SAT SOLVER

EFFICIENT WAY TO SOLVE NP-COMPLETE PROBLEM

- ANSHUL GUPTA (16305R001)

- KHURSHEED ALI (163059009)

Content

- Paper (**CHAFF**)
- DPLL Algorithm
- Implementation
- Reducing Sudoku to SAT

CHAFF : Engineering an Efficient SAT Solver

Chaff, gains performance through careful engineering of all aspects of the search –

- Efficient implementation of Boolean constraint propagation – DPLL
- A low overhead decision strategy i.e. heuristic based search – RAND, VSDIF
- RAND: Simplest possible strategy for decision assignment is RAND (Random). This is simpler to implement but it will rate of reducing the Formula into smaller formula is less
- Variable State Independent Decaying Sum (VSIDS) Decision Heuristic:
  - Decision assignment consists of the determination of which new variable and state should be selected each time.
  - Each variable in each polarity has a counter, initialized to 0
  - The (unassigned) variable and polarity with the *highest counter* is chosen at each decision.

Link: https://www.princeton.edu/~chaff/publication/DAC2001v56.pdf

Davis-Putnam-Logemann-Loveland 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 <- UnitClauses (C)

If UC has any literal with its negation, then 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

Implementation – SAT SOLVER

- Input to our program is "CNF" (conjunction normal form) Boolean formula.

Eg: (a V b V -c) AND (-b V d) ---> (Haskell) [[(P, "a") ,(P, "b"),(N, "c")] , [(N, "b") ,(P, "b")]]

P :: True value of variable N:: Negated value of variable

- New Data Type

```
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
```

Here "State" will maintain values (P/F/ND) associated with each literal of clause.

Implementation - SAT

eval::Clauses->State->Boolean

eval clauses = \state. if (F `elem` result) then F else (if (ND `elem` result) then ND else T)

where result=[(subEval sc state) | sc<-clauses] (sc:: sub clauses)

"eval" will check whether for a given state whether the Boolean formula give T/F/ND

Algo 1: bruteForceSolver'::Clauses->Symbols->State->SatResult

Algo 2: dpllSolver'::Clauses->State->SatResult

Reducing Sudoku to SAT

The notation we use to encode a cell and its value is a integer of 3 digits: XYZ
X is the row, Y is the Column and Z is the Value
A negative integer –XYZ represents that cell on row X and column Y cannot have a value Z

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

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

Paper: https://www.inesc-id.pt/ficheiros/publicacoes/2777.pdf

Image Ref: https://en.wikipedia.org/wiki/Sudoku

Sudoku Constraints

row

column

box

```
board1 = [['2', '.', '.', '.', '.', '1', '.', '3', '8'],
          ['.', '.', '.', '.', '.', '.', '.', '.', '5'],
          ['.', '7', '.', '.', '.', '6', '.', '.', '.'],
          ['.', '.', '.', '.', '.', '.', '.', '1', '3'],
          ['.', '9', '8', '1', '.', '.', '2', '5', '7'],
          ['3', '1', '.', '.', '.', '.', '8', '.', '.'],
          ['9', '.', '.', '8', '.', '.', '.', '2', '.'],
          ['.', '5', '.', '.', '6', '9', '7', '8', '4'],
          ['4', '.', '.', '2', '5', '.', '.', '.', '.']]
```

Sudoku Constraints

1. Cell Constraints
2. Row Constraints
3. Column Constraints
4. Box Constraints

Cell Constraints

- A Cell can hold a single value from 1 to 9 at a time
- $\Lambda_{x=[1-9]}\ \Lambda_{y=[1-9]} V_{z=[1-9]}$ xyz
- Eg.

111 112 113 114 … 119

121 122 123 124 … 129

……………………………….

991 992 993 994 … 999

- This will create **81** nine-ary clauses

Row Constraints

- A row can't have any duplicate values. Each cell within a row will have a distinct value between 1 and 9
- $\bigwedge_{x=[1-9]} \bigwedge_{z=[1-9]} \bigwedge_{y=[1-8]} \bigwedge_{i=[(y+1)-9]} (-xyz \vee -xiz)$
- Eg. -235 ∨ -265
- This will create **2916** binary clauses (9 * 9 * 9C2)

Column Constraints

- A row can't have any duplicate values. Each cell within a row will have a distinct value between 1 and 9
- $\bigwedge_{y=[1-9]} \bigwedge_{z=[1-9]} \bigwedge_{x=[1-8]} \bigwedge_{i=[(x+1)-9]} (-xyz \lor -iyz)$
- Eg. -235 ∨ -435
- This will create **2916** binary clauses (9 * 9 * 9C2)

Box Constraint

- $\wedge_{z=[1-9]} \wedge_{i=[0-2]} \wedge_{j=[0-2]} \wedge_{x=[1-3]} \wedge_{y=[1-3]} \wedge_{k=[(y+1)-3]} (-(3i + x)(3j + y)z \vee -(3i + x)(3j + k)z)$
- $\wedge_{z=[1-9]} \wedge_{i=[0-2]} \wedge_{j=[0-2]} \wedge_{x=[1-3]} \wedge_{y=[1-3]} \wedge_{k=[(x+1)-3]} \wedge_{l=[1-3]}(-(3i + x)(3j + y)z \vee -(3i + k)(3j + l)z)$
- Eg. -111 -221 / -111 –321
- This will create **2916** binary clauses

Extra Clauses

- There is at most 1 number in each cell
- Each number appears at least once in each row
- Each number appears at least once in each column
- Each number appears at least once in each box
- These will create another **3078** clauses
- So overall we have **11907** clauses out of which **11664** are binary clauses and **243** are nine-ary clauses