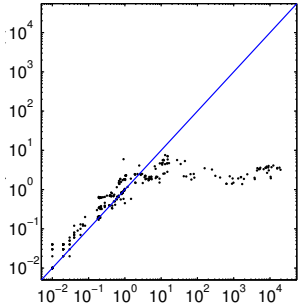
L08: CP Search Strategies and Algorithm Configuration



Prof. Tias Guns and Dr. Dimos Tsouros

KU LEUVEN

Partly based on slides from Pierre Flener, Uppsala University and Lars Kotthoff, University of Wyoming.

Goal of the lecture

Different solvers will behave differently for different problems...

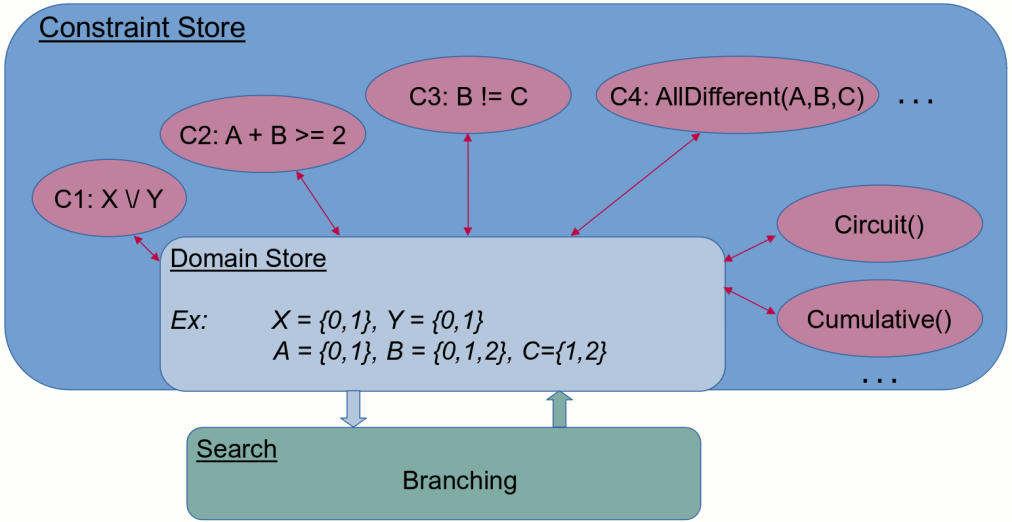
Typical CP problems are NP-hard, and we can not expect to solve all problems efficiently (unless P=NP).

But many problems of practical interest:

- Can be practically solved
- Can be solved even faster with the right solving technology
- And even even faster with the right solver configuration:
  generic solvers include many design choices and options to influence them.

Outline

CP solver structure



Propagation strength

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.
Not all impossible values need to be deleted:

- A domain-consistency (DC) propagator deletes all impossible values from the domains.
- A bounds-consistency (BC) propagator only deletes all impossible minimum and maximum values from the domains.
- A value-consistency (VC) propagator is only awoken when at least one of its decision variables became fixed.

There exist other, unnamed consistencies for propagators.
There is a trade-off between the time & space complexity of a propagator and its achieved deletion of domain values.

Example (Linear equality constraints)

Consider the linear constraint $3 * x + 4 * y = z$
with $\text{dom}(x) = \{0, 1\} = \text{dom}(y)$ and $\text{dom}(z) = \{0, \ldots, 10\}$:

- A bounds-consistency propagator reduces $\text{dom}(z)$ to $\{0, \ldots, 7\}$.
- A domain-consistency propagator reduces $\text{dom}(z)$ to $\{0, 3, 4, 7\}$.

Time complexity:

- A bounds-consistency propagator for a linear equality constraint can be implemented to run in $\mathcal{O}(n)$ time, where n is the number of decision variables in the constraint.
- A domain-consistency propagator for a linear equality constraint can be implemented to run in $\mathcal{O}(n \cdot d^2)$ time, where n is the number of decision variables in the constraint and d is the sum of their domain sizes, hence in time pseudo-polynomial = exponential in the input magnitude.

Controlling the CP Inference

In the past, some solvers allowed users to set the propagation level for individual constraints (most notably the Gecode solver).

Nowadays, solvers developers typically implement one (e.g. the strongest one) and run with it.

Outline

Historic perspective

CP as a depth-first search framework

- Historically, constraint programming solvers allow programming your own constraints as well as programmable search:
  variable/value ordering and choices, e.g. which variables, what branching decision are created... also full access to the constraint and domain store.

CP as generic satisfaction/optimisation solver

- Since a number of years, the CP community has been advancing on 'the holy grail' of black box search strategies that work well in general, so users can especially focus on the modeling part.
- Solver-indendent modeling languages have taken one step further away from programmable search, allowing setting at most a variable/value ordering as a solver parameter.

Search Strategies

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?

The search is usually depth-first search.

Variable Selection Strategy

The variable selection strategy has an impact on the size of the search tree.

Example (Impact of the variable selection strategy)

Consider $x \in \{1,2\}$, $y \in \{1,2,3,4\}$, $z \in \{1,\ldots,6\}$,
branching on all domain values, but no constraints:

- If selecting the decision variables in the order x, y, z, then the CP search tree has $1 + 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 6 = 59$ nodes and $2 \cdot 4 \cdot 6 = 48$ leaves.
- If selecting the decision variables in the order z, y, x, then the CP search tree has $1 + 6 + 6 \cdot 4 + 6 \cdot 4 \cdot 2 = 79$ nodes and also $6 \cdot 4 \cdot 2 = 48$ leaves.

Definition (First-Fail Principle)

To succeed, first try where you are most likely to fail. In practice:

- Select a decision variable with the smallest current domain.
- Select a decision variable involved in the largest number of constraints.
- Select a decision variable recently causing the most backtracks.

Example (Impact of the variable selection strategy)

Finding the first solution to 250-queens with CPMpy-Choco (CP):

search	seconds
default	0.17
smallest-domain, lb value	0.19
conflict-history search	0.22
activity-based search	0.23
domain size over weighted degree	> 30
input order, lb value	> 30

Domain Partitioning Strategy

(including value orderings)

Example (Impact of the domain partitioning strategy)

Consider $x \in \{1,2\}$, $y \in \{1,2,3,4\}$, $z \in \{1,\ldots,6\}$,
domain consistency for $x * y = z$, $x \neq y$, $x \neq z$, and $y \neq z$,
smallest-domain variable selection, and depth-first search:

- If the domain is split into singletons by increasing order,
  then 6 CP nodes are explored before finding the (unique) solution.
- If the domain is split into singletons by decreasing order, then only 2 CP nodes (the root and a leaf) are explored
  before finding the (unique) solution, without backtracking.

Definition (Best-First Principle)

First try a domain part that is most likely, if not guaranteed, to have values that lead to solutions.

This may be like how one would make the greedy choice in a greedy algorithm for the problem at hand, considering its objective function.

Example (Impact of the domain partitioning strategy)

Finding the first solution to 750-queens with CPMpy-Choco

search	seconds
smallest-domain, lb value	3.35
smallest-domain, ub value	2.88

Outline

Other search strategies

warmstarting if a previous solution is known
or a *solution hint*: what value to try first for specific variables (very useful in repeated solving)

heuristics/local search to quickly find a feasible solution
provides an upper bound (for minimisation problems), good user experience

probing try assigning a variable to a value and propagate it (no search), derives failing values, or hidden consequences (e.g. if comparing propagation effect of assigning a Boolean variable to True and to False)

restarts during search for clause learning solvers, backjump to the root node. Can be very effective (because variable activity was updated during previous search, so most active variables are now at 'top' of the tree)

Other solver configuration options

These are often very solver-specific!!

OrTools CP-SAT:

- ”cp_model_probing_level”: how much effort on probing
- ”search_branching”: search strategy (automatic, fixed, hint, LP-based, ...)
- ”symmetry_level”: symmetry detection and breaking before/during search
- ”use_phase_saving”: value selection = last value the variable was assigned to

Choco: In Python API only some search strategies are exposed. Java API: programmable search and tons of options!
Gurobi categories:

- Tolerance parameters which control solution quality requirements
- Algorithmic control parameters for Presolve, Simplex, Barrier, Scaling, MIP and MIP Cuts

Outline

Algorithm Selection and Configuration

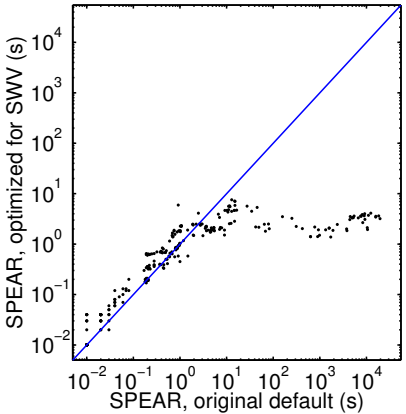
Which solver to use? Which choices to make?

Algorithm Selection choose the best *algorithm* for solving a problem
Algorithm Configuration choose the best *parameter configuration*

Large performance improvements possible, without changing the model!

Based on slides from Lars Kotthoff, Univ of Wyoming

Performance Improvements example



Hutter, Frank, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. “Boosting Verification by Automatic Tuning of Decision Procedures.” FMCAD 07.

Algorithm Selection

Given a problem, choose the best algorithm to solve it.

“algorithm” used in a very loose sense

- constraint solvers
- search strategies
- modelling choices
- different types of consistency
- . . .

As long as a finite (typically small) set of ”algorithms” is available: the *portfolio*.

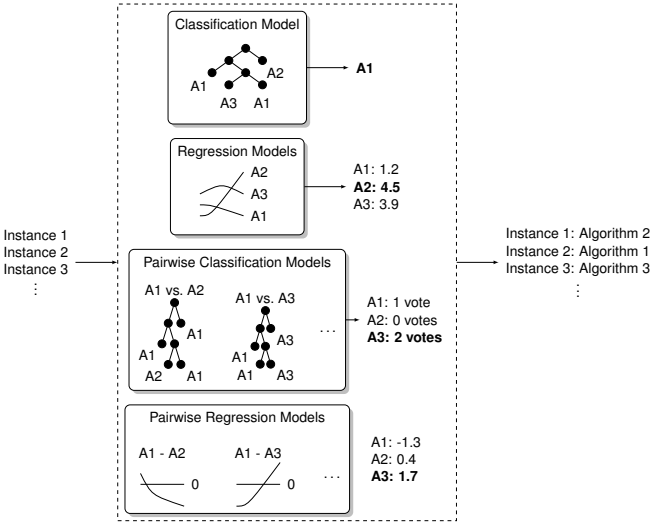
Rice, John R. “The Algorithm Selection Problem.” Advances in Computers 15 (1976): 65–118.

Building an Algorithm Selection System

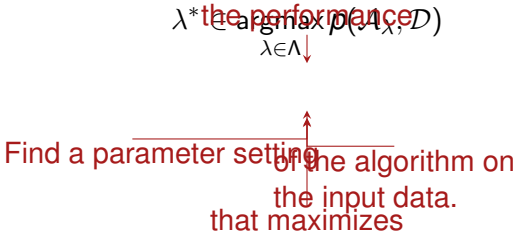
Most often 'instance specific' algorithm configuration:

- For an unseen instance: which algorithm (from the finite set) should I run?
- Instance-specific: allowed to analyse the instance and decide based on *features* that you can derive from it
  - nr of variables, constraints
  - types of constraints used
  - constraint network (e.g. GNNs)
  - probing (run an algorithm for short time, collect statistics)
- Most often Machine Learning is used to build a **performance model**, and predict which algorithm to use.

Types of Performance Models



Algorithm Configuration = Automated Parameter Tuning

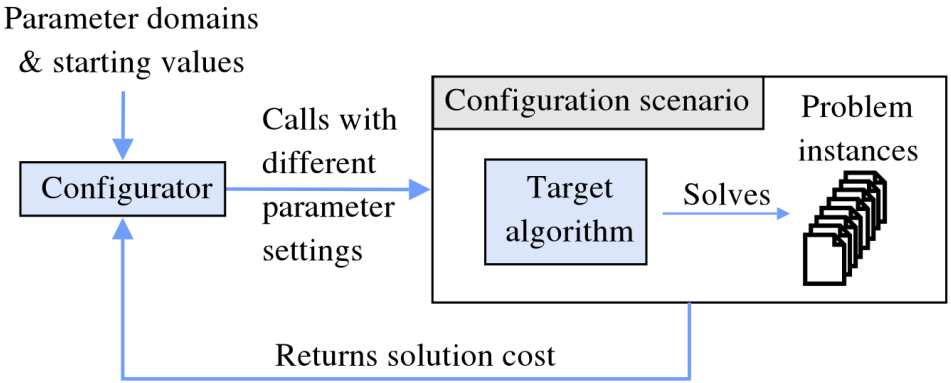


Parameters?

- anything you can change that makes sense to change
- typically multiple parameters with many values (possible continuous): a **large** (possibly infinite) set
- e.g. search heuristic, variable ordering, type of global constraint decompositions, presolve settings, different thresholds, ...
- Example: CPLEX MIP solver, 76 parameters; Spear SAT solver, 26 parameters

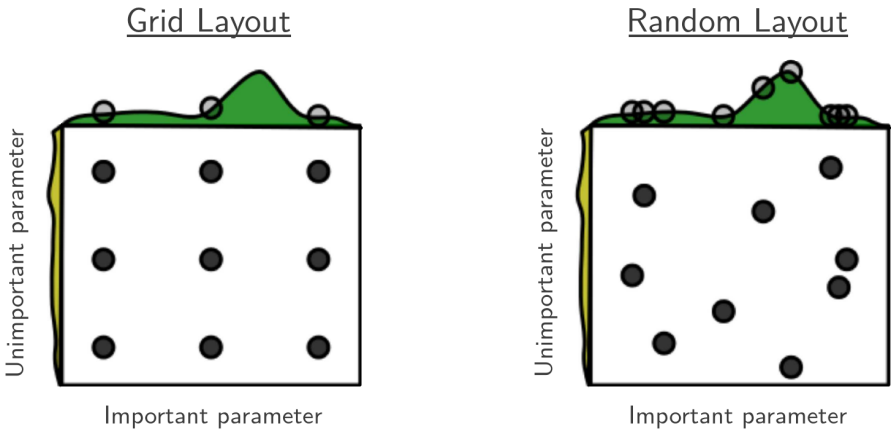
Typically for a *set* of related instances (e.g. nurse rostering problems that the hospital solves every day)

Algorithm Configuration



Frank Hutter and Marius Lindauer, “Algorithm Configuration: A Hands on Tutorial”, AAAI 2016

Grid and Random Search



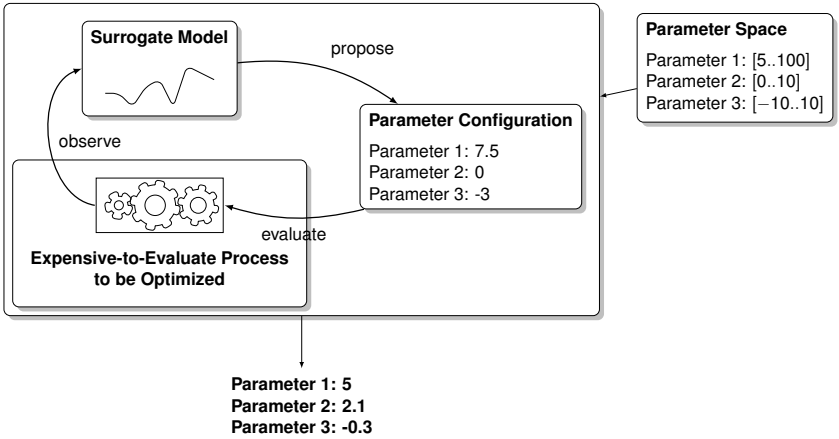
Bergstra, James, and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization.” J. Mach. Learn. Res. 13, no. 1 (February 2012): 281–305.

Model-Based Optimization

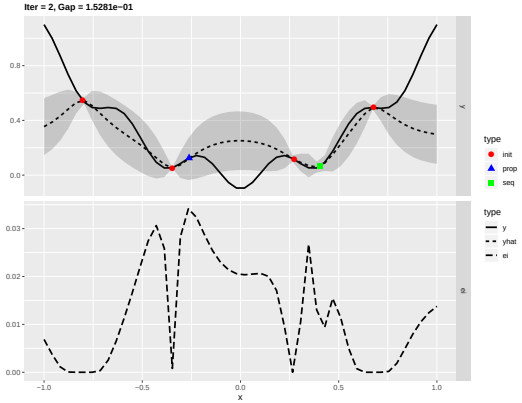
- evaluate small number of initial (random) configurations
- use (probabilistic) Machine Learning to train a surrogate model of parameter-performance surface based on this
- use *acquisition function* to decide most promising configuration to try next
- repeat, stop when resources exhausted or desired solution quality achieved

Allows targeted exploration of a limited number of promising configurations (time budget).

Model-Based Optimization

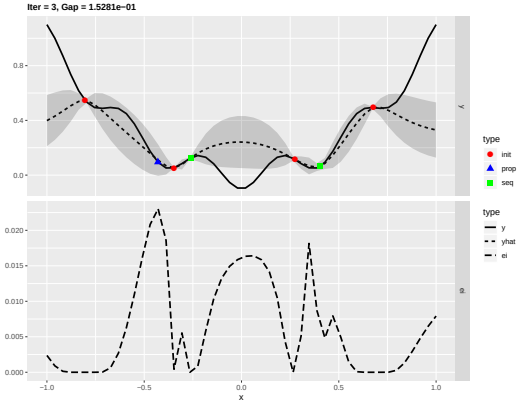


Sequential Model-Based Optimization (SMBO) Example



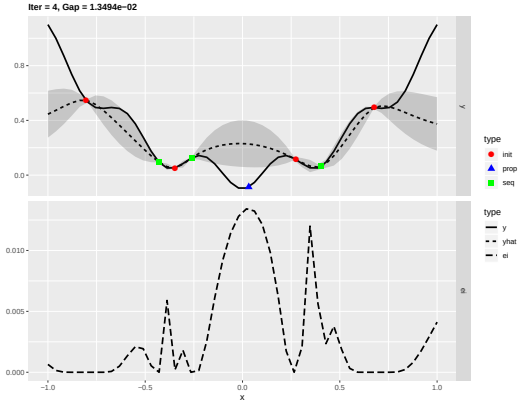
Bischl, Bernd, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. “MlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions,”

Sequential Model-Based Optimization (SMBO) Example



Bischl, Bernd, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. “MlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions,”

Sequential Model-Based Optimization (SMBO) Example



Bischl, Bernd, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. “MlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions,”

Time Budget

How much time/how many function evaluations?

- too much $\rightarrow$ wasted resources
- too little $\rightarrow$ suboptimal result
- use statistical tests?
- evaluate on parts of the instance set
- for runtime: adaptive capping = solver timeout with current best runtime.

Need to evaluate on an unseen test-set to avoid 'over-tuning'.

Summary

Algorithm Selection choose the best *algorithm* for solving a problem

Algorithm Configuration choose the best *parameter configuration* for solving a problem with an algorithm

- mature research areas
- can combine configuration and selection
- effective tools are available (e.g. HyperOpt, does SMBO (without adaptive capping...))