Boolean Satisfiability (SAT) Solving

Ben Williams '25, October 2023

Problem Representation

We consider all SAT problems in conjunctive normal form (CNF). Our .cnf files have can have any variable names they want (be it `112` or `BLUE`), except that a variable being false is represented with a `-` in front. Each line in the .cnf is a clause, where spaces between variables represent "or" operators.

For example, the line `-111 112 -113`, indicates that `112` is False or `112` is True or `113` is False.

The goal is to find an assignment to all the variables such that all the clauses are satisfied, if it exists. All the algorithms here are generic and can solve any SAT problem with the above formatting (with the `-` for False, otherwise True).

Sudoku

The main SAT problem that we solve here is Sudoku. We can explicitly define sudoku with boolean variables in the format `rcn` where `r` is the row, `c` is the column, and `n` is the number. Sudoku problems have 729 variables (9 rows * 9 columns * 9 numbers) and usually have about ~3000 clauses.

GSAT Solver

The GSAT algorithm is very simple. We do the following:

1) Generate a random assignment
2) If the assignment satisfies all clauses, return the assignment
3) With some probability h, flip a random variable in the assignment and return to step 2
4) Otherwise, flip the variable that would cause the most clauses to be satisfied (break ties randomly). Return to step 2

This algorithm can run infinitely if the problem is unsatisfiable. Our implementation has a `max_steps` parameter that forces the algorithm to break out of the algorithm after that many steps have occurred without success.

Scoring - Picking the best variable

For GSAT, the process of finding the best variable to flip is incredibly slow. We loop through each variable, temporarily flip its value in the assignment, and count how many clauses are satisfied.

For example, if we had 500 variables and 2000 clauses, this step would loop through about 1,000,000 times just to flip a single variable.

Discussion - Randomness

Allowing variables to be flipped randomly can push us out of local minima in the solution space, but also has the chance to slow us down significantly (by pushing us farther from a solution). Keeping a low `h` value balances this appropriately, but matters more in the Walksat as we will see later. The `h` value used for the GSAT testing was 0.2.

Results

GSAT is able to fill in all the cells of sudoku with numbers, do so following the row rules. More complex cnfs (such as the rows and columns combined) take too long for GSAT to complete

Having one number 1-9 in each Sudoku cell: (all_cells.cnf)

```
Total steps: 330
Time elapsed:  0:03:55.591404
4 5 2 | 4 4 6 | 7 3 5
7 2 1 | 4 8 9 | 8 9 2
8 2 2 | 6 8 2 | 1 5 9
---------------------
5 2 3 | 7 8 5 | 2 7 5
3 3 6 | 8 3 1 | 6 3 3
7 1 8 | 7 6 2 | 1 4 6
---------------------
2 4 3 | 3 5 1 | 1 7 5
7 1 2 | 5 7 8 | 7 1 9
8 2 3 | 9 6 7 | 9 8 8
```

Sudoku considering only the row rules: (rows.cnf)

```
Total steps: 418
Time elapsed:  0:05:03.520015
3 1 9 | 2 6 4 | 5 8 7
3 2 1 | 5 8 9 | 7 4 6
7 4 2 | 1 8 3 | 6 5 9
---------------------
3 9 7 | 4 8 5 | 2 6 1
7 3 9 | 6 4 2 | 1 5 8
4 1 5 | 8 9 3 | 2 7 6
---------------------
3 7 4 | 1 5 6 | 2 8 9
7 5 8 | 2 9 6 | 3 1 4
3 8 6 | 4 5 7 | 9 2 1
```

For the rows test, we average only ~1.38 steps per second.

Walksat

Walksat is very similar to GSAT, except that it takes a more refined approach for deciding which variables to score and which variables it might randomly flip. Walksat chooses an unsatisfied clause at random. Then with probability `h` it flips a random variable within that unsatisfied clause, otherwise it flips the variable within the unsatisfied clause that causes the most clauses to be True (the same scoring as GSAT).

Walksat is much faster because rather than looping through *all* variables, it only has to look at the variables in a single clause. Therefore, the speed of Walksat is partially determined by the average clause size. In Sudoku, most clauses are only 2 variables long, and the longest ones are only 9 variables long. Compared to looping through 729 * `num_clauses`, this is a major improvement. Nevertheless, Walksat still runs into the issue of local minima. We try to mitigate this with the random variable flipping. More about this in the below discussion.

Results

We are able to solve much more complex problems with the Walksat. While we still show the number of steps taken, note that each step in Walksat is much faster than each step in GSAT. We have `h=0.3` in the below tests.

We begin with the rows example, where we take a similar amount of steps, but each step is **much** faster:

```
Total steps: 404
Time elapsed:  0:00:01.417536
4 6 9 | 3 7 2 | 1 8 5
7 1 5 | 3 9 6 | 8 4 2
4 3 6 | 7 5 2 | 8 9 1
---------------------
7 1 5 | 2 3 8 | 4 6 9
2 8 9 | 4 3 7 | 6 5 1
7 3 8 | 1 5 2 | 4 9 6
---------------------
6 3 8 | 1 9 4 | 5 7 2
3 2 8 | 5 7 6 | 9 4 1
4 8 9 | 5 3 6 | 7 2 1
```

Here, we averaged ~286.52 steps per second. About 207 times faster per step than GSAT.

Now, we can use Walksat to solve the actual Sudoku puzzles (implied in this is the solutions to the rows_and_cols.cnf and rules.cnf, since this expands upon them). Here is the solution to puzzle1 (.cnf):

```
Total steps: 7096
Time elapsed:  0:00:38.581984
```

```
5 8 2 | 3 7 6 | 1 4 9
1 4 6 | 5 9 8 | 3 7 2
3 9 7 | 1 4 2 | 5 6 8
---------------------
8 3 9 | 2 6 7 | 4 5 1
6 1 4 | 8 5 9 | 7 2 3
7 2 5 | 4 3 1 | 8 9 6
---------------------
9 7 1 | 6 8 5 | 2 3 4
2 6 3 | 7 1 4 | 9 8 5
4 5 8 | 9 2 3 | 6 1 7
```

Here is the result to puzzle2.cnf, a more difficult puzzle:

```
Total steps: 32224
Time elapsed:  0:02:22.477109
1 3 7 | 2 9 6 | 5 4 8
2 8 4 | 1 7 5 | 6 3 9
6 9 5 | 3 4 8 | 1 7 2
---------------------
8 4 1 | 9 6 7 | 2 5 3
5 6 9 | 8 3 2 | 4 1 7
7 2 3 | 5 1 4 | 8 9 6
---------------------
9 7 8 | 6 5 1 | 3 2 4
3 1 6 | 4 2 9 | 7 8 5
4 5 2 | 7 8 3 | 9 6 1
```

Discussion - Randomness (again)

The randomness for the Walksat is especially important as we are more likely to run into local minima here. Experimentally in the Sudoku examples, we frequently get stuck with having 82 variables as True, where we should only have 81 in the final answer. We end up here quickly, but can take thousands (or tens of thousands) of steps to escape and finally find a solution.

For the `puzzle_bonus`, having an `h=0.3` caused the program to reach the `max_steps` breakpoint at both 100,000 and 200,000 steps respectively (at least for the random seed of 1). However, trying a `h=0.5`, which is a bit more extreme, allowed the problem to be solved in ~50,000 steps. This leads me to believe that what made `puzzle_bonus` difficult is the presence of **many local minima** in the solution space, though it is hard to say for certain.

Puzzle bonus solution:

```
Total steps: 76388
Time elapsed:  0:05:45.410848
5 3 4 | 6 7 8 | 9 1 2
```

```
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
```

Walksat Modified - Constants

In the modified Walksat (see SATExtra), we take into account constants in the clauses, taking care not to modify them. The normal Walksat would potentially flip constant terms in unsatisfied clauses, even though we already knew what a variable's value was supposed to be. Note that this is not necessarily a good thing, as we will see.

Is this a good idea?

While considering constants, and therefore reducing the amount of variables we have to consider may at first seem like a good thing, it reduces the amount of randomness in the algorithm. The reason we were flipping variables randomly was to be able to escape local minima.

The argument against considering constants is that it makes the algorithm more rigid; this could make it more likely to get stuck in local minima. Moreover, if we end up stuck in local minima, it may be harder to escape. After all, we made a sudoku problem harder by adding in constraints in the first place.

The argument for considering constants is that it reduces the amount of variables we need to consider, and therefore reduces the search space. We can go beyond just the explicitly defined constants and go on to reason out every guaranteed variable value before starting. This approach can further reduce the number of variables even more.

Only explicitly defined constants

It is trivial to find the explicitly defined constants in the clauses. If a clause only has one variable (regardless of it being defined as True or False), then the variable must be a constant.

All constants

It is a bit more difficult to find what I call "implied constants". To give an example, say we have the following clauses:

```
112
-111 -112
-112 -113 -114
```

In these clauses, `112` is explicitly defined as True, and we know that it is a constant. However, from the second line `-111 -112`, we know that either `111` is False **or** `112` is False. Since we know that `112` is True, that means that we know that `111` is False - so that the clause can stay true. So, `111` being False is implied by `112` being True. However, `112` being True does not give enough information for `113` or `114` to be able to determine their truth values.

In order to find all the implied constants, we first start a queue with all the clauses that the known constants are in, and initialize a visited set. Then we do the following: 1) Pop a clause from the queue 2) Loop through every variable in the clause. If **one** and only one variable is not a constant, then continue to step 3. Otherwise, go to the next clause (step 1) 3) Add the variable and its value to the constants 4) Add all clauses that this variable was involved in to the queue 5) Add this clause to the visited set so that we do not evaluate it again

This ensures that we add all implied constants, and implied constants of implied constants, and so on until we have done all the preprocessing that we can do.

Results

We try to solve puzzle1 with only the explicitly defined constants and then with all the constants (including implied ones). We can then compare this to the results of the normal Walksat to see what the most effective method is. We have `h=0.3` for all of these.

Puzzle1 results with only explicitly defined constants:

```
Total of 10 constants out of 729 variables
Total steps: 7844
Time elapsed:  0:00:46.555901
5 6 9 | 1 8 7 | 4 3 2
2 3 1 | 4 9 6 | 5 7 8
4 7 8 | 5 3 2 | 1 6 9
---------------------
8 2 5 | 7 6 1 | 9 4 3
9 1 3 | 8 2 4 | 6 5 7
7 4 6 | 3 5 9 | 8 2 1
---------------------
3 5 7 | 9 4 8 | 2 1 6
6 8 4 | 2 1 3 | 7 9 5
1 9 2 | 6 7 5 | 3 8 4
```

Puzzle1 results with **all** constants (including implied ones):

```
Total of 90 constants out of 729 variables
```

```
Total steps: 25140
Time elapsed:  0:02:28.644542
5 8 9 | 1 7 6 | 3 2 4
3 6 2 | 5 9 4 | 7 8 1
1 7 4 | 2 8 3 | 5 6 9
---------------------
8 1 3 | 9 6 7 | 4 5 2
9 4 6 | 8 5 2 | 1 7 3
7 2 5 | 3 4 1 | 6 9 8
---------------------
4 9 8 | 7 3 5 | 2 1 6
2 3 7 | 6 1 8 | 9 4 5
6 5 1 | 4 2 9 | 8 3 7
```

Puzzle2 results with only explicitly defined constants:

```
Total of 11 constants out of 729 variables
Total steps: 14545
Time elapsed:  0:01:32.308188
6 3 8 | 4 2 5 | 9 1 7
2 5 7 | 1 9 8 | 6 3 4
4 9 1 | 6 3 7 | 5 8 2
---------------------
8 1 5 | 7 6 4 | 2 9 3
9 6 2 | 8 5 3 | 7 4 1
7 4 3 | 9 1 2 | 8 5 6
---------------------
3 7 9 | 5 4 6 | 1 2 8
1 8 4 | 2 7 9 | 3 6 5
5 2 6 | 3 8 1 | 4 7 9
```

Puzzle2 results with **all** constants:

```
Total of 99 constants out of 729 variables
Total steps: 7524
Time elapsed:  0:00:41.547665
1 3 8 | 9 5 2 | 7 6 4
2 5 4 | 1 7 6 | 8 3 9
6 9 7 | 3 8 4 | 1 5 2
---------------------
8 1 2 | 5 6 7 | 9 4 3
9 6 5 | 8 4 3 | 2 1 7
7 4 3 | 2 9 1 | 5 8 6
---------------------
5 2 9 | 4 3 8 | 6 7 1
3 8 6 | 7 1 9 | 4 2 5
4 7 1 | 6 2 5 | 3 9 8
```

So far, the implementation of constants into the algorithm has had mixed results. It underperformed on puzzle1, but outperformed on puzzle2. Let us look at the puzzle_bonus - which we had to play with `h`'s values in order for normal Walksat to solve.

Puzzle_bonus only explicitly defined constants:

```
Total of 27 constants out of 729 variables
Total steps: 11642
Time elapsed:  0:01:08.593167
(puzzle ommitted for space)
```

Puzzle_bonus with **all** constants:

```
Total of 243 constants out of 729 variables
Total steps: 10400
Time elapsed:  0:00:48.456684
(puzzle ommitted for space)
```

So including the constants allows the modified Walksat to vastly outperform on the harder puzzle. So far, we had been using a constant random seed for all the trials. What if we took the average of 10 trials for each situation for puzzle1?

Normal walksat puzzle1 stats:

```
Average steps for successful solving: 26537.8
Total times failed: 0
Time elapsed:  0:25:21.898647
```

Puzzle1 with only explicitly defined constants:

```
Average steps for successful solving: 6596.1
Total times failed: 0
Time elapsed:  0:08:09.922284
```

Puzzle1 with **all** constants

```
Average steps for successful solving: 9222.6
Total times failed: 0
Time elapsed:  0:16:26.904236
```

Discussion

Overall, the results on this algorithm are quite interesting and a bit inconclusive. We can see that the constants-algorithm performed especially well on the harder puzzles. It appeared at first that the normal Walksat algorithm would outperform the constants-algorithm, but after further investigation that does not seem to be the case.

Overall, it seems that including the constants into the algorithm does in fact speed it up. However, that does not mean that it will **always** be faster than the other algorithm, due to the randomness embedded in them.

One final interesting thing to point out is how the only-explicit-constants outperformed the all-constants algorithm for puzzle1. Speculating on *how many* constants we should keep track of (from just a few to all of them) is a bit difficult, but perhaps there is an ideal balance, and would be an interesting thing to look at in the future.