中国科学技术大学计算机学院

《形式化方法导引》作业

2022.06.05

作业题目：实验大作业

学生姓名：黄科鑫

学生学号：PB19061283

计算机实验教学中心制

2019 年 9 月

1 Introduction - SAT solver

SAT Problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula, and a **SAT Solver** is the program to solve SAT Problems. To simplify the problem, here we focus on SAT problems formatted by CNF, which are boolean formulas like: $(X_1 \vee \neg X_2 \vee X_3) \wedge (...)$.

More specifically, the solver requires input file in DIMACS-CNF format:

- Text file "*.cnf", each line represents a single disjunction;
- Variables are represented by positive numbers, '0' means end of a line, and '-' means $\neg$;
- Line starts with 'c' is comment line;
- Line starts with 'p' is problem line, the format is: "p cnf $variable_num$ $clause_num$".

For example, the cnf file with following contents means $(x_1 \vee \neg x_5 \vee x_4) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee x_5 \vee x_3 \vee x_4)$:

```
c This is comment
p cnf 5 3
1 -5 4 0
1 2 0
-1 5 3 4 0
```

2 Algorithm

2.1 DPLL

DPLL is a classical algorithm for SAT problems, and the most important idea in DPLL is "Unit Resolution":

for clauses X and a unit clause l:

1 remove $\neg l$ from all clauses in X containing $\neg l$;

2 remove all clauses containing l.

After performing unit-resolution, clauses will be simplified or removed, so if we can continually do unit-resolution for the problem, we can finally reduce complex CNF input to simple conjunctive of unit clauses, then it will be very simple to solve the problem. And DPLL just does that:

```
function DPLL(Φ)
while there is a unit clause {l} in Φ do
Φ ← unit-propagate(l, Φ);
while there is a literal l that occurs pure in Φ do
```

```
Φ ← pure−literal−assign(l, Φ);
if Φ is empty then
return true;
if Φ contains an empty clause then
return false;
l ← choose−literal(Φ);
return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l);
```

The DPLL algorithm is very simple to implement, but it have some drawbacks: every time the function DPLL(X) is called, it must make a copy of current CNF problem, and it doesn't utilize any information during search procedure.

2.2 CDCL

As I mentioned above, DPLL has some drawbacks, but those disadvantages can be solved using CDCL algorithm.

CDCL use a set of unit clauses M representing information learnt during its search procedure, and M is updated by following rules:

1 **UnitPropagate**: $M \Rightarrow Ml$, if l is undefined, $M \models \neg C$, and $C \vee l$ is a clause;

2 **Decide**: $M \Rightarrow Ml^d$, if l is undefined in M (This is used when no UnitPropagate is possible);

3 **BackTrack**: $Ml^dN \Rightarrow M\neg l$ if Ml^dN introduce conflicts, where N contains no decision literals.

4 **Result**: if M contains all Variables and no conflicts, SAT; if M implies conflicts with no l^d in M, unSAT.

The basic CDCL algorithm can be derived by rule above:

```
loop
        unitpropagate()                 −propogate unit clauses
        if not conflict then
                if all variables assigned then
                        reutrn SAT
                else
                        decide()        − pick a variable to assign
        else
                if no variable is decided then
                        return unSAT
                else
                        backtrack()     − undo assignments
```

Though the basic CDCL algorithm is better than DPLL with those two drawbacks solved, we can still make some improvements on it:

1 **BackJump Rather than BackTrack**: according to the BackTrack rule, when conflict occurs, M just go back to previous decision, but sometimes the previous decision is not the reason why conflict occurs, conflicts often happen after several decisions made, however. So we should analyse after conflict occurs, and back jump to the decision which result in conflict rather than just go back to the most recent decision.

2 **Heuristic Decision**: determining which literal to assign can have great impact on the search space, like any other methods in traditional search problems, a good heuristic function here can make program more efficiently with less effort.

3 **Restart Mechanism**: when the problem becoming bigger, with thousands of variables to solve, the search space is so big that if the program goes several steps wrong, it may have to waste unacceptable time searching until it back-jumps. Like traditional search problem, If algorithm gets stuck in a local optimum, we can simply restart, and the method also applies here. When too many conflicts occur while solver still cannot find result, just "restart" to avoid unlimited search.

3 Implementation

3.1 DPLL

Considering DPLL as a baseline algorithm, I used Python to quickly implement it to test its performance. And the result is very poor: Python is known to have bad performance, and DPLL cannot simply search procedure, so the program can ONLY solve some examples presented in our lesson. If input size grows a little larger, the program will crack down because of too many recursion.

```
PS D:\SAT\src\pydemo> py .\dpll.py
input file path:1.cnf
  File ".\dpll.py", line 106, in <module>
    status,units = DPLL(unset_ids, clauses)
  File ".\dpll.py", line 65, in DPLL
    status,new_units = DPLL(next_ids, clauses+[Clause({1: True})])
  File ".\dpll.py", line 65, in DPLL
    status,new_units = DPLL(next_ids, clauses+[Clause({1: True})])
  File ".\dpll.py", line 65, in DPLL
    status,new_units = DPLL(next_ids, clauses+[Clause({1: True})])
  [Previous line repeated 992 more times]
  File ".\dpll.py", line 64, in DPLL
    c_copy = [C.copy() for C in clauses]
  File ".\dpll.py", line 64, in <listcomp>
    c_copy = [C.copy() for C in clauses]
  File ".\dpll.py", line 24, in copy
    return Clause(self.clause)
RecursionError: maximum recursion depth exceeded
PS D:\SAT\src\pydemo> py .\dpll.py
input file path:2.cnf
Time elapsed: 0.00015129999999996535
SAT {1: True, 2: False, 3: False, 4: True, 5: True}
PS D:\SAT\src\pydemo>
```

3.2 CDCL

3.2.1 miniSAT

After having failed in Python+DPLL, I decided to modify open source SAT solvers to implement CDCL. And I chose miniSAT. The miniSAT solver v1.13, as described in their doc, has surpassed contemporary public solvers in SAT competition 2005, and the version I use is v1.14.

The algorithm miniSAT v1.14 is a variant of CDCL, with following functions implemented:

1 **BackJump**: when assigning a literal, the solver will record the reason why it is set(aka. those clauses can imply value of the literal). And when the solver finds conflicts, it will go through the assignments stack and analyse their reason clauses, to make correct assignments for literal and add clauses learnt.

2 **Heuristic Decision**: each variable has an activity attached to it. Every time a variable occurs in a recorded conflict clause, its activity is increased. After recording the conflict, the activity of all the variables in the system are multiplied by a constant less than 1, thus decaying the activity of variables over time. Recent increments count more than old. The current sum determines the activity of a variable. This mechanism is called VSIDS(Variable State Independent Decaying Sum).

3 **Dynamic Restart**: when number of conflicts exceeds a threshold, the solver will restart, and set the threshold with a higher value.

4 **Other Mechanisms**: **SimplifyDB**: simplify the input clauses, removing redundant ones and tautologies; **ReduceDB**: reduce some learnt clauses when they are of too much amount; **Conflict Clause Minimization**: simplify learnt clauses.

Although miniSAT has so many optimizations, the heuristic it uses to make decisions is relatively old(proposed in 2001). So I tried to replace the old VSIDS heuristic with some modern techniques.

3.2.2 CHB Heuristic

Liang, Jia, et al. presented a new heuristic named CHB in 2016 [1], which is shown to outperform VSIDS heuristic. **So I embedded their algorithm into miniSAT code to implement a SAT solver using CHB heuristic.**

CHB Heuristic:

[1] "Exponential recency weighted average branching heuristic for SAT solvers." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 30. No. 1. 2016.

Algorithm 1 a simple CDCL solver with CHB as the branching heuristic.

$\alpha \leftarrow 0.4$
$numConflicts \leftarrow 0$
$plays \leftarrow \emptyset$
for $v \in Vars$ **do**
    $lastConflict[v] \leftarrow 0$
    $Q[v] \leftarrow 0$
end for
loop
    Boolean constraint propagation
    $plays \leftarrow plays \cup \{$variables propagated just now$\}$
    if a clause is in conflict **then**
        $multiplier \leftarrow 1.0$
    else
        $multiplier \leftarrow 0.9$
    end if
    for $v \in plays$ **do**
        $reward \leftarrow \frac{multiplier}{numConflicts - lastConflict[v] + 1}$
        $Q[v] \leftarrow (1 - \alpha) \times Q[v] + \alpha \times reward$
    end for
    if a clause is in conflict **then**
        $numConflicts \leftarrow numConflicts + 1$
        if $decisionLevel == 0$ **then**
            return UNSAT
        end if
        if $\alpha > 0.06$ **then**
            $\alpha \leftarrow \alpha - 10^{-6}$
        end if
        conflict analysis and learn a new clause
        $c \leftarrow \{$variables in conflict analysis$\}$
        $u \leftarrow$ the first UIP of the learnt clause
        non-chron. backtrack based on conflict analysis
        assert variable u based on new learnt clause
        for $v \in c$ **do**
            $lastConflict[v] \leftarrow numConflicts$
        end for
        $plays \leftarrow \{u\}$
    else
        if no more unassigned variables **then**
            return SAT
        end if
        $unassigned \leftarrow \{$unassigned variables$\}$
        $v^* \leftarrow argmax_{v \in unassigned} Q[v]$
        assign v^* to true or false based on polarity heuristic such as phase saving
        $plays \leftarrow \{v^*\}$
    end if
end loop

4 Experiments

Here I use SATLIB Uniform Random-3-SAT to test solvers' performance, every input file contains 200 variables and 860 clauses. Aside from default miniSAT and miniSAT with CHB, I modified miniSAT's arguments to let it randomly choose variable to assign, so that we can see the improvement made by heuristic decision. The result are as follow:

Test id	default miniSAT	CHB miniSAT	randomly decided solver	Z3
SAT1	0.01562	0.04687	22.4375	0.05068
SAT2	0.06250	0.20312	14.0938	0.15949
SAT3	0.01563	0.04688	2.57812	0.06238
SAT4	0.09375	0.03125	5.90625	0.22485
SAT5	0.15625	0.04687	6.25	0.25112
unSAT1	0.10937	0.48437	27.0625	0.18970
unSAT2	0.15625	0.57812	22.0496	0.28062
unSAT3	0.06250	0.23437	35.0781	0.20252
unSAT4	0.42187	1.03125	96.9375	1.51080
unSAT5	0.17187	0.59375	30.7031	0.32429
SAT AVG	**0.06875**	**0.07499**	**10.25313**	**0.14970**
unSAT AVG	**0.18437**	**0.58437**	**42.36616**	**0.50159**

As is shown above, both Z3 and miniSAT can greatly outperforms randomly decided SAT solver. And default miniSAT is several times faster than Z3, while miniSAT with CHB, which I modified, is a little bit slower. **But it's still faster than Z3 when solving problems that satisfy**.

Actually, when implementing CHB, I haven't got too much time to optimize my code. If I implement the $Q[v]$ in CHB algorithm as a Max Heap, it should be able to solve more quickly. My code has been submitted to GitHub, and I will try to optimize the CHB miniSAT's code after my exams.

5 Conclusion

During this homework, I implemented DPLL algorithm with Python, explored several optimization of CDCL algorithm, and implemented CHB heuristic for CDCL by modifying miniSAT's code. After that, I designed experiments to compare the performance of miniSAT, CHB miniSAT and Z3 solver.

6 Appendix

I will submit my code together with this report, and the structure of my code file is:

```
code/
├── src/
│   ├── chbsat/ -- miniSAT with CHB
│   ├── minisat/ -- miniSAT v1.14
│   ├── pydemo/ -- DPLL Python
│   ├── readme.md -- file explanation
│   └── z3sat.py -- z3 solver
└── test/
    ├── sat-200-860/ -- satisfiable tests
    └── unsat-200-860/ -- not satisfiable tests
```