# Declarative Modeling

Winter Term 2016/2017
Lecture Slides

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

# Rough Roadmap

1. Background
   - Grounding
   - Solving
2. Modeling
   - Satisfiability
   - Optimization
   - Incrementality
3. Applications
   - Puzzles
   - Planning
   - Scheduling

## Resources

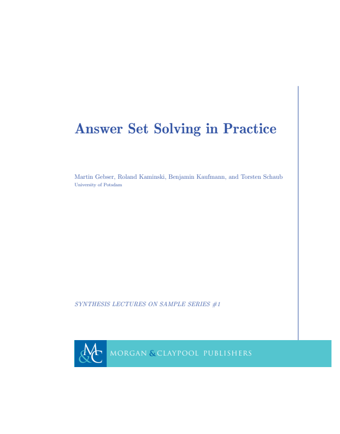
- Course material
  - `https://moodle.cs.uni-potsdam.de/course/view.php?id=39`
  - `https://potassco.org/teaching/`
  - `https://potassco.org/support/`
- Systems
  - *clasp*
  - *gringo*
  - *clingo*

`https://potassco.org`

# The Potassco Book

https://potassco.org/book/

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



# Administrative Information

- Lecture: 2x (at the beginning)
- Project: implementation and presentation (at end of semester)
- Credits: 6

- C(ourse)MS:
  `https://moodle.cs.uni-potsdam.de/course/view.php?id=39`
- Enrollment:
  `https://puls.uni-potsdam.de`

- Mark: Project implementation and presentation
  - Area: "Praktische Informatik" / "Angewandte Informatik" / "Wahlfrei"
  - Examiner (@PULS): Torsten Schaub

# Administrative Information

- Lecture: 2x (at the beginning)
- Project: implementation and presentation (at end of semester)
- Credits: 6

- C(ourse)MS:
  `https://moodle.cs.uni-potsdam.de/course/view.php?id=39`
- Enrollment:
  `https://puls.uni-potsdam.de`

- Mark: Project implementation and presentation
  - Area: "Praktische Informatik" / "Angewandte Informatik" / "Wahlfrei"
  - Examiner (@PULS): Torsten Schaub

# Administrative Information

- Lecture: 2x (at the beginning)
- Project: implementation and presentation (at end of semester)
- Credits: 6

- C(ourse)MS:
  `https://moodle.cs.uni-potsdam.de/course/view.php?id=39`
- Enrollment:
  `https://puls.uni-potsdam.de`

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

# Background: Overview

# Overview

# Human versus Computational Intelligence?


Picture from Wikimedia Commons

vs.

Picture from Wikimedia Commons

| | | | |
|---|---|---|---|
| Speed | ✔ | Speed | ✘ |
| Accuracy | ✔ | Accuracy | ✘ |
| Intuitions | | Intuitions | |
| Knowledge | | Knowledge | |

# Human versus Computational Intelligence?

Picture from Wikimedia Commons

- Speed ✔
- Accuracy ✔
- Intuitions ✘
- Knowledge ✘

vs.


Picture from Wikimedia Commons

- Speed ✘
- Accuracy ✘
- Intuitions ✔
- Knowledge ✔

# Human versus Computational Intelligence?


Picture from Wikimedia Commons

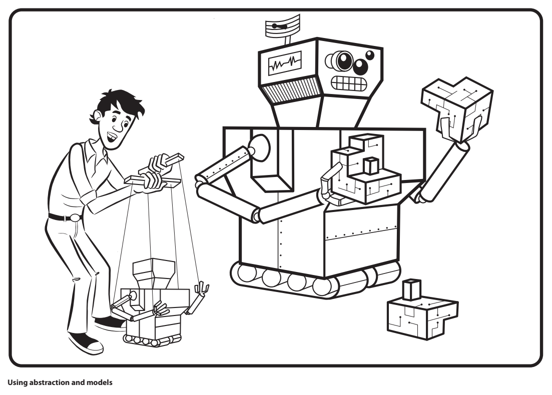
vs.

Picture from Wikimedia Commons

| | |
|---|---|
| Speed | ✔ |
| Accuracy | ✔ |
| Intuitions | ✘ |
| Knowledge | ✘ |

| | |
|---|---|
| Speed | ✘ |
| Accuracy | ✘ |
| Intuitions | ✔ |
| Knowledge | ✔ |

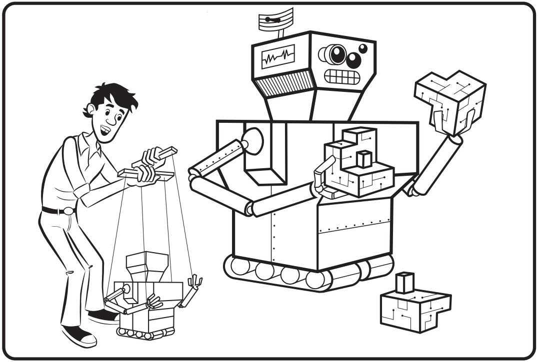
# Human plus Computational Intelligence!




Picture from Computational Thinking Illustrated

**KRR = Knowledge + Representation + Reasoning**

# Human plus Computational Intelligence!


Using abstraction and models

Picture from Computational Thinking Illustrated

**KRR = Knowledge + Representation + Reasoning**

# Illustration: Pigeonhole Principle

## One small step for man

Can one put 10 pigeons into 9 holes such that no pigeons share a hole?

# Illustration: Pigeonhole Principle

## One small step for man

Can one put 10 pigeons into 9 holes such that no pigeons share a hole?

**NO!**

Picture from Wikipedia, the free encyclopedia

# 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!**

Picture from Wikipedia, the free encyclopedia

# 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

| Each pigeon requires a hole | No pigeons share a hole | | |
|---|---|---|---|
| $p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n}$ | $\neg p_{1,1} \vee \neg p_{2,1}$ | $\ldots$ | $\neg p_{1,n} \vee \neg p_{2,n}$ |
| $p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n}$ | $\neg p_{1,1} \vee \neg p_{3,1}$ | $\ldots$ | $\neg p_{1,n} \vee \neg p_{3,n}$ |
| $\vdots$ | $\vdots$ | $\ldots$ | $\vdots$ |
| $p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n}$ | $\neg p_{n-1,1} \vee \neg p_{n+1,1}$ | $\ldots$ | $\neg p_{n-1,n} \vee \neg p_{n+1,n}$ |
| $p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n}$ | $\neg p_{n,1} \vee \neg p_{n+1,1}$ | $\ldots$ | $\neg p_{n,n} \vee \neg p_{n+1,n}$ |

☞ Runtime of (resolution-based) solvers:

# 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

| Each pigeon requires a hole | No pigeons share a hole | | |
|---|---|---|---|
| $p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n}$ | $\neg p_{1,1} \vee \neg p_{2,1}$ | $\ldots$ | $\neg p_{1,n} \vee \neg p_{2,n}$ |
| $p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n}$ | $\neg p_{1,1} \vee \neg p_{3,1}$ | $\ldots$ | $\neg p_{1,n} \vee \neg p_{3,n}$ |
| $\vdots$ | $\vdots$ | $\ldots$ | $\vdots$ |
| $p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n}$ | $\neg p_{n-1,1} \vee \neg p_{n+1,1}$ | $\ldots$ | $\neg p_{n-1,n} \vee \neg p_{n+1,n}$ |
| $p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n}$ | $\neg p_{n,1} \vee \neg p_{n+1,1}$ | $\ldots$ | $\neg p_{n,n} \vee \neg p_{n+1,n}$ |

☞ Runtime of (resolution-based) solvers:

# 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

**Each pigeon requires a hole**

$$\begin{array}{l}
p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n} \\
p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n} \\
\vdots \\
p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n} \\
p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n}
\end{array}$$

**No pigeons share a hole**

$$\begin{array}{lcl}
\neg p_{1,1} \vee \neg p_{2,1} & \ldots & \neg p_{1,n} \vee \neg p_{2,n} \\
\neg p_{1,1} \vee \neg p_{3,1} & \ldots & \neg p_{1,n} \vee \neg p_{3,n} \\
\vdots & \ldots & \vdots \\
\neg p_{n-1,1} \vee \neg p_{n+1,1} & \ldots & \neg p_{n-1,n} \vee \neg p_{n+1,n} \\
\neg p_{n,1} \vee \neg p_{n+1,1} & \ldots & \neg p_{n,n} \vee \neg p_{n+1,n}
\end{array}$$

☞ Runtime of (resolution-based) solvers:

# 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

#### Each pigeon requires a hole

$$\begin{array}{l}
p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n} \\
p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n} \\
\vdots \\
p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n} \\
p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n}
\end{array}$$

#### No pigeons share a hole

$$\begin{array}{lcl}
\neg p_{1,1} \vee \neg p_{2,1} & \ldots & \neg p_{1,n} \vee \neg p_{2,n} \\
\neg p_{1,1} \vee \neg p_{3,1} & \ldots & \neg p_{1,n} \vee \neg p_{3,n} \\
\vdots & \ldots & \vdots \\
\neg p_{n-1,1} \vee \neg p_{n+1,1} & \ldots & \neg p_{n-1,n} \vee \neg p_{n+1,n} \\
\neg p_{n,1} \vee \neg p_{n+1,1} & \ldots & \neg p_{n,n} \vee \neg p_{n+1,n}
\end{array}$$

☞ Runtime of (resolution-based) solvers:

# 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?

### "Naive" Formulation

**Each pigeon requires a hole**

$$\begin{array}{c} p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n} \\ p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n} \\ \vdots \\ p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n} \\ p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n} \end{array}$$

**No pigeons share a hole**

$$\begin{array}{lcl} \neg p_{1,1} \vee \neg p_{2,1} & \dots & \neg p_{1,n} \vee \neg p_{2,n} \\ \neg p_{1,1} \vee \neg p_{3,1} & \dots & \neg p_{1,n} \vee \neg p_{3,n} \\ \vdots & \dots & \vdots \\ \neg p_{n-1,1} \vee \neg p_{n+1,1} & \dots & \neg p_{n-1,n} \vee \neg p_{n+1,n} \\ \neg p_{n,1} \vee \neg p_{n+1,1} & \dots & \neg p_{n,n} \vee \neg p_{n+1,n} \end{array}$$

☞ Runtime of (resolution-based) solvers:

# 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

**Each pigeon requires a hole**

$$
\begin{array}{l}
p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n} \\
p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n} \\
\vdots \\
p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n} \\
p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n}
\end{array}
$$

**No pigeons share a hole**

$$
\begin{array}{lcl}
\neg p_{1,1} \vee \neg p_{2,1} & \ldots & \neg p_{1,n} \vee \neg p_{2,n} \\
\neg p_{1,1} \vee \neg p_{3,1} & \ldots & \neg p_{1,n} \vee \neg p_{3,n} \\
\vdots & \ldots & \vdots \\
\neg p_{n-1,1} \vee \neg p_{n+1,1} & \ldots & \neg p_{n-1,n} \vee \neg p_{n+1,n} \\
\neg p_{n,1} \vee \neg p_{n+1,1} & \ldots & \neg p_{n,n} \vee \neg p_{n+1,n}
\end{array}
$$

☞ Runtime of (resolution-based) solvers:

# 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

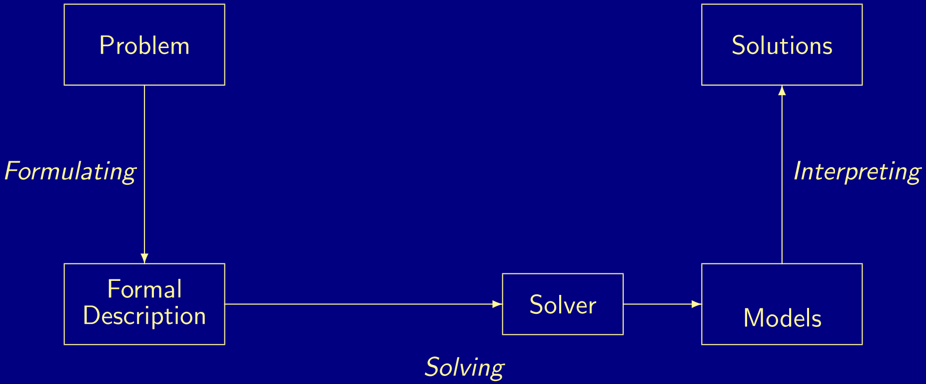
| Each pigeon requires a hole | No pigeons share a hole |
|---|---|
| $p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,n}$ | $\neg p_{1,1} \vee \neg p_{2,1} \quad \ldots \quad \neg p_{1,n} \vee \neg p_{2,n}$ |
| $p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,n}$ | $\neg p_{1,1} \vee \neg p_{3,1} \quad \ldots \quad \neg p_{1,n} \vee \neg p_{3,n}$ |
| $\vdots$ | $\vdots \quad \ldots \quad \vdots$ |
| $p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,n}$ | $\neg p_{n-1,1} \vee \neg p_{n+1,1} \quad \ldots \quad \neg p_{n-1,n} \vee \neg p_{n+1,n}$ |
| $p_{n+1,1} \vee p_{n+1,2} \vee \cdots \vee p_{n+1,n}$ | $\neg p_{n,1} \vee \neg p_{n+1,1} \quad \ldots \quad \neg p_{n,n} \vee \neg p_{n+1,n}$ |

☞ Runtime of (resolution-based) solvers: 

# Overview

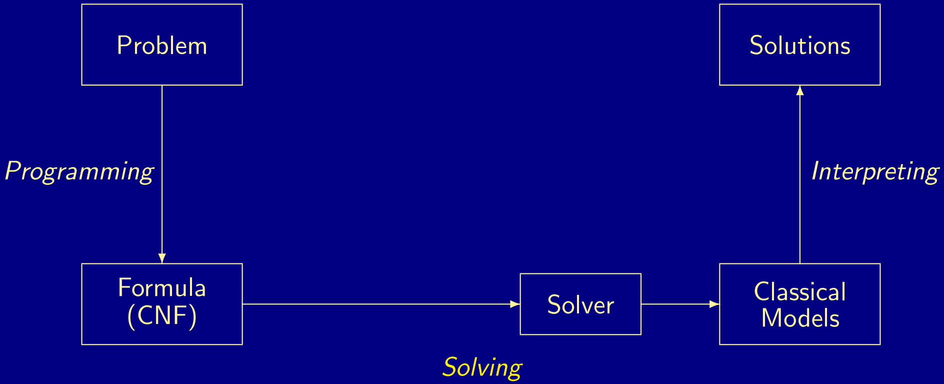
# Declarative Problem Solving

with SAT



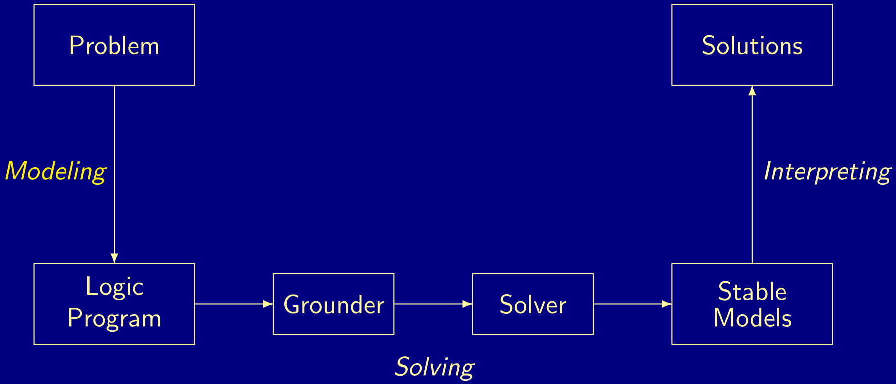
# Declarative Problem Solving

## with SAT



# Declarative Problem Solving

## with ASP



## Modeling Constructs

- Variables `p(X) :- q(X).`
- Conditional literals `p :- q(X) : r(X).`
- Disjunction `p(X) ; q(X) :- r(X).`
- Integrity constraints `:- q(X), p(X).`
- Choice `2 { p(X,Y) : q(X) } 7 :- r(Y).`
- Aggregates `s(Y) :- r(Y), 2 #sum{ X : p(X,Y), q(X) } 7.`
- Optimization
  - Weak constraints `:~ q(X), p(X,C). [C,X]`
  - Statements `#minimize { C,X : q(X), p(X,C) }.`

# Modeling Constructs

- Variables `p(X) :- q(X).`
- Conditional literals `p :- q(X) : r(X).`
- Disjunction `p(X) ; q(X) :- r(X).`
- Integrity constraints `:- q(X), p(X).`
- Choice `2 { p(X,Y) : q(X) } 7 :- r(Y).`
- Aggregates `s(Y) :- r(Y), 2 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization
  - Weak constraints `:~ q(X), p(X,C). [C@42,X]`
  - Statements `#minimize { C@42,X : q(X), p(X,C) }.`

# Reasoning Modes

- Satisfiability
- Optimization
- Enumeration†
- Projection†
- Intersection‡
- Union‡

- and combinations of them
- via single- or multi-threading

† without solution recording

‡ without solution enumeration

# Reasoning Modes

- Satisfiability
- Optimization
- Enumeration[†]
- Projection[†]
- Intersection[‡]
- Union[‡]

- and combinations of them
- via single- or multi-threading

[†] without solution recording

[‡] without solution enumeration

# 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

# 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

# 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

# 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
  - `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 = #min{ Y : p(Y) }`, etc.
  - Bind (global) variables on one side, if those on other 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

# Predicate Classification

## Built-in Predicates

- Term comparisons
  - `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 = #min{ Y : p(Y) }`, etc.
  - ☞ Bind (global) variables on one side, if those on other 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

# Predicate Classification

## Built-in Predicates

- Term comparisons
  - `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 = #min{ Y : p(Y) }`, etc.
  - ☞ Bind (global) variables on one side, if those on other 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

# Predicate Classification

## Built-in Predicates

- Term comparisons
  - `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 = #min{ Y : p(Y) }`, etc.
  - ☞ Bind (global) variables on one side, if those on other 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

# Predicate Classification

## Built-in Predicates

- Term comparisons
  - `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 = #min{ Y : p(Y) }`, etc.

  ☞ Bind (global) variables on one side, if those on other 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

## 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

Find some stable model (quickly)

Build some refutation (quickly), includes optimality proofs

# 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

Find some stable model (quickly)

Build some refutation (quickly), includes optimality proofs

# 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

Find some stable model (quickly)

Build some refutation (quickly), includes optimality proofs

# 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

# 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

# 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

# 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

# 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.)

# 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.)

# 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.)

# 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.)

# 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.)

# 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.)

# 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.)

# 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.)

# 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

# Running Example: Latin Square

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

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |

represented by facts:

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

**Wanted:** Assignment of $1, \dots, N$

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

# Running Example: Latin Square

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

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |

represented by facts:

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

**Wanted:** Assignment of $1, \ldots, N$

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 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(1,2,2) ... 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)
```

# Linearizing Inequality Tests

## 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
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

```
gringo latin0.lp | wc
```

1054728 6288392 21030438

```
gringo latin1.lp | wc
```

227336 1199112 4773202

# Linearizing Inequality Tests

## 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
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

```
gringo latin0.lp | wc
```

1054728 6288392 21030438

```
gringo latin1.lp | wc
```

227336 1199112 4773202

# Linearizing Inequality Tests

## 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
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

```
gringo latin0.lp | wc
```

1054728 6288392 21030438

```
gringo latin1.lp | wc
```

227336 1199112 4773202

# Linearizing Inequality Tests

## 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).

% DEFINE + TEST
occ(1,X-1,Y,    N) :- num(X,Y,N), square(X-1,Y).
occ(0,X,  Y-1,  N) :- num(X,Y,N), square(X,Y-1).
occ(D,X-D,Y+D-1,N) :- occ(D,X,Y,N), square(X-D,Y+D-1).
:- num(X,Y,N), occ(D,X,Y,N).
```

```
gringo latin0.lp | wc
```

1054728 6288392 21030438

```
gringo latin1.lp | wc
```

227336 1199112 4773202

# Linearizing Inequality Tests

## 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).

% DEFINE + TEST
occ(1,X-1,Y,    N) :- num(X,Y,N), square(X-1,Y).
occ(0,X,  Y-1,  N) :- num(X,Y,N), square(X,Y-1).
occ(D,X-D,Y+D-1,N) :- occ(D,X,Y,N), square(X-D,Y+D-1).
:- num(X,Y,N), occ(D,X,Y,N).
```

```
gringo latin0.lp | wc
```

1054728 6288392 21030438

```
gringo latin1.lp | wc
```

227336 1199112 4773202

# Linearizing Inequality Tests

## 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).

% DEFINE + TEST
occ(1,X-1,Y,    N) :- num(X,Y,N), square(X-1,Y).
occ(0,X,  Y-1,  N) :- num(X,Y,N), square(X,Y-1).
occ(D,X-D,Y+D-1,N) :- occ(D,X,Y,N), square(X-D,Y+D-1).
:- num(X,Y,N), occ(D,X,Y,N).
```

```
gringo latin0.lp | wc
```

1054728 6288392 21030438

```
gringo latin1.lp | wc
```

227336 1199112 4773202

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.
:- occ(D,Z,N,C), C != 1.

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

```
gringo latin2.lp | wc        gringo latin3.lp | wc
```

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.
:- occ(D,Z,N,C), C != 1.

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

```
gringo latin2.lp | wc
```

```
gringo latin3.lp | wc
```

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.
:- occ(D,Z,N,C), C != 1.

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

gringo latin2.lp | wc gringo latin3.lp | wc

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.
:- occ(D,Z,N,C), C != 1.

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

```
gringo latin2.lp | wc
```

```
gringo latin3.lp | wc
```

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, #count{ Y : num(X,Y,N) }=C, C=0..n.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) }=C, C=0..n.
:- occ(D,Z,N,C), C != 1.

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

☞ Internal transformation by *gringo*

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.     ✗
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.     ✗
:- occ(D,Z,N,C), C != 1.

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

```
gringo latin2.lp | wc
```

```
gringo latin3.lp | wc
```

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.
:- occ(D,Z,N,C), C != 1.

% DISPLAY
#show num/3.
```

```
gringo latin2.lp | wc
```

```
gringo latin3.lp | wc
```

49160 375816 2189400

# Using Aggregates

## 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
occ(1,X,N,C) :- X = 1..n, N = 1..n, C = #count{ Y : num(X,Y,N) }.
occ(0,Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ X : num(X,Y,N) }.
:- occ(D,Z,N,C), C != 1.

% DISPLAY
#show num/3.
```

```
gringo latin2.lp | wc
```

366600 6089736 34019251

```
gringo latin3.lp | wc
```

49160 375816 2189400

# Using Aggregates

## 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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.


% DISPLAY
#show num/3.
```

```
gringo latin2.lp | wc
```

```
366600 6089736 34019251
```

```
gringo latin3.lp | wc
```

# Using Aggregates

## 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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.



% DISPLAY
#show num/3.
```

```
gringo latin2.lp | wc
```

366600 6089736 34019251

```
gringo latin3.lp | wc
```

49160 375816 2189400

# 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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.


% DISPLAY
#hide. #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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.


% DISPLAY
#hide. #show num/3.
```

☞ Many symmetric solutions (mirroring, rotation, value permutation)

# 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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.


% DISPLAY
#hide. #show num/3.
```

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

% DISPLAY
#hide. #show num/3.
```

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

% DISPLAY
#hide. #show num/3.
```

☞ Let's compare enumeration speed!

# Breaking Symmetries

## The ultimate Latin square encoding?

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

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

% TEST
:- X = 1..n, N = 1..n, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin3.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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.



% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin3.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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin4.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, #count{ Y : num(X,Y,N) } != 1.
:- Y = 1..n, N = 1..n, #count{ X : num(X,Y,N) } != 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin4.lp | clasp -q 0
```

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

# Term Order

- Investigating terms along an order often helps to encode compactly

☞ Linking successive terms

## Example

```
item(i1).    item(i2).
prop(i1,p1). prop(i2,p1).
prop(i1,p2). prop(i2,p2).
prop(i1,p3). prop(i2,p3).
prop(i1,p4).

prop(P) :- prop(I,P). % Projection

next(P1,P2) :- prop(P1), prop(P2), P1 < P2, P <= P1 : prop(P), P < P2.

init(P1) :- prop(P1), P1 <= P : prop(P).
last(P2) :- prop(P2), P <= P2 : prop(P).
```

# Term Order

- Investigating terms along an order often helps to encode compactly

☞ Linking successive terms

## Example

```
item(i1).    item(i2).
prop(i1,p1). prop(i2,p1).
prop(i1,p2). prop(i2,p2).
prop(i1,p3). prop(i2,p3).
prop(i1,p4).

prop(P) :- prop(I,P). % Projection

next(P1,P2) :- prop(P1), prop(P2), P1 < P2, P <= P1 : prop(P), P < P2.

init(P1) :- prop(P1), P1 <= P : prop(P).
last(P2) :- prop(P2), P <= P2 : prop(P).
```

# 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 continued

```
{ has(I,P) } :- prop(I,P).

count(I,P1,0)   :- item(I), init(P1), not has(I,P1).
count(I,P1,1)   :- item(I), init(P1), has(I,P1).
count(I,P2,N)   :- count(I,P1,N), next(P1,P2), not has(I,P2).
count(I,P2,N+1) :- count(I,P1,N), next(P1,P2), has(I,P2).

counted(I,N) :- count(I,P,N).
counted(N)   :- counted(I,N).

most(I) :- counted(I,N), not counted(N+1).
```

# 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 continued

```
{ has(I,P) } :- prop(I,P).

count(I,P1,0)   :- item(I), init(P1), not has(I,P1).
count(I,P1,1)   :- item(I), init(P1), has(I,P1).
count(I,P2,N)   :- count(I,P1,N), next(P1,P2), not has(I,P2).
count(I,P2,N+1) :- count(I,P1,N), next(P1,P2), has(I,P2).

counted(I,N) :- count(I,P,N).
counted(N)   :- counted(I,N).

most(I) :- counted(I,N), not counted(N+1).
```

# 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 continued

```
{ has(I,P) } :- prop(I,P).

count(I,P1,0)   :- item(I), init(P1), not has(I,P1).
count(I,P1,1)   :- item(I), init(P1), has(I,P1).
count(I,P2,N)   :- count(I,P1,N), next(P1,P2), not has(I,P2).
count(I,P2,N+1) :- count(I,P1,N), next(P1,P2), has(I,P2).

counted(I,N) :- count(I,P,N).
counted(N)   :- counted(I,N).

most(I) :- counted(I,N), not counted(N+1).
```

# 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 continued

```
{ has(I,P) } :- prop(I,P).

count(I,P1,0)   :- item(I), init(P1).
count(I,P1,1)   :- item(I), init(P1), has(I,P1).
count(I,P2,N)   :- count(I,P1,N), next(P1,P2).
count(I,P2,N+1) :- count(I,P1,N), next(P1,P2), has(I,P2).

counted(I,N) :- count(I,P,N).
counted(N)   :- counted(I,N).

most(I) :- counted(I,N), not counted(N+1).
```

# 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 continued

```
{ has(I,P) } :- prop(I,P).

count(I,P1,0)   :- item(I), init(P1).
count(I,P1,1)   :- item(I), init(P1), has(I,P1).
count(I,P2,N)   :- count(I,P1,N), next(P1,P2).
count(I,P2,N+1) :- count(I,P1,N), next(P1,P2), has(I,P2).

counted(I,N) :- count(I,P2,N), last(P2).
counted(N)   :- counted(I,N).

most(I) :- counted(I,N), not counted(N+1).
```

# 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!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

# Encode With Care!

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

# Encode With Care!

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

# Encode With Care!

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

# Encode With Care!

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

# Encode With Care!

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

# Some Hints on (Preventing) Debugging

## Kinds of errors

- Syntactic ☞ follow error messages by the grounder
- Semantic ☞ (most likely) encoding/intention mismatch

## Ways to identify semantic errors (early)

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

# Some Hints on (Preventing) Debugging

## Kinds of errors

- Syntactic ☞ follow error messages by the grounder
- Semantic ☞ (most likely) encoding/intention mismatch

## Ways to identify semantic errors (early)

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

# Some Hints on (Preventing) Debugging

## Kinds of errors

- Syntactic ☞ follow error messages by the grounder
- Semantic ☞ (most likely) encoding/intention mismatch

## Ways to identify semantic errors (early)

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

# Some Hints on (Preventing) Debugging

## Kinds of errors

- Syntactic ☞ follow error messages by the grounder
- Semantic ☞ (most likely) encoding/intention mismatch

## Ways to identify semantic errors (early)

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

# Some Hints on (Preventing) Debugging

## Kinds of errors

- Syntactic ☞ follow error messages by the grounder
- Semantic ☞ (most likely) encoding/intention mismatch

## Ways to identify semantic errors (early)

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

# Some Hints on (Preventing) Debugging

## Kinds of errors

- Syntactic ☞ follow error messages by the grounder
- Semantic ☞ (most likely) encoding/intention mismatch

## Ways to identify semantic errors (early)

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

# Overcoming Performance Bottlenecks

## Grounding

- Monitor time spent by and output size of *gringo*
  1. System tools (eg. `time(gringo [...] | wc)`)
  2. Internal transformations (eg. `gringo --output-debug=translate [...]`)

☞ Once identified, reformulate "critical" logic program parts

## Solving

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

  If great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. Try predefined configurations (use `clasp --configuration=[...]`)
  2. Try manual fine-tuning (requires expert knowledge!)
  3. 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. Internal transformations (eg. `gringo --output-debug=translate [...]`)

☞ Once identified, reformulate “critical” logic program parts

## Solving

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

☞ If great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. Try predefined configurations (use `clasp --configuration=[...]`)
  2. Try manual fine-tuning (requires expert knowledge!)
  3. 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. Internal transformations (eg. `gringo --output-debug=translate [...]`)

☞ Once identified, reformulate "critical" logic program parts

## Solving

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

☞ If great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. Try predefined configurations (use `clasp --configuration=[...]`)
  2. Try manual fine-tuning (requires expert knowledge!)
  3. 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. Internal transformations (eg. `gringo --output-debug=translate [...]`)

☞ Once identified, reformulate "critical" logic program parts

## Solving

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

☞ If great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. Try predefined configurations (use `clasp --configuration=[...]`)
  2. Try manual fine-tuning (requires expert knowledge!)
  3. 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. Internal transformations (eg. `gringo --output-debug=translate [...]`)

☞ Once identified, reformulate "critical" logic program parts

## Solving

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

☞ If great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. Try predefined configurations (use `clasp --configuration=[...]`)
  2. Try manual fine-tuning (requires expert knowledge!)
  3. 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. Internal transformations (eg. `gringo --output-debug=translate [...]`)

☞ Once identified, reformulate "critical" logic program parts

## Solving

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

☞ If great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. Try predefined configurations (use `clasp --configuration=[...]`)
  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

# Overview

# 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

# 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

# Hard versus Soft Constraints

## Hard Constraints

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

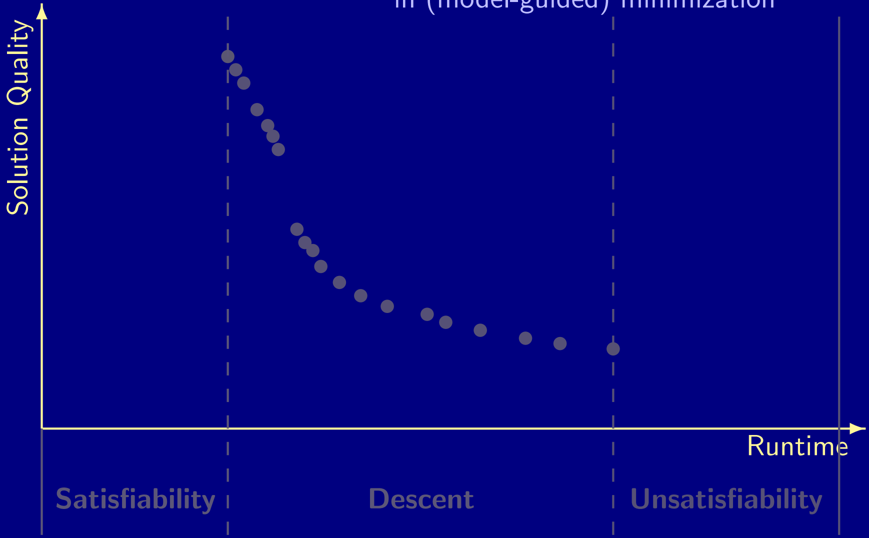
## Soft Constraints

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

☞ Discriminate viable solutions

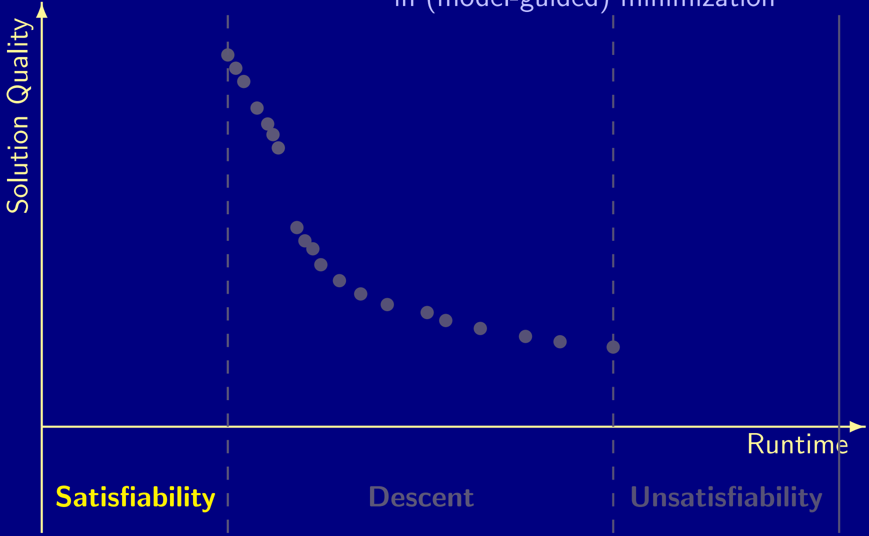
# Search Phases

in (model-guided) minimization



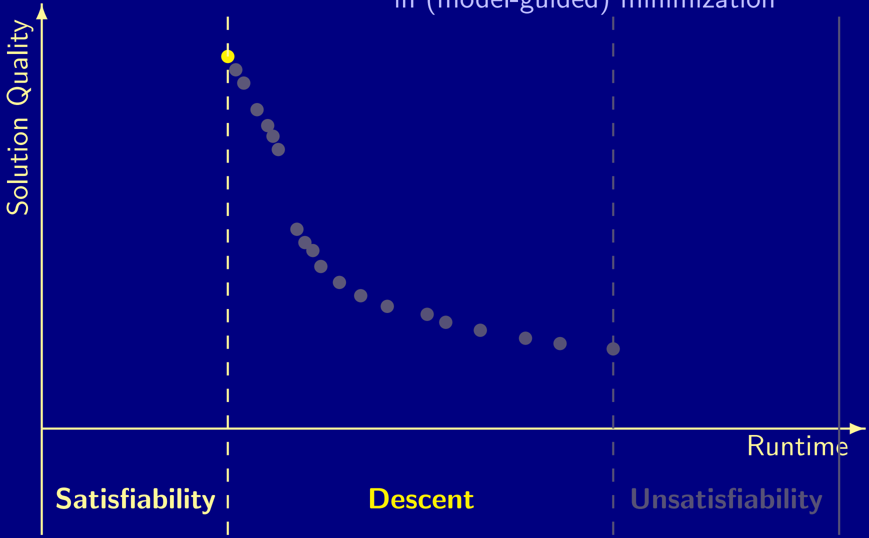
# Search Phases

## in (model-guided) minimization



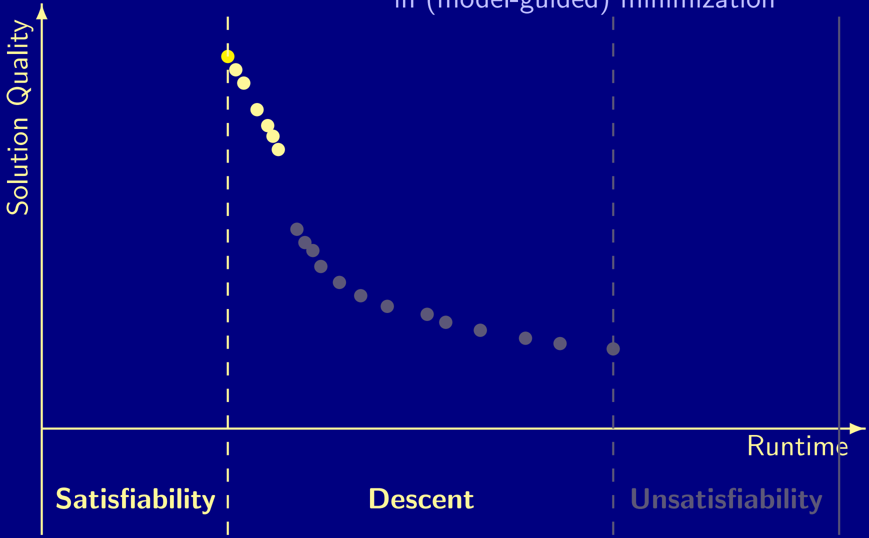
# Search Phases

in (model-guided) minimization



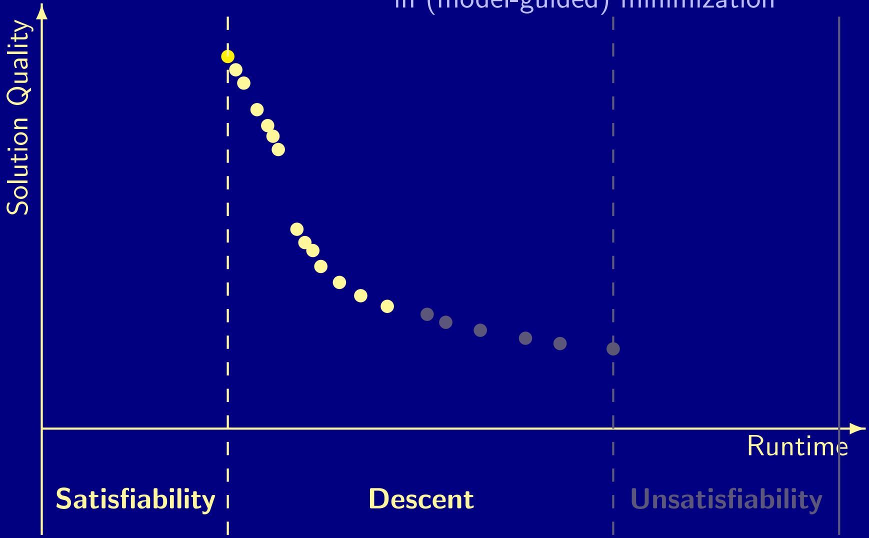
# Search Phases

in (model-guided) minimization



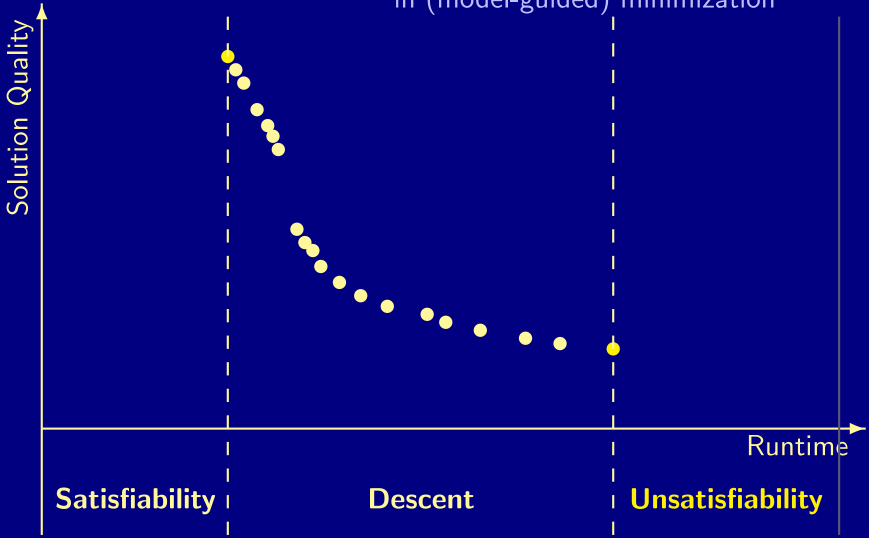
# Search Phases

in (model-guided) minimization



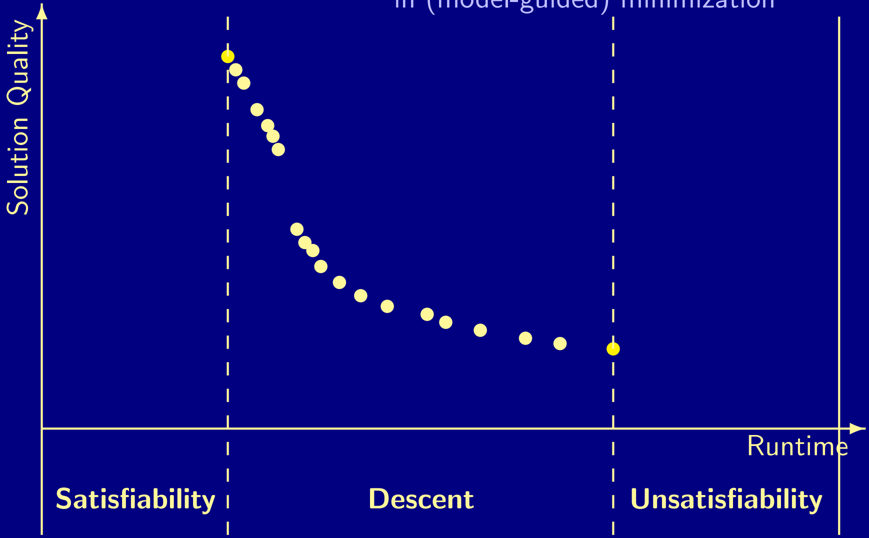
# Search Phases

in (model-guided) minimization



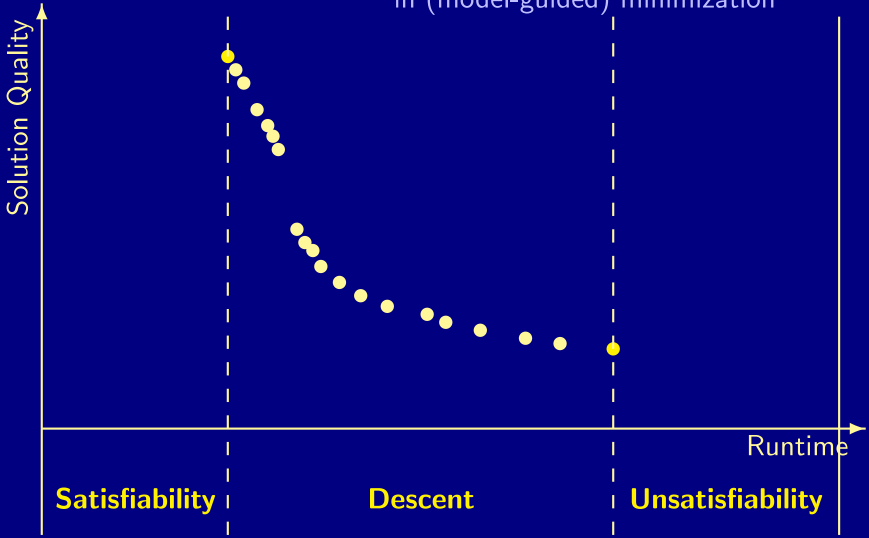
# Search Phases

in (model-guided) minimization



# Search Phases

in (model-guided) minimization



# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

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

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

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

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

☞ Add compact "redundant" constraints for shortcuts to (non-)solutions

☞ Try "aggressive" search strategies (*clasp* options like `--opt-strategy`, `--opt-heuristic`, and/or `--restart-on-model`)

☞ Abstract from particular candidate solutions
- General lower bounds
- Symmetry breaking
- Encoding methods

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

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

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

☞ Add compact "redundant" constraints for shortcuts to (non-)solutions

☞ Try "aggressive" search strategies (*clasp* options like `--opt-strategy`, `--opt-heuristic`, and/or `--restart-on-model`)

☞ Abstract from particular candidate solutions

- General lower bounds
- Symmetry breaking
- Encoding methods

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

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

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

☞ Add compact “redundant” constraints for shortcuts to (non-)solutions

☞ Try “aggressive” search strategies (*clasp* options like `--opt-strategy`, `--opt-heuristic`, and/or `--restart-on-model`)

☞ Abstract from particular candidate solutions

- General lower bounds
- Symmetry breaking
- Encoding methods

# Hard Regions

| Problem Criticality | **Satisfiability** | **Descent** | **Unsatisfiability** |
|---|---|---|---|
| hard-constrained | difficult | difficult | comparably easy ✔ |
| under-constrained | trivial ✔ | plenty solutions | very difficult |

☞ Add compact "redundant" constraints for shortcuts to (non-)solutions

☞ Try "aggressive" search strategies (*clasp* options like `--opt-strategy`, `--opt-heuristic`, and/or `--restart-on-model`)

☞ Abstract from particular candidate solutions
- General lower bounds
- Symmetry breaking
- Encoding methods

# Overview

# 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

```
#const n = 24.
n/2 { select(1..n) }.
#minimize{ 1,I : select(I) }.
```

## 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
```

# 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

```
#const n = 24.
n/2 { select(1..n) }.
#minimize{ 1,I : select(I) }.
```

## 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
```

# 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

```
#const n = 24.
n/2 { select(1..n) }.
#minimize{ 1,I : select(I) }.
```

## 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
```

# 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

```
#const n = 24.
n/2 { select(1..n) }.
#minimize{ 1,I : select(I) }.
```

## 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

```
#const n = 24.
n/2 { select(1..n) }.
count(I,1)     :- select(I).
count(I+1,N)   :- count(I,N), I < n.
count(I+1,N+1) :- count(I,N), select(I+1).
:- not count(n,n/2).
#minimize{ 1,I : select(I) }.
```

## Performance? ✔

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

# Counting Revisited

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

## Counter Encoding

```
#const n = 24.
n/2 { select(1..n) }.
count(I,1)     :- select(I).
count(I+1,N)   :- count(I,N), I < n.
count(I+1,N+1) :- count(I,N), select(I+1).
:- not count(n,n/2).
#minimize{ 1,N : count(n,N) }.
```

## Performance? ✔

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

# Counting Revisited

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

## Counter Encoding

```
#const n = 24.
n/2 { select(1..n) }.
count(I,1)     :- select(I).
count(I+1,N)   :- count(I,N), I < n.
count(I+1,N+1) :- count(I,N), select(I+1).
:- not count(n,n/2).
#minimize{ 1,N : count(n,N) }.
```

## Performance? ✔

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

# Counting Revisited

When we have to select at least $n/2$ out of $n$ items, how many do we need?

Let's count them!

## Counter Encoding

```
#const n = 24.
n/2 { select(1..n) }.
count(I,1)     :- select(I).
count(I+1,N)   :- count(I,N), I < n.
count(I+1,N+1) :- count(I,N), select(I+1).
:- not count(n,n/2).
#minimize{ 1,N : count(n,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

### Global Selection + Global Counting

```
#const n = 42.
n/3 { select(1..n) }.
#minimize{ 1,I : select(I) }.
```

# Localizing Penalties

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

## Local Selection + Global Counting

```
#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{ 1,I : select(I) }.
```

## Performance?

```
Optimization: 14
Time        : 16.640s (Solving: 16.64s 1st Model: 0.00s Unsat: 16.64s)
Choices     : 4324372
Conflicts   : 3913465
```

# Localizing Penalties

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

## Local Selection + Global Counting

```
#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{ 1,I : select(I) }.
```

## Performance?

```
Optimization: 14
Time        : 16.640s (Solving: 16.64s 1st Model: 0.00s Unsat: 16.64s)
Choices     : 4324372
Conflicts   : 3913465
```

# Localizing Penalties

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

## Local Selection + Global Counting

```
#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{ 1,I : select(I) }.
```

## Performance? ✘

```
Optimization: 14
Time        : 16.640s (Solving: 16.64s 1st Model: 0.00s Unsat: 16.64s)
Choices     : 4324372
Conflicts   : 3913465
```

# Localizing Penalties

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

## Local Selection + Global Counting

```
#const n = 42.
1 { select(3*I-D) : D = 0..2 } :- I = 1..n/3.
count(I,1)     :- select(I).
count(I+1,N)   :- count(I,N), I < n.
count(I+1,N+1) :- count(I,N), select(I+1).
#minimize{ 1,N : count(n,N) }.
```

## Performance? ✘

```
Optimization: 14
Time        : 16.640s (Solving: 16.64s 1st Model: 0.00s Unsat: 16.64s)
Choices     : 4324372
Conflicts   : 3913465
```

# Localizing Penalties

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

## Local Selection + Global Counting

```
#const n = 42.
1 { select(3*I-D) : D = 0..2 } :- I = 1..n/3.
count(I,1)     :- select(I).
count(I+1,N)   :- count(I,N), I < n.
count(I+1,N+1) :- count(I,N), select(I+1).
#minimize{ 1,N : count(n,N) }.
```

## Performance?

```
Time        : 0.030s (Solving: 0.02s 1st Model: 0.00s Unsat: 0.01s)
Conflicts   : 14
Variables   : 1778
Constraints : 5180
```

# Localizing Penalties

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

## Local Selection + Local Counting

```
#const n = 42.
1 { select(3*I-D) : D = 0..2 } :- I = 1..n/3.
count(3*I-D,1)   :- I = 1..n/3, D = 0..2, select(3*I-D).
count(3*I-D,N)   :- I = 1..n/3, D = 0..1, count(3*I-(D+1),N).
count(3*I-D,N+1) :- I = 1..n/3, D = 0..1, count(3*I-(D+1),N), select(3*I-D).
#minimize{ 1,I,N : count(3*I,N), I = 1..n/3 }.
```

## Performance?

```
Time        : 0.030s (Solving: 0.02s 1st Model: 0.00s Unsat: 0.01s)
Conflicts   : 14
Variables   : 1778
Constraints : 5180
```

# Localizing Penalties

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

## Local Selection + Local Counting

```
#const n = 42.
1 { select(3*I-D) : D = 0..2 } :- I = 1..n/3.
count(3*I-D,1)   :- I = 1..n/3, D = 0..2, select(3*I-D).
count(3*I-D,N)   :- I = 1..n/3, D = 0..1, count(3*I-(D+1),N).
count(3*I-D,N+1) :- I = 1..n/3, D = 0..1, count(3*I-(D+1),N), select(3*I-D).
#minimize{ 1,I,N : count(3*I,N), I = 1..n/3 }.
```

## Performance?

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

# Localizing Penalties

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

## Local Selection + Local Counting

```
#const n = 42.
1 { select(3*I-D) : D = 0..2 } :- I = 1..n/3.
count(3*I-D,1)   :- I = 1..n/3, D = 0..2, select(3*I-D).
count(3*I-D,N)   :- I = 1..n/3, D = 0..1, count(3*I-(D+1),N).
count(3*I-D,N+1) :- I = 1..n/3, D = 0..1, count(3*I-(D+1),N), select(3*I-D).
#minimize{ 1,I,N : count(3*I,N), 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

# Example: Traveling Sales-Person (TSP)

## Task

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

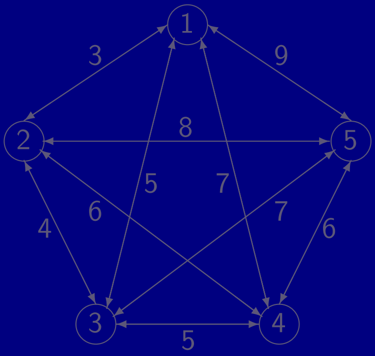
## (Complete) Graph Gadget: graph.lp

```
#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).

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



# Example: Traveling Sales-Person (TSP)

## Task

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

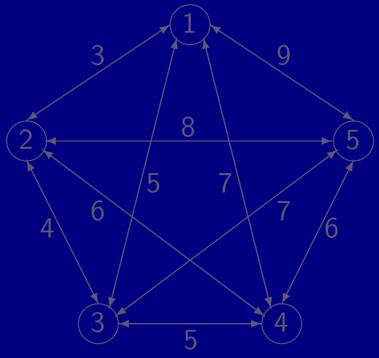
## (Complete) Graph Gadget: graph.lp

```
#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).

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



# Example: Traveling Sales-Person (TSP)

## Task

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

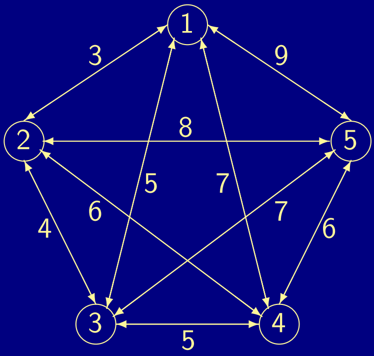
## (Complete) Graph Gadget: graph.lp

```
#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).

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



# 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{ Y : node(Y) }.
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

# 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{ Y : node(Y) }.
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

# 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{ Y : node(Y) }.
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

# 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{ Y : node(Y) }.
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

# 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

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

```
gringo -c n=5 graph.lp tsp0.lp | clasp --stats
```

```
Models       : 1
Optimization: 27
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
Choices      : 41
Conflicts    : 39
```

# 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

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

```
gringo -c n=5 graph.lp tsp0.lp | clasp --stats
```

```
Models       : 1
Optimization: 27
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
Choices      : 41
Conflicts    : 39
```

# 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

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

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

```
Models       : 1
Optimization: 27
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
Choices      : 41
Conflicts    : 39
```

# 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

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), 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)
Choices      : 9168349
Conflicts    : 7053956
```

# 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

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), 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)
Choices      : 9168349
Conflicts    : 7053956
```

# 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

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), 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)
Choices      : 9168349
Conflicts    : 7053956
```

**Ideas for improvements?**

# Mind the Gap(s)!

- Every node requires some outgoing (and incoming) edge

## 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{ Y : node(Y) }.
reached(Y) :- reached(X), cycle(X,Y).
:- node(Y), not reached(Y).

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

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

next(X,C1,C2) :- cost(X,C1), cost(X,C2), C1 < C2, C <= C1 : cost(X,C), C < C2.

gap(X,C1,C2-C1) :- next(X,C1,C2), cycle(X,Y), cost(X,Y,C2).
gap(X,C1,C2-C1) :- next(X,C1,C2), gap(X,C2,G).

#minimize{ G,X,C : gap(X,C,G) }.
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

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

next(X,C1,C2) :- cost(X,C1), cost(X,C2), C1 < C2, C <= C1 : cost(X,C), C < C2.

gap(X,C1,C2-C1) :- next(X,C1,C2), cycle(X,Y), cost(X,Y,C2).
gap(X,C1,C2-C1) :- next(X,C1,C2), gap(X,C2,G).

#minimize{ G,X,C : gap(X,C,G) }.
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

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

next(X,C1,C2) :- cost(X,C1), cost(X,C2), C1 < C2, C <= C1 : cost(X,C), C < C2.

gap(X,C1,C2-C1) :- next(X,C1,C2), cycle(X,Y), cost(X,Y,C2).
gap(X,C1,C2-C1) :- next(X,C1,C2), gap(X,C2,G).

#minimize{ G,X,C : gap(X,C,G) }.
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

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

next(X,C1,C2) :- cost(X,C1), cost(X,C2), C1 < C2, C <= C1 : cost(X,C), C < C2.

gap(X,C1,C2-C1) :- next(X,C1,C2), cycle(X,Y), cost(X,Y,C2).
gap(X,C1,C2-C1) :- next(X,C1,C2), gap(X,C2,G).

#minimize{ G,X,C : gap(X,C,G) }.
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ G,X,C : gap(X,C,G) }.
```

```
gringo -c n=12 graph.lp tsp1.lp | clasp --stats
```

```
Models       : 1
Optimization: 111
Time         : 60.017s (Solving: 60.01s 1st Model: 0.00s Unsat: 57.61s)
Conflicts    : 7053956
```

## 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

% DEFINE + TEST: Each node must be reached from starting node

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ G,X,C : gap(X,C,G) }.
```

```
gringo -c n=12 graph.lp tsp1.lp | clasp --stats
```

```
Models       : 1
Optimization: 20
Time         : 0.206s (Solving: 0.20s 1st Model: 0.00s Unsat: 0.20s)
Conflicts    : 20201
```

## 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

% DEFINE + TEST: Each node must be reached from starting node

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ G,X,C : gap(X,C,G) }.
```

### gringo -c n=12 graph.lp tsp1.lp | clasp --stats

```
Models       : 1
Optimization: 20
Time         : 0.206s (Solving: 0.20s 1st Model: 0.00s Unsat: 0.20s)
Conflicts    : 20201
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ G,X,C : gap(X,C,G) }.
```

```
gringo -c n=16 graph.lp tsp1.lp | clasp --stats
```

```
Models       : 1
Optimization: 20
Time         : 0.206s (Solving: 0.20s 1st Model: 0.00s Unsat: 0.20s)
Conflicts    : 20201
```

# 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

% DEFINE + TEST: Each node must be reached from starting node

% OPTIMIZE: Minimize accumulated edge costs of round trip
#minimize{ G,X,C : gap(X,C,G) }.
```

```
gringo -c n=16 graph.lp tsp1.lp | clasp --stats
```

```
Models       : 1
Optimization: 28
Time         : 31.926s (Solving: 31.91s 1st Model: 0.00s Unsat: 31.91s)
Conflicts    : 1991585
```

# Overview

# 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

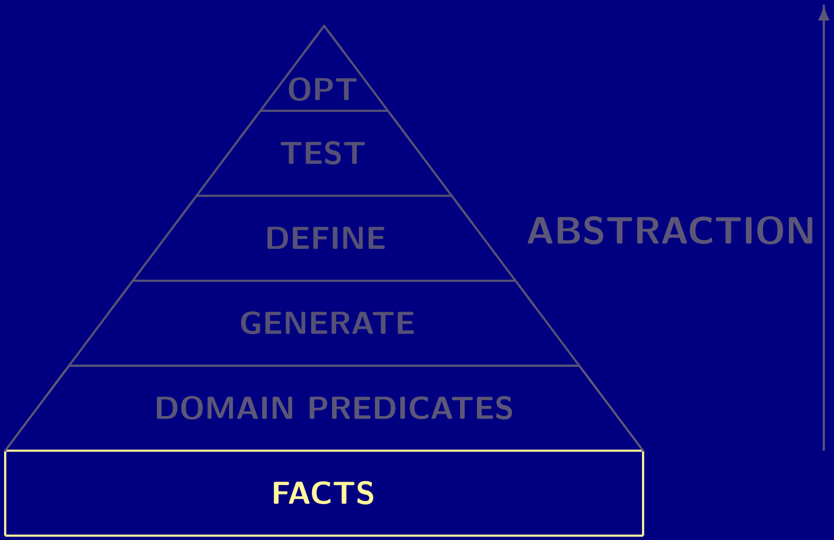
# Optimization Phenomena

- 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

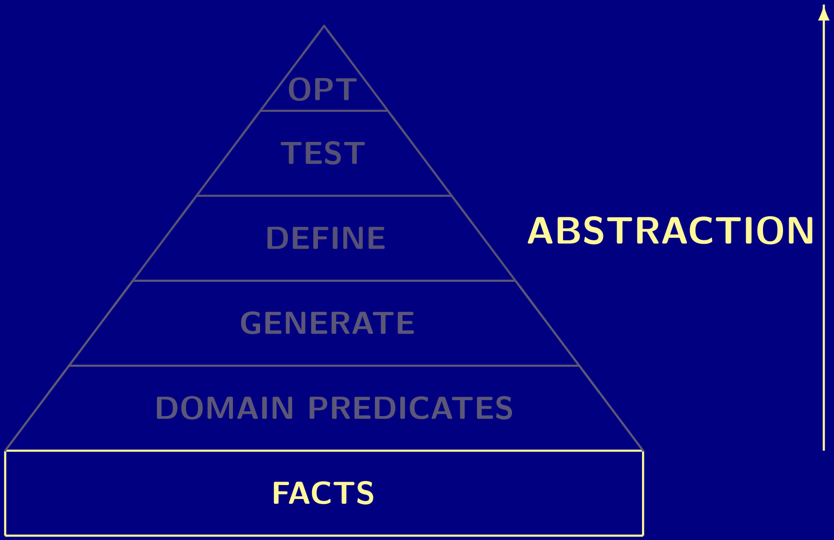
# Optimization Phenomena

- 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

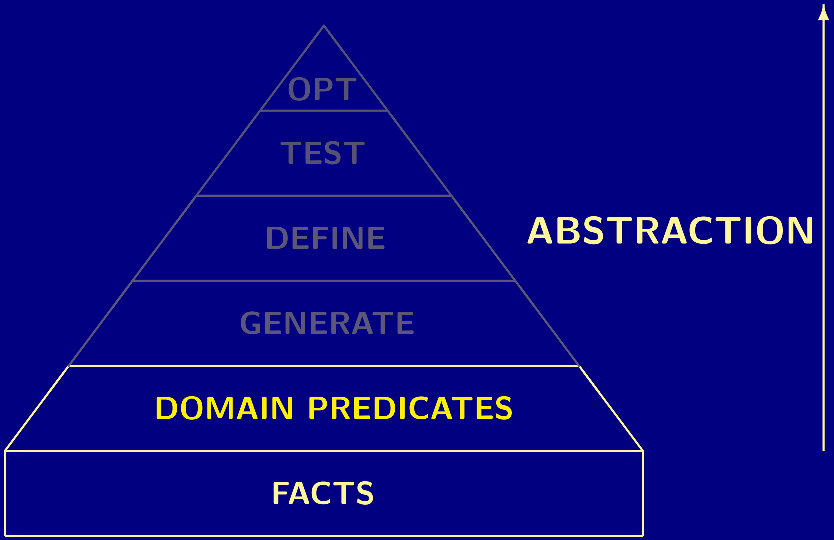
# Walk Like an Egyptian



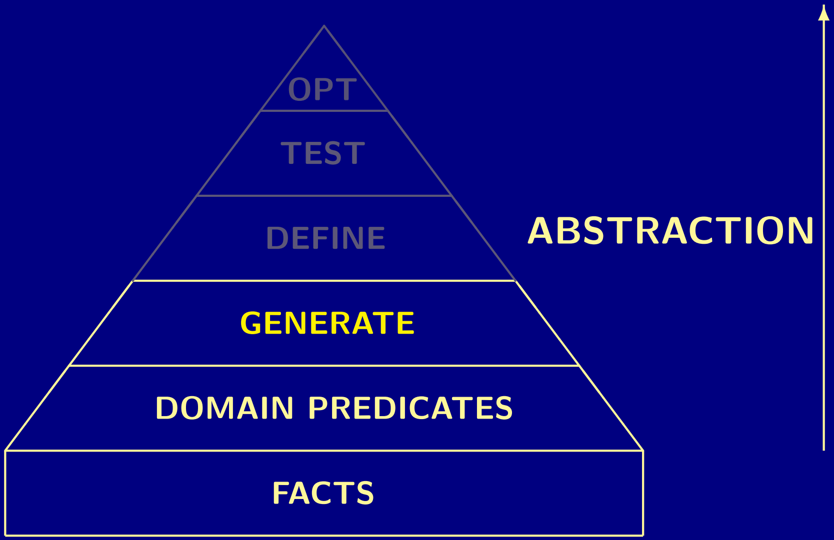
## Walk Like an Egyptian



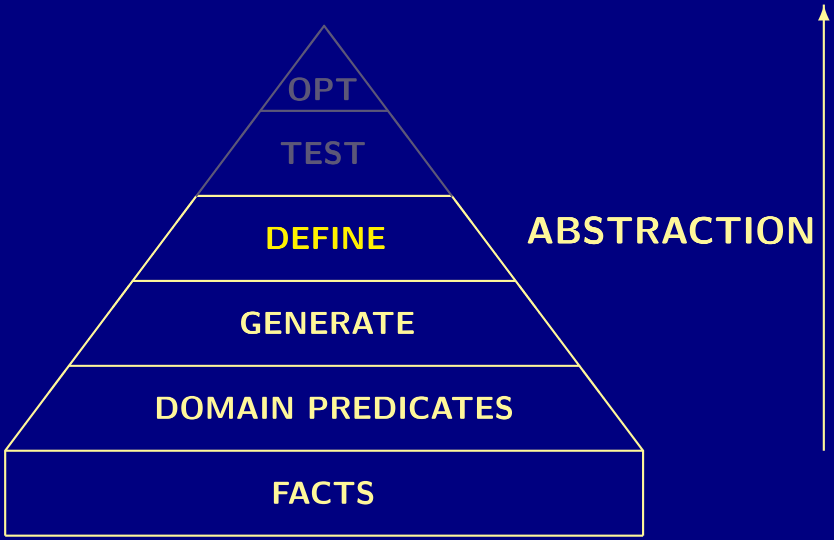
# Walk Like an Egyptian



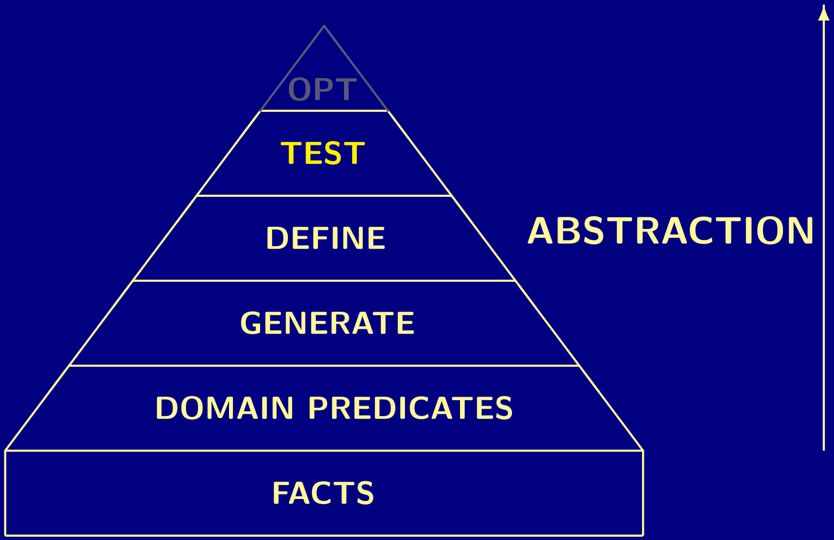
## Walk Like an Egyptian



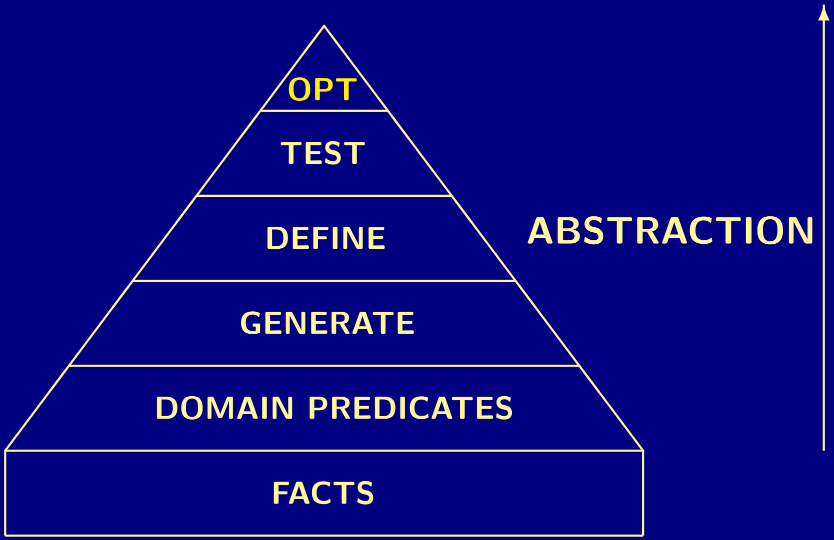
# Walk Like an Egyptian



## Walk Like an Egyptian



## Walk Like an Egyptian



## Walk Like an Egyptian