CS208 Logic Week 02 : Notes on SAT Solvers

Robert Atkey ⟨robert.atkey@strath.ac.uk⟩

These notes cover the same material as CS208 Videos 2.2 and 2.3.

1 SATisfiability Solving

As we saw in Video 2.1, if we had a way of efficiently finding satisfying valuations for large logical formulas, we would have a way of efficiently computing working combinations of installed packages. There are many other problems that can be efficiently solved if we have a way of testing formulas for satisfiability:

1. As we saw in Week 1, discovering whether or not an entailment holds can be reduced to the problem of whether or not a certain formula is satisfiable.

2. We can encode digital circuits as logical formulas. If we have two circuits, encoded as formulas P and Q, that we want to check are equivalent for all inputs, then we can encode this problem as asking whether there is a valuation that makes P true and Q false ($P \wedge \neg Q$) or vice versa ($\neg P \wedge Q$). If there is such a valuation, then we have found an input for which the two circuits differ. We will see examples of this in Week 3.

3. It is possible to encode the possible steps of finite state machine as logical formulas. Finding satisfying valuations in this case corresponds to finding sequences of states with certain properties, such as a sequence that might lead to a "bad" state.

There are so many problems that would be solvable if there were efficient ways of finding satisfying valuations that considerable effort has been spent on finding ways to do this. Programs that find satisfying valuations are known a SAT solvers.

Unfortunately, there is a stumbling block. The problem of finding satisfying valuations is *NP-complete*. "NP" refers to the class of problems that are solvable in *Non-deterministic Polynomial time*. These are problems that if we were able to guess the answer (using the non-determinism) then we could check it in polynomial time. Satisfiability is in this class: if we guess a valuation v, we can check it quickly by computing $[\![P]\!]v$ using the steps we saw in Week 1. However, if we can't guess a satisfying valuation there is currently no known *general* strategy better than trying all of them, which takes an amount of time exponential in the size of the input. The question of whether or not there is a general strategy that works in polynomial time is known as the P = NP problem, which is a major unsolved problem in Computer Science. Finding satisfying valuations is also NP-*complete*, which means that a polynomial time solution to this problem would give a polynomial time solution to all NP problems.

Despite there being no known *general* solution that is better than trying every possible valuation, there has been great progress on SAT solvers that do well on problems that arise in the "real world" like the ones listed above. Formulas that are generated from real world problems often have a large amount of regularity that it is possible for a SAT solver to exploit to avoid searching every possible valuation. In this section, I will describe what SAT solvers do, and outline how a simple but not completely naive SAT solver works.

1.1 What a SAT Solver Inputs and Outputs

Input SAT solvers do not take arbitrary formulas as input. The input formula must be in a special form called *Conjunctive Normal Form* (CNF). Formulas in CNF look like the following:

$$\begin{array}{ll} & (\neg a \vee \neg b \vee \neg c) \\ \wedge & (\neg b \vee \neg c \vee \neg d) \\ \wedge & (\neg a \vee \neg b \vee c) \\ \wedge & b \end{array}$$

where

1. the entire formula is a conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ of
2. *clauses*, where each *clause* C_i has the form $L_{i,1} \vee L_{i,2} \vee \cdots \vee L_{i,k}$ and is a disjunction of
3. *literals*, where each *literal* $L_{i,j}$ is either an atom $x_{i,j}$ or a negated atom $\neg x_{i,j}$.

We will see how to convert formulas to CNF in Week 3. Often, however, it is possible to arrange things so that our encoding of a particular problem is already in CNF. The Package Installation Problem from Video 2.1 is an example of this.

Output Given a formula in CNF as input, a SAT solver's job is to find a valuation v for the atoms in the formulas that satsifies the formula. Due to the special form of formulas in CNF, this valuation must make at least one literal in every clause is true. If there is such a valuation then the SAT solver returns SAT with the valuation, or returns UNSAT. (In practice, a SAT solver may also return UNKNOWN if it runs for more than a user-set amount of time, or runs out of memory.)

1.2 Partial Valuations

Even though a SAT solver returns a full valuation in the end, during its operation it constructs *partial valuations*. A partial valuation only assigns T or F to some of the atoms, unlike a (full) valuation which must assign a truth value to all the atoms. A partial valuation also keeps track of the status of each assignment in terms of whether it is a *d*ecision that has been made, or something that has been *f*orced. This information is used to guide the search, as I describe in the next section.

More formally, a *partial valuation* $v^?$ is a *sequence* of assignments of truth values to atoms; with each one marked as either a

1. *d*ecision point, if we guessed this value.
2. *f*orced, if we were forced to have this value.

So each assignment looks like $a :_d x$ or $a :_f x$, where a is an atom name and x is a truth value (T or F).

Notation Some notation, which will be useful for the next section:

1. I will write $v^?$ to stand for some partial valuation. Here are some examples:

$$v_1^? = [a :_d \mathsf{T}, b :_d \mathsf{F}, c :_f \mathsf{T}]$$
$$v_2^? = [a :_f \mathsf{F}, b :_d \mathsf{F}]$$

   Note that, unlike in normal valuations, the order of assignments matters. Partial valuations are used to keep track of the decisions made while searching for a satisfying assignment, and it matters which order the decisions are made.

2. We write

$$v_1^?, a :_d x, v_2^?$$

   for a partial valuation with $a :_d x$ somewhere in the middle.

3. We write

$$\mathit{decisionfree}(v^?)$$

   if none of the assignments in $v^?$ are marked d, i.e. all decisions in $v^?$ are $f$orced.

1.3 Finding Satisfying Valuations via Rules

The kind of SAT solver I will describe here works by incrementally building a partial valuation, until either the partial valuation is a valuation that satsifies the formula, or the solver works out that no such valuation is possible. The solver starts with an empty partial valuation and extends it by following the rules described below. In this section, I will introduce the rules one by one, with examples of how the solver works with those rules. The rules will all be summarised in Section 1.4.

Initial Partial Valuation We start the search for a satisfying valuation with the empty partial valuation:

$$v^? = []$$

With the empty valuation, we have made no commitments

Guessing If the current partial valuation $v^?$ does not contain an atom a that is in the clauses, then we can make a guess. We have a rule DecideTrue, which guesses that a ought to be T:

$$v^?, a :_d \mathsf{T}$$

and a rule DecideFalse, which guesses that a ought to be F:

$$v^?, a :_d \mathsf{F}$$

Because these assignments to a are guesses, we have marked these as $d$ecision points. Note that I have not said which atom a or which of T or F is chosen at any particular step. It is up to the implementor of a SAT Solver to invent a strategy for which one to pick at each step: for example, always choosing the atoms in alphabetical order, and always choosing T first. More sophisticated strategies are also possible, such as trying atoms that appear more often first.

Success If the current $v^?$ makes all the clauses true (for all i, $[\![C_i]\!]v^? = \mathsf{T}$), then we have a rule Success that allows the solver to stop with result $\mathsf{SAT}(v^?)$.

Example I With the steps we have so far, we are able to find satisfying assignments, as long as we always get lucky with our guesses. Let's look at how the rules we have so far work on an example formula:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

To illustrate how the solver works, I will colour in literals in *green* when they are true with the current partial valuation, and *red* when they are false. Remember: the goal is to make at least one literal in every clause *true*.

- We start with the initial partial valuation:

  1. $[]$

  This makes none of the literals true or false, so the formula is unannotated:

  $$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- We guess that the atom a ought to be F, so we update our partial valuation:

  2. $[a :_d \mathsf{F}]$

  The annotated formula is now:

  $$(\textit{green}\ \neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\textit{green}\ \neg a \vee \neg b \vee c) \wedge b$$

- We guess that the atom b ought to be T:

  3. $[a :_d \mathsf{F}, b :_d \mathsf{T}]$

  The annotated formula is now:

  $$(\textit{green}\ \neg a \vee \textit{red}\ \neg b \vee \neg c) \wedge (\textit{red}\ \neg b \vee \neg c \vee \neg d) \wedge (\textit{green}\ \neg a \vee \textit{red}\ \neg b \vee c) \wedge \textit{green}\ b$$

  We are now seeing some literals become false, but this is OK because our goal is to make *at least one* literal in every clause true. Crucially, the final clause, which is just b, is satisfied.

- We guess that the atom c ought to be T:

  4. $[a :_d \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}]$

The annotated formula becomes:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- We guess that d ought to be F:

  5. $[a :_d \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}, d :_d \mathsf{F}]$

  The annotated formula becomes:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- We now observe that for all clauses at least one literal is true. Therefore, we have found a satisfying valuation. The SAT solver now terminates with the result $\mathsf{SAT}(\{a \mapsto \mathsf{F}, b \mapsto \mathsf{T}, c \mapsto \mathsf{T}, d \mapsto \mathsf{F}\})$.

But what happens if we do not always get lucky with our guesses? If this happens we will have to backtrack and try again.

Backtracking If we have a partial valuation:

$$v_1^?, a :_d x, v_2^?$$

and $\mathit{decisionfree}(v_2^?)$, so that $a \mapsto x$ was our most recent guess, then we can backtrack (throw away $v_2^?$) and change our mind:

$$v_1^?, a :_f \neg x$$

now marking the assignment as $f$orced. This rule is called BackTrack.

Failure If all decisions in the current partial valuation $v^?$ are forced ($\mathit{decisionfree}(v^?)$), then we cannot backtrack. If there is at least one clause C_i such that $[\![C]\!]v^? = \mathsf{F}$, then we have run out of options and have to return UNSAT. This rule is called Fail.

Example II Let's look at our example formula again, but this time doing a more realistic search where we don't rely on lucky guesses. Instead, we systematically try the atoms in the order a, b, c, d and always try T first. We will rely on the backtracking steps to undo any bad decisions we make.

- We start again with the empty partial valuation:

  1. $[]$

  This makes none of the literals true or false, so the formula is unannotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Following our strategy, we first pick the atom a and guess that it is T:

  2. $[a :_d \mathsf{T}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Again we follow our strategy, and guess that the next atom b is T:

  3. $[a :_d \mathsf{T}, b :_d \mathsf{T}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- And again we follow our strategy, and guess that the next atom c is T:

  4. $[a :_d \mathsf{T}, b :_d \mathsf{T}, c :_d \mathsf{T}]$

The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now we have a problem: the first clause is all red, but our goal was to make every clause have at least one green. We backtrack to our most recent decision, and flip the assignment to c from T to F, making the decision as $f$orced:

  5. $[a :_d \mathsf{T}, b :_d \mathsf{T}, c :_f \mathsf{F}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now we have another problem: the third clause is all red. We backtrack again to the rightmost $d$ecision. This means that we flip the b from T to F and mark it as $f$orced:

  6. $[a :_d \mathsf{T}, b :_f \mathsf{F}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now the fourth clause has become completely red! We backtrack again, and flip our decision on a, marking it as $f$orced:

  7. $[a :_f \mathsf{F}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- No clauses are all marked red, so we resume with our guessing. Following our strategy, we guess that b is T (we already guessed this above, but in the context of guessing that a was T, we are now trying it when a is F):

  8. $[a :_f \mathsf{F}, b :_d \mathsf{T}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- No clauses are marked as failing, so we continue with our strategy and guess that c is T:

  9. $[a :_f \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Again, no clauses are marked as failing, so we continue with our strategy and guess that d is T:

  10. $[a :_f \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}, d :_d \mathsf{T}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now the second clause has failed! We must backtrack and undo our guess that d was T:

  11. $[a :_f \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}, d :_f \mathsf{F}]$

  The formula is now annotated:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- We now observe that for all clauses at least one literal is true. Therefore, we have found a satisfying valuation. The SAT solver now terminates with the result $\mathsf{SAT}(\{a \mapsto \mathsf{F}, b \mapsto \mathsf{T}, c \mapsto \mathsf{T}, d \mapsto \mathsf{F}\})$.

As can be seen from this example, backtracking has the potential to waste a lot of time when it is "obvious" from looking at the formula that trying a particular guess will not work. Looking at the formula in this example, we can see straight away that b must be T in any solution because it is in a clause by itself. To speed up the search, we use the formula to guide the search process by letting some decisions be forced by looking for "unit clauses". Unit clauses are those where there is only one literal left that is not yet marked red. Clauses with only one literal are always unit clauses.

Unit Propagation If we are in a situation like:

$$(\neg b \vee \neg c \vee \neg d)$$

then if the current valuation is to succeed in any way, it must be the case that $d \mapsto \mathsf{F}$. Similarly, in a situation like:

$$(\neg b \vee \neg c \vee d)$$

then if the current valuation is to succeed in any way, it must be the case that $d \mapsto \mathsf{T}$. We turn this observation into an extra rules that describe what steps a SAT solver can take.

The *Unit Propagation* step: If there is a clause

$$C \vee a$$

and $[\![C]\!]v^? = \mathsf{F}$, then the rule UnitPropTrue extends $v^?$ to:

$$v^?, a :_f \mathsf{T}$$

Symmetrically, if there is a clause

$$C \vee \neg a$$

and $[\![C]\!]v^? = \mathsf{F}$, then the rule UnitPropFalse extends $v^?$ to:

$$v^?, a :_f \mathsf{F}$$

Note: the a needn't necessarily appear at the end of the clause.

Example III With the Unit Propagation rule, we can go through the same example again, and see that unit propagation saves us from doing many of the backtrackings. When do have to make a decision, we will use the same strategy as above: we try the atoms in alphabetical order, and we try T first.

- We start again with the empty partial valuation:

  1. $[]$

  Again, this makes none of the literals true or false, so the formula is unannotated:

  $$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- We observe that there is a clause with exactly one unassigned literal: the final clause b. We apply the Unit Propagation rule, which says that b must be T for the whole formula to be satisfiable. This yields the following partial valuation:

  2. $[b :_f \mathsf{T}]$

  where the assignment to b is $f$orced. The formula is annotated like so:

  $$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now there are no unit clauses — every unsatisfied clause either has at least two unassigned literals or none. So we must make a decision. Using our strategy, we pick a to be T, yielding the following partial valuation:

  3. $[b :_f \mathsf{T}, a :_d \mathsf{T}]$

and the formula is annotated like so:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- There are now two unit clauses: the first one and the third one. We pick the first one (a real SAT solver may have some strategy for picking clauses based on some heuristics). The only non-F literal in this clause is $\neg c$, so to satisfy this clause we are forced to make c be F:

  4. $[b :_f \mathsf{T}, a :_d \mathsf{T}, c :_f \mathsf{F}]$

  and the formula is annotated like so:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now the third clause has failed! All of its literals are marked as F. This means that we must backtrack to the last *d*ecision we made, which was $a :_d \mathsf{T}$, and flip it to a *f*orced decision. Note that we skip over the forced assignment to c: this assignment was forced by the decisions made beforehand, so there is no point in changing it. The new partial valuation is this:

  5. $[b :_f \mathsf{T}, a :_f \mathsf{F}]$

  and now the formula is annotated like so:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now there are *no* unit clauses: there are clauses that only have one unassigned literal, but no clauses with one unassigned literal and all the others *false*. So the Unit Propagation rule does not apply. Instead, we follow our strategy and assign the next unassigned atom, c, to be T:

  6. $[b :_f \mathsf{T}, a :_f \mathsf{F}, c :_d \mathsf{T}]$

  and now the formula is annotated like so:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now we do have a unit clause: the second one. In order to satisfy this clause, we must set d to be F:

  7. $[b :_f \mathsf{T}, a :_f \mathsf{F}, c :_d \mathsf{T}, d :_f \mathsf{F}]$

  and now the formula is annotated like so:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

- Now every clause as at least one satisfied literal. The solver terminates with the result $SAT(\{a \mapsto \mathsf{F}, b \mapsto \mathsf{T}, c \mapsto \mathsf{T}, d \mapsto \mathsf{F}\})$.

Comparing the examples For ease of comparison, I have written out the sequences of partial valuations for the two examples side by side:

Without Unit Propagation	**With Unit Propagation**
1. $[]$	1. $[]$
2. $[a :_d \mathsf{T}]$	2. $[b :_f \mathsf{T}]$
3. $[a :_d \mathsf{T}, b :_d \mathsf{T}]$	3. $[b :_f \mathsf{T}, a :_d \mathsf{T}]$
4. $[a :_d \mathsf{T}, b :_d \mathsf{T}, c :_d \mathsf{T}]$	4. $[b :_f \mathsf{T}, a :_d \mathsf{T}, c :_f \mathsf{F}]$
5. $[a :_d \mathsf{T}, b :_d \mathsf{T}, c :_f \mathsf{F}]$	5. $[b :_f \mathsf{T}, a :_d \mathsf{F}]$
6. $[a :_d \mathsf{T}, b :_f \mathsf{F}]$	6. $[b :_f \mathsf{T}, a :_f \mathsf{F}, c :_d \mathsf{T}]$
7. $[a :_f \mathsf{F}]$	7. $[b :_f \mathsf{T}, a :_f \mathsf{F}, c :_d \mathsf{T}, d :_f \mathsf{F}]$
8. $[a :_f \mathsf{F}, b :_d \mathsf{T}]$	
9. $[a :_f \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}]$	
10. $[a :_f \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}, d :_d \mathsf{T}]$	
11. $[a :_f \mathsf{F}, b :_d \mathsf{T}, c :_d \mathsf{T}, d :_d \mathsf{F}]$	

Unit Propagation saves the solver from trying several dead ends by using the formula itself to guide the search. In this example, Unit Propagation meant that we never bothered to try b being F at any point, because this can *never* be a part of a solution to the formula, but the backtracking-only solver tried this out for steps 3-5. Also, while the right-hand trace still made the bad decision $a :_d \mathsf{T}$ in step 3, but this was undone by step 5 with Unit Propagation. It takes the trace on the left-hand side until step 7 to undo this decision.

1.4 The Rules Summarised

The following table summarises all the rules presented in the previous section. The first column gives the name of each rule, for identification purposes. The second column describes what the current partial valuation ought to look like before applying this rule. The fourth column describes what the partial valuation looks like after applying the rule. The final column describes what conditions must hold for this rule to be applicable.

DecideTrue	$v^?$	$\Longrightarrow$	$v^?, a :_d \mathsf{T}$	*if a is not assigned in $v^?$*
DecideFalse	$v^?$	$\Longrightarrow$	$v^?, a :_d \mathsf{F}$	*if a is not assigned in $v^?$*
Success	$v^?$	$\Longrightarrow$	$\mathsf{SAT}(v^?)$	*if $v^?$ makes all the clauses true.*
BackTrack	$v_1^?, a :_d x, v_2^?$	$\Longrightarrow$	$v_1^?, a :_f \neg x$	*if $v_2^?$ is decision free*
Fail	$v^?$	$\Longrightarrow$	UNSAT	*if $v^?$ is decision free, and makes at least one clause false.*
UnitPropTrue	$v^?$	$\Longrightarrow$	$v^?, a :_f \mathsf{T}$	*if there is a clause $C \vee a$ and $[\![C]\!](v^?) = \mathsf{F}$*
UnitPropFalse	$v^?$	$\Longrightarrow$	$v^?, a :_f \mathsf{F}$	*if there is a clause $C \vee \neg a$ and $[\![C]\!](v^?) = \mathsf{F}$*

Note that several of the rules may apply at the same time. For instance, the rules DecideTrue and DecideFalse will both be applicable if either one is. An actual implementation of a SAT solver will have some heuristic based strategy for choosing which one to pick in any given situation.

1.5 Two Examples

1. The following formula is satisfiable:

$$(\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge a$$

   If we take the strategy of assigning the atoms in the order a, b, c, d and always guessing T first, then we get the following steps:

1.	$[]$	*initial partial valuation*
2.	$[a :_f \mathsf{T}]$	*unit propagation (clause 5)*
3.	$[a :_f \mathsf{T}, b :_d \mathsf{T}]$	*guess*
4.	$[a :_f \mathsf{T}, b :_d \mathsf{T}, c :_f \mathsf{T}]$	*unit propagation (clause 1)*
5.	$[a :_f \mathsf{T}, b :_f \mathsf{F}]$	*backtrack (conflict in clause 3)*
6.	$[a :_f \mathsf{T}, b :_f \mathsf{F}, c :_f \mathsf{T}]$	*unit propagation (clause 4)*
7.	$[a :_f \mathsf{T}, b :_f \mathsf{F}, c :_f \mathsf{T}, d :_f \mathsf{T}]$	*unit propagation (clause 2)*

   So solver returns SAT and the satisfying valuation is $[a \mapsto \mathsf{T}, b \mapsto \mathsf{F}, c \mapsto \mathsf{T}, d \mapsto \mathsf{T}]$.

2. The following minor variation of the above formula is not satisfiable:

$$(\neg b \vee c) \wedge (\neg c \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge a$$

   (the d in the second clause has been replaced with a b). Using the same strategy we get the following steps, where the first six are the same:

1.	$[]$	*initial partial valuation*
2.	$[a :_f \mathsf{T}]$	*unit propagation (clause 5)*
3.	$[a :_f \mathsf{T}, b :_d \mathsf{T}]$	*guess*
4.	$[a :_f \mathsf{T}, b :_d \mathsf{T}, c :_f \mathsf{T}]$	*unit propagation (clause 1)*
5.	$[a :_f \mathsf{T}, b :_f \mathsf{F}]$	*backtrack (conflict in clause 3)*
6.	$[a :_f \mathsf{T}, b :_f \mathsf{F}, c :_f \mathsf{T}]$	*unit propagation (clause 4)*

   After line six we are stuck, because we have a partial valuation where all the decisions are *f*orced, but it makes all the clauses false. Therefore, we return UNSAT.