DEVELOPING OPTIMAL SOLUTIONS FOR ORGANIZATIONAL AND BUSINESS NEEDS USING OR (OPERATIONS RESEARCH) AND AI (ARTIFICIAL INTELLIGENCE)

DAY 2: Modeling and solving using Google's OR-Tools CP-SAT solver

Prof. Christos G. Gkogkos

University of Ioannina, Dept. of Informatics and Telecommunications

https://github.com/chgogos/INTRO_OR_AI

INTRODUCTION

- Many **theoretical proven hard optimization problems** can be solved in practice
- Two of the most successful approaches are MIP (Mixed Integer Programing) and CP (Constraint Programming)
  - MIP Solvers*
    - Commercial: Gurobi, IBM ILOG CPLEX
    - Open Source: SCIP, HIGHS
  - CP Solvers*
    - Commercial: IBM ILOG CP-Optimizer
    - Open Source: Google's OR-Tools CP-SAT (hybrid of a CP solver and a SAT solver)

*non-exhaustive list

THE SAT PROBLEM

- The boolean SATisfiability problem is the problem of determining if there exists an assignment of True/False values to variables that makes a given boolean formula to evaluate to True
- Why is SAT important?
  - Many practical problems (scheduling, hardware verification, cryptanalysis, etc.) can be reduced to SAT
  - Powerful SAT solvers exist that can solve instances with millions of variables

Formal Definition of SAT

Let ϕ be a Boolean formula over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, where each variable x_i can take a value in $\{\text{True}, \text{False}\}$. The Boolean formula ϕ is constructed using logical connectives: AND ($\wedge$), OR ($\vee$), and NOT ($\neg$).

Problem (SAT):
Given a Boolean formula ϕ, determine whether there exists a truth assignment

$$\sigma : X \to \{\text{True}, \text{False}\}$$

such that

$$\phi(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n)) = \text{True}.$$

If such an assignment σ exists, ϕ is said to be *satisfiable*; otherwise, it is *unsatisfiable*.

Example of SAT

$$\phi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$$

One possible satisfying assignment is:

$$x_1 = \text{True}, \quad x_2 = \text{False}, \quad x_3 = \text{True}$$

Substituting into ϕ:

$$(\text{True} \vee \neg\text{False}) \wedge (\text{False} \vee \text{True}) = (\text{True} \vee \text{True}) \wedge (\text{False} \vee \text{True}) = \text{True} \wedge \text{True} = \text{True}$$

Thus, the formula is *satisfiable*.

SAT SOLVERS

- Modern SAT solvers are highly optimized and use several techniques (e.g., backtracking, backjumping, avoidance of redundant search, heuristics, pruning)
  - DPLL algorithm (Davis-Putnam-Logemann-Loveland)
  - CDCL solvers (Conflict-Driven Clause Learning)

- SAT solvers*
  - MiniSAT
  - Kissat
  - Glucose
- OR-Tools CP-SAT uses CDCL solving technology and it extends it to handle integer variables and linear constraints directly

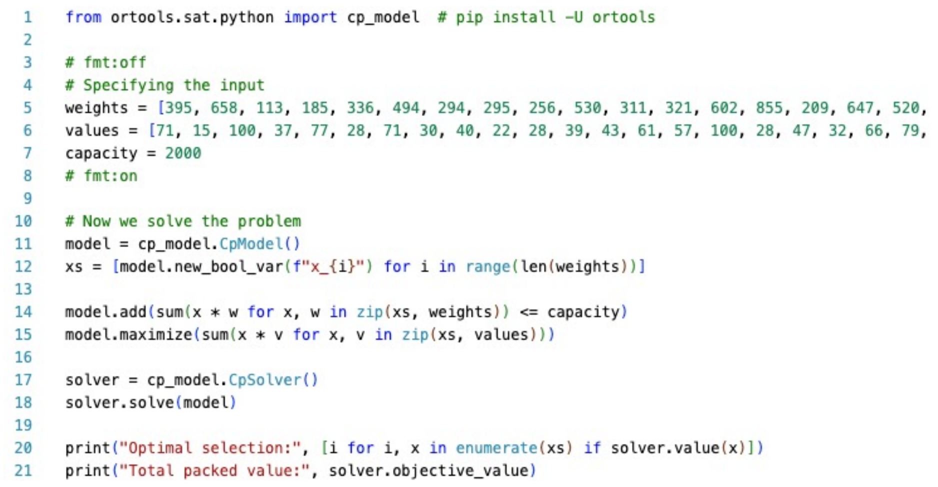
*non-exhaustive list

DECLARATIVE APPROACH OF SOLVING PROBLEMS

- **Declarative**: state what you want, not how to get it
- SQL is well-known example of a declarative language
- CP-SAT is not purely declarative, since how a problem is modeled impacts significantly the execution time needed to reach optimality or even get feasible solutions

CP-SAT: 0/1 KNAPSACK PROBLEM

- Full enumeration of all possible subsets of 100 items requires evaluation of $2^{100}=10^{30}$ possible solutions ➔ 31000 years of computation (10^{18} operations/second)
- CP-SAT finds a solution in ~0.01 seconds by making intelligent deductions and pruning of the search space
- CP-SAT has a high-level modeling interface that is richer than the corresponding MIP modeling interface (alllows easier modeling)



```python
from ortools.sat.python import cp_model  # pip install -U ortools

# fmt:off
# Specifying the input
weights = [395, 658, 113, 185, 336, 494, 294, 295, 256, 530, 311, 321, 602, 855, 209, 647, 520,
values = [71, 15, 100, 37, 77, 28, 71, 30, 40, 22, 28, 39, 43, 61, 57, 100, 28, 47, 32, 66, 79,
capacity = 2000
# fmt:on

# Now we solve the problem
model = cp_model.CpModel()
xs = [model.new_bool_var(f"x_{i}") for i in range(len(weights))]

model.add(sum(x * w for x, w in zip(xs, weights)) <= capacity)
model.maximize(sum(x * v for x, v in zip(xs, values)))

solver = cp_model.CpSolver()
solver.solve(model)

print("Optimal selection:", [i for i, x in enumerate(xs) if solver.value(x)])
print("Total packed value:", solver.objective_value)
```

https://d-krupke.github.io/cpsat-primer/00_intro.html

```
Optimal selection: [2, 14, 19, 20, 29, 33, 52, 53, 54, 58, 66, 72, 76, 77, 81, 86, 93, 94, 96]
Total packed value: 1161.0
```

CP-SAT INSTALLATION

- Using uv (recommended)

  See Day 1/slide 8

- Using pip

  ```
  $ pip3 install -U ortools
  ```

- Hardware requirements:
  - A standard laptop suffices (for experimentation and more)
  - Almost identical hardware requirements with Gurobi's, see:
    - https://support.gurobi.com/hc/en-us/articles/8172407217041-What-hardware-should-I-select-when-running-Gurobi
    - https://support.gurobi.com/hc/en-us/articles/28599065754001-How-many-cores-does-my-model-need
  - CP-SAT doesn't use GPUs

HOW CP-SAT WORKS?

- The design of CP-SAT embodies the concept behind the No-Free-Lunch (NFL) theorem:
  "No single optimization algorithm performs better than all others across every possible problem"
- CP-SAT is a portfolio solver based on a Lazy Clause Generation (LCG) Constraint Programming Solver
  - LCG = turns CP constraints into SAT clauses on the fly, then employs Conflict Driven Clause Learning (CDCL), a technique used in modern SAT solvers
- CP-SAT executes concurrently many and diverse algorithms that exchange information when:
  - A better solution is found
  - New tighter bounds emerge
- CP-SAT algorithms: **Lazy Clause Generation (LCG)**, Branch and Cut, Branch an Bound, Propagation Engines (AllDifferent, Cumulative, Circuit, …), Restart Policies, Variable and Value Selection Heuristics, presolve techniques, Relaxation Induced Neighborhood Search (RINS)

CP-SAT LIMITATIONS

- CP-SAT lacks support for continuous variables, workarounds may not be the best fit
- CP-SAT does not support iterative model building (e.g., when a callback is triggered to add more constraints)
- CP-SAT employs the Simplex algorithm and currently has no implementation of the Barrier algorithm (i.e., interior point) which is known to handle certain types of quadratic constraints better

BASIC MODELING WITH CP-SAT

- CP-SAT is equipped with a large set of constraints that reduce the need for modeling complex logic through linear constraints (as in MIP solvers): `all_different`, `add_abs_equality`, `add_multiplication_equality`, …
- CP-SAT operates differently from typical MIP solvers by relying less on linear relaxation and more on its underlying SAT-solver and CP propagators to efficiently manage logical constraints

CP-SAT: MODELING ELEMENTS (BASIC)

- **Variables**: `new_int_var, new_bool_var, new_constant, new_int_var_series, new_bool_var_series`
- **Custom Domain Variables**: `new_int_var_from_domain`
- **Objectives**: `minimize, maximize`
- **Linear Constraints**: `add, add_linear_constraint`
- **Logical Constraints (Propositional Logic)**: `add_implication, add_bool_or, add_at_least_one, add_at_most_one, add_exactly_one, add_bool_and, add_bool_xor`
- **Conditional Constraints (Reification)**: `only_enforce_if`
- **Absolute Values** and **Max/Min**: `add_min_equality, add_max_equality, add_abs_equality`
- **Multiplication, Division,** and **Modulo**: `add_modulo_equality, add_multiplication_equality, add_division_equality`
- **All Different**: `add_all_different`
- **Domains** and **Combinations**: `add_allowed_assignments, add_forbidden_assignments`
- **Array/Element Constraints**: `add_element, add_inverse`

CP-SAT: MODELING ELEMENTS (ADVANCED)

Constraints

- `add_circuit`
- `add_multiple_circuit`
- `add_automaton`
- `add_reservoir_constraint`
- `add_reservoir_constraint_with_active`
- `add_no_overlap`
- `add_no_overlap_2d`
- `add_cumulative`

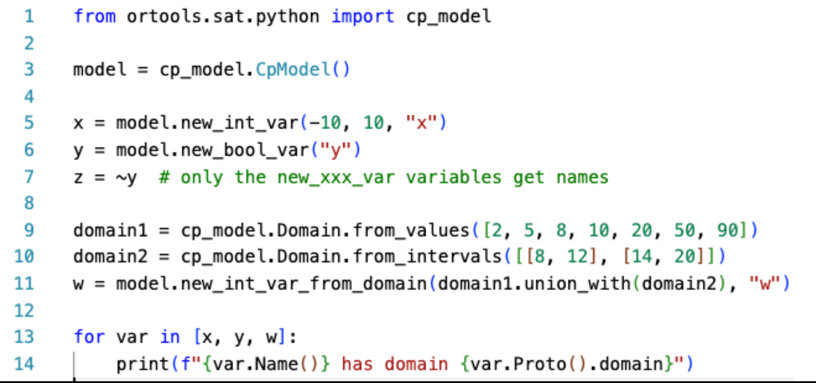
Variables

- `new_interval_var`
- `new_interval_var_series`
- `new_fixed_size_interval_var`
- `new_optional_interval_var`
- `new_optional_interval_var_series`
- `new_optional_fixed_size_interval_var`
- `new_optional_fixed_size_interval_var_series`

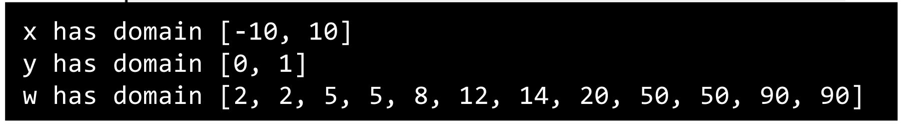
VARIABLES

- The two types of variables commonly used are **Boolean variables** and **Integer variables**
- For Integer variables lower and upper bounds must be specified (tip: keep the bounds tight)
- Continuous variables are not allowed – but this might not be a problem, why?
- Naming the variables can be helpful for debugging reasons

- Custom domain variables can also be used, e.g., {2, 5, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20, 50, 90}



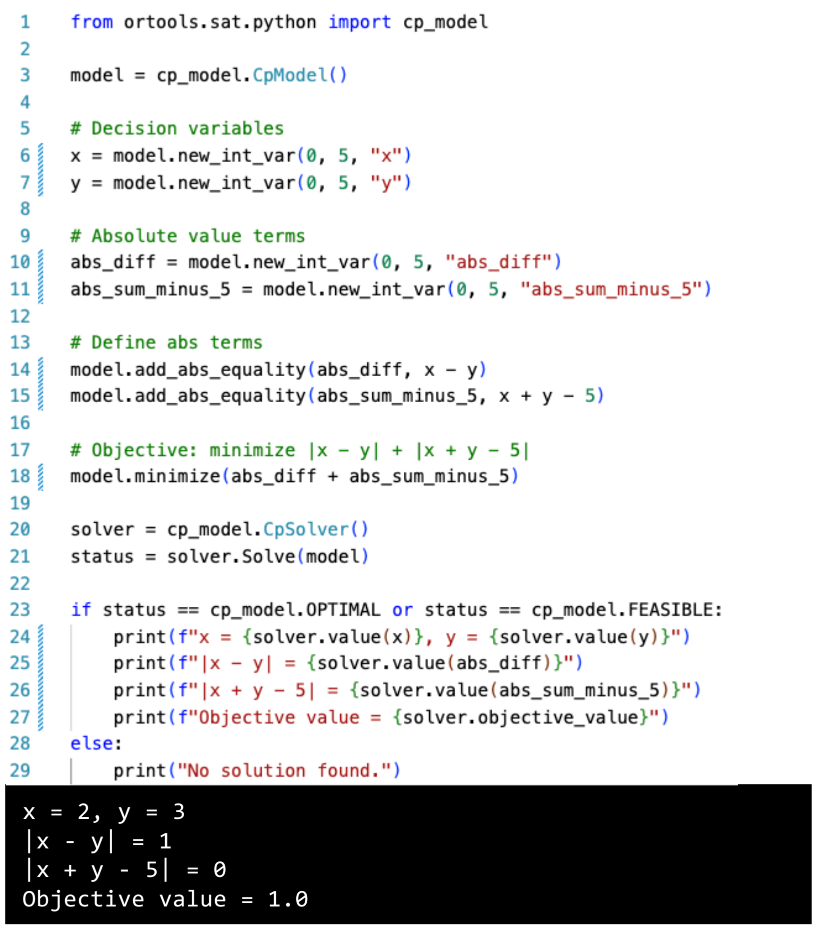
```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()

x = model.new_int_var(-10, 10, "x")
y = model.new_bool_var("y")
z = ~y  # only the new_xxx_var variables get names

domain1 = cp_model.Domain.from_values([2, 5, 8, 10, 20, 50, 90])
domain2 = cp_model.Domain.from_intervals([[8, 12], [14, 20]])
w = model.new_int_var_from_domain(domain1.union_with(domain2), "w")

for var in [x, y, w]:
    print(f"{var.Name()} has domain {var.Proto().domain}")
```



```
x has domain [-10, 10]
y has domain [0, 1]
w has domain [2, 2, 5, 5, 8, 12, 14, 20, 50, 50, 90, 90]
```

OBJECTIVES

- Sometimes finding a feasible solution suffices
- CP-SAT can also be used for minimizing or maximizing an objective function
- The objective function can be a linear expression of decision variables or **a more complex expression** that uses auxilliary variables

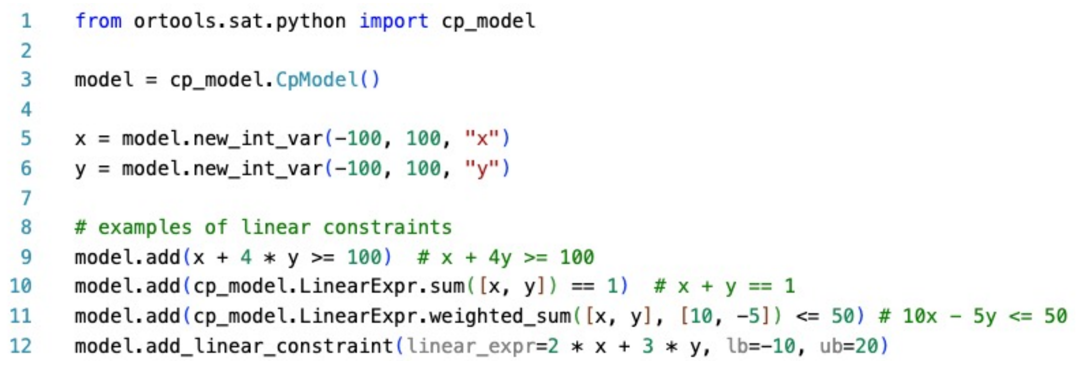


```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()

# Decision variables
x = model.new_int_var(0, 5, "x")
y = model.new_int_var(0, 5, "y")

# Absolute value terms
abs_diff = model.new_int_var(0, 5, "abs_diff")
abs_sum_minus_5 = model.new_int_var(0, 5, "abs_sum_minus_5")

# Define abs terms
model.add_abs_equality(abs_diff, x - y)
model.add_abs_equality(abs_sum_minus_5, x + y - 5)

# Objective: minimize |x - y| + |x + y - 5|
model.minimize(abs_diff + abs_sum_minus_5)

solver = cp_model.CpSolver()
status = solver.Solve(model)

if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
    print(f"x = {solver.value(x)}, y = {solver.value(y)}")
    print(f"|x - y| = {solver.value(abs_diff)}")
    print(f"|x + y - 5| = {solver.value(abs_sum_minus_5)}")
    print(f"Objective value = {solver.objective_value}")
else:
    print("No solution found.")
```

```
x = 2, y = 3
|x - y| = 1
|x + y - 5| = 0
Objective value = 1.0
```

LINEAR CONSTRAINTS

- Linear constraints: they involve only sums of multiplications of variables with constants
- Examples of linear constraints:
    - x + 4*y >= 100
    - x + y == 1
    - 10*x – 5*y <= 50
    - -10 <= 2*x + 3*y <= 20



```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()

x = model.new_int_var(-100, 100, "x")
y = model.new_int_var(-100, 100, "y")

# examples of linear constraints
model.add(x + 4 * y >= 100)  # x + 4y >= 100
model.add(cp_model.LinearExpr.sum([x, y]) == 1)  # x + y == 1
model.add(cp_model.LinearExpr.weighted_sum([x, y], [10, -5]) <= 50) # 10x - 5y <= 50
model.add_linear_constraint(linear_expr=2 * x + 3 * y, lb=-10, ub=20)
```

LOGICAL CONSTRAINTS (PROPOSITIONAL LOGIC)

- Logical constraints describe relationships between true or false statements using logical operators (i.e., AND, OR, XOR, NOT, IMPLICATION)
- IMPLICATION, used as A → B indicates that if A is true, B must also be true
- Examples of logical constraints assuming b1, b2, b3 are boolean variables:
  - `model.add_bool_or(b1, b2, b3) # b1 or b2 or b3 must be true`
  - `model.add_bool_and(b1, ~b2, ~b3) # b1 and not b2 and not b3 must all be true`
  - `model.add_bool_xor(b1, b2, b3)  # Returns true if an odd number of b1, b2, b3 are true`
  - `model.add_implication(b1, b2) # If b1 is true, then b2 must also be true`
- More logical constraints:
  - `model.add_exactly_one([b1, b2, b3])  # Exactly one of the variables must be true`
  - `model.add_at_most_one([b1, b2, b3])  # No more than one of the variables should be true`
  - `model.add_at_least_one([b1, b2, b3]) # Alternative to model.add_bool_or`

CONDITIONAL CONTRAINTS (REIFICATION)

- It might be desirable, when some conditions are met, to enforce certain constraints (e.g., if condition A is true, then constraint B should apply)
- In CP, reification associates a Boolean variable with the truthiness of a constraint
- Half reification:
  - if the Boolean variable is True, the constraint must be satisfied
  - if the Boolean variable is False, the constraint might be or might be not satisfied
- CP-SAT's `only_enforce_if` method activates a constraint only if a condition is met

```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()

# A value representing the load that needs to be transported
load_value = model.new_int_var(0, 100, "load_value")

# A variable to decide which truck to rent
truck_a = model.new_bool_var("truck_a")
truck_b = model.new_bool_var("truck_b")
truck_c = model.new_bool_var("truck_c")

# Rent at least one truck
model.add_at_least_one([truck_a, truck_b, truck_c])

# Depending on which truck is rented, the load value is limited
model.add(load_value <= 50).only_enforce_if(truck_a)
model.add(load_value <= 80).only_enforce_if(truck_b)
model.add(load_value <= 100).only_enforce_if(truck_c)

# some logic determined the load value to be ... 75
model.add(load_value == 75)

# Minimize the rent cost
model.minimize(30 * truck_a + 40 * truck_b + 80 * truck_c)

solver=cp_model.CpSolver()
solver.solve(model)

print(solver.Value(truck_a), solver.Value(truck_b), solver.Value(truck_c)) # 0 1 0
```

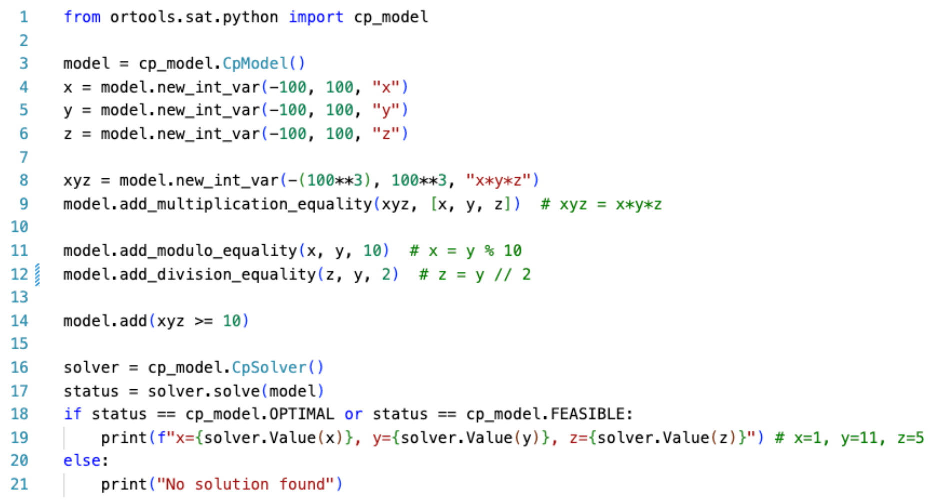
ABSOLUTE VALUES, MAXIMUM/MINIMUM FUNCTIONS WITH INTEGER VARIABLES

- In CP-SAT, `abs`, `max` and `min` must be handled using auxiliary variables and special constraints
- The auxiliary variables are then used in other constraints

```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()
x = model.new_int_var(-100, 100, "x")
y = model.new_int_var(-100, 100, "y")
z = model.new_int_var(-100, 100, "z")

# Create an auxiliary variable for the absolute value of x+z
abs_xz = model.new_int_var(0, 200, "|x+z|")
model.add_abs_equality(target=abs_xz, expr=x + z)

# Create auxiliary variables to capture the maximum and minimum of x, (y-1), and z
max_xyz = model.new_int_var(0, 100, "max(x, y-1, z)")
model.add_max_equality(target=max_xyz, exprs=[x, y - 1, z])

min_xyz = model.new_int_var(-100, 100, "min(x, y-1, z)")
model.add_min_equality(target=min_xyz, exprs=[x, y - 1, z])

model.add(abs_xz + max_xyz == min_xyz)

solver = cp_model.CpSolver()
solver.solve(model)

print(f"x={solver.Value(x)}, y={solver.Value(y)}, z={solver.Value(z)}") # x=0, y=-1, z=0
```

MULTIPLICATION, DIVISION AND MODULO

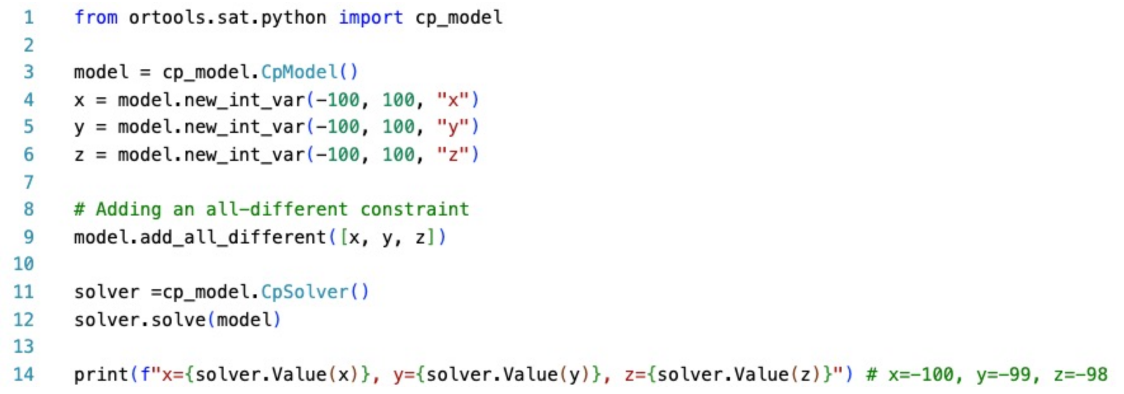
- Multiplication, division (integer) and modulo are supported, through the use of auxiliary variables and special constraints, such as abs, min, max



```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()
x = model.new_int_var(-100, 100, "x")
y = model.new_int_var(-100, 100, "y")
z = model.new_int_var(-100, 100, "z")

xyz = model.new_int_var(-(100**3), 100**3, "x*y*z")
model.add_multiplication_equality(xyz, [x, y, z])  # xyz = x*y*z

model.add_modulo_equality(x, y, 10)  # x = y % 10
model.add_division_equality(z, y, 2)  # z = y // 2

model.add(xyz >= 10)

solver = cp_model.CpSolver()
status = solver.solve(model)
if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
    print(f"x={solver.Value(x)}, y={solver.Value(y)}, z={solver.Value(z)}") # x=1, y=11, z=5
else:
    print("No solution found")
```

ALLDIFFERENT

- One of the most useful constraints is ALLDIFFERENT, it enforces a set of variables to assume different values
- Use of `add_all_different` simplifies modeling and utilizes a specialized performant domain-based propagator



```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()
x = model.new_int_var(-100, 100, "x")
y = model.new_int_var(-100, 100, "y")
z = model.new_int_var(-100, 100, "z")

# Adding an all-different constraint
model.add_all_different([x, y, z])

solver =cp_model.CpSolver()
solver.solve(model)

print(f"x={solver.Value(x)}, y={solver.Value(y)}, z={solver.Value(z)}") # x=-100, y=-99, z=-98
```

DOMAINS AND COMBINATIONS

- Hardcoded allowed configurations of variable values can be enforced with the `add_allowed_assignments` method

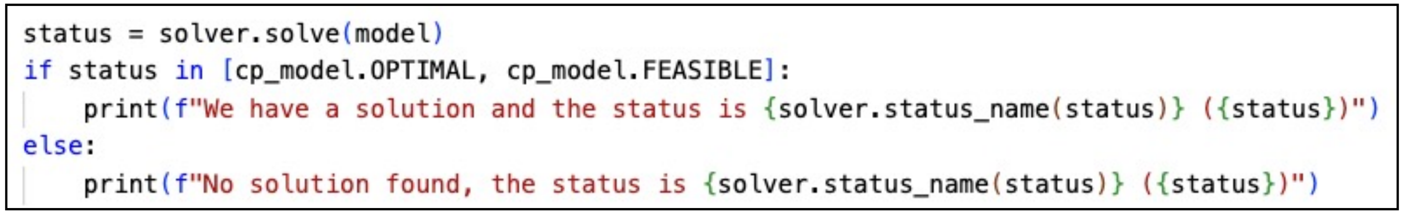
Employee 1	Employee 2	Employee 3	Employee 4
1	0	1	0
0	1	1	0
1	0	0	1
0	1	0	1

- Similarly, variable values can be prohibited with the `add_forbidden_assignments` method
- A related method is `add_linear_expression_in_domain` which enforces the result of a linear expression to be a value of a domain

```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()
x_employee_1 = model.new_bool_var("x_employee_1")
x_employee_2 = model.new_bool_var("x_employee_2")
x_employee_3 = model.new_bool_var("x_employee_3")
x_employee_4 = model.new_bool_var("x_employee_4")

# Define the allowed assignments
allowed_assignments = [
    [1, 0, 1, 0],
    [0, 1, 1, 0],
    [1, 0, 0, 1],
    [0, 1, 0, 1],
]

model.add_allowed_assignments(
    [x_employee_1, x_employee_2, x_employee_3, x_employee_4], allowed_assignments
)

solver = cp_model.CpSolver()
solver.solve(model)

for var in [x_employee_1, x_employee_2, x_employee_3, x_employee_4]:
    print(f"{var.Name()}={solver.Value(var)}")

# x_employee_1=0
# x_employee_2=1
# x_employee_3=0
# x_employee_4=1
```

CP-SAT PARAMETERS (1/3)

- **Logging**
  - `solver.parameters.log_search_progress = True # enables logging`
- **Time Limit**
  - `solver.parameters.max_time_in_seconds = 60  # 60s time limit`
- **Status**
  - `OPTIMAL (4)`
  - `FEASIBLE (2)`
  - `INFEASIBLE (3)`
  - `MODEL_INVALID (1)`
  - `UNKNOWN (0)`



```python
status = solver.solve(model)
if status in [cp_model.OPTIMAL, cp_model.FEASIBLE]:
    print(f"We have a solution and the status is {solver.status_name(status)} ({status})")
else:
    print(f"No solution found, the status is {solver.status_name(status)} ({status})")
```

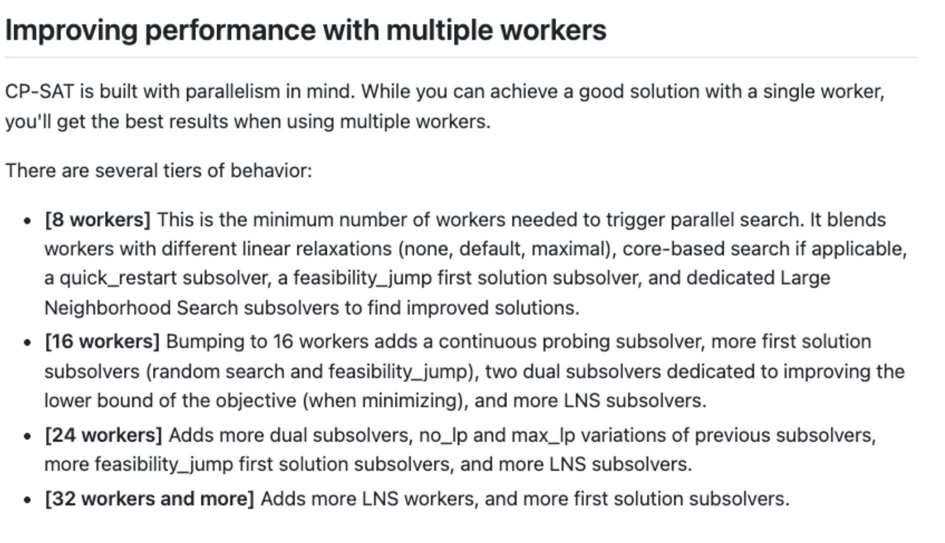
- **Gap to optimality**
  - `solver.parameters.relative_gap_limit = 0.05 # stop when the solution is within 5% of the optimum`

CP-SAT PARAMETERS (2/3)

- **Hints**: can be used to warm-start the solver
  - `model.add_hint(x, 1)  # Suggest that x will probably be 1`
  - `solver.parameters.fix_variables_to_their_hinted_value = True  # fix variables to hints`
  - `solver.parameters.debug_crash_on_bad_hint = True  # throw error if a hint is incorrect`
  - `model.clear_hints()  # clear hints, for model copying avoidance`
- **Assumptions**: temporary constraints added for a single solve call
  - Assumptions can only be stated for booleans
  - `model.add_assumptions([b1, ~b2])  # assume b1=True, b2=False`
  - `model.clear_assumptions()  # clear all assumptions`
- **Presolve**: a step that tries to simplify the model, prior to solving it (reduces search space, ehnances performance)
  - `solver.parameters.cp_model_presolve = False  # disable presolve`
  - `solver.parameters.presolve_probing_deterministic_time_limit = 5  # use 5s to try fixing variables for finding solutions`

CP-SAT PARAMETERS (3/3)

- **Parallelization**
  - `solver.parameters.num_workers = 8  # use 8 cores`
- **Subsolvers**:
  - Full problem subsolvers: solvers that search the full problem space, (e.g., by a branch-and-bound algorithm)
  - Incomplete subsolvers: solvers that work heuristically (e.g., Large Neighborhood Search, feasibility pump)
  - First solution subsolvers: solvers that try to find a first solution as soon as possible



Improving performance with multiple workers

CP-SAT is built with parallelism in mind. While you can achieve a good solution with a single worker, you'll get the best results when using multiple workers.

There are several tiers of behavior:

- **[8 workers]** This is the minimum number of workers needed to trigger parallel search. It blends workers with different linear relaxations (none, default, maximal), core-based search if applicable, a quick_restart subsolver, a feasibility_jump first solution subsolver, and dedicated Large Neighborhood Search subsolvers to find improved solutions.
- **[16 workers]** Bumping to 16 workers adds a continuous probing subsolver, more first solution subsolvers (random search and feasibility_jump), two dual subsolvers dedicated to improving the lower bound of the objective (when minimizing), and more LNS subsolvers.
- **[24 workers]** Adds more dual subsolvers, no_lp and max_lp variations of previous subsolvers, more feasibility_jump first solution subsolvers, and more LNS subsolvers.
- **[32 workers and more]** Adds more LNS workers, and more first solution subsolvers.

https://github.com/google/or-tools/blob/main/ortools/sat/docs/troubleshooting.md#improving-performance-with-multiple-workers

CALLBACKS

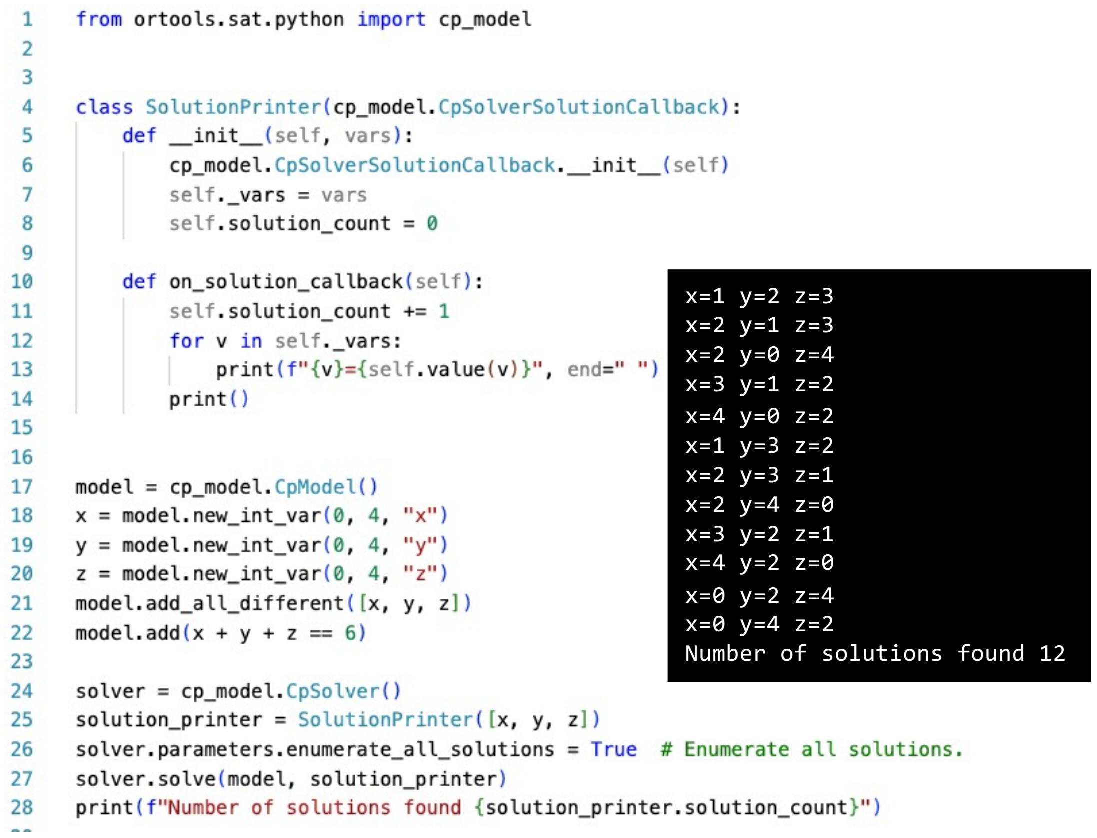
- A callback is a mechanism that lets the user interact with the solver while it is running
- Types of callbacks
  - Solution callback: triggered every time a new feasible solution is found
  - Bounds callback*: triggered when the proven bound improves
  - Log callback: triggered every time a new solution or a new bound is found

```
solver.parameters.log_search_progress = True  # Enable logging
solver.log_callback = lambda msg: print("LOG:", msg)
```

*added in version 9.10 of ORTools

CALLBACK EXAMPLE

Assume 3 variables that take values from the domain {0,1,2,3,4}. Given that these variables should have different values, and their sum should be 6, print all possible value assignments



```python
from ortools.sat.python import cp_model


class SolutionPrinter(cp_model.CpSolverSolutionCallback):
    def __init__(self, vars):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self._vars = vars
        self.solution_count = 0

    def on_solution_callback(self):
        self.solution_count += 1
        for v in self._vars:
            print(f"{v}={self.value(v)}", end=" ")
        print()


model = cp_model.CpModel()
x = model.new_int_var(0, 4, "x")
y = model.new_int_var(0, 4, "y")
z = model.new_int_var(0, 4, "z")
model.add_all_different([x, y, z])
model.add(x + y + z == 6)

solver = cp_model.CpSolver()
solution_printer = SolutionPrinter([x, y, z])
solver.parameters.enumerate_all_solutions = True  # Enumerate all solutions.
solver.solve(model, solution_printer)
print(f"Number of solutions found {solution_printer.solution_count}")
```

```
x=1 y=2 z=3
x=2 y=1 z=3
x=2 y=0 z=4
x=3 y=1 z=2
x=4 y=0 z=2
x=1 y=3 z=2
x=2 y=3 z=1
x=2 y=4 z=0
x=3 y=2 z=1
x=4 y=2 z=0
x=0 y=2 z=4
x=0 y=4 z=2
Number of solutions found 12
```

HANDS-ON ACTIVITIES

- Assignment Problem
- Capacitated Facility Location Problem
- Set Partitioning Problem

HANDS-ON – ASSIGNMENT PROBLEM

- A set of tasks has to completed by assigning them to an equal number of available employees. Each employee should be assigned to exactly one task and vice versa. The cost of assigning each task to each employee is given. Find the minimum cost assignment using CP-SAT
- The data of the problem are in `assign10.txt`
- A solution having two missing parts is in `day2_ho1_template.py`. Fill in the missing parts
- Solve the bigger problem found in assign800.txt. How much time is needed for CP-SAT to reach the optimal solution?

```
# The first line is the number of tasks T (also the number of employees)
# The next T values are the costs of assigning the first task to
# each employee (i.e., task1 to employee 1 cost=52, task1 to
# employee 2 cost=89 etc.). The next T values are the costs of
# assigning the second task to each employee and so on and so forth.

 10
 52 89 40 77 89 14 9 77 92 77
 20 17 80 96 14 43 4 69 5 29
 17 34 50 77 71 31 74 26 100 20
 55 63 85 35 31 46 51 70 21 24
 70 13 14 26 49 80 24 48 31 42
 99 72 15 91 70 94 95 50 35 30
 77 83 59 20 49 97 66 21 82 42
 37 78 99 75 77 2 39 53 54 90
 79 6 19 90 57 16 64 15 50 2
 23 75 22 81 44 22 28 67 75 81
```

HANDS-ON – ASSIGNMENT PROBLEM – MODEL

Given a cost matrix c_{ij} where $i = 1, 2, \ldots, n$ (tasks) and $j = 1, 2, \ldots, n$ (employees), the goal is to assign each task to exactly one employee and each employee to exactly one task such that the total cost is minimized.

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i = 1, \ldots, n \\ & \sum_{i=1}^{n} x_{ij} = 1 \quad \forall j = 1, \ldots, n \\ & x_{ij} \in \{0, 1\} \end{aligned}$$

where: $x_{ij} = 1$ if task i is assigned to employee j, and 0 otherwise.

HANDS-ON – CAPACITATED FACILITY LOCATION PROBLEM

- A number of warehouses will serve a number of customers. Some of the warehouses might remain closed if this is advantageous. For each warehouse it is given the maximum demand that it can serve and the cost of opening the facility. The demand of each customer is given alongside with the cost of serving all of its demand by each warehouse. Find which warehouses should open and how much of each customer's demand will be served by each warehouse.
- Note that the demand of each customer assumes an integer value and the parts of the customer's demand that will be served by each warehouse are also integral.
- The data of the problem are in w4_c8.txt
- A solution having a missing part is in day2_ho2_template.py. Fill in the missing part.
- Solve the bigger problem found in file w16_c50.txt. How much time is needed for CP-SAT to reach the optimal solution?

```
# The first line contains the number of warehouses and the number of customers
# The text <number of warehouses> lines contain the maximum demand that each
# warehouse can serve and the setup cost of the warehouse
# The remaining lines contain:
# - the demand of each customer
# - the cost of serving the full demand of the customer by each warehouse
# Here, the demand of the first customer is 146, and the cost of serving this demand
# from warehouse 1 is 6739.725, from warehouse 2 10355.05, etc.
# The demand of the second customer is 87, and the cost of serving this demand
# from warehouse 1 is 3204.8625, from warehouse 2 5457.075, etc.

 4 8
 5000 7500
 5000 0
 5000 7500
 5000 7500
 146
 6739.725 10355.05 7650.4 5219.5
 87
 3204.8625 5457.075 3845.4 2396.85
 672
 4914 26409.6 19622.4 13876.8
 1337
 32372.1125 29982.225 21024.325 29681.4
 559
 6421.5125 23701.6 16197.025 10383.425
 1370
 81972.375 28499.25 43134 65767.5
 1089
 33391.4625 26544.375 6370.65 16770.6
 6412
 372672.6 229188 203364 322800
```

HANDS-ON – CAPACITATED FACILITY LOCATION PROBLEM – MODEL

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m} f_i y_i + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{i=1}^{m} x_{ij} = d_j \quad \forall j = 1, \dots, n \\
& \sum_{j=1}^{n} x_{ij} \le Q_i y_i \quad \forall i = 1, \dots, m \\
& x_{ij} \ge 0 \quad \forall i = 1, \dots, m; \; j = 1, \dots, n \\
& y_i \in \{0, 1\} \quad \forall i = 1, \dots, m
\end{aligned}
$$

Parameters:

- f_i: fixed cost to open warehouse i
- c_{ij}: cost to serve customer j from warehouse i
- d_j: demand of customer j
- Q_i: capacity of warehouse i

Decision variables:

- y_i: 1 if warehouse i is opened, 0 otherwise
- x_{ij}: amount of demand of customer j served by warehouse i

HANDS-ON – SET PARTITIONING PROBLEM

- Suppose that we have a set of products and we want to deliver them to their destinations. Products can be grouped in sets for delivery and every such group has an associated cost of delivery. Find the minimum cost for delivering all products. Use CP-SAT!
- The data of the problem are in `sp_135_51975.txt`
- An almost complete solution is in `day2_ho3_template.py`. Fill in the missing part

```
# This instance gives you the number of products and the number of subsets in the first line.
# The rest of the lines gives you information about the subsets in the following format: <cost>
<number of products> <the products>

# In this instance, you have 135 products and 51975 subsets.
# The first subset costs 5325 and has 4 products: 1, 4, 57, 92
# The second subset costs 5382 and has 4 products: 1, 6, 56, 103
# and so on and so forth

135 51975
5325 4 1 4 57 92
5382 4 1 6 56 103
9339 6 1 36 56 57 73 96
2184 3 1 56 57
972 1 1
5817 4 2 3 85 91
5817 4 2 5 98 106
10230 6 2 32 55 81 91 98
8943 6 2 32 55 91 98 127
8907 6 2 32 55 91 98 128
8922 5 2 81 91 98 134
7641 5 2 91 98 127 134
7599 5 2 91 98 128 134
2619 3 2 91 98
...
```

HANDS-ON – SET PARTITIONING PROBLEM – MODEL

Given a set of products $j = 1, 2, \ldots, m$ and a set of subsets $i = 1, 2, \ldots, n$, where each subset i has a cost c_i and covers certain products, the objective is to select a collection of subsets such that every product is covered exactly once (partitioned), while minimizing the total cost.

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} c_i x_i \\
\text{s.t.} \quad & \sum_{i=1}^{n} a_{ij} x_i = 1 \quad \forall j = 1, \ldots, m \\
& x_i \in \{0, 1\} \quad \forall i = 1, \ldots, n
\end{aligned}$$

where:

- $x_i = 1$ if subset i is selected, and 0 otherwise.
- $a_{ij} = 1$ if product j is covered by subset i, and 0 otherwise.

REFERENCES

- OR-Tools CP-SAT Solver, https://developers.google.com/optimization/cp/cp_solver
- CP-SAT Primer, https://d-krupke.github.io/cpsat-primer/
- https://www.feasible.club/