CIS 189

Lecture 8: Intro to Constraint Programming

Rohan Menezes rohanmenezes@alumni.upenn.edu

Logistics

- Final project partners due **this Thursday 3/16**
- Final project proposals due **next Thursday 3/23**
- Homework 4: Grace Hopper Conference
  - **Due Tuesday 3/28 by 4pm**
  - May not be able to finish part 2 until next week
  - Last homework! If you haven't used your late day

Constraints

- Recall: many decision problems involve checking if there is a solution that satisfies certain constraints
- A **constraint** is just a rule that limits which possible solutions are acceptable
- **Ex:** CNF-SAT
    - Solution: a truth assignment
    - Constraints: in each clause, at least one variable is assigned to True

Constraint Satisfaction

- A **constraint satisfaction problem** is defined by:
  - a set of **variables**, each with its own range of **values**
  - a set of **constraints**
- A **candidate solution** is any assignment of vars to values
- Candidate solutions that satisfy all constraints are **feasible**

Constraint Programming

- "Like IP, but with more complex constraints"
    - Combinatorial constraints, possibly non-linear
- OR-Tools has a new constraint programming solver called **CP-SAT**
- Behind the scenes: turns constraints into clauses, then uses SAT solver!
    - vast oversimplification...
- Very successful! "State of the art"

Results of Minizinc CP Challenge 2021

Category	Gold	Silver
Fixed	SICStus Prolog	JaCoP
Free	OR-Tools	PicatSAT
Parallel	OR-Tools	PicatSAT
Open	OR-Tools	sunny-cp
Local Search	Yuck	OscaR/CBLS

CP-SAT Documentation

- For reference (variables, constraints):

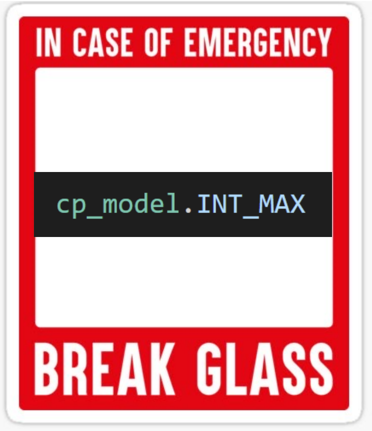
google.github.io/or-tools/python/ortools/sat/python/cp_model.html

- Recommended: keep a tab open while working with CP-SAT until you memorize all the constraints

Basic Variables in CP-SAT

- `model.NewIntVar(lower_bnd, upper_bnd, name)`
- `model.NewBoolVar(name)`
    - Equivalent to `model.NewIntVar(0, 1, name)`

- Returns newly created variable (just like MIP)
- CP-SAT **only** works over discrete, finite domains
    - No `NumVars`, integers only!
    - No infinite bounds



Linear Constraints in CP-SAT

- Adding/scalar multiplying vars gives a (linear) **expression**
- Linear expr. with an (in)equality gives a **linear constraint**
  - Unlike MIP, we can also use not equals `(!=)`
- Unlike MIP, coefficients **must** also be integers
  - If you have fractional coefficients, you need to scale them up to integers or use MIP solver instead
- `model.Add(linear_constraint)`

Basic Nonlinear Constraints

- `model.AddAbsEquality(target, var)`
  - Adds constraint: `target == |var|`
- `model.AddMultiplicationEquality(target, [v1,v2])`
  - Adds constraint: `target == v1 * v2`
- `model.AddMaxEquality(target, var_arr)`
  - Adds constraint: `target == Max(var_arr)`
- Annoying: no first-class nonlinear expressions; must build up complex math expressions piece-by-piece using intermediate variables
  - OR-Tools is really for combinatorial optimization, not equations

The AllDifferent Constraints

- `model.AddAllDifferent(var_arr)`
- Forces all vars in the array to take on different values!
- Very common in practice
  - Esp. for assignment problems, scheduling, etc.

Classic Example: Cryptarithms

- In a cryptarithmetic puzzle, want to replace each letter with a different digit to make the arithmetic valid
    - no leading zeros

```
      S E N D          9 5 6 7
+     M O R E    =>  + 1 0 8 5
-------------      -------------
    M O N E Y      1 0 6 5 2
```

Classic Example: Cryptarithms

Constraint program:

- Variables for each letter, most with range [0…9]
  - S, M have range [1…9], since no leading zeros
- Constraint 1: the arithmetic expression holds
- Constraint 2: all vars have different value

```
    S E N D
+   M O R E
-----------
  M O N E Y
```

Cryptarithms in OR-Tools

- Initializing the model and declaring variables

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

model = cp_model.CpModel()
S = model.NewIntVar(1, 9, 'S')
E = model.NewIntVar(0, 9, 'E')
N = model.NewIntVar(0, 9, 'N')
D = model.NewIntVar(0, 9, 'D')
M = model.NewIntVar(1, 9, 'M')
O = model.NewIntVar(0, 9, 'O')
R = model.NewIntVar(0, 9, 'R')
Y = model.NewIntVar(0, 9, 'Y')
```

```
      S E N D
+     M O R E
-------------
    M O N E Y
```

Cryptarithms in OR-Tools

- Add arithmetic and all different constraints (yes, that easy!)

```
model.Add(
                1000*S + 100*E + 10*N + D
  +             1000*M + 100*O + 10*R + E
  ==   10000*M + 1000*O + 100*N + 10*E + Y
)
model.AddAllDifferent([S,E,N,D,M,O,R,Y])
```

```
     S E N D
+    M O R E
------------
   M O N E Y
```

Cryptarithms in OR-Tools

- Solve and print the solution

```
solver = cp_model.CpSolver()
if solver.Solve(model) == cp_model.OPTIMAL:
    print([f'{v}={solver.Value(v)}' for v in [S,E,N,D,M,O,R,Y]])
```

- Output:

```
['S=9', 'E=5', 'N=6', 'D=7', 'M=1', 'O=0', 'R=8', 'Y=2']
```

```
      S E N D
+     M O R E
-------------
    M O N E Y
```

Optimization with CP-SAT

- We can also maximize/minimize an expression, e.g.

```
model.Maximize(7*a + b)
```

```
model.Minimize(
    sum(x[i] for i in range(10))
)
```

The Element Constraint

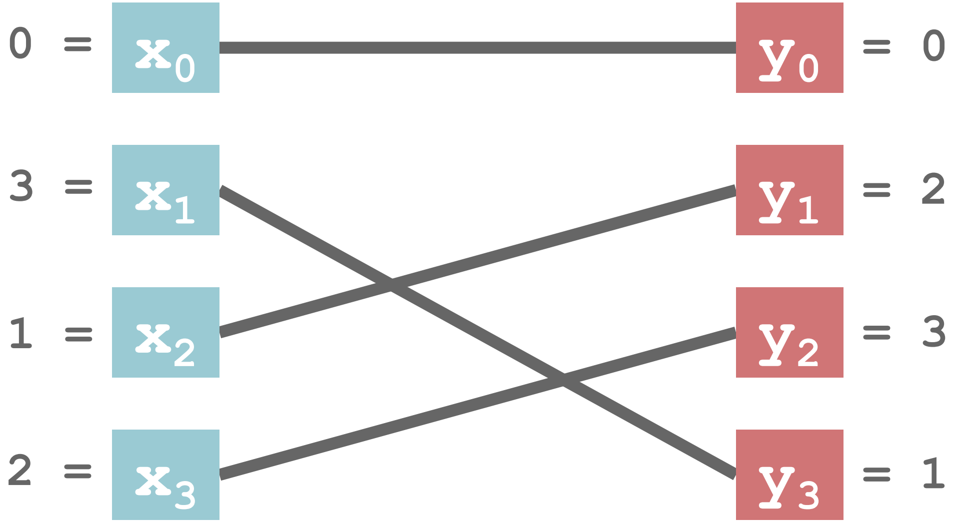
- **`model.AddElement(index, var_arr, target)`**
- Adds constraint: `target == var_arr[index]`
- Useful because `index` can be a variable
- The `var_arr` can also contain constants!

The Inverse Constraint

- `model.AddInverse(var_arr, inv_arr)`
- The arrays should have the same size n (can't use dicts)
- The vars in both arrays can only take values from 0 to $n-1$
- Adds the following constraints:
  - If `var_arr[i] == j`, then `inv_arr[j] == i`
  - If `inv_arr[j] == i`, then `var_arr[i] == j`
- Intuition: sets up a "perfect matching" between the two sets of variables

The Inverse Constraint

- `model.AddInverse([x0,x1,x2,x3], [y0,y1,y2,y3])`



Ex: Taxi Assignment

- A taxi service has n customers waiting for pickup
- There are n taxis available, one for each customer
- We know the distance between each taxi and customer
- Want to assign taxis to customers in order to minimize the total distance traveled by all taxis (save gas)
  - See code example (taxis.py) for worked solution

Interval Variables

- CP-SAT has special variables that provide “syntactic sugar” for representing time intervals
- `model.NewIntervalVar(start, duration, end, name)`
- Represents an interval, enforcing `end - start == duration`
  - `start, end, duration` can be constants or variables!
- Note: there is no way to access `start, end, duration` of an interval by default
  - Recommended: directly add them as fields of the interval object

Interval Variables

- Note: there is no way to access `start, end, duration` of an interval by default
  - Recommended: directly add them as fields of the interval, e.g. `interval.start = start`
- **`model.AddNoOverlap(interval_arr)`**
- Powerful constraint: enforces that all intervals in the array do not overlap with each other!
  - It's OK to have shared start/endpoints

Job Shop Scheduling

- m **machines** that do **tasks** which take varying time to finish
  - Machines can do only one task at a time
  - Once a task is started, it must be finished
- n **jobs,** each consisting of a list of tasks
  - Each task must be performed on one specific machine
  - Each task in a job cannot be started until the previous task in the job finished
- **Goal:** minimize the **makespan** (time to finish all jobs)

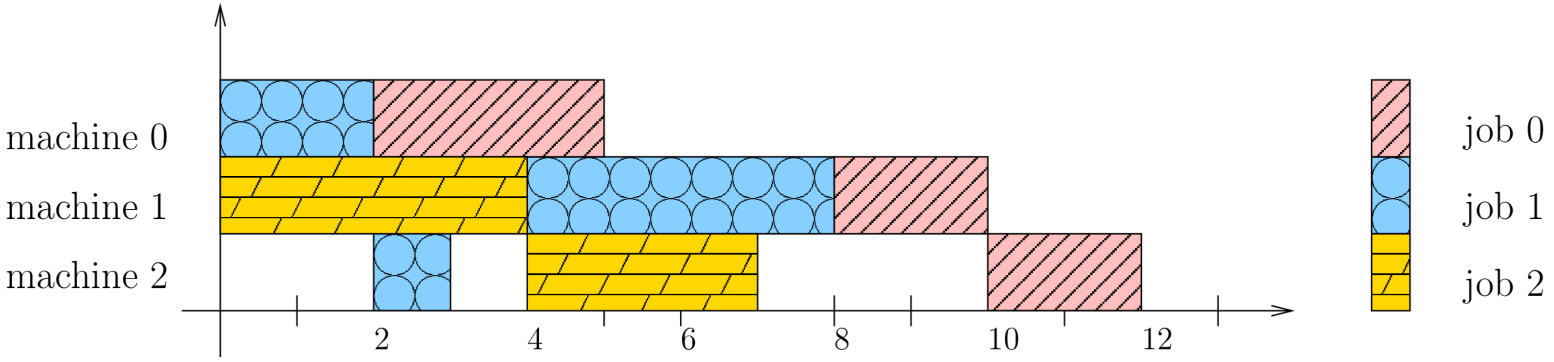
Ex: Job Shop Scheduling

- 3 machines, numbered 0, 1, 2
- Tasks are pairs of (which machine, time required)
- 3 jobs:

```
jobs_data = [  # task = (machine_id, processing_time).
    [(0, 3), (1, 2), (2, 2)],  # Job 0
    [(0, 2), (2, 1), (1, 4)],  # Job 1
    [(1, 4), (2, 3)]           # Job 2
]
```

Ex: Job Shop Scheduling

- Sample feasible (not optimal) solution



- What's the makespan of this solution?
  - See code example (jobshop.py) for worked solution