Sudoku Propositional Logic Report

Paolo Takagi-Atilano

Introduction

Boonlean satisfiability (SAT) problems are ones in which many propositional logic clauses are put into place that serve as constraints for a specific problem. In order to solve these, the algorithm must reach a point where every single constraint has been satisfied. These algorithms use random walks rather than backtracking, in order to find solution variable assignements in which all the constraints have been satisfied. This report will look into the efficiency of these algorithms compared to each other, and what sort of optimizations can be made to make them faster or better in diferent ways.

Design

I wrote the `SAT.py` file, which contained all of the code for solving general CNF problems. It contains the following data/functions:

Data:

`clauses` - set of propositional logic clauses (in string format)

`assignment` - variable assignments, a list of boolean values

`var_to_index` - dictionary mapping clause variables (strings) to corresponding assignment indices in `assignment` list

`index_to_var` - dictioanry mapping `assignment` list indices to corresponding clause variables (strings)

`threshold` - float value which is the probability of scoring rather than choosing random assignemnt/candidate variables

`max_flips` - integer which is the number of maximum flips that walksat wil try before giving up

Functions:

`setup(self, cnf_filename)` - sets all the data up, from given filename

`gsat(self)` - gsat search algorithm

`score_assignment(self)` - returns set of assignment variables with highest # of satisfied clauses after being flipped (used in gsat)

`walksat(self)` - walksat saerch algorithm

`score_candidates(self)` - returns set of walksat candidate variables with highest # of clauses satisfied after being flipped (used in walksat)

`satisfied_clauses(self, candidate)` - returns number of satisfied clauses of the current assignment, as well as updates the set of unsatisfied clauses for when `candidate` is specified to `True`

`random_assignment(self)` - sets current assignemnt to one of completely (pseudo)random `True` and `False` values

`write_solution(self, sol, filename)` - writes the solution to a .sol file with given filename for use by other modules to output solution with some syntax

GSAT

Description:

First, I implemented the GSAT algorithm. This works as follows: First it sets the assignment list to one of (pseudo)random assignments of True and False values. Then, while all clauses have not been satisfied, it generates a (pseudo)random float between 0 and 1. If that number is greater than the predefined `threshold`, then it will flip a random variable in the assignment. Otherwise, it will score the assignment, and will flip a random variable that will result in the greatest number of possible satisfied clauses in that following variable assignment.

Code (simplified - no print statements or iteration tracking):

```
def gsat(self):
    # setup
    self.random_assignment()
    temp = self.satisfied_clauses(False)

    # while solution not found
    while not temp == len(self.clauses):

        # decided not to score the assignment list
        if random.random() > self.threshold:
            i = random.randint(0, len(self.assignment) - 1)
            self.assignment[i] = not self.assignment[i]
        # score the assignment list
        else:
            i = random.choice(list(self.score_assignment()))
            self.assignment[i] = not self.assignment[i]

        temp = self.satisfied_clauses(False)

    return True
```

Results:

One Cell (threshold = 0.7):

```
GSAT solved after 4 iterations
--- 0.0037900000000000017 seconds ---
```

All Cells (threshold = 0.7):

```
GSAT solved after 477 iterations
--- 1749.612294 seconds ---
```

Conclusions:

GSAT is clearly quite slow. Many of the iteration steps are very slow which is because each scoring step must iterate through every variable. If there are 729 variables and a few thousand clauses, then each scoring iteration results in more than a million loops, which is quite costly. Furthermore, selecting a variable completely at random means that often, a variable in a potentially correct state will be selected, which puts the GSAT algorithm further away from reaching a solution.

WalkSAT

Description:

I then implemented the WalkSAT search algorithm. This works as follows: First it sets the assignment list to one of (pseudo)random assignments of True and False values. Then, it iterates until the max_flips value is reached. If it has reached a state where all clauses are satisfied, it will return true. Otherwise, it will pick a random float in between 0 and 1. If it is grether than the `threshold`, then it will select a random unsatisfied clause, and then select a random variable from within that to flip. Otherwise, it will (re)score the candidates, and select a (pseudo)random variable from a (pseudo)randomly selected clause within the new list of unsatisfied clauses.

Code (simplified - no print statements or iteration tracking):

```
def walksat(self):
    # setup
    self.random_assignment()
    temp = self.satisfied_clauses(True)

    # iterate until max flips value is reached
    for i in range(self.max_flips):

        # check to see if solution is found
        if temp == len(self.clauses):
            return True

        # decided not to score the candidate list
        if random.random() > self.threshold:
            i = random.choice(random.choice(list(self.walksat_candidates)).replace('-
', '').split())
            self.assignment[self.var_to_index[i]] = not self.assignment[self.var_to_i
ndex[i]]
        # decided to score the candidate list
        else:
            i = random.choice(list(self.score_candidates()))
            self.assignment[self.var_to_index[i]] = not self.assignment[self.var_to_i
ndex[i]]

        # test to see if assignment is satisfied for next iteration, also find new se
t of candidates
        temp = self.satisfied_clauses(True)

    return False
```

Results:

One Cell (threshold = 0.7):

```
WALKSAT solved after 5 iterations
--- 0.001010999999999998 seconds ---
```

All Cells (threshold = 0.7):

```
WALKSAT solved after 282 iterations
--- 4.536429999999999 seconds ---
```

Rows (threshold = 0.7):

```
WALKSAT solved after 406 iterations
--- 9.627497 seconds ---
```

Rows and Columns (threshold = 0.7):

```
WALKSAT solved after 2906 iterations
--- 111.592767 seconds ---
```

Puzzle 1 (threshold = 0.7):

```
WALKSAT solved after 21188 iterations
--- 859.2022900000001 seconds ---
```

Puzzle 2 (threshold = 0.7)

```
WALKSAT solved after 25332 iterations
--- 946.7058509999999 seconds ---
```

Conclusions:

WalkSAT is clearly a lot faster than GSAT. It managed to solve all cells in 4.5 seconds, which is much faster than the time it took GSAT to do it in. Furthermore, it also did so in much fewer iterations. This is probably because it selects variables from "candidates", rather than just a random vraiable. In order for a variable to be a candidate, it must be a member of a unsatisfied clause. This means that the algorithm is considering far fewer variables than GSAT at any point in time, as well as selecting variables which are more likely to lead to a final solution, as they will more likely satisfy more clauses. Clearly this is a far superior algorithm.

Running Tests:

Type this into the terminal (when you are in the correct directory):

`python3 solve_sudoku.py x.cnf`

Where `x` is name of the puzzle that you want to solve. Also in `solve_sudoku.py` you will see that lines 18-21 will have all but one line commented out. These correspond to whichever saerch algorithm/variation you would like to run. In order to change these, just change whichever one is not commented out before you run the program.

Resetting WalkSAT (Bonus)

Description:

For my first extention, I tried to make a version of WalkSAT that would reset to a completely new random variable assignement after a certain number of iterations. This was relatively simple to implement, I just ran my WalkSat method a certain number of times, and return `True` if it returned `True`. If WalkSAT returned `False` for each of those times, so did this version of the algorithm. This was to make sure that WalkSAT did not return failure just beacause it chose the wrong variables to flip or had an unlucky initial assignment. The main point of this was to try to be able to solve the bonus, as whenever I tried with the default WalkSAT, it would time out.

Results:

```
WALKSAT solved after 33170 iterations
--- 1362.844714 seconds ---
```

Conclusions:

The resetting WalkSAT was able to find a solution for the bonus puzzle, probably because eventualy it got lucky with a good starting variable assignment configuration. So in all, the goal was accomplished.

Aging WalkSAT (Bonus)

Description:

Next as an extention, I tried to create a version of WalkSAT that tried to address some flaws that I had noticed in the original algorithm. Initially, it does not make much sense to score very often because selected variables are more likely to be correct. But later, it makes sense to score more often because there are not that many unsatisfied clauses, so it is more likely to pick the wrong one without scoring. Hence, I decided to make the algorithm score more often over time. To do this, instead of using the `threshhold` as the probability, I chose to divide the number of unsatisfied clauses with the total number of clauses.

Results:

All Cells:

```
Aging WALKSAT solved after 282 iterations
--- 6.098102 seconds ---
```

Rows

```
Aging WALKSAT solved after 318 iterations
--- 8.079812 seconds ---
```

Rows and Columns:

```
Aging WALKSAT solved after 28086 iterations
--- 1583.859066 seconds ---
```

Conclusions:

This idea showed potential initially, as it solved Rows faster than the default WalkSAT, and in fewer iterations as well. But then, it took exceedingly long to solve Rows and Columns, which makes me suspect that perhaps it is not the best approach. Perhaps my idea is only good for easier problems. It might be because generally a lot of clauess are already satisfied, so that it is more just a very high threshold, rather than starting low and becoming high. Also, it is possibe that the random choices may be very helpful in completing the problem at the very end, as at Rows and Columns, this version of the algorithm spend much time one or two clauses awawy from completion.

Ivor Spence Formulation (Bonus)

As an extention, I decided to try the Ivor Spence formulation described in the assignment, of adding additional redundant constraints, as well as the corresponding simple version provided as well. I used the basic WalkSAT algorithm for these tests. Below are results (I ran tests multiple times as it was reasonably fast):

Results:

Ivor Formulation:

```
WALKSAT solved after 1358 iterations
--- 147.376277 seconds ---

WALKSAT solved after 1210 iterations
--- 121.447424 seconds ---

WALKSAT solved after 746 iterations
--- 69.109538 seconds ---

WALKSAT solved after 718 iterations
--- 68.489958 seconds ---

WALKSAT solved after 544 iterations
--- 47.515231 seconds ---

WALKSAT solved after 968 iterations
--- 94.971503 seconds ---

WALKSAT solved after 648 iterations
--- 61.359355 seconds ---
```

Simple:

```
WALKSAT solved after 2688 iterations
--- 101.97136499999999 seconds ---

WALKSAT solved after 4040 iterations
--- 146.306825 seconds ---

WALKSAT solved after 3674 iterations
--- 135.32663 seconds ---

WALKSAT solved after 4974 iterations
--- 189.17088600000002 seconds ---

WALKSAT solved after 1560 iterations
--- 53.645431 seconds ---

WALKSAT solved after 4610 iterations
--- 174.677594 seconds ---
```

Conclusion:

It seems that there is a rather large variance between results each time. This makes some sense because

random variables are being flipped each time, so sometimes you might get lucky and flip the correct variables, and other times you will not. Regardless, some interesting conclusions can be made. First is that the redundant formulation that Ivor uses did indeed use fewer iterations to complete the problem. And it seems that in general, it was also faster. But, it is also worth noting that sometimes the simple version was faster despite iterating more. Perhaps this is because there were fewer clauses to iterate through each round, as well as fewer unsatisfied clauses for the most part. In the end, it appears that in general, Ivor's formulation does end up being faster for the most part, but that is not always the case based on luck of the draw.

Literature Review (Bonus)

For further exploration, I decided to read "Chaff: Engineering an Efficient SAT Solver" by Moskewics, Madigan, Zhao, Zhang, and Malik. I decided upon this article, because frankly, these methods seems to be much less efficient than backtracking search. It seems exceeding slow to me to spend 4 seconds on filling up an entire Sudoku board with random numbers that do not even have to do anything with the rules of the game itself. It seems like that could have been accomplished with a simple nested for loop that would take a total of 81 iterations. Hence, I was hoping that this article would provide a more efficient version of a SAT solver, which could compete with backtracking search. The article proposes such an algorithm, called "Chaff".

In all, it seems like backtracking search may just be superior. Chaff is a variant of Davis-Putnam backtrack search, which uses backtracking rather than random walks to find variable assignments that satisfy all clauses. Chaff makes certain optimizations over the traditional Davis-Putnam search, which the authors claim make it at least an order of magnitude faster. Below is a run-down of said optimizations.

It runs a boolean constriant propagation (BCP), which is meant to identify any variable assignemnts in the current assignment that are required by the current variable state to satisfy the problem. Those are essentially the low hanging fruit, and putting them in the correct state will speed either success of failure, both of which will lead to finding the final solution faster. In order to optimize this, Chaff keeps track of the number of zero literals in each clause (variables assigned to `False`). It ignores each clause as long as there are `N-2` or fewer variables that are set to `False` in each clause (where `N` is the number of variables in each clause). Only when the number of variables in the clause that are assigned to `False` reach `N-1` does the BCP determine that the last variable must be `True`. Keeping track of this each time means that it is not necessary to iterate through each cause every time in the BCP step of the algorithm.

Chaff uses clause detection to know when certain clauses are no longer important in the scheme of the problem, as they are easy to satify. It then marks those clauses as deleted, so that they need not be iterated through later in the problem.

Finally, Chaff uses restarts, or resets, similar to what I implemented in the Resetting WalkSAT algorithm that I wrote. The advantage of the restarts that Chaff uses is that they calculate the ideal time in which a restart may be necesary, as a local minima may have been reached. This means that it will not restart when the solution is

nearing the actual solution. Also, CHaff does not restart to a random variable assignment like how mine does. Rather, it restarts to a assignment state where it knows that a certain number of clauses have been satisfied, so that it can avoid doing a lot of the same work all over again.

I thought that these optizations were interesting both from a SAT problem point of view, as well as for optimizing bracktracking search, which has to do with the previous CSP assignment as well. It is interseting to see how these methods can be used for multiple kinds of problems, and how they can generate solutions relatively quickly.