More on Constraint Programming

Ishaan Lal

November 3, 2024

1 Introduction

Well done, you've learned a lot throughout the many weeks of the semester and are equipped with a lot of tools to solve a variety of difficult problems! Last week, we introduced Constraint Programming. Today, we will continue that discussion further, and solve some more fun problems.

2 Boolean Variables

Many of the constraints we introduced last week were with respect to integer variables. There exist some constraints that only work with boolean variables. Recall, to instantiate a boolean variable, we write:

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

and under the hood, this is absolutely equivalent to writing:

```
model.NewIntVar(0, 1, name)
```

The only difference that separates boolean variables from other integer variables is that the former can only take on values $\in \{0, 1\}$.

The following expressions exist to allow for boolean logic:

- `boolvar.Not()` $\implies$ negates a boolean variable $\overline{\text{boolvar}}$
- `model.AddBoolOr([v1, v2, ..., vn])` $\implies$ where $v_1, v_2, ..., v_n$ are boolean variables, creates the expression $v_1 \vee v_2 \vee ... \vee v_n$
- `model.AddBoolAnd([v1, v2, ..., vn])` $\implies$ where v1, v2, ..., vn are boolean variables, creates the expression $v_1 \wedge v_2 \wedge ... \wedge v_n$
- `model.AddImplication(b1, b2)` $\implies$ where b_1, b_2 are boolean variables, creates the expression $b_1 \implies b_2$

3 Magic Sequence

Let's introduce the first problem that we will solve:

A **magic sequence** is a sequence $s_0, s_1, ..., s_n$ where s_i equals the number of occurrences of value i in the sequence.

As an example, $2, 1, 2, 0, 0$ is a magic sequence with $n = 4$ (length 5). We see that $s_0 = 2$, and indeed, there are 2 occurrences of the value 0 in the sequence. Additionally, $s_1 = 1$, and there is indeed 1 occurrence of the value 1. And so on.

Suppose we wanted to determine if a magic sequence exists for any value of n. Moreover, if a magic sequence exists, we want to find what the sequence is!

3.1 Solving with Constraint Programming

Of course, we will use constraint programming to solve this! To do so, we need to define three things: **the variables, their values, and the constraints**.
The variables should be relatively straightforward: we will have a variable for each s_i. But what are the values that an arbitrary s_i can take on? Well, the lower bound is 0, and the upper bound is the length of the sequence, $n + 1$.

Pause

Take a moment to ensure you understand why the upper bound is $n + 1$. Hint: it follows from the definition of a magic sequence.

Great. But what about the constraints? The main constraint of the problem is that $s_i = j$, where j denotes the number of occurrences of i in the sequence. But how do we actually encode this? There are a lot of moving parts with the varying i and j, so it seems like indicators/booleans would be helpful. Moreover, we define:

$$eq[i, j] = \begin{cases} 1 & s_i = j \\ 0 & \text{else} \end{cases} \qquad \text{for} \quad 0 \leq i \leq n, \quad 0 \leq j \leq n$$

That is, $eq[i, j]$ represents a **variable**, that our solver will determine the value of, but **it is our responsibility** ensure that the value given to the variable matches the definition. While these variables are helpful, they are not yet tied to our variables yet. That is, there is no way of our "solver" knowing that $s_i = j$. We do this by implementing a constraint. Remember, we want to implement the idea *"if* $eq[i, j] = 1$ *then* $s_i = j$*"*. How do we go about implementing an implication as a constraint?

3.2 Reification

To this point, we have only looked at constraints that involve variables. But what if we want to make constraints based on other constraints? What if we want to implement a constraint that is an implication? For that, we need **reification**.
As a motivating example, suppose we wanted to implement the idea that *"If* x *is equal to* 5*, then* y *must be greater than* 7*"*. Until now, we have no way of implementing this constraint.

Reification treats constraints as first-class citizens, and can accomplish our goal by introducing a new boolean variables b which is true **if and only if** constraint c holds ($b \iff c$). Essentially, we are "naming" the truth value of c with a variable b.

With respect to our motivating example, let's say we have an integer variable $x \in [0..10]$, and another variable y. We want to encode the constraint that "if $x = 5$ then $y > 7$". To do this, we introduce a boolean variable `is_x_five`, which is 1 when $x = 5$, and 0 otherwise.
Then, we tie together the boolean indicator with the constraint:

$$x == 5 \iff \texttt{is\_x\_five} = 1$$

We can then add further constraints using `is_x_five` such as $\texttt{is\_x\_five} = 1 \implies y > 7$
In code, the above would look like this:

```
is_x_five = model.NewBoolVar("is_x_five")

model.Add(x == 5).OnlyEnforceIf(is_x_five)
model.Add(x != 5).OnlyEnforceIf(is_x_five.Not())

model.add(y > 7).OnlyEnforceIf(is_x_five)
```

Implementation Note

OR-Tools uses **half-reification**. Instead of $b \iff c$, it supports $b \implies c$. Thus, to impose the desired logic, we can fully reify by adding both $b \implies c$ and $\overline{b} \implies \overline{c}$.

This idea of reification is done with:

```
constraint.OnlyEnforceIf(bool_var)
```

Unfortunately, OR-Tools is limited in the constraints that it supports for `OnlyEnforceIf`. It will only work for constraints that are:

- `Add`
- `AddBoolOr`
- `AddBoolAnd`
- `AddLinearExpressionInDomain`

3.3 Return to Magic Sequence

With reification, we can encode our constraints. That is, we need:

$$s_i = j \iff eq[i, j] = 1$$

We can do this in code via:

```
model.Add(S[i] == j).OnlyEnforceIf(eq[i, j])
model.Add(S[i] != j).OnlyEnforceIf(eq[i, j].Not())
```

where $eq[i, j]$ has been defined as a BoolVar, and s is an array of values s_i.

There is still one more constraint we need to impose: that s_i is actually equal to the number of occurrences of i in the sequence.
For this, we can consider: $\sum_{j=0}^{n} eq[j, i]$, and set s_i equal to that as a constraint.
In (pseudo)code:

```
for 0 <= i <= n:
    model.Add(
        s[i] == sum(eq[j, i] for 0 <= j <= n
    )
```

And this fully solves the problem

4 LinkedIn Queens Puzzle

The LinkedIn Queens Puzzle is described as follows:

- You are presented with an $n \times n$ board, with subdivided regions, as well as n queens
- Your job is to place all n queens on the board
- No two queens can be in the same row
- No two queens can be in the same column

- No two queens can be in the same region (equivalently, there must be exactly one queen in each region)
- Queens cannot touch one-step diagonally. That is, for example, if a queen is placed in row 4, column 5, a different queen **cannot** be placed in row 5 column 6, but it *could* be placed in row 6 column 7.

We will solve this using constraint programming!

4.1 Building the Solution

Step 1 of solving constraints is to define the variables. Remember, the variables are the quantities that *change*, whose values are determined by the constraint programming solver, and should indicate the solution to your problem.

In this case, a logical choice would be to have n variables that will take on values which indicate where each of the n queens should be placed on the grid (one variable for each queen).

We do this by maintaining a `row` and `column` variable for each queen. When we think of the values that these variables can take on, if we 0-index the rows and columns, we have $\texttt{row} \in [0..n-1]$ and $\texttt{column} \in [0..n-1]$

```
rows = [model.NewIntVar(0, n-1)]
columns = [model.NewIntVar(0, n-1)]
```

Step 2 involves implementing constraints.

The first two constraints we will implement "each queen must be in a different row" and "each queen must be in a different column". Since we have variables that take on values indicating the row value and column value of the queens, this is incredibly easy to implement:

```
model.AddAllDifferent(rows)
model.AddAllDifferent(columns)
```

The next constraint we turn to is that "queens cannot be one step diagonal from one another". This turns out to be a simple constraint to implement, just a bit ugly. We'll perform a bit of an unintuitive, "hacky" solution. Consider the following expression for two queens i, j:

$$|\texttt{row}[i] - \texttt{row}[j]| + |\texttt{col}[i] - \texttt{col}[j]| = 2$$

When does this equality hold? There are three possibilities:

- The row values of the two queens differ by 2, and the columns differ by 0.
- The column values of the two queens differ by 2, and the rows differ by 0
- The row values differ by 1 and the columns differ by 1

Well, the first possibility would indicate that the two queens are in the same column. And the second possibility indicates that the two queens are in the same row. But we already have constraints ensuring this will not happen! So, when this equality is met, with the other constraints in place, it must mean that the row values differ by 1 and the columns differ by 1, which implies diagonally adjacent queens. Thus, our goal is to ensure the **inequality** holds:

$$|\texttt{row}[i] - \texttt{row}[j]| + |\texttt{col}[i] - \texttt{col}[j]| \neq 2$$

In code:

```
for 0 <= i < j < n:
    abs_row_diff = model.NewIntVar(0, n-1)
    abs_col_diff = model.NewIntVar(0, n-1)

    model.AddAbsEquality(abs_row_diff, rows[i] - rows[j])
    model.AddAbsEquality(abs_col_diff, columns[i] - columns[j])

    model.Add(abs_row_diff + abs_col_diff != 2)
```

Then, for each pairs of queens, we ensure that the two respective cell numbers are not diagonal.

The actual code is omitted due to its ugly nature.

The last, most interesting, constraint is that queens cannot be in the same region. This is inherently different from ensuring the queens are in different rows and columns. The rows and columns constraints were easy, because our variables were defined with respect to the rows and columns. Here, the regions could be of any shape, and hence isn't as easy as ensuring $\texttt{row}[i] \neq \texttt{row}[j]$.

The idea is, for each cell in a region, maintain an indicator for if a queen is present in that cell. Then, group all cells of a region together, and ensure that **the sum** of the indicators for these cells is equal to 1. This constraint says that exactly one queen is in the region (exactly one indicator is equal to 1, the rest are 0). The question is: *"Is reification necessary?"* And the answer is: **yes**. Defining boolean variables (indicators) on their own *doesn't tell our solver about the position of our queens*. We need to **link** the boolean variables to our queen variables. (Pseudo)-code of implementation is as follows. We will assume that `regions` is a list of pairs, where each list represents a region, and pairs indicate the cells within that region.

```
for region_number, region in enumerate(regions):
    in_region = []
    for (row, col) in region:
        cell_var = model.NewBoolVar()
        in_region.append(cell_var)
        model.Add(rows[region_number] == row).OnlyEnforceIf(cell_var)
        model.Add(columns[region_number] == col).OnlyEnforceIf(cell_var)
    model.Add(sum(in_region) == 1)
```

5 Non-contiguous Domains

Recall that with Constraint Programming, our solver requires our variables to come from discrete, finite domains. For example, when we define the following variable:

```
x = model.NewIntVar(0, 10, 'x')
```

we are saying that $x \in [0..10]$. However, not all variables may come from a single integer interval. Perhaps, we instead wish to define a finite **set** of values that a variable can take on. Or perhaps our variable comes from *multiple, discontiguous intervals*. Both options are available to us.

To do this with CP-SAT, we use the method:

```
model.NewIntVarFromDomain(domain, name)
```

This requires us to define a domain. For this, you must use the "Domain Class" offered by CP-SAT a la:

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

The first domain yields the set of values $\{0, 2, 4, 6, 8\}$. The second domain yields the set of values $\{0, 1, 2, 6, 7, 8\}$.

5.1 Linear Expressions on Domains

Moreover, defining domain may be helpful beyond just defining the scope of a variable, but also the scope of an expression. We have the ability to enforce that the **result** of a linear expression must fall into a domain. Take the following example:

```
cp_model.AddLinearExpressionInDomain(
    x + y,
    cp_model.Domain.FromValues([0, 2, 4])
)
```

The above will ensure that it only considers values of x, y such that $(x + y) \in \{0, 2, 4\}$.

6 Shipping Allotments Problem

Consider the following problem: A shipping company has n ships with carrying capacity of 100 each. We have a bunch of shipments of varying sizes, and we want to load them onto the ships. However, we have a **goal** to maximize the number of ships which have at least 20 capacity unused (we might want that extra space in case of an emergency!). **NOTE:** The words "weight" and "size" will be used interchangeably.

They mean the same thing in this problem.

6.1 Attempt 1: Greedy

We may choose to attempt to solve this problem in a greedy fashion. Roughly, we may choose to sort the shipments in decreasing order of weights. Then, for each remaining shipment, allocate it to the least-filled ship. As an example, suppose we had four shipments with weights $45, 45, 40, 40$ and we had two ships.

Following our procedure, we would allocate 45 to the least filled ship. In this case, both ships are equally

unfilled, so we arbitrarily break ties, and give it to ship 1. The next 45 will be placed on ship 2. Then, for the first 40, we arbitrarily break the tie and assign it to ship 1, and the second 40 goes to ship 2.
By following the procedure, both ships have used 85 capacity, meaning that we have no ships with the desired 20 unused capacity.

Notice, however, that we could have instead allocated the two shipments of 45 to ship 1, and the remaining two shipments of 40 to ship 2. Here, we have one ship (ship two) with the 20 unused capacity. So it seems that this initial approach does not work.
Bonus: As an exercise to the reader: Does a dynamic programming solution exist for this problem? If so, what is its time complexity? If not, what is the bottleneck?

6.2 Attempt 2: Constraint Programming

We solve the problem via constraint programming. First, let us formalize the problem:

- We have s ships.
- Each ship has capcity of 100, and we are aiming to maximize the number of ships with 20 capacity free
- As input, we are given a list of pairs (w, c), where w denotes the weight of a shipment, and k denotes the quantity of that shipment. Shipments of the same weight can be distributed among ships.

First, we must define our variables. Remember, our variables are the quantities that change, and whose values should indicate the solution to our problem. Logically, we want to know the amount of shipments of each size that are on each ship. So, that is exactly what our variables will be:

$$n_{k,s} = \text{The number of shipments of size } k \text{ on ship } s$$

The lower bound for each of these variables is 0, but the upper bound depends on the quantity of shipments available. That is, for an arbitrary shipment w, we have that $n_{w,s} \leq c$, where c is the total amount of that shipment at the start.

Implementing this in (pseudo)code:

```
for (size, count) in shipments:
    for each ship s:
        n[size, s] = model.NewIntVar(0, count)
```

Now, we turn to the coveted constraints. Let's start with the easy ones:

Constraint 1: Each shipment must be on exactly one ship.
Suppose we have shipments (w, c), where w is the weight, and c is the quantity of that shipment. Then, the sum of the number of shipments of size w across all ships ($\sum_s n_{w,s}$) over all must be equal to c. In code:

```
for (size, count) in shipments:
    model.Add(
        sum(n[size, s] for s in SHIPS) == count
    )
```

Constraint 2: Next, we must ensure that the sum of the shipments on a single ship does not exceed the ship's capacity of 100. To do this, for each ship, count the number of identically weighted shipments on it, multiply it by the weight, and add them together to get the total weight. And ensure this is less than capacity:

```
for s in SHIPS:
    model.Add(
        sum(size * n[size, s] for (size, _) in shipments) <= SHIP_CAPACITY
    )
```

Constraint 3: Lastly, we aim to maximize the number of ships with 20 free capacity. For this, we turn to reification:

Define: `ship_free[s]` to be `TRUE` if and only if ship s has 20 free capacity. Then, for each ship, determine the load that it is carrying, and enforce:

$$\texttt{ship load} \leq (\texttt{SHIP CAPACITY}) - 20 \iff \texttt{ship\_free[s] = TRUE}$$

Lastly, we must add to the solver the aim of optimizing the number of ships with free capacity. Students should refer to the code on the website for the full implementation of everything above.

7 An Aside on Speed Up

CP-SAT allows us to adjust internals to speed up the search. However, this is often far less important than just having a good encoding. Regardless...

- CP-SAT allows us to run solver computation in parallel across multiple threads

- We can give the model a **hint** to try setting a variable to a specified value. For example, suppose you wanted to model to try setting $x = 5$ first. You can do this with `model.AddHint(x, 5)`

Generally, finding an optimal solution can take far longer than finding a feasible solution. This is because an optimal solution is a feasible solution, but a feasible solution is not always an optimal solution. In practice, we often don't really care about having the true optimal value, but rather are content with something "close enough".

With CP-SAT, if solving an optimization problem, you can interrupt the solver early, with a keyboard interrupt, and the solver will return a feasible solution instead of optimal.

However, what if we are solving a non-optimization problem? That is, what if a non-optimization problem is too hard to solve? We can't interrupt early for a "good enough" solution, because an intermediate solution is either feasible or it is not. But what if we were OK with a "not quite feasible" solution? What could this possibly mean?

8 Soft Constraints

Constraints like `Add(...)` are **hard constraints** in that they *must* be satisfied. **Soft constraints** are constraints that *can be violated*, but incurs a penalty. Thus, we can transform a feasibility problem into an optimization problem by *minimizing penalty*.

8.1 Example

Consider the classic *graph coloring* problem, where given a graph, we want to assign colors to its vertices so that vertices that share an edge are different colors.
Up until now, we would introduce a hard constraint of:

$$\forall (u, v) \in E, \quad \texttt{color}(u) \neq \texttt{color}(v)$$

We could instead introduce a soft constraint of:

$$\texttt{penalty} = \text{number of edges}(u, v) \text{ with } \texttt{color}(u) = \texttt{color}(v)$$

8.2 Optimization with Soft Constraints

Soft constraints are cool, but they add a very interesting challenge when it comes to optimization problems. Formally, suppose we are working on a problem which asks us to minimize some objective function o. We then choose to introduce soft constraints, which yield penalty p. How do we optimize?

Clearly, since our goal is to minimize o, we should do that, but we also likely want our penalty to be small. This can impose a very big challenge if o and p are inversely related.

It should make sense why such a scenario is likely – allowing a soft constraint to be violated likely improves our objective, but we incur an increase in penalty. That is, the objective function is decreasing while the penalty is increasing. How do we find the balance?

Idea 1: Minimize $o + p$

This doesn't work. As a toy counterexample: suppose we have an objective o representing the cost of some operation that we aim to minimize. And suppose there is penalty p incurred if we use more resources than a specified threshold.

- **Case A:** We use low resources, incurring a penalty of $p = 0$, but our cost is $o = 10$
- **Case B:** We use some resources, incurring penalty of $p = 1$, which in turn improves our cost to $o = 8$

By minimizing $o + p$, we would choose **Case B**, but this is problematic, as we are incurring a penalty. It is likely more desirable to not incur the penalty and go with **Case A**.

Idea 2: Avoid the interdependence of o and p by minimizing p first, and using o to break ties.

This idea works, and we can implement this by using a large constant M and set up the combined objective $Mp + o$. A suitable choice for M is $M = o_{\max} - o_{\min} + 1$