Sudoku and Propositional logic

Ping-Jung Liu October 25th 2017

1. Introduction and Background

In this project, I implemented a general solver with two boolean satisfiability algorithms, gsat and walksat, to solve the classic Sudoku and N-queens problems. The solver will read in cnf file and write the assignments of variariables to a separate solution file. A cnf file consists of lines of clauses; a clause contains a logic that needs to be satisfied. In each clause, space " " represents OR, and each clause are connected by invicible AND. For example:

```
-111 -112
-111 -113
```

This represents (not 111 OR not 112) AND (not 111 OR not 113). The solver will find a variables assignment that satisfiesd the whole cnf logics with gsat and xsat, which will be further discussed.

2-1. SAT Solver--GSAT

In a boolean satisfiability problem, the goal is to assign each variable a value so that the whole assignments satisfy the given cnf logics. In my implementation, the variables are stored in a list X. They are positive integers to make it easier for testing and flipping between True and False. I used a dictionary to store the keys, variables, and values, 1 or -1. And the provided cnf logics are stored in a list of integer lists. For example:

```
-111 -112
-111 -113
```

This can be represented with [[-111, -112], [-111, -113]].

The steps of GSAT as follow:
1. Randomly assign each variable a value; in my case all -1
2. If the assignment satisfies all logics in cnf, stop.
3. Pick a random number between 0 and 1. If the number is greater than a theshold h, randomly pick a variable and flip its value.
4. If the number is smaller than h, for each variable, calculate the total number of clauses would be satisfied if it were flipped.
5. If several variables have high numbers, randomly pick one to flip, then go back to step 2.

GSAT code as follow:

```
while not self.is_solution():

    counter = counter + 1

    # if a random int between 0 to 1 is greater than h
    # randomly flip a variable
    if random.randint(1, 100)/100 > self.h:

        ind = random.randint(1, len(self.X))
        self.sol[ind] = -1 * self.sol[ind]

    # if not, perform gsat
    else:

        # If key var is flipped, value clauses will be satisfied
        var_match = {}
        max_count = 0
        # list of variable to choose from
        pool = []

        for var in self.sol:
            # get the number of clause satisfied if var is flipped
            count = self.get_match(var)
            var_match[var] = count
            # update highest number of clause satisfied
```

```
            if count > max_count:
                max_count = count

        #print(max_count)
        # get the pool of variable to choose from
        for var in var_match:
            if var_match[var] == max_count:
                pool.append(var)

        varr = random.choice(pool)

        self.sol[varr] = -1 * self.sol[varr]

        #if counter == 5000:
        #   break

return True
```

This is an intuitive algorithm with a rather long runtime, which is also why the code should be easy to follow. I will not show the helper functions here but the comments should be clear enough.

This GSAT implementation was able to solve one*cell.cnf, all*cells.cnf, rows.cnf, rows*and*cols.cnf in less than five minutes.

2-2. SAT Solver--WALKSAT

Walksat is essentially a modification of GSAT to boost performance. When scoring in GSAT, the program needs to loop more than a million times, in the sudoku case, just to flip one variable, and this yields terrible performance. In Walksat, the program will randomly choose one of the unsatisfied clauses with the function candidate_clause:

```
# get a list of all unsatisfied clauses
def candidate_clause(self):

    cand = []

    for clause in self.clause:
        flag = False
        for atom in clause:
            varr = abs(atom)
            if varr * self.sol[varr] == atom:
                flag = True
                break
        if not flag:
            cand.append(clause)

    return cand
```

And then use the same scoring system to score the variables involved in this clause. I designed so that every 2000 iterations, it will take the GSAT scoring system in hopes of escaping local minimum.

Walksat codes as follow:

```
while not self.is_solution():

    counter = counter + 1

    # if a random int between 0 to 1 is greater than h
    # randomly flip a variable
    if random.randint(1, 100)/100 > self.h:

        ind = random.randint(1, len(self.X))

    !!! if not [ind * self.sol[ind]] in self.clause:
            self.sol[ind] = -1 * self.sol[ind]

    # if not, perform walksat
```

```
    else:

        # same as gsat
        var_match = {}
        max_count = 0
        pool = []

        # randomly choose a unsatisfied clause
        cand = self.candidate_clause()
        clauses = random.choice(cand)
        # list of variables in this clause
        rand_clause = []
        # get the variables in this clause
        for i in range(0, len(clauses)):
            rand_clause.append(abs(clauses[i]))

        # every 2000 iterations, perform gsat in hopes of escaping local minimum
        if counter % 2000 == 0:
            var_candidate = list(self.sol)
        else:
            var_candidate = rand_clause

        for var in var_candidate:
            # get the number of clause satisfied if var is flipped
            count = self.get_match(var)
            var_match[var] = count
            # update highest number of clause satisfied
            if count > max_count:
                max_count = count

        print(max_count)
        # get the pool of variable to choose from
        for var in var_match:
            if var_match[var] == max_count:
                pool.append(var)

        varr = random.choice(pool)

    !!! if not [varr * self.sol[varr]] in self.clause:
            self.sol[varr] = -1 * self.sol[varr]

        #if counter == 5000:
        #   break

return True
```

Note there were two lines with !!! at the front. Those are additional features that will be further explained in later sections.

With Walksat, the program could solve rules.cnf and all the easier cnf in less than 5 minutes. The program took less than an hour to solve both puzzle*1.cnf and puzzle*2.cnf. All puzzle solutions are included in the zipfile in .sol format.

I figure there is not much point to just post some of the solved puzzles here because they would not demonstrate anything. (I might have manually typed in the solution?!)

3. N-Queens (Extra Credits)

First of all, this was a fun but complicated model to design precisely, so I think, just maybe, this extra feature can worth more than one point? :p

I implemented a N-Queens model that can take in a number of queens Q and solve a N-Queens problem. Queens can move horizontally, vertically, and diagnolly. The objective of the problem is to place n queens on a n * n board so that none of them can attack each other, and the generated cnf closely follow these restrictions. The program can solve for all N greater than 3. It took less than a minute to solve 10-Queens and less than five minutes to solve 15-Queens. Below I will show some of the resulting N-Queens the program solved, and of course I know I could be manually writing the answers, but I just found them intersting so I felt like showing them here.

```
10-Queens
```

```
488 iterations
Q # # # # # # # # #
# # # # # # # # Q #
# # # # # # Q # # #
# # Q # # # # # # #
# # # # # # # Q # #
# Q # # # # # # # #
# # # # Q # # # # #
# # # # # # # # # Q
# # # # # Q # # # #
# # # Q # # # # # #


15-Queens
273iterations
# # # # # # # # # Q # # # # #
# # # Q # # # # # # # # # # #
# # # # # # # # # # # # Q # #
# # # # # # # # # # Q # # # #
# # # # # # # # Q # # # # # #
# # # # # # Q # # # # # # # #
# # # # # # # # # # # # # # Q
Q # # # # # # # # # # # # # #
# # # # # # # # # # # # # Q #
# Q # # # # # # # # # # # # #
# # # # # Q # # # # # # # # #
# # # # # # # Q # # # # # # #
# # # # # # # # # # # Q # # #
# # # # Q # # # # # # # # # #
# # Q # # # # # # # # # # # #
```

4. Known Variables (Extra Credits)

Obviously, a clause not necessarily has to be several variables OR together. There are cases when a clause contains one single variable; in this situation, we can consider the variable a known one, because there is only one way to satisfy a clause with one variable. But in our Walksat implementation, it neglects the fact that some variables are known already and flip them still.

The good thing about this is, ignoring known variables can occasionally allow program to reach another variable space potentially with solutions. Like in a maze situation when you have to go around some barriers to reach the goal.

The obvious disadvantage is that the program often times flip an already known variable and cause extra runtimes.

Though there are pros and cons for both approaches, I think it is better to not flip a known variable at anytime. Below is the code of this modification:

```
if not [varr * self.sol[varr]] in self.clause:
    self.sol[varr] = -1 * self.sol[varr]
```

Short indeed, it basically says only flip a variable if there is no clause with only the variable in it. While the original Walksat could not solve puzzle_bonus, at least after several hours of testing, the new version luckily solved puzzle bonus in less than 30 minutes (I was like WOW).