SAT Solver - CSCE 411 Extra Credit

December 7, 2015

By: Bailey, Matthew, Jennings

Clause Generator

The ClauseGenerator in this code generates the boolean CNF clauses that will get fed into the SAT solver. The variables on the same line are joined by ORs and each line is joined by ANDs, per Knuth's specification.

1. Each cell must have value 1 through n^2
2. Each row must have value 1 through n^2
3. Each column must have value 1 through n^2
4. Each nxn block must have value 1 through n^2
5. If a cell has a particular value, that cell cannot be any other value
6. If a cell has a particular value, the cells in the same row cannot be the same value
7. If a cell has a particular value, the cells in the same column cannot be the same value
8. If a cell has a particular value, the cells in the same block cannot be the same value

In the PDDL folder, the parser generates PDDL clauses given a sudoku txt file. Compared to the boolean CNF clauses, there is a significantly less number of clauses. For example, for the 25x25 example given, the pddl file only contains less than 1700 lines and the 4x4 sudoku file needs less than 100 lines of clauses.

General Sudoku Rules

The code we have written takes in a given input size n and creates a sudoku board of size n^2 x n^2. Each cell can be one of n^2 values ranging from 1 to n^2, Therefore, each cell has n^2 boolean variables. These are represented through the convention r{row num}c{col num}_{value}, for example r1c1_1.

Since each cell can be one of n^2 values, clauses are generated to reflect this. For example, if n = 2 and n^2 = 4, then the first cell can have a value of 1 through 4 and so on:

```
r1c1_1 r1c1_2 r1c1_3 r1c1_4
r1c2_1 r1c2_2 r1c2_3 r1c2_4
r1c3_1 r1c3_2 r1c3_3 r1c3_4
r1c4_1 r1c4_2 r1c4_3 r1c4_4
.
.
.
r4c4_1 r4c4_2 r4c4_3 r4c4_4
```

Similar clauses are generated to represent the rules for rows, columns, and blocks as well. Each row, column, and block must have values 1 through n^2 and are written like the clauses for the cell rules. Below are the rules for the values of row 1:

```
r1c1_1 r1c2_1 r1c3_1 r1c4_1
r1c1_2 r1c2_2 r1c3_2 r1c4_2
r1c1_3 r1c2_3 r1c3_3 r1c4_3
r1c1_4 r1c2_4 r1c3_4 r1c4_4
```

Similarly, below are the rules for the values of column 1:

```
r1c1_1 r2c1_1 r3c1_1 r4c1_1
r1c1_2 r2c1_2 r3c1_2 r4c1_2
r1c1_3 r2c1_3 r3c1_3 r4c1_3
r1c1_4 r2c1_4 r3c1_4 r4c1_4
```

Below are the rules for the values of the block starting at 1,1:

```
r1c1_1 r1c2_1 r2c1_1 r2c2_1
r1c1_2 r1c2_2 r2c1_2 r2c2_2
r1c1_3 r1c2_3 r2c1_3 r2c2_3
r1c1_4 r1c2_4 r2c1_4 r2c2_4
```

Finally, rules must be generated that state how if a cell is a particular value then no other cells in the same row, column or block can have that value. This is ecapsulated by the statement r1c1_1 -> (~r1c2_1 ^ ~r1c3_1 ^ ~r1c4_1) which is the same as ~r1c1_1 v (~r1c2_1 ^ ~r1c3_1 ^ ~r1c4_1). This is seen by the below statements.

```
~r1c1_1 ~r1c1_2
~r1c1_1 ~r1c1_3
~r1c1_1 ~r1c1_4
~r1c1_2 ~r1c1_3
~r1c1_2 ~r1c1_4
~r1c1_3 ~r1c1_4
```

The same type of thing is done for the remaining rows, columns, and blocks.

Sudoku Puzzle

All of the above represent the general rules for sudoku. If you have a given puzzle with some values filled in, you can translate those values into clauses.

If a given cell is given as the value 1, then that cell cannot be any other value:

```
r1c1_1
```

```
~r1c1_2
~r1c1_3
~r1c1_4
```

Additionally, no other cells in the same row, column or block can have that value:

```
~r1c2_1
~r1c3_1
~r1c4_1

~r2c1_1
~r3c1_1
~r4c1_1

~r2c2_1
```

This is done for every given value in the original puzzle.

SAT Solvers

DPLL

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a method of solving boolean expressions in Conjunctive Normal Form (CNF). The algorithm is a backtracking algorithm with two heuristics. The Pure Symbol heuristic and the Unit Clause Heuristic.

The Pure Symbol heuristic works by finding a symbol in all of the clauses that always has the same sign. For example, if the symbol "a" always appears with the negation "~a", then "a" should be bound to false.

The Unit Clause Heuristic works by finding clauses that do not yet evaluate to true and have all but one symbol already bound. The one unbound variable must receive a binding such that the clause evaluates as true, else the clause may not evaluate to true at all.

These two heuristics offer a major improvement over backtracking, especially with rule-bases that have lots of redundancies.

WalkSAT

WalkSAT is a random algorithm. WalkSAT first randomly binds each symbol to either *true* or *false*. From there if the model is true, the model is returned. Otherwise a random clause that is false is chosen and 50% of the time a random symbol flips is binding (*true* to *false* and *false* to *true*) and the other 50% of the time the symbol that will maximize the number of satisfied clauses is flipped.

The WalkSAT algorithm keeps running until either it returns a model, or it hits *maxFlips* which is set in advance. If the program hits *maxFlips* it gives up and reports that it was unable to find a valid model. Unfortunately, this does not guarantee that no model exists, only that the likelihood of a model existing is low.

PDDL Solver As mentioned before, the number of clauses used for PDDL is not nearly as much as other SAT solvers. However, the speed at which the problem is solved is much slower. These solvers take an initial state, moves that can be made, and a goal state. Using the initial state, the solver generates all possible states where a valid move can be made and continues to do so until the goal state is met. Generating all of these possible states can take an immense amount of time, especially when lots of moves are required to make to reach the goal state.

Compile and Run

To compile, open a linux.cse.tamu.edu terminal. Navigate to the folder and type 'make'.

To run, type the command below:

```
javac ClauseGenerator7 {input file} {n} {clause output file}
```

For example, to run the 4x4 sudoku board you would type:

```
javac ClauseGenerator7 input2x2.txt 2 clauses2x2.txt
```