SAT Solver Notes

Tom

1 SAT Question

Before understanding the SAT Solver, it is first necessary to clarify what a SAT question is. Section 1.4 of the textbook describes what is propositional satisfiability: given a propositional formula, if there is an explanation that makes this A formula is said to be satisfiable if the truth value under this interpretation is true.

SAT problems (also known as Boolean satisfiability problems, propositional satisfiability problems, or SATISFIABILITY problems) are given a propositional formula and determine whether the formula is satisfiable. The SAT problem was the first known and very well-known NPC problem. The definition of the NPC problem is as follows:

- P-problem: A problem that can be answered in polynomial time is a P-problem.
- NP problem: A problem is NP if it is possible to determine in polynomial time whether the answer is correct or not, given the input and the answer. The P problem is a subset of the NP problem.
- NPC problem: The NPC problem is the hardest part of the NP problem. It is an NP problem by itself, but harder than all other NP problems.

Precisely because the SAT problem is an NPC problem, solving this problem has become an important task in computer science. mark.

2 SAT Solver

In this chapter, we will introduce the SAT Solver. First, we define what is SAT Solver, which is a class of programs that automatically solve SAT problems; second, we will introduce the judgment algorithm of the well-formed formula, which is used to determine the SAT

The validity of the input of the Solver; then, we will introduce the paradigm, the SAT Solver requires its input formula to conform to a certain type of paradigm, in order to improve the efficiency of the SAT Solver to solve the problem; finally, we will introduce DPLL, which is a commonly used SAT Solver. algorithm.

2.1 Definitions

A tool that automatically solves SAT problems is called a SAT solver, its input is a propositional logic formula, and the output is SAT (satisfiable) or UNSAT (unsatisfiable). When the SAT solver outputs the SAT, it also gives an explanation that makes the formula true under that explanation.

For example, given a formula: (P $\overset{\wedge}{ÿ}$ Q) $\overset{\vee}{ÿ}$ (¬P $\overset{\wedge}{ÿ}$ Q), SAT Solver will output SAT and give an explanation such that the truth value of this formula is true under this explanation, such as P = T, Q = T. Note that we do not require the SAT Solver to give all explanations, just one explanation. And for the formula (P $\overset{\wedge}{ÿ}$ Q) $\overset{\wedge}{ÿ}$ (¬P $\overset{\wedge}{ÿ}$ ¬Q), the SAT Solver outputs UNSAT because the truth of this formula is false under any interpretation.

2.2 Judgment algorithm of well-formed formula

Given an expression, we must first determine whether the expression is meaningful before we can start solving it, that is, to determine whether it is a well-formed formula (WFF). The well-formed formula is defined as follows:

1. Atomic propositions are well-formed formulas.

2. If A and B are well-formed formulas, then (¬A),(A ÿ B),(A ÿ B),(A ÿ B),(A ÿ B) are all well-formed official.

3. If and only if the symbol string composed of the above two rules is used for a limited number of times, it is a well-formed formula.

The definition here is slightly different from the textbook definition in the use of parentheses (we require double parentheses in the ¬ operation), but the essence is the same. The following describes an algorithm for automatically determining whether an expression is a well-formed formula. This algorithm is determined strictly according to the above definition, and more or less parentheses are not allowed. The input is an expression P and the output is true or false.

1. First build a tree with only one node, and the node is P.

2. Returns true if all nodes of the tree are atomic propositions.

3. Select a leaf node, which is not an atomic proposition, but an expression f.

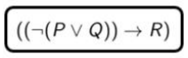
4. If f does not begin with an opening parenthesis or end without a closing parenthesis, return false.

5. If f = (¬Q), then add a new child node to the leaf node where f is located to store Q. Go to step 2

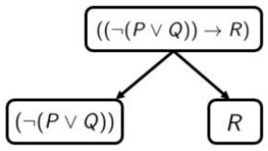
6. Now let f = (F), scan F from left to right until the first non-empty expression A is found such that the number of left and right parentheses of A is equal. If no such A is found, return false.

7. If f = (A ÿ B), ÿ ÿ {ÿ, ÿ, ÿ, ÿ}, then add two new child nodes to the leaf node where f is located, one is One for A and one for B. Go to step 2.

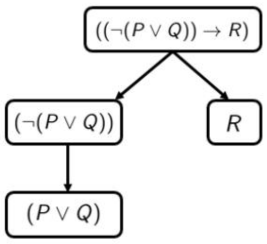
8. Return false.



The algorithm flow is described in detail below by taking ((¬(P ÿ Q)) ÿ R) as an example. First build a tree with only one node. Then we find that this node is not an atomic proposition, so go to step 3, f = ((¬(P ÿ Q)) ÿ R), f is surrounded by a pair of left and right parentheses, so 4 will not return false. The condition of step 5 is not satisfied, now go to step 6, F = (¬(P ÿ Q)) ÿ R, scan from left to right until the number of left and right parentheses is equal, and find A = (¬(P ÿ Q)). Step 7, find that B = Q, ÿ =ÿ, then add two new child nodes to the current node.



Now starting from step 2, it is found that there is no atomic proposition in the left leaf node, so now process this leaf node. Step 3, f = (¬(P ÿ Q)). In step 4, the expression is surrounded by left and right parentheses. Step 5, add a new child node to this leaf node, and put (P ÿ Q). Go to step 2.



Now starting from step 2, it is found that there is no atomic proposition in the left leaf node, so now process this leaf node. Step 3, f = (P ÿ Q). In step 4, the expression is surrounded by left and right parentheses. Step 5, the expression is not preceded by ¬, continue down. Step 6, F = P ÿ Q, A = P. Step 7, ÿ = ÿ, add two new child nodes to store P and Q. Go to step 2.

Now there are atomic propositions in all leaf nodes, and the output is true, which is a well-formed formula.

2.3 Paradigm

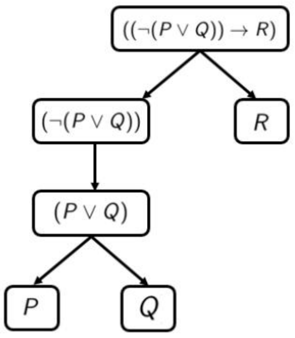
The input requirements for the SAT solver are in a standard form to facilitate algorithm design and solving. In propositional logic, this standard form is called normal form, including conjunctive normal form (CNF), disjunctive normal form (DNF), main normal form, etc. Please read Chapter 2.6 of the textbook carefully for this part of the content, and will not repeat it here.

The SAT solver generally uses CNF as the standard form of the input formula, and any formula can be converted into CNF. After changing the formula into CNF, the following DPLL algorithm can be used to solve it.

2.4 DPLL

Please read this section after reading 1.4 and 2.6 of the textbook.

The basic principle of the SAT solver is the DPLL algorithm, trying to find an explanation such that the true value of the formula is T. The basic idea of the DPLL algorithm is to guess the truth value of a literal (that is, the "literal" in the definition of textbook 2.6), and then according to this truth value



Derive the truth value of other literals. If the truth value of a certain clause of CNF is found to be F, it means that the previous truth value of a literal was wrongly guessed. At this time, backtracking is performed, and the truth value of that literal is re-guessed, and so on. , if you find that no matter how you guess, the truth value of the formula is F, indicating that the formula is unsatisfiable, otherwise it is satisfiable.

In the following explanation, we replace the ÿ of CNF with "," to represent, take A ÿ ¬B, B ÿ ¬C, C ÿ A as an example Explain the flow of the DPLL algorithm. DPLL mainly includes the following basic rules:

2.4.1 Decide Rule

If we have never identified the truth value of a literal, then the literal is called an undefined literal.
Decide rule is to say, select an undefined literal, guess its truth value, and mark this literal as a decision literal. For example, for A ÿ ¬B, B ÿ ¬C, C ÿ A, the truth value of A can be guessed as F according to the decide rule, and A is marked as a decision literal. Note that if you choose B or C first to guess the true value, it is also possible to guess the true value of A first.

2.4.2 Unitpropagate Rule

After a literal has a truth value, the truth value of other literals can be deduced. According to the unitpropagate rule, the truth value of A in A ÿ ¬B, B ÿ ¬C, C ÿ A is F, then it can be deduced that the truth value of B is F, because if and only if the truth value of B is F, A The truth value of ÿ ¬B can be T, and the truth value of the entire CNF formula can be T. Then it can be found that we can also deduce that the truth value of C is F, because in the current situation, if and only if the truth value of C is F, the truth value of B ÿ ¬C can be T, and the truth value of the entire CNF formula is only Possibly T.

2.4.3 Backtrack Rule

The backtrack rule means that if the truth value of a clause of the current CNF is found to be F, that is to say, when the truth value of the entire CNF is already F, it means that the assignment to a literal is wrong, and it is necessary to backtrack to find the nearest decision. Literal reassignment, the clause whose truth value is F is called conflicting clause. In this example, the truth value of A is F, the truth value of B is F, and the truth value of C is F. It is found that the truth value of the clause C ÿ A is F, and backtracking is performed. At present, the most recent decision literal is A, A The truth value guessed at the beginning is F. According to the backtrack rule, the truth value of A is re-guessed to be T, and A is marked as a non-decision literal, indicating that both truth values of A have been considered. Since the true values of B and C are deduced on the premise that the true value of A is F, the true values of B and C are now invalid, and should be re-derived according to these rules of DPLL.

2.4.4 Fail Rule

In the above example, since it is found that A is a decision literal, the backtrack rule can be used for backtracking. Fail rule means that if a conflicting clause appears but the decision literal cannot be found, it means that the formula is unsatisfiable, and unsat is output directly.

2.4.5 Pureliteral Rule

Carefully observe the example A ÿ ¬B, B ÿ ¬C, C ÿ A, we find that only A does not have ¬A, then we can directly think that the truth value of A is T, because if A is T, then all clauses containing A The truth value of A is T. If the truth value of A is T, the truth value of the formula is still F, then even if the truth value of A is changed to the truth value of the formula F, the truth value of the formula is still F. Similarly, if it is found that there is only ¬A in the formula without A, then the truth value of A is directly assumed to be F.

2.4.6 Summary

1. Use the pureliteral rule to determine the truth value of some literals, go to step 2.
2. Use the unitpropagate rule to derive as many literal truths as possible, and go to step 3.
3. If the true value of the formula is T at this time, go to step 4; if there is a conflicting clause, go to step 5. Otherwise, use the decision rule to guess the true value of an undefined literal, mark it as a decision literal, and then go to step 2.
4. Output sat and the truth value of each propositional variable, end.
5. See if there is a decision literal currently, if not, output unsat according to the fail rule, and end; otherwise, backtrack according to the backtrack rule, find the nearest decision literal, re-guess its true value, and mark it as a non-decision literal. Then go to step 2.

2.4.7 Examples

The following formula is used as an example to explain the flow of the DPLL algorithm.

ÿ ÿ 1 ÿ 2, $\overline{1}$ ÿ $\overline{2}$, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4

We use numbers to represent the different propositional variables, with $\overline{1}$ for ¬1. Write the truth value of each propositional variable that is currently considered On the left side of ÿ, ÿ means that the truth value of any propositional variable has not yet been determined, $\overline{1}$ means that the truth value of 1 is currently considered to be F, and $\overline{1}^d$ means that the current 1 is a decision literal.

Since the pureliteral rule cannot be used, the decision rule is used to guess that the true value of 1 is F.

ÿ ÿ 1 ÿ 2, $\overline{1}$ ÿ $\overline{2}$, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (Decide)

$\overline{1}^d$ ÿ 1 ÿ 2, $\overline{1}$ ÿ $\overline{2}$, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4

Machine Translated by Google

Then use the unitpropagate rule to derive as many literal truths as possible.

ÿ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (Decide)

$\overline{1}^{d}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4

ÿ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (Decide)

$\overline{1}^{d}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ $\overline{4}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4

ÿ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (Decide)

$\overline{1}^{d}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ $\overline{4}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ $\overline{4}$ 3 ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4

At this time, I found that the conflicting clause is 3ÿ$\overline{4}$, and we need to use the backtrack rule to backtrack. The most recent decision literal is 1, reguess that the truth value of 1 is T.

ÿ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (Decide)

$\overline{1}^{d}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (UnitP ropagate) 1 ÿ 2, 1 ÿ 2, 2 ÿ 3, 3 ÿ 4, 1 ÿ 4 ÿ

$\overline{1}^{d}$ $\overline{2}$ $\overline{4}$ ÿ (UnitP ropagate)

$\overline{1}^{d}$ $\overline{2}$ $\overline{4}$ 3 ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4 ÿ (Backtrack)

1 ÿ 1 ÿ 2, 1 ÿ $\overline{2}$, 2 ÿ $\overline{3}$, 3 ÿ $\overline{4}$, 1 ÿ 4

Then use the unitpropagate rule to deduce the truth value of other literals.

ÿ ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (Decide)

$\overline{1}^d$ ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^d$ $\overline{2}$ ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^d$ $\overline{2}$ $\overline{4}$ ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (UnitP ropagate)

$\overline{1}^d$ $\overline{2}$ $\overline{4}$ 3 ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (Backtrack)

1 ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (UnitP ropagate)

1 $\overline{2}$ ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (UnitP ropagate)

1 $\overline{2}$ 3 ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4 ÿ (UnitP ropagate)

1 $\overline{2}$ 3 4 ÿ 1 ÿ 2, $\overline{1}$ ÿ 2, $\overline{2}$ ÿ 3, $\overline{3}$ ÿ 4, 1 ÿ 4

It is found that the true value of the current formula is T, and output sat. If the true value of 1, 3, and 4 is T, and the true value of 2 is F, then the common

The true value of the formula is T.

3 Program Analysis

An important application of the SAT Solver is to analyze the correctness of programs. This section will explain how to convert the program to

Program analysis into propositional logic expressions. The program state is the value of each variable, and the execution of the program is modifying the program

State, analyzing a program is to determine whether a certain moment of program execution satisfies the properties the programmer wants. with the following procedure

For example,

```
void swap ( bool& a a = a^b ;         bool& b) {

    b = a^b ;
    a = a^b ;
}
```

The program state is two bool variables a and b, so the initial program state state0 can use two propositional variables

A0 and B0 represent, represent the current value of a and b respectively. state0 executes the first line of code to become state1, state1 uses

A1 and B1 represent the current values of a and b such that A1 ÿ (A0 ÿ ¬B0) ÿ (¬A0 ÿ B0) and B1 ÿ B0. This

In this way, each time a statement is completed, the current values of a and b can be represented by the previous program state. And so on, I

We can convert the program into the following propositional logic formula.

(A1 ÿ (A0 ÿ ¬B0) ÿ (¬A0 ÿ B0))ÿ
(B1 ÿ B0)ÿ
(B2 ÿ (A1 ÿ ¬B1) ÿ (¬A1 ÿ B1))ÿ
(A2 ÿ A1)ÿ
(A3 ÿ (A2 ÿ ¬B2) ÿ (¬A2 ÿ B2))ÿ
(B3 ÿ B2)

Now prove that the values of a and b must be interchanged, that is, the above formula can deduce that the value of a is equal to the initial value of b value, the value of b is equal to the initial value of a, expressed as the following formula.

(A1 ÿ (A0 ÿ ¬B0) ÿ (¬A0 ÿ B0))ÿ
(B1 ÿ B0)ÿ
(B2 ÿ (A1 ÿ ¬B1) ÿ (¬A1 ÿ B1))ÿ
(A2 ÿ A1)ÿ
(A3 ÿ (A2 ÿ ¬B2) ÿ (¬A2 ÿ B2))ÿ
(B3 ÿ B2)ÿ
ÿ
((A3 ÿ B0) ÿ (B3 ÿ A0))

To prove that the values of a and b must be interchanged is to prove that the formula is always true, and it is to prove that the negation of this formula is unsatisfiable. The next step is to hand it over to the SAT solver.

Consider the if statement next.

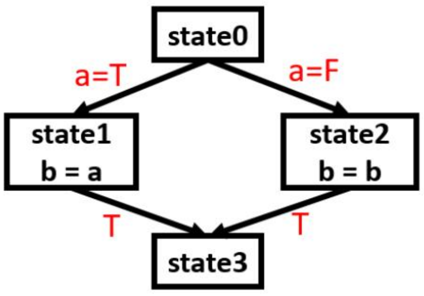
```
void swap ( bool a i    bool b) {
   f ( a ) b = a ;

   e l s e
      b = b ;
}
```

This program can be represented

as the following figure. If a of state0 is T, then b of state3 is equal to b of state1, otherwise b of state3 is equal to state2 , so the program turns into the following formula.

(B1 ÿ A0)ÿ
(B2 ÿ B0)ÿ
(B3 ÿ (A0 ÿ B1) ÿ (¬A0 ÿ B2))



If you want to prove that b must be T in the end, then prove that the following formula is always true.

(B1 ÿ A0)ÿ

(B2 ÿ B0)ÿ

(B3 ÿ (A0 ÿ B1) ÿ (¬A0 ÿ B2))

ÿ

B3

That is to say, the negation of this formula is proved to be unsatisfactory, and it is handed over to the SAT solver, and it is found that when A0=F, B0=F, b is finally F.

Consider the loop below. If you know that the loop executes at most n times, you can unroll the loop n times, so that it becomes a program without loops, which can be solved by the previous method. If you don't know how many times the loop is executed at most, you can directly expand it k times, where k is an integer set by yourself, which can also be analyzed, but may miss bugs. If there is an int variable, since the computer uses 32 bits to store an int, 1 bit corresponds to 1 proposition variable,

So an int value can be represented by 32 propositional variables. For the following program,

```
void swap ( i n t& a a = a^b ; b =      i n t& b) {
    a^b ; a = a^b ;



}
```

A program state consists of 64 propositional variables, and the method of converting the program into a logical expression is the same as before.

The operators in the program can be represented as digital circuits in the computer, operating on {0,1} bits, so the operators naturally It can also be expressed as a propositional logic expression.