Chapter 7
Specialized Theory Solvers

In the previous chapter, we discussed how to extend SAT solvers to first-order theories by invoking a black-box theory solver. In this chapter, we study the *Simplex algorithm* for solving conjunctions of literals in linear real arithmetic. Then, we extend the solver to natively handle rectified linear units (ReLUs), which would normally be encoded as disjunctions and thus dealt with using the SAT solver.

7.1 Theory Solving and Normal Forms

The Problem

The theory solver for LRA receives a formula F as a conjunction of linear inequalities:

$$\bigwedge_{i=1}^{n} \left(\sum_{j=1}^{m} c_{ij} \cdot x_j \leqslant b_i \right)$$

where $c_{ij}, b_i \in \mathbb{R}$. The goal is to check satisfiability of F, and, if satisfiable, discover a model $I \models F$.

Notice that our formulas do not have strict inequalities ($<$). The approach we present here can be easily generalized to handle strict inequalities, but for simplicity, we stick with inequalities.[1]

[1]See Dutertre and De Moura (2006) for how to handle strict inequalities. In most instances of verifying properties of neural networks, we do not need strict inequalities to encode network semantics or properties.

Simplex Form

The Simplex algorithm, discussed in the next section, expects formulas to be in a certain form (just like how DPLL expected propositional formulas to be in CNF). Specifically, Simplex expects formulas to be conjunctions of *equalities* of the form

$$\sum_{i=1} c_i \cdot x_i = 0$$

and *bounds* of the form

$$l_i \leqslant x_i \leqslant u_i$$

where $u_i, l_i \in \mathbb{R} \cup \{\infty, -\infty\}$.

Therefore, given a formula F, we need to translate it into an equivalent formula of the above form, which we will call the *Simplex form*. It turns out that translating a formula into Simplex form is pretty simple. Say

$$F \triangleq \bigwedge_{i=1}^{n} \left(\sum_{j=1}^{m} c_{ij} \cdot x_j \leqslant b_i \right)$$

Then, we take every inequality and translate it into two conjuncts, an equality and a bound. From the ith inequality,

$$\sum_{j=1}^{m} c_{ij} \cdot x_j \leqslant b_i$$

we generate the equality

$$s_i = \sum_{j=1}^{m} c_{ij} \cdot x_j$$

and the bound

$$s_i \leqslant b_i$$

where s_i is a new variable, called a *slack variable*.

Example 7.A Consider the formula F, which we will use as our running example:

$$\begin{aligned} x + y &\geqslant 0 \\ -2x + y &\geqslant 2 \\ -10x + y &\geqslant -5 \end{aligned}$$

For clarity, we will drop the conjunction operator and simply list the inequalities. We convert F into a formula F_s in Simplex form:

$$\begin{aligned} s_1 &= x + y \\ s_2 &= -2x + y \\ s_3 &= -10x + y \\ s_1 &\geqslant 0 \\ s_2 &\geqslant 2 \\ s_3 &\geqslant -5 \end{aligned}$$

■

This transformation is a simple rewriting of the original formula that maintains satisfiability, as formalized in the following theorem:

Theorem 7.A Let F_s be the Simplex form of some formula F.

1. If $I_s \models F_s$, then $I_s \models F$.
2. If $I \models F$, then $I' \models F_s$, where

$$I' = I \cup \left\{ s_i \mapsto \sum_{j=1}^{n} c_{ij} \cdot I(x_j) \right\}_{i=1}^{m}$$

■

The second case is a bit more involved, since F_s has more variables that F: m slack variables, one for each inequality in F. So any model of F need be extended with assignments to slack variables.

7.2 The Simplex Algorithm

We are now ready to present the Simplex algorithm. This is a very old idea, due to George Dantizg, who developed it in 1947 (Dantzig, 1990). The goal of the algorithm is to find a satisfying assignment that maximizes some objective function. Our interest in verification is typically to find *any* satisfying assignment, and so the algorithm we will present is a subset of Simplex.

Intuition

One can think of the Simplex algorithm as a procedure that simultaneously looks for a model and a proof of unsatisfiability. It starts with some interepretation, and continues to update it in every iteration, until it finds a model or discovers a proof of unsatisfiability. We start from the interpretation I that sets all variables to 0. This assignment satisfies all the equalities, but may not satisfy the bounds. In every iteration of Simplex, we pick a bound that is not satisfied, and we modify I to satisfy it, or we discover the the formula is unsatisfiable. Let us see this process pictorially on a satisfiable example before we dig into the math.

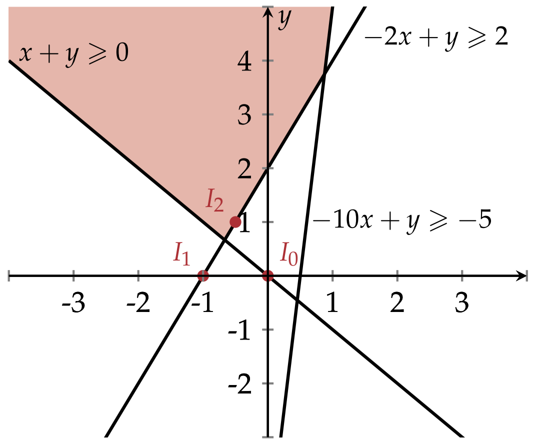

Figure 7.1 Simplex example

Example 7.B Recall the formula F from our running example, illustrated in Figure 7.1, where the satisfying assignments are shaded.

$$\begin{aligned} x + y &\geqslant 0 \\ -2x + y &\geqslant 2 \\ -10x + y &\geqslant -5 \end{aligned}$$

Simplex begins with the initial interpretation

$$I_0 = \{x \mapsto 0, y \mapsto 0\}$$

shown in Figure 7.1, which is not a model of the formula.

Simplex notices that $I_0 \not\models -2x + y \geqslant 2$, and so it decreases the interpretation of x to -1, resulting in I_1. Then, it notices that $I_1 \not\models x + y \geqslant 0$, and so it increases the interpretation of y to 1, resulting in the satisfying assignment I_2. (Notice that x also changes in I_2; we will see why shortly.) In a sense, Simplex plays Whac-A-Mole, trying to satisfy one inequality only to break another, until it arrives at an assignment that satisfies all inequalities. Luckily, the algorithm actually terminates. ■

Basic and non-basic variables

Recall that Simplex expects an input formula to be in Simplex form. The set of variables in the formula are broken into two subsets:

Basic variables are those that appear on the left hand side of an equality; initially, these are the slack variables.

Non-basic variables are all other variables.

As Simplex progresses, it will rewrite formulas, thus some basic variables will become non-basic and vice versa.

Example 7.C In our running example, initially the set of basic variables is $\{s_1, s_2, s_3\}$ and non-basic variables is $\{x, y\}$. ■

To ensure termination of Simplex, we will fix a total ordering on the set of all (basic and non-basic) variables. So, when we say *"the first variable that..."*, we are referring to the first variable per our ordering. To easily refer to variables, we will assume they are of the form $x_1, \ldots, x_n$. Given a basic variable x_i and a non-basic variable x_j, we will use c_{ij} to denote the coefficient of x_j in the equality

$$x_i = \ldots + c_{ij} \cdot x_j + \ldots$$

For a variable x_i, we will use l_i and u_i to denote its lower bound and upper bound, respectively. If a variable does not have an upper bound (resp. lower bound), its upper bound is be ∞ (resp. $-\infty$). Note that non-slack variables have no bounds.

Simplex in detail

We are now equipped to present the Simplex algorithm, shown in Algorithm 1. The algorithm maintains the following two invariants:

1. The interpretation I always satisfies the equalities, so only the bounds may be violated. This is initially true, as I assigns all variables to 0.

2. The bounds of non-basic variables are all satisfied. This is initially true, as non-basic variables have no bounds.

In every iteration of the while loop, the algorithm looks for a basic variable whose bounds are not satisfied, and attempts to fix it. There two symmetric cases, encoded as two branches of the *if* statement, $x_i < l_i$ or $x_i > u_i$. Let us consider the first case. Since x_i is below l_i, we need to increase its assignment in I. We do this indirectly, by modifying the assignment of a non-basic variable x_j. But which x_j should we pick? In principle, we can pick any x_j such that the coefficient $c_{ij} \neq 0$, and adjust the interpretation of x_j accordingly. But, if you look at the algorithm, there are a few extra conditions. If we cannot find an x_j that satisfies these conditions, then the problem is UNSAT. We will discuss the unsatisfiability conditions shortly. For now, assume we have found an x_j. We can increase its current interpretation by $\frac{l_i - I(x_i)}{c_{ij}}$; this makes the interpretation of x_i increase by $l_i - I(x_i)$, thus barely satisfying the lower bound $I(x_i) = l_i$. Note that the assignments of basic variables are assumed to change automatically when we change the interpretation of non-basic variables. This maintains the first invariant of the algorithm.[2]

After we have updated the interpretation of x_j, there is a chance that we have violated one of the bounds of x_j. Therefore, we rewrite the formulas such that x_j becomes a basic variable and x_i a non-basic variable. This is known as the *pivot* operation, and it is mechanically done as follows: Take the following equality, where N is the set of indices of non-basic variables:

$$x_i = \sum_{k \in N} c_{ik} x_k$$

[2]Basic variables are sometimes called *dependent* variables and non-basic variables *independent* variables, indicating that the assignments of basic variables depend on those of non-basic variables.

Algorithm 1: Simplex

```
Data: A formula $F$ in Simplex form
Result: $I \models F$ or UNSAT

Let $I$ be the interpretation that sets all variables $fv(F)$ to 0
while true do
    if $I \models F$ then return $I$
    Let $x_i$ be the first basic variable s.t. $x_i < l_i$ or $x_i > u_i$
    if $x_i < l_i$ then
        Let $x_j$ be the first non-basic variable s.t.
            $(I(x_j) < u_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) > l_j \text{ and } c_{ij} < 0)$
        if If no such $x_j$ exists then return UNSAT
        $I(x_j) \leftarrow I(x_j) + \frac{l_i - I(x_i)}{c_{ij}}$
    else
        Let $x_j$ be the first non-basic variable s.t.
            $(I(x_j) > l_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) < u_j \text{ and } c_{ij} < 0)$
        if If no such $x_j$ exists then return UNSAT
        $I(x_j) \leftarrow I(x_j) + \frac{u_i - I(x_i)}{c_{ij}}$
    Pivot $x_i$ and $x_j$
```

and rewrite it by moving x_j to the left-hand side:

$$x_j = \underbrace{-\frac{x_i}{c_{ij}} + \sum_{k \in N \setminus \{j\}} \frac{c_{ik}}{c_{ij}} x_k}_{\text{replace } x_j \text{ with this}}$$

Now, replace x_j in all other equalities with the expression above. This operation results in a set of equalities where x_j only appears once, on the left-hand side. And so after pivoting it x_j becomes a basic variable and x_i a non-basic one.

Example 7.D Let us now work through our running example in detail. Recall, our formula is:

$$\begin{aligned}
s_1 &= x + y \\
s_2 &= -2x + y \\
s_3 &= -10x + y \\
s_1 &\geqslant 0 \\
s_2 &\geqslant 2 \\
s_3 &\geqslant -5
\end{aligned}$$

Say the variables are ordered as follows:

$$x, y, s_1, s_2, s_3$$

Initially, the bounds of s_1 and s_3 are satisfied, but s_2 is violated, because $s_2 \geqslant 2$ but $I_0(s_2) = 0$, as all variables are assigned 0.

First iteration In the first iteration, we pick the variable x to fix the bounds of s_2, as it is the first one in our ordering. Note that x is unbounded (i.e., its bounds are $-\infty$ and ∞); so it easily satisfies the conditions. To increase the interpretation of s_2 to 2, and satisfy its lower bound, we can decrease $I_0(x)$ to -1, resulting in the following satisfying assignment:

$$I_1 = \{x \mapsto -1, y \mapsto 0, s_1 \mapsto -1, s_2 \mapsto 2, s_3 \mapsto 10\}$$

We now pivot s_2 and x, producing the following set of equalities (the bounds always remain the same):

$$\begin{aligned}
x &= 0.5y - 0.5s_2 \\
s_1 &= 1.5y - 0.5s_2 \\
s_3 &= -4y + 5s_2
\end{aligned}$$

Second iteration The only basic variable not satisfying its bounds is now s_1, since $I_1(s_1) = -1 < 0$. The first non-basic variable that we can tweak is y. We can increase the value of $I(y)$ by 1, resulting in the following interpretation:

$$I_2 = \{x \mapsto -0.5, y \mapsto 1, s_1 \mapsto 0.5, s_2 \mapsto 2, s_3 \mapsto 6\}$$

At this point, we pivot y with s_1.

Third iteration Simplex terminates since $I_2 \models F$.

■

Why is Simplex Correct?

First, you may wonder, why does Simplex terminate? The answer is due to the fact that we order variables and always look for the *first* variable violating bounds. This is known as *Bland's rule* (Bland, 1977). It ensures that we never revisit the same set of basic and non-basic variables.

Second, you may wonder, is Simplex actually correct? If Simplex returns an interpretation I, it is easy to see that $I \models F$, since Simplex checks that condition before it terminates. But what about the case when it says UNSAT? To illustrate correctness in this setting, we will look at an example, and refer the reader to Dutertre and De Moura (2006) for a full proof.

Example 7.E Consider the following formula in Simplex form:

$$\begin{aligned} s_1 &= x + y \\ s_2 &= -x - 2y \\ s_3 &= -x + y \\ s_1 &\geqslant 0 \\ s_2 &\geqslant 2 \\ s_3 &\geqslant 1 \end{aligned}$$

This formula is UNSAT—use your favorite SMT solver to check this. Imagine an execution of Simplex that performs the following two pivot operations: (1) s_1 with x, and (2) s_2 with y.

The first pivot results in the following formula:

$$\begin{aligned} x &= s_1 - y \\ s_2 &= -s_1 - y \\ s_3 &= -s_1 + 2y \end{aligned}$$

The second pivot results in the following formula:

$$\begin{aligned} x &= 2s_1 + s_2 \\ y &= -s_2 - s_1 \\ s_3 &= -3s_1 - 2s_s \end{aligned}$$

The algorithm maintains the invariant that all non-basic variables satisfy their bounds. So we have $s_1, s_2 \geqslant 0$. Say s_3 violates its bound, i.e.,

$$-3s_1 - 2s_2 < 1$$

The only way to fix this is by decreasing the interpretations of s_2 and s_3. But even if we assign s_2 and s_3 the value 0, we cannot make $s_3 \geqslant 1$. So Simplex figures out the the formula is UNSAT. The conditions for choosing variable x_j in Algorithm 1 encode this argument. ■

7.3 The Reluplex Algorithm

Using the Simplex algorithm as the theory solver within DPLL(T) allows us to solve formulas in LRA. So, at this point in our development, we know how to algorithmically reason about neural networks with piecewise-linear activations, like ReLUs. Unfortunately, this approach has been shown to not scale to large networks. One of the reasons is that ReLUs are encoded as disjunctions, as we saw in Chapter 5. This means that the SAT-solving part of DPLL(T) will handle the disjunctions, and may end up considering every possible case of the disjunction—ReLU begin active (output is 0) or active (output = input)—leading to many calls to Simplex, exponential in the number of ReLUs.

To fix those issues, the work of Katz et al. (2017) developed an extension of Simplex, called *Reluplex*, that natively handles ReLU constraints in addition to linear inequalities. The key idea is to try to *delay* case splitting on ReLUs. In the worst case, Reluplex may end up with an exponential explosion, just like DPLL(T) with Simplex, but empirically it has been shown to be a promising approach for scaling SMT solving to larger neural networks. In what follows, we present the Reluplex algorithm.

Reluplex form

Just like with Simplex, Reluplex expects formulas to be in a certain form. We will call this form *Reluplex form*, where formulas contain (1) equalities (same as Simplex), (2) bounds (same as Simplex), and (3) *ReLU constraints* of the form

$$x_i = \mathrm{relu}(x_j)$$

Given a conjunction of inequalities and ReLU constraints, we can translate them into Reluplex form by translating the inequalities into Simplex form. Additionally, for each ReLU constraint $x_i = \mathrm{relu}(x_j)$, we can add the bound $x_i \geqslant 0$, which is implied by the definition of a ReLU.

Example 7.F Consider the following formula:

$$\begin{aligned} x + y &\geqslant 2 \\ y &= \mathrm{relu}(x) \end{aligned}$$

We translate it into the following Reluplex form:

$$\begin{aligned} s_1 &= x + y \\ y &= \mathrm{relu}(x) \\ s_1 &\geqslant 2 \\ y &\geqslant 0 \end{aligned}$$

■

Reluplex in detail

We now present the Reluplex algorithm. The original presentation by Katz et al. (2017) is a set of rules that can be applied non-deterministically to arrive at an answer. Here, we present a specific schedule of the Reluplex algorithm.

The key idea of Reluplex is to call Simplex on equalities and bounds, and then try to massage the interpretation returned by Simplex to satisfy all ReLU constraints. Reluplex is shown in Algorithm 2.

Initially, Simplex is invoked on the formula F', which is the original formula F but without the ReLU constraints. If Simplex returns UNSAT, then

Algorithm 2: Reluplex

```
Data: A formula $F$ in Reluplex form
Result: $I \models F$ or UNSAT

Let $I$ be the interpretation that sets all variables $fv(F)$ to 0
Let $F'$ be the non-ReLU constraints of $F$
while true do
    ▷ Calling Simplex (note that we supply Simplex with a reference
    to the initial interpretation and it can modify it)
    $r \leftarrow \text{Simplex}(F', I)$
    if r is UNSAT then
        return UNSAT
    else
        if r is an interpretation and $r \models F$ then return $r$

    ▷ Handle violated ReLU constraint
    Let ReLU constraint $x_i = \text{relu}(x_j)$ be s.t. $I(x_i) \neq \text{relu}(I(x_j))$
    if $x_i$ is basic then
        pivot $x_i$ with non-basic variable $x_k$, where $k \neq j$ and $c_{ik} \neq 0$
    if $x_j$ is basic then
        pivot $x_j$ with non-basic variable $x_k$, where $k \neq i$ and $c_{jk} \neq 0$
    Perform one of the following operations:

        $I(x_i) \leftarrow \text{relu}(I(x_j))$   or   $I(x_j) \leftarrow I(x_i)$

    ▷ Case splitting (ensures termination)
    if $u_j > 0$, $l_j < 0$, and $x_i = relu(x_j)$ considered more than $\tau$ times then
        $r_1 \leftarrow \text{Reluplex}(F \wedge x_j \geqslant 0 \wedge x_i = x_j)$
        $r_2 \leftarrow \text{Reluplex}(F \wedge x_j \leqslant 0 \wedge x_i = 0)$
        if $r_1 = r_2 = UNSAT$ then return UNSAT
        if $r_1 \neq UNSAT$ then return $r_1$
        return $r_2$
```

we know that F is UNSAT, this is because $F \Rightarrow F'$ is valid. Otherwise, if Simplex returns a model $I \models F'$, it may not be the case that $I \models F$, since F' is a weaker (less constrained) formula.

If $I \not\models F$, then we know that one of the ReLU constraints is not satisfied. We pick one of the violated ReLU constraints $x_i = \text{relu}(x_j)$ and modify I to make sure it is not violated. Note that if any of x_i and x_j is a basic variable, we pivot it with a non-basic variables. This is because we want to modify the interpretation of one of x_i or x_j, which may affect the interpretation of the other variable if it is a basic variable and $c_{ij} \neq 0$.[3] Finally, we modify the interpretation of x_i or x_j, ensuring that $I \models x_i = \text{relu}(x_j)$. Note that the choice of x_i or x_j is up to the implementation.

The problem is that fixing a ReLU constraint may end up violating a bound, and so Simplex need be invoked again. We assume that the interpretation I in Reluplex is the same one that is modified by invocations of Simplex.

Case splitting

Note that if we simply apply Reluplex without the last piece of the algorithm—case splitting—it may not terminate. Specifically, it may get into a loop where Simplex satisfies all bounds but violates a ReLU, and then satisfying the ReLU causes a bound to be violated, and on and on.

The last piece of Reluplex checks if we are getting into an infinite loop, by ensuring we do not attempt to fix a ReLU constraint more that τ times. If this threshold is exceeded, then the ReLU constraint $x_i = \text{relu}(x_j)$ is split into its two cases:

$$F_1 \triangleq x_j \geqslant 0 \land x_i = x_j$$

and

$$F_2 \triangleq x_j \leqslant 0 \land x_i = 0$$

and Reluplex is invoked recursively on two instances of the problem, $F \land F_1$ and $F \land F_2$. If both instances are UNSAT, then the F is UNSAT. If any of the instances is SAT, then F is SAT. This is due to the fact that

$$F \equiv (F \land F_1) \lor (F \land F_2)$$

[3]These conditions are not explicit in Katz et al. (2017), but their absence may lead to wasted iterations (or Update rules in Katz et al. (2017)) that do not fix violations of ReLU constraints.

Looking Ahead

We are done with constraint-based verification. In the next part of the book, we will look at different approaches that are more efficient at the expense of failing to provide proofs in some cases.