A4: Constraint Satisfaction Problem

Kevin King

Dartmouth COSC 76, Fall 2023

Implementation

The SAT class is designed to solve the satisfiability problem using CNF files. The class offers implementations of the GSAT and WalkSAT algorithms.

Initialization and Setup:

- `__init__` : Initializes the SAT object with the filename of a CNF file, parses the file to set up the problem.
- `setup_problem` : Reads the CNF file, parses each clause, and maps variables to indices for efficient handling.

Assignment and Evaluation

- `random_assignment` : Assigns random Boolean values to all variables to start the search for a solution.
- `check_all_clauses` : Checks if all clauses are satisfied with the current variable assignments.
- `num_satisfied` : Counts how many clauses would be satisfied if the value at a given index is flipped.
- `is_clause_satisfied` : Checks if a specific clause is satisfied.

SAT Solving Algorithms

- `gsat` : Implements the GSAT algorithm to find a satisfying assignment, with a random element and a mechanism to flip variable assignments to maximize the number of satisfied clauses.
- `walksat` : Implements the WalkSAT algorithm, a more advanced and often more effective version of GSAT, to find a satisfying assignment.

Testing

`GSAT`

```
Testing GSAT with one_cell
Time elapsed: 0.0006668567657470703 seconds
1 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
```

```
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0

Testing GSAT with all_cells
Time elapsed: 381.08830308914185 seconds
1 4 6 | 6 8 8 | 7 2 5
4 7 7 | 3 2 7 | 1 4 4
5 2 8 | 2 4 2 | 6 6 8
---------------------
7 9 5 | 2 2 1 | 2 5 7
6 1 9 | 4 5 5 | 1 1 3
6 6 3 | 9 6 4 | 8 9 1
---------------------
8 2 4 | 7 7 5 | 7 9 9
9 9 2 | 2 6 2 | 1 7 5
4 3 7 | 9 5 1 | 6 2 2
```

Discussion of Results

- `one_cell`
  - The algorithm solves this trivial case almost instantaneously, taking a negligible amount of time, which is expected given the simplicity of the task.
- `all_cells`
  - The time taken by GSAT for this case is considerably high, clocking in at a bit over 6 minutes. This could be due to the algorithm making numerous attempts to satisfy all clauses (constraints) of the SAT problem representing the Sudoku puzzle. Given that the constraints were probably inconsistent (due to the lack of uniqueness in numbers), the algorithm might have had difficulty converging to a solution, leading to the long runtime.

`WalkSAT`

```
Testing WalkSAT with one_cell
Time elapsed: 2.002716064453125e-05 seconds
1 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0

Testing WalkSAT with all_cells
Time elapsed: 1.5894019603729248 seconds
9 8 2 | 6 2 7 | 3 1 7
6 3 7 | 9 9 1 | 8 1 8
6 4 3 | 3 5 9 | 4 3 5
---------------------
7 3 1 | 1 2 5 | 2 3 8
1 2 6 | 5 8 6 | 3 2 1
8 1 3 | 4 5 2 | 8 1 3
---------------------
```

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

Testing WalkSAT with rows
Time elapsed: 2.6339499950408936 seconds
9 5 3 | 8 4 1 | 6 2 7
8 6 4 | 3 2 9 | 1 5 7
1 6 8 | 5 7 3 | 9 4 2
---------------------
2 5 1 | 7 6 4 | 9 8 3
8 1 2 | 9 5 4 | 6 3 7
7 1 8 | 6 3 5 | 9 2 4
---------------------
9 8 5 | 6 7 1 | 2 4 3
9 8 2 | 6 3 5 | 1 4 7
5 1 8 | 9 7 6 | 4 3 2

Testing WalkSAT with rows_and_cols
Time elapsed: 37.30877900123596 seconds
5 7 2 | 1 6 9 | 3 4 8
6 8 4 | 2 9 7 | 5 1 3
4 9 8 | 6 1 3 | 7 5 2
---------------------
7 1 6 | 4 2 8 | 9 3 5
2 6 9 | 3 5 4 | 1 8 7
3 5 7 | 8 4 1 | 6 2 9
---------------------
1 2 5 | 7 3 6 | 8 9 4
9 3 1 | 5 8 2 | 4 7 6
8 4 3 | 9 7 5 | 2 6 1

Testing WalkSAT with rules
Time elapsed: 69.68923807144165 seconds
3 4 8 | 9 2 7 | 1 5 6
5 1 2 | 6 4 8 | 9 3 7
6 7 9 | 1 5 3 | 4 2 8
---------------------
7 8 3 | 4 1 2 | 6 9 5
9 2 4 | 5 7 6 | 3 8 1
1 6 5 | 8 3 9 | 2 7 4
---------------------
8 5 6 | 3 9 1 | 7 4 2
4 3 7 | 2 6 5 | 8 1 9
2 9 1 | 7 8 4 | 5 6 3

Testing WalkSAT with puzzle1
Time elapsed: 44.229771852493286 seconds
5 3 9 | 2 7 6 | 1 8 4
6 7 4 | 1 9 8 | 3 5 2
1 2 8 | 5 3 4 | 7 6 9
---------------------
8 9 3 | 7 6 5 | 4 2 1
2 5 1 | 8 4 3 | 9 7 6
7 4 6 | 9 2 1 | 5 3 8
---------------------
3 1 5 | 6 8 9 | 2 4 7
4 8 7 | 3 1 2 | 6 9 5
9 6 2 | 4 5 7 | 8 1 3

Testing WalkSAT with puzzle2
```

```
Time elapsed: 507.68140602111816 seconds
1 3 7 | 9 2 6 | 5 8 4
5 2 8 | 1 3 4 | 6 7 9
4 9 6 | 5 7 8 | 3 1 2
---------------------
8 5 9 | 2 6 7 | 1 4 3
2 6 4 | 8 1 3 | 9 5 7
7 1 3 | 4 9 5 | 8 2 6
---------------------
6 7 5 | 3 8 2 | 4 9 1
3 8 1 | 7 4 9 | 2 6 5
9 4 2 | 6 5 1 | 7 3 8

Testing WalkSAT with puzzle2
Time elapsed: 348.23854184150696 seconds
1 3 4 | 2 5 8 | 7 6 9
6 7 2 | 1 9 3 | 5 8 4
5 9 8 | 7 4 6 | 3 1 2
---------------------
8 2 9 | 4 6 7 | 1 5 3
3 6 1 | 8 2 5 | 9 4 7
7 4 5 | 9 3 1 | 8 2 6
---------------------
9 1 6 | 5 7 2 | 4 3 8
4 8 3 | 6 1 9 | 2 7 5
2 5 7 | 3 8 4 | 6 9 1
```

Discussion of Results

It appears that WalkSAT has been used to solve different instances of Sudoku puzzles. The `one_cell` test seems to be a base case where only one cell is filled, `all_cells` where all cells are filled (not according to Sudoku rules), `rows` where each row is populated while maintaining the unique number constraint, and `rows_and_cols` where both rows and columns are adhering to the Sudoku rules. The WalkSAT algorithm's efficiency varies depending on the complexity of the constraints. It solves simpler puzzles quickly but takes considerably longer as constraints increase. WalkSAT also seems to operate much faster than GSAT on the more difficult Sudoku puzzles.

Observations

- `one_cell` - only one cell is filled in the Sudoku grid.
    - This is likely a base case to ensure the system is operational. The WalkSAT algorithm probably did not have much work to do here.
- `all_cells` - all cells are filled but do not adhere to the Sudoku rules, i.e., there are repetitions within rows, columns, and blocks.
    - The algorithm filled all cells quickly but did not enforce Sudoku constraints, leading to an invalid solution.
- `rows` - rows are filled correctly according to Sudoku rules, but columns and blocks constraints are not maintained.
    - The algorithm likely had constraints for rows but not for columns or blocks. The increase in time is probably due to the added complexity of maintaining unique numbers in each row.
- `rows_and_cols` - both rows and columns are populated while following Sudoku rules, but blocks constraints are not followed.
    - The significant increase in time suggests that adding column constraints increased complexity considerably. The solution is still invalid for a standard Sudoku puzzle due to the lack of block constraints.

- `rules`
  - The WalkSAT algorithm has a considerable increase in computation time, suggesting that this test case was more complex due to enforcing all Sudoku rules. This is reflected in the elapsed time which is relatively higher than the previous test cases. The resulting grid adheres to the standard rules of Sudoku, indicating a valid solution.
- `puzzle1`
  - The WalkSAT algorithm's computation time is substantial but lower than the full `rules` test, suggesting that while it is a standard Sudoku puzzle, it might be of medium difficulty or the initial configuration made it relatively easier to solve.
- `puzzle2`
  - The computation times are different between the two runs. The first run took much longer than the second, which suggests that WalkSAT, being a stochastic algorithm, can have varying performance on the same problem instance depending on the random paths it takes during search.

Extensions

1. Map-Coloring Problem

Aims to solve the classic map coloring problem using a SAT solver. The goal of the map coloring problem is to color the map in such a way that no two adjacent regions share the same color.

The design breaks down the problem into three main parameters:

- `variables` : Each region (or state) is represented by a variable (e.g., WA, NT, etc.).
- `neighbors` : This specifies which regions are adjacent or neighboring to each other.
- `domains` : This specifies the colors available for coloring the map.

Implementation:

- The `setup_problem` method writes the CNF clauses based on the following constraints:
  - Assign at least one color to each region: This ensures every region is colored.
  - Ensure each region only has one color: This ensures no region gets two colors.
  - Ensure neighbors do not have the same color: This ensures adjacent regions don't have the same color.
- The `display_solution` method reads a given solution file and prints out the non-negated assignments (i.e., the positive solutions).

Two different runs of the program yield two different, but valid, solutions for the coloring of the map:

```
First Run
WA_red
NT_blue
SA_green
Q_red
NSW_blue
V_red
T_blue

Second Run
WA_blue
NT_red
SA_green
```

```
Q_blue
NSW_red
V_blue
T_blue
```

2. N-Queens Problem

Aims to solve the N-Queens problem using a SAT solver. The N-Queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other (i.e., no two queens are on the same row, column, or diagonal).

Implementation:

- The `setup_problem` method breaks down the N-Queens constraints into CNF clauses:
  - Row and Column Constraints: Each row and each column must have exactly one queen.
  - No Two Queens in Same Row or Column: No two queens can be placed in the same row or the same column.
  - Diagonal Constraints: No two queens can be on the same diagonal. This is checked for both the primary diagonal (top-left to bottom-right) and the secondary diagonal (top-right to bottom-left).
- The `display_solution` method takes a solution file as input and displays the board with queen positions.

Two runs of the program produce two different, yet valid, solutions to the 8-queens problem:

```
First Run
..Q.....
......Q.
.Q......
.......Q
....Q...
Q.......
...Q....
.....Q..

Second Run
....Q...
..Q.....
.......Q
...Q....
......Q.
Q.......
.....Q..
.Q......
```