BOOLEAN SATISFIABILITY CHECK: IMPLEMENT A BASIC SAT SOLVER

Runwei Zhou
Department of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47906, USA
`zhou1329@purdue.edu`

1 MOTIVATION

Boolean satisfiability is a classical NP-complete problem on whether a boolean formula is satisfiable. We've learned a number of algorithms and corresponding heuristics of an SAT solver for the boolean satisfiability problems in class. In this course project, we hope to gain hands-on experience on how to implement a **DPLL** solver with two heuristics as **Efficient Boolean Constraint Propagation (BCP)** and **Variable State Independent Decaying Sum (VSIDS)**.

2 ALGORITHMS AND DATA STRUCTURES

Basic algorithms used in this implementation is summarized:

Algorithm 1 DPLL with BCP and VSIDS

```
1:  procedure MYSAT($\psi$, $\nu$)
2:      if (UnitPropagation($\psi$, $\nu$)==CONFLICT) then                          ▷ Initialization
3:          return UNSAT
4:      end if
5:      $l \leftarrow 0$
6:      c = VSIDS_INIT($\psi$)          ▷ Initialize literal count list by number of which appears
7:      while All variables are allocated do                                        ▷ Iteration begins
8:          $x$ = PickBranchingVariable($\psi$, $\nu$)                          ▷ Chaff decision heuristics
9:          $\nu \leftarrow \nu \cup x$                                                          ▷ BCP in Chaff
10:         if (UnitPropagation($\psi$, $\nu$)==CONFLICT) then
11:             $\psi$, $\beta$=ConflictAnalysis($\psi$, $\nu$)                          ▷ Learning new clause
12:             c = VSIDS_UPDATE($\psi$, c)           ▷ Update count list by current contradictions
13:             $l \leftarrow l + 1$
14:             if ($\beta < 0$) then return UNSAT
15:             else Backtrack($\psi$, $\nu$, $\beta$)
16:                 $l \leftarrow \beta$
17:             end if
18:         end if
19:
20:     end while
21: end procedure
```

In which corresponding functions used are:

`UnitPropagation`(ψ, ν): Check whether there are unit clauses in the expression. Return conflict if there is a unite clause and its complement. Chaff's efficient boolean constraint propagation is implemented.

`PickBranchingVariable`(ψ, ν): Pick the next branching variable that has never been used. In DPLL, branching is done chronologically. However, Chaff's decision heuristic determines the next variable based on the count list introduced later.

`ConflictAnalysis`(ψ, ν): After detecting the conflict, update the cause clause to ψ.

`Backtrack`(ψ, ν, β): Backtrack decision level to β.

`VSIDS_INIT`(ψ): Each variable in each polarity has a counter, initialized to the number of the corresponding literal that appears.

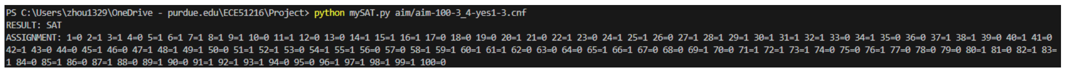
`VSIDS_UPDATE`(ψ, c): Update count list accordingly. A fixed decay rate of 5% is chosen.

In my implementation, clauses are stored as `list` in Python, whose values are marked with `int` and their sign. If an element is positive, it's `True`. If an element is negative, it becomes its complement.

Another important data structure we employed to implement the 2-literal watch BCP is the literal watch dictionary. `dictionary` stores literal name (treat literal and its complement as different ones) as key and clause index list as value. The watched clause is also recorded in the form of `list`. Assigned literals can be identified accordingly.

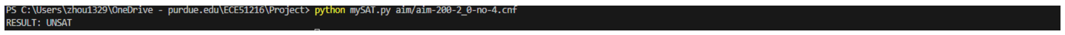
3 Results

We first verify our solver by testing existing benchmark cnf test cases. Fig. 1 and Fig. 2 shows two example outputs. Compared with results provided by teaching assistant in the email, our solver is valid for solving SAT problems



```
PS C:\Users\zhou1329\OneDrive - purdue.edu\ECE51216\Project> python mySAT.py aim/aim-100-3_4-yes1-3.cnf
RESULT: SAT
ASSIGNMENT: 1=0 2=1 3=1 4=0 5=1 6=1 7=1 8=1 9=1 10=0 11=1 12=0 13=0 14=1 15=1 16=1 17=0 18=0 19=0 20=1 21=0 22=1 23=0 24=1 25=1 26=0 27=1 28=1 29=1 30=1 31=1 32=1 33=0 34=1 35=0 36=0 37=1 38=1 39=0 40=1 41=0
42=1 43=0 44=0 45=1 46=0 47=1 48=1 49=1 50=0 51=1 52=1 53=0 54=1 55=1 56=0 57=0 58=1 59=1 60=1 61=1 62=0 63=0 64=0 65=1 66=1 67=0 68=0 69=1 70=0 71=1 72=1 73=1 74=0 75=0 76=1 77=0 78=0 79=0 80=1 81=0 82=1 83=
1 84=0 85=1 86=0 87=1 88=0 89=1 90=0 91=1 92=1 93=1 94=0 95=0 96=1 97=1 98=1 99=1 100=0
```

Figure 1: Output example with `aim-100-3_4-yes1-3.cnf`



```
PS C:\Users\zhou1329\OneDrive - purdue.edu\ECE51216\Project> python mySAT.py aim/aim-200-2_0-no-4.cnf
RESULT: UNSAT
```

Figure 2: Output example with `aim-200-2_0-no-4.cnf`

Then we continue testing the performance of our method and conventional DPLL solver. Test functions are open-source benchmarks that are provided in the class. There are 80 clauses with 50 literals in the first 8 files while there are 100 clauses in the last 2. `DPLL` and `MYSAT` are tested. The `DPLL` solver is implemented by turning off heuristic features in `MYSAT`. We are using a 64-bit Windows desktop with a 13th Gen Intel(R) Core(TM) i7-13700 2.10 GHz processer and 42.0 GB RAM to perform the tests. Each benchmark is evaluated more than three times and the median is chosen. Results is summarized in the Table 1. Compared with basic DPLL, heuristics significantly improved

Table 1: CPU time of DPLL and implemented method

	aim-50-1_6-no-1	**aim-50-1_6-no-2**	**aim-50-1_6-no-3**	**aim-50-1_6-no-4**	**aim-50-1_6-yes1-1**	**aim-50-1_6-yes1-2**	**aim-50-1_6-yes1-3**	**aim-50-1_6-yes1-4**	**aim-50-2_0-no-1**	**aim-50-2_0-no-2**
DPLL	1.984375	1.40625	1.203125	1.3125	0.546875	0.875	0.125	0.484375	1.34375	0.90625
MYSAT	0.03125	0.296875	6.125	0.015625	0.015625	0	0	0	0.015625	0.046875

the efficiency of solving SAT problems. SAT solver with Chaff's heuristics surpasses conventional DPLL in 9 out of 10 test cases. `aim-50-1_6-no-3.cnf` is an exceptional point, which implies that heuristic methods are not guaranteed to have better performance. It's also obvious that unsatisfiable problems usually take more time to be evaluated. Since BCP procedure is the major cost in either DPLL or Chaff's, the computation time is assumed to be proportional to the problem size while it is also affected by problem structures. That's one of the reasons that VSIDS succeeds.

4 Conclusions

In this project, we successfully implemented the basic DPLL algorithm and Chaff's heuristics. We notice that it is necessary to develop fast SAT algorithms for such an NP-complete problem. Chaff's heuristics are indeed helpful in improving the general performance of the conventional DPLL solver.