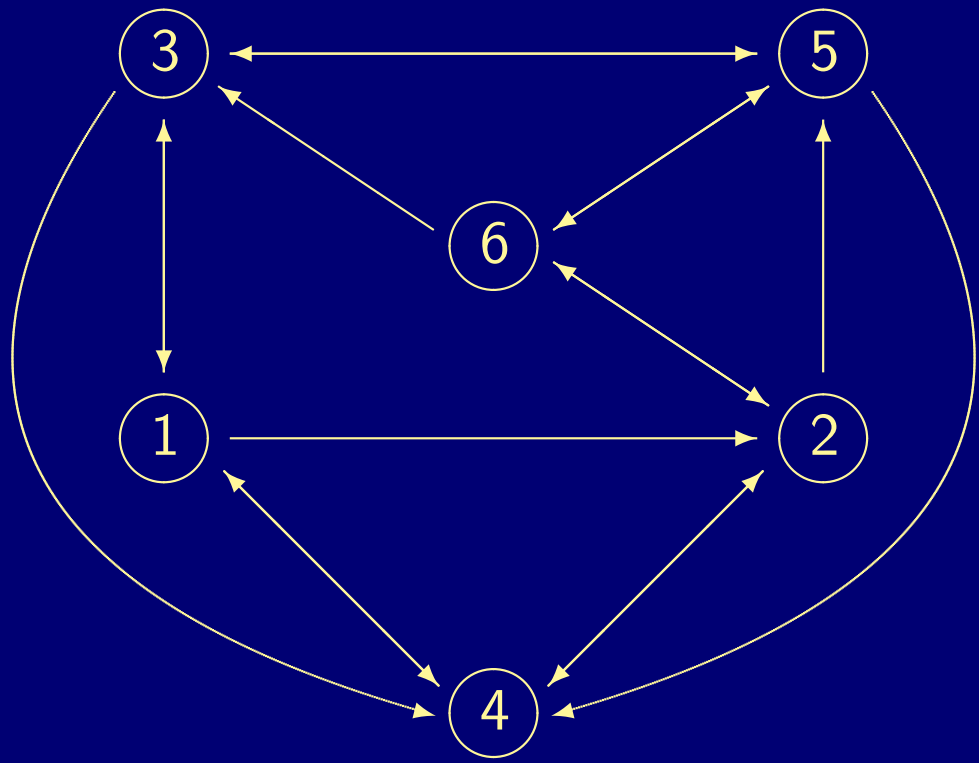


Hamiltonian Cycle

Automated (Intelligent) Grounding

```
gringo --text graph.lp cycle.lp
```

```
node(1).  ...  node(6).
edge(1,2). edge(1,3). edge(1,4). ...
edge(6,2). edge(6,3). edge(6,5).

1 #count{ cycle(3,1), cycle(4,1) } 1.
1 #count{ cycle(1,2), cycle(1,3),
                      cycle(1,4) } 1.
...
1 #count{ cycle(2,6), cycle(5,6) } 1.
1 #count{ cycle(6,2), cycle(6,3),
                      cycle(6,5) } 1.

reach(1).
reach(2) :- cycle(1,2).
reach(2) :- reach(4), cycle(4,2).
reach(2) :- reach(6), cycle(6,2).
...
:- not reach(2).  ...  :- not reach(6).
```

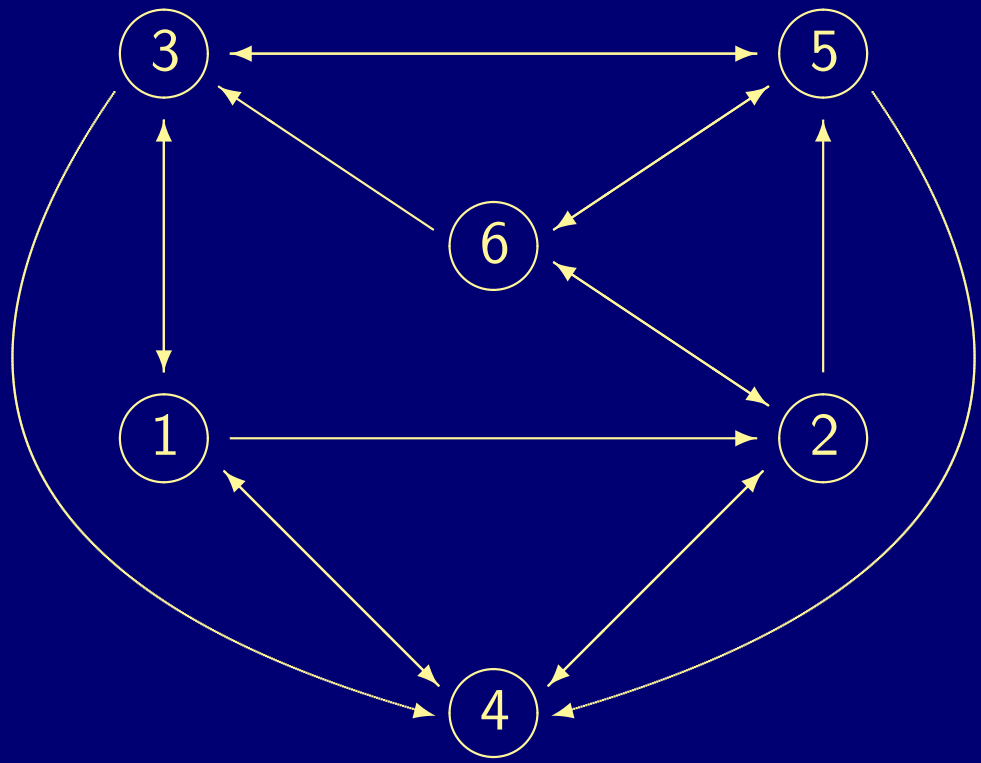


Hamiltonian Cycle

Automated (Intelligent) Grounding

```
gringo --text graph.lp cycle.lp
```

```
1 2 0 0
1 3 0 0
1 4 0 0
...
3 3 26 27 28 1 0 29
2 30 3 0 1 26 27 28
1 1 2 1 30 29
2 31 3 0 2 26 27 28
1 1 2 0 31 29
1 29 0 0
...
0
26 cycle(6,5)
27 cycle(6,3)
28 cycle(6,2)
...
0
```

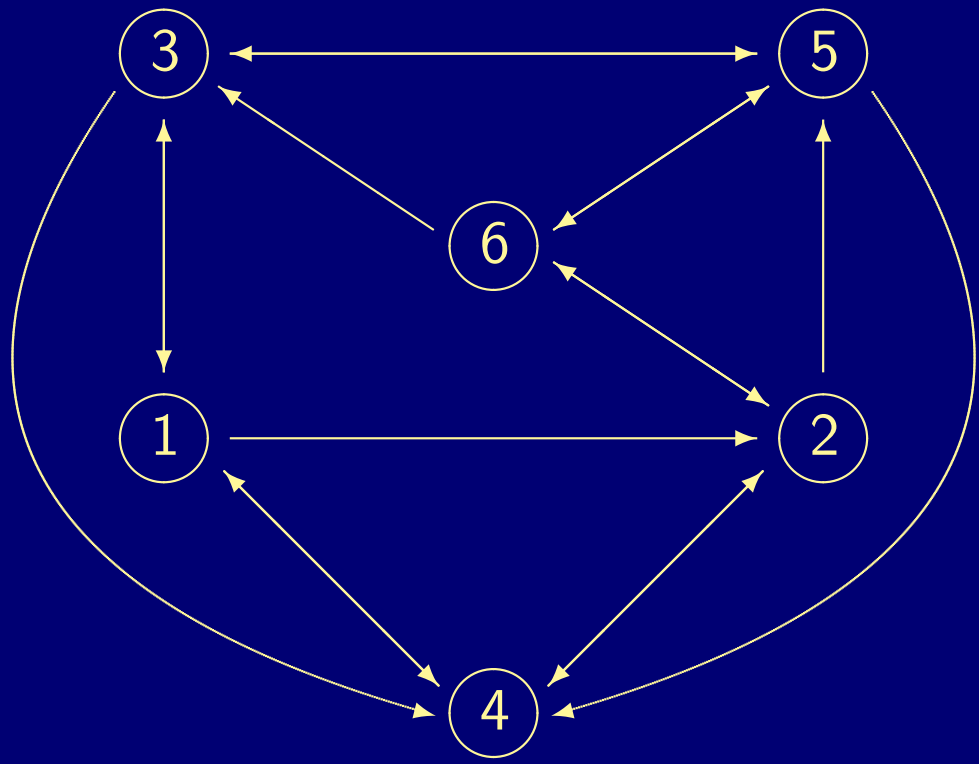


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```

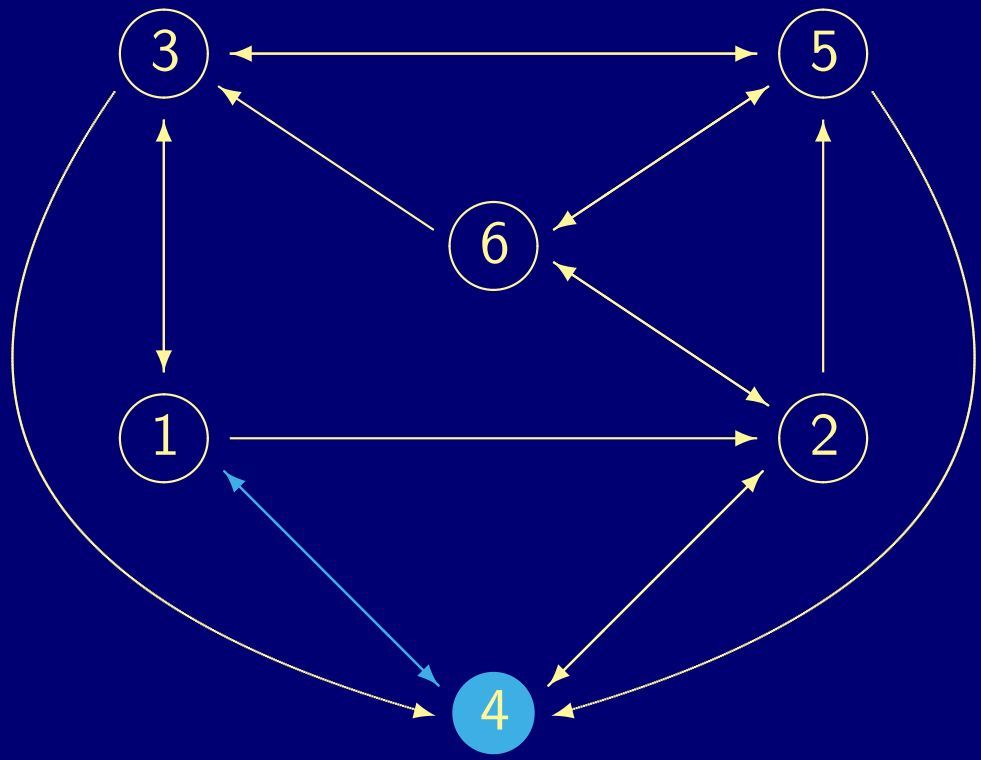


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```

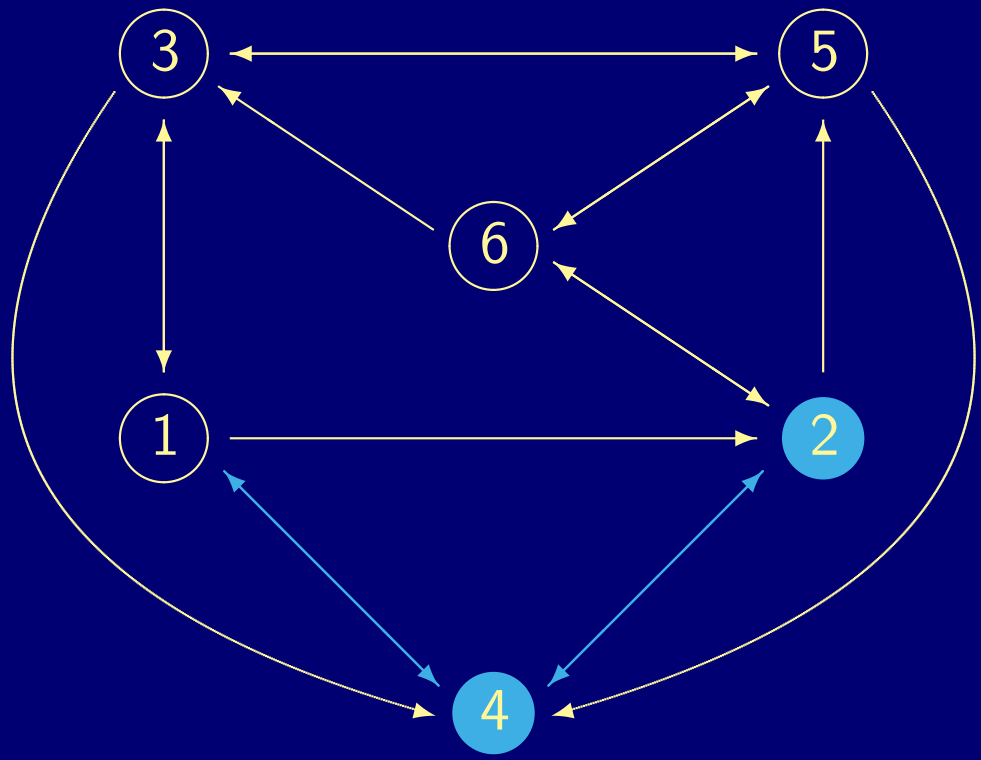


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```

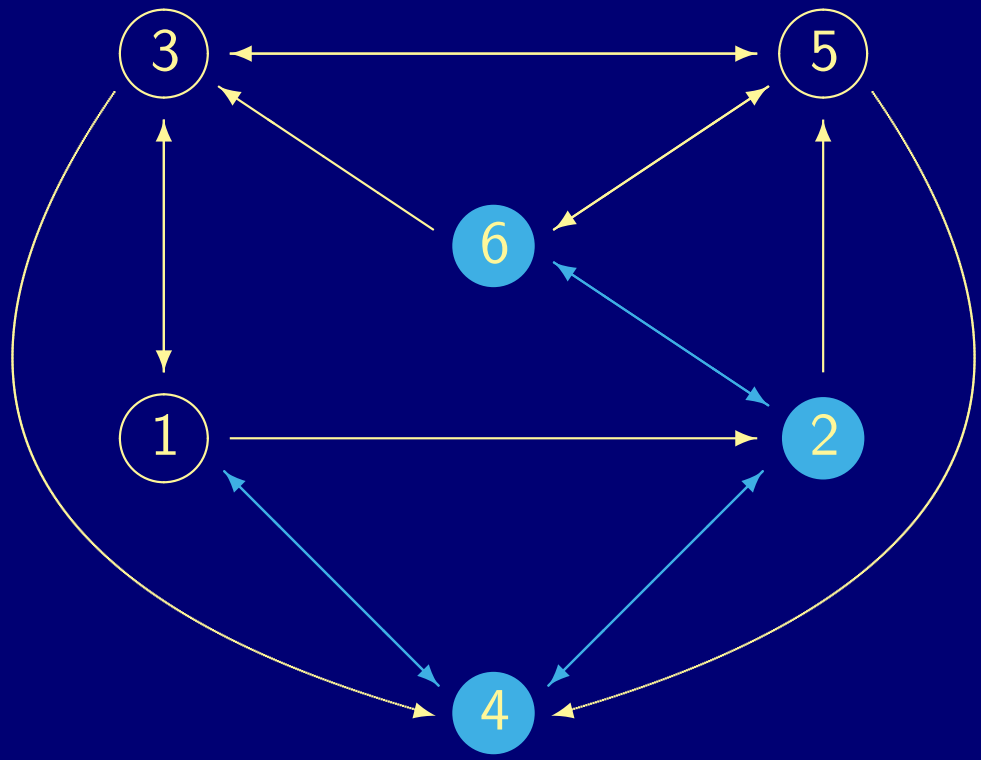


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```

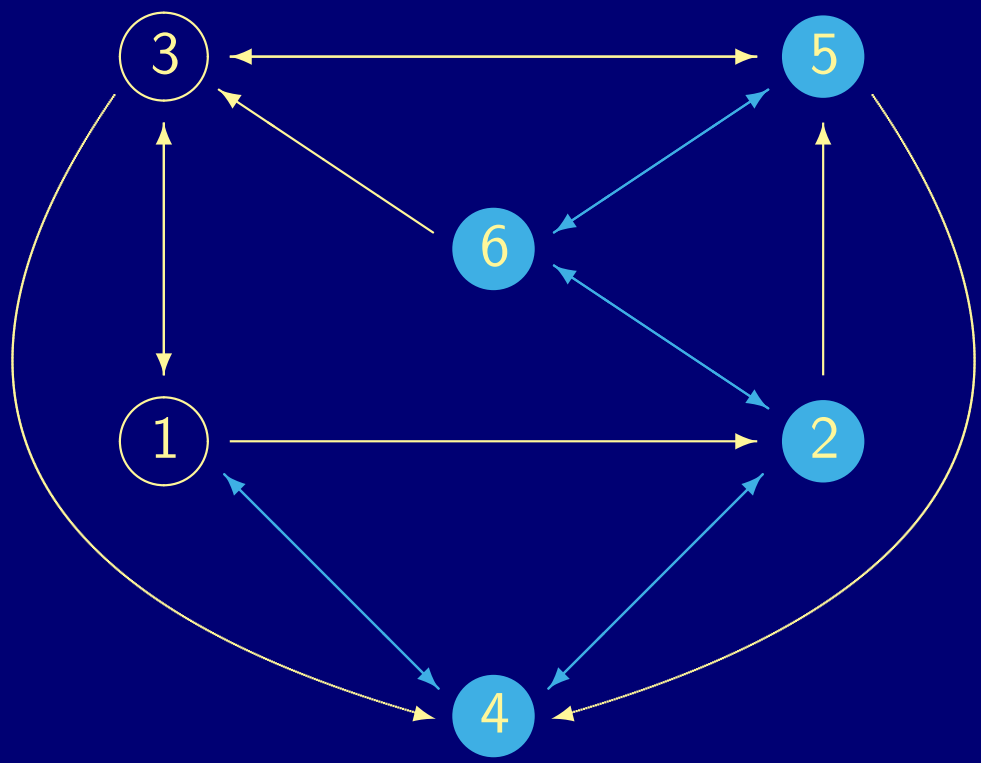


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```

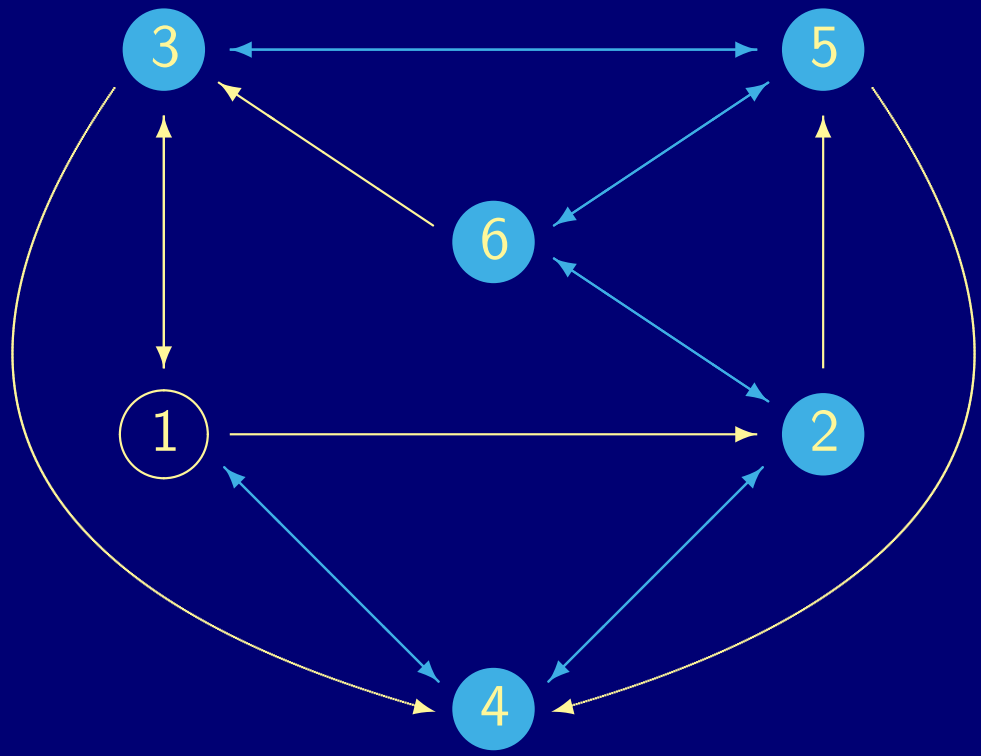


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```

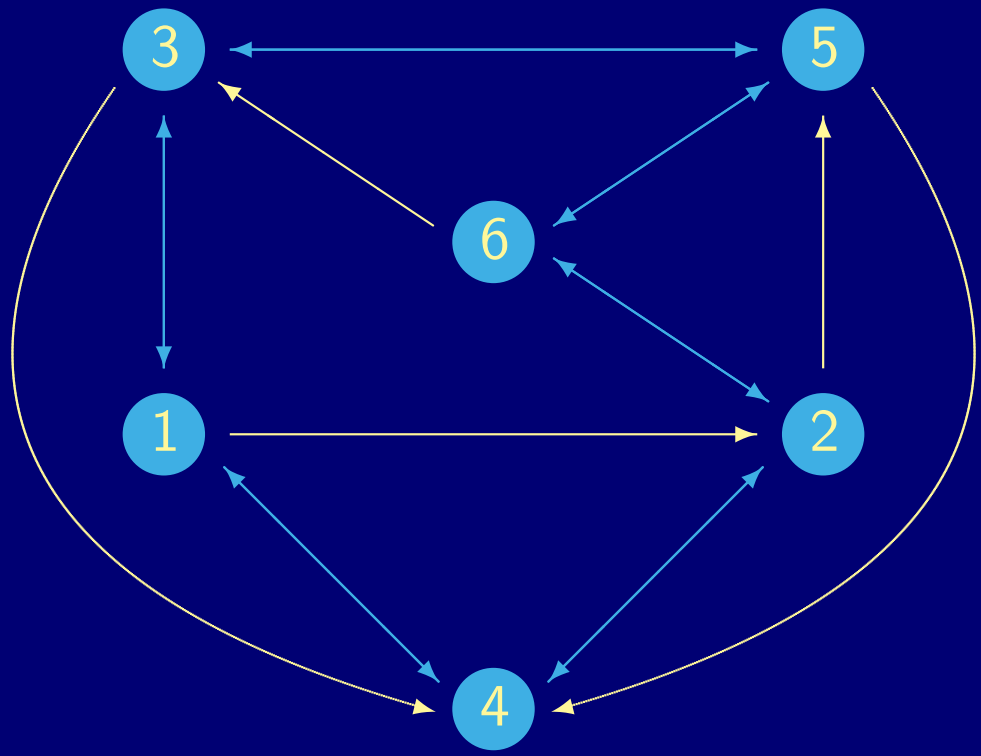


Hamiltonian Cycle

Automated (Conflict-Driven) Search

```
gringo graph.lp cycle.lp | clasp
```

```
clasp version 2.0.4
Reading from stdin
Solving...

Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6)
cycle(6,5) cycle(5,3) cycle(3,1)
SATISFIABLE

Models       : 1+
Time         : 0.001s
CPU Time     : 0.000s
```



Hamiltonian Cycle

(Conflict-Driven) Enumeration in Polynomial Space

```
gringo graph.lp cycle.lp | clasp 0
```

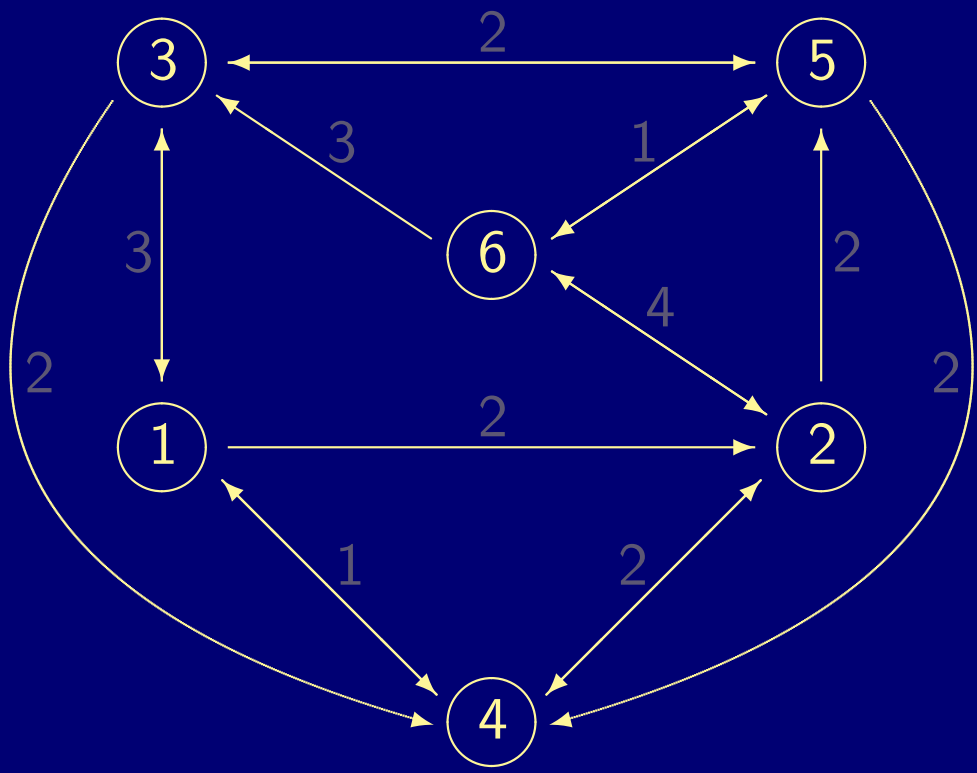
```
Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
Answer: 2
cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)
Answer: 3
cycle(1,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,4) cycle(4,1)
Answer: 4
cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
Answer: 5
cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
Answer: 6
cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
SATISFIABLE

Models       : 6
Time         : 0.001s
CPU Time     : 0.000s
```

Hamiltonian Cycle

Problem Instance as Facts

A directed graph (graph.lp)

```
node(1).  node(2).  node(3).
node(4).  node(5).  node(6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).
```

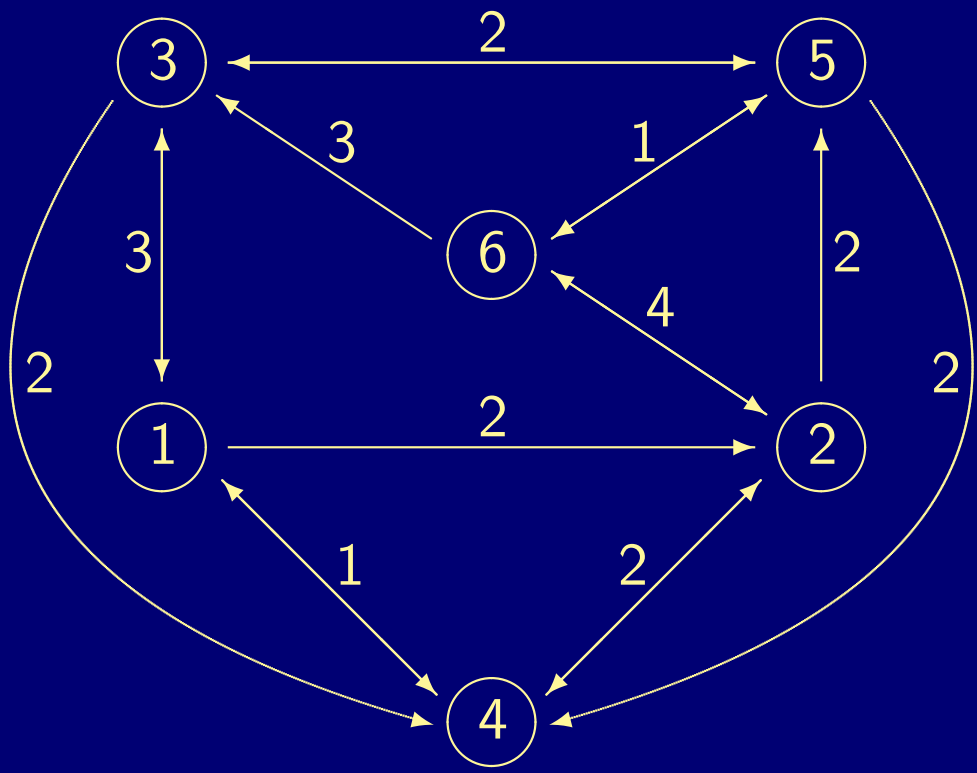


Traveling Salesperson

Problem Instance as Facts

A directed graph with edge costs (`costs.lp`)

```
cost(1,2,2).
cost(1,3,3). cost(3,1,3).
cost(1,4,1). cost(4,1,1).
cost(2,4,2). cost(4,2,2).
cost(2,5,2).
cost(2,6,4). cost(6,2,4).
cost(3,4,2).
cost(3,5,2). cost(5,3,2).
cost(5,4,2).
cost(5,6,1). cost(6,5,1).
cost(6,3,3).
```



Hamiltonian Cycle

Problem Encoding

(Logical) Recipe: Generate + Define + Test (`cycle.lp`)

```
% each node has exactly one incoming and outgoing edge
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).

% determine all nodes reachable from some first node
reach(X) :- X = #min[ node(Y) = Y ].
reach(Y) :- reach(X), cycle(X,Y).

% exclude existence of any unreached node
:- node(Y), not reach(Y).
```

Traveling Salesperson

Problem Encoding

(Logical) Recipe: Generate + Define + Test + Optimize (`price.lp`)

```
% each node has exactly one incoming and outgoing edge
...
% determine all nodes reachable from some first node
...
% exclude existence of any unreached node
...

% search for cycle with minimum sum of edge costs
#minimize[ cycle(X,Y) = C : cost(X,Y,C) ].
```

Traveling Salesperson

Automated (Conflict-Driven) Optimization

```
gringo graph.lp costs.lp cycle.lp price.lp | clasp 0
```

```
Answer: 1
cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
Optimization: 13
Answer: 2
cycle(1,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,4) cycle(4,1)
Optimization: 12
Answer: 3
cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
Optimization: 11
OPTIMUM FOUND

Models       : 1
Time         : 0.001s
CPU Time     : 0.000s
```

Answer Set Programming =

Knowledge Representation + Database Techniques + Search

What we do @ Potsdam

1. grounding `gringo`
   first-order variables, aggregates, deduction
2. solving `clasp`
   conflict-driven search, enumeration, optimization
3. more `http://potassco.sourceforge.net`

What you do

1. modeling
   first-order problem encoding + instances as facts
2. push the button

Answer Set Programming =

Knowledge Representation + Database Techniques + Search

What we do @ Potsdam

1. grounding — `gringo`
   - first-order variables, aggregates, deduction
2. solving — `clasp`
   - conflict-driven search, enumeration, optimization
3. more — `http://potassco.sourceforge.net`

What you do

1. modeling
   - first-order problem encoding + instances as facts
2. push the button

Answer Set Programming =

Knowledge Representation + Database Techniques + Search

What we do @ Potsdam

1. grounding `gringo`
   - first-order variables, aggregates, deduction
2. solving `clasp`
   - conflict-driven search, enumeration, optimization
3. more `http://potassco.sourceforge.net`

What you do

1. modeling

   first-order problem encoding + instances as facts
2. push the button

Answer Set Programming =

Knowledge Representation + Database Techniques + Search

What we do @ Potsdam

1. grounding `gringo`
   - first-order variables, aggregates, deduction
2. solving `clasp`
   - conflict-driven search, enumeration, optimization
3. more `http://potassco.sourceforge.net`

What you do

1. modeling

   first-order problem encoding + instances as facts
2. push the button

Answer Set Programming =

Knowledge Representation + Database Techniques + Search

What we do @ Potsdam

1. grounding `gringo`
   - first-order variables, aggregates, deduction
2. solving `clasp`
   - conflict-driven search, enumeration, optimization
3. more `http://potassco.sourceforge.net`

What you do

1. modeling
   - first-order problem encoding + instances as facts
2. push the button

Answer Set Programming =

Knowledge Representation + Database Techniques + Search

What we do @ Potsdam

1. grounding `gringo`
   - first-order variables, aggregates, deduction
2. solving `clasp`
   - conflict-driven search, enumeration, optimization
3. more `http://potassco.sourceforge.net`

What you do

1. modeling
   - first-order problem encoding + instances as facts
2. push the button

Outline

Background

The challenge

- large distributions (more than 50000 packages)
- hard constraints (conflicts, dependencies, installation requests)
- user preferences (with different priorities)

☞ powerful search methods needed for optimal configuration

The honor

Mancoosi International Solver Competition (MISC)

Many thanks to Ralf Treinen and colleagues for

organizing MISC and

posing challenging multi-criteria optimization problems!

Background

The challenge

- large distributions (more than 50000 packages)
- hard constraints (conflicts, dependencies, installation requests)
- user preferences (with different priorities)

☞ powerful search methods needed for optimal configuration

The honor

- Mancoosi International Solver Competition (MISC)

☞ Many thanks to Ralf Treinen and colleagues for

- organizing MISC and
- posing challenging multi-criteria optimization problems!

Background

The challenge

- large distributions (more than 50000 packages)
- hard constraints (conflicts, dependencies, installation requests)
- user preferences (with different priorities)

☞ powerful search methods needed for optimal configuration

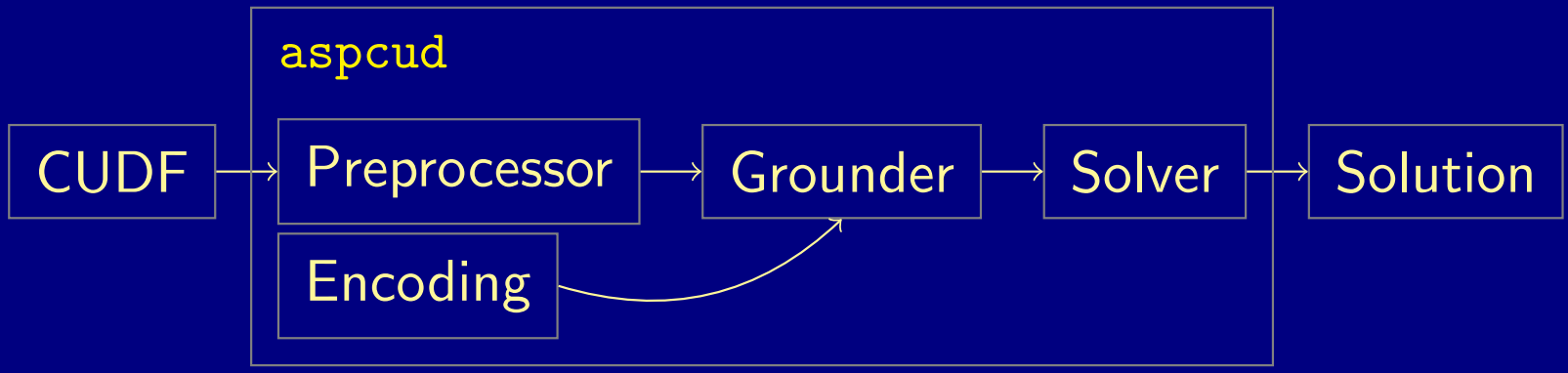
The honor

- Mancoosi International Solver Competition (MISC)

☞ Many thanks to Ralf Treinen and colleagues for

- organizing MISC and
- posing challenging multi-criteria optimization problems!

ASP-Based Package Configuration



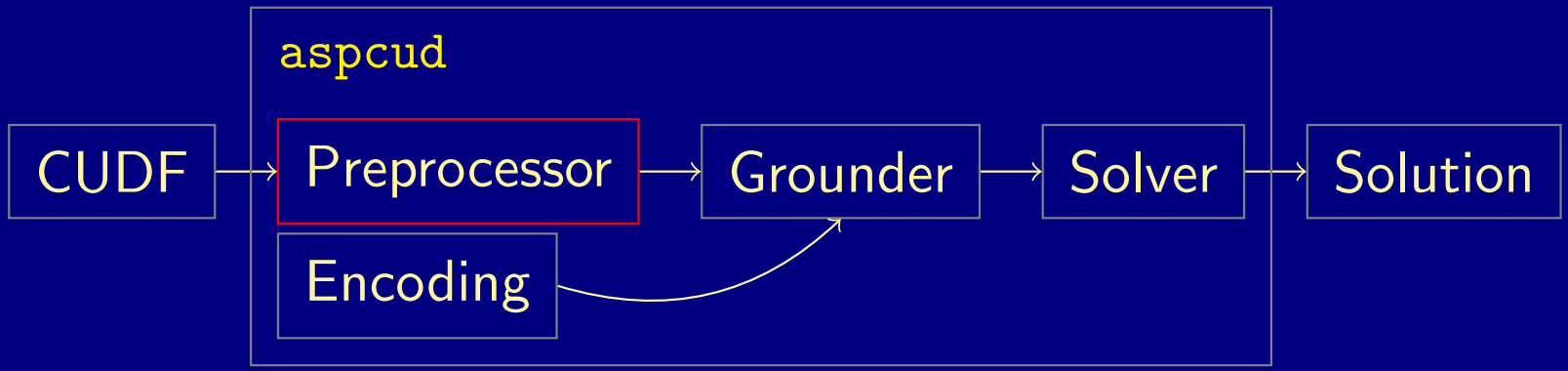
Preprocessor converts CUDF input to facts

Encoding first-order problem specification

Grounder instantiates first-order variables

Solver searches for (optimal) answer sets

ASP-Based Package Configuration



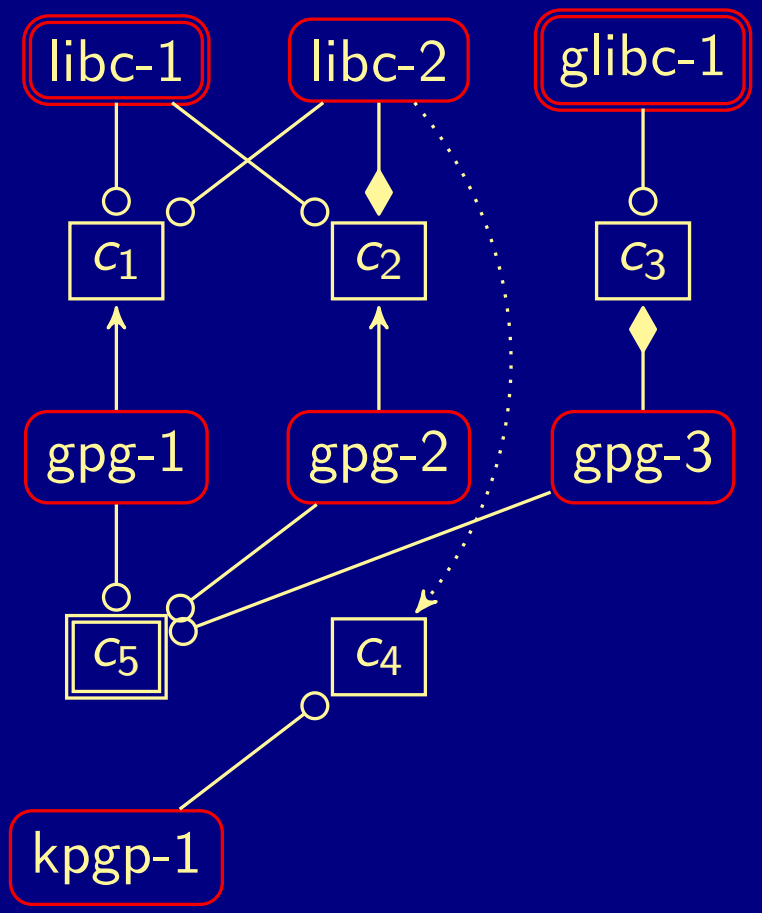
Preprocessor converts CUDF input to facts

Encoding first-order problem specification

Grounder instantiates first-order variables

Solver searches for (optimal) answer sets

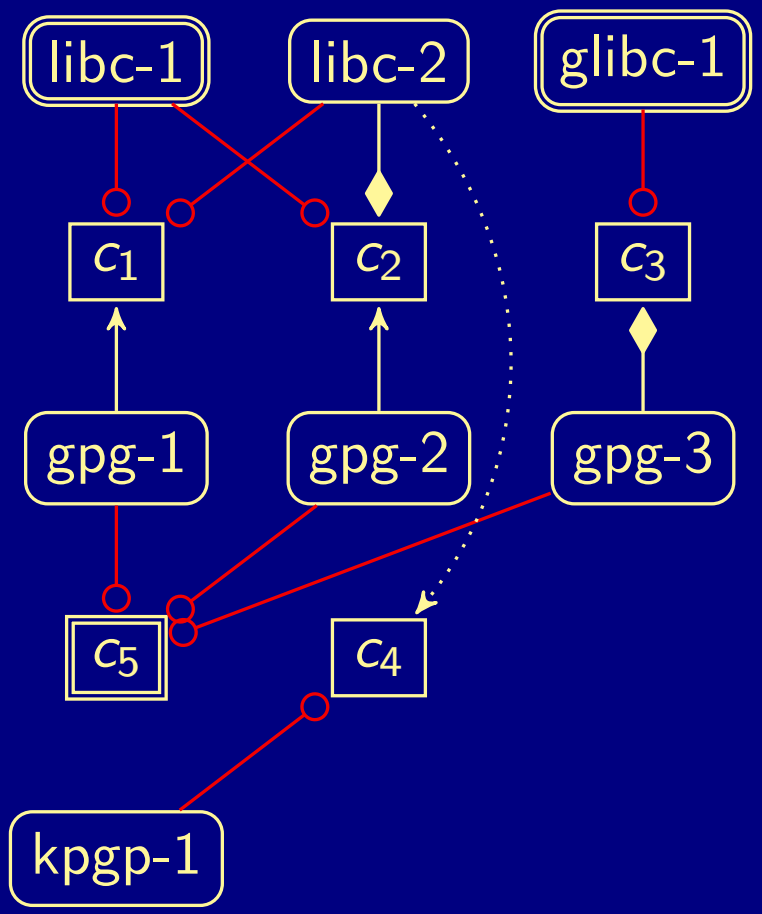
Instance Format



Installable Packages

```
package(libc,1).
package(libc,2).

package(glibc,1).

package(gpg,1).
package(gpg,2).
package(gpg,3).

package(kpgp,1).
```

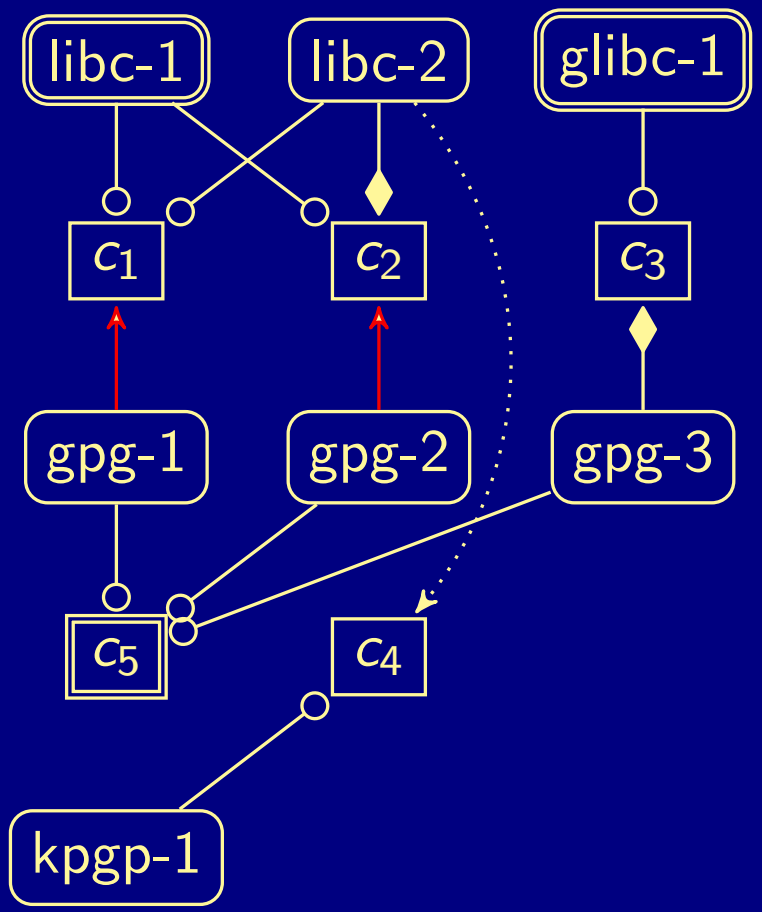
Instance Format



Package Clauses

```
satisfies(libc,1,c1).
satisfies(libc,1,c2).
satisfies(libc,2,c1).

satisfies(glibc,1,c3).

satisfies(gpg,1,c5).
satisfies(gpg,2,c5).
satisfies(gpg,3,c5).

satisfies(kpgp,1,c4).
```

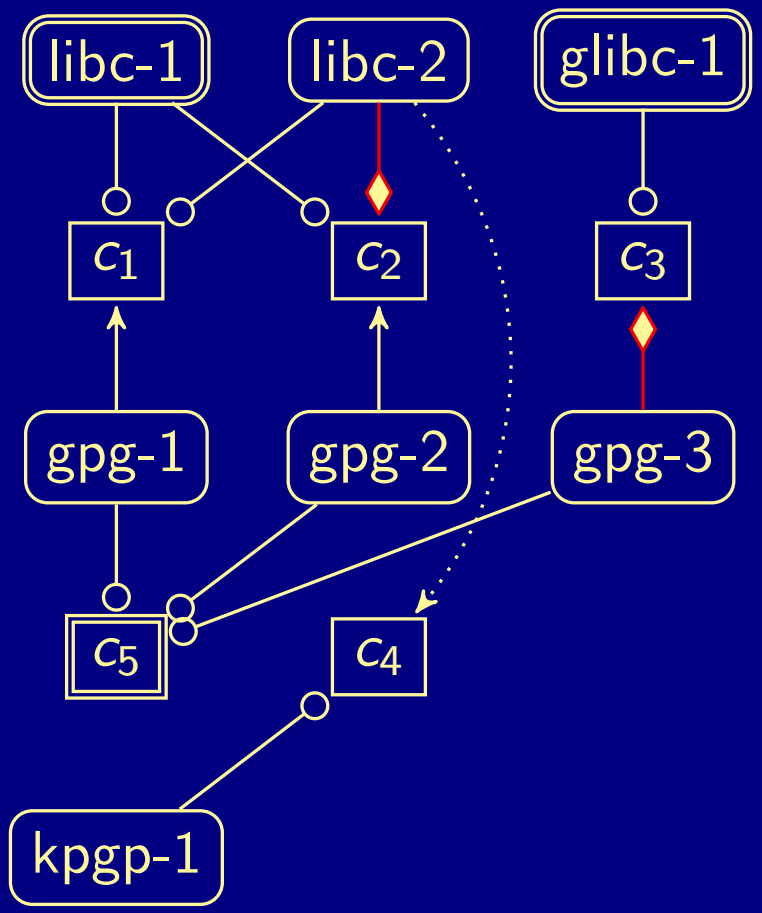
Instance Format



Package Dependencies

```
depends(gpg,1,c1).
depends(gpg,2,c2).
```

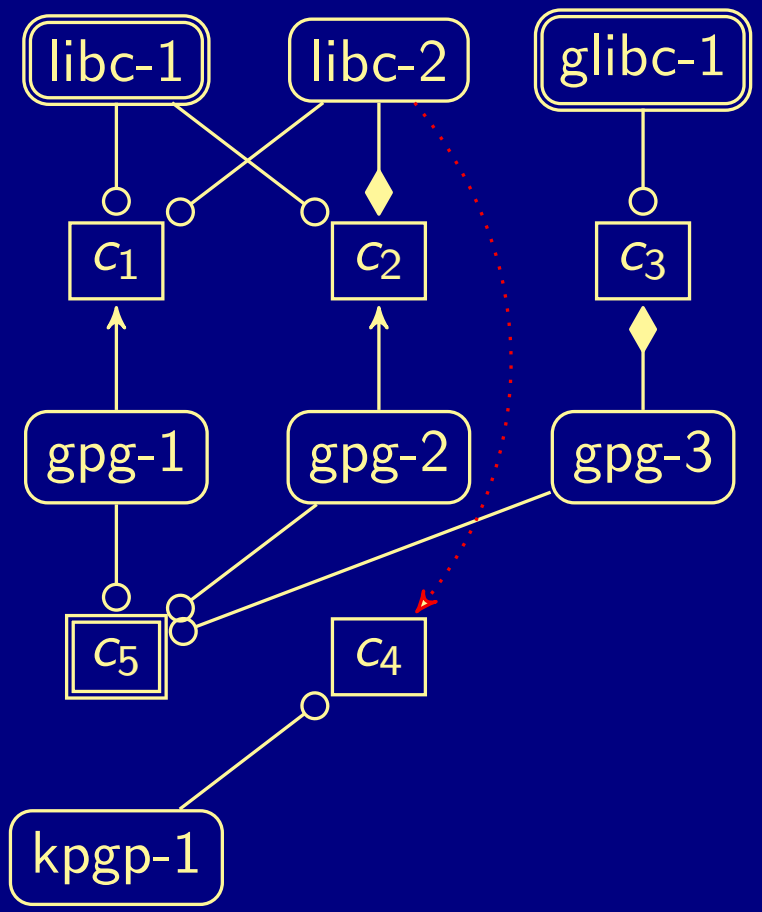
Instance Format



Package Conflicts

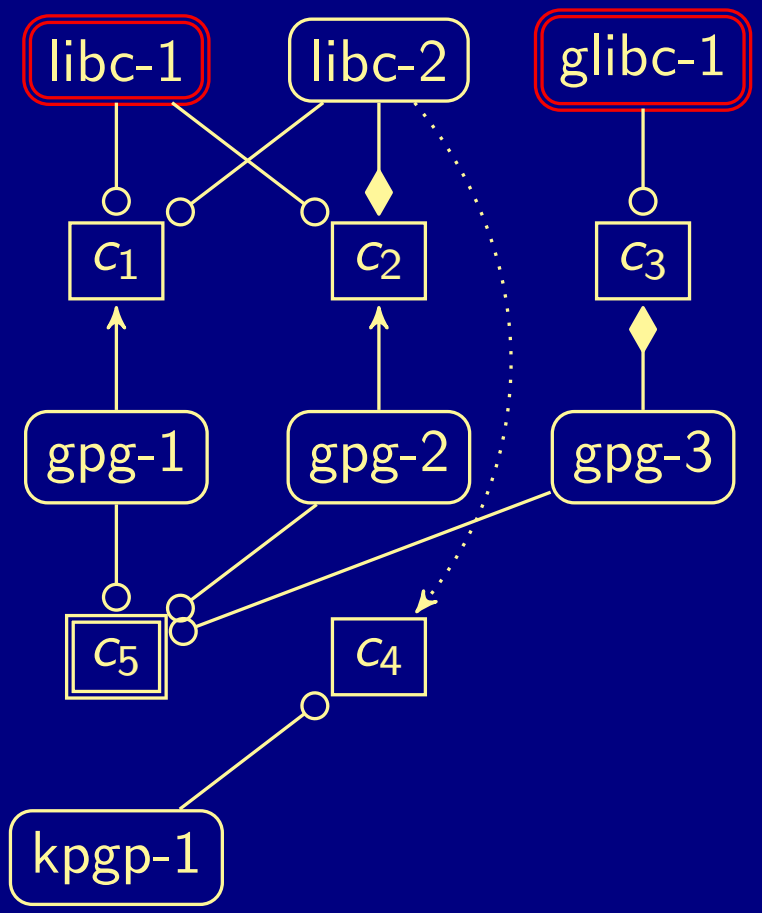
```
conflicts(libc,2,c2).

conflicts(gpg,3,c3).
```

Instance Format



Package Recommendations

```
recommends(libc,2,c4).
```

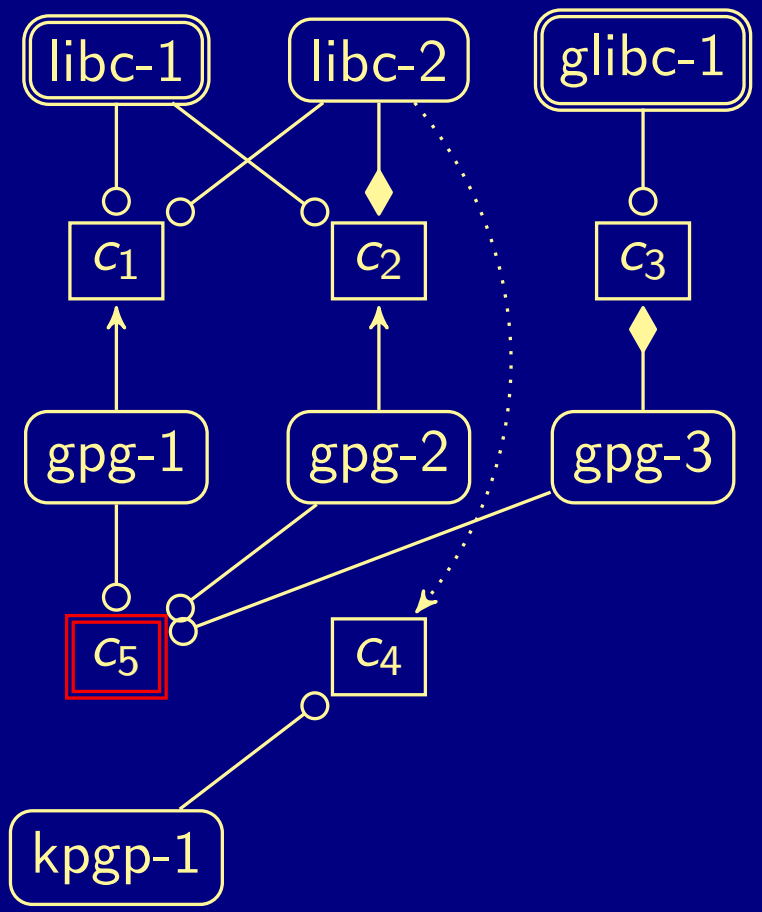


Instance Format

Installed Packages

```
installed(libc,1).

installed(glibc,1).
```

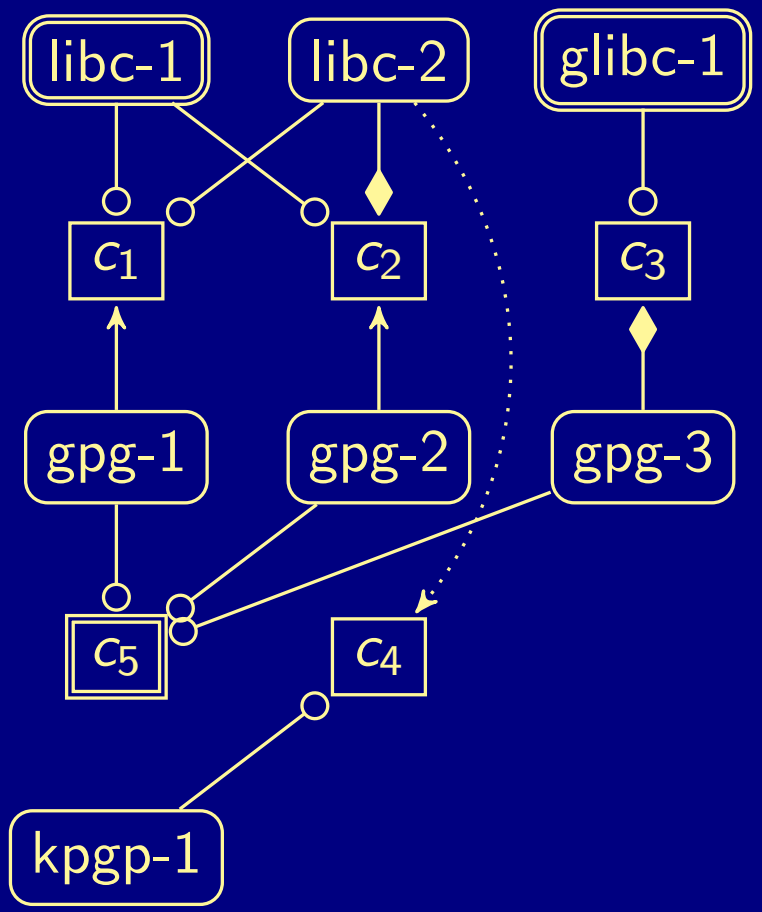
Instance Format



Installation Requests

```
request(c5).
```

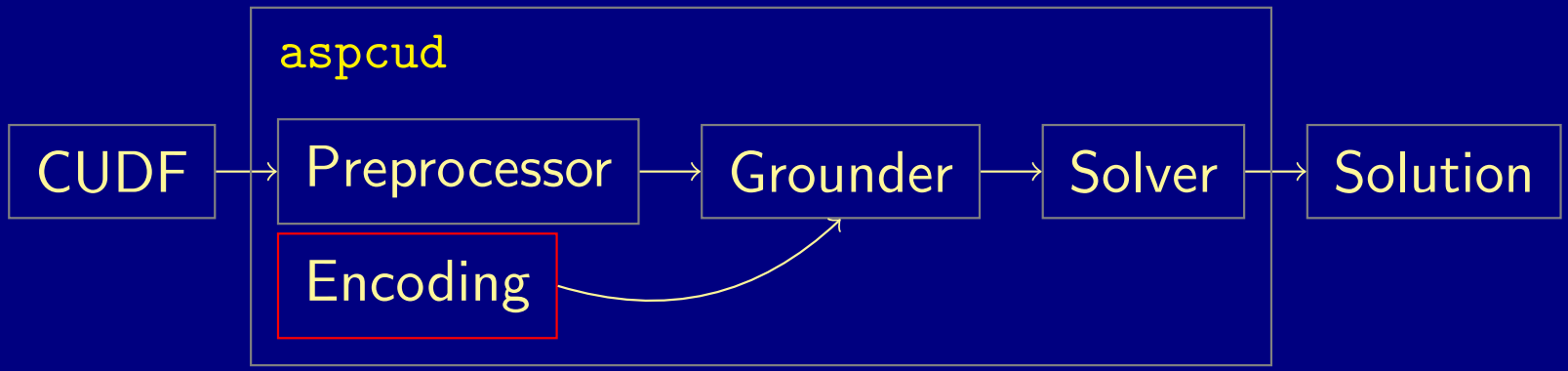
Instance Format



Optimization Criteria

```
utility(delete,-1).
utility(change,-2).
```

ASP-Based Package Configuration



Preprocessor converts CUDF input to facts

Encoding first-order problem specification

Grounder instantiates first-order variables

Solver searches for (optimal) answer sets

Problem Encoding

Hard Constraints

```
% choose packages to install
{ install(N,V) } :- package(N,V).

% derive required clauses
exclude(C) :- install(N,V), conflicts(N,V,C).
include(C) :- install(N,V), depends(N,V,C).
% derive satisfied clauses
satisfy(C) :- install(N,V), satisfies(N,V,C).

% assert required clauses to be (un)satisfied
:- exclude(C),     satisfy(C).
:- include(C), not satisfy(C).
:- request(C), not satisfy(C).
```

Problem Encoding

Hard Constraints

```
% choose packages to install
{ install(N,V) } :- package(N,V).

% derive required clauses
exclude(C) :- install(N,V), conflicts(N,V,C).
include(C) :- install(N,V), depends(N,V,C).
% derive satisfied clauses
satisfy(C) :- install(N,V), satisfies(N,V,C).

% assert required clauses to be (un)satisfied
:- exclude(C),     satisfy(C).
:- include(C), not satisfy(C).
:- request(C), not satisfy(C).
```

Problem Encoding

Hard Constraints

```
% choose packages to install
{ install(N,V) } :- package(N,V).

% derive required clauses
exclude(C) :- install(N,V), conflicts(N,V,C).
include(C) :- install(N,V), depends(N,V,C).
% derive satisfied clauses
satisfy(C) :- install(N,V), satisfies(N,V,C).

% assert required clauses to be (un)satisfied
:- exclude(C),     satisfy(C).
:- include(C), not satisfy(C).
:- request(C), not satisfy(C).
```

Problem Encoding

Soft Constraints

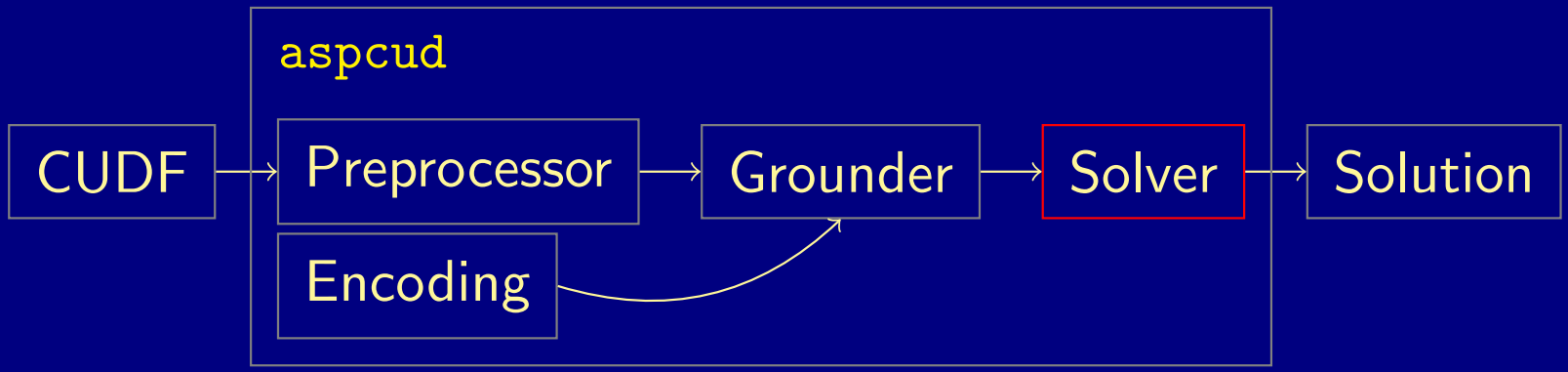
```
% auxiliary definitions
install(N)   :- install(N,V).
installed(N) :- installed(N,V).

% derive optimization criteria violations
violate(newpkg,N) :- utility(newpkg,L),
                     install(N), not installed(N).
violate(delete,N) :- utility(delete,L),
                     installed(N), not install(N).
% similar for other criteria
...

% impose soft constraints
#minimize[ violate(U,N) = 1 @ -L : utility(U,L) : L < 0 ].
#maximize[ violate(U,N) = 1 @  L : utility(U,L) : L > 0 ].
```

Problem Encoding

Soft Constraints

```
% auxiliary definitions
install(N)   :- install(N,V).
installed(N) :- installed(N,V).

% derive optimization criteria violations
violate(newpkg,N) :- utility(newpkg,L),
                     install(N), not installed(N).
violate(delete,N) :- utility(delete,L),
                     installed(N), not install(N).
% similar for other criteria
...

% impose soft constraints
#minimize[ violate(U,N) = 1 @ -L : utility(U,L) : L < 0 ].
#maximize[ violate(U,N) = 1 @  L : utility(U,L) : L > 0 ].
```

(Multi-Criteria) Optimization Approaches



Optimization via satisfiability — `clasp`

- decrease upper bounds (costs) w.r.t. witnesses
- proceed to next criterion upon unsatisfiability

Optimization via unsatisfiability — `unclasp`

- weaken soft constraints w.r.t. unsatisfiable subsets
- proceed to next criterion upon satisfiability

Outline

Summary and Outlook

ASP = Knowledge Representation + Database Techniques + Search

- first-order input languages
- flexible reasoning engines
- ☞ high degree of automation

It's for free

- `http://potassco.sourceforge.net`
- `http://www.cs.uni-potsdam.de/~torsten/ijcai11tutorial/`

Future challenges

solver autoconfiguration; parallelism; automatic encoding optimization; "software engineering" tools; non-Boolean variables; stream reasoning; ...

Summary and Outlook

ASP = Knowledge Representation + Database Techniques + Search

- first-order input languages
- flexible reasoning engines
- ☞ high degree of automation

It's for free

- `http://potassco.sourceforge.net`
- `http://www.cs.uni-potsdam.de/~torsten/ijcai11tutorial/`

Future challenges

solver autoconfiguration; parallelism; automatic encoding optimization; "software engineering" tools; non-Boolean variables; stream reasoning; ...

Summary and Outlook

ASP = Knowledge Representation + Database Techniques + Search

- first-order input languages
- flexible reasoning engines
- ☞ high degree of automation

It's for free

- `http://potassco.sourceforge.net`
- `http://www.cs.uni-potsdam.de/~torsten/ijcai11tutorial/`

Future challenges

solver autoconfiguration; parallelism; automatic encoding optimization; "software engineering" tools; non-Boolean variables; stream reasoning; ...