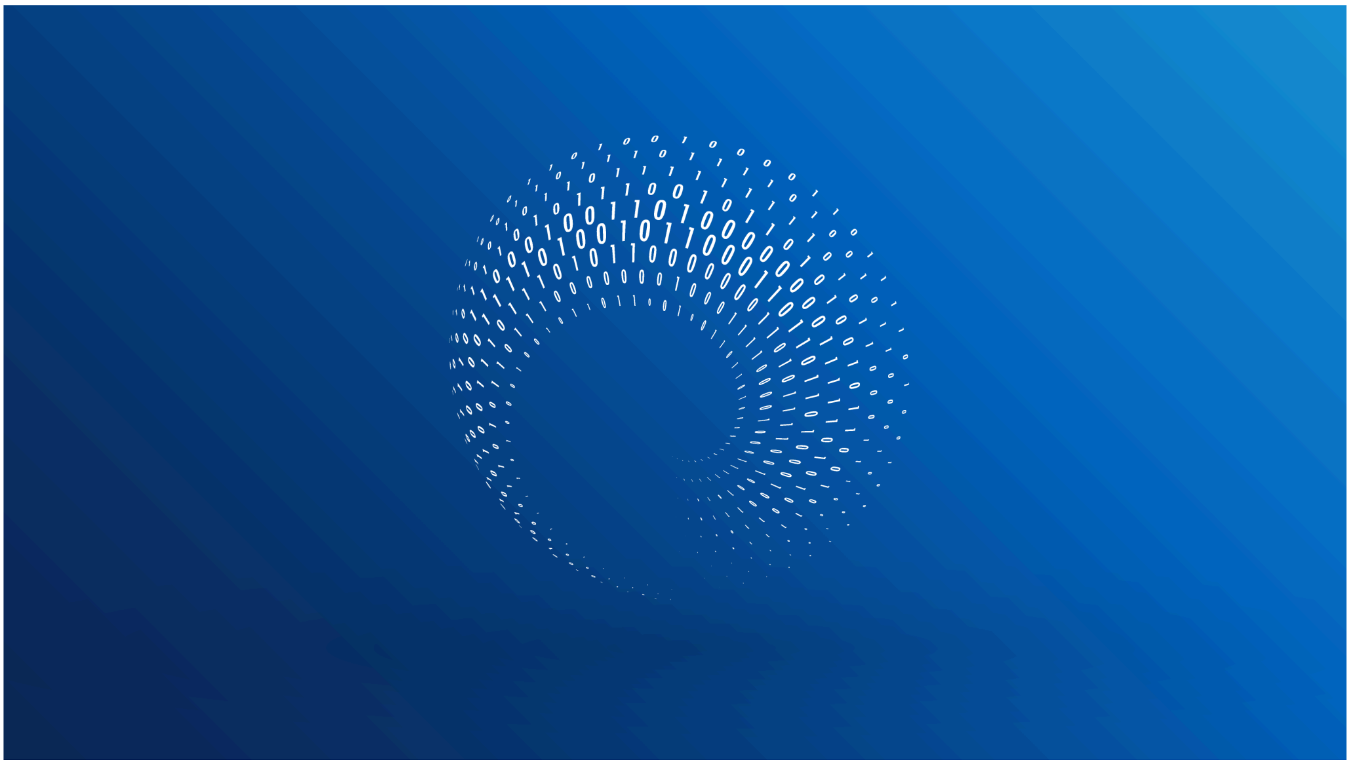
Tutorials

Tutorial #10: SAT Solvers II: Algorithms

Dec. 09, 2020

S. Prince, C. Srinivasa



In part I of this tutorial, we introduced the SAT problem and discussed its applications. The SAT problem operates on Boolean logical formulae and we discussed how to convert these to conjunctive normal form. Here, a set of clauses are logical ANDed together. Each clause logically ORs a set of literals (variables and their complements). For example:

$$\phi := (x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2 \lor x_3) \land (x_1 \lor x_2 \lor \overline{x}_3). \tag{1}$$

In this tutorial we focus exclusively on the SAT solver algorithms that are applied to this problem. We'll start by introducing two ways to manipulate Boolean logic formulae. We'll then exploit these manipulations to develop algorithms of increasing complexity. We'll conclude with an introduction to *conflict-driven clause learning* which underpins most modern SAT solvers.

Operations on Boolean formulae

SAT solvers rely on repeated algebraic manipulation of the formula that we wish to test for satisfiability. Two such manipulations are *conditioning* and *resolution*. In this section we will discuss each in turn.

Conditioning

In conditioning, we set a variable x_i to a concrete value (i.e., true or false). When we set x_i to true, we can simplify the formula using two rules:

1. All clauses containing x_i can be removed from the formula. These clauses are now satisfied.
2. Any terms $\overline{x}_i$ in the remaining clauses can be removed. These must now evaluate to false and hence cannot be used to satisfy their clauses.

For example, consider the formula:

$$\phi := (x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2 \lor x_3) \land (x_1 \lor x_2 \lor \overline{x}_3). \tag{2}$$

When we set $x_1 = \text{true}$, this becomes

$$\phi \land x_1 := (x_2 \lor x_3). \tag{3}$$

where the first, third and fourth clause have been removed as they are now satisfied (by rule 1) and the term $\overline{x}_1$ has been removed from the second clause as this term is now false (by rule 2).

Similarly, when we condition by setting a variable to fals
do any terms x_i in the remaining clauses. Setting x_1 t

Note that variable x_i must be either true or false and so:

$$\begin{aligned}\phi &= (\phi \wedge x_i) \vee (\phi \wedge \overline{x}_i)\\ &= (x_2 \vee x_3) \vee ((x_2 \vee x_3) \wedge (\overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3)).\end{aligned} \tag{5}$$

Here we apply the conditioning operation twice and the result is to remove the variable x_i from the formula ϕ to yield two simpler formulae, $(\phi \wedge x_i)$ and $(\phi \wedge \overline{x}_i)$ which are logically ORed together. Note though, that the result is not in conjunctive normal form.

Resolution

The second common operation applied to Boolean formulae is *resolution*. Consider two clauses c_1 and c_2 where $x_i \in c_1$ and $\overline{x}_i \in c_2$. When we resolve by x_i, we replace these two clauses with a single clause $(c_1 \setminus x_i) \vee (c_2 \setminus \overline{x}_i)$. This clause is known as the *resolvent* and contains the remaining terms in c_1 and c_2 after x_i and $\overline{x}_i$ are removed.

This is best illustrated with an example. Consider the formula:

$$\phi := (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_4) \wedge (x_2 \vee x_4 \vee x_5). \tag{6}$$

We note that x_2 is in the first clause and $\overline{x}_2$ is in the second clause and so we can resolve with respect to x_2 by combining the remaining terms from the first and second clause:

$$\phi := (x_1 \vee \overline{x}_3 \vee x_4) \wedge (x_2 \vee x_4 \vee x_5). \tag{7}$$

Note that the third clause is unaffected by this operation.

The underlying logic is as follows. If x_2 is false, then for the first clause to be satisfied we must have $x_1 \vee \overline{x}_3$. However, if x_2 is true, then for the second clause to be satisfied, we must have x_4. Since either x_2 or $\overline{x}_2$ must be the case, it follows that w resolving with respect to is a unit clause (i.e., only contains a single literal). For example,

$$\phi := (x_1 \lor \overline{x}_3 \lor \overline{x}_4) \land x_4 \tag{8}$$

Resolution between these two clauses works as normal. However, we can go further. Since we know that x_4 must be $\texttt{true}$ from the second clause, the effect of resolution here is the same as conditioning. We can remove *all* clauses containing x_4 and remove all terms $\overline{x}_4$ from the remaining clauses. So unit resolution can be seen as either a special case of resolution or as a conditioning operation depending how you look at it.

Unit propagation

A unit resolution operation may create more unit clauses. In this case, we can repeatedly apply unit resolution to the expression and at each stage we eliminate one of the variables from consideration. This procedure is known as *unit propagation*.

SAT solving algorithms based on resolution

We now present a series of learning algorithms that use conditioning and resolution to solve the satisfiability problem. In this section, we will use resolution to solve the 2-SAT problem and show why this can be solved in polynomial time. Then we'll introduce the directional resolution algorithm which uses resolution to solve 3-SAT problems and above, but we'll see that this becomes more computationally complex. In the next section, we'll move to algorithms that primarily exploit the conditioning algorithm to solve SAT problems.

Solving 2-SAT by unit propagation

To solve a 2-SAT problem we first condition on an arbitrarily chosen variable. This sets off a unit propagation process (a chain of unit resolutions) in which variables are removed one-by-one. This continues until either the formula is satisfied or we are left with a contradiction $x_i \land \overline{x}_i$.

Worked example: This process is easiest to understand using a concrete example. Consider the following 2-SAT problem in four variables:

$$\phi := (x_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor \overline{x}_3) \land (x_2 \lor x_3) \land (\overline{x}_2 \lor x_4) \land (x_3 \lor \overline{x}_4). \tag{9}$$

then we set x_1 to false and try again and if neither are satisfiable, then the expression is not satisfiable as a whole.

Let's work through this process explicitly. Setting x_1 to true gives:

$$\phi \land x_1 = (x_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor \overline{x}_3) \land (x_2 \lor x_3) \land (\overline{x}_2 \lor x_4) \land (x_3 \lor \overline{x}_4) \land x_1. \tag{10}$$

We now perform unit resolution with respect to x_1 which means removing any clauses that contain x_1 and removing $\overline{x}_1$ from the rest of the formula to get:

$$\phi \land x_1 = \overline{x}_3 \land (x_2 \lor x_3) \land (\overline{x}_2 \lor x_4) \land (x_3 \lor \overline{x}_4). \tag{11}$$

Notice that we are left with another unit clause $\overline{x}_3$ so we know x_3 must be false and we can perform unit resolution again to yield:

$$\phi \land x_1 \land \overline{x}_3 = x_2 \land (\overline{x}_2 \lor x_4) \land \overline{x}_4. \tag{12}$$

This time, we have two unit clauses. We can perform unit resolution with respect to either. We'll choose x_2 so we now now that x_2 is true and we get:

$$\phi \land x_1 \land \overline{x}_3 \land x_2 = x_4 \land \overline{x}_4 = \text{false}. \tag{13}$$

Clearly this is a contradiction, and so we conclude that the formula is not satisfiable if we set x_1 to true.

We now repeat this process with x_1 = false, which gives

$$\begin{aligned}\phi \land \overline{x}_1 &= (x_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor \overline{x}_3) \land (x_2 \lor x_3) \land (\overline{x}_2 \lor x_4) \land (\overline{x}_4 \lor x_3) \land \overline{x}_1 \\ &= \overline{x}_2 \land (x_2 \lor x_3) \land (\overline{x}_2 \lor x_4) \land (\overline{x}_4 \lor x_3).\end{aligned} \tag{14}$$

$$\phi \land \overline{x}_1 \land \overline{x}_2 = x_3 \land (\overline{x}_4 \lor x_3) \tag{15}$$

which gives the unit clause x_3 and so we set x_3 to true. Now something different happens. The entire of the right hand side disappears. Since there are no clauses left to be satisfied, the formula is satisfiable:

$$\phi \land \overline{x}_1 \land \overline{x}_2 \land x_3 = \text{true} \tag{16}$$

Note that the formula is satisfiable regardless of the value of x_4 (it is on neither side of the equation) so we have found two satisfiable solutions $\{\overline{x}_1, \overline{x}_2, x_3, x_4\}$ and $\{\overline{x}_1, \overline{x}_2, x_3, \overline{x}_4\}$.

Complexity: If there are V variables, there are at most V rounds of unit resolution for each of the two values of the initial conditioned variable. Each unit resolution procedure is linear in the number of clauses C so the algorithm has total complexity $\mathcal{O}[CV]$.

It's possible to reach a case where the chain of unit propagation stops and we have to condition on one of the remaining variables to start it again. However, this only occurs when subsets of the variables have no interaction with one another and so it does not add to the complexity.

Directional resolution

Now consider what happens if we apply the unit resolution approach above to a 3-SAT problem. When we condition on the first variable x_i, we remove clauses that contain x_i and remove $\overline{x}_i$ from the rest of the clauses. Unfortunately, this doesn't create another unit clause (at best it just changes a subset of the 3-clauses to 2-clauses), and so it's not clear how to proceed.

Directional resolution is a method that uses resolution to tackle 3-SAT and above. The idea is to choose an ordering of the variables and then perform all possible resolution operations with each variable in turn before moving on. We continue until we find a contradiction or reach the end. In the latter case, we work back in the reverse order to find the values that satisfy the expression.

Worked example: Again, this is best understood via a worked example. Consider the formula:

$\phi := (x \lor \overline{x} \lor \overline{x}) \land (\overline{x} \lor x \lor \overline{x})\land$

We sort the clauses into bins. Those containing x_1 or $\overline{x}_1$ are put in the bin 1 and any remaining clauses containing x_2 or $\overline{x}_2$ are put in bin 2 and so on:

$$\begin{aligned}
x_1 &: (x_1 \lor \overline{x}_2 \lor \overline{x}_4), (\overline{x}_1 \lor x_3 \lor \overline{x}_5)\\
x_2 &: (x_2 \lor x_3 \lor \overline{x}_4), (\overline{x}_2 \lor \overline{x}_4 \lor x_5)\\
x_3 &: (\overline{x}_3 \lor x_4 \lor x_5)\\
x_4 &:\\
x_5 &:
\end{aligned} \tag{18}$$

We work through these bins in turn. For each bin we perform all possible resolutions and move the resulting generated clauses into subsequent bins. So for bin 1 we resolve the clauses $(x_1 \lor \overline{x}_2 \lor \overline{x}_4)$ and $(\overline{x}_1 \lor x_3 \lor \overline{x}_5)$ with respect to x_1 to get the new clause $(\overline{x}_2 \lor \overline{x}_4 \lor x_3 \lor \overline{x}_5)$. We add this to bin 2 as it contains a x_2 term:

$$\begin{aligned}
x_1 &: (x_1 \lor \overline{x}_2 \lor \overline{x}_4), (\overline{x}_1 \lor x_3 \lor \overline{x}_5)\\
x_2 &: (x_2 \lor x_3 \lor \overline{x}_4), (\overline{x}_2 \lor \overline{x}_4 \lor x_5), (\overline{x}_2 \lor \overline{x}_4 \lor x_3 \lor \overline{x}_5)\\
x_3 &: (\overline{x}_3 \lor x_4 \lor x_5)\\
x_4 &:\\
x_5 &:
\end{aligned} \tag{19}$$

We then consider bin 2 and resolve the clauses with respect to x_2 in all possible ways. In bin 2 there is one clause containing x_2 and we can resolve it against the two clauses containing $\overline{x}_2$. This creates two new clauses that we simplify and add to bin 3 since they contain terms in x_3:

$$\begin{aligned}
x_1 &: (x_1 \lor \overline{x}_2 \lor \overline{x}_4), (\overline{x}_1 \lor \overline{x}_3 \lor \overline{x}_5)\\
x_2 &: (x_2 \lor x_3 \lor \overline{x}_4), (\overline{x}_2 \lor \overline{x}_4 \lor x_5), (\overline{x}_2 \lor \overline{x}_4 \lor x_3 \lor \overline{x}_5)\\
x_3 &: (\overline{x}_3 \lor x_4 \lor x_5), (x_3 \lor \overline{x}_4 \lor x_5), (x_3 \lor \overline{x}_4 \lor \overline{x}_5)\\
x_4 &:\\
x_5 &:
\end{aligned} \tag{20}$$

Now we consider bin 3. Again, there are three clauses
creates two new clauses. Resolving the first and second

created any contradictions of the form $x_i \wedge \overline{x}_i$ during this resolution process

Finding the certificate: To find an example that satisfies the expression, we work backwards, setting the bin value to true or false in such a way that it satisfies the clause. There are no clauses in bin 5 and so we are free to choose either value. We'll set x_5 to be true. Similarly, there are no clauses in bin 4 and so we will arbitrarily set x_4 to true as well. After these changes we have:

$$\begin{aligned}
&x_1 : (x_1 \vee \overline{x}_2 \vee \overline{x}_4), (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_5)\\
&x_2 : (x_2 \vee x_3 \vee \overline{x}_4), (\overline{x}_2 \vee \overline{x}_4 \vee x_5), (\overline{x}_2 \vee \overline{x}_4 \vee \overline{x}_3 \vee \overline{x}_5)\\
&x_3 : (\overline{x}_3 \vee x_4 \vee x_5), (x_3 \vee \overline{x}_4 \vee x_5)\\
&x_4 : \text{true}\\
&x_5 : \text{true}
\end{aligned} \tag{21}$$

Now we consider the third bin. We substitute in the values for x_4 and x_5 and see that both clauses evaluate to true, regardless of the value of x_3, so again, we can choose any value that we want. We'll set x_3 to false to give:

$$\begin{aligned}
&x_1 : (x_1 \vee \overline{x}_2 \vee \overline{x}_4), (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_5)\\
&x_2 : (x_2 \vee x_3 \vee \overline{x}_4), (\overline{x}_2 \vee \overline{x}_4 \vee x_5), (\overline{x}_2 \vee \overline{x}_4 \vee \overline{x}_3 \vee \overline{x}_5)\\
&x_3 : \text{false}\\
&x_4 : \text{true}\\
&x_5 : \text{true}
\end{aligned} \tag{22}$$

Progressing to the second bin, we observe that the second and third clause are already satisfied by the previous assignments, but the first clause is not since x_3 is false and x_4 is true. Consequently, we must satisfy this clause by setting x_2 to true:

$$\begin{aligned}
&x_1 : (x_1 \vee \overline{x}_2 \vee \overline{x}_4), (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_5)\\
&x_2 : \text{true}\\
&x_3 : \text{false}\\
&x_4 : \text{true}\\
&x_5 : \text{true}
\end{aligned} \tag{23}$$

Complexity: The directional resolution procedure works, but is not especially efficient. For large problems, the number of clauses can expand very quickly: if there were C clauses and half contain x_1 and the other half $\overline{x}_1$ then we could create $C^2/4$ new clauses in the first step. For a K -SAT problem, each of these clauses are larger than the original ones with size $2(K-1)$.

It is possible to improve the efficiency. Any time we generate a unit clause, we can perform unit propagation which may eliminate many variables. Also in our example we organized the bins by the variable index, but this was an arbitrary choice. This order can have a big effect on the total computational cost and so careful selection can improve efficiency. However, even with these improvements, this approach is not considered viable for large problems.

SAT solving algorithms based on conditioning

In this section, we will develop algorithms that are fundamentally centered around the conditioning operation (although they also have unit resolution embedded). We'll describe both the DPLL algorithm and clause learning algorithms which underpin most modern SAT solvers. To understand these methods, we first need to examine the connection between conditioning and tree search.

SAT as binary search

We'll use the running example of the following Boolean formula with $C=7$ clauses and $V=4$ variables:

$$\begin{aligned}\phi := (x_1 \lor x_2) \land (x_1 \lor \overline{x}_2 \lor \overline{x}_3 \lor x_4) \land (x_1 \lor \overline{x}_3 \lor \overline{x}_4) \land \\ (\overline{x}_1 \lor x_2 \lor \overline{x}_3) \land (\overline{x}_1 \lor x_2 \lor \overline{x}_4) \land (\overline{x}_1 \lor x_3 \lor x_4) \land (\overline{x}_2 \lor x_3),\end{aligned} \tag{24}$$

Consider conditioning on variable x_1 so that we have:

$$\phi = (\phi \land \overline{x}_1) \lor (\phi \land x_1). \tag{25}$$

This equation makes the obvious statement that in any satisfying solution x_1 is either true or false. We could first investigate the case where x_1 is f
are done, and if not we consider the case where x_1 is
could condition each of these two cases on x_2 to get:

and now we could consider each of the four combinations $\{\overline{x}_1\overline{x}_2\}, \{\overline{x}_1x_2\}, \{x_1\overline{x}_2\}$ and $\{x_1x_2\}$ in turn, terminating when we find a solution that is SAT.

One way to visualise this process is as searching through a binary tree (figure 1). At each node of the tree we branch on one of the variables. When we reach a leaf, we have known values for each variable and we can just check if the solution is SAT.

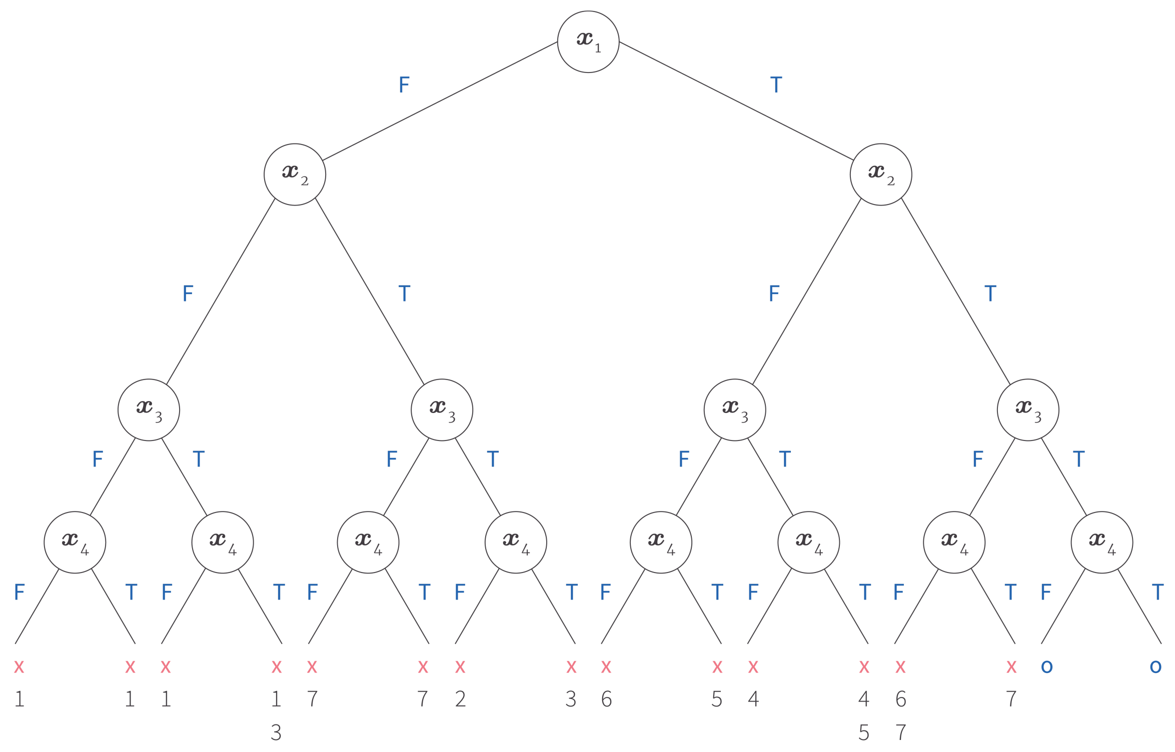


Figure 1. SAT as binary search. At each node of the search tree we condition on a variable, splitting into a left sub-tree in which this variable is set to false and a right sub-tree in which it is set to true. There is one level in the tree for each variable so that at each leaf all the variables are set and we can test if the formula is satisfied. For the example in equation 24, the formula evaluates to false for the first 14 leaves and the individual clauses that are violated are indicated in grey at each leaf. The last two leaves are both satisfying solutions. In practice, we would stop searching when we found the first satisfying solution and so we would only need to test 15 leaves in this example.

This example was deliberately constructed to be pathological in that the first 14 combinations (or equivalently leaves of the tree) all make the formula evaluate to false. These are signified in the plot by red crosses. We number the clauses:

$$\begin{aligned}
1 &: (x_1 \vee x_2)\\
4 &: (\overline{x}_1 \vee x_2 \vee \overline{x}_3)\\
5 &: (\overline{x}_1 \vee x_2 \vee \overline{x}_4)\\
6 &: (\overline{x}_1 \vee x_3 \vee x_4)\\
7 &: (\overline{x}_2 \vee x_3)
\end{aligned} \tag{27}$$

and for each leaf of the tree in figure 1, the clauses that were contradicted are indicated in grey. In this case, both of the last two combinations (leaves) satisfy the formula, and once we find the first one $(x_1, x_2, x_3, \overline{x}_4)$ we can return SAT.

Note, we have not yet obviously made the algorithm more efficient. We might still have to search all 2^V combinations of variables to establish satisfiability or lack thereof. However, viewing SAT solving as tree search is the foundation that supports more efficient algorithms.

Efficient binary search

We can immediately improve the efficiency of the binary search method by some simple bookkeeping. As we pass through the tree we keep track of which clauses are satisfied and which are not. As soon as we find one that is not satisfied, we do not need to explore further and we can backtrack. Similarly, if we find a situation where all of the clauses are already satisfied before we reach a leaf then we can return SAT without exploring further. This means that the variables below this point can take any value.

In our worked example, when we pass down the first branch and set x_1 to false and x_2 to false we have already contradicted clause 1 which was $(x_1 \vee x_2)$, and so there is no reason to proceed further. Continuing in this way we only need to search a subset of the full tree (figure 2). We find the first satisfying solution when $x_1, x_2, x_3 = \text{true}, \text{true}, \text{true}$ and need not continue to the leaf. As we saw from the full tree in figure 1, the setting of x_4 is immaterial.

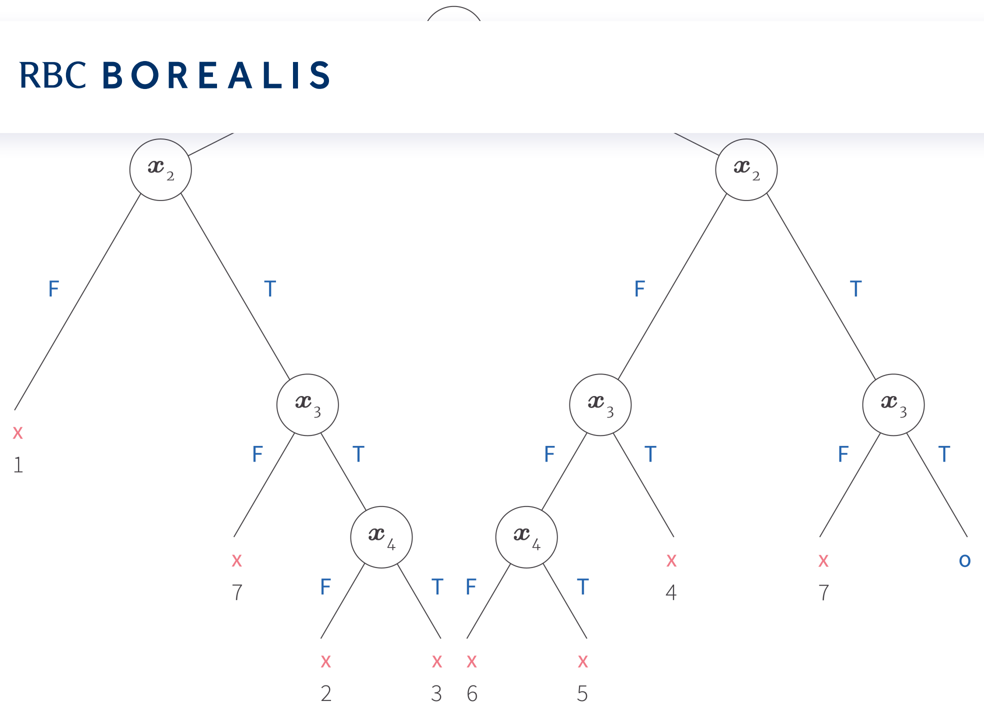


Figure 2. Efficient binary search. With some simple bookkeeping, tree search can be made much more efficient. If we track the status of each clause then we can backtrack as soon as one of the clauses is violated. Again, the index of the violated clause is shown in grey. Similarly, when we have satisfied all of the clauses we can return SAT even though we have not yet reached a leaf. The variables below this can be set to any value.

DPLL

We can also consider the tree search from an algebraic point of view. Each time we make a decision at a node in the tree, we are conditioning on a given variable. So when we set x_1 to false, the resulting formula is

$$\phi \land \overline{x}_1 := x_2 \land (\overline{x}_2 \lor \overline{x}_3 \lor x_4) \land (\overline{x}_3 \lor \overline{x}_4) \land (\overline{x}_2 \lor x_3), \tag{28}$$

where we have used the usual recipe of removing all clauses containing $\overline{x}_1$ and removing the term x_1 from the remaining clauses.

The Davis–Putnam–Logemann-Loveland (DPLL) algorithm takes tree search one step further, by embedding unit propagation into the search algorithm (figure 3). For example, when we condition on $\overline{x}_1$ and yield the new expression in equation 28, we generate the unit clause x_2. We can perform unit resolution using x_2 to get:

$$\phi \land \overline{x}_1 \land x_2 := (\overline{x}_3 \lor x_4) \land (\overline{x}_3 \lor \overline{x}_4) \land x_3, \tag{29}$$

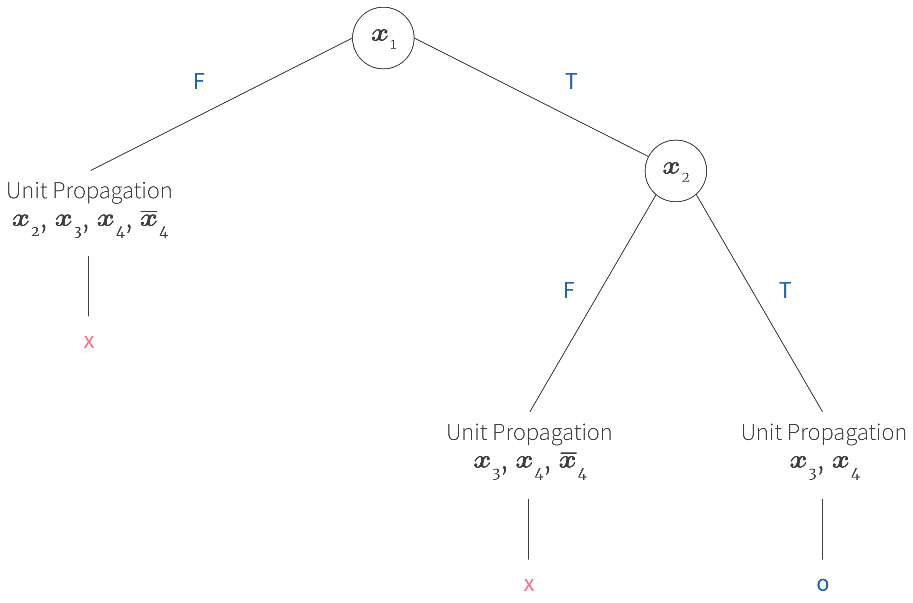
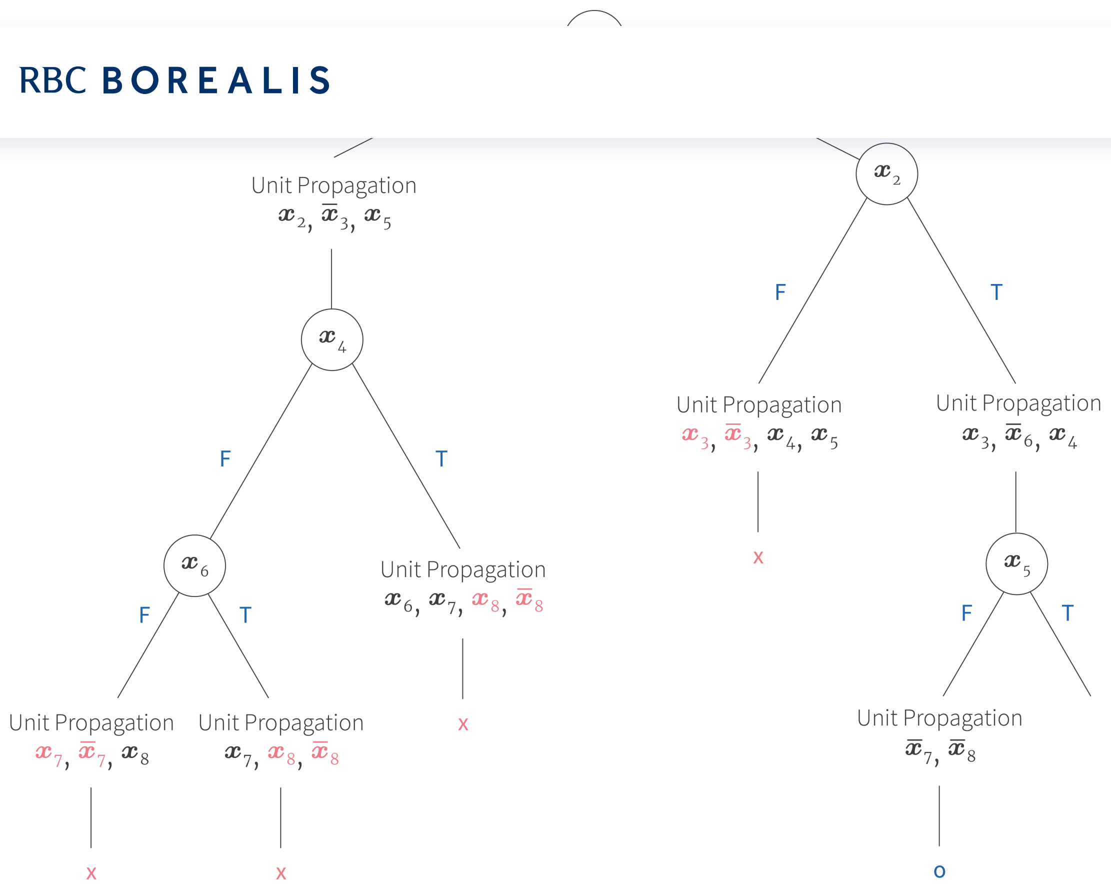


Figure 3. DPLL algorithm. By performing unit propagation where possible, we can eliminate many variables very efficiently. In this case, once we condition on $\overline{x}_1$, this creates a unit clause in x_2, which starts a chain of unit resolution operations that establishes a contradiction. Similar effects happen at other points in the tree.

To summarize, the DPLL algorithm consists of tree search, where we perform unit propagation whenever unit clauses are produced. Since unit resolution can be done in linear time, this is much more efficient than the tree search that it replaces.

Note that in our worked example, the unit propagation process always generated a contradiction or a SAT solution. However, this is not necessarily the case in a larger problem. After unit resolution there will usually be non-unit clauses left containing the remaining variables have neither been conditioned on, nor eliminated using unit resolution. At this point, we condition on the next available variable and continue down the tree, performing unit resolution when we can (figure 4).



Unit Propagation
$x_2, \overline{x}_3, x_5$
x_4
F
T
x_6
F
T
Unit Propagation
$x_7, \overline{x}_7, x_8$
x
Unit Propagation
$x_7, x_8, \overline{x}_8$
x
Unit Propagation
$x_6, x_7, x_8, \overline{x}_8$
x
x_2
F
T
Unit Propagation
$x_3, \overline{x}_3, x_4, x_5$
x
Unit Propagation
$x_3, \overline{x}_6, x_4$
x_5
F
T
Unit Propagation
$\overline{x}_7, \overline{x}_8$
o

Figure 4. DPLL algorithm in practice. In a real problem the DPLL algorithm will alternate between conditioning on variables and performing unit resolution. The effect of this is that we condition on different variables in different paths of the tree.

Conflict Driven Clause learning

The DPLL algorithm makes SAT solving by tree search much more efficient, but there can still be considerable wasted computation. Consider the case where we have set x_1 to false and then set x_2 to false (figure 5). However, imagine that there are clauses that mean that when x_2 is false, there is no way to set the variables x_3 and x_4 in a valid way. For example, the following combination of clauses will achieve this:

$$(x_2 \lor x_3 \lor x_4) \land (x_2 \lor x_3 \lor \overline{x}_4) \land (x_2 \lor \overline{x}_3 \lor x_4) \land (x_2 \lor \overline{x}_3 \lor \overline{x}_4). \tag{30}$$

As we work through the sub-tree in the blue region in figure 5, we duly establish that there is no possible solution.

As we search through the tree, we will eventually come to another place where we set x_2 to false and now we must work through exactly the same calculations again to establish that there is no valid solution (yellow region in figure 5). In a large prob

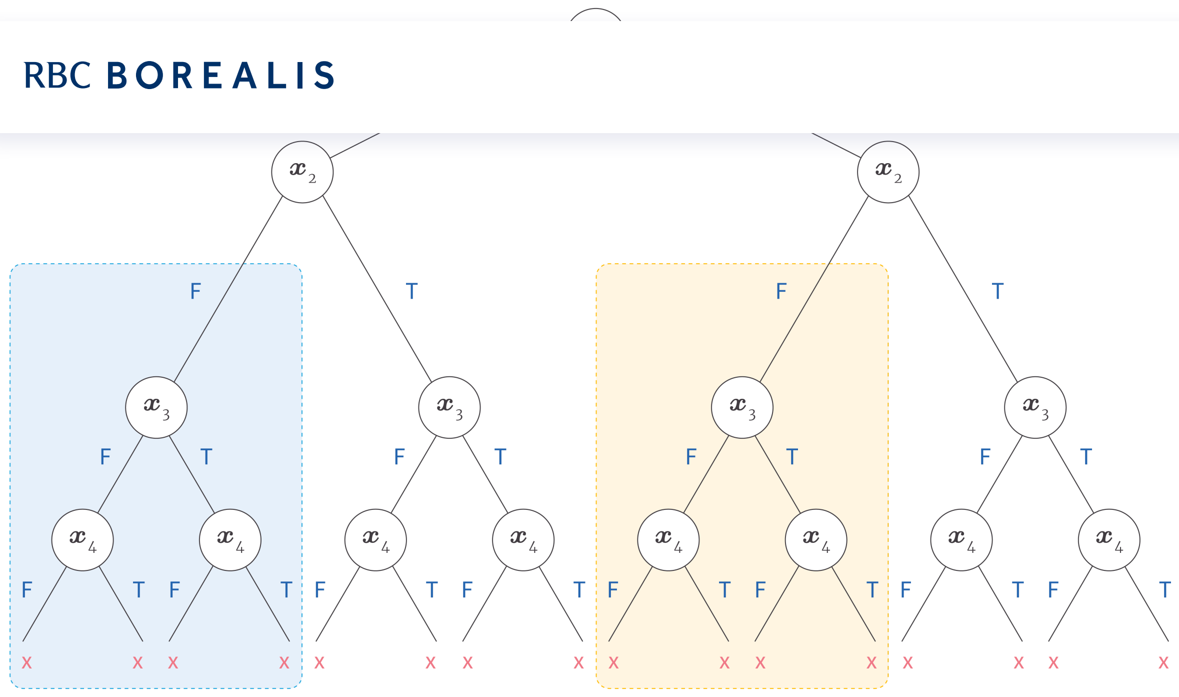


Figure 5. Motivation for conflict-driven clause learning. Consider the case where setting x_2 to false inevitably results in a conflict where there is no way to set x_3 and x_4. Without taking action, we will have to repeat the computations to find this conflict in every sub-tree where x_2 is false (blue and yellow rectangles). When conflict-driven clause learning algorithms find a conflict in a sub-tree, they add a new clause to the original expression that prevents redundant exploration of sub-trees.

Conflict-driven clause learning aims to reduce this redundancy. When a conflict occurs, the cause is found and we add a new clause to the original statement that prevents exploration of redundant sub-trees. For example, in this simple case, we could add the clause (x_2) which would prevent exploration of trees where x_2 is false.

Unfortunately, the causes of a conflict are usually more complex than a single variable. To find the combinations of variables that are ultimately responsible for the conflict, we build a structure called an *implication graph* as we search through the tree.

Figure 6a provides a concrete example of a SAT problem where there are 11 clauses and 10 variables. Figure 6b illustrates the situation where we are mid-way through the DPLL search in which we have interleaved processes of conditioning (blue shaded areas) and unit resolution (yellow shaded areas). We have just established a conflict at clause 11 (at the blue arrow) which cannot be satisfied when we set x_5 to true.

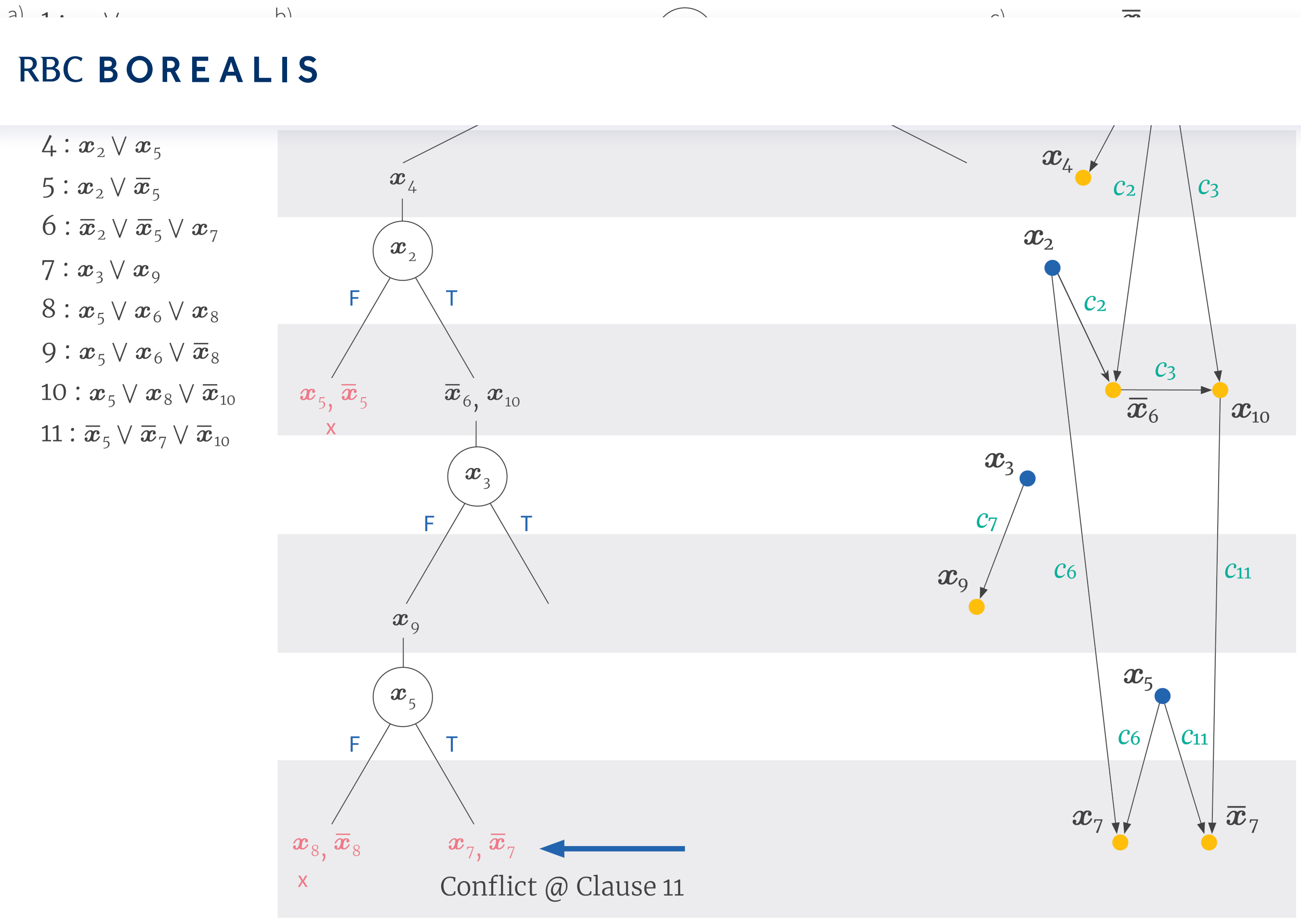


Figure 6. Implication graphs. a) A SAT problem with 11 clauses in 10 variables. b) We are mid-way though the DPLL process (at the blue arrow) and have just found a conflict at clause 11. c) The implication graph representing the current state. Each vertex corresponds to a variable (blue if conditioned, yellow if inferred by unit resolution). The incoming edges to yellow vertices correspond to the variables that caused the vertex variable to be inferred and are labelled with the relevant clause. So, for example, x_6 is set to false by clause c_2 because x_1 is false and x_2 is true. The conflict results when setting variable x_5 implies both x_7 and it's complement $\overline{x}_7$.

Figure 6c is the implication graph associated with this point in the search, which contains all of the variables that we have established so far. The literals $\overline{x}_1, x_2, x_3, x_5$ that we conditioned on are depicted with blue vertices and the literals $x_4, \overline{x}_6, x_{10}, x_9, x_7$ and $\overline{x}_7$ that resulted from unit propagation are shown as yellow vertices. Each edge depicts a contribution to the unit resolution process. For example, the edge between $\overline{x}_1$ and x_4 represents the fact that when x_1 is set to false, we must set x_4 to true. This is due to clause 1 and the edge is accordingly labelled with c_1. Similarly, we can see that x_{10} has become true by clause c_3 because previously in the search process x_1 and x_6 were both set to false.

You can see from this implication graph exactly how the conflict happened. When we condition on x_5, clause c_6 implied that x_7 must be true given that x_2 and x_5 were both true, but clause c_{11} implied that x_7 must be false given that x_5 and x_{10} were both true. So, one interpretation is that the conflict is inevitable given states x_2, x_5 and x_{10} that were inputs to these contradictory clauses.

However, this is not the only interpretation. For examp

this was only set to true because we previously set x_1

variables are the source vertices of the edges that were cut.

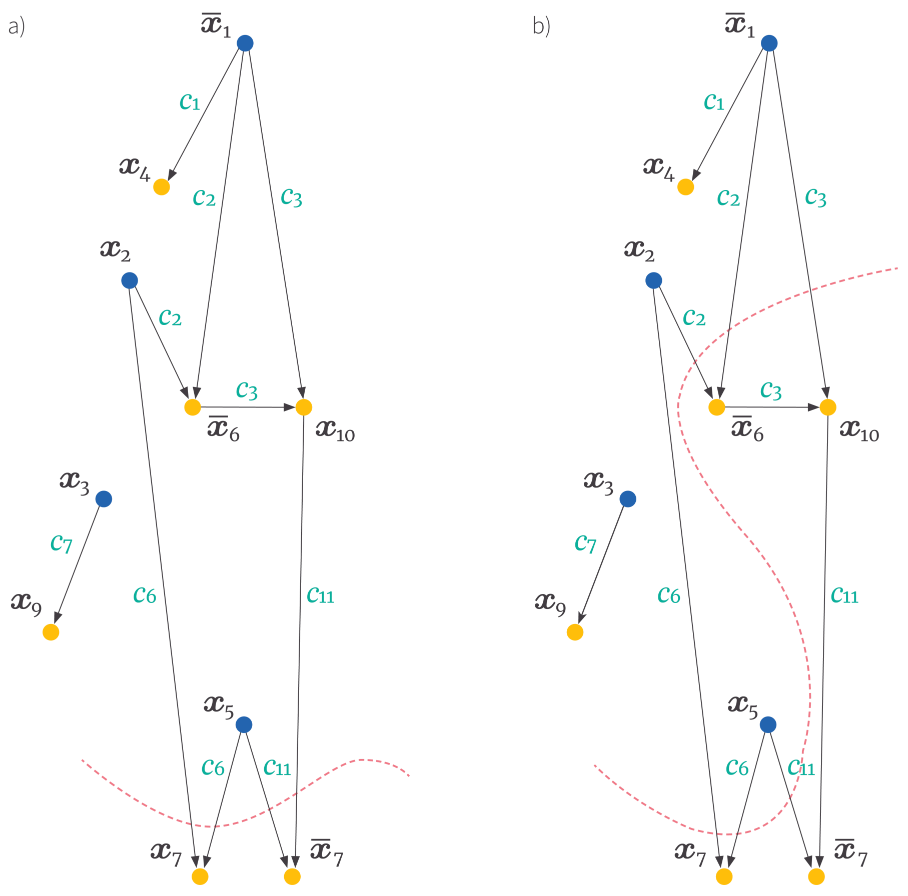


Figure 7. Cutting the implication graph to find the causes of a conflict. Any cut that separates the conditioned nodes (in blue) from the conflict can be interpreted as a possible cause. a) In this case, the cause of the conflict is attributed to variables x_2, x_5, x_{10}, since the edges from these variables are cut. b) In this second case, the cause is attributed to variables $\overline{x}_1, x_2, x_5$.

Having established a cause, we must now derive a new clause that prevents the SAT solver from exploring similar dead-ends in the future. If the case was attributed to $\overline{x}_1, x_2, x_5$, then we would add the clause $(x_1 \lor \overline{x}_2 \lor \overline{x}_5)$ to prevent this combination happening. We continue exploring the tree, by jumping back up the tree structure to a sensible point and resuming with this new constraint.

Machine leaning powered CDCL

The previous discussion outlined the main ideas of conflict-driven clause learning algorithms, but there are many additional choices to be made in a modern system. For example, we must decide the order of variables to condition on. In our examples, we have done this in numerical order, but this choice was arbitrary and there is no particular reas
we go down different branches in the tree. Much worl
making this choice. For example, we might prioritize va

There are also many other decisions to make. In CDCL, we must choose which of many potential explanations for a conflict is superior and decide exactly where we should jump back to in the tree. Some solvers periodically restart the solution process to avoid wasting the available computation time fruitlessly searching a single branch, and we must decide when exactly to perform these restarts. One approach to making these decisions is to use machine learning to guide the choices.

For example, Liang *et* al., (2016) developed uses a reward function to choose the order in which variables in a CDCL solver are considered. A reward function $r[i]$ is defined for each variable x_i:

$$r[i] \propto \frac{1}{numConflicts - lastConflict[i])} \tag{31}$$

Here $numConflicts$ keeps track of the total number of conflicts the solver has encountered so far whereas $lastConflict[i]$ keeps track of the last time variable x_i was involved in a conflict. From this we can see that a variable which was recently involved in a conflict would get a high reward. This reward is then incorporated into a score function:

$$Q[i] \longleftarrow (1-\alpha)Q[i] + \alpha r[i] \tag{32}$$

At any iteration where a new variable must be selected for conditioning, the variable with the highest score is picked, provided that it is not currently already conditioned on. This is known as the *conflict history branching heuristic*.

In the above formulation, the terms appearing in the reward will always be known and can hence be computed exactly. However, Liang *et* al., (2016) also dealt with a case where the reward function is defined such that the terms appearing in it have associated uncertainty. This new reward definition improves the branching heuristic and the authors show how tools from a Multi-Armed Bandit framework in RL can be used to estimate the uncertainty and further improve performance.

In further work Liang *et* al., 2017 the same authors show how gradient based methods can be used to optimize another branching heuristic, one based on how many learnt clauses can be obtained from each decision. Other examples of machine learning in the SAT community include Nejati *et* al., (2017) where reinforcement learning
solver.

algorithms based on these operations. We started with using resolution (via unit propagation) to efficiently solve 2-SAT and then investigated the directional resolution for 3-SAT and above. We re-framed the SAT solving problem as tree search where conditioning is used at each branch in the tree. This led to the DPLL and CDCL algorithms.

For further information about SAT solving including the DPLL and CDCL algorithms, consult the Handbook of Satisfiability. Most chapters are available on the internet if you search for their titles individually. A second useful resource is Donald Knuth's Facsicle 6.

In part III of this article, we'll investigate a completely different approach to SAT solving that relies on belief propagation in factor graphs. Finally, we'll show how the machinery of SAT solving can be extended to continuous variables by introducing satisfiability module theory (SMT) solvers.