Eduardo Szeckir
Rebecca Marshall
Emma Dewit
Sylvain Taghaoussi

SAT-based Sudoku Solver

1. Background

This paper presents a Python implementation of a SAT-based Sudoku Solver. Sudoku is a puzzle in which the objective is to fill an n by n grid with values ranging from 1 to n, generally every value only appears once in a column, row or sub-grid. This report is written under the assumption that the puzzles are 9x9 grids with 3x3 sub-grids. Two main programs were implemented, sud2sat and sat2sud along with variations of each that build onto the two main programs that use different encodings. The first will translate sudoku puzzles to conjunctive normal form (CNF) and subsequently solve the clauses using satisfiability, while the latter will take the output of miniSAT based off the first implementation and convert it back into a solved sudoku puzzle.

2. Implementations

2.1 Sudoku to Satisfiability problem.

Initially, each sudoku puzzle is encoded in a text file as a string of 81 characters where each is either a value, in our case between 1 or 9 representing an already filled in cell or a wildcard character in likes of a “0”, ”.“ , “*” or “?” which indicates an empty cell (see Fig 1.0a,b for example encodings). We created a function that reads the unsolved puzzle that’s encoded from STDIN. This function will then loop over the inputted puzzle and change all wildcard characters into zeros and keep the values as is and return this unsolved puzzle in an array. This allows us to have a consistent format to work with throughout our implementation.

```
1638.5.7.
..8.4..65
..5..7..8
45..82.39
3.1....4.
7........
839.5....
6.42..59.
....93.81
```

Fig 1.0a: Example sudoku puzzle encoding

```
163805070008040065005007008450082039301000040700000000839050000604200590000093081
```

Fig 1.0b: Example sudoku puzzle encoding

Once we have the unsolved puzzle in an array, the next step is to create a function that finds the clauses in order to get the puzzles into CNF formulas. This function takes in the puzzle array as a parameter and uses the constraints based on a particular encoding to get into clausal form. This implementation uses minimal encoding. For further implementations and more detailed explanations of encodings go to the section on *Encodings*. There are four main constraints associated with minimal encoding in order to solve sudoku puzzles: row, column, sub-grid and that every cell has a value (see Section 3.1). Before continuing on with the implementation we have to define some syntax: let the variable X_{ijk} denote the i^{th} row, j^{th} column and k be the value in cell (i,j). Thus, we have a variable for every single possible assignment to a cell in the 9 x 9 grid.

For the implementation, we initialize an empty array of clauses and loop through the unsolved puzzle to determine any already filled cells. If the cells are pre-filled, we append the variable to the array. However, since we are using the DIMACS format each clause is represented as a nonzero integer so we must convert each X_{ijk} variable to a unique positive integer before appending the variable to the clause. We use the following formula to encode the variable:

$$81 * (i - 1) + 9 * (j - 1) + (k - 1) + 1$$

Recall that i is the row, j is the column and k is the value. Once we have enforced that all the pre-filled values of the sudoku puzzles have been added, we can continue to the four constraints (see Section 3.1 for detailed version of constraints).

2.1.1 Value constraint

The first constraint is to ensure that every cell has at least one value. This is accomplished by having a nested loop that goes through every row and column, which forms the conjunctions. Then we append every value between 1 and 9 for every cell in its encoded form which creates a disjunction of all the clauses for all possible assignments.

2.1.2 Row & Column constraints

The next two constraints that we must satisfy are very similar in terms of their process so we will just cover one: the row constraint. We have a group of nested loops for, in the order of outermost to innermost, row (i), value (k), column (j), and every column to the right (l) which will form the conjunction of all the clauses. In the innermost loop we append the inverse of the encoding (the negative version of the original encoding) at X_{ijk} and X_{ljk}. This ensures that if a value k is assigned to cell (i,j) , then k will not appear in any cell to the right which is sufficient to satisfy the constraint.

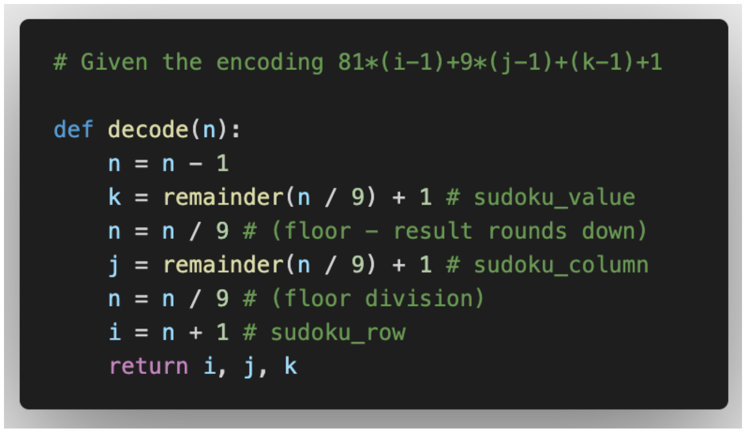
The column constraint is identical except we check that if a value k is assigned in cell (i,j) then k will not appear directly below it in any cell so we just swap i and j in the encoding.

2.1.3 Sub-grid constraint

Lastly, we have the sub-grid constraint – the most difficult to implement. We start in the top left cell of the grid, move to the right cell by cell until the rightmost cell in the grid, then move down a row and start at the leftmost cell again, this process repeats until we have reached the final cell in the grid. If we say that the first visited cell is 1 and the last is 9, then we know that if a value k appears in a cell, then it cannot appear in a cell with a greater number. We do a number of nested for loops to perform this process and to construct the conjunctions needed for CNF. We encode the variable in the innermost formula, forming a clause for every possible outcome in the sub-grids using the negative encoding. Now that we have all of the clauses in CNF, we convert the clauses into DIMACS, write to STDOUT and input into the miniSAT solver to determine satisfiability of the puzzle.

2.2 Satisfiability problem to a Solved Sudoku

The next task of this project is to read the output of miniSAT from the encoded puzzle formed in the implementation of sudoku to SAT and convert it into a solved sudoku puzzle. We first read in the output from STDIN and initialize an array of arrays of zeros to form a 9x9 “grid”. We loop through the output given by miniSAT, and check to see whether the values are positive or negative. If they are negative it means that this encoded value does not satisfy the puzzle, and so are assigned "false", and we loop to the next value. Otherwise, if they are positive, they are assigned "true" and we know that this value satisfies the constraints and we continue on to decoding the value. We must decode these values to determine their placement in the grid and what value is contained in that cell. In order to decode the values, we must solve for i, j and k to determine what row, column and what value is in each of these cells. The process of decoding goes as follows:



```
# Given the encoding 81*(i-1)+9*(j-1)+(k-1)+1

def decode(n):
    n = n - 1
    k = remainder(n / 9) + 1 # sudoku_value
    n = n / 9 # (floor - result rounds down)
    j = remainder(n / 9) + 1 # sudoku_column
    n = n / 9 # (floor division)
    i = n + 1 # sudoku_row
    return i, j, k
```

After we have decoded all the values and determined which cell they are assigned to, we place each value k in the array at the correct cell. Now we have a completed transformation from SAT to sudoku. Lastly, we print off the solved sudoku puzzle in a 9x9 grid. This concludes our implementation for converting a SAT sudoku problem to a solved sudoku.

3. Encodings

Sudoku puzzles can be easily translated into a SAT problem. Recall that we are solely focusing on 9x9 sudoku puzzles with 3x3 sub-grids. For every possible assignment of a value to a cell in the main grid, each cell can be assigned a value from 1 to 9, so in total there are 9x9x9=729 variables. Remember X_{ijk} denotes the cell (i, j) that contains value k, i.e., X_{113} represents cell (1,1) which holds the value 3. These variables will be used as a basis for the minimal, efficient and extended encodings covered in this project to solve the sudoku SAT problem.

3.1 Minimal Encoding

The minimal encoding of a sudoku puzzle uses the least number of constraints needed to successfully solve the sudoku problem [1]. In this particular encoding there are four constraints.

1. Every cell contains at least one number

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigvee_{k=1}^{9} x_{ijk}$$

2. Each number appears at most once in every row

$$\bigwedge_{i=1}^{9}\bigwedge_{k=1}^{9}\bigwedge_{j=1}^{8}\bigwedge_{\ell=j+1}^{9}(\neg x_{ijk} \vee \neg x_{i\ell k})$$

3. Each number appears at most once in every column

$$\bigwedge_{i=1}^{9}\bigwedge_{k=1}^{9}\bigwedge_{j=1}^{8}\bigwedge_{\ell=j+1}^{9}(\neg x_{ijk} \vee \neg x_{i\ell k})$$

4. Each number appears at most once in every 3x3 sub-grid

$$\bigwedge_{k=1}^{9}\bigwedge_{a=0}^{2}\bigwedge_{b=0}^{2}\bigwedge_{u=1}^{3}\bigwedge_{v=1}^{2}\bigwedge_{w=v+1}^{3}(\neg x_{(3a+u)(3b+v)k} \vee \neg x_{(3a+u)(3b+w)k})$$

$$\bigwedge_{k=1}^{9}\bigwedge_{a=0}^{2}\bigwedge_{b=0}^{2}\bigwedge_{u=1}^{2}\bigwedge_{v=1}^{3}\bigwedge_{w=u+1}^{3}\bigwedge_{t=1}^{3}(\neg x_{(3a+u)(3b+v)k} \vee \neg x_{(3a+w)(3b+t)k})$$

3.2 Efficient Encoding

The efficient encoding adds a certain constraint that could possibly result in faster run times over those implementations using minimal encodings or extended encodings. This is because these extra constraints could aid in lowering the amount of clauses. For this project we chose to add the following constraint on top of the minimal encoding constraints;

1. There is at most one number in each cell.

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{8}\bigwedge_{\ell=k+1}^{9}(\neg x_{ijk}\vee\neg x_{ij\ell})$$

We have implemented the efficient encoding in the *sud2sat2* file. The process is identical to the one described above for just the minimal encoding but we cover the one additional encoding.

3.3 Extended Encoding

The extended encoding builds on to the minimal encoding by adding more constraints that the puzzle must adhere to. There are numerous additional constraints that one could add to create an extended encoding but we have implemented our programs based on these four additional constraints.

1.There is at most one number in each cell.

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{8}\bigwedge_{\ell=k+1}^{9}(\neg x_{ijk}\vee\neg x_{ij\ell})$$

2. Every number appears at least once in each row.

$$\bigwedge_{i=1}^{9}\bigwedge_{k=1}^{9}\bigvee_{j=1}^{9}x_{ijk}$$

3. Every number appears at least once in each column.

$$\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{9}\bigvee_{i=1}^{9}x_{ijk}$$

4. Every number appears at least once in each sub-grid.

$$\bigwedge_{k=1}^{9}\bigwedge_{a=0}^{2}\bigwedge_{b=0}^{2}\bigvee_{u=1}^{3}\bigvee_{v=1}^{3} x_{(3a+u)(3b+v)k}$$

We have implemented the extended encoding in the *sud2sat3* file. The process is identical to the one described above for just the minimal encoding but additionally we cover the encodings above.

4. Performance

A performance analysis was conducted on the minimal, efficient and extended encoding. The same 50 puzzles were used for testing the performance of each encoding. The extended task 1's performance was also analyzed with the hard sudoku input puzzle file which has 95 sudoku puzzles. The results from the performance analyses are detailed below.

4.1 Minimal Encoding Performance

	Average Case Statistic	**Worst Case Statistic**
Number of Variables	729	729
Number of Clauses	3751.02	4563
Parse Time (s)	0.0	0.0
Simplification Time (s)	0.0	0.0
Number of Conflicts	8.52	117
Number of Restarts	1.02	2
Number of Decisions	16.76	155
Number of Propagations	961.32	5316
Memory Used (MB)	11.0	11.0

CPU Time (s)	0.00301564	0.005832

4.2 Efficient Encoding Performance

	Average Case Statistic	**Worst Case Statistic**
Number of Variables	729	729
Number of Clauses	3956.3	5015
Parse Time (s)	0.0	0.0
Simplification Time (s)	0.0	0.0
Number of Conflicts	9.42	107
Number of Restarts	1.04	2
Number of Decisions	17.66	153
Number of Propagations	1038.1	6161
Memory Used (MB)	11.0	11.0
CPU Time (s)	0.00329172	0.005653

4.3 Extended Encoding Performance

	Average Case Statistic	**Worst Case Statistic**
Number of Variables	729	729
Number of Clauses	4015.46	5182
Parse Time (s)	0.0	0.0
Simplification Time (s)	0.0	0.0
Number of Conflicts	0.26	3
Number of Restarts	1.0	1
Number of Decisions	1.5	6
Number of Propagations	741.06	936
Memory Used (MB)	11.0	11.0
CPU Time (s)	0.00317772	0.004633

4.4 Extended Task 1 Performance

	Average Case Statistic	**Worst Case Statistic**
Number of Variables	729	729

Number of Clauses	4762.51579	5285
Parse Time (s)	0.00294737	0.01
Simplification Time (s)	0.000210526	0.01
Number of Conflicts	155.28421	1144
Number of Restarts	1.90526	7
Number of Decisions	275.36842	1630
Number of Propagations	6750.631579	47440
Memory Used (MB)	11.0	11.0
CPU Time (s)	0.01001826	0.028868

5. Results

Above, we calculated the worst case and average case statistics for all three encodings and for the task where we had to solve hard sudoku puzzles. We will now compare the results in terms of the different encodings.

Comparing efficient encoding statistics to those of the minimal encoding, the efficient encoding had a larger set of clauses for both average and worst cases. The average case running time for efficient encoding was larger, but it actually lowered the running time of the *worst* case statistic. Comparing the stats for extended encoding to that of the minimal and efficient encodings, it has an average running time greater than the minimal encoding but less than the efficient encoding. In terms of the worst case statistic running time, it has a significantly lower running time than both the minimal and efficient encoding, as well as a much bigger size than both in terms of number of clauses.

So, the minimal encoding has the fastest average case running time and smallest size for both the average and worst case. The extended coding has the smallest running time out of all for the *worst* case statistic.

Now, in terms of extended task 1, where the solver was tested on hard inputs, we got the largest sizes compared to any of the other encodings. Similarly for the running times, both statistics are significantly larger than all the other encodings.

6. References

[1] "Prop_SAT_Solver", Lecture Slides for CSC 322: Logic and Programming, UVic, Victoria, BC, Canada.