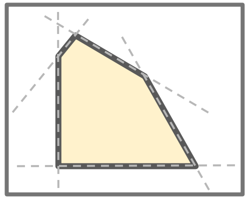
OR-Tools and CP-SAT

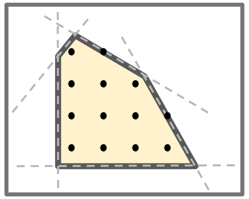
Laurent Perron (lperron@google.com)

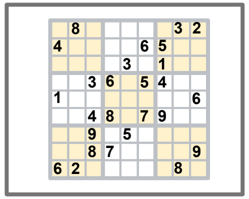
Operations Research @ Google

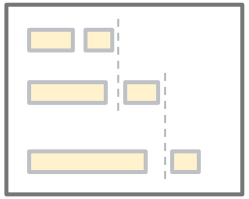
- The Operations Research (OR) team is mostly based in Paris and Cambridge (US)
- Started ~17 years ago, currently 40+ people
- Mission: **Optimize Google**
    - Internal consulting: build and help build optimization applications
    - Tools: develop core optimization algorithms
- Other software engineers with OR backgrounds are distributed throughout the company

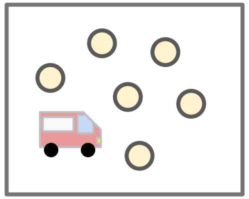
Combinatorial optimization

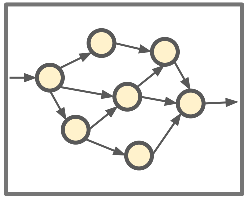




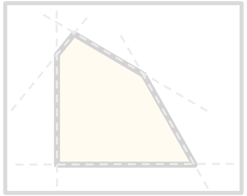
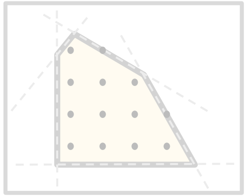
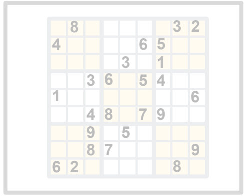
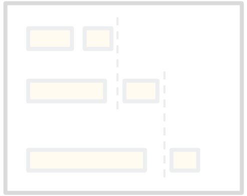
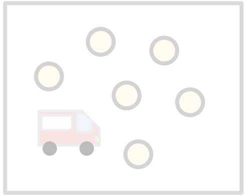
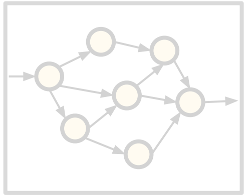






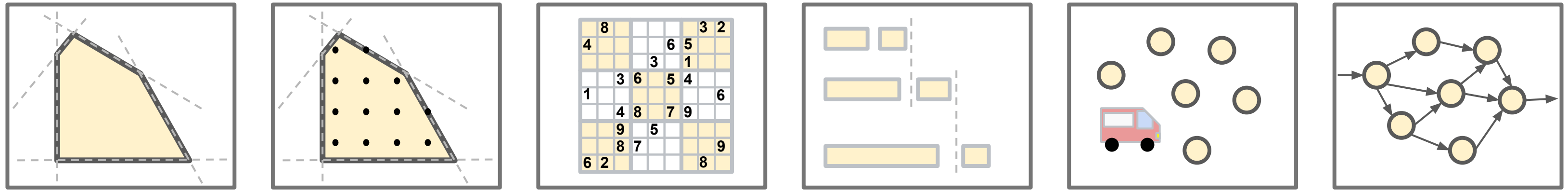


Solvers

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)

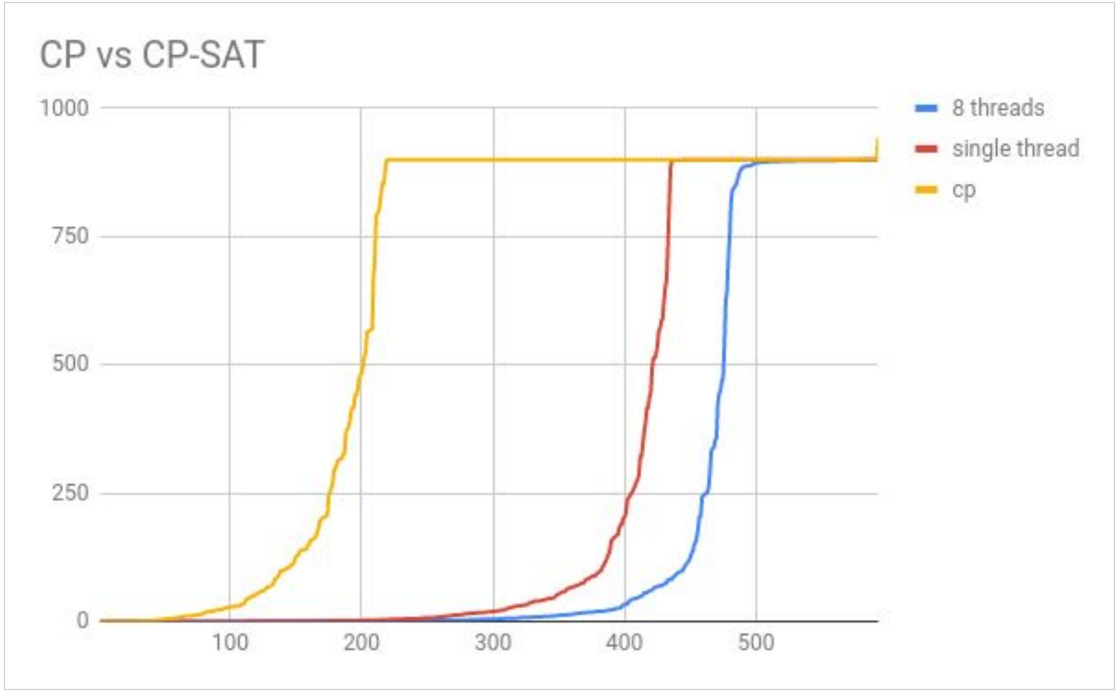
User profiles:

- 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

What is CP-SAT?

A Breakthrough: CP vs CP-SAT

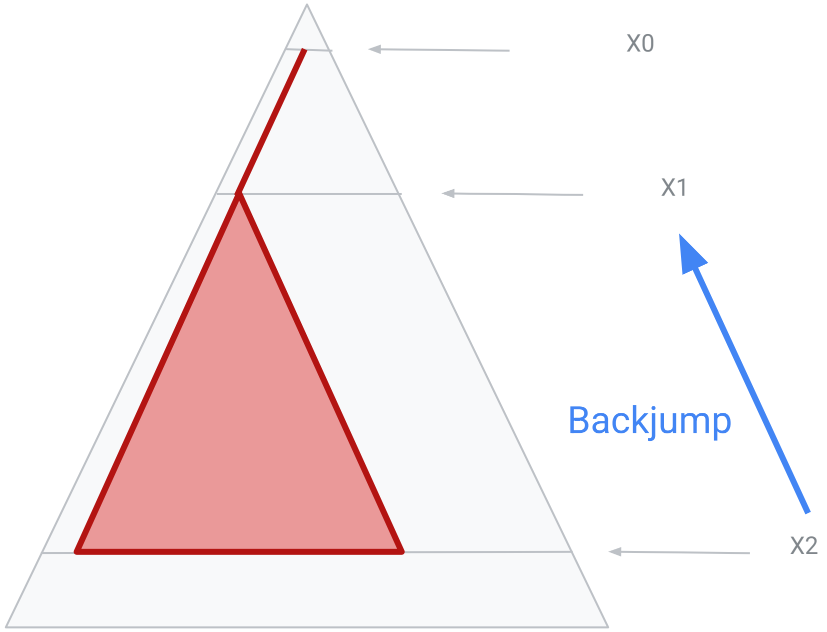


Google

Proprietary + Confidential

SAT in a nutshell

- X0, X1, X2 cannot be true at the same time
- half of the branches out of the red triangle will fail.
- SAT solvers will learn the nogood or(not X0, not X1, not X2), backjump to the assignment of X1, add the nogood, and kill half of the red subtree
- bonus point: X0, X1, X2 are important variables for the search heuristic



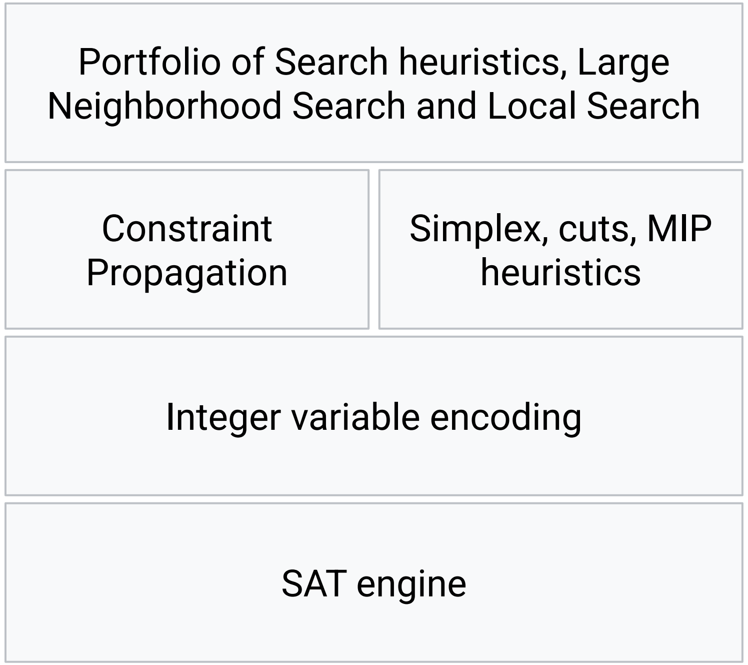
Google Cloud Google

CP-SAT 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** 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.

Proprietary + Confidential

The CP-SAT[-LP] solver architecture


Portfolio of Search heuristics, Large Neighborhood Search and Local 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.

CP-SAT workflow

1. Validate model, check for overflow
2. Presolve
3. Expand complex constraints
4. Presolve again
5. Analyse the model: probing, symmetry detection
6. One more round of presolve
7. Load the model into the solver
8. Solve
9. Postsolve
10. Check solution

Proprietary + Confidential

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

Other encoding are possibles (log, adaptive) but require significant investment

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)
- product, division, modulo with fixed mod argument
- Alldiff (all variables should take a different value) when not a permutation
- Circuit (hamiltonian path) and Multiple Circuit (VRP constraint)
- Scheduling constraint (no_overlap, cumulative, no_overlap_2d)
- Min and Max may be removed

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

- Large Neighborhood Search
- Violation based Local Search

Proprietary + Confidential

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.

Proprietary + Confidential

Where we are

MaxSAT: Competitive robust solver. We do not participate in the challenge. State of the art results with parallelism on hard unsat problems.

Constraint Programming: We won all gold medals in the last 6 years of the minizinc challenge.

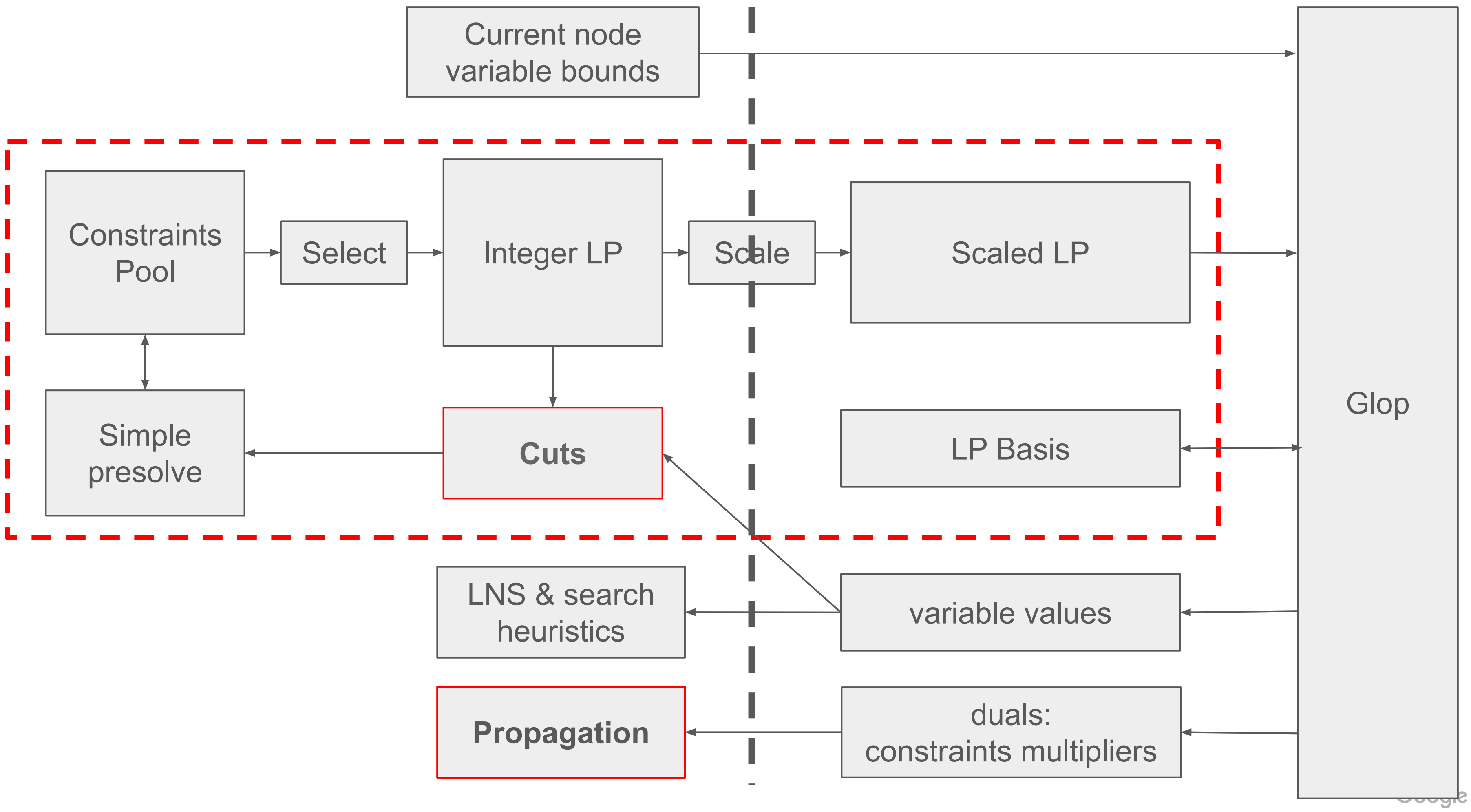
Linear Integer Programming: Only solves pure ILP problems. Better than all open source solvers, closing in on the best commercial solvers, not better on average. Still CP-SAT can be good enough and often competitive.

Scheduling: competitive or better than state of the art on academic benchmarks. Better than commercial solvers on small to medium scheduling problems, missing heuristics for large instances.

Routing: Promising, hard, researchy, would benefit the routing library in OR-Tools.

Proprietary + Confidential

MIP on SAT


Current node variable bounds
Constraints Pool
Select
Integer LP
Scale
Scaled LP
Glop
Simple presolve
Cuts
LP Basis
LNS & search heuristics
variable values
Propagation
duals: constraints multipliers
Google

CP-SAT is an EXACT solver !

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

- We use only int64/int128 arithmetic.
- No epsilon! But have to deal with integer overflow.
- **vs MIP**: simplify the code & complexity a lot.

And for the cuts, we compute everything with integers.

Exact LP propagation

Ingredient 1 : Integer LP

Bounded integer variables: $X_i \in [lb_i, ub_i]$ $X_i \in$ **int64**

Integer linear objective: minimize $\sum_i obj_i X_i$ $obj_i \in$ **int64**
valid integer linear constraints (initial linearization or cuts) :

$$lhs <= \sum_i coeff_i X_i <= rhs \qquad lhs, rhs, coeff_i \in \textbf{int64}$$

Compared to a MIP solver:

- Everything is integer (int64) and bounded.
- integer-overflow precondition: min/max constraint activity fit on int64.
- The constraints do not need to describe the full problem.

Ingredient 2 : Linear combination of constraints

Given any set of floating point constraint multipliers λ_i

Scale them to integer $M_i = \text{round}(s * \lambda_i)$ with a factor s (we use a power of 2) as large as possible and compute exactly:

$$
\begin{aligned}
&\textstyle\sum_{\{\text{positive } Mi\}} \quad M_i * (\text{constraint}_i \le \text{rhs}_i) \\
+ &\textstyle\sum_{\{\text{negative } Mi\}} \quad M_i * (\text{constraint}_i \ge \text{lhs}_i)
\end{aligned}
$$

= new_linear_terms <= new_rhs

Details:

- We compute the new_rhs using int128
- s chosen so that all other coefficients fit on an int64

Google

Ingredient 3: Propagating an integer linear constraint

Canonicalize to: $\sum_i \texttt{coeff64}_i\ X'_i \texttt{ <= rhs128}$ $\qquad \texttt{coeff64}_i > 0$

Compute: $\texttt{min\_activity128} = \sum_i \texttt{coeff64}_i\ \texttt{lb64}(X'_i)$

$\texttt{slack128 = rhs128 - min\_activity128}$

`slack128 < 0:` conflict!
`slack128 >= int64max:` no propagation.

Otherwise, $\forall i$, $\texttt{new\_ub64}(X'_i) = \texttt{lb64}(X'_i) + \lfloor \texttt{slack64} / \texttt{coeff64}_i \rfloor$

Propagating dual-infeasible LP

- Take dual ray (aka infeasibility proof) as constraint multipliers
- New constraint should give rise to a conflict (i.e. min_activity > rhs).

Note: Even if not conflicting, it will likely have small slack (rhs - min_activity).

Propagating dual-feasible LP

Take negated dual LP values (scaled by s) as constraint multipliers:

```
              new_linear_terms <= new_rhs                  [from multipliers]
      s * objective_linear_term <= s * objective_var    [objective definition]

   terms  + s * minus_objective_var  <=  new_rhs
```

Normal propagation of this new constraint:

- Push LP lower bound of objective var
- Or is infeasible if new objective lb > best_known_solution_objective.
- Push upper (or lower) bound of variables (i.e. reduced cost fixing !)

Generic LP Cuts

Goal

Given:

- Integer LP
- Current LP solution for each variables

Derive a new and valid integer linear constraint (linear terms <= rhs) that is

- violated by the current LP solution (LP solution activity > rhs)
- Has good efficacy = violation / $\|\text{constraint}\|_2$

Note that only generic MIP cut here: No TSP, scheduling cuts, etc…

Example: clique cut

X, Y, Z integer variable in [0,1], current LP solution 0.5 for each.

Constraints:

$$X + Y <= 1,$$
$$X + Z <= 1,$$
$$Y + Z <= 1$$

This could be the optimal to maximize X + Y + Z for instance.

But, given the integrality constraint (ignored by the LP), we can derive
X + Y + Z <= 1, this is a violated cut.

General MIP cut framework

Almost all cuts (gomory, MIR, zero half, cover, flow, …) follow this:

- Aggregate many constraints from the integer lp into one (terms <= rhs)
- From such single constraint, rewrite it slightly with complementation, implied bounds, and using positive variables.
- Apply a super-additive function f() to get the cut.
- Rewrite everything in term of the original variables.

Aggregation

Same as for explanation, take linear-combination of constraints, but use slacks

$\text{lhs} <= \sum_i \text{coeff}_i X_i <= \text{rhs} \quad => \quad \sum_i \text{coeff}_i X_i - S = 0$

new slack integer variable $S \in [\text{lhs}, \text{rhs}]$

Why slacks?

- Final constraint is always tight for LP
- After all steps, it is possible slack still there, it will be substituted back, and that can lead to stronger cut.

Different aggregation heuristics

- No aggregations ! try each cut heuristic on row or -row.
- **MIR** heuristic: combine small number (<= 6) of constraints to eliminate fractional variables.
- **Chvatal-Gomory** (get multipliers λ_i from $\mathbf{B}^{-1}.\mathbf{e}_j$)
  Should lead to single variables not at its bound in aggregated equation!
- **Zero-half** cuts (only use +1/-1 multipliers, heuristic mod 2)
  Idea is to get odd rhs, but even coeff for important variables.
- **Clique** (Weighted Bron Kerbosch max-clique enumeration)

Linear equality rewriting (heuristics too)

Propagate/presolve: minor impact but still useful.

rewrite using positive variables: $X' = X - lb$ $X' \in [0, range = ub - lb]$

Maybe complement some variables: $X = range - X^c$

Maybe use some implied bounds (Bool => X >= v):
$X = v * B + (X - v * B) = v * B + S,\ S \in [0, range]$
This is especially powerful if B already in the constraint!

Make term more “integral”: $1 * X[0, 10'000] \rightarrow 100 * Y[0, 100] \rightarrow 10'000 * Z[0, 1]$

Super-additive function f()

- $f: \mathbb{Z} \to \mathbb{Z}$
- $f(x) + f(y) <= f(x + y)$

This is nice, because:

$rhs128 = \sum_i coeff64_i X_i$, X_i positive

$f(rhs128) = f(\sum_i coeff64_i X_i)$ [>= works too]
$f(rhs128) >= \sum_i f(coeff64_i X_i)$ [super-additivity]
$f(rhs128) >= \sum_i f(coeff64_i) X_i$ [X_i positive integer]

This is the step that goes from “tight constraint” to “violated constraint”

Clique exemple revisited (a bit artificial)

- Recall $X + Y \le 1$,
  $X + Z \le 1$,
  $Y + Z \le 1$
- We can sum them all: $2X + 2Y + 2Z \le 3$
- And apply $f(x) = \lfloor x / 2 \rfloor$ (this is super-additive)
- We get $X + Y + Z \le 1$

Note that this is the same f() used by zero-half cuts.

Cover cut (or Knapsack cut) example

$6X + 4Y + 10Z <= 9$ (X=1.0, Y=0.5, Z=0.2, all in [0,1])

X and Y form a “**cover**” (i.e. $6 + 4 > 9$)

Lets **complement** the cover: $-6X^c -4Y^c + 10Z <= -1$

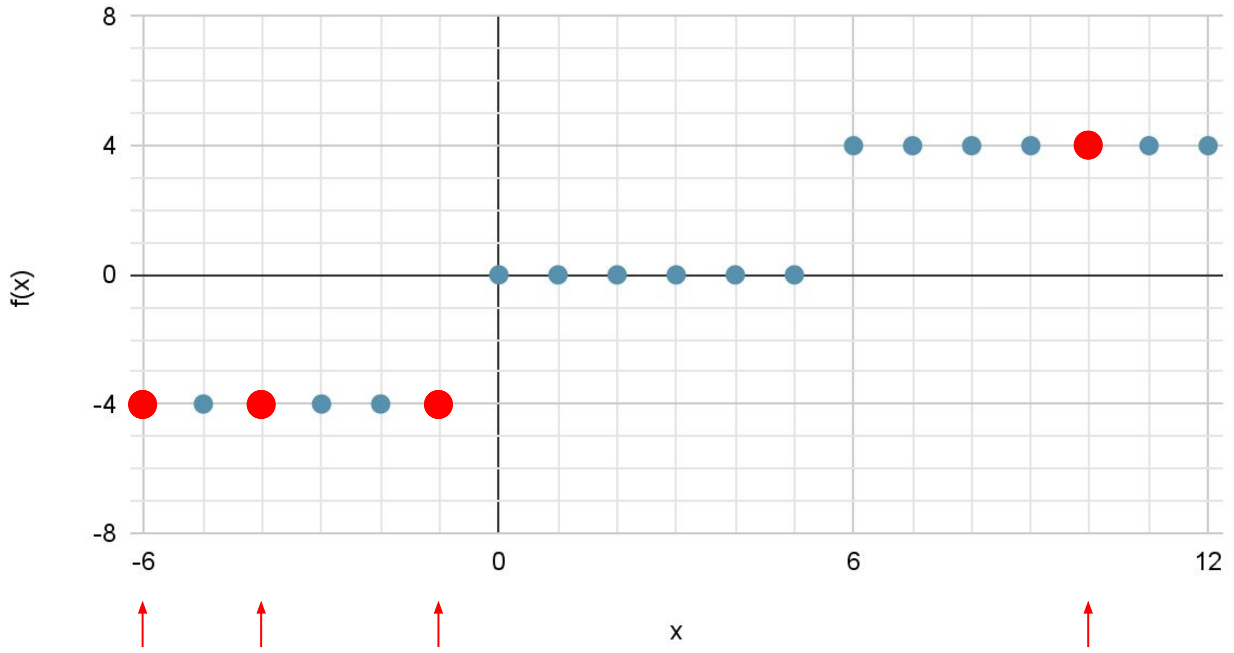
Apply $f(x) = \lfloor x/6 \rfloor$

we get: $-X^c -Y^c + Z <= -1$ (note the lifting of Z)

substituting back: $X + Y + Z <= 1$ (violation = 0.7 !)

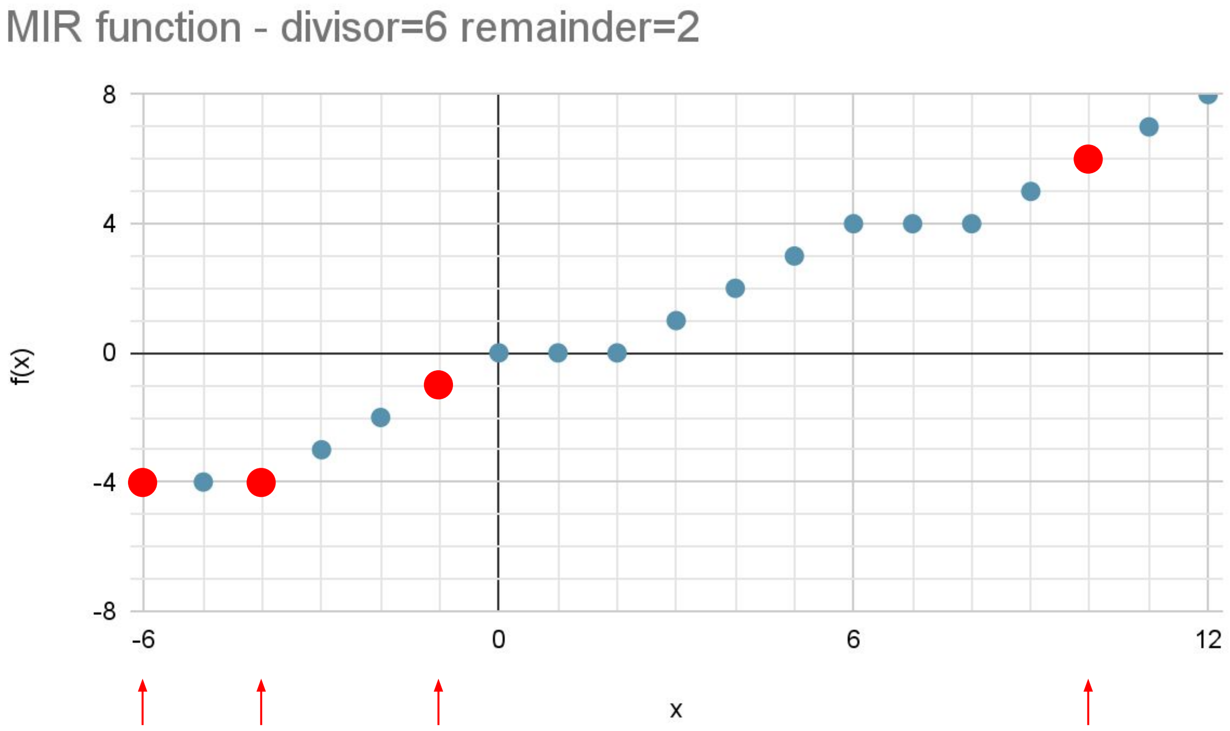
Various choices for f()

Division / 6 (rescaled)



Google

Various choices for f()



Google

MIR cuts

For **cover**, we complement so that all coeff positive.

For **MIR**, we complement so that lp value of each term is smallest.

Then, try to take as divisor for f(), coefficients of terms with large lp values

We define **remainder = rhs % divisor** and **scale = divisor - remainder**

We use MIR super-additive function:

$$f(x) = \text{scale} * \lfloor x/\text{div} \rfloor + \max(0, (x \,\%\, \text{div}) - \text{remainder})$$

Various other tricks

- When choosing f() we want final coeff to be small. So we use slightly more complex MIR function so that f(divisor) = scale stay small.
- Once f() chosen, we can try other possible complementation
- We can drop small terms rather than applying f() to them.
- We can also try to use different implied bounds once f() is chosen.
- …

Probably still a lot of room for improvement in that code !!
And still other heuristic to write (path mixing cuts)

Real example from log on beasleyC2.mps

```
INPUT:
coeff=    1454428938435252 lp=0.888889 range=9
coeff=   11635431507481974 lp=0.888889 range=9
coeff=    1454428938435255 lp=0.496158 range=1
coeff=    1454428938435254 lp=0.503842 range=1
coeff=    1454428938435254 lp=0.496158 range=1
coeff=    1454428938435255 lp=0.547352 range=1
coeff=    1454428938435254 lp=0.315313 range=1
coeff=    1454428938435254 lp=0.728197 range=1
coeff=  -11635431507481978 lp=0.111111 range=1
coeff=   -93083452059856042 lp=0.111111 range=1
coeff=-1060278696119297370 lp=0.098765 range=1
coeff=    2908857876870509 lp=0.95649  range=1
…
coeff=  -34906294522445990 lp=0        range=9
coeff=                 -16 lp=0        range=9
…
<=      -97446738875161748
```

```
f():   Div      1060278696119297370
       Rem       962831957244135622
       scale                      8


CUT:   coeff=-1 lp=0.111111 range=1
       coeff=-7 lp=0.111111 range=1
       coeff=-8 lp=0.098765 range=1
       …
       coeff=-3 lp=0        range=9
       coeff=-1 lp=0        range=9
       …
       <= -8
```

RLT cuts

Reformulation Linearization Technique (RLT) cuts

Given 3 Booleans and linear equation X + Y + Z >= 1.

This can be tight with say X=0.7, Y=0.1, Z=0.2.

But using integrality, we have (1-X) * (1-Y) <= Z (quadratic relation),
which is not satisfied ! (0.3 * 0.9 = 0.27 vs 0.2)

Similarly, from (1 - X) * Y = (1-X) - (1-X) * (1-Y) >= (1 - X) - Z
And this is also violated ! (0.3 * 0.1 vs   0.1)

Reformulation Linearization Technique (RLT) cuts

Start by base equation: $\sum_i \text{coeff}_i X_i = \text{rhs}$, everything positive.

- multiply by Y to get a quadratic equation:
  $\sum_i \text{coeff64}_i Y * X_i = \text{rhs} * Y$
- bound each quadratic terms with a linearization using one of:
  - $Y * X >= 0$ (Drop)
  - $Y * Y >= Y$ (Square)
  - $Y * X >= Y - (1 - X)$ (McCormick, normal linear relaxation)
  - $Y * X >= Y - U$ (We need relation $Y * (1 - X) <= U$ from $X + (1 - Y) + U >= 1$)
- Note that we can also complement variable before bounding !
- Hopefully, this lead to violated cut (We only try “interesting” Y).

CP-SAT and Scheduling

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

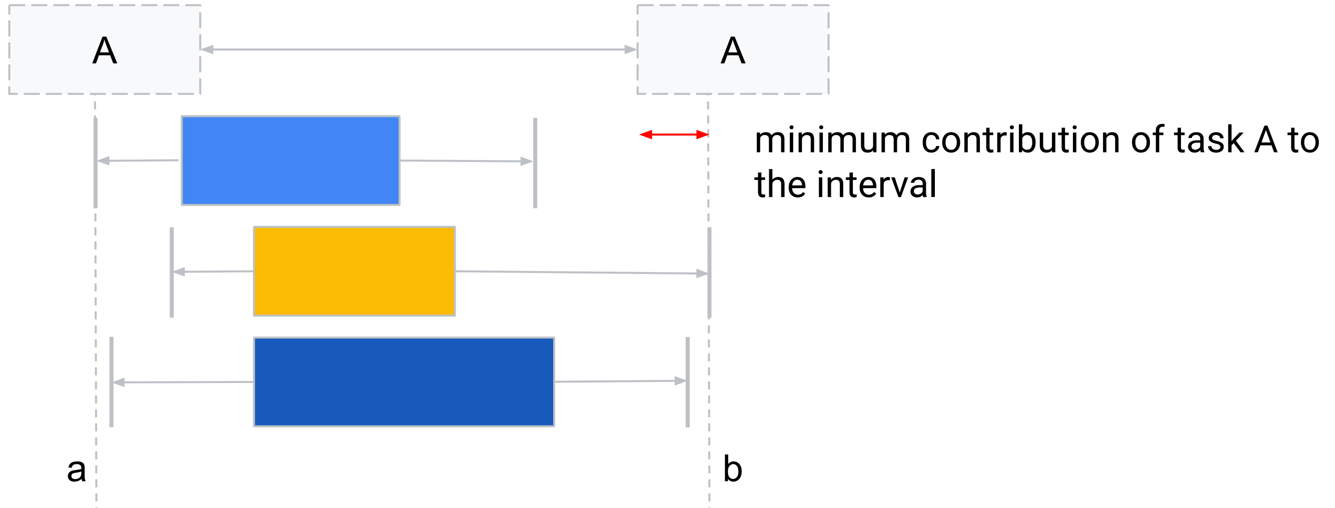
Flexible Job-Shop Scheduling

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]

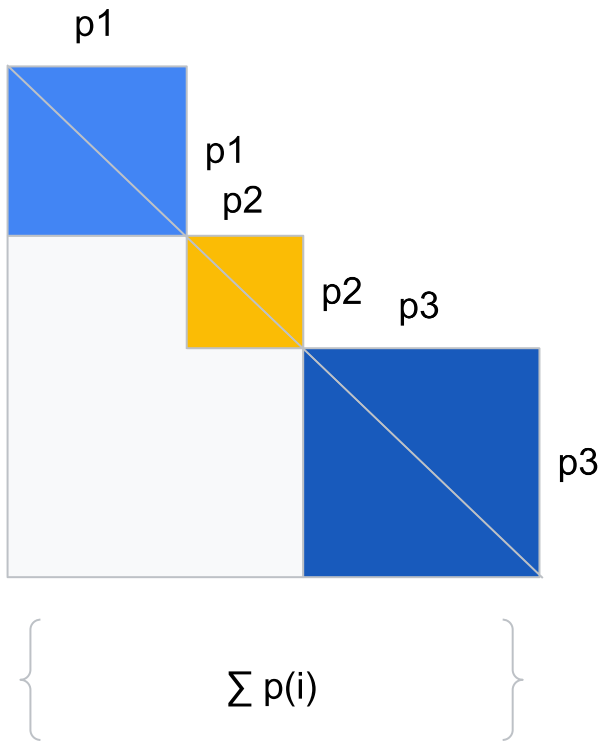


Google

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



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.

Stronger Scheduling Propagation

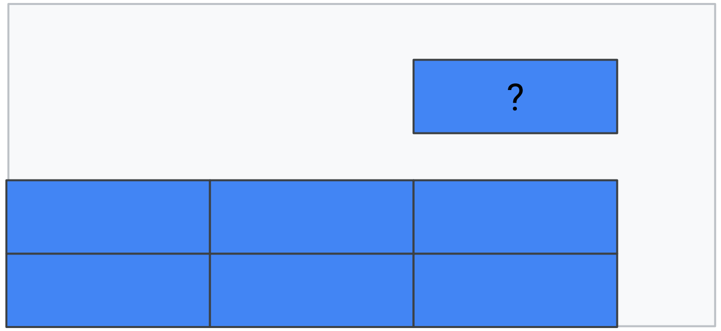
- 2 k + 1 tasks of duration p
- cumulative of horizon (k + ½) * p
- minimize capacity used

Pigeon hole problem

Mitigation ideas:

1. divide by gcd
2. subset-sum
3. use MIR like super-additive techniques

applied to linear relaxation, cuts or propagation?



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, modeling)
- Non violation based Local Search?
- Progress on Routing (branching heuristics, LNS, propagation)

Thank you