CIS 189

# Lecture 9: More Constraint Programming

Rohan Menezes rohanmenezes@alumni.upenn.edu

# Logistics

- **Project proposals** due this Thursday 3/23
- **HW4** due next Tuesday 3/28
- Please submit your project pairing on Gradescope if you haven't already!

# Recap: Constraint Programs

- Find an assignment of variables to values, subject to general constraints
- Discrete, finitely bounded domains (integers only)
- May or may not optimize an objective

# Constraints for BoolVars

- Recall `model.NewBoolVar(name)`
  - Equivalent to `model.NewIntVar(0, 1, name)`
- `boolvar.Not()`
- `model.AddBoolOr(boolvars_list)`
- `model.AddBoolAnd(boolvars_list)`
- `model.AddImplication(b1, b2)`

# Ex: Magic Sequence

- A **magic sequence** is a sequence $s_0, s_1, \ldots, s_n$ where $s_i$ = number of occurrences of $i$ in the sequence
- Ex:

| $s_0$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|
| ? | ? | ? | ? | ? |

# Ex: Magic Sequence

- A **magic sequence** is a sequence $s_0, s_1, \dots, s_n$ where $s_i$ = number of occurrences of $i$ in the sequence
- Ex:

| $s_0$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|
| 2 | 1 | 2 | 0 | 0 |

# Reification

- What if we want to make constraints based on other constraints (rather than just variables)?
- **Reification:** constraints as first-class citizens

- Introduce a new boolean (0/1) variable $b$ which is true if and only if constraint $c$ holds ($b \Leftrightarrow c$)
    - Essentially, name truth value of $c$ with variable $b$

# Reification in OR-Tools

- OR-Tools API uses **half-reification:** instead of $b \Leftrightarrow c$, just supports $b \Rightarrow c$
  - Can fully reify by combining $b \Rightarrow c$ and $\overline{b} \Rightarrow \overline{c}$
- `constraint.OnlyEnforceIf(bool_var)`
  - Means `bool_var ⇒ constraint`

# ⚠️ Reification Warning

- `constraint.OnlyEnforceIf` only works for these constraints:
    - `Add`
    - `AddBoolOr`
    - `AddBoolAnd`
    - `AddLinearExpressionInDomain` (haven't seen this one yet)

- This is usually all you need

# Magic Sequence in OR-Tools

- Initialize model and $s_i$ variables

```
model = cp_model.CpModel()

# Create s_i variables
S = {}
for i in range(n+1):
    S[i] = model.NewIntVar(0, n+1, f's_{i}')
```

# Magic Sequence in OR-Tools

- Reify constraints $s_i = j$ into new boolean variables

```
# Reified constraints: eq[i, j] <-> s_i == j
eq = {}
for i in range(n+1):
    for j in range(n+1):
        eq[i, j] = model.NewBoolVar(f's_{i} == {j}')
        model.Add(S[i] == j).OnlyEnforceIf(eq[i, j])
        model.Add(S[i] != j).OnlyEnforceIf(eq[i, j].Not())
```

# Magic Sequence in OR-Tools

- Make $s_i$ equal to number of occurrences of $i$

```
# s_i = number of occurrences of i in sequence
for i in range(n+1):
    model.Add(
        S[i] == sum(eq[j, i] for j in range(n+1))
    )
```

# Magic Sequence in OR-Tools

- Solve and print the output

```
solver = cp_model.CpSolver()
if solver.Solve(model) == cp_model.FEASIBLE:
    print([f's_{i}={solver.Value(S[i])}' for i in range(n+1)])
```

```
['s_0=2', 's_1=1', 's_2=2', 's_3=0', 's_4=0']
```

# Non-contiguous Domains

- `cp_model.Domain.FromValues([0,2,4,6,8])`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

- `cp_model.Domain.FromIntervals([0, 2],[6, 8])`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

- `model.NewIntVarFromDomain(domain, name)`

# Linear Expressions on Domains

- Enforce that result of a linear expression must fall into a domain
- ```
  cp_model.AddLinearExpressionInDomain(
      x + y,
      cp_model.Domain.FromValues([0,2,4])
  )
  ```

| | | | | |
|---|---|---|---|---|
| 0,0 | 1,0 | 2,0 | 3,0 | 4,0 |
| 0,1 | 1,1 | 2,1 | 3,1 | 4,1 |
| 0,2 | 1,2 | 2,2 | 3,2 | 4,2 |
| 0,3 | 1,3 | 2,3 | 3,3 | 4,3 |
| 0,4 | 1,4 | 2,4 | 3,4 | 4,4 |
  ```

# Ex: Shipping Allotments

- Shipping company has $n$ ships with capacity 100 each
- Want to load all shipments of varying sizes onto ships
- **Goal:** maximize number of ships which have at least 20 capacity unused (in case of emergency)
  - See worked solution in additional code (ships.py)

# Tuning the CP-SAT Solver

- We can play around with CP-SAT internals to possibly speed up the search
- There are tons of parameters that can be adjusted
    - Some are documented better than others...
    - https://github.com/google/or-tools/blob/stable/ortools/sat/sat_parameters.proto
- **Warning:** these things are generally far less important than having a good encoding

# Parallelization

- We can run solver computation in parallel across multiple threads

```
solver = cp_model.CpSolver()
solver.parameters.num_search_workers = 4
```

  - By default, CP-SAT will try to use all available cores

# Hinting

- We can give the model a **hint** to try setting a variable to a specified value

```
# try setting x = 5 first
model.AddHint(x, 5)
```

# Quick & Dirty Optimization

- Finding an optimal solution can take far longer than finding a feasible solution
- Often in practice, we don't *really* care about having the true optimal value with total certainty
    - Just want it to be "close enough"

# Quick & Dirty Optimization

**Solution:**

- Optimize objective and run solver for a reasonable amount of time (depends on your patience)
- Interrupt early with Ctrl+C or `max_time_in_seconds` param
    - If interrupted, solver returns FEASIBLE instead of OPTIMAL
- Print the intermediate objective value and solution and decide if it's "good enough"
    - For tough problems, no guarantee that you are close to optimal!
    - `best_bound` in response stats gives best LB (when minimizing) or UB (when maximizing) proved so far for optimal value

# Quick & Dirty Optimization

- Helpful: set `log_search_progress` param to True
  - Prints every time a new best solution is found
- Sometimes helpful: custom solution callback
  - Called each time any new feasible solution is found

```python
class BestSolutionFinder(cp_model.CpSolverSolutionCallback):

    def __init__(self, minimizing=True):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.minimizing = minimizing
        self.best_value = (1 if minimizing else -1) * float('inf')

    def on_solution_callback(self):
        obj = self.ObjectiveValue()
        if (self.minimizing and obj < self.best_value) \
        or (not self.minimizing and obj > self.best_value):
            self.best_value = self.ObjectiveValue()
            print(f'New best value: {self.best_value}')
```

```python
solver = cp_model.CpSolver()
solver.parameters.num_search_workers = 6
solver.parameters.log_search_progress = True
# Our solution callback is redundant to logging
best = BestSolutionFinder()
solver.SolveWithSolutionCallback(model, best)
```

# Approximating Feasibility

- What if non-optimization problem is too hard to solve?
- Can't interrupt early for a "good enough" solution; intermediate solution is feasible or it is not
- What if we were OK with a "not quite feasible" solution?
  - What could "not quite feasible" mean?

# Soft Constraints

- Constraints like `Add(...)` are **hard constraints**
  - Must be satisfied
- **Soft constraints**: can be violated, but incurs a penalty
- Transform feasibility problem into optimization problem by minimizing penalty
  - Allows interrupting early if you're OK with some violated constraints
  - Can sometimes be faster than solving with hard constraints!

# Ex: Soft Graph Coloring

- Hard constraint:

  for every edge $(u, v)$, $color(u) \neq color(v)$

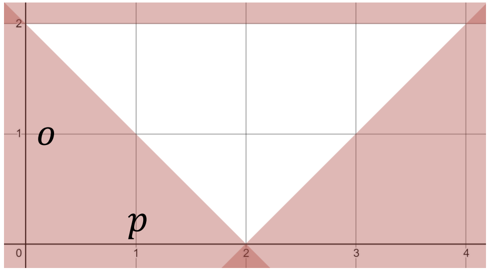
- Soft constraint

  $penalty$ = num. edges $(u, v)$ with $color(u) = color(v)$

- Can count number of violated constraints using reification

# Optimizing Pairs of Objectives

- What if we want to add soft constraint with penalty $p$ but problem already optimizes (say, minimizes) objective $o$?
- **Key idea:** why not minimize both?
- Attempt 1: minimize $\boldsymbol{o} + \boldsymbol{p}$
  - **Problem:** $o$ and $p$ may be interrelated
  - E.g., minimum possible value of $o$ may be lower when $p = 1$ than when $p = 0$



# Optimizing Pairs of Objectives

- **Observation:** avoid interdependence by minimizing $p$ first and using $o$ to break ties
    - Aka, minimize $(p, o)$ over the **lexicographic ordering**
- How to make sure that $p$ is minimized before $o$?
- Attempt 2:
    - minimize $\boldsymbol{Mp} + \boldsymbol{o}$, where $M = o_{max} - o_{min} + 1$
- Can generalize to maximization & general tuples

# Optimizing Pairs of Objectives

- Previous approach doesn't scale well for >2 objectives
- What's another way to do it using multiple calls to `Solve`?

```
model.Minimize(p)
solver.Solve(model)

# Hint (may speed up solving)
model.AddHint(p, solver.Value(p))
model.AddHint(o, solver.Value(o))

# Minimize o (and constrain p based on previous optimal value)
model.Add(p == solver.Value(p)) # use >= or <= if not optimal
model.Minimize(o)
```

# General CP-SAT Modeling Tips

- Don't be afraid to add new variables/constraints, but be aware of roughly how many you have ($O(n)$? $O(n^3)$?)
- Try to restrict range of values for each variable
- Use boolean variables/constraints when possible
- Experiment with hard vs. soft constraints
- If possible, split into subproblems, then combine solutions
- Make it easy to toggle constraints on/off for debugging

# MIP vs CP-SAT

| MIP | CP-SAT |
|---|---|
| • Supports infinite bounds<br>• Supports fractional variables and coefficients<br>• Better handles LP-style problems (with integers mixed in)<br>• Reification of constraints is possible, but requires algebraic modeling trick | • Better handles combinatorial problems, Booleans<br>• More sophisticated interface<br>• Lots of specialized modeling objects<br>• Modeling may be easier<br>• Models may be more extensible<br>• Reification is easier, more performant |

- Neither is clearly more performant in general
- Neither is an evolution of the other