Topic 7: Solving Technologies
(Version of 2nd December 2018)

Jean-Noël Monette and Pierre Flener

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL441:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

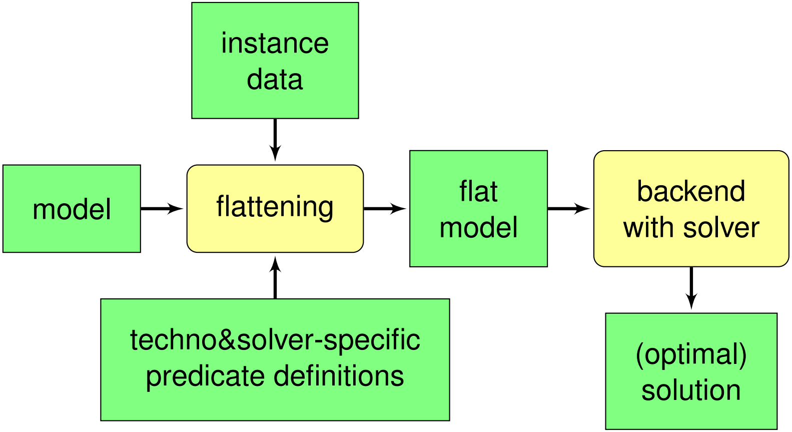
UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Outline

MiniZinc: Model Once, Solve Everywhere!



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

UPPSALA UNIVERSITET

Outline

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 a model as input:

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

Methodologies, *usually without* modelling and solvers:

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

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, or 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, CP, LCG, and MIP solvers.

Definition (Local Search)

Start from a candidate solution and iteratively modify it. It is the basic idea behind LS and GA solvers.

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Boolean Satisfiability Solving (SAT)

Modelling Language:

- Only Boolean variables.
- A conjunction (/\) of clauses:
  - 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

- 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 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” $\neq$ “reduced from”.
- There has been intensive research since the 1960s.
- We focus here on systematic search, namely DPLL [Davis-Putnam-Logemann-Loveland, 1962].

DPLL

Tree Search:

1. At the root of the tree, start from the empty valuation.
2. Perform inference (see below).
3. If some clause is unsatisfied, then backtrack.
4. If all variables have a value, then we have a solution.
5. Pick an unvalued variable α and make two branches: one with $\alpha = \texttt{true}$, and the other one with $\alpha = \texttt{false}$.
6. Explore each of the two branches, starting from step 2.

Inference:

- Unit propagation: If all the literals in a clause evaluate to `false`, except one whose 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 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.

SAT @ MiniZinc and Uppsala University

- The MiniZinc toolchain has been extended with the PicatSAT backend for the SAT solver Lingeling.
- 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

SAT Modulo Theories (SMT)

Modelling Language:

- Language of SAT: Boolean variables and clauses.
- Several theories extend the language, say bit vectors, uninterpreted functions, or linear integer arithmetic.
- Mainly for satisfaction problems.

Definition

A theory

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

Not all SMT solvers support the same collection of theories.

Example (Linear integer arithmetic)

- Variables: `var int: x; var int: y;`
- Constraints:

```
constraint x >= 0; 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 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 `true` (respectively `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 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.

SMT @ MiniZinc and Uppsala University

- The MiniZinc toolchain has been extended with the fzn2smt compiler (do not use it in this course), which generates SMTlib models that can be fed to any SMT solver, by default Yices, but also CVC4, Z3, . . .
- The Embedded Systems research group at Uppsala University *designs* SMT solvers.
- Several other 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 discusses SMT solving and has a homework where an SMT model is designed and fed to an SMT solver.

UPPSALA UNIVERSITET

Outline

Integer (Linear) Programming (IP)

Modelling Language:

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

Example

- Variables: `var int: p; var int: q;`
- Constraints:

```
constraint p >= 0 /\ 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)equalities over variables over domain $\{0, 1\}$.
- Linear programming (LP):
  linear (in)equalities over floating-point variables.
- Mixed integer (linear) programming (MIP):
  linear (in)equalities over floating-point & int. variables.
- Quadratic programming (QP):
  quadratic objective function.
- $\dots$

There has been intensive research since the 1940s.

IP Solving

Basic Idea = Relaxation:

- Polytime algorithms for solving LP models exist, such as the simplex and interior-point methods.
- 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:

1. Solve the LP relaxation.
2. If the LP was unsatisfiable, then backtrack.
3. If all variables have an integral value in the LP solution, then the latter is also an IP solution:
   add the constraint that the next solution must have a better objective value than this solution.
4. Otherwise, some variable α has a non-integral value ρ. Make two branches:
   one with $\alpha \leq \lfloor \rho \rfloor$, and the other one with $\alpha \geq \lceil \rho \rceil$.
5. 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 encode 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), used 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);
  - 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

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: not all CP solvers support the same collections of variable types and constraint predicates.
- Some solvers support even higher-level variable types, such as graphs and strings, and associated predicates.

Domains

Definition

The domain of a variable α, denoted here by dom(α), is the set of values that α 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.
- A model is unsatisfiable if a variable domain is 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:

Satisfaction problem:

1. At the root, set each variable domain as in the model.
2. Perform inference (see the next slide).
3. If the domain of some variable is empty, then backtrack.
4. If all variables are fixed, then we have a solution.
5. Pick an unfixed variable α, partition its domain into two parts π_1 and π_2, and make two branches:
   one with $\alpha \in \pi_1$, and the other one with $\alpha \in \pi_2$.
6. Explore each of the two branches, starting from step 2.

Optimisation problem: when a solution is found, add the constraint that the next solution must have a better objective value (see step 3 of branch-and-bound for IP on slide 28).

CP Inference

Definition

A propagator for a predicate γ removes from the current domains of the variables of a γ-constraint ~~the~~ values that cannot be part of a solution to that constraint.

Examples

- For `x<y`: if dom(`x`) = `1..4` and dom(`y`) = `-1..3`, then remove `3..4` from dom(`x`) and `-1..1` from dom(`y`).
- For `all_different([x,y,z])`: if dom(`x`) = `{1,3}` = dom(`y`) and dom(`z`) = `1..4`, then remove `1` & `3` from dom(`z`) so that it becomes the non-interval `{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 split the current 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 removed: 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:
  - Scheduling
  - Timetabling
  - Rostering
  - . . .

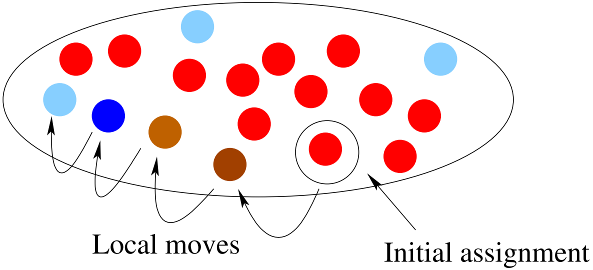
CP @ MiniZinc and Uppsala University

- The MiniZinc toolchain has been extended with backends for numerous CP solvers, such as Choco, Gecode (bundled), 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 (1DL441) course covers CP in depth.

Outline

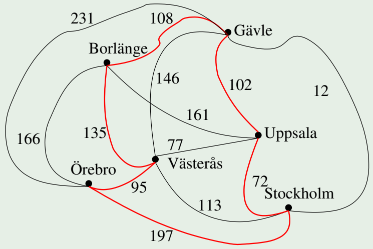
Local Search (LS)

- 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 has been found, or when an allocated resource has been exhausted, such as time spent or iterations made.



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:** s:

We now design a local-search heuristic for this problem.

Example (Travelling Salesperson: Choices)

We must define:

1. The **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 circuit: 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 circuit: NP-hard!
2. The **neighbourhood** of candidate moves:
   Replace two edges on the circuit s by two other edges so that s is still a circuit.
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 circuit: NP-hard!
2. The **neighbourhood** of candidate moves:
   Replace two edges on the circuit s by two other edges so that s is still a circuit.
3. The **cost** of an assignment:
   The sum of all distances on the circuit: $\displaystyle\sum_{(a,b)\in s} \text{Dist}(a,b)$,
   as no constraint violation must be taken into account.
4. The **neighbour selector**:

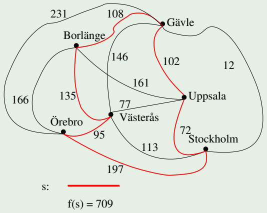
Example (Travelling Salesperson: Choices)

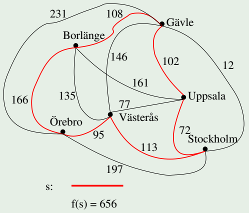
We must define:

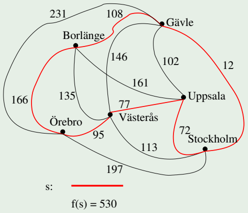
1. The **initial assignment**:
   An edge set $s \subseteq E$ that forms a circuit: NP-hard!
2. The **neighbourhood** of candidate moves:
   Replace two edges on the circuit s by two other edges so that s is still a circuit.
3. The **cost** of an assignment:
   The sum of all distances on the circuit: $\displaystyle\sum_{(a,b)\in s} \text{Dist}(a, b)$,
   as no constraint violation must be taken into account.
4. The **neighbour selector**:
   Select a random best neighbour.

Example (Travelling Salesperson: Sample Run)

Three consecutive improving satisfying 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 candidate solutions and cross them.

Systematic Search (as in SAT, SMT, MIP, 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: heuristics, parameters, . . .

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 <= 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.

Example

For `p = a * b`, whenever the value α of `a` or the value β 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 <== a * b` in OscaR.cbls, but MiniZinc does not make such a distinction.

Definition

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

Example

For `all_different(...)`, the initial candidate solution 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.

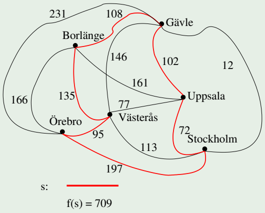
This is much more involved than just flattening and solving.

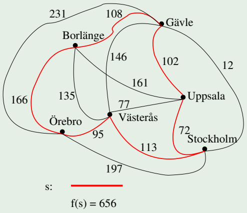
Example (Travelling Salesperson: Model and Solve)

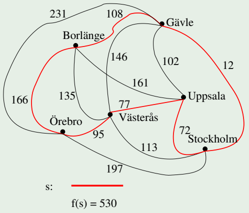
Recall the model, from Topic 1: Introduction, with a variable `Next[c]` for each city `c`:

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

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







(CB)LS @ MiniZinc and Uppsala University

- The MiniZinc toolchain can be 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 and *uses* them (see slide 39).
- 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 (1DL441) covers CBLS in some depth.
  - Machine Learning (1DT071) discusses LS.

Outline

Crossfertilisation

- Each technology has advantages and drawbacks.
- Good ideas from one techno 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)

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 to generate linear inequalities for non-linear constraints.

UPPSALA UNIVERSITET

Hybrids @ MiniZinc and Uppsala University

- The MiniZinc toolchain has been extended with:
  - LCG backends: Chuffed (bundled), Google OR-Tools;
  - a CIP backend: SCIP;
  - LNS backends: the solvers of the Gecode and Google OR-Tools backends can perform LNS (prescribed via MiniZinc annotations).
- The Optimisation research group at Uppsala University *uses* hybrid solvers (see slide 39).

Outline

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 `all_different`

```
int: n;
array[1..n] of var 1..(n-1): Hole;
constraint all_different(Hole);
solve satisfy;
```

Using binary disequalities

```
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

```
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 variables (where `Y[i,v]=true` iff `X[i]=v`, and `A[i,v]=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, ...

```
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:

n	backend	`all_different`	disequalities
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
100000	`mzn-gecode`	< 1	time-out
1000000	`mzn-gecode`	5	time-out

UPPSALA UNIVERSITET

Outline

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 kinds 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, finding the right search strategy is more important than coming up with 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.
2nd edition, Springer, 2012.