Boolean SAT-Solver

Anshul Gupta (16305R001)
Khursheed Ali (163059009)

November 22, 2017

Contents

Chapter 1

Introduction

1.1 Problem Statement

Solving real world problem using SAT Solver.

1.2 Motivation

SAT is once of the hardest problems in computing theory. It lies in NP-Complete domain that means all the problems in NP and NP-Hard are reducible to SAT in polynomial time. SAT lies at the core of many practical application domains including Automatic theorem proving, automatic test generation and so on.

Chapter 2

Approach

2.1 Implementation

We have implemented two algorithm one "Brute Force" another one "Davis-Putnam-Logemann-Loveland (DPLL)". Brute Force gives worst result with respect to time and boolean size. If the boolean formula has too many variables but less clause then it will take too much of time. But DPLL [2] reduces the clause while it progress. For a given formula it checks whether it is already *satisfiable* or not. If its is not then it will take (unassigned) literal and assign True False value it. Then it *simplifies* the current formal and recurse on the update formal. In paper [2] they have discussed about choose best decision (unassigned) literal such that formula can be simplified as much as possible

2.2 Input and Data types

Input to our program is "CNF" (conjunction normal form) Boolean formula.
Eg: $(a \vee b \vee -c) \wedge (-b \vee d)$
Haskell Representation: [[(P, "a") ,(P, "b"),(N, "c")] , [(N, "b") ,(P, "b")]]
P :: True value of variable N:: Negated value of variable

```
data Boolean = T | F | ND deriving (Show, Eq, Read)
-- ND:: Not Define. When sufficient variables are not set in Formula then "eval"
-- will return ND
data Sign = P | N deriving (Show, Eq)
type Literal = (Sign, Symbol)
type State =Literal->Boolean
-- State:: State will maintain values (P/F/ND) associated with each literal o
-- clause
```

2.3 Algorithm: Evaluator

Eval will check whether for a given state whether the Boolean formula give T/F/ND eval::Clauses -> State->Boolean

```
eval clauses state = if (F `elem` result) then
                            return F
                     else if (ND `elem` result) then
                            return ND
                     else return T
    where result=[ (subEval sc state)  | sc<-clauses]  ( sc:: sub clauses
```

Algo 1: bruteForceSolver::Clauses->Symbols->State->SatResult
Algo 2: dpllSolver: :Clauses->State->SatResult

2.4 Alorithm: Bruteforce

BruteForceSolver(clauses,symbols,state)

```
   if  symbols == [] then evalSat clauses state
    -- Checking Satisfibility & setting sym to True
   if trueSymEval == T  then convertToSatResult T (update state (sym,T))
   -- Checking Satisfibility & setting sym to False
   if falseSymEval== T  then convertToSatResult T (update state (sym,F))
   -- When less symbols are set for evaluation
   else if (fst trueSymSolver) == Satisfiable
                then trueSymSolver
                else if (fst falseSymSolver) == Satisfiable
                then falseSymSolver
                else convertToSatResult F state
   where
   sym= head symbols
   trueSymEval = eval clauses (update state (sym,T))
   falseSymEval= eval clauses (update state (sym,F))
   trueSymSolver=bruteForceSolver clauses (tail symbols) (updatestate (sym,T))
   falseSymSolver=bruteForceSolver clauses (tail symbols) (update state (sym,F))
```

2.5 Algorithm: DPLL

DPLL (C) C:: Clauses in CNF Form

```
    If C has no clause then,
        return True;
    If C has any empty clause then
        return False;
```

```
UC <- UnitClause (C)
If UC has any literal with its negation
    return False
If there is any UC then
    return DPLL ( Simplify( C,  literal of UC ) )
sym <- choose any literal from C
If DPLL (Simplify ( C, (sym,True) ) ) is True then,
     return true
Else return DPLL (Simplify ( C, (sym, False) ) )
```

Simplify (C , (l, B))
C:: Clause, l:: literal B:: True/False

```
Remove clauses from C where l is B
Remove (not B) l from every clauses of C
Return update C
```

UnitClause (clauses):: Clauses-¿Clause

```
return [ head c | c<-clauses,length c == 1]
```

Chapter 3

Sudoku

3.1 Reducing Sudoku to SAT

To encode the Sudoku problem in SAT, we need a notation to represent a cell and its value [1].

A cell is denoted as an integer of 3 digits. It can be positive or negative. The first digit represents the row number, second digit represents the column number and the third digit represents the value of the cell. A negative value denotes that the cell cannot have that value.

Example 1: 143 represents row 1 column 4 has a value of 3

Example 2: -143 represents row 1 column 4 shouldn't have 3

Clauses are given in CNF. Each clause in written in a new line and literals in a conjunction are separated by space.

3.2 Sudoku Constraints

Figure 3.1 shows the notation of Sudoku. Constraints are:

- **A cell should have at most one value between 1 and 9**
- **A cell should have at least one value between 1 and 9**
- **A number appears at most once in each row**
- **A number appears at least once in each row**
- **A number appears at most once in each column**
- **A number appears at least once in each column**
- **A number appears at most once in each box**
- **A number appears at least once in each box**

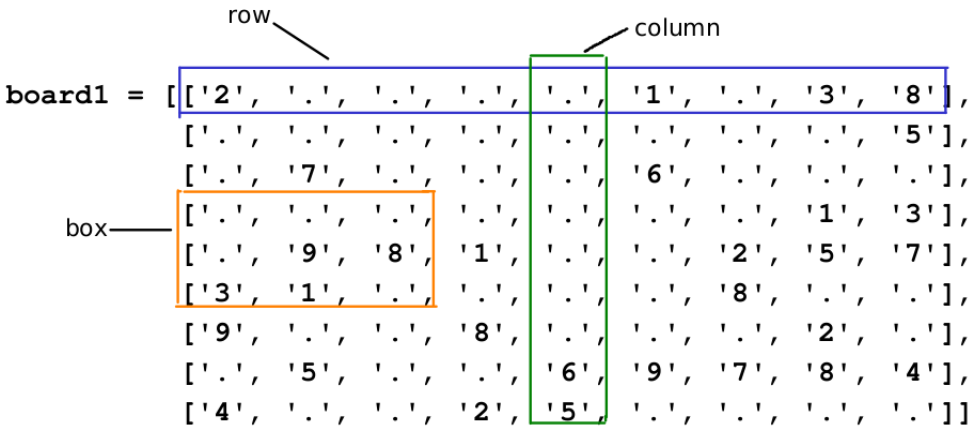

Figure 3.1: Sudoku Constraints

3.2.1 Cell Constraints

$$\wedge_{x=1}^{9} \wedge_{y=1}^{9} \vee_{z=1}^{9} xyz \tag{3.1}$$

$$\wedge_{x=1}^{9} \wedge_{y=1}^{9} \wedge_{z=1}^{8} \wedge_{i=z+1}^{9} -xyz \vee -xyi \tag{3.2}$$

3.2.2 Row Constraints

$$\wedge_{x=1}^{9} \wedge_{z=1}^{9} \wedge_{y=1}^{8} \wedge_{i=y+1}^{9} -xyz \vee -xiz \tag{3.3}$$

$$\wedge_{y=1}^{9} \wedge_{z=1}^{9} \wedge_{z=1}^{9} xyz \tag{3.4}$$

3.2.3 Column Constraints

$$\wedge_{y=1}^{9} \wedge_{z=1}^{9} \wedge_{x=1}^{8} \wedge_{i=x+1}^{9} -xyz \vee -iyz \tag{3.5}$$

$$\wedge_{x=1}^{9} \wedge_{z=1}^{9} \wedge_{y=1}^{9} xyz \tag{3.6}$$

3.2.4 Box Constraints

$$\wedge_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} \wedge_{k=y+1}^{3} -(3i+x)(3j+y)z \vee -(3i+x)(3j+k)z \tag{3.7}$$

$$\wedge_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} \wedge_{k=x+1}^{3} \wedge_{l=1}^{3} -(3i+x)(3j+y)z \vee -(3i+x)(3j+k)z \tag{3.8}$$

$$\vee_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} (3i+x)(3j+y)z \tag{3.9}$$

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 3.2: Unsolved Sudoku pic: https://en.wikipedia.org/wiki/Sudoku

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 3.3: Solved Sudoku pic: https://en.wikipedia.org/wiki/Sudoku

Bibliography

[1] **Inês Lynce and Joël Ouaknine. Sudoku as a sat problem. In *In Proc. of the Ninth International Symposium on Artificial Intelligence and Mathematics*. Springer, 2006.**

[2] **M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, June 2001.**