LECTURE 8

Ishaan Lal

October 25, 2024

1 Introduction

Today, we dive into the world of Constraint Programming. We've spent good time exploring LP and MIP, but now we take a step towards further generalization.

2 Constraint Programming

With Mixed-Integer Programming, we were tasked with optimizing a *linear* objective function subject to *linear constraints*. While we spent time massaging the requirement of the objective function being linear (by considering pointwise discontinuous linear functions and piecewise linear functions), we haven't yet considered non-linear constraints.

Many *decision* problems involve checking if there is a solution that satisfies certain constraints. The same is true for optimization problems, but instead of checking existential, we find the best solution. For now, we will stay in the world of decision problems.

A **constraint** is imply a rule that limits which possible solutions are acceptable.

For example, with the boolean satisfiability problem (SAT), the solution we seek is a truth assignment to variables, and the constraint is that in each clause, at least one variable is assigned to True.

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 variables to values. Candidate solutions that satisfy all of the constraints are called **feasible**.

As an example, consider the following instance of SAT:

$$\varphi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_4)$$

Here, the **variables** are $\{x_1, x_2, x_3, x_4\}$, each have **values** of $\{\texttt{TRUE}, \texttt{FALSE}\}$. Two **candidate solutions** are:

$$x_1 = \texttt{FALSE}, x_2 = \texttt{TRUE}, x_3 = \texttt{FALSE}, x_4 = \texttt{FALSE}$$
$$x_1 = \texttt{TRUE}, x_2 = \texttt{FALSE}, x_3 = \texttt{TRUE}, x_4 = \texttt{FALSE}$$

The first candidate solution is *not feasible*, but the second candidate solution **is feasible**, because it fulfills the constraint of every clause having at least one true literal.

Notice how, in its current formulation, we can't easily describe SAT as an instance of MIP. Despite some similarities of solving a problem subject to constraints, they are noticeably different due to the nature of what the constraints are, as well as MIP being an optimization problem and SAT being a decision problem.

Constraint Programming is similar to MIP, but with more complex constraints (combinatorial constraints, possibly non-linear). OR-Tools gives us a solver called CP-SAT to solve constraint programming problems. Behind the scenes, as a gross oversimplification, specified constraints are turned into clauses, and then a SAT solver is used.

3 Using CP-SAT

Programming with CP-SAT is similar to programming with CBC-MIP.

One staggering difference is that CP-SAT **only** works over discrete, finite domains. This implies that our variables can only be integers – no float values allowed! Additionally, we cannot have infinite bounds. But as we have discussed before, most problems have an implied upper and lower bound on their variables.

3.1 Defining Variables

To define a numerical integer variable:

```
model.NewIntVar(lower_bound, upper_bound, name)
```

To define a boolean variable (0/1 var):

```
model.NewBoolVar(name)
```

3.2 Defining Linear Constraints

With MIP we dealt with linear constraints. With CP-SAT, we still want to be able to consider them. Recall that a **linear constraint** is a linear expression with an (in)equality.

For good news, unlike MIP, CP-SAT also gives us the ability to use "not equals" (!=).

For bad news, unlike MIP, CP-SAT requires coefficients to be integers.

Pro Tip 1

Suppose we had the following MIP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 5 \\ & 2x_1 - 3x_2 \leq 4 \end{aligned}$$

Using a MIP solver, the optimal solution is:

$$x_1 = 3.8 \quad x_2 = 1.2$$

CP-SAT would not return this, because it will only return integer solutions. To remedy this, we can modify our MIP by scaling *the constants of the inequality* by a factor of 10 (or a higher power of 10):

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 50 \\ & 2x_1 - 3x_2 \leq 40 \end{aligned}$$

Here, the MIP solver and CP-SAT solver both return:

$$x_1 = 38 \quad x_2 = 12$$

And we can obtain the correct solution by dividing the solution by the scale.

3.3 Some New Constraints

Of course, we don't just want linear constraints, because then we would just be living in the world of MIP. Let's look at some new, powerful constraints:

```
model.AddMultiplicationEquality(target, [v1, v2, ..., vn]
```

This will add the constraint:

$$\text{target} == v_1 \cdot v_2 \cdot \ldots \cdot v_n$$

```
model.AddMaxEquality(target, [v1, v2, ..., vn])
```

This will add the constraint:

$$\text{target} == \max(v_1, v_2, \ldots, v_n)$$

```
model.AddAllDifferent([v1, v2, ..., vn])
```

This will add a constraint to ensure that all variables in the array take on different values.

4 Solving a Cryptarithm Puzzle

A cryptarithmetic puzzle is a fusion of arithmetic and wordplay. Consider the following instance of a cryptarithm:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

To solve the puzzle, assign each letter a **different digit** and ensure that the arithmetic is still valid. Importantly, note that leading zeros are not allowed. So you cannot assign $S = 0$ or $M = 0$
The solution to the puzzle is:

$$S = 9 \quad E = 5 \quad N = 6 \quad D = 7 \quad M = 1 \quad O = 0 \quad R = 8 \quad Y = 2$$

4.1 Solving Cryptarithm with CP-SAT

We can describe an instance of a Cryptarithm puzzle as a constraint program. To do this, we must define our **variables, their range of values, and the constraints**.

In this case, the variables are the letters themselves. Each variable will take on a value in the integer interval [0..9], except for S and M, which has range [1..9]

The constraints are a bit more interesting. We have:

- Constraint 1: The arithmetic expression holds
- Constraint 2: all the variables have different values.

Please refer to the code posted on the website for how to solve this problem programatically.

5 Disclaimer

With CP-SAT, we still have the option of minimizing or maximizing an objective function. This is done the same way we did it with CBC-MIP.

6 More Constraints and Variables

Lets take a look at a few more new constraints:

6.1 The Element Constraint

```
model.AddElement(index, var_arr, target)
```

This will add the constraint

```
target == var_arr[index]
```

This can be useful because "index" can be a variable, not just a constant! Also, "var_arr" can contain both variables and constants!

6.2 The Inverse Constraint

```
model.AddInverse(var_arr, inv_arr)
```

This constraint has two parameters – both of which must be arrays of the same size.
This constraint ensures that if var_arr[i] == j, then inv_arr[j] == i.

6.3 The Taxi Assignment Problem

The Taxi Assignment Problem is described as follows:

- There are n customers waiting for pickup by taxis
- There are n taxis available, one for each customer
- We know the distance between each taxi and each customer
- We want to assign taxis to customers in order to minimize the total distance traveled by all taxis.

To see a worked through solution to this problem, refer to the code on the website.

6.4 Interval Variables

Interval Variables are a type of variables that allow us to define a "time interval" by defining a start time, end time and duration.

```
model.NewIntervalVar(start, duration, end, name)
```

This variable represents an interval such that: start + duration = end. Additionally, these variables can be constants or variables!

6.5 No Overlap Constraint

```
model.AddNoOverlap(interval_arr)
```

This constraint takes an array of interval variables, and enforces that all intervals in the array do not overlap with each other (besides shared start/endpoints if necessary).

6.6 The Job Shop Scheduling Problem

We now solve the following problem:
There are m machines that complete tasks which take varying time to finish. Machines can only process one task at a time, and once a task is started, it must be finished before starting another task.
There are n jobs. A "job" consists 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 is finished.
The goal of the problem is to minimize the **makespan** (the time to finish all jobs)
Please refer to the slides and the code for visualizations and the solution.