Answer Set Solving in Practice

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

Potassco

Potassco Slide Packages are licensed under a Creative Commons Attribution 3.0 Unported License.
Advanced Modeling: Overview

1. Tweaking N-Queens
2. Do’s and Don’t’s
3. Hints
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 \( \sim \frac{8!}{2} = 20160 \) “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 \( \sim \frac{8!}{2} = 20160 \) "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 $\sim \frac{8!}{2} = 20160$ “operations”
Outline

1. Tweaking $N$-Queens
2. Do’s and Don’t’s
3. Hints
Problem Specification

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

$N = 4$
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

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Placement

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Q</td>
<td></td>
<td>Q</td>
</tr>
</tbody>
</table>
Tweaking $N$-Queens

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

```lp
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.
```

Anything missing?

Torsten Schaub (KRR@UP)  Answer Set Solving in Practice  October 20, 2018  457 / 535
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

```lp
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

```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.
```
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

```lp
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.
```
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.
```
Tweaking $N$-Queens

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.
```

Anything missing?
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.
Tweaking N-Queens

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
Tweaking $N$-Queens

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!

gingo -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
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
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.
Tweaking $N$-Queens

A First Refinement

At least $N$ queens?

Exactly one queen per row and column!

queues_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!

```lp
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!

```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
Tweaking $N$-Queens

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
Restart : 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

gingo -c n=122 queens_1.lp | wc

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys 0m0.090s
Tweaking \( N \)-Queens

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
Tweaking $N$-Queens

A First Refinement

Grounding Time $\sim$ Space

```prolog
queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
```
A First Refinement
Grounding Time $\sim$ Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
A First Refinement

Grounding Time \sim \text{Space}

queens_1.lp

\%
\% DOMAIN
\#const n=4. square(1..n,1..n).
\%
\% GENERATE
\{ queen(X,Y) \} :- square(X,Y).
\%
\% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
\%
\% DISPLAY
#show queen/2.
Tweaking $N$-Queens

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

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

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

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

% DISPLAY
\#show queen/2.
A First Refinement
Grounding Time $\sim$ Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
Tweaking N-Queens

A First Refinement

Grounding Time \sim Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
Tweaking $N$-Queens

A First Refinement

Grounding Time $\sim$ Space

---

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
Tweaking N-Queens

A First Refinement
Grounding Time \sim Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
Tweaking $N$-Queens

A First Refinement
Grounding Time $\sim$ Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
Tweaking N-Queens

A First Refinement
Grounding Time $\sim$ Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.
Tweaking $N$-Queens

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

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

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

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

% DISPLAY
#show queen/2.

Diagonals make trouble!
Enumerating Diagonals

\[ N = 4 \]

\[
\begin{array}{cccc}
4 & 3 & 2 & 1 \\
3 & 2 & 1 & \cdot \\
2 & 1 & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot \\
\end{array}
\]

\[
\begin{array}{cccc}
4 & 3 & 2 & 1 \\
3 & 2 & 1 & \cdot \\
2 & 1 & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot \\
\end{array}
\]

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

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

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

#### Enumerating Diagonals

For each $N$, indexes $1, \ldots, (2N) - 1$ refer to squares on the diagonals.

**Note**

- For each $N$, indexes $1, \ldots, (2N) - 1$ refer to squares on the diagonals.

**#diagonal\(_1\) = (#row + #column) - 1**

**#diagonal\(_2\) = (#row - #column) + N**

---

**N = 4**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

---

**Answer Set Solving in Practice**

October 20, 2018 465 / 535
Tweaking $N$-Queens

Enumerating Diagonals

$N = 4$

#diagonal$_1 = (#row + #column) - 1$

#diagonal$_2 = (#row - #column) + N$

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

\[ N = 4 \]

\[ \begin{array}{cccc}
4 & 5 & 6 & 7 \\
3 & 4 & 5 & 6 \\
2 & 3 & 4 & 5 \\
1 & 2 & 3 & 4 \\
\end{array} \]

\[ \begin{array}{cccc}
4 & 7 & 6 & 5 \\
3 & 6 & 5 & 4 \\
2 & 5 & 4 & 3 \\
1 & 4 & 3 & 2 \\
\end{array} \]

\#diagonal_1 = (\#row + \#column) - 1

\#diagonal_2 = (\#row - \#column) + N

- Note For each \( N \), indexes 1, \ldots, (2*N) - 1 refer to squares on \#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!

```bash
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

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

<table>
<thead>
<tr>
<th>Models</th>
<th>1+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.210s</td>
</tr>
<tr>
<td>Choices</td>
<td>11036</td>
</tr>
<tr>
<td>Conflicts</td>
<td>499</td>
</tr>
<tr>
<td>Restart</td>
<td>3</td>
</tr>
</tbody>
</table>

| 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
Tweaking $N$-Queens

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
Tweaking \( N \)-Queens

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
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
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
Tweaking $N$-Queens

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
Tweaking \textit{N}-Queens

A Case for Oracles
Let’s Place 600 Queens!

\texttt{gringo -c n=600 queens_3.lp | clasp --stats}
\texttt{--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
Tweaking $N$-Queens

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

1. Tweaking $N$-Queens
2. Do’s and Don’t’s
3. Hints
Do's and Don't's

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).
Do’s and Don’ts

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).
```
Do's and Don'ts

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 ‘ː’)** ... 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(asparagus,fresh).
pro(asparagus,tasty).
pro(cucumber,cheap).
pro(cucumber,fresh).
pro(cucumber,tasty).
pro(asparagus,clean).
Do’s and Don’t’s

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).
```
Do’s and Don’ts

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).
```
Do’s and Don’t’s

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).
pro(asparagus, cheap).
pro(asparagus, fresh).
pro(asparagus, tasty).
pro(asparagus, clean).

veg(cucumber).
pro(cucumber, cheap).
pro(cucumber, fresh).
pro(cucumber, tasty).
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
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
5 6 1 2 3 4
6 1 2 3 4 5
```

represented by facts:

- \( \text{square}(1,1) \)
- \( \text{square}(2,1) \)
- \( \text{square}(3,1) \)
- \( \text{square}(4,1) \)
- \( \text{square}(5,1) \)
- \( \text{square}(6,1) \)
- \( \text{square}(1,2) \)
- \( \text{square}(2,2) \)
- \( \text{square}(3,2) \)
- \( \text{square}(4,2) \)
- \( \text{square}(5,2) \)
- \( \text{square}(6,2) \)
- \( \text{square}(1,3) \)
- \( \text{square}(2,3) \)
- \( \text{square}(3,3) \)
- \( \text{square}(4,3) \)
- \( \text{square}(5,3) \)
- \( \text{square}(6,3) \)
- \( \text{square}(1,4) \)
- \( \text{square}(2,4) \)
- \( \text{square}(3,4) \)
- \( \text{square}(4,4) \)
- \( \text{square}(5,4) \)
- \( \text{square}(6,4) \)
- \( \text{square}(1,5) \)
- \( \text{square}(2,5) \)
- \( \text{square}(3,5) \)
- \( \text{square}(4,5) \)
- \( \text{square}(5,5) \)
- \( \text{square}(6,5) \)
- \( \text{square}(1,6) \)
- \( \text{square}(2,6) \)
- \( \text{square}(3,6) \)
- \( \text{square}(4,6) \)
- \( \text{square}(5,6) \)
- \( \text{square}(6,6) \)

**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
```

represented by atoms:

- \( \text{num}(1,1,1) \)
- \( \text{num}(1,2,2) \)
- \( \text{num}(1,3,3) \)
- \( \text{num}(1,4,4) \)
- \( \text{num}(1,5,5) \)
- \( \text{num}(1,6,6) \)
- \( \text{num}(2,1,2) \)
- \( \text{num}(2,2,3) \)
- \( \text{num}(2,3,4) \)
- \( \text{num}(2,4,5) \)
- \( \text{num}(2,5,6) \)
- \( \text{num}(2,6,1) \)
- \( \text{num}(3,1,3) \)
- \( \text{num}(3,2,4) \)
- \( \text{num}(3,3,5) \)
- \( \text{num}(3,4,6) \)
- \( \text{num}(3,5,1) \)
- \( \text{num}(3,6,2) \)
- \( \text{num}(4,1,4) \)
- \( \text{num}(4,2,5) \)
- \( \text{num}(4,3,6) \)
- \( \text{num}(4,4,1) \)
- \( \text{num}(4,5,2) \)
- \( \text{num}(4,6,3) \)
- \( \text{num}(5,1,5) \)
- \( \text{num}(5,2,6) \)
- \( \text{num}(5,3,1) \)
- \( \text{num}(5,4,2) \)
- \( \text{num}(5,5,3) \)
- \( \text{num}(5,6,4) \)
- \( \text{num}(6,1,6) \)
- \( \text{num}(6,2,1) \)
- \( \text{num}(6,3,2) \)
- \( \text{num}(6,4,3) \)
- \( \text{num}(6,5,4) \)
- \( \text{num}(6,6,5) \)
Do's and Don'ts

Running Example: Latin Square

**Given:** an $N \times N$ board

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, ..., $N$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

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)` ...

Torsten Schaub (KRR@UP)
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

gingroup 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
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”

```bash
gringo latin_0.lp | wc
105480 2558984 14005258
```

```bash
gringo latin_1.lp | wc
42056 273672 1690522
```
Do's and Don'ts

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
Unraveling Symmetric Inequalities

Another Latin square encoding

\[
\begin{align*}
% \text{DOMAIN} \\
\text{const n=32. square(1..n,1..n).} \\
% \text{GENERATE} \\
1 \{ \text{num(X,Y,N) : N = 1..n } \} 1 \text{ :- square(X,Y).} \\
% \text{TEST} \\
\text{ :- num(X,Y,N), num(X,Y',N), Y \neq Y'.} \\
\text{ :- num(X,Y,N), num(X',Y,N), X \neq X'.}
\end{align*}
\]

- 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
Do's and Don't's

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”)

grego latin_2.lp | wc
2071560 12389384 40906946
grego 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
```
Do’s and Don't’s

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
```
Do's and Dont's

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!

griego latin_3.lp | wc
1055752 6294536 21099558
griego 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
Do's and Don'ts

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.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.
   :- num(X,Y,N), gtX(X,Y,N).
gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
   :- 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
griungo latin_4.lp | wc
1055752 6294536 21099558
228360 1205256 4780744
Do's and Dont's

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.
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.
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.
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.
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
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.
```
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.
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.

gingo latin_5.lp | wc 304136 5778440 30252505
gingo latin_6.lp | wc
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.
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)
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!
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!
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
Do's and Don'ts

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
Hints

Outline

1. Tweaking $N$-Queens
2. Do's and Don't's
3. Hints
Hints

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) identical 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.
Hints

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) identical 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.
Hints

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) identical 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.
Hints

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.
Hints

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) identical 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.
Hints

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) identical 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/](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 (e.g. 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 (e.g. 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 (e.g., 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 (e.g., 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 (e.g., `time(gringo [...] | wc)`)  
  2. grounding info (e.g., `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 `acclingo`)  
    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 (e.g. `time(gringo [...] | wc)`)
  2. grounding info (e.g. `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
  - system tools (eg. `time(gringo [...]) | wc`)
  - 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
Hints

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of gringo
  1. system tools (e.g. `time(gringo [...] | wc)`)
  2. grounding info (e.g. `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
Computing answer sets using program completion.  

Knowledge Representation, Reasoning and Declarative Problem Solving.  


Logic programming and knowledge representation.  

Towards an integration of answer set and constraint solving.

Adaptive restart strategies for conflict driven SAT solvers.  

PicoSAT essentials.  

*Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. 
Answer set programming at a glance. 

Answer set optimization. 

Negation as failure. 

*Handbook of Tableau Methods.* 
Complexity and expressive power of logic programming.  
In *Proceedings of the Twelfth Annual IEEE Conference on  

A machine program for theorem-proving.  

A computing procedure for quantification theory.  

Solving satisfiability problems with preferences.  

M. Ostrowski, and T. Schaub.
Conflict-driven disjunctive answer set solving.


On the computational cost of disjunctive logic programming: Propositional case.  

*Answer Set Programming: A Primer*.  

[22] F. Fages.  
*Consistency of Clark's completion and the existence of stable models*.  

*Answer sets for propositional theories*. 

*Mathematical foundations of answer set programming.*  

*A Kripke-Kleene semantics for logic programs.*  

*Abstract Gringo.*  
*Potassco User Guide.*

A user’s guide to gringo, clasp, clingo, and iclingo.

Engineering an incremental ASP solver.

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

*Answer Set Solving in Practice.*

clasp: A conflict-driven answer set solver.
In Baral et al. [3], pages 260–265.

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.
Advanced preprocessing for answer set solving.

The conflict-driven answer set solver clasp: Progress report.

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
Hints


[41] M. Gelfond.  
**Answer sets.**  


Logic programming and knowledge representation — the A-Prolog perspective.  

The stable model semantics for logic programming.

Logic programs with classical negation.

Answer set programming based on propositional satisfiability.

Zum intuitionistischen Aussagenkalkül.

Die formalen Regeln der intuitionistischen Logik.


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

*Graphs and colorings for answer set programming.*

A model-theoretic counterpart of loop formulas.

The DLV system for knowledge representation and reasoning.

[54] V. Lifschitz.
Answer set programming and plan generation.

Introduction to answer set programming.

[56] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
[57] F. Lin and Y. Zhao. 
ASSAT: computing answer sets of a logic program by SAT solvers. 

[58] V. Marek and M. Truszczyński. 
*Nonmonotonic logic: context-dependent reasoning*. 

*Stable models and an alternative logic programming paradigm*. 

*Conflict-driven clause learning SAT solvers*. 
In Biere et al. [8], chapter 4, pages 131–153.

GRASP: A search algorithm for propositional satisfiability. 

Integrating answer set reasoning with constraint solving techniques.

Integrating answer set programming and constraint logic programming.

A SAT solver primer.


Efficient algorithms for clause-learning SAT solvers.  

Extending and implementing the stable model semantics.  

[71] T. Son and E. Pontelli.  
Planning with preferences using logic programming.  

Lparse 1.0 user’s manual, 2001.

Hints

The well-founded semantics for general logic programs.  

[74] M. Veloso, editor.  
Proceedings of the Twentieth International Joint Conference on  

Efficient conflict driven learning in a Boolean satisfiability solver.  
In R. Ernst, editor, Proceedings of the International Conference on  