Topic 5: Solving Technologies
(Version of 18th October 2017)

Jean-Noël Monette and Pierre Flener

ASTRA Research Group
on Combinatorial Optimisation
Uppsala University
Sweden

Course 1DL448:
Modelling for Combinatorial Optimisation

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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)
- ...

UPPSALA UNIVERSITET

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?

Guarantee:

- 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?

UPPSALA UNIVERSITET

Outline

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.
- There has been intensive research since the 1960s.

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

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: $x = 0, y = 0$

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 allows quantified formulae under some conditions.
- It is mainly used in hardware and software verification.

UPPSALA UNIVERSITET

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 Programming (IP)

Modelling Language:

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

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 programming (MIP):
  linear (in)equalities over floating-point & int. variables.
- Quadratic programming (QP):
  quadratic objective function.
- ...

There has been intensive research since the 1950s.

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:
  - G12.MIP (do not use it in this course);
  - Cbc (open-source);
  - Gurobi Optimizer (commercial: requires a license);
  - CPLEX Optimizer (commercial: requires a license).
- The research group of Prof. Di Yuan *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.

UPPSALA UNIVERSITET

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.

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

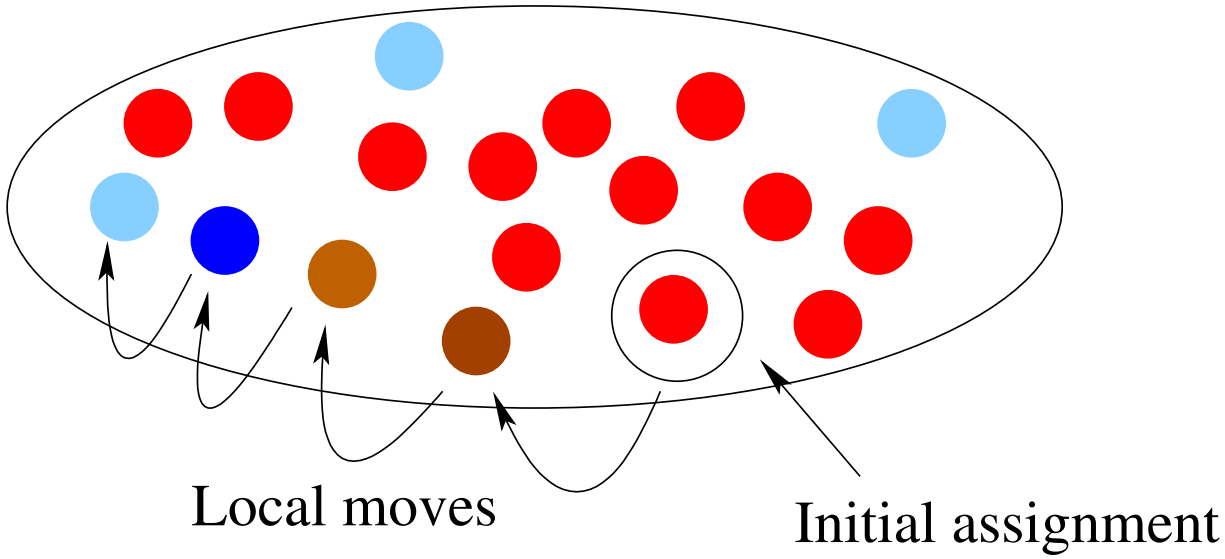
- The MiniZinc toolchain has been extended with backends for numerous CP solvers, such as Choco, G12.FD (bundled; do not use it in this course), Gecode (bundled), JaCoP, Mistral, SICStus Prolog, ...
- My ASTRA Combinatorial Optimisation research group at Uppsala University contributes to the *design* of CP solvers and *uses* them, say for air traffic management, the configuration of wireless sensor networks, robot task sequencing, etc.
- My Combinatorial Optimisation using Constraint Programming (1DL441) course covers CP in depth.

UPPSALA UNIVERSITET

Outline

Perturbative / Local Search (LS)

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



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 in order to:

- avoid getting trapped in cycles and local optima;
- explore many parts of the search space; and
- 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.

Constructive 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, ...

Perturbative / 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 the found solution is optimal.
+ Often scales well to large instances.
− May need a lot of tweaking: heuristics, parameters, ...

Local search trades exactness 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 slide.
  - 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 constructive approaches.

A CBLS model has up to three sorts of constraints:

- A constraint with violation is explicit in the model and soft: it can be violated during search but ought to be satisfied in a solution.
- A one-way constraint is explicit in the model and hard: it is kept satisfied during search.
- An implicit constraint is not in the model but hard: it is kept satisfied during search (by satisfying it in the initial candidate solution and then only making satisfaction-preserving moves).

When building a CBLS model, a MiniZinc backend must:

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

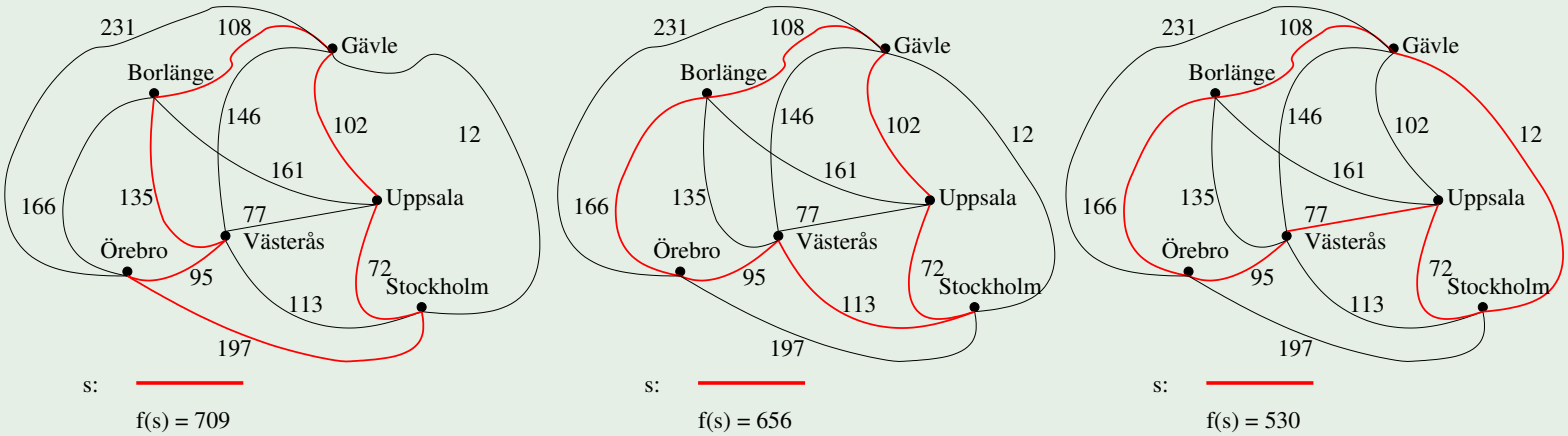
This is much more involved than just flattening and solving.

Example (Travelling Salesperson: Model and Solve)

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

```
3 solve minimize sum(c in Cities)(Dist[c,Next[c]]);
4 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.
- My ASTRA Combinatorial Optimisation research group at Uppsala University contributes to the *design* of CBLS solvers and *uses* them (see slide 24).
- 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.
  - Combinatorial Optimisation using Constraint Programming (1DL441) covers CBLS in some depth.
  - Machine Learning (1DT071) discusses LS.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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), G12.lazyFD (bundled; do not use it in this course), Google OR-Tools, and Opturion CPX;
  - a CIP backend: SCIP.
  - LNS backends: the solvers of the Gecode and Google OR-Tools backends can perform LNS (prescribed via MiniZinc annotations).
- The Embedded Systems research group at Uppsala University *designs* a SAT+CP hybrid solver.
- My ASTRA Combinatorial Optimisation research group at Uppsala University *uses* hybrid solvers (see slide 24).

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

Pigeonhole: Models

Using `alldifferent`

```
1 int: n;
2 array[1..n] of var 1..(n-1): Hole;
3 constraint alldifferent(Hole);
4 solve satisfy;
```

Using binary disequalities

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

Constraint Predicate Definitions

Built-in `alldifferent` 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 `alldifferent` 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) =
   [...]

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;
```

UPPSALA UNIVERSITET

Pigeonhole: Experimental Comparison

Time, in seconds, to prove unsatisfiability:

n	backend	`alldifferent`	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?

UPPSALA UNIVERSITET

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.