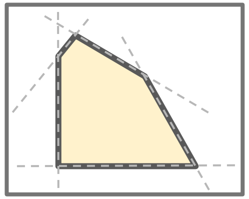
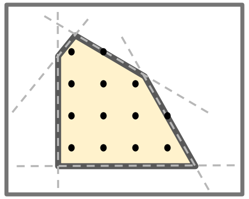
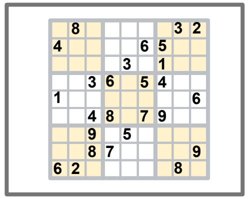
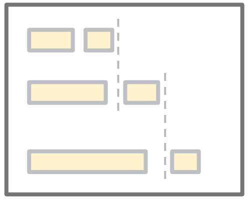
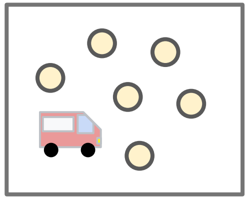
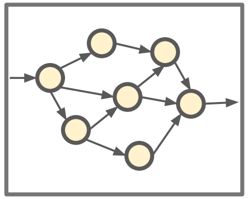
OR-Tools

Laurent Perron (lperron@google.com)
Vincent Furnon (vfurnon@google.com)
Corentin Le Molgat (corentinl@google.com)

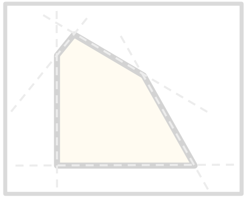
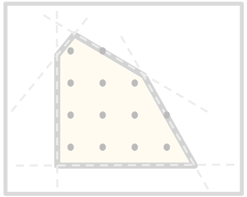
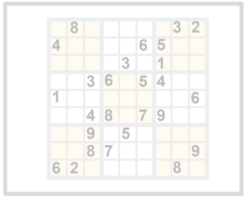
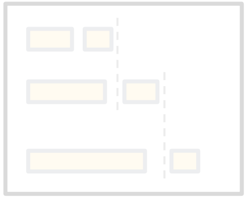
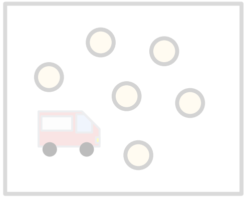
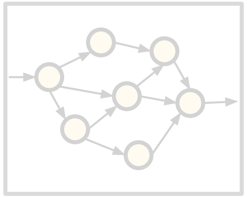
Operations Research @ Google

- The Operations Research (OR) team is mostly based in Paris and Cambridge (US)
- Started ~15 years ago, currently 40+ people
- Mission: **Optimize Google**
    - Internal consulting: build and help build optimization applications
    - Tools: develop core optimization algorithms
- Other software engineers with OR backgrounds are distributed throughout the company

Combinatorial optimization

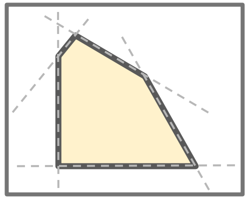
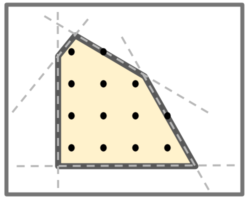
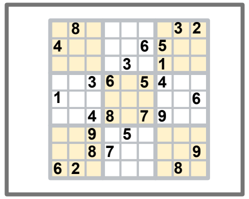
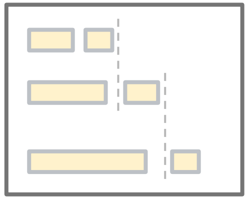
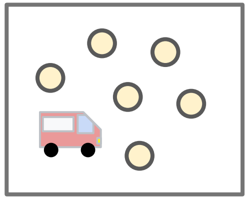
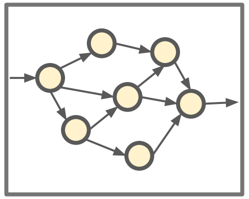







Solvers

LP MIP SAT CP VRP Graph

OR-Tools

LP

MIP

SAT

CP

VRP

Graph

Tech Stack Overview

- Apache 2.0 licence
- C++ Framework with SWIG/pybind11 wrappers in .Net, Python, Java Go (internal), WASM
- Uses external dependencies (abseil, protobuf and third party solvers Gurobi, SCIP, CoinOR CBC/CLP, CPLEX, XPRESS)
- Packages available:
  - C# Google.OrTools on nuget.org
  - Java ortools-java on maven
  - Python ortools on pypi.org

OSS Ecosystem support

- Ecosystem integration is mandatory to help adoption.
- C++ ecosystem have no strong consensus for a package manager
  - Windows: vcpkg (increasing adoption, backed by Microsoft),
  - brew on macOS X (popular)
  - Linux distros package managers (packages are often outdated)

User profiles:

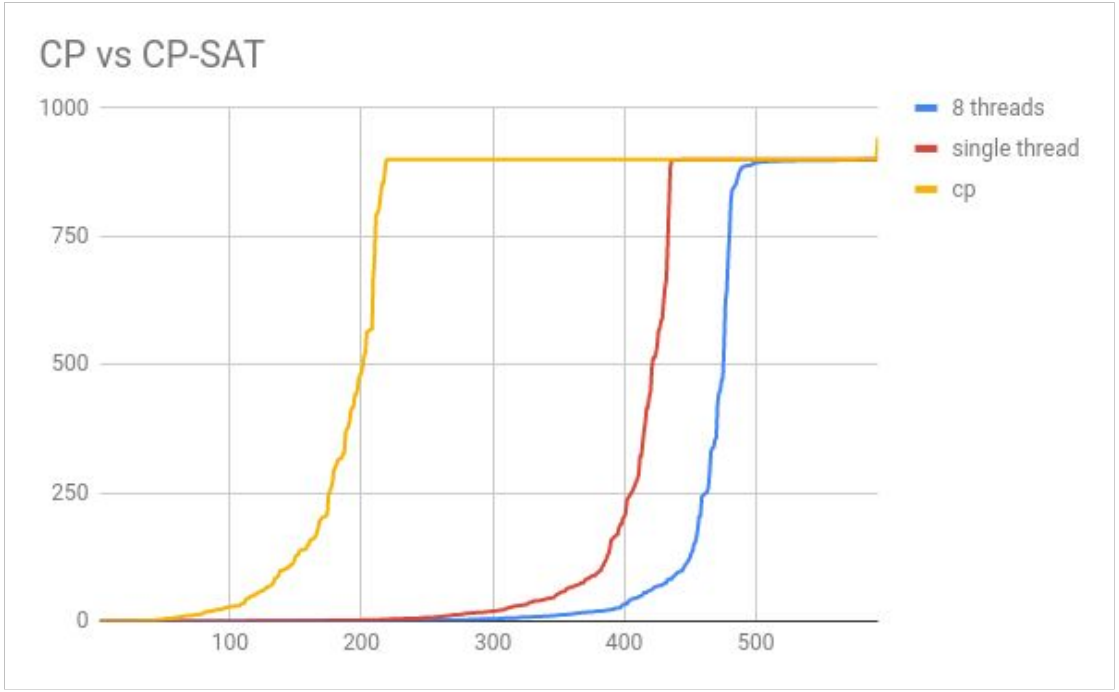
- Academic: Python, (julia)
- Industry: .Net holding, mostly python now
- Devops/startupers: Go / node.js / python (WASM early adopter ?)
- Java: fading out ?
- C++: no usage visibility (proprietary software ?)

CP-SAT-LP

Laurent Perron (lperron@google.com)
Frédéric Didier (fdid@google.com)
Steven Gay (stevengay@google.com)

What is CP-SAT-LP?

A Breakthrough: CP vs CP-SAT



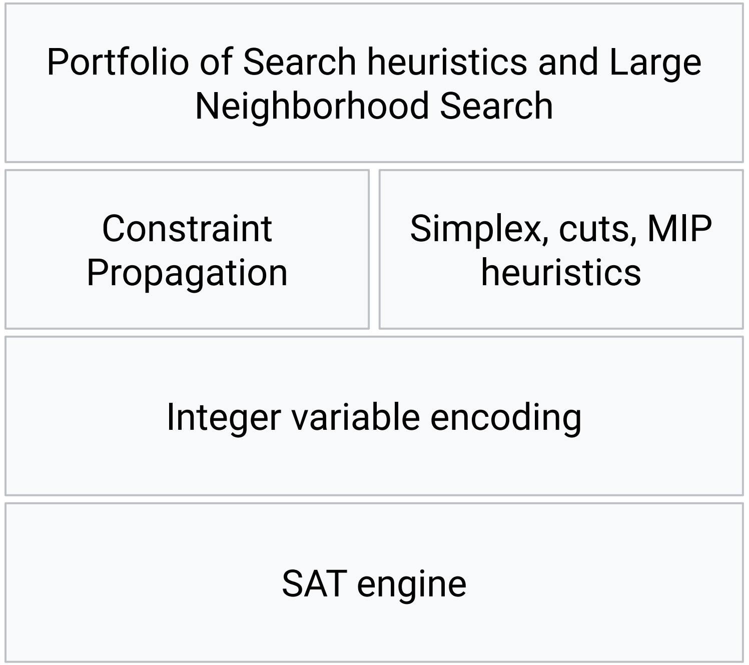
Google

Google Research

CP-SAT-LP influence

- Conflict Directed Clause Learning was a breakthrough in the SAT community in 1999-2000.
- This revolution can benefit the rest of the combinatorial optimization community.
- CP-SAT-LP is a reboot of an hybrid Constraint Programming solver, an Integer Linear Programming solver, and a MaxSAT solver on top of a CDCL SAT solver.
- The key takeaway is that CDCL allows investing more time in costly techniques that benefit only on a subset of problems.
- This work is inspired by Chuffed and by Peter Stuckey's presentation Search is dead
- It is a follow-up of our CPAIOR 2020 masterclass presentation.

The CP-SAT-LP solver architecture


Portfolio of Search heuristics and Large Neighborhood Search
Constraint Propagation
Simplex, cuts, MIP heuristics
Integer variable encoding
SAT engine

A complete reimplementation of a Constraint Programming Solver on top of a SAT engine.

A rich modeling layer that offers linear arithmetic and more complex discrete optimization constraints (including scheduling and routing constraints).

Complemented by a Simplex that solves a (dynamic) linear relaxation of the problem.

Enriched by MIP techniques (presolve, cuts, branching heuristics) extended to cover the Constraint Programming layer.

Encoding Integer Variables on top of SAT

Order encoding is dynamic.

- The solver creates integer literals (x <= 5) on demand, and uses them everywhere
- Creating new Boolean literals attached to integer literals is only done when branching. Conflict resolution expands integer literals until it falls back on existing Boolean literals

Value encoding is static

- (x == value) Boolean variables are created during presolve
- The intuition is that in the majority of cases, if you need value encoding, you should not use integer variables

Most CP constraints are expanded

We expand:

Element, Table (positive and negative), Automaton (regular), Inverse, Alldiff (when it is close to a permutation, or small enough), Reservoir

We keep:

- Boolean constraints (and, or, xor)
- Linear constraints (with half-reification aka indicator constraints)
- min, max, product, division, modulo with fixed mod argument
- Alldiff (all variables should take a different value) when not a permutation
- Circuit (hamiltonian path)
- Scheduling constraint (no_overlap, cumulative, no_overlap_2d)

The Best of all Worlds

(Max)SAT:

- Core based search
- Model reductions
- Clause Learning

Constraint Programming:

- Rich modeling layer (structure is not lost in the solver)
- Advanced deduction algorithms

Linear Integer Programming:

- Linear Relaxation + Cuts
- Presolve

Meta-heuristics

- Fast solution improvement techniques

Why this is (NP) hard

1. This is combinatorial optimization.
2. Lots of knowledge is hidden (commercial), forgotten (coder has left the team), or lost (team was disbanded).
3. Finding good solutions is mostly luck. Good solvers have 100s of dedicated heuristics.
4. Proving optimality, or finding better lower bounds of the objective function uses hard math, and complex combinations of scattered information.
5. Solvers are the results of large efforts (100s of work-years for CPLEX, Gurobi, CP Optimizer).
6. One consequence is that research is slow/rare as the frontier is very far.
7. The nature of clause learning makes debugging difficult.

Where we are

MaxSAT: Competitive robust solver. We do not participate in the challenge.
Constraint Programming: We won all gold medals in the last 5 years of the minizinc challenge.
Linear Integer Programming: Only solves pure ILP problems. Better than all open source solvers, closing in on the best commercial solvers, not competitive on average. Still CP-SAT can be good enough and it offers different performance characteristics.
Scheduling: competitive or better than state of the art on academic benchmarks. Better than commercial solvers on small to medium scheduling problems, misses heuristics for large instances.
Routing: Promising, hard, researchy, would benefit the routing library in OR-Tools.

MIP on SAT

Linear programming constraint

Bounded integer variables: $X_i \in [lb_i, ub_i] \quad X_i \in \mathbb{Z}$

Integer linear objective: minimize $\sum_i obj_i X_i \quad obj_i \in \mathbb{Z}$

Pool of linear constraints: $[lhs <=] \sum_i coeff_i X_i \ [<= rhs] \ coeff_i \in \mathbb{Z}$

Compared to a MIP solver:

- Everything is integer (int64) and bounded.
- “No overflow” precondition: min/max constraint activity fit on int64.
- “Optional”, all constraints in the pool already in the CP-SAT engine.

Initial pool content: Linear relaxation of CP model

Controlled by a parameter **linearization_level:{0,1,2,3}**

- linear constraint [level 1]
- Encoding of Boolean constraints [level 2]
- Half-reified linear constraint encoded with big M [level 2]
- Integer encoding: i.e. Integer = ∑ Boolean * value [level 2]
- …
- Implication graph & at most one: max clique [level 2]
- circuit constraint ⇒ dynamic subtour elimination cut [level 2]
- Product: McCormick inequalities [level 2]
- Scheduling relaxation [level 2]

Cuts

When to run the LP solver ?

Run at every node! But:

- Lowest propagator priority
- Run only for a short number of simplex iterations:
  LP dual simplex will continue at next node thanks to incrementality.
- Starts with empty LP and add constraints from the pool in small batches.
- Periodically remove constraints that have been BASIC (i.e not used) in the last xx optimal solutions.

Mainly at “level zero”: We also generate cuts.

What is it used for ?

Bounds propagation (with SAT explanations):

- LP infeasibility
- Objective lower bound
- Variables lower/upper bound (reduced cost fixing)

Heuristics:

- Branching: LP optimal value & reduced costs can be used.
- Large neighborhood search (i.e. RINS).

Our propagation is EXACT !

Even though LP solver is inexact, we just use its output as a “hint”.

- We use only int64 arithmetic (kind of adaptable fixed precision).
- No epsilon! But have to deal with integer overflow (not too hard).
- **vs MIP**: simplify the code & complexity a lot.

First ingredient required for this:

- Our base LP only use integer coefficients (we scale them for the LP solver).
- Any cuts we generate are also pure integer (not too hard actually).

Ingredient 2: Computing new valid linear constraints

Given any set of floating point constraint multipliers λ_i

Scale them to integer `M`$_i$ `= round(s * λ`$_i$`)` with a factor `s` (we use a power of 2) as large as possible and compute exactly:

```
    Sum_{positive M_i}  M_i * (constraint_i <= rhs_i)
  + Sum_{negative M_i}  M_i * (constraint_i >= lhs_i)
  = new_linear_terms <= new_rhs
```

Details:

- Set to zero `M`$_i$ with wrong sign (we usually only combine tight LP constraints)
- Make sure there is no integer overflow (could use int128 if needed)
- Can divide at the end using MIR cut like technique to reduce magnitude

Propagating the LP

LP infeasible: take dual ray as constraint multipliers,
new constraint will give rise to a conflict (i.e. implied lower bound > rhs).

LP Optimal (or dual feasible): take dual LP values as constraint multipliers
+ add objective (scaled by same integer factor s):

```
    new_linear_terms + s * objective_linear_term <= new_rhs + s * objective_var
```

Normal propagation of this new constraint:

- Push LP lower bound of objective var
- Push upper (or lower) bound of variables (i.e. reduced cost fixing !)

Explanation of a linear constraint

Ex: X + Y + 3*Z <= 6 with (X>=1), (Y>=1), (Z>=0)

Compute: slack = rhs - implied_lower_bound = 6 - 2 = 4
Upper bound of Z: LB_of_Z + ⌊slack / 3⌋ = 1
Basic explanation: lower bound of all other variables (X>=1)(Y>=1)
But slack of 3 is enough: relaxation “budget” of 1

2 options: (X>=0)(Y>=1) or (X>=1)(Y>=0)

we can choose during conflict resolution !

Generic MIP Cuts - Still exact here

Constraint aggregation:

- Chvatal Gomory (get multipliers λ_i from $\mathbf{B}^{-1}.\mathbf{e}_j$)
- MIR heuristic (combine small number of constraints)
- Zero half cuts (heuristic mod 2)

Cut generation on the new constraint:
Various integer rounding heuristic (MIR), knapsack, and cover heuristics

Basic cut management:
Sort by efficacy and orthogonality, keep number of cuts in check.

How to improve CP-SAT-LP?

Google Research

Controversial statements

How did we get there ? How do we continue improving performance ?

In other words, how do we improve a CP solver ?

Answer: by looking at MIP solvers.

This does **not** literally mean becoming a MIP solver.

It means:

- **Focusing** on benchmarks
- Integrating ideas from **all communities**
- **Presolve** is critical
- **Linear relaxations** and **cuts** help

CP-SAT-LP is a CP solver on top of a SAT engine with a big emphasis on the linear sub-model.

LP cuts on CP constraints

Take the edge-finder on a no-overlap constraint. It can propagate
    intervals $\{i_1, .., i_n\}$ push interval i_0

This is part of the CP propagation algorithm. Bounds are passed on to the LP relaxation

But the LP values can violate basic CP constraints: LP-intervals overlap, optional intervals could push the 'other' interval more

This raises the question of which constraints to communicate between the CP and the LP engines

Flexible Job-Shop Scheduling

M machines, N jobs (each an ordered list of tasks t_1, … t_k).

Each task can run on subset of the M machines (possibly with different durations).

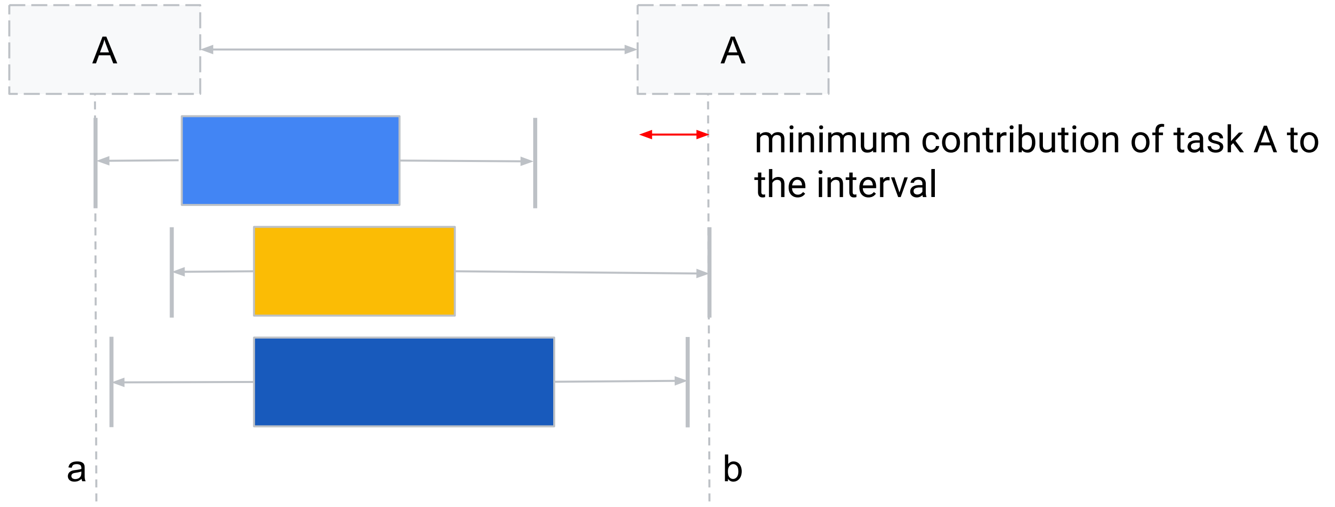
Goal is to minimize makespan.

Model uses optional intervals and one no_overlap constraint per machine.
LP relaxation:
  For each task :   ∑ presence_literal = 1
  For each machine: ∑ presence_literal * interval_size <= span
 ...
8 threads, 15 min, from 205 to 246 proven out of 313 “classical” instances.

More Scheduling Cuts (extended energetic relaxation)

Lifting energetic cuts

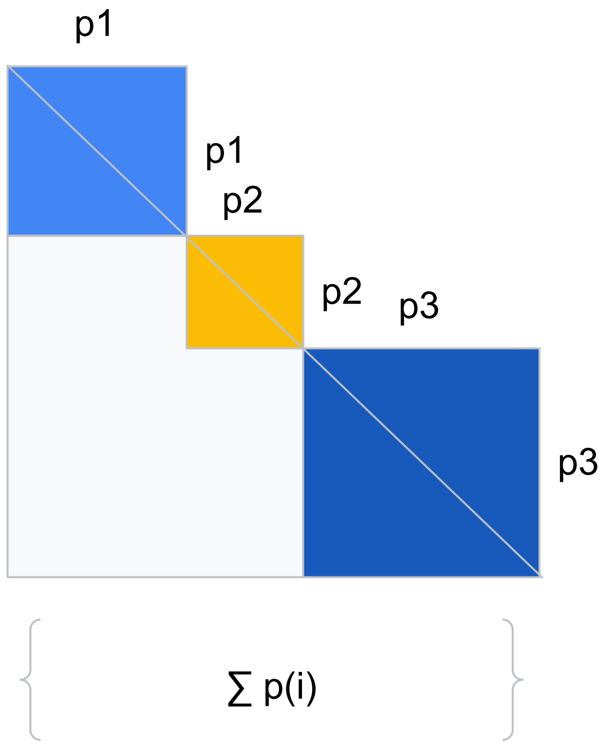
- For a time window [a, b], the basic cut takes all intervals contained in [a, b]
- Lifting adds contribution from intervals that must intersect with [a, b]



Completion Time Cuts

Smith rule: sum(pi * ei) >= ½(sum(pi^2) + sum(pi)^2)

- This cut pushes tasks apart
- Can be adapted to cumulative by dividing pi by the capacity max
- Can be adapted to no_overlap_2d with the same idea
- Can be lifted by incorporating intervals that must intersect with proper scaling



Google

Dealing with Energy

In the PSPLIB, the duration and the demand are fixed per recipe.

Thus the energy of a demand can be linearized as
*sum (recipe_i_bool_var * fixed_energy_i)*

This structure is discovered by inspecting the model.

This is very effective on the PSPLIB as the energetic cuts take into account the Boolean variables of the selected recipes.

This is also useful for 2d packing with rotating boxes (energy is constant) or non fixed energy.

Conclusion

Conclusion

- **CP-SAT-LP** is a very good CP solver 😀
- In CP-SAT-LP, the **linear relaxation** is on by default.
- **Clause Learning** is critical as it offsets the cost of running the simplex.
- It implements the **Scheduling on MIP** literature from the 80s, 90s and subsequent improvements.
- It could incorporate **different** relaxation/specialized (expensive) propagation algorithms, provided they can be **explained**, or provided they can be useful at the root node.

What's next?

- Reduce encoding size
- Incorporate cost management techniques from CSP solvers
- MIP improvements (cuts, presolve)
- Improved scheduling (branching heuristics, LNS, propagation)
- Incremental violation local search on non linear constraints
- Non violation based Local Search?
- Progress on Routing (branching heuristics, LNS, propagation)

Thank you