CS360 Fall 2025

Mini Project – SAT-based Sudoku Solving (Deadline December 2nd)

In this project, you will write a simple program to translate partially solved Sudoku puzzles into CNF formulas such that the CNF is satisfiable iff the puzzle that generated it has a solution. You can refer to the paper *Sudoku as a SAT Problem* (`https://sat.inesc-id.pt/~ines/publications/aimath06.pdf`) to get ideas about how to encode puzzles as CNF formulas.

Programming Task

To complete the programming task, you must write code to implement two *programs*

- `sud2sat` reads a Sudoku puzzle (in some specified text format) and converts it to a CNF formula suitable for input to the `miniSAT` SAT solver (described below). You should implement both the minimal encoding and the extended encoding. (See the paper.)
- `sat2sud` reads the output produced by `miniSAT` for a given puzzle instance and converts it back into a solved Sudoku puzzle (as a text file, with newlines for readability.)

You may use any language to implement your translator as long as we can test it as described below.

You may work on it alone, or with someone else (i.e., a group of size at most two).

Background

We will assume that a Sudoku puzzles is encoded as a string of 81 characters each of which is either a digit between `1` or `9` or a "wildcard character" which could be any of `0`, ., $*$ or ? and which indicates an empty entry. Puzzle encodings may have arbitrary whitespace including newlines, for readability. An example puzzle could look like this:

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

Equivalently, this puzzle might be encoded as:

```
163805070008040065005007008450082039301000040700000000839050000604200590000093081
```

The output of your first program should be in the standard SAT-challenge (DIMACS) format, which is standard for most SAT solvers

```
p cnf <# variables> <# clauses>
<list of clauses>
```

Each clause is given by a list of non-zero numbers terminated by a `0`. Each number represents a literal. Positive numbers `1,2,...` are unnegated variables. Negative numbers are negated variables. Comment lines preceded by a `c` are allowed. For example the CNF formula $(x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_4)$ would be given by the following file:

```
c A sample file
p cnf 4 3
1 3 4 0
-1 2 0
-3 -4 0
```

Note that variables are just represented as single numbers. The encoding given in the paper uses variables with three subscripts $x_{i,j,k}$ where $1 \leq i, j, k \leq 9$ (representing the fact that cell (i, j) contains number k.) We need to code each one of these variables as a unique positive integer. A natural way to do this is to think of (i, j, k) as a base-9 number, and converting it to decimal, i.e., $(i, j, k) \rightarrow 81 \times (i-1) + 9 \times (j-1) + (k-1) + 1$ (Note tha this isn't quite converting to decimal. We have to add 1 due to the restriction that variables are encoded as *strictly positive* natural numbers. Also, note we subtract 1 from all of the indices to get them into the range $0, \ldots, 8$, which correspond to the base-9 digits.) Note that for your second program, you are going to have to also define the inverse of the encoding function. I'll leave it up to you to figure out how to do this.

Interfacing with `miniSAT`

Your commands should read their input from `STDIN` and write to `STDOUT`. The following session shows how they should work.

```
$ cat puzzle.txt
...1.5...
14....67.
.8...24..
.63.7..1.
9.......3
.1..9.52.
..72...8.
.26....35
...4.9...
$ ./sud2sat <puzzle.txt >puzzle.cnf
$ minisat puzzle.cnf assign.txt >stat.txt
$ ./sat2sud <assign.txt >solution.txt
$ cat solution.txt
672 145 398
145 983 672
389 762 451
263 574 819
958 621 743
714 398 526
597 236 184
426 817 935
831 459 267
```

Note that after we execute the command

```
$ minisat puzzle.cnf solution.txt >stat.txt
```

the file `assign.txt` will consist of two lines if puzzle.cnf is satisifiable. The first line will just be `SAT`, while the second line will be a sequence of positive and negative variable numbers followed by `0`. A positive variable number indicates the variable is assigned `true` while a negative variable number indicates that the variable is assigned `false`. If the CNF is not satisfiable, there will be just one line – `UNSAT`. The file `stat.txt` will contain various information and statistics about the execution of the command.
You should test your SAT-based Sudoku solver, with respect to both the mininal and extended encodings, on the set of examples provided at

projecteuler.net/project/resources/p096_sudoku.txt

and produce a report which summarizes the results of your test, based on the statistics provided in the file `stat.txt` produced by `miniSAT`. Give average and worst-case statistics. Note that you will need to write a testing harness that interfaces with your commands and with `miniSAT` to do the testing. You do not need to include the code for this testing harness in your submission.
Moreover, you should test your solver on the "hard" inputs provided at

magictour.free.fr/top95

To do this, you will have to modify your solver to handle the input format for these samples.

You should provide a report for these samples similar to that of the above, with respect to both the minimal and the extended encoding. In your report, specify how the minimal-vs-extended encoding choice impacts the problem, e.g., with respect to the size of the encoding, solution time, etc.

A Note About Efficiency

You should note that if you want to make `sud2sat` more efficient, there is no need to re-do the translation from the rules of Sudoku into CNF each time you read a new puzzle. This part of the CNF never changes. You can do it once and hard-wire it into your solution (either in the code, or in a CNF file template) – just take care that you get *the number of lines of CNF correct*, as this may change with different puzzle inputs.

Deliverables and Detailed Grade Breakdown

Your submission should include

1. Your code, with documentation on how to use it. For minimal encodings, your code should produce two *Linux executables*: `sud2sat` reads a *single Sudoku* description from `STDIN` and writes a CNF description to `STDOUT`, and `sat2sud` reads the output produced by `miniSAT` for a single puzzle instance from `STDIN` and writes a formatted version of a solved puzzle on `STDOUT`.

   For extended encodings, produce separate commands, named as follows: `sud2sat1`, `sat2sud1`.

2. A `README` file describing the entire contents of the submission as well as any details you feel are relevant. *Make sure you put the name and Student ID of all group members here.*

3. A report giving background, anything to know about your implementation, and any test results obtained as described above.

Submit everything as a single `.tar.gz` file. The name of the file should be the ID of the group member who submits the file (only one submission per group is required.) The submitted file should extract to a single directory with the same name as the `.tar.gz` file. More specifically, to create the submission, you should be in a directory which contains the directory `subid`, and execute the following:

```
tar cvzf subid.tar.gz subid
```

where `subid` is the ID of the submitter, as described above.)

DO NOT submit any executable of `minisat`

The grader will test your submission as follows: after executing `tar xvzf subid.tar.gz`, the grader should either be able to find the *executables* described above in the top level of directory `subid` or be able to execute `make clean`, followed by `make target` to create them. If you are not able to use `make` or include the excutables, you should give clear instructions on how to build your commands.

You only need to provide one submission for your group.

Task	
Code	7
Report (general)	1
Report (performance evaluation)	1
`README`	1

Table 1: Grade Breakdown