

Section 4 Notes, January 30, 2004, by Kristin Branson

1 Problem Statement and Terminology

The DPLL and Walksat algorithms determine if an input CNF boolean formula is satisfiable. A formula is *satisfiable* if there is some assignment to the variables that results in the formula being true.

CNF stands for Conjunctive Normal Form, and means that the formula is a conjunction (that is, an AND) of any number of disjunctions (that is, an OR). Here is an example:

$$(a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d) \wedge (a \vee b \vee \neg d) \wedge (\neg a \vee \neg d \vee \neg d). \quad (1)$$

Each disjunction is called a *clause*. In the example, there are four clauses. In order for the formula to be satisfiable, each clause must be satisfied.

A *literal* is a variable or its negation. In the example, each clause contains three literals. If all clauses of a CNF formula contain at most three literals, then the formula is a 3-CNF formula. You can assume that the inputs for your assignment are 3-CNF.

2 The DPLL Algorithm

The DPLL algorithm chooses values for one variable at a time. After each assignment, it simplifies the set of clauses that must be satisfied. After each simplification, it checks to see if:

- It is evident that all clauses are satisfiable. If this is the case, return *true*.
- It is evident that some clause is not satisfiable. If this is the case, DPLL backtracks and tries a different value for an instantiated variable.

Here is pseudocode for a recursive algorithm that does this:

```
function DPLL(clauses, assignment)
1. If all clauses are satisfiable, then return true.
2. If some clause is not satisfiable, then return false.
3. // Simplify clauses using forward checking.
4.  $u \leftarrow \text{choose\_variable}(\textit{clauses}, \textit{assignment})$ .
5. Return DPLL(simplify(clauses, assignment,  $u = \textit{true}$ )) or
   DPLL(simplify(clauses, assignment,  $u = \textit{false}$ )).
```

A recursive DPLL algorithm is a good idea because, if the arguments are call-by-value, backtracking is easy. If a call returns unsuccessfully, the old values of the data structures are automatically restored.

How does DPLL simplify clauses? DPLL keeps track of a list of clauses that still need to be satisfied. For each of these clauses, it keeps track of the uninstantiated literals that, if true, will satisfy the clause. Thus, there are two ways that DPLL can simplify its list of clauses:

- For each clause, if one of its variables is instantiated so that the clause is true, then the clause can be deleted from the list of clauses that must be satisfied.
- If one of a clause's literals is instantiated to be false, then the clause can be simplified to be a shorter clause with just the remaining literals.

Example: Suppose we set $d = \textit{true}$ in formula (1). We can then remove all clauses which contain the literal d . This simplifies our formula to $(a \vee \neg b \vee c) \wedge (a \vee b \vee \neg d) \wedge (\neg a \vee \neg d \vee \neg d)$. We can also remove the literal $\neg d$ from all clauses. This simplifies our formula to

$$(a \vee \neg b \vee c) \wedge (a \vee b) \wedge (\neg a). \quad (2)$$

Here is pseudocode for simplifying clauses:

```
function simplify(clauses, assignment,  $u = v$ )
1. for each clause  $i$  satisfied by  $u = v$ ,
   remove_clause(clauses,  $i$ )
2. for each clause  $i$  satisfied by  $\neg u = v$ ,
   remove_literal(clauses[ $i$ ],  $\neg u$ )
3.  $\textit{assignment}[u] \leftarrow v$ 
4. return clauses, assignment
```

Given these simplifications,

- It is evident that all clauses are satisfiable if the list of clauses that still need to be satisfied is empty.

- It is evident that some clause is not satisfiable if the list of clauses contains an empty clause.

We can therefore update our DPLL pseudocode:

```

function DPLL(clauses, assignment)
1. If clauses is empty, then return true.
2. If some clause in clauses is empty, then return false.
3. // Simplify clauses using forward checking.
4. u ← choose_variable(clauses, assignment).
5. Return DPLL(simplify(clauses, assignment, u = true)) or
   DPLL(simplify(clauses, assignment, u = false)).

```

DPLL further simplifies the clauses by doing forward-checking, which for CNF-SAT is called *unit propagation*. After assigning a variable, it checks for unit clauses, which are clauses containing exactly one variable. When we have a unit clause, then the correct value of the variable in it is obvious. We can assign this variable this correct value, and simplify the clauses. Performing unit propagation with x , i.e. with $x = true$, causes further unit propagation for all binary clauses of the pattern $\neg x \vee y$.

Let us try applying unit propagation to formula (2). The last clause of the formula is a unit clause, so we can set its literal to true: $a = false$. We then must simplify the formula. We remove all clauses containing $\neg a$, resulting in $(a \vee \neg b \vee c) \wedge (a \vee b)$. We remove a from all clauses, resulting in $(\neg b \vee c) \wedge (b)$. Now there is another unit clause. We set $b = true$, and simplify the formula, resulting in (c) . We are left with another unit clause. We set $c = true$, and simplify, resulting in an empty set of clauses.

Here is some pseudocode for doing unit propagation:

```

function unit_propagation(clauses, assignment)
1. For each clause i in clauses,
   If clause i is a unit clause satisfied by  $u = v$ , then
     clauses, assignment ← simplify(clauses, assignment, u = v)
     return unit_propagation(clauses, assignment)
2. return clauses, assignment

```

Adding in the unit propagation step, we get the following algorithm:

```

function DPLL(clauses, assignment)
1. If clauses is empty, then return true.
2. If some clause in clauses is empty, then return false.
3. If clause of clauses is a unit clause, then
   return DPLL(unit_propagation(clauses, assignment, i)).
4. u ← choose_variable(clauses, assignment).
5. Return DPLL(simplify(clauses, assignment, u = true)) or
   DPLL(simplify(clauses, assignment, u = false)).

```

You are to come up with your own heuristic for choosing which variable to assign a value. An example heuristic was presented in class. For each variable u , this heuristic counted how many binary clauses remained after doing unit propagation with $u = true$ (call this $pc(u)$), and how many binary clauses remained after doing unit propagation with $u = false$ (call this $nc(u)$). Remaining binary clauses are good because then future applications of unit propagation will cause a cascade of simplifications. Thus, we want to choose variable u such that both $pc(u)$ and $nc(u)$ are high. One widely used heuristic is to maximize

$$score(u) = 1000 * pc(u) * nc(u) + pc(u) + nc(u).$$

The large constant 1000 makes score higher if $pc(u)$ and $nc(u)$ are equally large, instead of one being very large and the other being small. Note: there are also some corner cases that should be included, e.g. you want to choose a variable if one assignment results in an empty list of clauses or both assignments result in an empty clause.

3 DPLL Example

Let us try applying DPLL to the following formula:

$$(a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee c \vee \neg d) \wedge (\neg a \vee b \vee c). \quad (3)$$

The original list of clauses is not empty and does not contain an empty or unit clause. So, we choose a variable using the heuristic. To do this, we perform unit propagation with each variable set to each possible value. This results in

Variable	Value	Clauses	Count	Score
a	$true$	$(\neg b \vee \neg c) \wedge (c \vee \neg d) \wedge (b \vee c)$	3	
a	$false$	$(\neg b \vee \neg d)$	1	3004
b	$true$	$(a \vee \neg d) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee c \vee \neg d)$	2	
b	$false$	$(\neg a \vee c \vee \neg d) \wedge (\neg a \vee c)$	1	2003
c	$true$	$(a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg b)$	1	
c	$false$	$(a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg d) \wedge (\neg a \vee b)$	2	2003
d	$true$	$(a \vee \neg b) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg a \vee b \vee c)$	2	
d	$false$	$(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee c)$	0	2

Variable a has the highest score, so we first try to set $a = true$ and simplify the list of clauses. This results in $(\neg b \vee \neg c) \wedge (c \vee \neg d) \wedge (b \vee c)$. The list of clauses is not empty, and there are no empty or unit clauses, so we must choose another variable to assign. We calculate our heuristic for each assignment to each unassigned variable:

Variable	Value	Clauses	Count
b	$true$	empty list of clauses	0
b	$false$	empty list of clauses	0
c	$true$	empty list of clauses	0
c	$false$	empty list of clauses	0
d	$true$	empty list of clauses	0
d	$false$	$(\neg b \vee \neg c) \wedge (b \vee c)$	2

We see in our variable selection that setting b or c to any value results in an empty list of clauses. No matter which variable we choose, when we perform unit propagation, we will result in an empty list of clauses. Thus, we know that the formula is satisfiable. Another example formula:

$$(\neg a \vee b \vee c) \wedge (\neg a \vee d \vee \neg b) \wedge (\neg a \vee d \vee \neg c) \wedge (\neg a \vee \neg d \vee \neg b) \wedge (\neg a \vee \neg d \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \quad (4)$$

The original list of clauses is not empty, nor is there an empty or unit clause, so we must choose a variable to assign using our heuristic. To do this, we perform unit propagation with each variable set to each possible value. This results in

Variable	Value	Clauses	Count	Score
a	$true$	$(b \vee c) \wedge (d \vee \neg b) \wedge (d \vee \neg c) \wedge (\neg d \vee \neg b) \wedge (\neg d \vee \neg c)$	5	
a	$false$	$(b \vee c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$	3	15008
b	$true$	$(\neg a \vee d) \wedge (\neg a \vee d \vee \neg c) \wedge (\neg a \vee \neg d) \wedge (\neg a \vee \neg d \vee \neg c) \wedge (a \vee c) \wedge (a \vee \neg c)$	4	
b	$false$	$(\neg a \vee c) \wedge (\neg a \vee d \vee \neg c) \wedge (\neg a \vee \neg d \vee \neg c) \wedge (a \vee c)$	2	8006
c	$true$	$(\neg a \vee d \vee \neg b) \wedge (\neg a \vee d) \wedge (\neg a \vee \neg d \vee \neg b) \wedge (\neg a \vee \neg d) \wedge (a \vee \neg b)$	3	
c	$false$	$(\neg a \vee b) \wedge (\neg a \vee d \vee \neg b) \wedge (\neg a \vee \neg d \vee \neg b) \wedge (\neg a \vee \neg d) \wedge (a \vee b) \wedge (a \vee \neg b) \wedge (a \vee \neg b)$	4	12007
d	$true$	$(\neg a \vee b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c)$	2	
d	$false$	$(\neg a \vee b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c)$	2	4004

Variable a has the highest score, so we try setting $a = true$. This results in the simplified list of clauses $(b \vee c) \wedge (d \vee \neg b) \wedge (d \vee \neg c) \wedge (\neg d \vee \neg b) \wedge (\neg d \vee \neg c)$. Again, the list of clauses is not empty, nor are there any empty or unit clauses, so we must choose another variable with our heuristic. We try both assignments for the remaining variables:

Variable	Value	Clauses	Count
b	$true$	(empty clause)	0
b	$false$	(empty clause)	0
c	$true$	(empty clause)	0
c	$false$	(empty clause)	0
d	$true$	(empty clause)	0
d	$false$	(empty clause)	0

We see that there are no assignments to any of the variables that does not result in an empty clause, so no matter which variable we choose, after performing unit propagation we will result in an empty clause. Therefore, we backtrack and try $a = false$. This results in the simplified list of clauses $(b \vee c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$. The list of clauses is not empty, and there are no unit or empty clauses, so we must choose a variable to assign with our heuristic.

Variable	Value	Clauses	Count
<i>b</i>	<i>true</i>	(empty clause)	0
<i>b</i>	<i>false</i>	empty list of clauses	0
<i>c</i>	<i>true</i>	empty list of clauses	0
<i>c</i>	<i>false</i>	(empty clause)	0
<i>d</i>	<i>true</i>	$(b \vee c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$	3
<i>d</i>	<i>false</i>	$(b \vee c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$	3

We see that setting either $b = false$ or $c = true$ results in an empty list of clauses, so the formula must be satisfiable.

4 DPLL Data Structures

You will need to have efficient data structures to represent the current assignment to the variables and the current list of clauses. What sorts of demands are put on these data structures?

- The `assignment` data structure must have an entry for each variable. The assignment to a single variable can be one of three values: *true*, *false*, and *unassigned*.
- The `clauses` data structure must have an entry for each clause. This entry must keep track of the uninstantiated literals that will satisfy this clause. Also, it must be easy to tell if a clause is empty or a unit clause, or if all clauses have been satisfied.
- When a variable is assigned, we must update every clause containing that variable. The update is different for clauses containing the unnegated variable and clauses containing the negated variable. There should be some data structure keeping track of which clauses each variable appears negated and unnegated in.
- If you keep a stack of previous `assignment` and `clauses` states in order to backtrack, then you will have many copies of each of these data structures. Therefore, you will need a small data structure to represent each of these.