A2

In this assignment you will implement a solver for the Satisfiability (SAT) problem. In particular, you will implement the Davis–Putnam–Logemann–Loveland (DPLL) algorithm (R&N 3ed Section 7.6, Figure 7.17). This algorithm takes as input a logical statement in CNF format and outputs a assignment of True or False to each propositional symbol.

We have provided a template for you to start with. You can download all the code as a zip here *(https://coursys.sfu.ca/2019fa-cmpt-310-d1/pages/A2.zip/view)* [Updated Oct 22th, 2019].

File you'll edit:

```
DPLLsat.py         Where your DPLL algorithm will reside.
```

Files you might want to look at:

```
All files end with.cnf
 Each file describes a CNF formula in DIMACS CNF file format.
 The file name indicates whether the formula is satisfiable (*sat.cnf) or not (*unsat.cnf).
```

You can ignore all other supporting files.

What to submit: You should submit a single zip file that contains the following:

- DPLLsat.py
- (Optional) the contest folder

Evaluation: Your code will be *autograded* for technical correctness and efficiency. Please do not change the names of any provided functions or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

We have included an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

Note: There is a 1-minute time limit for every test. You should be able to solve each test in seconds if you implement the algorithm correctly.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Part 0 DIMACS CNF file format

This format is used to define a Boolean expression, written in conjunctive normal form (CNF), that may be used as an example of the satisfiability problem.

The satisfiability problem considers the case in which N boolean variables are used to form a Boolean expression involving negation (NOT), conjunction (AND) and disjunction (OR). The problem is to determine whether there is any assignment of values to the Boolean variables which makes the formula true. It's something like trying to flip a bunch of switches to find the setting that makes a light bulb turn on.

For simplicity, it is common to require that the boolean expression be written in conjunction normal form or "CNF". A formula in conjunctive normal form consists:

```
* clauses joined by AND;
* each clause, in turn, consists of literals joined by OR;
* each literal is either the name of a variable (a positive literal,
  or the name of a variable preceded by NOT (a negative literal.)
```

An example of a boolean expression in 3 variables and 2 clauses:

```
( x1 OR ( NOT x3 ) )
AND
( x2 OR x3 OR ( NOT x1 ) ).
```

The CNF file format is an ASCII file format.

The file may begin with comment lines. The first character of each comment line must be a lower case letter "c". Comment lines typically occur in one section at the beginning of the file but are allowed to appear throughout the file.

The comment lines are followed by the "problem" line. This begins with a lower case "p" followed by a space, followed by the problem type, which for CNF files is "cnf", followed by the number of variables followed by the number of clauses.

The remainder of the file contains lines defining the clauses, one by one.

A clause is defined by listing the index of each positive literal, and the negative index of each negative literal. Indices are 1-based, and for obvious reasons, the index 0 is not allowed.

The definition of a clause may extend beyond a single line of text.

The definition of a clause is terminated by a final value of 0.

The file terminates after the last clause is defined.

Some of the examples of DIMACS CNF file can be found from SATLIB at UBC
https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html *(https://www.cs.ubc.ca/%7Ehoos/SATLIB/benchm.html)*

Example:

Here is the CNF file that corresponds to the simple formula discussed above:

```
c  simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

Part 1 DPLL SAT Solver (10 points)

As a refresher—the motivation for an SAT solver is to determine if, for a given boolean formula, there exists an assignment of true and false to the variables such that the entire formula evaluates to true. Algorithms that solve the boolean satisfiability problem are employed by formal verification tools under-the-hood to determine the satisfiability of higher-level constraint models. In general, efficient SAT-solving is an extremely important and widely-used problem in computer science.

DPLL expects as input a formula in conjunctive normal form, i.e., a set of clauses that are joined together with “and”. This means that to satisfy the input clause set, every clause must be satisfied. We call a clause with only one literal a “unit clause” and a clause with no literals the “empty clause”, which is the same as falsehood. A literal is “pure” if, whenever it appears in the clause set, it always has the same sign. A set of unit clauses is consistent if no two have different signs for the same variable.

Input Specification

Your program should take in a formula in DIMACS CNF file format, where a formula is a list of clauses separated by 0s and each clause is a list of literals separated by spaces. Literals are non-zero numbers where a positive number denotes that the variable must be true and a negative number denotes it must be false. There is an implicit “or” between each literal and an implicit “and” between clauses.

In DPLLsat.py, we've provided the template for one possible implementation of reading input and internal clauses representation in python. Feel free to use it directly or implement your own.

OUTPUT specification:

Your program should print “UNSAT” if a formula is unsatisfiable and "SAT" if it is satisfiable.

Additionally, if verbosity is set to True, and the formula is satisfiable, the program should also print the solution in the form:

```
SAT
list of true literals
```

where you print a list of literals that are true in the *ascending* order.

To run the program, use the following command:

```
python DPLLsat.py -i <inputCNFfile>
```

or

```
python DPLLsat.py -i <inputCNFfile> -v
```

if you want to see the solution.

** Special Note:**

For this assignment, you are allowed to import new libraries as long as:

1. It's available on CSIL machines;
2. It's used to support your implementation and it's not part of any tasks specific routine for this assignment. Ask on piazza if you are unsure about the library you want to use. In general, you can use the python standard library and numpy.

You can define new classes and even modify the template but please make sure you are following the input/output specification.

Q1.1 Recursive Backtracking (4 points)

Implement the Backtracking portion of the DPLL algorithm in DPLLsat.py.

Recursive Backtracking forms the bases of DPLL. It picks some variable to branch on. The satisfiability problem is then split into two sub-problems: whether the formula is satisfiable with the chosen variable assigned as either true or false. Essentially, the algorithm “guesses” a variable to be true, recursively determines if that subproblem is satisfiable; if it is not, the algorithm then “guesses” the variable to be false and tries again.

You’re free to choose any reasonable method of picking variables to branch on. You might randomly select a variable, pick the variable that occurs the most in the formula, or even just take the alphabetically-next variable. By “reasonable” we mean a method that leads to correct results.

Your code should quickly find a solution for:

```
python DPLLsat.py -i test_sat.cnf -v
```

Correct Output:

```
SAT
[2, 3]
```

However, you will see that it takes about 1 minute to solve a sudoku instance:

```
python DPLLsat.py -i sudoku6sat.cnf
```

Correct Output:

```
SAT
```

Hint: Please make sure your recursive backtracking is implemented correctly before doing the next step. Unit Propagation and Pure Literal Elimination only make the SAT solver more efficient. They do not affect the correctness of the solver.

Q1.2 Unit Propagation (3 points)

Add unit propagation to your solver in DPLLsat.py.

The real power of the DPLL algorithm lies in its formula simplification techniques.

Unit propagation is one aspect of this formula simplification. If a clause is a unit clause (one with only one variable), then the only way for the formula to be satisfiable is for that variable to be assigned consistently with its sign in that clause. Thus, we know that variable must take on that assignment, which tells us that all other clauses with that literal are now satisfied. In addition, we know that clauses with the literal’s negation will not be satisfied by that component, so we can remove the negation from all clauses.

Test your SAT solver again on the same sudoku instance:

```
python DPLLsat.py -i sudoku6sat.cnf
```

You should be able it solve it in a few seconds.

Q1.3 Pure Literal Elimination (3 points)

Add pure literal elimination to your solver in DPLLsat.py.

The second aspect of formula simplification is eliminating pure literals (ones that occur with only one polarity throughout the formula). If a literal is pure, then each clause it appears in can be satisfied by assigning the variable consistent to the sign it appears with. Thus, without changing the satisfiability of the formula, we can add the pure literal as a unit clause and assign the T/F value to the literal which makes it evaluated to true.

Test your SAT solver with

```
python DPLLsat.py -i 1000sat.cnf
```

Our implementation solved this instance within 1 second.

Part 2 (Optional) SAT solving contest (bonus 2 points)

This contest involves solving as many random hard instances as you can in 5 minutes.

Getting Started

1. Copy your improved solver to the contest folder
2. Change to the working directory to the contest folder
3. Compile the SAT instance generator by:

```
gcc -o sgen sgen.c -lm
```

4. Run the contest script by:

```
python contest.py
```

- You can test your solver with individual instances by this command:

```
./sgen -n num-of-variables -sat -s 310 > out.cnf
python DPLLsat.py -i out.cnf
```

All test instances are satisfiable.

What to submit

Put your modified sat solver in the contest folder.

Also, include a result.txt that contains the output of the contest script. You will also need to provide a brief description of what you did and include relevant references.

Restrictions

You are free to modify the DPLL solver. It's ok to implement advanced DPLL algorithms you found from internet/academic papers, but you need to provide the reference to them and write every line of the code by yourself!

Multi-threading and CUDA are allowed.

WARNING: SAT solver optimization is a very time-consuming job. Please do so only if you have the interest and time!

Academic Dishonesty

While we won't validate your result, we still expect you not to falsely represent your work. Please don't let us down.

Part 3 Feedback

Give one short piece of feedback about the course so far. What have you found most interesting? Is there a topic that you had trouble understanding? Are there any changes that could improve the value of the course to you?

How many hours did you spend on this assignment?

Please provide your answers in DPLLsat.py.

*Instruction for DPLL SAT solver is adopted from the Logic for Systems course offered at Brown University.

Updated Mon Dec. 02 2019, 15:29 by liuhengl.