Topic 7: Solving Technologies
(Version of 28th September 2022)

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
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
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\&data as input:

- Boolean satisfiability (SAT)
- SAT (resp. optimisation) modulo theories (SMT \& OMT)
- (Mixed) integer linear programming (IP \& MIP)
- Constraint programming (CP)

☞ part 2 of 1DL442

- \ldots

- Hybrid technologies (LCG = CP + SAT, \ldots)  

Methodologies, *usually without* modelling and solvers:

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

- \ldots
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 solvers exact, given enough time: will they find all solutions, prove optimality, and prove unsatisfiability?
- If not, is there an approximation ratio?

Features:
- Can the modeller guide the solving? If yes, then how?
- In which areas has the techno 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, & MIP solvers.

**Definition (Local Search)**
Start from a candidate solution and iteratively modify it a bit. It is the basic idea behind LS and GA solvers.
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 variables.
- A set of clauses, denoting their conjunction (\/
):• A clause is a disjunction (\/
) of literals.
• A literal is a Boolean variable or its negation.
- Only for satisfaction problems: no objective function; otherwise: iterate over candidate objective values.

Example (in MiniZinc syntax)
- Variables: var bool: w, x, y, z;
- Clauses:
  
  constraint (not w \/
  not 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 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” ≠ “reduced from”. There are recipes to clausify non-Boolean constraints.

- There has been intensive research since the 1960s.

- We focus here on systematic search, namely DPLL [Davis-Putnam-Logemann-Loveland, 1962].
DPLL

Tree Search, upon starting from the empty valuation:

1. Perform inference (see below).
2. If some clause is unsatisfied, then backtrack.
3. If all variables have a value, then we have a solution.
4. Select an unvalued 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 \( \text{false} \), except one whose variable has no value yet, then that literal is made to evaluate to \( \text{true} \) so that the clause becomes satisfied.
Strategies and Improvements over DPLL

Search Strategies:
- On which 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: limited ways to guide the solving.
- It can scale to millions of variables and clauses.
- Encoding a problem can yield a huge SAT model.
- Some solvers can extract an unsatisfiable core, that is a subset of clauses that make the model unsatisfiable.
- It is mainly used in hardware and software verification.
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 discusses SAT solving and has a homework where a SAT model is designed 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 variables and clauses.
- Several theories extend the language, say bit vectors, uninterpreted functions, or linear integer arithmetic.
- SMT is only for satisfaction problems.
- OMT (optimisation modulo theories) extends SMT.

Definition

A theory

- defines variable types and constraint predicates, and
- is associated with a sub-solver for any conjunction of the supported constraint predicates.

Different SMT/OMT solvers may have different theories.
Example (Linear integer arithmetic; in MiniZinc syntax)

- **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 \(\lor\) b \(\lor\) (c \(\lor\) d) \(\lor\) (e \(\lor\) f \(\lor\) g);
    ```
SMT Solving: DPLL\((T)\)

Basic Idea:

- Separate the theory constraints and Boolean skeleton: each variable in the Boolean skeleton represents 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 \texttt{true} (respectively \texttt{false}).
Strategies and Improvements

Search Strategies:
- On which 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: limited ways to guide the solving.

- It is based on very efficient SAT technology.

- It is mainly used in hardware and software verification.
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, ...  

The **Embedded Systems** research group at Uppsala University *designs* SMT solvers.

Several other research groups at Uppsala University *use* SMT and OMT solvers, such as:

- Algorithmic Program Verification  
- Programming Languages  
- Theory for Concurrent Systems

My **Algorithms & Datastructures 3** (1DL481) course discusses SMT solving and has a homework where an SMT model is designed 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
Integer (Linear) Programming (IP, ILP)

Modelling Language:
- Only integer variables.
- A set of linear equality & inequality constraints (no ≠).
- Only for optimisation problems (otherwise: optimise a value): linear objective function.

Example (in MiniZinc syntax)
- Variables: var int: p; var int: q;
- Constraints:
  ```minizinc
  constraint p >= 0; constraint q >= 0;
  constraint p + 2 * q <= 5;
  constraint 3 * p + 2 * q <= 9;
  ```
- Objective: maximize 3 * p + 4 * q;
- Unique optimal solution: p = 1, q = 2
Mathematical Programming

- **0-1 linear programming**: linear (in)equality over variables over domain \{0, 1\}.

- **Linear programming (LP)**: linear (in)equality over floating-point variables.

- **Mixed integer (linear) programming (MIP)**: linear (in)equality over floating-point & int. 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 by dropping its integrality requirement on the 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 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 variable $v$ has a non-integer value $\rho$. 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 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 to linearise 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.
- ...
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).

The Optimisation research group at Uppsala University uses MIP solvers for 4G / 5G network planning and optimisation, etc.

My Algorithms & Datastructures 3 (1DL481) course discusses MIP solving and has a homework where a MIP 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
Constraint Programming (CP)

Modelling Language = full MiniZinc:

- Boolean, integer, enum, float, and / or set 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 variable types and constraint predicates (under distinct names).
- Some solvers support even higher-level variable types, such as graphs and strings, and associated predicates.
Domains

Definition

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

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

Note the difference between:

- a domain as a technology-independent declarative entity at the modelling level; and
- a domain as a procedural data structure for CP 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 variable is empty, then backtrack.
3. If all variables are fixed, then we have a solution.
4. Select a non-fixed variable $v$, partition its domain into two parts $\pi_1$ and $\pi_2$, 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 $\gamma$ deletes from the current domains of the variables of a $\gamma$-constraint the values that cannot be part of a solution to that constraint.

Examples

- For $x < y$: if $\text{dom}(x) = 1..4$ and $\text{dom}(y) = -1..3$, then delete $3..4$ from $\text{dom}(x)$ and $-1..1$ from $\text{dom}(y)$.

- For $\text{all_different}([x,y,z])$: if $\text{dom}(x) = \{1,3\} = \text{dom}(y)$ and $\text{dom}(z) = 1..4$, then delete $1$ & $3$ from $\text{dom}(z)$ so that it becomes the non-range $\{2,4\}$. The propagator of a constraint remains active as long as the Cartesian product of the domains of its variables is not known to contain only solutions to the constraint.
Strategies and Improvements

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

Improvements:
- Propagators, including for all 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; and
  - 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
  - ...
The MiniZinc toolchain was extended with backends for numerous 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 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
Local Search (LS)

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

![Diagram](image-url)
Example (Travelling Salesperson)

- **Problem**: Given a set of cities with connecting roads, find a tour (a Hamiltonian circuit) that visits each city exactly once, with the minimum travel distance.

- **Representation**: We see the set of cities as vertices $V$ and the set of roads as edges $E$ in a (not necessarily complete) undirected graph $G = (V, E)$.

- **Example**: We now design a local-search heuristic for this problem.
Example (Travelling Salesperson: 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 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 an assignment: The sum of all distances on the tour: \( \sum_{(a, b) \in s} \text{Dist}(a, b) \), as no constraint violation must be taken into account.

4. The **neighbour selector**: Select a random best neighbour.
Example (Travelling Salesperson: 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 an assignment:

4. The **neighbour selector**:
Example (Travelling Salesperson: 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 an assignment:

4. The neighbour selector:
Example (Travelling Salesperson: 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 an initial assignment.

2. The **neighbourhood** of candidate moves:

3. The **cost** of an assignment:

4. The **neighbour selector**:
Example (Travelling Salesperson: 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 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 an assignment:

4. The **neighbour selector**:
Example (Travelling Salesperson: 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 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 an assignment:
   The sum of all distances on the tour: \[
   \sum_{(a,b)\in s} \text{Dist}(a, b),
   \]
as no constraint violation must be taken into account.

4. **The neighbour selector:**
Example (Travelling Salesperson: 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 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 an assignment:
   The sum of all distances on the tour: $\sum \text{Dist}(a, b),
   (a,b)\in s$
   as no constraint violation must be taken into account.

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 Ågren.
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:
- Avoid cycles of moves & escape local optima.
- 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 much better to large instances.
  - May need a lot of tweaking: (meta)heuristics, . . .

Local search trades completeness and quality for speed!
Constraint-Based Local Search (CBLS)

- MiniZinc-style modelling language:
  - Boolean, integer, and / or 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 approximation ratio.

- White-box: one must design a search algorithm, which probes the cost impacts for guidance.

- More scalable than systematic approaches.
**Definition**

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

**Example**

For $a \leq b$, let $\alpha$ and $\beta$ 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.

Example

For \( p = a \times b \), whenever the value \( \alpha \) of \( a \) or the value \( \beta \) of \( b \) is 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 syntax for one-way constraints, such as \( p \leftarrow a \times b \) 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(...)` , the initial assignment has distinct values for all variables, and the neighbourhood only has moves that swap the values of two variables, assuming the number of variables is equal to the number of values.

When building a CBLS model, a MiniZinc backend must:

- Aptly assort the otherwise all explicit & 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 variable $\text{Next}[c]$ for each city $c$:

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

Three consecutive current assignments, preserving the satisfaction of the $\text{circuit}(\text{Next})$ constraint and improving the objective value:

\begin{align*}
f(s) &= 709 \\
f(s) &= 656 \\
f(s) &= 530
\end{align*}
The MiniZinc toolchain was extended with:
- our fzn-oscar-cbls backend to the OscaR.cbls solver;
- 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:
- Algorithms & Datastructures 3 (1DL481) discusses LS and has a homework where an LS program is written.
- Artificial Intelligence (1DL340) discusses LS.
- Part 2 of Combinatorial Optimisation and Constraint Programming (1DL442) covers CBLS in some depth.
- 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
Crossfertilisation

- Each technology has advantages and drawbacks.
- Good ideas from one technology can be applied to another.
- 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 COP)
Follow an LS procedure, but each move is performed by:
1. Undo the values for a subset of the variables.
2. Use CP 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.
The MiniZinc toolchain was extended with:

- LCG backends: Chuffed (bundled), Google CP-SAT;
- a CIP backend: SCIP;
- LNS backends: the solvers of the Gecode 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.
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
Example: Pigeonhole Problem

Example (Pigeonhole)

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

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;
- that it is often important for solving efficiency to use pre-defined constraint predicates.
### Pigeonhole: Models

#### Using `allDifferent`

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

#### Using `!=`

```plaintext
int: n;
array[1..n] of var 1..(n-1): Hole;
constraint forall(i,j in 1..n where i < j) (Hole[i] != Hole[j]);
solve satisfy;
```
Constraint Predicate Definitions

**Built-in** `all_different` **for probably all CP solvers**

```plaintext
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**

```plaintext
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

```plaintext
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] \ Y 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 variables (where `Y[i,v]=true` iff `X[i]=v`, and `A[i,v]=true` iff `v in X[1..i]`) as well as \(O(n^2)\) clauses of 2 or 3 literals. This is more compact and usually more efficient than the direct encoding with \(O(n^3)\) clauses of 2 literals over only `Y`.
The MiniZinc Toolchain

Comparison Criteria

SAT

SMT & OMT

IP & MIP

CP

LS & CBLS

Hybrid Technologies

Case Study

Choosing a Technology and Backend

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

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

<table>
<thead>
<tr>
<th>n</th>
<th>backend</th>
<th>all_different</th>
<th>!=</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>mzn-gecode</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>10</td>
<td>mzn-gurobi</td>
<td>&lt; 1</td>
<td>58</td>
</tr>
<tr>
<td>11</td>
<td>mzn-gecode</td>
<td>&lt; 1</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>mzn-gurobi</td>
<td>&lt; 1</td>
<td>285</td>
</tr>
<tr>
<td>12</td>
<td>mzn-gecode</td>
<td>&lt; 1</td>
<td>113</td>
</tr>
<tr>
<td>12</td>
<td>mzn-gurobi</td>
<td>&lt; 1</td>
<td>3704</td>
</tr>
<tr>
<td>100</td>
<td>mzn-gecode</td>
<td>&lt; 1</td>
<td>time-out</td>
</tr>
<tr>
<td>100</td>
<td>mzn-gurobi</td>
<td>&lt; 1</td>
<td>time-out</td>
</tr>
<tr>
<td>300</td>
<td>mzn-gecode</td>
<td>&lt; 1</td>
<td>time-out</td>
</tr>
<tr>
<td>300</td>
<td>mzn-gurobi</td>
<td>24</td>
<td>time-out</td>
</tr>
<tr>
<td>100000</td>
<td>mzn-gecode</td>
<td>&lt; 1</td>
<td>time-out</td>
</tr>
<tr>
<td>1000000</td>
<td>mzn-gecode</td>
<td>5</td>
<td>time-out</td>
</tr>
</tbody>
</table>

Time, in seconds, to prove unsatisfiability:
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
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 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 can be more suited to different backends.
- The performance on small instances does not always scale up to larger instances.
- Sometimes, a good search strategy (see Topic 8: Inference & Search in CP & LCG) is more important than a good model.
- Not all backends that use the same technology have comparable performance.
- Some pure problems can be solved by specialist tools, say Concorde for the travelling salesperson problem: 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 & 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.