Planning and Reasoning

Sveva Pepe

Thursday 26th November, 2020

1 Unit Propagation with Horn clause

Unit Propagation is complete in the case of Horn clauses, where Horn clauses are defined as set of clauses where only one literal in a clause may be positive.

$$x \vee \neg y \vee w \quad \text{or} \quad \neg x \vee \neg z$$

We can see that we have at most one positive literal and the remaining ones are negative. When Unit Propagation stops there are not unit clauses left because otherwise the Unit propagation cannot stop and replace them, so at the end we have only clauses that contains at least two literals. If our clauses are Horn at most one is positive and at least one is negative.
The point is that by assign all variables to false we are sure that we satisfied all clauses. This method cannot work in general because we can have a formula like:

$$x \vee y$$

which is not satisfied by assign all variables to false.
This is why only in the particular case of Horn formula the mechanism of running Unit Propagation and then setting all unassigned variables to false works.

How to make the mechanism linear?
We would like to use Unit Propagation inside backtracking algorithm, so in every recursive call of backtracking mechanism we need to run Unit Propagation this mean that we want to be as efficient as possible, so we do not want Unit Propagation polynomial but linear. We will see in Section 1.1.1 the proof that Unit Propagation runs in linear time.

Example Horn CNF

$\{ x_1 \vee \neg x_2 \vee \neg x_3,\ \neg x_1 \vee x_2 \vee \neg x_3,\ x_4 \vee \neg x_5,\ x_5,\ \neg x_1 \vee x_3 \vee \neg x_4,\ \neg x_2 \vee \neg x_4 \vee \neg x_5 \}$

x_5 is a unit clause

set x_5=*true*

replace x_5 with *true*

$\{ x_1 \vee \neg x_2 \vee \neg x_3,\ \neg x_1 \vee x_2 \vee \neg x_3,\ x_4 \vee \cancel{\neg x_5},\ \cancel{x_5},\ \neg x_1 \vee x_3 \vee \neg x_4,\ \neg x_2 \vee \neg x_4 \vee \cancel{\neg x_5} \}$

simplify:

$\{ x_1 \vee \neg x_2 \vee \neg x_3,\ \neg x_1 \vee x_2 \vee \neg x_3,\ x_4,\ \neg x_1 \vee x_3 \vee \neg x_4,\ \neg x_2 \vee \neg x_4 \}$

set x_4=*true*

replace:

$\{ x_1 \vee \neg x_2 \vee \neg x_3,\ \neg x_1 \vee x_2 \vee \neg x_3,\ \cancel{x_4},\ \neg x_1 \vee x_3 \vee \cancel{\neg x_4},\ \neg x_2 \vee \cancel{\neg x_4} \}$

simplify:

$\{ x_1 \vee \neg x_2 \vee \neg x_3,\ \neg x_1 \vee x_2 \vee \neg x_3,\ \neg x_1 \vee x_3,\ \neg x_2 \}$

$\{ x_1 \vee \neg x_2 \vee \neg x_3,\ \neg x_1 \vee x_2 \vee \neg x_3,\ \neg x_1 \vee x_3,\ \neg x_2 \}$

clause $\neg x_2$ is unary

set x_2=*false*

replace:

$\{ \cancel{x_1 \vee \neg x_2 \vee \neg x_3},\ \neg x_1 \vee x_2 \vee \neg x_3,\ \neg x_1 \vee x_3,\ \cancel{\neg x_2} \}$

simplify:

$\{ \neg x_1 \vee \neg x_3,\ \neg x_1 \vee x_3 \}$

no unit clause

propagation stops

set remaining variables to *false*:
all clauses are binary and Horn,
so they contain at least a negative literal each
setting all variables to *false* makes them true

result is assignment $\{ x_1$=*false*, x_2=*false*, x_3=*false*, x_4=*true*, x_5=*true* $\}$

this is a model of the original CNF

We can see that when we do not have unit clause we just assign all remaining variables to false e solve it, obtianing a model.

1.1 Running time

In GSAT we just set a bound, so we decide how long to run. If we find some model that's fine, otherwise we just set the timeout in which we set number of iteration, so the number of restart and this will be the upper bound of the running time (*how long we want to try*).
In Unit propagation we can make it run in linear time, but this is reallly not obvious because the algorithm is really quadratic if you run it in the obvious way. This because we have to scan all formula, read all formula checking for unit clauses then we collect the unit clauses. After this we rescan all formula and do the replacement, and during this run we may collect the new unit clauses but at the same time we need to start again the new replacements.

Worst case for UP, dumb version

$\{ x_1 \vee \neg x_2,\ x_2 \vee \neg x_3,\ \dots\ x_{98} \vee \neg x_{99},\ x_{99} \vee \neg x_{100},\ x_{100} \}$

first scan produces $I'=\{x_{100}=true\}$

scan for replacing removes x_{100} and simplifies $x_{99} \vee \neg x_{100}$ to x_{99}

second scan produces $I'=\{x_{99}=true\}$

scan for replacing removes x_{99} and simplifies $x_{98} \vee \neg x_{99}$ to x_{98}

- two complete scans of the set for x_{100}
- two scans up to the second-last clause for x_{99}
- ...

100+99+98...=?

like a triangle, area is base*height/2

cost is in the order of n^2, where n is the number of variables

This is the *worst case*, in which the final is the only unary clause. The problem is that when the obvious algorithm runs on this particular formula check if that clause is unary untile the last one. The restart form the beginning if clauses contains x_{100}, only the second-last contains $\neg x_{100}$. So we can see that $x_{99} \vee \neg x_{100}$ becomes unary. We have a new unary clause and we restart from the beginning to do the replacing of this new unit clause. Do it again and again, and this is why this mechanism is consider quadratic, even though all clauses are binary.

1.1.1 How to make it linear?

The problem here is that we have to scan again the formula to find that the only clause in which the assignement is relevat it the clause before the unit clauses.
The idea to make it linear is that we have an assignement, we know the variable that we have to update and rather that scanning the all formula searching the clauses that contain the variable we should use indeces, so data structure that can tell us the clauses that we have to look for replace that variable. With data structure we have a kind of direct arrow, a pointer from variable assigned to the clauses that contain it. In this way i do not have to scan all clauses but only clauses that contain the variable assigned.

Example of Data structure

array of n lists, where n is the number of variables

each list contains the clauses that contain a variables

first, assign number to clauses:

$\{ x_1 \vee \neg x_2,\ x_2 \vee \neg x_3,\ \dots\ x_{98} \vee \neg x_{99},\ x_{99} \vee \neg x_{100},\ x_{100} \}$
$C_1 \quad C_2 \quad \dots C_{98} \quad C_{99} \quad C_{100}$

second, scan the set and add each clauses to the lists

x_1 : C_1
x_2 : C_1 C_2
...
x_{99} : C_{98} C_{99}
x_{100} : C_{99} C_{100}

$\{ x_1 \vee \neg x_2,\ x_2 \vee \neg x_3,\ \dots\ x_{98} \vee \neg x_{99},\ x_{99} \vee \neg x_{100},\ x_{100} \}$
$C_1 \quad C_2 \quad \dots C_{98} \quad C_{99} \quad C_{100}$
x_1 : C_1
x_2 : C_1 C_2
...
x_{99} : C_{98} C_{99}
x_{100} : C_{99} C_{100}

scan the set, get $x_{100}=true$

the list of x_{100} contains C_{99} and C_{100}

simplify these clauses

C_{99} becomes unary, set $x_{99}=true$

repeat

the data structure eliminates the need for scanning the set of clauses for simplifying

In the first step we just keep name of the clauses and store them somewhere. Then we scan the formulas and say which variables is contained in that clause. For instance , $x_1 : C_1$ means that x_1 is only contained in clause C_1. The advantage of this method is that when we are at the end, we can just go "straight" from variable that we assign, so x_{100} to the clauses that contain it, in this case we go directly to C_{99}.
In the general case the algorithm is not really linear, the algorithm is linear if all clauses have costant size.

2 DPLL

In modern implementation DPLL its just backtracking and Unit Propagation. Originally we have also a mechanism was called *clause subsumption*, but then was discovered that clause subsumption was not really useful. It was abandon several years ago. A *pure literal rule* was kind of useful but in some cases it was less efficient than Unit Propagation and interfered with optimization of Unit Propagation, so it is not so much use today.

2.1 Algorithm

Backtracking means that we have something that we can see to different values. Back tracking is a true recursive algorithm in which we set a variable to each possible value in turn and so for each value recursively repeat, run the algorithm.

algorithm:

1. choose a variable x_i
2. check satisfiability of F + (x_i=*true*)
3. check satisfiability of F + (x_i=*false*)

Algorithm to check satsfiability, and check satisfiability means to do two recursive calls. We assign a variable to true and false, then we check the satisfiability for true and then for false; and each check is a recursive call. Since the recursive call need a partial assignement, like $x_i = true$ and $x_i = false$, and the next assignement will be $x_j = true$ and $x_j = false$, then we need the additional parameter which is a partial assignement. This is not a full assignement it is just an assignement that it was initially empty.

Backtracking algorithm with partial interpretation

algorithm (some parts missing):

boolean sat(formula F, partial_interpretation I)

1. ... (see below)
2. choose x_i that I does not assign
3. return sat(F, I ∪ { x_i=true }) or sat(F, I ∪ { x_i=false })

satisfiability of F = satisfiability of F with I=∅

missing: base case of recursion, choice of x_i

This algorithm return a boolean value and it check the satisfiability of a formula with a partial interpretation that it is initially empty. Then we choose an unassigned variable and we establish the satisfiability of the formula by adding the assignement of the variable that we choose before in the case of true and then in the case of false. Of course if the first case, so satisfiability of the formula with $x_i = true$, is satisfiable we do not need to check the second case is we are just looking for a model.

Base case of recursion

recursion adds a x_i=*value* to I

at some point, all variables are assigned

we can now check whether F is true or false

but:

sometimes, we can check whether F is true or false even if some variables are still unassigned

The point is simple because if we can't choose a variable because I is a complete assignement then we stop. But thre is another possibility because this is a recursive algorithm so we call many times and everytime we do two calls and in each call we do two calls and so on and so forth. It is better to finish as early as possible because we can save possibly exponential number of recursive subcalls. Already with partial assignement we can stop because in some cases we can say that a formula is satisfiable or not.

For example just set the value of the variables and replace them, same that we do in Unit Propagation. Replace variables with their value. There are various possibility, for instance:

$I=\{x=true, z=false\}$

$F = \{ x \vee y, \neg x \vee \neg y \vee z \}$

replace variables with values:

$F = \{ x \vee y, \neg x \vee \neg y \vee z \} =$
$\{ true \vee y, \neg true \vee \neg y \vee false \} =$
$\{ true, \neg y \} =$
$\{ \neg y \}$

formula is not true nor false

value depends on the value of variable y

We can say that in the first recursive call we set x=true and z=false. We analize the branch of a truee in which x=true and z=false and then we analize the formula. We replace values of variables in the formula and we can see that the formula is not true and not false in the partial interpretation because the value of the formula depends on the missing variables y. I cannot say anythin about the satisfiability of the formula, we need to proceed.

$I=\{x=true, z=false\}$

$F = \{ \neg x \vee y, \neg x \vee z \}$

replace variables with values:

$F = \{ \neg x \vee y, \neg x \vee z \} =$
$\{ \neg true \vee y, \neg true \vee false \}$
$\{ false \vee y, false \vee false \} =$
$\{ y, false \vee false \} =$
$\{ y, false \}$

formula is **false**

all clauses have to be satisfied

even a single false clause implies that the formula is false

(even if the first clause were *true* instead of z, formula would have been false

This is the second case in which we have the same interpretation but different formula. We do the replacement and we can see that the second clause is already false and since the formula is a conjunction of clauses the formula is false. Even though the partial interpretation does not contain y the formula is already false.
In some cases, even if the assignement is partial, some formula could be already false.

$I=\{x=true, z=false\}$

$F = \{x \vee y \vee z, \neg y \vee \neg z\}$

$F = \{x \vee y \vee z, \neg y \vee \neg z\} =$
$\{true \vee y \vee false, \neg y \vee \neg false\} =$
$\{true \vee y \vee false, \neg y \vee true\} =$
$\{true, true\}$

all clauses are true

formula is true

This is the third case in which we have the same interpretation but different formula. Both clauses are already after the replacement even if we have a partial interpretation, so y is not assigned.

If the formula is true or false we do not need to go any further we just return true if the formula is satisfiable or false if it is not. If the formula is false the partial assignement cannot be extended to form a model. Thereis no way to make a model with this partial assignement.

Complete backtracking algorithm

boolean sat(formula F, partial_interpretation I)

- if ($I \Rightarrow F$) return true
- if ($I \Rightarrow \neg F$) return false
- choose x_i that I does not assign
- return sat(F, I $\cup$ { x_i=true }) or sat(F, I $\cup$ { x_i=false })

Most moder programming languages by default have this features that they do not evaluate boolean expression in FOL if the value is already determinant. For instance, if we have $a \vee b$ and if a=true then b is not evaluated at all. This is important in the case of two recursive call, so avoid to repeat the second call if the first is already true.

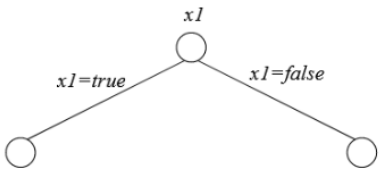
Example of Backtracking algorithm - satisfiability

$\{ \neg x_1 \vee \neg x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3 \}$

start with empty assignment $\{\}$

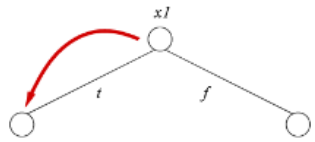
choose a variable, *for example* x_1

do two recursive calls with assignments $\{x_1=true\}$ and $\{x_1=false$



two recursive calls with assignments $\{x_1=true\}$ and $\{x_1=false\}$

We choose a variable that may could be x_1 but may be not because there is not a specific order. One of the elements that makes backtracking or DPLL efficient or not is exactly the choice of the branching layer. It is important but we do not have an obbligation to assign variables in order. It is not mandatory to choose x_1; in fact, sometimes x_1 is a good choice sometimes not.
What this diagram means is that every node is a recursive call and x_1 is the variable choosen in the recursive call, so the unassigned variable. In the first recursive subcall we try $x_1 = true$ and in the second recursive subcall we try $x_1 = false$. We have two recursive subcalls.



first recursive call is with assignment $\{x_1=true\}$

$\{ \neg x_1 \vee \neg x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3 \}$

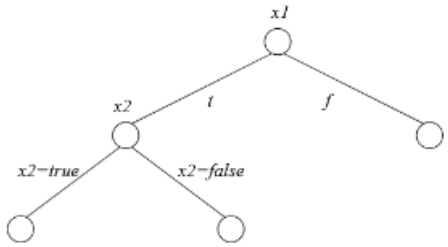
no clause of $\{\neg x_1 \vee \neg x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3\}$ is falsified by $\{x_1=true\}$

no contradiction: choose an unassigned variable

The point is that we move from x_1 to the true brach so with $x_1 = true$. We do the replacement and we have

$$\{\neg x_2, true, \neg x_3\}$$

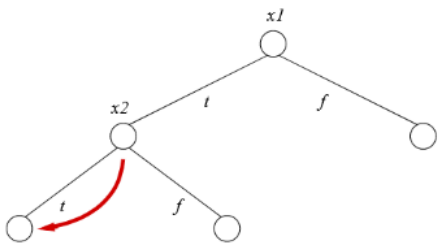
The formula is not true and not false, so we continue.
Let's say that the second variable that we choose is x_2, so we do the same thing that we did before. We create two branch: one for $x_2 = true$ and anothe for $x_2 = false$.
Now we have two subcalls so our assignements will be: $\{x_1 = true, x_2 = true\}$ and $\{x_1 = true, x_2 = false\}$.



branching variable x_2 (for example)

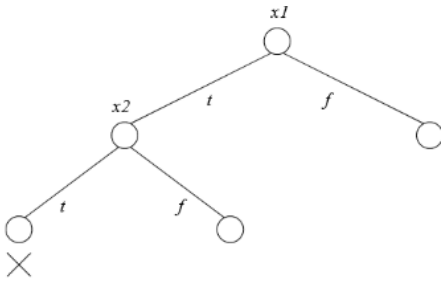
do two recursive calls adding the two possible evaluations of x_2 to the original one

partial interpretations in the recursive calls are then $\{x_1=true, x_2=true\}$ and $\{x_1=true, x_2=false\}$



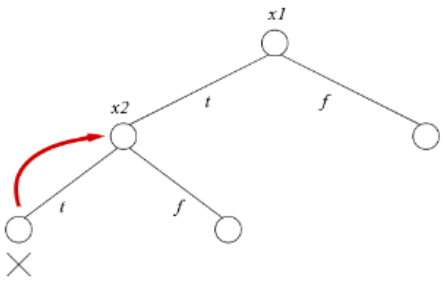
first recursive call with assignment $\{x_1=true, x_2=true\}$:

in $\{\neg x_1 \vee \neg x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3\}$, the clause $\neg x_1 \vee \neg x_2$ is falsified



contradiction, close branch of the tree

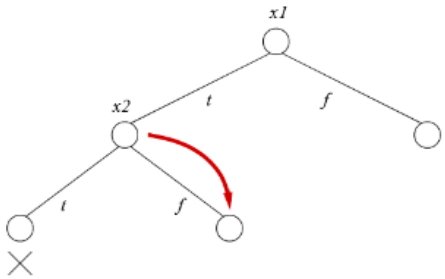
When we do the replacement for the interpretation $\{x_1 = true, x_2 = true\}$ we obtain: $\{false, true, \neg x_3\}$. We have a false, so we return false because we know that we have a conjunction of clauses so if at least one is false, the formula is false. We stop we do not proceed to do recursion of this branch because the formula is already false also with partial interpretation. We mark this with "cross" , that it is mean *unsatisfiable*, but this cannot mean that all formula is unsatisfiable it means that if I have this partial assignement the formula is unsatisfiable, but we could find another assignement in which the formula is satisfiable.



go back to node labeled x_2

x_2=*true* already tried

now try x_2=*false*

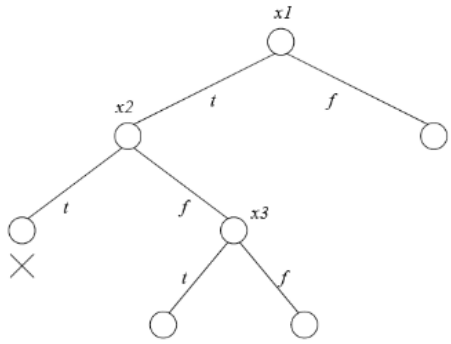


assignment $\{x_1$=*true*, x_2=*false*$\}$

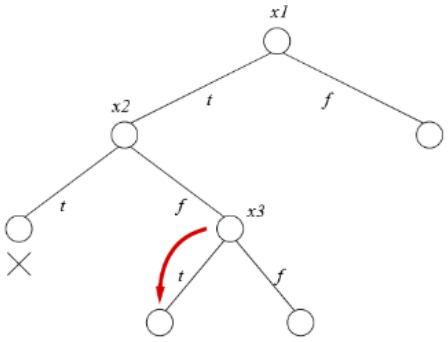
formula $\{\neg x_1 \vee \neg x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3\}$, is not falsifie

choose variable: only left unassigned is x_3

We go back to the node x_2 and we expand the other branch, so now we consider the partial interpretation $\{x_1 = true, x_2 = false\}$. We do the replacement and we see that we obtain $\{true, true, \neg x_3\}$. The formula is not true and not false, it depends on x_3. We proceed choosing x_3 that it is the last unassigned variable.



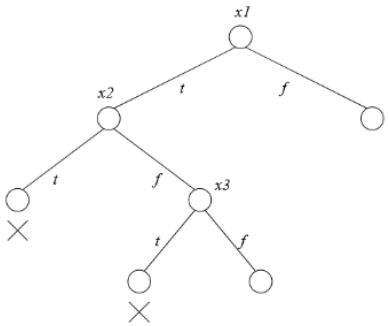
two recursive calls: x_3=*true*, x_3=*false*



first recursive call has assignment $\{x_1$=*true*, x_2=*false*, x_3=*true*$\}$

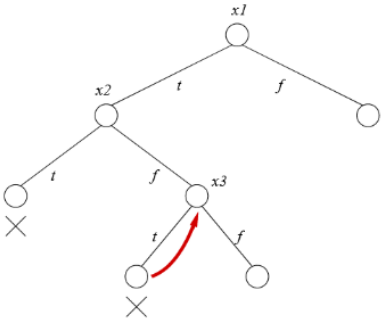
in formula $\{\neg x_1 \vee \neg x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3\}$, the clause $\neg x_1 \vee \neg x_3$ is falsified

In the case of the choice of $x_3 = true$, we have partial interpretation that it is $\{x_1 = true, x_2 = false, x_3 = true\}$. We do the replacement and we obtain that the formula is false because $\{true, true, false\}$.



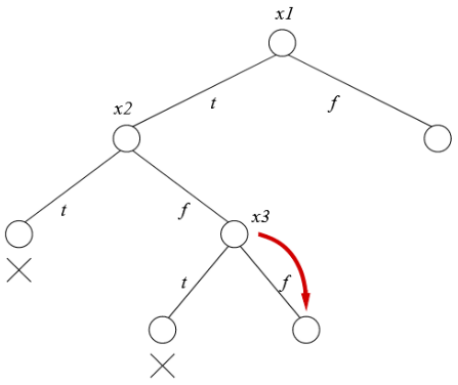
clause is falsified=formula is falsified

close branch



backtrack to node labeled x_3

We mark the cross and return unsatisfiable, so we go back and try for $x_3 = false$. What's important here is that at this point there is really no doubt because all variables are assigned, so there is no way, the formula should be true or false.



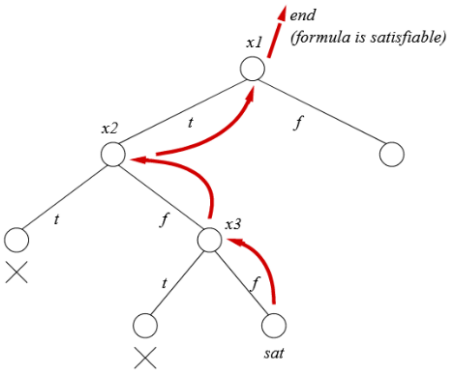
second recursive call for x_3

value x_3=*false*

assignment is {x_1=*true*, x_2=*false*, x_3=*false*}

all clauses in {$\neg x_1 \vee \neg x_2$, $x_1 \vee \neg x_2$, $\neg x_1 \vee \neg x_3$}, are satisfied!

$\neg x_1 \vee \neg x_2$
because x_2=*false*
$x_1 \vee \neg x_2$
because x_1=*true*
$\neg x_1 \vee \neg x_3$
because x_3=*false*



no other recursive calls

if a subcall returns *true*, the call returns *true* as well

this means: in this case, we go back to original call and return *true*

model found, no need to go ahead

formula is satisfiable

Now the assignement is $\{x_1 = true, x_2 = false, x3 = false\}$, and it is no more partial. We obtain that the formula is true because we do the replacement $\{true, true, true\}$ is the result. All clauses are satisfied.
What should we do now? We just go back and we do not need to try other possibilities because now when we return we return true and not false because we found a model. We just return that the formula is satisfiable. Of course if we want **all models** we should continue.

Another example - unsatisfiability

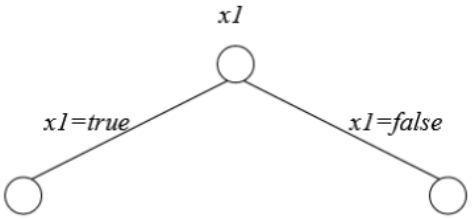
{ $\neg x_1 \vee \neg x_2$, $\neg x_1 \vee x_2$, $x_1 \vee \neg x_2$, $x_2 \vee \neg x_3$, $x_1 \vee x_3$ }

start with empty assignment

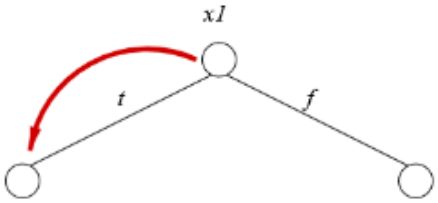
formula is not false under this interpretation

choose a variable

as an example, we choose x_1



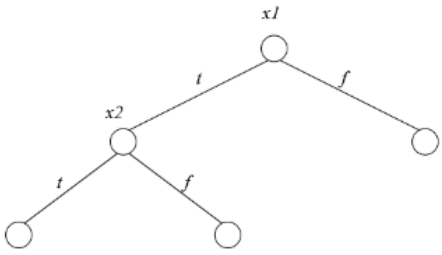
branch on x_1



first recursive calls with x_1=*true*

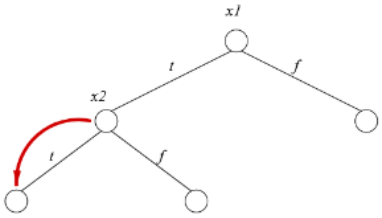
formula { $\neg x_1 \vee \neg x_2$, $\neg x_1 \vee x_2$, $x_1 \vee \neg x_2$, $x_2 \vee \neg x_3$, $x_1 \vee x_3$ } not made false by this assignment

choose an unassigned variable



as an example, we choose x_2

two other recursive calls, with assignments {x_1=*true*, x_2=*true*} and {x_1=*true*, x_2=*false*}



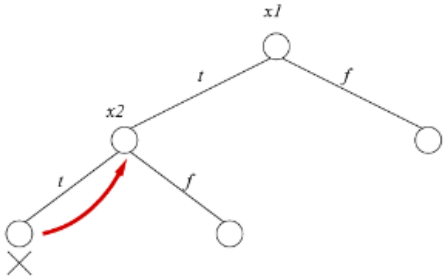
first recursive (sub)call:
assignment $\{x_1=true, x_2=true\}$

formula was $\{ \neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3 \}$

clause $\neg x_1 \vee \neg x_2$ false

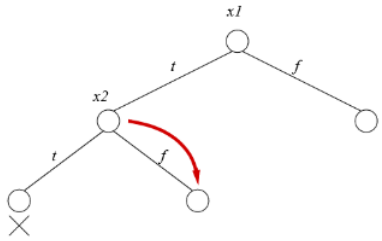
call returns *false*

no need to proceed any further, even if x_3 is still unassigned



recursion goes back to node marked x_2

partial assignment were $\{x_1=true\}$ there

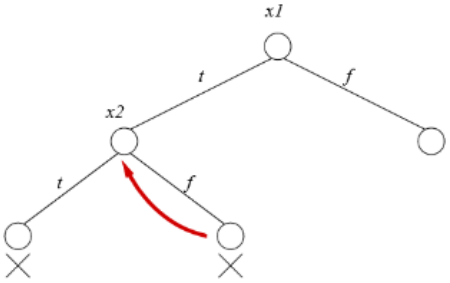


do second recursive (sub)call adding $x_2=false$ to $x_1=true$

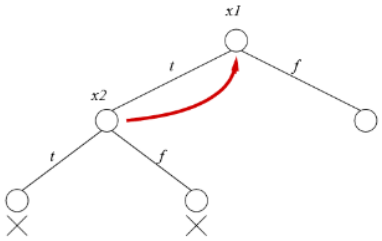
$\{ \neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3 \}$

clause $\neg x_1 \vee x_2$ false

close branch

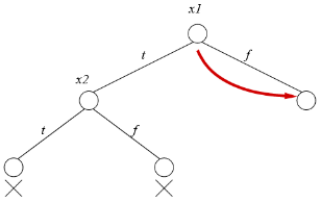


branch closed, go back to x_2



both recursive subcalls returned false, call returns false

go back to the first call, where $x_1=false$ is left to try

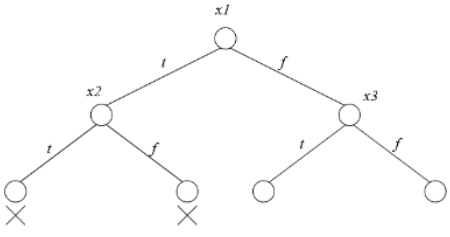


partial assignment is $\{x_1=false\}$

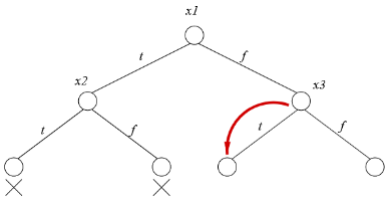
$\{ \neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3 \}$

formula is not false

choose a variable



as an example, we choose x_3

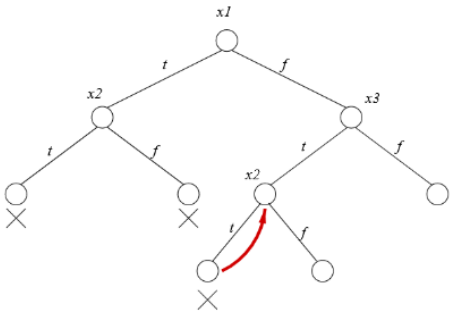


recursive call with partial assignment $\{x_1=false, x_3=true\}$

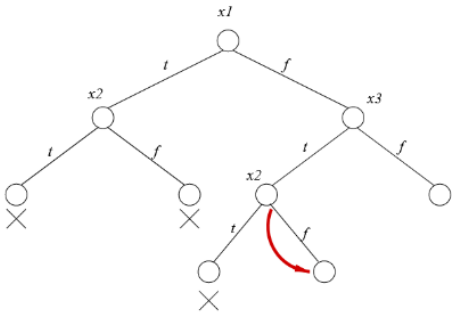
$\{ \neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3 \}$

formula is not false in this assignment

choose another variable and set it to *true* and *false*



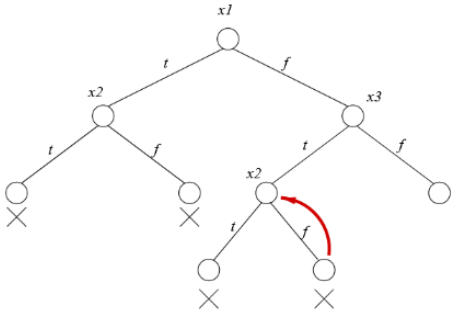
backtrack to x_2



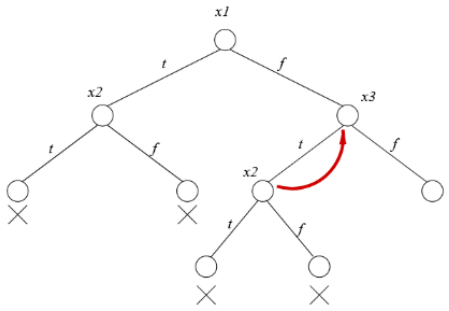
assignment $\{x_1$=*false*, x_3=*true*, x_2=*false*$\}$

$\{ \neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3 \}$

clause $x_2 \vee \neg x_3$ is falsified

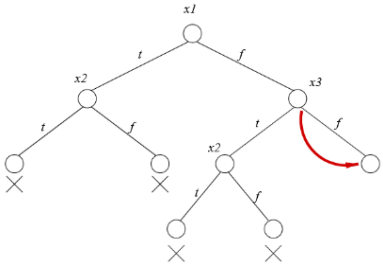


backtrack to x_2



both calls from node x_2 returned *false*

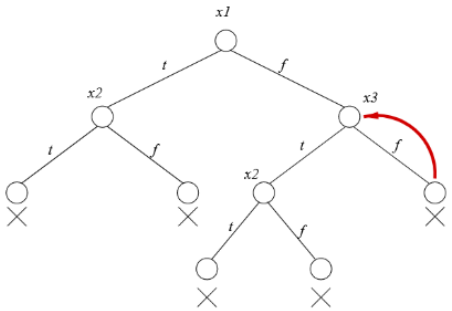
go back to node x_3



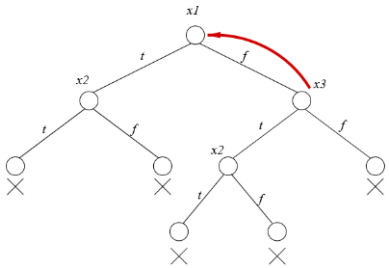
assignment $\{x_1$=*false*, x_3=*false*$\}$

$\{ \neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3 \}$

clause $x_1 \vee x_3$ falsified

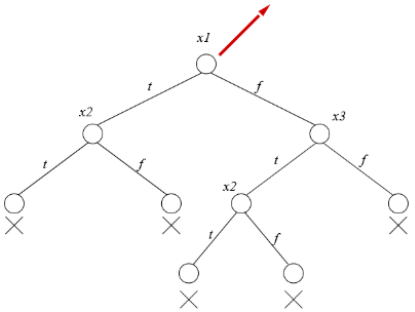


calls from x_3 both returned *false*



go back to x_1

we already tried x_1=*true* and x_1=*false*



return *false*

formula is unsatisfiable

If a formula is unsatisfiable Backtracking is typically longer because as soon we find a satisfy point we just return without considering the other recursive subcalls so it is the same as returning immediately. When a formula is satisfiable the problem on average is typically simple because we do less search because we stop as soon as we find a model.

N.B.
We can choose different variables in different branches, like in the previous example in which we have x_2 in the left branch of x_1 and x_3 in the right branch of x_1. Actually it is even simpler ketting the algorithm chooses branching literals differently because we do not have to maintain this information. Once the subtree of x_2 (in the left side) is over we do not need to save the order of the literals, the only information that we need is true or false. Only in the case of true we store also the model.

2.2 DPLL with Unit Propagation

boolean sat(formula F, partial_interpretation I)

- if ($I \Rightarrow F$) return true
- if ($I \Rightarrow \neg F$) return false
- **F,I = up(F,I)**
- **if I is inconsistent return false**
- choose x_i that I does not assign
- if sat(F, I ∪ { x_i=true }) return true
- if sat(F, I ∪ { x_i=false }) return true
- return false

extra advantage: UP may discover inconsistency

The idea is that we can simply apply the Unit Propagation. We just start it from using from the starting formula. So we have a formula, we run Unit Propagation and then so we run DPLL.
Why it is useful?
Because the execution of DPLL is a tree of recursive calls. This means that it is exponential in the number of variables. For instance, for the case of 3 variable the tree could have 8 nodes, $2^3 = 8$. So, when we remove a variable we are not just removing one variable, we are potentially cutting this tree enough. One variable less means half of the search tree.

The additions that we have are the things in bold. Unit propagation takes the formula and the partial interpretation and it modifies the formula and the partial interpretation because it removes the variables in the partial interpretation from the formula. If some unit clause occurs it also adds them into partial interpretation. The result is a new simplified formula and a modified improved possibly enlarge partial interpretation. We can check if the partial interpretation is unsatisfiable, if it occurs we stop and return. Otherwise, continue.
Remember that Unit Propagation is linear so it may only increase linear time for each recursive call.

Example DPLL with UP

$\{x_2 \vee x_1, \neg x_1, \neg x_2 \vee \neg x_3, x_3 \vee x_1\}$

up says x_1 is false

remove from clauses where occurs positive:

$x_2 \vee x_1$
  becomes $x_2 \vee \cancel{x_1}$, which is x_2
$x_3 \vee x_1$
  becomes $x_3 \vee \cancel{x_1}$, which is x_3

as a result, both x_2 and x_3 are true

clause $\neg x_2 \vee \neg x_3$ is contradicted

We start consider $x_1 = false$ so we apply Unit Propagation, we do the replacement and obtain a new formula that is

$$\{x_2, true, \neg x_2 \vee \neg x_3, x_3\}$$

As result both x_2 and x_3 should be true but there is a contraddiction so we stop and return false without consider any other branches. In this particular case, just by adding Unit Propagation I saved all recursive calls. Just 1 recursive calls instead of 9.

What happen if the original formula does not contain any unit clause?
We simply apply Unit Propagation and it does not do anything. We can still apply UP in the further processing of DPLL.

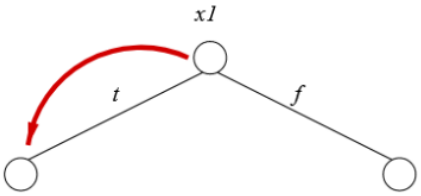
second of the examples above:

$\{\neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3\}$

no unit clause in the original set

two recursive calls

first recursive call with x_1=*true*



x_1=*true* is like an additional unit clause $\{x_1\}$

apply unit propagation

$\{\neg x_1 \vee \neg x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, x_2 \vee \neg x_3, x_1 \vee x_3\}$

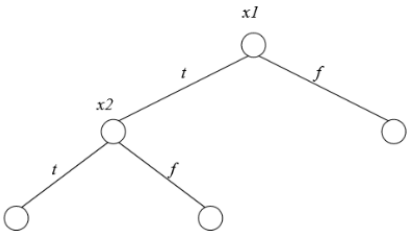
recursive call with x_1=*true*

remove x_1 where negative:

$\neg x_1 \vee \neg x_2$
$\not{\neg x_1} \vee \neg x_2$ becomes $\neg x_2$
$\neg x_1 \vee x_2$
$\not{\neg x_1} \vee x_2$ becomes x_2

contradiction is reached

recall that backtracking does a recursive call instead:



Here the formula does not contain any unit clause. We proceed normally like in the only backtracking case. Let's say for example that we choose and consider the branch in which w have $x_1 = true$. The important part is that $x_1 = true$ is basically the same as obtained in Unit Propagation, so even if the formula does not contain any unit clause I can still replace the variables with their value. We can do the semplification of the formula according with this assignement.
We obtain $\{\neg x_2, x_2, x_2 \vee \neg x_3, true\}$ and we can see that there is a contraddiction, we stop and return false. Instead of doing further recursive call already in this point without choosing another variable, just because $x_1 = true$ so propagating $x_1 = true$, we obtain a contraddiction so we save two recursive calls.

2.3 Pure Literal Rule

Another optimization that can be don in DPLL is the *pure literal rule*.

what about *a* in the following formula?

$\{a \vee \neg b \vee \neg c, a \vee c, b \vee \neg d\}$

a is only positive, it does not occur negative in this set. The point is that is I set $a = true$ we have only the clause $b \vee \neg d$ to satisfies because the others are already true. Setting $a = true$ is only an advantage because since our aim is to find a model, have two of three clauses that are satisfied. The variable a is called pure because it is always positive (same in the case of always negative).

in general (*a* not pure):

$\{a \vee \neg b \vee \neg c, a \vee c, b \vee \neg c, \neg a \vee b\}$

- *a=true* → a literal is made true in the first two clauses and false in the last
- *a=false* → a literal is made true in the last clause and false in the first two ones

if *a* is pure:

$\{a \vee \neg b \vee \neg c, a \vee c, b \vee \neg c\}$

- *a=true* → a literal is made true in the first two clauses
- *a=false* → a literal is made false in the first two clauses

setting *a=true* has some advantage and no disadvantage

In general is a is not pure when we set to true we satisfy $a \vee \neg b$ and $a \vee c$ but we make the clause $\neg a \vee b$ harder to be satisfied because we leave only one possibility to make it true.
But this is not the case in which a is only positive because we satisfy clauses $a \vee \neg b$ and $a \vee c$ and nothing changes on the rest of the formula.
In particular, b is not pure because we have $\neg b$ and b, so it occurs both positive and negative. But since a is only positive we remaining only with $b \vee \neg c$ and b is now pure because it only positive (also c is pure because it is only negative). For instance, we set b equal to true and the formula is satisfiable. Or we can set c = false and also in this case the formula is satisfiable.

The simplified formula may contains new pure literals. We do things just like in UP, check for pure literal, we simplify and repeat again.

complete algorithm:

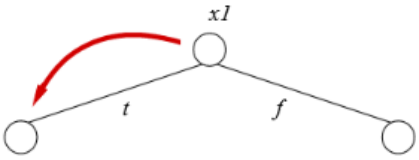
boolean sat(formula F, partial_interpretation I)

- if ($I \Rightarrow F$) return true
- if ($I \Rightarrow \neg F$) return false
- F,I = up(F,I)
- if I is inconsistent return false
- F,I = pure(F,I)
- if F = ∅ return true
- choose x_i that I does not assign
- if sat(F, I ∪ { x_i=true }) return true
- if sat(F, I ∪ { x_i=false }) return true
- return false

First we do UP and then apply the pure literal rule because UP may produce new pure literal but pure literal rule may produce new pure literal but not unit clauses because it only remove clauses not make them shorter.

DPLL full example

$\{ \neg x_1 \vee x_3 \vee x_4,\ \neg x_2 \vee x_6 \vee x_4,\ \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2,\ x_2 \vee \neg x_3 \vee \neg x_1,\ x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4,\ x_1 \vee x_5,\ x_1 \vee x_6,$
$\neg x_6 \vee x_3 \vee \neg x_5,\ x_1 \vee \neg x_3 \vee \neg x_5 \}$



choose branching variable x_1 (for example)

try x_1=*true* first

apply up and pure

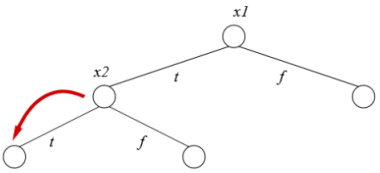
with $\{x_1$=*true*$\}$ the clauses become:

$\{ \Rightarrow\!\!\!\!\!\!\cancel{x_1} \vee x_3 \vee x_4,\ \neg x_2 \vee x_6 \vee x_4,\ \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2,\ x_2 \vee \neg x_3 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_1},\ x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4,\ \cancel{x_1 \vee x_5},\ \cancel{x_1 \vee x_6},$
$\neg x_6 \vee x_3 \vee \neg x_5,\ \cancel{x_1 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_3} \vee \Rightarrow\!\!\!\!\!\!\cancel{x_5}} \}$
=
$\{ x_3 \vee x_4,\ \neg x_2 \vee x_6 \vee x_4,\ \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2,\ x_2 \vee \neg x_3,\ x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4,$
$\neg x_6 \vee x_3 \vee \neg x_5 \}$

x_5 only occurs negated

can be set to false, removing clause

$\{ x_3 \vee x_4,\ \neg x_2 \vee x_6 \vee x_4,\ \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2,\ x_2 \vee \neg x_3,\ x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4 \}$



choose variable x_2, value *true* first

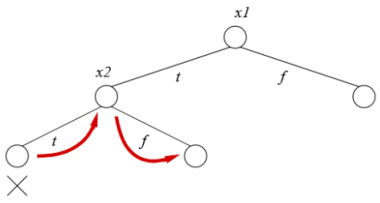
with $\{x_1$=*true*, x_2=*true*$\}$ the clauses become:

$\{ x_3 \vee x_4,\ \Rightarrow\!\!\!\!\!\!\cancel{x_2} \vee x_6 \vee x_4,\ \Rightarrow\!\!\!\!\!\!\cancel{x_2} \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_2},\ \cancel{x_2 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_3}},\ \cancel{x_2 \vee x_6 \vee x_3},$
$\cancel{x_2 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_6} \vee \Rightarrow\!\!\!\!\!\!\cancel{x_4}} \}$
=
$\{ x_3 \vee x_4,\ x_6 \vee x_4,\ \neg x_6 \vee \neg x_3,$
$\neg x_4 \}$

from $\neg x_4$ we derive x_3 and x_6

they falsify the clause $\neg x_6 \vee \neg x_3$

contradiction, no need to apply pure



contradiction reached, backtrack

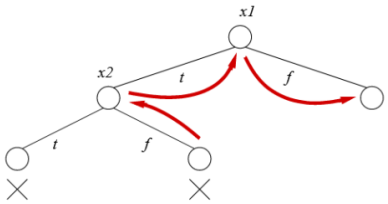
with $\{x_1$=*true*, x_2=*false*$\}$, clauses become:

$\{ x_3 \vee x_4,\ \Rightarrow\!\!\!\!\!\!\cancel{x_2 \vee x_6 \vee x_4},\ \Rightarrow\!\!\!\!\!\!\cancel{x_2 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_6} \vee \Rightarrow\!\!\!\!\!\!\cancel{x_3}},$
$\Rightarrow\!\!\!\!\!\!\cancel{x_4 \vee \Rightarrow\!\!\!\!\!\!\cancel{x_2}},\ x_2 \vee \neg x_3,\ x_2 \vee x_6 \vee x_3,$
$\cancel{x_2} \vee \neg x_6 \vee \neg x_4 \}$
=
$\{ x_3 \vee x_4,$
$\neg x_3,\ x_6 \vee x_3,$
$\neg x_6 \vee \neg x_4 \}$

from $\neg x_3$ we derive x_4 and x_6

they contradict clause $\neg x_6 \vee \neg x_4$

contradiction, no need to apply pure



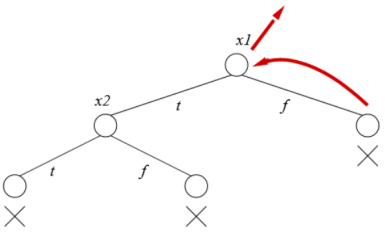
backtrack to first node, try other branch

with $\{x_1{=}false\}$ clauses become:

$\{ \cancel{\neg x_1 \vee x_3 \vee x_4}, \neg x_2 \vee x_6 \vee x_4, \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2, \cancel{x_2 \vee \neg x_3 \vee \neg x_1}, x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4, \cancel{x_1} \vee x_5, \cancel{x_1} \vee x_6,$
$\neg x_6 \vee x_3 \vee \neg x_5, \cancel{x_1} \vee \neg x_3 \vee \neg x_5 \}$
=
$\{ \neg x_2 \vee x_6 \vee x_4, \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2, x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4, x_5, x_6,$
$\neg x_6 \vee x_3 \vee \neg x_5, \neg x_3 \vee \neg x_5 \}$

from x_5 we derive $\neg x_3$

since x_6 is true, clause $\neg x_6 \vee x_3 \vee \neg x_5$ is falsified



contradiction reached on last node

set is unsatisfiable

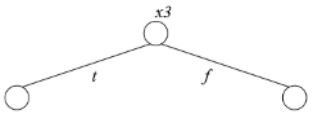
One of the things that was assumed to be important is the "sign" that we decide to choose, so usually we decide to expand "true" branch and then "false" branch. At the end was discovered that this order is not too much important, so you can choose the order that you want. Instead the choice of the variable is really important, it is crucial. The efficiency depends much on the choice of the variable.

Try the same example that we do before but choosing another variable

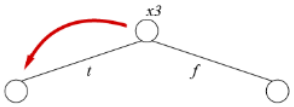
same set as previous example

$\{ \neg x_1 \vee x_3 \vee x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_2 \vee \neg x_6 \vee \neg x_3,$
$\neg x_4 \vee \neg x_2, x_2 \vee \neg x_3 \vee \neg x_1, x_2 \vee x_6 \vee x_3,$
$x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6,$
$\neg x_6 \vee x_3 \vee \neg x_5, x_1 \vee \neg x_3 \vee \neg x_5 \}$

choose x_3 first
then other variables



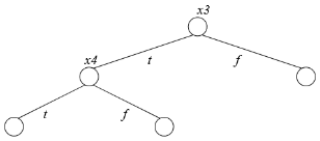
branch on x_3

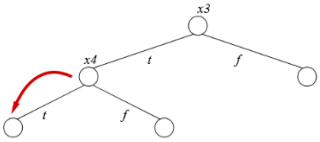


recursive call with $x_3{=}true$

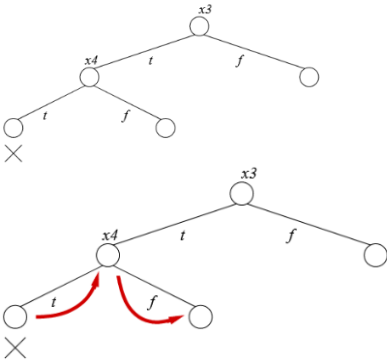
$\{ \cancel{\neg x_1 \vee x_3 \vee x_4}, \neg x_2 \vee x_6 \vee x_4, \neg x_2 \vee \neg x_6 \vee \cancel{\neg x_3}, \neg x_4 \vee \neg x_2, x_2 \vee \cancel{\neg x_3} \vee \neg x_1, \cancel{x_2 \vee x_6 \vee x_3}, x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6, \cancel{\neg x_6 \vee x_3 \vee \neg x_5}, x_1 \vee \cancel{\neg x_3} \vee \neg x_5 \}$
=
$\{ \neg x_2 \vee x_6 \vee x_4, \neg x_2 \vee \neg x_6, \neg x_4 \vee \neg x_2, x_2 \vee \neg x_1, x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$

no contradiction, choose a branching variable



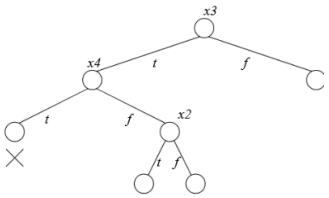


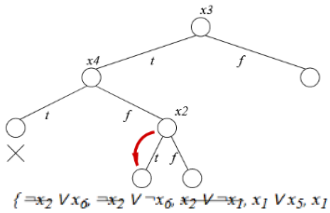
$\{ \cancel{\neg x_2 \vee x_6 \vee x_4}, \neg x_2 \vee \neg x_6, \cancel{\neg x_4} \vee \neg x_2, x_2 \vee \neg x_1, x_2 \vee \neg x_6 \vee \cancel{\neg x_4}, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
=
$\{ \neg x_2 \vee \neg x_6, \neg x_2, x_2 \vee \neg x_1, x_2 \vee \neg x_6, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
= (with $\neg x_2$)
$\{ \cancel{\neg x_2 \vee \neg x_6}, \cancel{\neg x_2}, \cancel{x_2} \vee \neg x_1, \cancel{x_2} \vee \neg x_6, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
=
$\{ \neg x_1, \neg x_6, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
=
contradiction: $\neg x_1, \neg x_6, x_1 \vee x_6$



$\{ \neg x_2 \vee x_6 \vee \cancel{x_6}, \neg x_2 \vee \neg x_6, \cancel{\neg x_4 \vee \neg x_2}, x_2 \vee \neg x_1, \cancel{x_2 \vee \neg x_6 \vee \neg x_4}, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
=
$\{ \neg x_2 \vee x_6, \neg x_2 \vee \neg x_6, x_2 \vee \neg x_1, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$

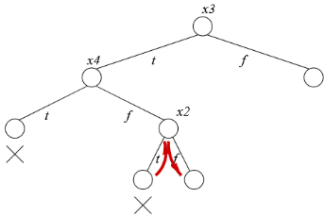
no contradiction, no unit clause





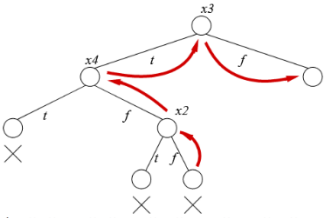
$\{ \neg x_2 \vee x_6, \neg x_2 \vee \neg x_6, x_2 \vee \neg x_1, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
=
$\{ x_6, \neg x_6, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$

contradiction



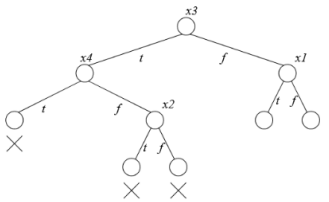
$\{ \neg x_2 \vee x_6, \neg x_2 \vee \neg x_6, x_2 \vee \neg x_1, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$
=
$\{ \neg x_1, x_1 \vee x_5, x_1 \vee x_6, x_1 \vee \neg x_5 \}$

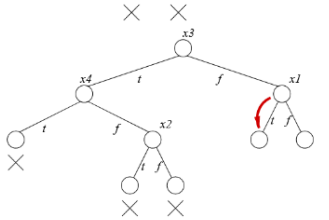
contradictin: from $\neg x_1$ both x_5 and $\neg x_5$ follow



$\{ \neg x_1 \vee x_3 \vee x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_2 \vee \neg x_6 \vee \neg x_3, \neg x_4 \vee \neg x_2, x_2 \vee \neg x_3 \vee \neg x_1, x_2 \vee x_6 \vee x_3, x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6, \neg x_6 \vee x_3 \vee \neg x_5, x_1 \vee \neg x_3 \vee \neg x_5 \}$
=
$\{ \neg x_1 \vee x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_4 \vee \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6, \neg x_6 \vee \neg x_5 \}$

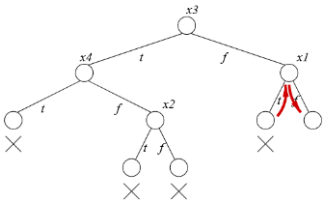
no contradiction, no unit clause





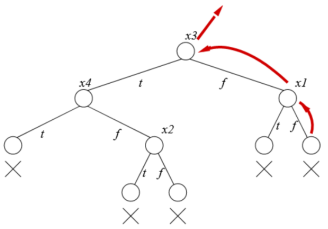
$\{ \neg x_1 \vee x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_4 \vee \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6, \neg x_6 \vee \neg x_5 \}$
=
$\{ x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_4 \vee \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6 \vee \neg x_4, \neg x_6 \vee \neg x_5 \}$
= (with x_4=*true*)
$\{ x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_4 \vee \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6 \vee \neg x_4, \neg x_6 \vee \neg x_5 \}$
=
$\{ \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6, \neg x_6 \vee \neg x_5 \}$

contradiction: $\neg x_2$ generates both x_6 and $\neg x_6$



$\{ \neg x_1 \vee x_4, \neg x_2 \vee x_6 \vee x_4, \neg x_4 \vee \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6 \vee \neg x_4, x_1 \vee x_5, x_1 \vee x_6, \neg x_6 \vee \neg x_5 \}$
=
$\{ \neg x_2 \vee x_6 \vee x_4, \neg x_4 \vee \neg x_2, x_2 \vee x_6, x_2 \vee \neg x_6 \vee \neg x_4, x_5, x_6, \neg x_6 \vee \neg x_5 \}$

contradiction: $x_5, x_6, \neg x_6 \vee \neg x_5$



set is unsatisfiable

Essentially we loosing a lot of time setting the wrong variable, like in this case. In general a bad choice of the branching literal may lead to an exponential larger tree of recursion. The same algorithm on the same formula just with a different choice of the variable take exponential time.
It is not immediately obvious that choosing x_1 is better than choose x_3.

Why x_1 is good and x_3 is bad?
Because was I'm trying to do is to minimize the number of recursive calls. I would like to produce a small tree. The point is that the number of subcalls depends on the number of unassigned variables because if I have few variables that are not assigned the recursive tree will be small, instead if I have many variables that are unassigned the tree will be on average larger. For instance, if I have 8 unassigned variable I will have $2^8 = 64$ leaves in the worst case. So, on average I can estimate the size of the subtree just by the number of variables that I have not assigned. The aim is to make UP to reduce this number as much as possible. This because if I have many UP in recursive call, many new values for new variables. For example, I choose x_1 and I get that 10 new variables are assigned this means that 10 variables less to be assigned by recursive call, less recursive subcalls.
What's important is that the binary clauses is that if a variable appear in 10 binary clauses it will probably produce 10 new assignement. The point is that if at the first step of UP we have a variable that when we assign produce many new assigned variables than the other choice we take it as our first choice.

It is a quick estimate for the number of binary clauses of how many variables will be assign and the number of assigned variables is an estimate of the size of the tree. We choose tha variables that it is in many binary clauses because this is fast to compute, fast to check how many binary clauses contain a variable. I should also try to balance the number of positive variables and negative variable in the binary clauses. For instance, if we have x_3 that appears positive in 10 binary clauses and negative in none binary clause, instead x_8 appear positive in 4 binary clause and negative in 4 binary clause we prefer to choose x_8.

Why?

x_3 (positive in 10, negative in none)
=true: zero
=false: 10
x_8 (positive in 4, negative in 4)
=true: 4
=false: 4

cost is exponential in the number of variables
assume 15 total

x_3
cost= $2^{15-1-10}+2^{15-1}= 2^4+2^{14}= 8+16384= 16394$
x_8
cost= $2^{15-1-4}+2^{15-1-4}= 2^{10}+2^{10}= 1024+1024= 2048$

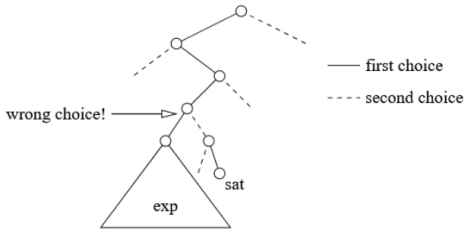
If we set $x_10 = false$ we have 10 new variables and for $x_10 = true$ I have no new variables. Instead, if $x_8 = true$ we have 4 new variables and the same for the case of $x_8 = false$.
Let's try to imagine that the number of recursive subcalls is really equal to the number of unassigned variables (15).
The cost (number of unassigned variables) will be exponential in the number of unassigned variables. For x_3 in one branche (false branch) we have $2^{15-1-10}$, 15- 11 because we assign 10 variables + the variable itself (x_3) and in the other branch (true branch) we have only 15-1 because we do not assign anything. The final cost for x_3 is $2^{15-1-10} + 2^{15-1} = 16394$. Instead for x_8 the cost is $2^{15-4-1} + 2^{15-4-1} = 2048$
The winner is x_8 because the number of recursive subcalls is less than x_3.

An old method consist in compute the number of binary clauses in which a variable is positive, p_i, and number of binary clauses in which the variables appear negative, n_i. And choose that variable that maximize $1024 \times p_i n_i + p_i + n_i$.

Finding the model
Let's assume that the formula is satisfiable, since the model exists we should try to find it as soon as possible. The idea would be try to estimate the value of each variable, so we make some calculation. For example, look at the formula and count the literals (positive and negative). Try to estimate the value of the first variable and then try other value, it is kind of guessing. The point is that this is totally ineffective, it doesn't increase the efficiency.

Why?
Because I'm wasting time because I'm try to guess so I have to be very luck to find immediately that formula is satisfiable, I have to make all choices right but we know that is almost impossible.

What happen if I make one choice wrong?



wrong choice=no model in the subtree

formula unsatisfiable with partial model

unsatisfiability: search tree may be exponential

therefore: most of the time spent on the unsatisfiable subformula

I could have an exponentially large tree.
Which part of this structure is the large part? Where do I spend most of the time?
Forgetting to the wrong choice I have a linear number of recursive calls but the number in the wrong choice could be exponential. The point is that if I'm right we do this in linear time, instead, if I'm wrong even once I'm potentially exponential. So, I'm spending a lot of time, effort, trying to guess the right value to choose first only to end up to have waste you time because still one wrong choice leads to exponentially large subtree. This means that in all recursive call I'm still evaluating true or false but for nothing because there is no model there.
Most of the time the algorithm runs on unsatisfiable subproblem, most of the time it spend not in the path of the model but in a subproblem where there is no solution.