Declarative Modeling

Winter Term 2012/2013
Lecture Slides

Martin Gebser
University of Potsdam
gbser@cs.uni-potsdam.de

Potassco Slide Packages are licensed under a Creative Commons Attribution 3.0 Unported License.
Rough Roadmap

1 Background
   - Answer Set Programming
   - Grounding
   - Solving

2 Modeling
   - Satisfiability
   - Optimization
   - Incrementality
   - Meta-Programming

3 Applications
   - Puzzles
   - Planning
   - Scheduling
Resources

- Course material
  - http://moodle.cs.uni-potsdam.de
  - http://potassco.sourceforge.net/teaching.html
  - http://sourceforge.net/p/potassco/wiki/Home/

- Systems
  - clasp
  - claspD
  - gringo
  - clingo
  - iclingo
  - oclingo

http://potassco.sourceforge.net
The Potassco Book


1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions
Administrative Information

- Lecture: 2h (weekly)
- Exercise: 1–2h (weekly)
- Credits: 3

- C(ourse)MS:
  http://moodle.cs.uni-potsdam.de

- General Info:
  http://apache.cs.uni-potsdam.de/de/profs/ifi/wv/lehre/

- Mark: Project implementation and documentation
  - Area: “Praktische Informatik” / “Angewandte Informatik” / “Wahlfrei”
  - Examiner (@PULS): Torsten Schaub
Background: Overview

1 Motivation

2 Answer Set Programming

3 Grounding and Solving

4 Modeling by Example
Overview

1 Motivation

2 Answer Set Programming

3 Grounding and Solving

4 Modeling by Example
Human versus Computational Intelligence?

- Speed: ✔
- Accuracy: ✔
- Intuitions: ✗
- Knowledge: ✗

VS.

- Speed: ✗
- Accuracy: ✗
- Intuitions: ✔
- Knowledge: ✔
Human plus Computational Intelligence!

$KRR = Knowledge + Representation + Reasoning$

Picture from Computational Thinking Illustrated
Collaborative Problem Solving

Human Perspective

- Approach: Semantic
- Means: Subjective
- Outcome: Singular

Computational Perspective

- Approach: Syntactic
- Means: Objective
- Outcome: Reproducible

Bridging the Gap

- Formal problem representation
- Transfer (of intuitions/knowledge) from semantic to syntactic level
- Communication channel between human and machine
- Minimize “time to solution”!

Martin Gebser (KRR@UP)
Illustration: Pigeonhole Principle

One small step for man

Can one put \( n+1 \) pigeons into \( n \) holes such that no pigeons share a hole?

**NO!**
Illustration: Pigeonhole Principle

One worst case for machines

Can one put $n+1$ pigeons into $n$ holes such that no pigeons share a hole?

“Naive” Formulation

<table>
<thead>
<tr>
<th>Each pigeon requires a hole</th>
<th>No pigeons share a hole</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{1,1} \lor p_{1,2} \lor \cdots \lor p_{1,n}$</td>
<td>$\neg p_{1,1} \lor \neg p_{2,1}$</td>
</tr>
<tr>
<td>$p_{2,1} \lor p_{2,2} \lor \cdots \lor p_{2,n}$</td>
<td>$\neg p_{1,1} \lor \neg p_{3,1}$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>$p_{n,1} \lor p_{n,2} \lor \cdots \lor p_{n,n}$</td>
<td>$\neg p_{n-1,1} \lor \neg p_{n+1,1}$</td>
</tr>
<tr>
<td>$p_{n+1,1} \lor p_{n+1,2} \lor \cdots \lor p_{n+1,n}$</td>
<td>$\neg p_{n,1} \lor \neg p_{n+1,1}$</td>
</tr>
</tbody>
</table>

(runtime of (resolution-based) solvers: \quad)
Illustration: Pigeonhole Principle

With a little help from my friends

Can one put \( n+1 \) pigeons into \( n \) holes such that no pigeons share a hole?

"Clever" Formulation

<table>
<thead>
<tr>
<th>Each pigeon requires a hole</th>
<th>No pigeons share a hole</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_{1,1} )</td>
<td>( \neg p_{1,1} \lor \neg p_{2,1} )</td>
</tr>
<tr>
<td>( p_{2,1} \lor p_{2,2} )</td>
<td>( \neg p_{1,1} \lor \neg p_{3,1} )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( p_{n,1} \lor p_{n,2} \lor \cdots \lor p_{n,n} )</td>
<td>( \neg p_{n-1,1} \lor \neg p_{n+1,1} )</td>
</tr>
<tr>
<td>( p_{n+1,1} \lor p_{n+1,2} \lor \cdots \lor p_{n+1,n} )</td>
<td>( \neg p_{n,1} \lor \neg p_{n+1,1} )</td>
</tr>
</tbody>
</table>

Runtime of (resolution-based) solvers:
Overview

1 Motivation

2 Answer Set Programming

3 Grounding and Solving

4 Modeling by Example
Answer Set Programming (ASP)

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
tailored to Knowledge Representation and Reasoning
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability checking)
- ASP allows for solving all search problems in \( NP \) (and \( NP^{NP} \)) in a uniform way
- ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP, CASC, MISC, PB, and SAT competition.
- ASP embraces many emerging application areas
Declarative Problem Solving
with ASP

Problem

Modeling

Logic Program

KR

Grounder

Solver

Solutions

Interpreting

Flowchart:
- Problem
  - Modeling
  - KR
  - Logic Program
  - LP
  - Grounder
  - DB
  - Solving
  - SAT
  - Stable Models
  - DB+LP+KR
(Some) Modeling Constructs

- **Variables (over the Herbrand Universe)**
  - \( p(X) :- q(X), \neg r(X) \).  
    over constants \( a, b, c \) stands for  
    \( p(a) :- q(a), \neg r(a). \ p(b) :- q(b), \neg r(b). \ p(c) :- q(c), \neg r(c). \)

- **Conditional Literals**
  - \( p :- q(X) : r(X) \).
    given \( r(a), r(b), r(c) \) stands for  
    \( p :- q(a), q(b), q(c). \)

- **Integrity Constraints**
  - \( :- p(X), \neg q(X). \)

- **Choice Rules**
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y). \)

- **Aggregates**
  - \( s(Y) :- r(Y), 2 \#count\{ p(X,Y) : q(X) \} 7. \)
    also: \#sum, \#min, \#max, \#avg, \#even, \#odd

- **Optimize Statements**
  - \#minimize[ \( p(X,Y) = Y : q(X) : r(Y) \)].
    also: \#maximize
Reasoning Modes

- Satisfiability
- Optimization
- Enumeration\(^\dagger\)
- Projection\(^\dagger\)
- Intersection\(^\dagger\)
- Union\(^\ddagger\)

and combinations of them

via single- or multi-threading

\(^\dagger\) without solution recording
\(^\ddagger\) without solution enumeration
What is ASP good for?

- Combinatorial search problems in the realm of \( P \), \( NP \), and \( NP^{NP} \) (some with substantial amount of data), like
  - Assisted Living
  - Automated Planning
  - Compiler (Super-)Optimization
  - Composition of (Harmonic and Melodic) Music
  - Data Integration
  - Decision Support for NASA Shuttle Controllers
  - General Game Playing
  - Hardware Synthesis
  - Model Checking
  - Product Configuration
  - Reviewer Assignment
  - Robotics
  - Systems Biology
  - (Industrial) Team Building
  - and many more
Overview

1. Motivation
2. Answer Set Programming
3. Grounding and Solving
4. Modeling by Example
Grounding Basics

Task Instantiate first-order rules (encoding) relative to facts (instance)

Approach

- Head atom(s) of a rule (or fact) are derivable if all positive elements of the rule body are derivable
- Iterative instantiation of derivable atoms and resulting rule bodies, starting from facts, yields all relevant ground rules

Semi-naive Evaluation

Safety Requirement

- Any variable of a rule must appear (outside of arithmetic expressions)
  - globally in some positive element of the rule body or
  - locally on the right-hand side of "::" in a condition
Predicate Classification

Built-in Predicates

- Term comparisons
  - $X == Y, f(X1,Y1) != f(X2,Y2), (X1,Y1) < (X2,Y2)$, etc.
  - Can be viewed as negative elements, not binding variables to values

- Term assignments
  - $X := Y+1, f(X,Y) := f(XX+1,YY-1), X := \text{#min}[ p(Y) = Y ]$, etc.
  - Bind variables on left-hand side, if those on right-hand side are bound

Domain Predicates (including built-ins)

- Predicates whose atoms neither
  - occur in heads of choice rules or depend, transitively, on them nor
  - negatively depend, transitively, on atoms of the same predicate
  - Are fully evaluated upon grounding, not subject to search upon solving
    (right-hand sides of conditions, currently, require domain predicates)
Solving Basics

Task Find some (optimal) stable model of a propositional logic program

Approach

- Consider atoms, rule bodies, and aggregates as propositional variables
- Unit propagation (extended to aggregates, unfounded sets, and optimize statements) yields deterministic consequences
- Decision guesses some literal when fixpoint is partial and conflict-free
- Conflict-driven learning records nogood and directs backjumping from deadend

Two Sides to Every Story

Satisfiability Find some stable model (quickly)

Unsatisfiability Build some refutation (quickly), includes optimality proofs
Overview

1. Motivation

2. Answer Set Programming

3. Grounding and Solving

4. Modeling by Example
### 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**

![Chessboard Image]

**Placement**

![Placement Image]
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

```
queens0.lp

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

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

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

% DISPLAY
#hide. #show queen/2.
```
A First Encoding
Let’s Place 8 Queens!

gringo -c n=8 queens0.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!**

```sh
gringo -c n=22 queens0.lp | clasp --stats
```

<table>
<thead>
<tr>
<th>Answer: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...</td>
</tr>
</tbody>
</table>

**SATISFIABLE**

<table>
<thead>
<tr>
<th>Models   : 1+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time     : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)</td>
</tr>
<tr>
<td>CPU Time : 147.480s</td>
</tr>
<tr>
<td>Choices  : 594960</td>
</tr>
<tr>
<td>Conflicts: 574565</td>
</tr>
<tr>
<td>Restarts : 19</td>
</tr>
</tbody>
</table>

| Variables : 17271 |
| Constraints: 16787 |
A First Refinement

At least \( N \) queens?

Exactly one queen per row and column!

```prolog
queens0.lp

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

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

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

% DISPLAY
#hide. #show queen/2.
```

Martin Gebser (KRR@UP)
A First Refinement
Let’s Place 22 Queens!

```
gringo -c n=22 queens1.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!

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>gringo -c n=122 queens1.lp</td>
<td>clasp --stats</td>
</tr>
<tr>
<td><strong>Answer:</strong> 1</td>
<td></td>
</tr>
<tr>
<td>queen(1,24) queen(2,52)</td>
<td>queen(3,37) queen(4,60) queen(5,76) ...</td>
</tr>
<tr>
<td>SATISFIABLE</td>
<td></td>
</tr>
</tbody>
</table>

- **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 queens1.lp | clasp --stats

1241358 7402724 24950848

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

Just kidding :-)!

Grounding makes the problem!
A First Refinement

Grounding Time $\sim$ Space

```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
#hide. #show queen/2.
```

Diagonals make trouble!

Martin Gebser (KRR@UP)
Enumerating Diagonals

\[ N = 4 \]

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

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

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

For each \( N \), indexes 1, \ldots, (2*N)−1 refer to squares on \( \text{#diagonal}_{1/2} \)
A Second Refinement
Let’s go for Diagonals!

queens2.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
#hide. #show queen/2.
A Second Refinement
Let’s Place 122 Queens!

```
gringo -c n=122 queens2.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!

gingo -c n=300 queens2.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 Third Refinement

Let's Precalculate Indexes!

```
queens3.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
{ 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.
:- D := 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D := 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```
A Third Refinement
Let's Place 300 Queens!

gringo -c n=300 queens3.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 queens3.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!

```bash
gringo -c n=600 queens3.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

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models</td>
<td>1+</td>
</tr>
<tr>
<td>Time</td>
<td>76.798s</td>
</tr>
<tr>
<td>(Solving: 65.81s</td>
<td></td>
</tr>
<tr>
<td>1st Model: 65.75s</td>
<td></td>
</tr>
<tr>
<td>Unsat: 0.00s)</td>
<td></td>
</tr>
<tr>
<td>CPU Time</td>
<td>68.620s</td>
</tr>
<tr>
<td>Choices</td>
<td>869379</td>
</tr>
<tr>
<td>Conflicts</td>
<td>25746</td>
</tr>
<tr>
<td>Restarts</td>
<td>12</td>
</tr>
<tr>
<td>Variables</td>
<td>365994</td>
</tr>
<tr>
<td>Constraints</td>
<td>4794</td>
</tr>
</tbody>
</table>
# The Modeling and Solving Process

## Problem Comprehension

1. Create a working encoding
2. Verify correctness on small (toy) instances

## Scaling up

1. Compact constraint formulation
2. Reduce (magnitude of) instantiation size
   Desirable: Linear to instance size (terms in facts)
3. Reduce (magnitude of) instantiation time
   Desirable: Few discarded instantiations (eg. due to built-in predicates)
4. Incorporate more knowledge
   Conceivable: Redundant constraints, symmetry breaking, etc.

## Final Punch (only)

1. Tweak solver parameters
2. Increase computing power (multi-cores, clusters, etc.)
Modeling Methodology: Overview

5 Quantification
6 Projection
7 Inequality
8 Assignment
9 Symmetry
10 Ordering
11 Counting
12 Minutes
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,P) : pref(P).
```
Running Example: Latin Square

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

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

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, ..., $N$

```
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:

- `num(1,1,1)` ...
- `num(1,6,6)`
- `num(2,1,2)` ...
- `num(2,6,1)`
- `num(3,1,3)` ...
- `num(3,6,2)`
- `num(4,1,4)` ...
- `num(4,6,3)`
- `num(5,1,5)` ...
- `num(5,6,4)`
- `num(6,1,6)` ...
- `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(X1,Y1), N := 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N := 1..n, not num(X2,Y1,N) : square(X2,Y1).

Unreused “singleton variables”

<table>
<thead>
<tr>
<th>gringo latin0.lp</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td>105480 2558984 14005258</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gringo latin1.lp</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td>42056 273672 1690522</td>
<td></td>
</tr>
</tbody>
</table>
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(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.

Duplicate ground rules (swapping Y1/Y2 and X1/X2 gives the “same”)

ggringo latin2.lp | wc
2071560 12389384 40906946

ggringo latin3.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
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.  
gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.  
   :- num(X,Y,N), gtX(X,Y,N).

Uniqueness of N in a row/column checked by ENUMERATING PAIRS!

gringo latin3.lp | wc
1055752 6294536 21099558

gringo latin4.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[ square(X,n) = X ].

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

% DISPLAY
#hide. #show num/3. #show sigma/1.
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 #count{ num(X,Y,N) } 1.
:- Y := 1..n, N := 1..n, not 1 #count{ num(X,Y,N) } 1.

% DISPLAY
#hide. #show num/3.

gringo -c n=5 latin7.lp | clasp -q 0

Models : 161280       Time : 2.078s
Term Order

- Investigating terms along an order often helps to encode compactly
- Linking successive terms
- Attributing sequence numbers

**Example:** Vegetables reloaded

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).

pro(P) :- pro(X,P).

order,(P1,P2) :- pro(P1;P2), P1 < P2, P <= P1 : pro(P) : P < P2.

num(P1,1) :- pro(P1), not order(P,P1) : order(P,P1).
num(P2,N+1) :- num(P1,N), order(P1,P2).
Implementing Counters

- Aggregates do not always fit (eg. comparisons among many objects)
- Exact outcome is often unnecessary!
- Interval outcome is often easier to handle

Example: Vegetables continued

\[
\begin{align*}
\{ \text{has}(X,P) \} & : \text{pro}(X,P). \\
\text{count}(X,N+1,0) & : \text{veg}(X), \text{num}(P2,N), \text{not order}(P2,P) : \text{order}(P2,P). \\
\text{count}(X,N-1,C+1) & : \text{count}(X,N,C), \text{num}(P,N-1), \text{has}(X,P). \\
\text{count}(X,N-1,C) & : \text{count}(X,N,C), \text{num}(P,N-1), \text{not has}(X,P). \\
\text{counted}(X,C) & : \text{count}(X,N,C), 0 < C. \\
\text{counted}(C) & : \text{counted}(X,C). \\
\text{buy}(X) & : \text{counted}(X,C), \text{not counted}(C+1).
\end{align*}
\]
Walk Like an Egyptian

- Opt
- Test
- Define
- Generate
- Domain Predicates
- Facts

Abstraction
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!
   ▶ 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)

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 -t)
   - 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. Profiling info (e.g., `gringo --gstats --verbose=3 [...] > /dev/null`)  
- Once identified, reformulate “critical” logic program parts

Solving

- Check solving statistics (use `clasp --stats`)  
- If great search efforts (Conflicts/Choices/Restarts), then  
  1. Try auto-configuration (offered by `claspfolio`)  
  2. Try manual fine-tuning (requires expert knowledge!)  
  3. If possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver
Optimization Problems: Overview

13 From Satisfiability to Optimization

14 Counting-based Optimization

15 Summation-based Optimization

16 Minutes
Overview

13 From Satisfiability to Optimization

14 Counting-based Optimization

15 Summation-based Optimization

16 Minutes
Hard versus Soft Constraints

Hard Constraints

- Requirements to be fulfilled by any solution
  - Specification limits
  - Resource compliance
  - ...

Soft Constraints

- Desiderata whose violation can be tolerated
  - Preferences
  - Penalties
  - Utilities
  - ...

 Discriminate viable solutions
Search Phases in Minimization

- Satisfiability
- Descent
- Unsatisfiability

Solution Quality vs. Runtime
Hard Regions

<table>
<thead>
<tr>
<th>Problem Criticality</th>
<th>Satisfiability</th>
<th>Descent</th>
<th>Unsatisfiability</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard-constrained</td>
<td>difficult</td>
<td>difficult</td>
<td>comparably easy</td>
</tr>
<tr>
<td>under-constrained</td>
<td>trivial ✔</td>
<td>plenty solutions</td>
<td>very difficult</td>
</tr>
</tbody>
</table>

Add compact “redundant” constraints for shortcuts to (non-)solutions
Try “aggressive” search strategies (clasp options like --restart-on-model, --opt-heuristic, and/or --opt-hierarch)
Abstract from particular candidate solutions
- General lower bounds
- Symmetry breaking
- Encoding methods
Overview

13 From Satisfiability to Optimization

14 Counting-based Optimization

15 Summation-based Optimization

16 Minutes
Pigeonhole Principle Revisited

When we have to select at least \( n/2 \) out of \( n \) items, how many do we need? “Expert knowledge”: \( n/2! \)

Naive Encoding

```prolog
#const n = 24.
n/2 { select(1..n) }.
#minimize{ select(_) }.
```

Performance?

Answer: 1
select(13) select(14) select(15) select(16) select(17) select(18) ...
Optimization: 12
OPTIMUM FOUND

Time : 7.351s (Solving: 7.35s 1st Model: 0.00s Unsat: 7.35s)
Choices : 1831336
Conflicts : 1830166
Counting Revisited

When we have to select at least \( n/2 \) out of \( n \) items, how many do we need? Let’s count them!

Naive Encoding

```prolog
#const n = 24.

n/2 { select(1..n) }.

count(I,1) :- select(I).
count(I+1,C) :- count(I,C), I < n.
count(I+1,C+1) :- count(I,C), select(I+1).
:- not count(n,n/2).
#minimize{ count(n,_) }.
```

Performance?

- Time: 0.008s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
- Choices: 105
- Conflicts: 67
Localizing Penalties

- Counting over $n$ items (to minimize) requires $O(n^2)$ rules
  - Often penalties can be attributed to subgroups

Local Selection $+$ Local Counting

```prolog
#const n = 42.
1 { select(3*I-D) : D := 0..2 } :- I := 1..n/3.
% 1 { select(1), select(2), select(3) }.
% 1 { select(4), select(5), select(6) }.
% 1 { ... }.
#minimize{ count(3*I,_) : I := 1..n/3 }.
```

Performance?

Time : 0.003s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
Conflicts : 14
Variables : 140
Constraints : 266
Overview

13 From Satisfiability to Optimization

14 Counting-based Optimization

15 Summation-based Optimization

16 Minutes
Example: Traveling Sales-Person (TSP)

Task
Given a (directed) graph with positive edge costs, find a round trip with minimum accumulated edge costs.

(Complete) Graph Gadget: graph.lp

```prolog
#const n = 5.

node(1..n).

cost(X,Y,2*Y-X) :- X := 1..n-1,
                   Y := X+1..n.

cost(Y,X,C) :- cost(X,Y,C).

degree(X,Y) :- cost(X,Y,C).
```

```
1 -- 3
|    |
|    |
2 -- 4
|    |
|    |
3 -- 5
```

Martin Gebser  (KRR@UP)  Declarative Modeling  February 11, 2013  67 / 106
Taking TSP Literally

**Straightforward Encoding: tsp0.lp**

% GENERATE: Precisely one outgoing and incoming edge per node
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

% DEFINE + TEST: Each node must be reached from starting node
reached(X) :- X := #min[ node(Y) = Y ].
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize[ cycle(X,Y) = C : cost(X,Y,C) ].

gringo -c n=12 graph.lp tsp0.lp | clasp --stats

Models : 1
Optimization: 111
Time : 60.017s (Solving: 60.01s 1st Model: 0.00s Unsat: 57.61s)
Mind the Gap(s)!

- Every node requires some outgoing (and incoming) edge
- Gaps to minimum outgoing edge cost provide penalty per node!

**Elaborate Encoding:** tsp1.lp

```
% GENERATE: Precisely one outgoing and incoming edge per node
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

% DEFINE + TEST: Each node must be reached from starting node
reached(X) :- X := #min[ node(Y) = Y ].
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
cost(X,C) :- cost(X,_,C).

order(X,C1,C2) :- cost(X,C1;C2), C1 < C2, C <= C1 : cost(X,C) : C < C2.
```
Cycle Orientation

- When costs (and edges) are symmetric, any round trip can be reversed

Encoding Part for Symmetry Breaking: symmetry.lp

```prolog
% DOMAIN
asym :- cost(X,Y,C), not cost(Y,X,C).
init(X) :- X := #min[ node(Y) = Y ], not asym.
next(Y1,Y2) :- init(X), edge(Y1;Y2,X), Y1 < Y2, Y <= Y1 : edge(Y,X) : Y < Y2.

% BREAK SYMMETRY
skip(Y1) :- next(Y1,Y2), init(X), cycle(X,Y2).
skip(Y1) :- next(Y1,Y2), skip(Y2).
:- init(X), skip(Y), cycle(Y,X).
```

griego -c n=16 graph.lp tsp1.lp symmetry.lp | clasp --stats

Models : 1
Optimization: 28
Time : 12.995s (Solving: 12.99s 1st Model: 0.00s Unsat: 12.98s)
Conflicts : 923734
Overview

13 From Satisfiability to Optimization

14 Counting-based Optimization

15 Summation-based Optimization

16 Minutes
Optimization Phenomena

- Addressing candidate solutions (unfiltered) makes optimization hard
  - Consider candidates’ properties, but not candidates themselves
  - Orient baseline at hard constraints, to not optimize the empty set

- Sharing penalties (utilities) uniformly may sacrifice problem structure
  - Attribute penalties to subgroups, when there is a known partition
  - Consider local baselines for subgroups, in case they are divergent

- Transferring preference relationships to the propositional level is useful
  - Counting abstracts from particular items, which need no distinction
  - Chaining along gaps relates diverse quantitative values, if available
Objective Transformations

- Preference-preserving objective rewritings keep (optimal) outcomes

Elimination of Negative Costs

\[
\text{\#minimize[ } a = -1 \text{ ]. can be turned into:} \\
\text{\#minimize[ } \text{not } a = 1 \text{ ].}
\]

Replacement of \text{\#maximize}

\[
\text{\#maximize[ } a = 1 \text{ ]. can be turned into:} \\
\text{\#minimize[ } \text{not } a = 1 \text{ ].}
\]

- #maximize[ a = -1 ]. can be turned into: #minimize[ a = 1 ].

Upon grounding, \textit{gringo} rewrites #\text{maximize}(s) and negative costs

Objective values by \textit{clasp} refer to rewritten (rather than initial) input
Lexicographical Optimization

- In case of several objectives, priority can (and should) be declared
  - Priority \( P \) in "\( @ P \)" is greater than \( P-1 \) in "\( @ P-1 \)"

<table>
<thead>
<tr>
<th>Error-prone Declaration</th>
<th>Clean Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>#minimize{ unimportant }.</td>
<td></td>
</tr>
<tr>
<td>#minimize{ important }.</td>
<td></td>
</tr>
<tr>
<td>#minimize{ unimportant @ 1 }.</td>
<td></td>
</tr>
<tr>
<td>#minimize{ important @ 2 }.</td>
<td></td>
</tr>
</tbody>
</table>

Priorities of Costs

#minimize[ unimportant = 2 @ 1, important = 1 @ 2 ].

- Unless specified otherwise, clasp improves tuples of objective values
  - Cumbersome in case of plenty intermediate solutions
- Option --opt-hierarch of clasp offers descents relative to priorities
Meta-Programming: Overview

17 Motivation

18 Saturation

19 Meta-Optimization
Overview

17 Motivation

18 Saturation

19 Meta-Optimization
Complex Preferences

- Preferences, e.g., based on inclusion minimality or Pareto efficiency, can lead to a significant increase in computational complexity by combining an \( NP \) with a \( coNP \) problem, where
  1. the first one defines feasible solutions and
  2. the second one checks that there are no better solutions
- Such preferences can be modeled in disjunctive ASP via saturation
  ❁ Saturation is quite involved and hardly usable by ASP laymen
- Complex preferences are vital in
  - Argumentation
  - Belief change
  - Bio-informatics
  - Circumscription
  - Decision making
  - Diagnosis
  - Inconsistency handling
  - System design
  - etc.
Circumscription

Given clauses $C = \{p \lor q, q \lor \neg r\}$ over atoms $V = \{p, q, r\}$, we get

1. Truth assignments
2. Models
3. Models minimal on $q$ with $p$ fixed (and $r$ allowed to vary)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$r$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Overview

17 Motivation

18 Saturation

19 Meta-Optimization
Ingredients

Solution Candidates

- “Normal” logic program with solution candidates as stable models
  - Does not implement complex (optimality) conditions

Counterexamples

- Disjunctive logic program with counterexamples as stable models
  1. Disjunctive rules allow for guessing potential counterexamples
  2. Definite rules yield error-indicating atom $\bot$ for invalid counterexamples
  3. Saturation immediately derives all atoms (of disjunctions) from $\bot$
     - Minimality of stable models (wrt. reduct) aims at avoiding $\bot$
  4. Query stipulates $\bot$ to hold
     - Necessity of $\bot$ indicates non-existence of any valid counterexample
     - Atoms standing for (non-existing) counterexamples are “meaningless”
Solution Candidates for Circumscription

- Recall clauses $C = \{ p \lor q, q \lor \neg r \}$ over atoms $V = \{ p, q, r \}$

Generating Models

% DOMAIN

lit(1,pos,p).  lit(2,pos,q).
lit(1,pos,q).  lit(2,neg,r).

var(X) :- lit(_,_,X).
cls(C) :- lit(C,_,_).

% GENERATE

{ hold(X) } :- var(X).
:- cls(C), hold(X) : lit(C,neg,X), not hold(X) : lit(C,pos,X).

힌  Models: $\{ p \}, \{ p, q \}, \{ p, q, r \}, \{ q \}, \{ q, r \}$
Counterexamples for Circumscription

- Models: \{p\}, \{p, q\}, \{p, q, r\}, \{q\}, \{q, r\}
- Minimal on q with p fixed (and r allowed to vary)

Generating Counterexamples

% GENERATE (counterexample)
true(X) | fail(X) :- var(X).

% DEFINE (counterexample)
bot :- cls(C), true(X) : lit(C,neg,X), fail(X) : lit(C,pos,X).

bot :- fix(X), true(X), not hold(X). bot :- fix(X), fail(X), hold(X).

same(X) :- min(X), true(X). bot :- min(X), true(X), not hold(X).
same(X) :- min(X), not hold(X). bot :- same(X) : min(X).

% TEST (non-existence of counterexample)
true(X) :- var(X), bot. fail(X) :- var(X), bot. :- not bot.
Overview

17 Motivation

18 Saturation

19 Meta-Optimization
**Methodology**

**Goal:** Relieve user from making saturation encodings by hand

**Approach**

1. Reification of “normal” program along with `#minimize(#maximize)`
   - Structural representation of ground instantiation in facts
2. Meta-encoding of optimal stable models reinterpreting `#minimize`
   a. “Normal” subprogram providing solution candidates
   b. Disjunctive subprogram providing counterexamples
   c. Comparator subprogram selecting undominated solution candidates

- Hierarchy of objectives via priorities
- Pareto semantics via different weights (for literals of same priority)
- Comparison in terms of (multiset) cardinality, inclusion, or preferences

Optimal (wrt. custom criteria) stable models of object program are read off from stable models of meta-encoding (on reified program)
Circumscription Revisited

- **Given:** Clauses $C = \{ p \lor q, q \lor \neg r \}$ over atoms $V = \{ p, q, r \}$
- **Wanted:** Models minimal on $q$ with $p$ fixed (and $r$ allowed to vary)

### “Normal” Encoding: circ.lp

```
% DOMAIN
var(X) :- lit(_,_,X).
cls(C) :- lit(C,_,_).

% GENERATE
{ hold(X) } :- var(X).
:- cls(C), hold(X) : lit(C,neg,X), not hold(X) : lit(C,pos,X).

% OPTIMIZE
#minimize{ hold(X) : min(X), hold(X) : fix(X), not hold(X) : fix(X) }.
```

This does the job when interpreting `#minimize` in terms of inclusion
gringo --reify form.lp circ.lp

% ...

wlist(0,0,pos(atom(hold(p))),1).
rule(pos(sum(0,0,1)),pos(conjunction(0))). % { hold(p) }.
wlist(1,0,pos(atom(hold(q))),1).
rule(pos(sum(0,1,1)),pos(conjunction(0))). % { hold(q) }.
wlist(2,0,pos(atom(hold(r))),1).
rule(pos(sum(0,2,1)),pos(conjunction(0))). % { hold(r) }.
set(1,neg(atom(hold(q)))).
set(1,neg(atom(hold(p)))).
rule(pos(false),pos(conjunction(1))). % :- not hold(p), not hold(q).
set(2,pos(atom(hold(r)))).
set(2,neg(atom(hold(q)))).
rule(pos(false),pos(conjunction(2))). % :- hold(r), not hold(q).
wlist(3,0,pos(atom(hold(p))),1).
wlist(3,1,pos(atom(hold(q))),1).
wlist(3,2,neg(atom(hold(p))),1).
minimize(1,3). % #minimize[ hold(q) = 1 @1, hold(p) = 1 @1, not hold(p) = 1 @1 ].

Reified programs can be fed back into gringo
Custom Optimization via Meta-Encoding

http://www.cs.uni-potsdam.de/wv/metasp/

Facts from reification can be combined with meta-encoding files:
1. `meta.lp` encoding stable models serving as solution candidates
2. `metaD.lp` encoding stable models serving as counterexamples
3. `meta0.lp` comparing candidates to counterexamples for optimality

Customizable comparison criteria: cardinality, inclusion, preferences

Comparison(s) in terms of inclusion: `incl.lp`

```
optimize(P,W,incl) :- minimize(P,L), wlist(L,_,_,W).
```

Queries regarding optimal stable models can be posted at meta-level:

Check for optimal stable models such that `hold(r)` holds: `query.lp`

```
:- not hold(atom(hold(r))).
```
Circumscription via Meta-Optimization

- Models: \{p\}, \{p, q\}, \{p, q, r\}, \{q\}, \{q, r\}
  - Minimal on q with p fixed (and r allowed to vary)
  - Where r holds

```
gringo --reify form.lp circ.lp | \ngringo - meta.lp metaD.lp meta0.lp incl.lp query.lp | claspD 0
```

Answer: 1

```
hold(atom(hold(r))) hold(atom(hold(q))) hold(atom(cls(1))) ...
```

Answer: 2

```
hold(atom(hold(q))) hold(atom(cls(1))) ...
```

Answer: 3

```
hold(atom(hold(p))) hold(atom(cls(1))) ...
```
Summary

- Meta-programming allows for reinterpreting input language constructs
  - Reification yields term-level representation of (ground) object programs
  - Meta-encoding applied to reified program gives input for an ASP solver
- Approach used to implement complex preferences in a general fashion
  - Stable models of object program serve as solution candidates
  - Stable models of object program serve as counterexamples
  - Comparison selects undominated solution candidates (custom criteria)
- Meta-encoding eases access to the expressive power of disjunctive ASP
  - Relieves user from making saturation encodings by hand
  - Competitive with hand-written saturation encodings (in case studies)
Temporal Reasoning: Overview

- Motivation
- Blocks World Planning
- Incremental Grounding and Solving
Overview

Motivation

Blocks World Planning

Incremental Grounding and Solving
Temporal Problems

- Many real-world applications, involving exponential state spaces, like
  - Automated planning,
  - Biological systems,
  - Model checking,
  - etc.

  have associated $PSPACE$-decision problems relative to time

Example

- Plan existence problem of deterministic planning is $PSPACE$-complete
  - But existence of a plan of (polynomially) bounded length is in $NP$

⇒ Let’s have a look at (planning) problems relying on time!
Overview

20 Motivation

21 Blocks World Planning

22 Incremental Grounding and Solving
Problem Instance

Block(1..9)  Step(1..10)

3  6  9
2  5  8
1  4  7

8  6  5
7  4  3
2  1  9

world.lp

block(1..9).  step(1..10).

init(3,2).  init(6,5).  init(9,8).
init(2,1).  init(5,4).  init(8,7).
init(1,0).  init(4,0).  init(7,0).

goal(8,6).
goal(6,4).  goal(5,7).
goal(4,2).  goal(7,3).
goal(2,1).  goal(3,9).
```
blocks0.lp

% DOMAIN
location(0). location(B) :- block(B).
% GENERATE
{ move(B,L,T) } :- block(B), location(L), step(T), B != L.

% Make at most one move per time
object(B,T) :- move(B,_,T). target(L,T) :- move(_,L,T).
:- step(T), 2 #count{ object(_,T) }. :- step(T), 2 #count{ target(_,T) }.

% Propagate move effects
on(B,L,0) :- init(B,L). on(B,L,T) :- move(B,L,T).
on(B,L,T) :- on(B,L,T-1), step(T), not object(B,T).

% Move (to) free blocks only
blocked(B,T) :- on(_,B,T), block(B), step(T+1).
:- object(B,T), blocked(B,T-1). :- target(B,T), blocked(B,T-1).

% Assert goal conditions
:- goal(B,L), step(T), not step(T+1), not on(B,L,T).
```
### “Children’s” Problem Encoding II

<table>
<thead>
<tr>
<th>Command</th>
<th>Execution Time</th>
<th>Solving Time</th>
<th>First Model Time</th>
<th>Unsat Time</th>
<th>Conflicts</th>
<th>Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>gringo blocks0.lp world.lp</td>
<td>0.021s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>6</td>
<td>2730</td>
<td>6211</td>
</tr>
<tr>
<td>clasp --heu=vsids --stats</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Execution Time</th>
<th>Solving Time</th>
<th>First Model Time</th>
<th>Unsat Time</th>
<th>Conflicts</th>
<th>Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>gringo blocks0.lp world9n.lp -c n=4</td>
<td>2179.760s</td>
<td>2177.93s</td>
<td>2177.93s</td>
<td>0.00s</td>
<td>1572121</td>
<td>192744</td>
<td>543330</td>
</tr>
<tr>
<td>clasp --heu=vsids --stats</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
First Encoding Refinement

Moving blocks on top of other blocks makes sense in view of goal only

blocks1.lp

% GENERATE
{ move(B,L,T) } :- block(B), location(L), step(T), goal(B,L) : block(L).

gringo blocks1.lp world9n.lp -c n=4 | clasp --heu=vsids --stats

Time : 3.139s (Solving: 2.98s 1st Model: 2.97s Unsat: 0.00s)
Conflicts : 60973
Variables : 16842
Constraints : 35114
“Redundant” State Constraints

State constraints hold at each point in time (no matter the actions)

blocks2.lp

% Block locations are unambiguous
:- block(B), step(T), 2 #count{ on(B,_,T) }.
:- block(B), step(T), 2 #count{ on(_,B,T) }.

% Each block rests on table
above(B,T) :- on(B,0,T), step(T).
above(B,T) :- on(B,L,T), above(L,T).
:- block(B), step(T), not above(B,T).

gringo blocks2.lp world9n.lp -c n=5 | clasp --heu=vsids --stats

Time : 2.622s (Solving: 2.39s 1st Model: 2.39s Unsat: 0.00s)
Conflicts : 53027
Parallel Actions

Actions may take place “in parallel” if there is some sequential order

blocks3.lp

% Project moves
object(B,T) :- move(B,_,T).
target(L,T) :- move(_,L,T).
:- step(T), 2 #count{ object(_,T) }.
:- step(T), 2 #count{ target(_,T) }.

gringo blocks3.lp world9n.lp -c n=5 | clasp --heu=vsids --stats

| Time     | 0.253s (Solving: 0.01s 1st Model: 0.00s Unsat: 0.00s) |
| Conflicts| 1                                                      |
Minimal Horizon Plans

Let’s look at plans of minimum (parallel) length

blocksM.lp

% OPTIMIZE
move(T) :- object(_,T).
#minimize{ move(_) }.

gringo blocks3.lp world9n.lp -c n=9 blocksM.lp | /
clasp --heu=vsids --stats

Time : 54.903s (Solving: 53.86s 1st Model: 0.03s Unsat: 53.76s)
Conflicts : 52670

See Potassco book for further encoding enhancements:
Overview

20 Motivation

21 Blocks World Planning

22 Incremental Grounding and Solving
The *iclingo* System

- Incremental grounding and solving
- Offline solving in dynamic domains, like automated planning
- Basic architecture of *iclingo*:

```
	gringo

	clasp
```

```
iclingo
```
Incremental ASP Solving Process

- **Logic Program:** $P_k$
- **Grounding:** $Q_k$
- **Solver:**
- **Answer:**

**Modeling:**

- $B$
- $Q_k$
- $Q_3$
- $Q_n$
- $P_n$
### Incremental Blocks World

#### blocks3i.lp

**#base. % Program part B**

% DOMAIN

location(0). location(B) :- block(B).

on(B,L,0) :- init(B,L).

**#cumulative t. % Program part P[t]**

% GENERATE

{ move(B,L,t) } :- block(B), location(L),
    goal(B,L) : block(L).

% Project moves

object(B,t) :- move(B,_,t).

target(L,t) :- move(_,L,t).

% Propagate move effects

on(B,L,t) :- move(B,L,t).

on(B,L,t) :- on(B,L,t-1), not object(B,t).

% Move (to) free blocks only

blocked(B,t-1) :- on(_,B,t-1), block(B).

:- object(B,t), blocked(B,t-1).

:- target(B,t), blocked(B,t-1).

% Block locations are unambiguous

:- block(B), 2 #count{ on(B,_,t) }.

:- block(B), 2 #count{ on(_,B,t) }.

% Each block rests on table

above(B,t) :- on(B,0,t).

above(B,t) :- on(B,L,t), above(L,t).

:- block(B), not above(B,t).

**#volatile t. % Program part Q[t]**

% Assert goal conditions

:- goal(B,L), not on(B,L,t).
## Stemming Blocks Incrementally

Let’s look at plans of minimum (parallel) length

```plaintext
iclingo blocks3i.lp world9n.lp -c n=9 --stats
```

<table>
<thead>
<tr>
<th>Total Steps :</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time        :</td>
<td>0.030</td>
</tr>
<tr>
<td>Choices     :</td>
<td>21</td>
</tr>
<tr>
<td>Conflicts   :</td>
<td>0</td>
</tr>
</tbody>
</table>
Modeling Incrementally

- Incremental ASP solving addresses problems without a priori horizons

- Base, cumulative, and volatile program parts must be “compositional”
  - Include $t+c$ (for a unique $c$) in head atoms of cumulative/volatile rules
  - Do not, in rule bodies, refer to atoms defined by later program parts

- Aim at high level of reuse of incrementally generated ground rules
  - Volatile program part should be as slim as possible
  - Consider alternatives to “standard” encodings for single-pass processing