



Topic 7: Solving Technologies

(Version of 20th August 2024)

Jean-Noël Monette and [Pierre Flener](#)

[Optimisation Group](#)

Department of Information Technology
Uppsala University
Sweden

Course 1DL442:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation



Outline

- 1. The MiniZinc Toolchain**
- 2. Comparison Criteria**
- 3. SAT**
- 4. SMT & OMT**
- 5. IP & MIP**
- 6. CP**
- 7. LS & CBLS**
- 8. Hybrid Technologies**
- 9. Case Study**
- 10. Choosing a Technology and Backend**

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

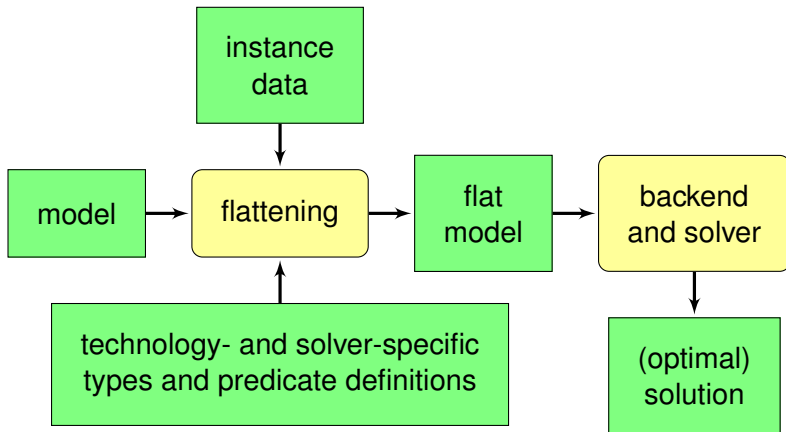
Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



MiniZinc: Model Once, Solve Everywhere!



From a **single** language, one has access transparently to a wide range of solving technologies from which to choose.



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend



Objectives

An overview of some solving technologies:

- to understand their advantages and limitations;
- to help you choose a technology for a particular model;
- to help you adapt a model to a particular technology.



Examples (Solving technologies)

With general-purpose solvers, taking model and data as input:

- Boolean satisfiability (SAT)
- SAT (resp. optimisation) modulo theories (SMT and OMT)
- (Mixed) integer linear programming (IP and MIP)
- Constraint programming (CP)
- ...
- Hybrid technologies (LCG = CP + SAT, ...)

👉 part 2 of 1DL442

Methodologies, *usually without* modelling and solvers:

- Dynamic programming (DP)
- Greedy algorithms
- Approximation algorithms
- Local search (LS)
- ...



How to Compare Solving Technologies?

Modelling Language:

- What types of decision variables are available?
- Which constraint predicates are available?
- Can there be an objective function?

Guarantees:

- Are its solvers **exact**, given enough time:
will they find all solutions, prove optimality, and prove unsatisfiability?
- If not, is there an **approximation** ratio for the solution quality?

Features:

- Can the modeller guide the solving? If yes, then how?
- In which application areas has the technology been successfully used?
- How do solvers work?



How Do Solvers Work? (Hooker, 2012)

Definition (Solving = Search + Inference + Relaxation)

- **Search**: Explore the space of candidate solutions.
- **Inference**: Reduce the space of candidate solutions.
- **Relaxation**: Exploit solutions to easier problems.

Definition (Systematic Search)

Progressively build a solution, and backtrack if necessary.

Use **inference** and **relaxation** to reduce the search effort.

It is used in most SAT, SMT, OMT, CP, LCG, and MIP solvers.

Definition (Local Search)

Start from a candidate solution and iteratively modify it a bit.

It is the basic idea behind LS and genetic algorithms (GA) technologies.



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend



Boolean Satisfiability Solving (SAT)

Modelling Language:

- Only Boolean decision variables.
- A conjunction (\wedge) of clauses. A **clause** is a disjunction (\vee) of literals. A **literal** is a Boolean decision variable or its negation (**not**).
- Only for satisfaction problems; else: iterate over candidate obj. values.

Example (in MiniZinc syntax)

- Decision variables: `var bool: w, x, y, z;`

- Clauses:

```
constraint (not w \\/ not y) /\ (not x \\/ y)
           /\ (not w \\/ x \\/ not z)
           /\ (x \\/ y \\/ z) /\ (w \\/ not z);
```

- A solution: `w=false, x=true, y=true, z=false`



The SAT Problem

Given a clause set, find a **valuation**, that is Boolean values for all the decision variables, so that all the clauses are satisfied.

- The decision version of this problem is NP-complete.
- **Any combinatorial problem can be encoded into SAT.**
Careful: “encoded into” is not “reduced from”, but “reduced to”.
There are recipes for clausifying non-Boolean constraints.
- There has been intensive research since the 1960s.
- We focus here on systematic search, namely DPLL
[Davis-Putnam-Logemann-Loveland, 1962].



Tree Search, upon starting from the empty valuation:

- 1 Perform **inference** (see below).
- 2 If some clause is unsatisfied, then backtrack.
- 3 If all decision variables have a value, then we have a solution.
- 4 Select an unvalued decision variable b and make two branches: one with $b = \text{true}$, and the other one with $b = \text{false}$.
- 5 Recursively explore each of the two branches.

Inference:

- **Unit propagation:** If all the literals in a clause evaluate to **false**, except one whose decision variable has no value yet, then that literal is made to evaluate to **true** so that the clause becomes satisfied.



Strategies and Improvements over DPLL

Search Strategies:

- On which decision variable to branch next?
- Which branch to explore next?
- Which search (depth-first, breadth-first, ...) to use?

Improvements:

- Backjumping
- Clause learning
- Restarts
- A lot of implementation details
- ...



SAT Solving

- Guarantee: exact, given enough time.
- Mainly black-box: there are limited ways to guide the solving.
- It can scale to millions of decision variables and clauses.
- Encoding a problem can yield a huge SAT model.
- For model debugging purposes, solvers can extract an **unsatisfiable core**, that is a subset of the clauses that make the model unsatisfiable.
- It is mainly applied in hardware verification and software verification.



SAT @ MiniZinc and Uppsala University

- The MiniZinc toolchain was extended with the [PicatSAT](#) backend, which uses the SAT solver [Plingeling](#).
- Several research groups at Uppsala University *use* SAT solvers, such as:
 - [Algorithmic Program Verification](#)
 - [Embedded Systems](#)
 - [Programming Languages](#)
 - [Theory for Concurrent Systems](#)
- My [Algorithms & Datastructures 3 \(1DL481\)](#) course explains SAT solving and has a homework where a model is generated and fed to a SAT solver.



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend



SAT Modulo Theories (SMT) and OMT

Modelling Language:

- Language of SAT: Boolean decision variables and clauses.
- Several theories extend the language, such as bit vectors, uninterpreted functions, or linear integer arithmetic.
- SMT is only for satisfaction problems.
- OMT (optimisation modulo theories) extends SMT.

Definition

A theory

- defines types for decision variables and defines constraint predicates;
- is associated with a sub-solver for any conjunction of its predicates.

Different SMT or OMT solvers may have different theories.



Example (Linear integer arithmetic; in MiniZinc syntax)

- Decision variables: `var int: x; var int: y;`

- Constraints:

```
constraint x >= 0; constraint y <= 0;  
constraint x = y + 1 \/ x = 2 * y;  
constraint x = 2 \/ y = -2 \/ x = y;
```

- Unique solution: $x = 0, y = 0$

- Decomposition:

- Theory constraints, using reified constraints:

```
a <-> x >= 0;      b <-> y <= 0;  
c <-> x = y + 1;   d <-> x = 2 * y;  
e <-> x = 2;       f <-> y = -2;      g <-> x = y;
```

- Boolean skeleton:

```
a /\ b /\ (c \/ d) /\ (e \/ f \/ g);
```



SMT Solving: DPLL(T)

Basic Idea:

- Separate the theory constraints and Boolean skeleton: each decision variable in the Boolean skeleton denotes whether a constraint holds or not.
- Use DPLL to solve the Boolean skeleton.
- If a constraint must hold as per DPLL, then submit it to the relevant theory solver.
- A theory solver operates on a constraint conjunction:
 - It checks whether the conjunction is satisfiable.
 - It tries to **infer** that other constraints must (respectively cannot) hold and it sets the corresponding Boolean variables to **true** (respectively **false**).



Strategies and Improvements

Search Strategies:

- On which decision variable to branch next?
- Which branch to explore next?
- Which strategy (depth-first, breadth-first, ...) to use?

Improvements to SAT Solving:

- See slide 14.

Improvements to the Theory Solvers:

- More efficient **inference** algorithms: incrementality.
- Richer theories.
- ...



SMT and OMT Solving

- Guarantee: exact, given enough time.
- Mainly black-box: there are limited ways to guide the solving.
- They are based on the very efficient SAT technology.
- They are mainly applied in hardware verification and software verification.



SMT and OMT @ MiniZinc and Uppsala University

- The MiniZinc toolchain was extended with:
 - [fzn2smt](#): generates [SMTlib](#) models that can be fed to any SMT solver, such as [CVC4](#), [Yices 2](#), [Z3](#), ...
 - [emzn2fzn + fzn2omt](#): generates models that can be fed to any OMT solver, such as [OptiMathSAT](#), [Z3](#), ...

- Several research groups at Uppsala University *use* SMT solvers, such as:
 - [Algorithmic Program Verification](#)
 - [Programming Languages](#)
 - [Theory for Concurrent Systems](#)

- My [Algorithms & Datastructures 3 \(1DL481\)](#) course explains SMT solving and has a homework where a model is written and fed to an SMT solver.



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



Integer (Linear) Programming (IP = ILP)

Modelling Language:

- Only integer decision variables.
- A set of linear equality and inequality constraints (note: no disequality \neq).
- Only for optimisation problems: linear objective function (else: a value).

Example (in MiniZinc syntax)

- Decision variables: `var int: p; var int: q;`
- Constraints:
`constraint p >= 0; constraint q >= 0;`
`constraint p + 2 * q <= 5;`
`constraint 3 * p + 2 * q <= 9;`
- Objective: `solve maximize 3 * p + 4 * q;`
- Unique (in this case) optimal solution: $p = 1, q = 2$



Mathematical Programming

- **0-1 linear programming:**
linear (in)equalities over decision variables over the domain $\{0, 1\}$.
- **Linear programming (LP):**
linear (in)equalities over floating-point decision variables.
- **Mixed integer (linear) programming (MIP):**
linear (in)equalities over floating-point and integer decision variables.
- **Quadratic programming (QP):**
quadratic objective function.
- ...

There has been intensive research since the 1940s.



IP Solving

Basic Idea = Relaxation:

- Polytime algorithms (such as the interior-point method and the ellipsoid method) and exponential-time but practical algorithms (such as the simplex method) exist for solving LP models very efficiently.
- Use them for IP by occasionally **relaxing** an IP model via dropping its integrality requirement on the decision variables.

Implementations:

- Branch and bound = **relaxation** + **search**.
- Cutting-plane algorithms = **relaxation** + **inference**.
- Branch and cut = **relaxation** + **search** + **inference**.



Branch and Bound

Tree Search, upon initialising the incumbent to $\pm\infty$:

- 1 **Relax** the IP model into an LP model, and solve it.
- 2 If the LP model is unsatisfiable, then backtrack.
- 3 If all the decision variables have an integer value in the optimal LP solution, then backtrack upon updating, if need be, the incumbent to the objective value of that IP solution.
- 4 If the objective value of the optimal LP solution is no better than the incumbent, then backtrack.
- 5 Otherwise, some decision variable v has a non-integer value ρ .
Make two branches: one with $v \leq \lfloor \rho \rfloor$, and the other one with $v \geq \lceil \rho \rceil$.
- 6 Recursively explore each of the two branches.



Strategies and Improvements

Search Strategies:

- On which decision variable to branch next?
- Which branch to explore next?
- Which search (depth-first, breadth-first, ...) to use?

Improvements:

- **Cutting planes:** Add **implied linear constraints** that improve the objective value of the LP **relaxation**.
- **Decomposition:** Split into a master problem and a subproblem, such as by the Benders decomposition.
- **Solving the LP relaxation:**
 - Primal-dual methods.
 - Efficient algorithms for special cases, such as flows.
 - Incremental solving.
- ...



IP Solving

- Guarantee: exact, given enough time.
- Mainly black-box: limited ways to guide the solving.
- It scales well.
- Any combinatorial problem can be encoded into IP.
There are recipes for linearising non-linear constraints.
- Advantages:
 - Provides both a lower bound and an upper bound on the objective value of optimal solutions, if stopped early.
 - Naturally extends to MIP solving.
 - ...
- Central method of operations research (OR),
applied in production planning, vehicle routing, ...



MIP @ MiniZinc and Uppsala University

- The MiniZinc toolchain comes bundled with a backend that can be hooked to the following MIP solvers:
 - [Cbc](#) (open-source, bundled);
 - [CPLEX Optimizer](#) (commercial: requires a license);
 - [FICO Xpress Solver](#) (commercial: requires a license);
 - [Gurobi Optimizer](#) (commercial: requires a license);
 - [HiGHS](#) (open-source, bundled).

- The [Optimisation](#) research group at Uppsala University *uses* MIP solvers for 4G / 5G network planning and optimisation, etc.

- My [Algorithms & Datastructures 3 \(1DL481\)](#) course explains MIP solving and has a homework where a model is designed and fed to a MIP solver.



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



Constraint Programming (CP)

Modelling Language = full MiniZinc:

- Any combination of Boolean, integer, enumeration, set decision variables.
- Constraints based on a large vocabulary of predicates.
- For satisfaction problems and optimisation problems.

Many solvers:

- There will be no standard for what is to be supported: different CP solvers may have different sets of types for decision variables and different constraint predicates (under different names).
- Some solvers support even higher-level types for decision variables, such as graphs and strings, and associated predicates.



Domains

Definition

The **domain** of a decision variable v , denoted here by $\text{dom}(v)$, is the set of values that v can still take during **search**:

- The domains of the decision variables are reduced by **search** and by **inference** (see the next two slides).
- A decision variable is said to be **fixed** if its domain is a singleton.
- **Unsatisfiability** occurs if the domain of a decision variable goes empty.

Note the difference between:

- a domain as a technology-independent declarative entity when modelling;
- a domain as a CP-technology procedural data structure when solving.



CP Solving

Tree Search, upon initialising each domain as in the model:

Satisfaction problem:

- 1 Perform inference (see the next slide).
- 2 If the domain of some decision variable is empty, then backtrack.
- 3 If all decision variables are fixed, then we have a solution.
- 4 Select a non-fixed decision variable v , partition its domain into two parts π_1 and π_2 , and make two branches: one with $v \in \pi_1$, and the other one with $v \in \pi_2$.
- 5 Recursively explore each of the two branches.

Optimisation problem: when a feasible solution is found at step 3, first add the constraint that the next solution must be better and then backtrack.



CP Inference

Definition

A **propagator** for a predicate γ deletes from the domains of the variables of a γ -constraint the values that cannot be in a solution to that constraint. The propagator of a constraint is **active** as long as the Cartesian product of the domains of its variables is not known to contain only solutions to the constraint.

Examples

- For $x < y$: when $\text{dom}(x) = 1..4$ and $\text{dom}(y) = -1..3$, delete $3..4$ from $\text{dom}(x)$ and $-1..1$ from $\text{dom}(y)$. The propagator remains active.
- For `all_different` ($[x, y, z]$): when $\text{dom}(x) = \{1, 3\} = \text{dom}(y)$ and $\text{dom}(z) = 1..4$, delete 1 and 3 from $\text{dom}(z)$ so that it becomes the non-range $\{2, 4\}$. The propagator becomes inactive after $\text{dom}(x)$ loses 1.



Strategies and Improvements

Search Strategies:

- On which decision variable to branch next?
- How to partition the domain of the chosen decision variable?
- Which search (depth-first, breadth-first, ...) to use?

Improvements:

- **Propagators**, including for the predicates in Topic 3: Constraint Predicates. Not all impossible domain values need to be deleted: there is a compromise between algorithm complexity and achieved **inference**.
- **Partition** the chosen domain into at least two parts.
- Domain representations.
- **Order** in which propagators are executed.
- ...



CP Solving

- Guarantee: exact, given enough time.
- White-box: one can design one's own **propagators** and **search strategies**, and choose among predefined ones.
- The higher-level modelling languages enable (for details, see Topic 8: Inference & Search in CP & LCG):
 - **inference** at a higher level;
 - **search strategies** stated in terms of problem concepts.

They inspired the MiniZinc modelling language.

- Successful application areas:
 - Configuration
 - Personnel rostering
 - Scheduling and timetabling
 - Vehicle routing
 - ...



CP @ MiniZinc and Uppsala University

- The MiniZinc toolchain was extended with backends for many CP solvers, such as [Gecode](#) (bundled), [Choco](#), [JaCoP](#), [Mistral](#), [SICStus Prolog](#), . . .
- The [Optimisation](#) research group at Uppsala University contributes to the *design* of CP solvers and *uses* them, say for air traffic management, the configuration of wireless sensor networks, robot task sequencing, etc.
- Part 2 of my [Combinatorial Optimisation and Constraint Programming \(1DL442\)](#) course covers CP in depth.



Outline

1. The MiniZinc Toolchain
2. Comparison Criteria
3. SAT
4. SMT & OMT
5. IP & MIP
6. CP
- 7. LS & CBLS**
8. Hybrid Technologies
9. Case Study
10. Choosing a Technology and Backend

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

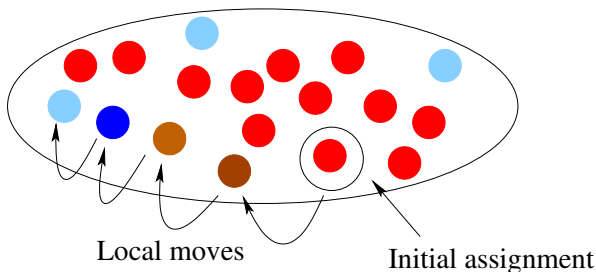
Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend

Local Search (LS)

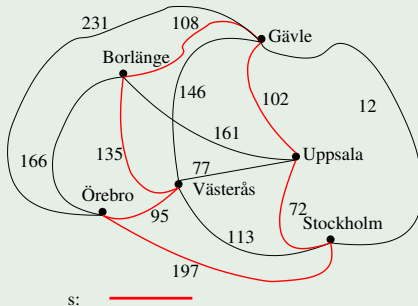
- Each decision variable is fixed **all the time**.
- Search proceeds by moves: each **move** modifies the values of a few decision variables in the **current assignment**, and is **selected** upon **probing** the **cost** impacts of several candidate moves, called the **neighbourhood**.
- Stop when either a good enough assignment was found, or when an allocated resource was exhausted, such as time spent or iterations made.





Example (Travelling Salesperson, TSP)

- **Problem:** Given a set of cities with connecting roads, find a Hamiltonian circuit (**tour**) visiting each city exactly once, with minimal travel distance.
- **Representation:** We see the cities as vertices V and the roads as edges E of a (not necessarily complete) undirected graph $G = (V, E)$.
- **Example:**



We now design a local-search heuristic for TSP, without a modelling language.



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

- 1 The **initial assignment**:
- 2 The **neighbourhood** of candidate moves:
- 3 The **cost** of a current assignment s (not necessarily its objective value):
- 4 The **neighbour selector**:



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

- 1 The **initial assignment**:
An edge set $s \subseteq E$ that forms a tour
- 2 The **neighbourhood** of candidate moves:
- 3 The **cost** of a current assignment s (not necessarily its objective value):
- 4 The **neighbour selector**:



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

- 1 The **initial assignment**:
An edge set $s \subseteq E$ that forms a tour: NP-hard!
- 2 The **neighbourhood** of candidate moves:
- 3 The **cost** of a current assignment s (not necessarily its objective value):
- 4 The **neighbour selector**:



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

1 The **initial assignment**:

An edge set $s \subseteq E$ that forms a tour: NP-hard!

Complete E by adding infinite-distance edges:

any permutation of V yields a tour as an initial assignment.

2 The **neighbourhood** of candidate moves:

3 The **cost** of a current assignment s (not necessarily its objective value):

4 The **neighbour selector**:



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

1 The **initial assignment**:

An edge set $s \subseteq E$ that forms a tour: NP-hard!

Complete E by adding infinite-distance edges:

any permutation of V yields a tour as an initial assignment.

2 The **neighbourhood** of candidate moves:

Replace two edges on the tour s by two other edges so that s is still a tour.

3 The **cost** of a current assignment s (not necessarily its objective value):

4 The **neighbour selector**:



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

1 The **initial assignment**:

An edge set $s \subseteq E$ that forms a tour: NP-hard!

Complete E by adding infinite-distance edges:

any permutation of V yields a tour as an initial assignment.

2 The **neighbourhood** of candidate moves:

Replace two edges on the tour s by two other edges so that s is still a tour.

3 The **cost** of a current assignment s (not necessarily its objective value):

The sum of all distances on the tour:

$$f(s) = \sum_{(a,b) \in s} \text{Distance}(a, b), \text{ as no constraint violation can happen.}$$

4 The **neighbour selector**:



Example (Travelling Salesperson: Algorithmic Choices)

We must define:

1 The **initial assignment**:

An edge set $s \subseteq E$ that forms a tour: NP-hard!

Complete E by adding infinite-distance edges:

any permutation of V yields a tour as an initial assignment.

2 The **neighbourhood** of candidate moves:

Replace two edges on the tour s by two other edges so that s is still a tour.

3 The **cost** of a current assignment s (not necessarily its objective value):

The sum of all distances on the tour:

$$f(s) = \sum_{(a,b) \in s} \text{Distance}(a, b), \text{ as no constraint violation can happen.}$$

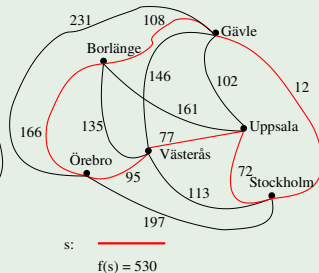
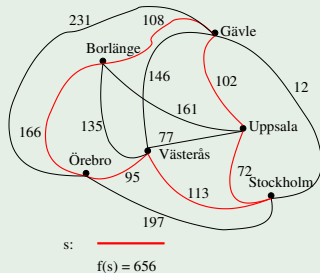
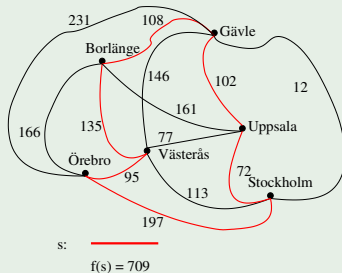
4 The **neighbour selector**:

Select a random best neighbour.



Example (Travelling Salesperson: Sample Run)

Three consecutive improving current assignments:



This section is so far based on material by Magnus Rattfeldt.



Heuristics drive the search to (good enough) solutions:

- Which decision variables are modified in a move?
- Which new values do they get in the move?

Metaheuristics drive the search to global optima of the cost:

- Avoid cycles of moves and escape local optima of the cost.
- Explore many parts of the search space.
- Focus on promising parts of the search space.

Examples (Metaheuristics)

- **Tabu search** (1986):
forbid recent moves from being done again.
- **Simulated annealing** (1983):
perform random moves and accept degrading ones with a probability that decreases over time.
- **Genetic algorithms** (1975):
use a pool of current assignments and cross them.



Systematic Search (as in SAT, SMT, OMT, MIP, and CP):

- + **Will** find an (optimal) solution, if one exists.
- + Will give a proof of unsatisfiability, otherwise.
- May take a **long time** to complete.
- Sometimes does not scale well to large instances.
- May need a lot of tweaking: search strategies, ...

Local Search: (Hoos and Stützle, 2004)

- + **May** find an (optimal) solution, if one exists.
- Can rarely give a proof of unsatisfiability, otherwise.
- Can rarely guarantee that a found solution is optimal.
- + Often scales well to large instances.
- May need a lot of tweaking: (meta)heuristics, ...

Local search trades solution quality for speed!



Constraint-Based Local Search (CBLS)

- MiniZinc-style modelling language:
 - Any combination of Boolean, integer, enumeration set decision variables.
 - Constraints based on a large vocabulary of predicates.
 - Three sorts of constraints: see the next three slides.
 - For satisfaction problems and optimisation problems.
- Fairly recent: around the year 2000.
- Guarantee: **inexact** on most instances (that is: there is no promise to find all solutions, to prove optimality, or to prove unsatisfiability), without an approximation ratio.
- White-box: one **must** design a search algorithm, which probes the cost impacts for guidance.
- More scalable than systematic technologies.



Definition

Each constraint predicate has a violation function:
the **violation** of a constraint is zero if it is satisfied in the current assignment,
else a positive value proportional to its dissatisfaction.

Example

For $a \leq b$, let α and β be the current values of a and b :
define the violation to be $\alpha - \beta$ if $\alpha \not\leq \beta$, and 0 otherwise.

Definition

A **constraint with violation** is explicit in a CBLS model and **soft**:
it can be violated during search but ought to be satisfied in a solution.

The constraint violations are queried during search.



Definition

A **one-way constraint** is explicit in a CBLS model and **hard**: it is kept satisfied during search by keeping the value of a decision variable equal to a total function on its other decision variables.

Example

For $p = a * b$, whenever either the value α of a , or the value β of b , or both are modified by a move, the value of p is automatically modified by the solver so as to remain equal to $\alpha \cdot \beta$.

CBLS solvers offer a special syntax for one-way constraints, such as $p \leq a * b$ in `OscaR.cbls`, but MiniZinc does not make such a syntactic distinction.



Definition

An **implicit constraint** is not in a CBLS model but hard: it is kept satisfied during search by choosing a feasible initial assignment and only making satisfaction-preserving moves, by the use of a **constraint-specific neighbourhood**.

Example

For `all_different(...)` when the number of decision variables is equal to the number of their domain values: the initial assignment has distinct values for all decision variables, and the neighbourhood only has moves that swap the values of two decision variables.

When building a CBLS model, a MiniZinc backend must:

- Aptly assort the otherwise all explicit and all soft constraints.
- Add a suitable heuristic and meta-heuristic.

This is **much** more involved than just flattening and solving.

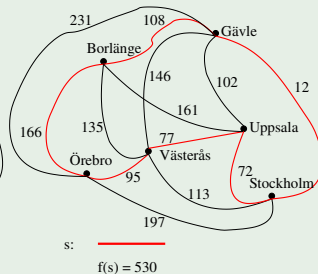
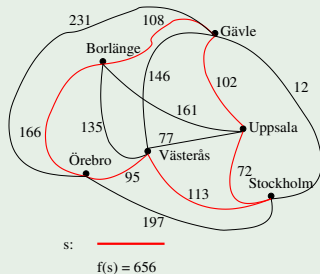
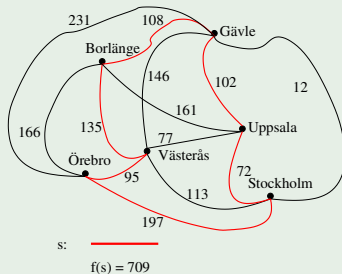


Example (Travelling Salesperson: Model and Solve)

Recall the model, from Topic 1: Introduction,
with a decision variable $\text{Next}[c]$ for each city c :

```
3 solve minimize sum(c in Cities) (Distance[c, Next[c]]);  
4 constraint circuit(Next); % ideally made implicit
```

Three consecutive current assignments, preserving the satisfaction of
the `circuit`(Next) constraint and improving the objective value:





(CB)LS @ MiniZinc and Uppsala University

- The MiniZinc toolchain was extended with:
 - our [fzn-oscar-cbbs](#) backend to the [OscaR.cbbs](#) solver;
 - our Atlantis CBLS backend;
 - the [Yuck](#) CBLS backend.
- The [Optimisation](#) research group at Uppsala University contributes to the *design* of CBLS solvers.
- Several courses at Uppsala University discuss (CB)LS:
 - My [Algorithms & Datastructures 3 \(1DL481\)](#) course explains LS and has a homework where an LS program is written.
 - [Artificial Intelligence \(1DL340\)](#) discusses LS.
 - [Machine Learning \(1DT071\)](#) discusses LS.



Outline

1. The MiniZinc Toolchain

2. Comparison Criteria

3. SAT

4. SMT & OMT

5. IP & MIP

6. CP

7. LS & CBLS

8. Hybrid Technologies

9. Case Study

10. Choosing a Technology and Backend

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



Crossfertilisation

- Each technology has advantages and drawbacks.
- Good ideas from one technology can be applied to another technology.
- A **hybrid technology** combines several technologies.
- This can yield new advantages with fewer drawbacks.
- Some hybrid technologies are **loosely coupled**: separate solvers or sub-solvers cooperate.
- Other hybrid technologies are **tightly coupled**: a single solver handles the whole model.

Example (Loose hybrid technology)

Logic-based Benders decomposition: divide the problem into two parts: a master problem, solved by IP, and a subproblem, solved by CP.



Tight Hybrid Technologies: Examples

Example (Lazy clause generation, LCG)

Use CP propagators to generate clauses in a SAT solver.

Example (Large-neighbourhood search, LNS on a COP)

Find a first solution by CP and then follow an LS procedure, where each move is performed by:

- 1 Restoring the domains for a subset of the decision variables.
- 2 Using a CP solver to find an (optimal) solution to the subproblem.

Example (Constrained integer programming, CIP)

Use CP propagators in an IP solver in order to generate linear inequalities for non-linear constraints.



Hybrids @ MiniZinc and Uppsala University

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend

- The MiniZinc toolchain was extended with:
 - LCG backends: [Chuffed](#) (bundled) and [Google CP-SAT](#);
 - a CIP backend: [SCIP](#);
 - LNS backends: the solvers of the [Gecode](#) (bundled) and [Google CP-SAT](#) backends can perform LNS (prescribed via MiniZinc annotations).

- The [Optimisation](#) research group at Uppsala University contributes to the *design* of hybrid solvers and *uses* them (see slide 39).



Outline

1. The MiniZinc Toolchain
2. Comparison Criteria
3. SAT
4. SMT & OMT
5. IP & MIP
6. CP
7. LS & CBLS
8. Hybrid Technologies
- 9. Case Study**
10. Choosing a Technology and Backend

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



Example: Pigeonhole Problem

Example (Pigeonhole)

Place n pigeons into $n - 1$ pigeonholes so that all pigeons are placed and no two pigeons are placed in the same pigeonhole.

This problem is trivially unsatisfiable, but is a popular benchmark for solvers.

We will use this problem to show:

- how solvers may use different definitions of the same constraint predicate;
- it is often important for efficiency to use pre-defined constraint predicates.



Pigeonhole: Models

Using `all_different`

```
1 int: n; % the number of pigeons
2 % Hole[p] = the hole of pigeon p:
3 array[1..n] of var 1..(n-1): Hole;
4 constraint all_different(Hole);
5 solve satisfy;
```

Using `!=`

```
4 constraint forall(i, j in 1..n where i < j)
    (Hole[i] != Hole[j]);
```



Constraint Predicate Definitions

Built-in `all_different` for probably all CP solvers

```
predicate all_different_int(array[int] of var int: X);  
  
predicate int_ne(var int: x, var int: y);
```

Non-built-in `all_different` for SMT solvers

```
predicate all_different_int(array[int] of var int: X) =  
    forall(i, j in index_set(X) where i < j) (X[i] != X[j]);  
  
predicate int_ne(var int: x, var int: y);
```



Boolean-isation for SAT solvers

```

predicate all_different_int(array[int] of var int: X) =
  let {
    array[int,int] of var bool: Y = int2bools(X);
    array[...,...] of var bool: A;
  } in forall(i in ..., j in ...)
    ((A[i-1,j] -> A[i,j])
     /\ (Y[i,j] <-> (not A[i-1,j] /\ A[i,j])));
function array[int,int] of var bool: int2bools
  (array[int] of var int: X) = [...];

```

When X has n decision variables over domains of size m , this **ladder encoding** yields the **two** arrays Y and A of $n \cdot m$ Boolean decision variables (where $Y[i, v] = \text{true}$ iff $X[i] = v$, and $A[i, v] = \text{true}$ iff $v \in X[1..i]$) as well as $\mathcal{O}(n^2)$ clauses of 2 or 3 literals. This is more compact and usually more efficient than the **direct encoding**, with $\mathcal{O}(n^3)$ clauses of 2 literals over **only** Y .



Linearisation for MIP solvers: Cbc, CPLEX, Gurobi, HiGHS, ...

```
predicate all_different_int(array[int] of var int: X) =  
  let {array[int,int] of var 0..1: Y = eq_encode(X)  
    } in forall(d in index_set_2of2(Y))  
      (sum(i in index_set_1of2(Y)) (Y[i,d]) <= 1);
```

```
predicate int_ne(var int: x, var int: y) =  
  let {var 0..1: p}  
  in x - y + 1 <= ub(x - y + 1) * (1 - p)  
  /\ y - x + 1 <= ub(y - x + 1) * p;
```

```
% ... continued on next slide ...
```



Linearisation for MIP solvers (end)

```
% ... continued from previous slide ...
```

```
function array[int,int] of var int:
```

```
  eq_encode(array[int] of var int: X) =  
    [... equality_encoding(...) ...]
```

```
predicate equality_encoding(var int: x,  
                           array[int] of var 0..1: Y) =  
  
  x in index_set(Y)  
  /\  
  sum(d in index_set(Y)) (Y[d]) = 1  
  /\  
  sum(d in index_set(Y)) (d * Y[d]) = x;
```



Pigeonhole: Experimental Comparison

Time, in seconds, to prove unsatisfiability:

<i>n</i>	backend	all_different	!=
10	mzn-gecode	< 1	< 1
10	mzn-gurobi	< 1	58
11	mzn-gecode	< 1	9
11	mzn-gurobi	< 1	285
12	mzn-gecode	< 1	113
12	mzn-gurobi	< 1	3704
100	mzn-gecode	< 1	time-out
100	mzn-gurobi	< 1	time-out
300	mzn-gecode	< 1	time-out
300	mzn-gurobi	24	time-out
100,000	mzn-gecode	< 1	time-out
1,000,000	mzn-gecode	5	time-out



Outline

1. The MiniZinc Toolchain
2. Comparison Criteria
3. SAT
4. SMT & OMT
5. IP & MIP
6. CP
7. LS & CBLS
8. Hybrid Technologies
9. Case Study
- 10. Choosing a Technology and Backend**

The MiniZinc
Toolchain

Comparison
Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid
Technologies

Case Study

Choosing a
Technology
and Backend



Some Questions for Guidance

- Do you need guarantees that a found solution is optimal, that all solutions are found, and that unsatisfiability is provable?
- What types of decision variables are in your model?
- What constraint predicates are in your model?
- Does your problem look like a well-known problem?
- How do backends perform on easy problem instances?
- What is your favourite technology or backend?



Some Caveats

- Each problem can be modelled in many different ways.
- Different models of the same problem suit better for different backends.
- Performance on small instances does not always scale to larger instances.
- Sometimes, a good **search strategy** is more important than a good model (see Topic 8: Inference & Search in CP & LCG).
- Not all backends of the same technology have comparable performance.
- Some pure problems can be solved by specialist solvers, such as **Concorde** for the travelling salesperson problem, but real-life side constraints often make them inapplicable.
- Some problems are maybe even solvable in polynomial time and space.



Take-Home Message:

- There are many solving technologies and backends.
- It is useful to highlight the commonalities and differences.
- No solving technology or backend can be universally better than all the others, unless $P = NP$.

👉 Try them!

To go further:



John N. Hooker.

[Integrated Methods for Optimization.](#)

2nd edition, Springer, 2012.