Answer Set Solving in Practice

Torsten Schaub
University of Potsdam
torsten@cs.uni-potsdam.de

Advanced Modeling: Overview

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements

  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Outline

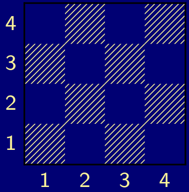
N-Queens Problem

Problem Specification

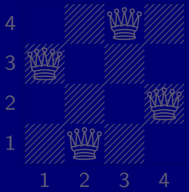
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally)

$N = 4$

Chessboard



Placement



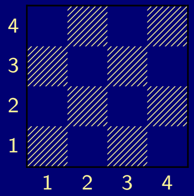
N-Queens Problem

Problem Specification

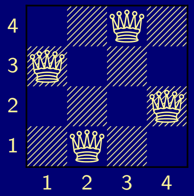
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally)

$N = 4$

Chessboard



Placement



A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X,Y'), Y < Y'.

% DISPLAY
#show queen/2.
```

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of queen in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X,Y'), Y < Y'.

% DISPLAY
#show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of queen in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X',Y), X < X'.

% DISPLAY
#show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.

% DISPLAY
#show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#show queen/2.
```

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

Anything missing?

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model
4. We have to place (at least) *N* queens

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 8 Queens!

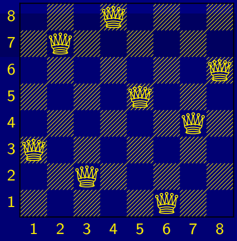
```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

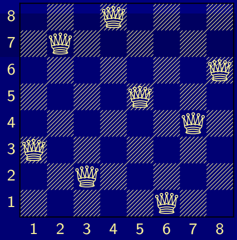



```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats

Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```



A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models       : 1+
Time         : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time     : 147.480s
Choices      : 594960
Conflicts    : 574565
Restarts     : 19

Variables    : 17271
Constraints  : 16787
```

Potassco

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models      : 1+
Time        : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time    : 147.480s
Choices     : 594960
Conflicts   : 574565
Restarts    : 19

Variables   : 17271
Constraints : 16787
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X,Y'), Y < Y'.
:- queen(X,Y), queen(X',Y), X < X'.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y), X < X'.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.


% DISPLAY
#show queen/2.
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models      : 1+
Time        : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.020s
Choices     : 132
Conflicts   : 105
Restarts    : 1

Variables   : 7238
Constraints : 6710
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models      : 1+
Time        : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.020s
Choices     : 132
Conflicts   : 105
Restarts    : 1

Variables   : 7238
Constraints : 6710
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models       : 1+
Time         : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time     : 6.930s
Choices      : 1373
Conflicts    : 845
Restarts     : 4

Variables    : 1211338
Constraints  : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | clasp --stats

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                      O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                      O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                         O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                         O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.           O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                        O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                        O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.             O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                      O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                       O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                          O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                          O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.          O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                          O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                          O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.               O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                        O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                        O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.             O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ~ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                    O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                     O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                     O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.          O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ~ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                              O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                              O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                                 O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                                 O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.                   O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                          O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                          O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.          O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                       O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                       O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                          O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                          O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

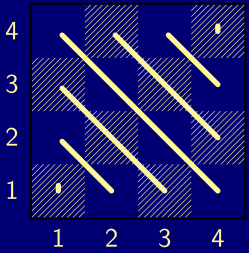
queens_1.lp

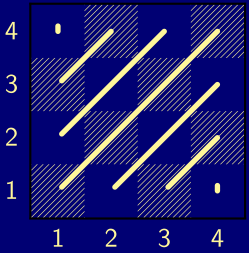
```
% DOMAIN
#const n=4. square(1..n,1..n).                                        O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                        O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.      O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Diagonals make trouble!

Enumerating Diagonals

$N = 4$





$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \dots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \dots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating Diagonals

$N = 4$

	1	2	3	4
4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

	1	2	3	4
4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

Potassco

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \dots, (2{*}N){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.


% DISPLAY
#show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1


% DISPLAY
#show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models       : 1+
Time         : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time     : 0.210s
Choices      : 11036
Conflicts    : 499
Restarts     : 3

Variables    : 16098
Constraints  : 970
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models       : 1+
Time         : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time     : 0.210s
Choices      : 11036
Conflicts    : 499
Restarts     : 3

Variables    : 16098
Constraints  : 970
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

Potassco

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_3.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time    : 7.320s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

Potassco

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models       : 1+
Time         : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time     : 68.620s
Choices      : 869379
Conflicts    : 25746
Restarts     : 12

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,90) queen(2,452) queen(3,494) queen(4,145) queen(5,84) ...
SATISFIABLE

Models      : 1+
Time        : 22.654s (Solving: 10.53s 1st Model: 10.47s Unsat: 0.00s)
CPU Time    : 15.750s
Choices     : 1058729
Conflicts   : 2128
Restarts    : 6

Variables   : 403123
Constraints : 49636
```

Outline

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).
```

Potassco

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap). pro(cucumber,cheap).
pro(asparagus,fresh). pro(cucumber,fresh).
pro(asparagus,tasty). pro(cucumber,tasty).
```

```
buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty), pro(X,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).
pro(asparagus,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap). pro(cucumber,cheap). pre(cheap).
pro(asparagus,fresh). pro(cucumber,fresh). pre(fresh).
pro(asparagus,tasty). pro(cucumber,tasty). pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).                          pre(clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Potassco

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).
```

Potassco

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).    pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).    pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).    pre(tasty).
pro(asparagus,clean).                            pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ . . . obsolete if properties change ✘
2. use variable-sized conjunction (via ':') . . . adapts to changing facts ✔
3. use negation of complement . . . adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).                          pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Running Example: Latin Square

Given: an $N \times N$ board

1						
2						
3						
4						
5						
6						
	1	2	3	4	5	6

represented by facts:

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

Wanted: assignment of $1, \ldots, N$

1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5
	1	2	3	4	5	6

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Running Example: Latin Square

Given: an $N \times N$ board

1						
2						
3						
4						
5						
6						
	1	2	3	4	5	6

represented by facts:

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

Wanted: assignment of $1, \dots, N$

1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5
	1	2	3	4	5	6

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X,Y), N = 1..n, not num(X,Y',N) : square(X,Y').
:- square(X,Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X,Y), N = 1..n, not num(X,Y',N) : square(X,Y').
:- square(X,Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X,Y), N = 1..n, not num(X,Y',N) : square(X,Y').
:- square(X,Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X), N = 1..n, not num(X,Y',N) : square(X,Y').
:- squareY(Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X), N = 1..n, not num(X,Y',N) : square(X,Y').
:- squareY(Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

gringo latin_0.lp | wc	gringo latin_1.lp | wc
105480 2558984 14005258	42056 273672 1690522

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y != Y'.
:- num(X,Y,N), num(X',Y,N), X != X'.
```

- Note duplicate ground rules
  (swapping Y/Y' and X/X' gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y != Y'.
:- num(X,Y,N), num(X',Y,N), X != X'.
```

- Note duplicate ground rules
  (swapping `Y/Y'` and `X/X'` gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y != Y'.
:- num(X,Y,N), num(X',Y,N), X != X'.
```

- Note duplicate ground rules
  (swapping `Y/Y'` and `X/X'` gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y <  Y'.
:- num(X,Y,N), num(X',Y,N), X <  X'.
```

- Note duplicate ground rules
  (swapping `Y/Y'` and `X/X'` gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y <  Y'.
:- num(X,Y,N), num(X',Y,N), X <  X'.
```

- Note duplicate ground rules
  (swapping Y/Y' and X/X' gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y < Y'.
:- num(X,Y,N), num(X',Y,N), X < X'.
```

- Note uniqueness of N in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
```

```
228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y < Y'.
:- num(X,Y,N), num(X',Y,N), X < X'.
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y < Y'.
:- num(X,Y,N), num(X',Y,N), X < X'.
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of N in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of N in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.                              ✔

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc                gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C { num(X,Y,N) } C, C = 0..n.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C { num(X,Y,N) } C, C = 0..n.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

- Note internal transformation by gringo

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.     ✘
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.     ✘
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

Potassco

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.



% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

Potassco

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

- Note many symmetric solutions (mirroring, rotation, value permutation)

Potassco

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

- Note easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

- Note easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

- Note Let's compare enumeration speed!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.



% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.



% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

```
Models : 161280    Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 161280     Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 1344        Time : 0.024s
```

Outline

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Potassco

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic . . . follow error messages by the grounder
- semantic . . . (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo --text`)
  - if stable models are unintended, investigate conditions that fail to hold
  - if stable models are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with “interesting features”
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo --text`)
  - if stable models are unintended, investigate conditions that fail to hold
  - if stable models are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo --text`)
  - if stable models are unintended, investigate conditions that fail to hold
  - if stable models are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic . . . follow error messages by the grounder
- semantic . . . (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo --text`)
   - if stable models are unintended, investigate conditions that fail to hold
   - if stable models are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo --text`)
   - if stable models are unintended, investigate conditions that fail to hold
   - if stable models are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo --text`)
   - if stable models are unintended, investigate conditions that fail to hold
   - if stable models are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

  if great search efforts (`Conflicts`/`Choices`/`Restarts`), then

  try prefabricated settings (using `clasp` option '`--configuration`')
  try auto-configuration (offered by `claspfolio` or `accclingo`)
  try manual fine-tuning (requires expert knowledge!)
  if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  - try prefabricated settings (using `clasp` option '`--configuration`')
  - try auto-configuration (offered by `claspfolio` or `accclingo`)
  - try manual fine-tuning (requires expert knowledge!)
  - if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using clasp option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using clasp option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using clasp option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

[1] Y. Babovich and V. Lifschitz.
Computing answer sets using program completion.
Unpublished draft, 2003.

[2] C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.
Cambridge University Press, 2003.

[3] C. Baral, G. Brewka, and J. Schlipf, editors.
Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[4] C. Baral and M. Gelfond.
Logic programming and knowledge representation.
Journal of Logic Programming, 12:1–80, 1994.

[5] S. Baselice, P. Bonatti, and M. Gelfond.
Towards an integration of answer set and constraint solving.

In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

[6] A. Biere.
Adaptive restart strategies for conflict driven SAT solvers.
In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer-Verlag, 2008.

[7] A. Biere.
PicoSAT essentials.
Journal on Satisfiability, Boolean Modeling and Computation, 4:75–97, 2008.

[8] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*.

IOS Press, 2009.

[9] G. Brewka, T. Eiter, and M. Truszczyński.
Answer set programming at a glance.
Communications of the ACM, 54(12):92–103, 2011.

[10] G. Brewka, I. Niemelä, and M. Truszczyński.
Answer set optimization.
In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 867–872. Morgan Kaufmann Publishers, 2003.

[11] K. Clark.
Negation as failure.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[12] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.
Handbook of Tableau Methods.
Kluwer Academic Publishers, 1999.

[13] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov.
Complexity and expressive power of logic programming.
In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity (CCC'97)*, pages 82–101. IEEE Computer Society Press, 1997.

[14] M. Davis, G. Logemann, and D. Loveland.
A machine program for theorem-proving.
Communications of the ACM, 5:394–397, 1962.

[15] M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

[16] E. Di Rosa, E. Giunchiglia, and M. Maratea.
Solving satisfiability problems with preferences.
Constraints, 15(4):485–515, 2010.

[17] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub.

Conflict-driven disjunctive answer set solving.
In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008.

[18] C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub.
Heuristics in conflict resolution.
In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, number UNSW-CSE-TR-0819 in School of Computer Science and Engineering, The University of New South Wales, Technical Report Series, pages 141–149, 2008.

[19] N. Eén and N. Sörensson.
An extensible SAT-solver.
In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004.

[20] T. Eiter and G. Gottlob.
On the computational cost of disjunctive logic programming: Propositional case.
Annals of Mathematics and Artificial Intelligence, 15(3-4):289–323, 1995.

[21] T. Eiter, G. Ianni, and T. Krennwallner.
Answer Set Programming: A Primer.
In S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M. Rousset, and R. Schmidt, editors, *Fifth International Reasoning Web Summer School (RW'09)*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer-Verlag, 2009.

[22] F. Fages.
Consistency of Clark's completion and the existence of stable models.
Journal of Methods of Logic in Computer Science, 1:51–60, 1994.

[23] P. Ferraris.
Answer sets for propositional theories.

In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 119–131. Springer-Verlag, 2005.

[24] P. Ferraris and V. Lifschitz.
Mathematical foundations of answer set programming.
In S. Artëmov, H. Barringer, A. d'Avila Garcez, L. Lamb, and J. Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay*, volume 1, pages 615–664. College Publications, 2005.

[25] M. Fitting.
A Kripke-Kleene semantics for logic programs.
Journal of Logic Programming, 2(4):295–312, 1985.

[26] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub.
Abstract Gringo.
Theory and Practice of Logic Programming, 15(4-5):449–463, 2015.
Available at `http://arxiv.org/abs/1507.06576`.

[27] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele.
Potassco User Guide.
University of Potsdam, second edition edition, 2015.

[28] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.

[29] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
Engineering an incremental ASP solver.
In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2008.

[30] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.

On the implementation of weight constraint rules in conflict-driven ASP solvers.
In Hill and Warren [49], pages 250–264.

[31] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Answer Set Solving in Practice.
Synthesis Lectures on Artificial Intelligence and Machine Learning.
Morgan and Claypool Publishers, 2012.

[32] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
clasp: A conflict-driven answer set solver.
In Baral et al. [3], pages 260–265.

[33] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. [3], pages 136–148.

[34] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso [74], pages 386–392.

[35] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Advanced preprocessing for answer set solving.
In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

[36] M. Gebser, B. Kaufmann, and T. Schaub.
The conflict-driven answer set solver clasp: Progress report.
In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 509–514. Springer-Verlag, 2009.

[37] M. Gebser, B. Kaufmann, and T. Schaub.
Solution enumeration for projected Boolean search problems.
In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*

(CPAIOR'09), volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.

[38] M. Gebser, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren [49], pages 235–249.

[39] M. Gebser and T. Schaub.
Tableau calculi for answer set programming.
In S. Etalle and M. Truszczyński, editors, *Proceedings of the Twenty-second International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 2006.

[40] M. Gebser and T. Schaub.
Generic tableaux for answer set programming.
In V. Dahl and I. Niemelä, editors, *Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2007.

[41] M. Gelfond.
Answer sets.
In V. Lifschitz, F. van Harmelen, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier Science, 2008.

[42] M. Gelfond and Y. Kahl.
Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.
Cambridge University Press, 2014.

[43] M. Gelfond and N. Leone.
Logic programming and knowledge representation — the A-Prolog perspective.
Artificial Intelligence, 138(1-2):3–38, 2002.

[44] M. Gelfond and V. Lifschitz.
The stable model semantics for logic programming.

In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[45] M. Gelfond and V. Lifschitz.
Logic programs with classical negation.
In D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, pages 579–597. MIT Press, 1990.

[46] E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.
Journal of Automated Reasoning, 36(4):345–377, 2006.

[47] K. Gödel.
Zum intuitionistischen Aussagenkalkül.
Anzeiger der Akademie der Wissenschaften in Wien, page 65–66, 1932.

[48] A. Heyting.

Die formalen Regeln der intuitionistischen Logik.
In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, page 42–56. Deutsche Akademie der Wissenschaften zu Berlin, 1930. Reprint in Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik, Akademie-Verlag, 1986.

[49] P. Hill and D. Warren, editors.
Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[50] J. Huang.
The effect of restarts on the efficiency of clause learning.
In Veloso [74], pages 2318–2323.

[51] K. Konczak, T. Linke, and T. Schaub.
Graphs and colorings for answer set programming.
Theory and Practice of Logic Programming, 6(1-2):61–106, 2006.

[52] J. Lee.

A model-theoretic counterpart of loop formulas.
In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

[53] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello.
The DLV system for knowledge representation and reasoning.
ACM Transactions on Computational Logic, 7(3):499–562, 2006.

[54] V. Lifschitz.
Answer set programming and plan generation.
Artificial Intelligence, 138(1-2):39–54, 2002.

[55] V. Lifschitz.
Introduction to answer set programming.
Unpublished draft, 2004.

[56] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?

ACM Transactions on Computational Logic, 7(2):261–268, 2006.

[57] F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.
Artificial Intelligence, 157(1-2):115–137, 2004.

[58] V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.
Artifical Intelligence. Springer-Verlag, 1993.

[59] V. Marek and M. Truszczyński.
Stable models and an alternative logic programming paradigm.
In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[60] J. Marques-Silva, I. Lynce, and S. Malik.
Conflict-driven clause learning SAT solvers.
In Biere et al. [8], chapter 4, pages 131–153.

[61] J. Marques-Silva and K. Sakallah.

GRASP: A search algorithm for propositional satisfiability.
IEEE Transactions on Computers, 48(5):506–521, 1999.

[62] V. Mellarkod and M. Gelfond.
Integrating answer set reasoning with constraint solving techniques.
In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.

[63] V. Mellarkod, M. Gelfond, and Y. Zhang.
Integrating answer set programming and constraint logic programming.
Annals of Mathematics and Artificial Intelligence, 53(1-4):251–287, 2008.

[64] D. Mitchell.
A SAT solver primer.
Bulletin of the European Association for Theoretical Computer Science, 85:112–133, 2005.

[65] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

[66] I. Niemelä.
Logic programs with stable model semantics as a constraint programming paradigm.
Annals of Mathematics and Artificial Intelligence, 25(3-4):241–273, 1999.

[67] R. Nieuwenhuis, A. Oliveras, and C. Tinelli.
Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T).
Journal of the ACM, 53(6):937–977, 2006.

[68] K. Pipatsrisawat and A. Darwiche.
A lightweight component caching scheme for satisfiability solvers.

In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[69] L. Ryan.
Efficient algorithms for clause-learning SAT solvers.
Master's thesis, Simon Fraser University, 2004.

[70] P. Simons, I. Niemelä, and T. Soininen.
Extending and implementing the stable model semantics.
Artificial Intelligence, 138(1-2):181–234, 2002.

[71] T. Son and E. Pontelli.
Planning with preferences using logic programming.
Theory and Practice of Logic Programming, 6(5):559–608, 2006.

[72] T. Syrjänen.
Lparse 1.0 user's manual, 2001.

[73] A. Van Gelder, K. Ross, and J. Schlipf.

The well-founded semantics for general logic programs.
Journal of the ACM, 38(3):620–650, 1991.

[74] M. Veloso, editor.
Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI/MIT Press, 2007.

[75] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik.
Efficient conflict driven learning in a Boolean satisfiability solver.
In R. Ernst, editor, *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. IEEE Computer Society Press, 2001.