Lecture 10

Ishaan Lal

November 8, 2024

1 Introduction

It's been a while since we have discussed SAT. It is now time to revisit the topic. We mentioned briefly when introducing Constraint Programming that the solver writes constraints as clauses, and then solves a SAT formula. We'll get into this today!

2 Prerequisites

A thorough understanding of Constraint Program (lectures 8 and 9) as well as CDCL (lecture 4) is incredibly helpful in understanding the content of this lecture.

3 Setting the Stage

As a loose analogy to set the stage:

- **Constraint Programming** is akin to a *programming language*, where it is high level, and you are "telling the computer what you want".
- **CP-SAT** is akin to a *compiler*, a way of transferring written code to a language that the machine will understand.
- **The SAT Formula** is akin to *assembly language*, the low level language that the computer understands.
- **The SAT solver** is akin to the hardware, what actually carries out the computations.

3.1 Admitting a Lie

It turns out that this idea of CP-SAT just taking constraints and codifying them as clauses and handing it off to a SAT solver isn't completely accurate. Realistically, it is more of a "conversation" between CP-SAT and the solver. This is an active area of research, and many details are necessarily left out in the remainder of the notes. Any errors are mine.

3.2 Recall

To continue, we must recall some information from previous lectures.

First, recall CDCL: Conflict Driven Clause Learning. This was the algorithm we used for SAT solving. It incorporated unit propagation, as well as back*jumping*.

```
cdcl(φ):
    if unit_propagate() = CONFLICT: return UNSAT
    while not all variables have been set:
        let x = pick_variable()
        create new decision level; set x = T
        while unit_propagate() = CONFLICT:
            if level = 0: return UNSAT
            let (conflict_cls, assrt_lvl) = analyze_conflict()
            let φ = φ∪ { conflict_cls }
            # discard all assignments after asserting level
            backjump(assert_lvl)
    return SAT
```

Also, remember that as CDCL progresses, once a conflict is detected, **a new clause is added to the formula**.

Conflict analysis involved building an implication graph, finding a set of literals that caused the conflict, and adding a learned clause to the formula.

4 Finite Domain Solvers

Recall that the Constraint Programming solvers we have seen require **discrete, finite domains**. That is, our variables must be integers, and they are bounded.

To understand how they work, we first need to understand how traditional finite domain solvers work.

Finite domain solvers work by maintaining a domain $\mathcal{D}$ that tracks the possible values for each variable. Notably, $\mathcal{D}$ need not be contiguous.

Let $\min_{\mathcal{D}}(x)$ and $\max_{\mathcal{D}}(x)$ denote the minimum and maximum possible values for x in domain $\mathcal{D}$.

We say that a constraint c involving variables $x_1, ..., x_n$ is **bounds consistent** with domain $\mathcal{D}$ if for each x_i:

- when setting $x_i = \min_{\mathcal{D}}(x_i)$, we can find

  $$(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n) \in \mathcal{D}_1 \times \mathcal{D}_2 \times ... \times \mathcal{D}_{i-1} \times \mathcal{D}_{i+1} \times ... \times \mathcal{D}_n$$

  such that $(x_1, ..., x_n)$ satisfies the constraint

- when setting $x_i = \max_{\mathcal{D}}(x_i)$, we can find

  $$(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n) \in \mathcal{D}_1 \times \mathcal{D}_2 \times ... \times \mathcal{D}_{i-1} \times \mathcal{D}_{i+1} \times ... \times \mathcal{D}_n$$

  such that $(x_1, ..., x_n)$ satisfies the constraint

To test bounds consistency, choose one variable, and set it to one of the bounds of its domain. Then, see if it is possible for the remaining variables to fall within their domain and follow the constraint.

As an example, let $\mathcal{D}(x) = [4..7], \mathcal{D}(y) = [1..5], \mathcal{D}(z) = [-1..2]$ with constraint $x = y + z$.

- Start with variable x, and set $x = \min_{\mathcal{D}(x)}(x) = 4$. Then, we can have $y = 4, z = 0$, and we have $x \in \mathcal{D}_x, y \in \mathcal{D}_y, z \in \mathcal{D}_z$ and it satisfies $x = y + z$
- Now let $x = \max_{\mathcal{D}(x)}(x) = 7$. With $y = 5, z = 2$, this checks out.
- Moving to y, let $y = \min_{\mathcal{D}(y)}(y) = 1$. Here, regardless of our choice for x, we will never satisfy the constrain. Thus, the constraint is **not bounds consistent** with domains $\mathcal{D}_x, \mathcal{D}_y, \mathcal{D}_z$

4.1 Propagators

We say that a **propagator** for constraint c is an algorithm that accepts a domain $\mathcal{D}$, and returns:

- a new domain $\mathcal{D}'$ where c is bounds consistent with $\mathcal{D}'$
- Implications "explaining" the updated bounds in $\mathcal{D}'$

Different constraints have different propagation rules for finding $\mathcal{D}'$

Example

Recall our example of constraint $x = y + z$. How do we ensure bounds consistency for it?

We can rewrite to isolate each variables:

$$x = y + z \qquad y = x - z \qquad z = x - y$$

Now, derive a pair of inequalities for each:

$$\begin{aligned} x &\geq \min_{\mathcal{D}}(y) + \min_{\mathcal{D}}(z) \quad \text{and} \quad x \leq \max_{\mathcal{D}}(y) + \max_{\mathcal{D}}(z) \\ y &\geq \min_{\mathcal{D}}(x) - \max_{\mathcal{D}}(z) \quad \text{and} \quad y \leq \max_{\mathcal{D}}(x) - \min_{\mathcal{D}}(z) \\ z &\geq \min_{\mathcal{D}}(x) - \max_{\mathcal{D}}(y) \quad \text{and} \quad z \leq \max_{\mathcal{D}}(x) - \min_{\mathcal{D}}(y) \end{aligned}$$

We can then tighten the upper and lower bounds accordingly to get $\mathcal{D}'$
As an example, since $y \geq \min_{\mathcal{D}}(x) - \max_{\mathcal{D}}(z) = 4 - 2$, we update the original domain of y from $\mathcal{D}_y = [1..5]$ to $\mathcal{D}'_y = [2..5]$.

With a domain update, we tie it with an explanation. For the above, the explanation would be:

$$(x \geq 4) \wedge (z \leq 2) \implies y \geq 2$$

Many traditional Constraint Programming solvers use **finite domain propagation**, whereby you start with the initial domain $\mathcal{D}_0$ specified by the user. Then, try adding a new constraint (e.g. assigning a variable). Then, repeatedly run all constraint propagators on $\mathcal{D}$ until: 1) a variable has no possible values, where you must backtrack, and add $\neg c$, or 2) nothing changes, where you add another constraint and repeat.

An example walk-through of Finite domain propagation in action is available on the lecture slides.

You might recognize that this procedure feels very similar to DPLL – and it is! To highlight this comparison, adding a constraint in FD is similar to making a decision in DPLL. Running constraint propagators is akin to unit propagation. Backtracking is like... well, backtracking!

If there are so many similarities, then why don't we just try to do this entire procedure in SAT?

Issue: What are the boolean variables?

After all, if we are trying to represent a problem as SAT, we need to define our variables, and by definition, SAT variables are boolean.

Attempt 1

Our first attempt at defining boolean variables is as follows:
for each CP variable x, create boolean variables $[[x = i]]$ for $lb(x) \leq i \leq ub(x)$

What are the consequences of this definition? First, the number of variables is linear with respect to the size of the domain. Secondly, and more glaringly, we need very long clauses to represent inequalities (e.g. $x \leq 10$ for $lb(x) = 0$ and $ub(x) = 20$ would require 11 variables in the clause. As a result, unit propagation would be harder to come by.

Attempt 2

Our second attempt is a logarithmic encoding. The idea behind this is to encode a boolean variable for each bit of x.

This fixes the issue of the number of variables we have (now logarithmic instead of linear), but it turns out to have even worse propagation strength.

4.2 Order Encoding

Attempt 3 is called **Order Encoding**. For each CP variable x, we will create the following boolean variables:

$$[[x = i]] \qquad \text{for } lb(x) \leq i \leq ub(x)$$
$$[[x \leq i]] \qquad \text{for } lb(x) \leq i \leq ub(x)$$

Note that $x \geq i \equiv \neg[[x \leq i - 1]]$ and $x \neq i \equiv \neg[[x = i]]$.

Tied with these variables are consistency clauses: logical implications that must hold for our variables to actually make sense and reflect their meaning:

$$[[x \leq i]] \implies [[x \leq i + 1]] \qquad \text{for } lb(x) \leq i \leq ub(x) - 1$$
$$[[x = i]] \iff [[x \leq i]] \wedge \neg[[x \leq i - 1]]$$

See that the amount of variables we need is linear with respect to the size of the domain, however, the propagation strength is much better (that is, it is more likely for us to be able to propagate).

With these variables, let us try to write a constraint. Suppose we wanted to write the constraint $x = y + z$ with clauses. This turns out to be a bit more tedious than expected, because this must be enforced no matter where in their respective domains the values x, y, z come from. Further, if two variables are known, they together determine the third value.

Thus, for each $lb \leq i, j \leq ub$, we would add the clauses:

$$[[y = i]] \wedge [[z = j]] \implies [[x = i + j]]$$
$$[[x = i]] \wedge [[z = j]] \implies [[y = i - j]]$$
$$[[x = i]] \wedge [[y = j]] \implies [[z = i - j]]$$

In this particular example, the amount of clauses we are adding is $\mathcal{O}(|ub - lb|^2)$. Not good! And if our constraint added more variables, the amount of clauses would undergo an exponential blowup!

5 Lazy Clause Generation

A key observation is as follows: even though it takes *a lot* of clauses to represent a CP constraint, most of the clauses are never used. **Lazy Clause Generation** utilizes this observation, and rather than generate all of these clauses before solving the CP, it will "just generate the ones we need when we need them!"

That seems a bit abstract and almost too good to be true. Let's see how it works.

When we introduced Finite Domain propagators, we said that they return an "explanation" for updating bounds, e.g. $(x \geq 4) \wedge (z \leq 2) \implies y \geq 2$. Thankfully, it is pretty easy to express these explanations as clauses. So we will run propagators during execution of a CDCL solver, than add *explanation clauses* to the formula. If we only introduce explanation clauses when the Left Hand Side of the implication is currently true, they will immediately become unit clauses!

Below is the pseudocode for Lazy Clause Generation:

```
lazy_clause_generation(constraint_program):
    let P = make_propagators(constraint_program)
    if lcg_propagate() = CONFLICT: return INFEASIBLE
    while not all variables have been set:
        let x = pick_variable()
        create new decision level; set x = T
        while lcg_propagate(P, φ) = CONFLICT:
            if level = 0: return INFEASIBLE
            let (cls, lvl) = analyze_conflict()
            let φ = φ∪ { cls }
            backjump(lvl)
    return FEASIBLE

lcg_propagate(P, φ):
    while True:
        if unit_prop() = CONFLICT:
            return CONFLICT
        for propagator p ∈ P:
            let explanation_clauses = p(φ)
            let φ = φ∪ explanation_clauses
        if φ did not change:
            return SUCCESS
```

Please refer to the lecture slides for a walk-through example of LCG in action.