CSC322 Spring 2023

Project – SAT-based Sudoku Solving

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 course notes to get ideas about how to encode puzzles as CNF formulas. Additional encodings are given below.

You should work in groups, preferably of 3 members (but groups of 2 or 4 are OK.) Only one submission per group is required. You may use any language you want for your implementation, but it should be possible for the grader to run and test your code on `linux.csc.uvic.ca`, without installing any new software. The `minisat` system is available on `linux.csc.uvic.ca`.

Basic Task

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

- `sud2sat` reads a **single** Sudoku puzzle (in some specified text format) and converts it to a CNF formula suitable for input to the `miniSAT` SAT solver (described below.) For the basic task, you only need to consider the "minimal" enoding of puzzles as CNF formulas (described in class).
- `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.

Note that the above programs are meant to solve a **single** Sudoku instance. You may want to write additional code in order to do testing and performance evaluation. You should at least document your approach to doing this, but we will not evaluate or run any code you have created for testing and evaluation. The grader will only look at your code for `sud2sat` and `sat2sud`.

Background

We will assume that a Sudoku puzzle 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. Positve 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 class 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 assign.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 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.

Extended Tasks

The following extended tasks are each worth 10% of the grade.

For the first extended task, 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 for the basic task.

The second and third extended tasks require implementations based on encodings other than the minimal encoding presented in class. For each extended task, you should implement and evaluate an additional encoding and in your report consider how it impacts the problem, e.g., with respect to the size of the encoding, solution time, etc.
The starting point for alternate encodings are the following 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 subgrid
$$\bigwedge_{k=1}^{9}\bigwedge_{a=0}^{2}\bigwedge_{b=0}^{2}\bigvee_{u=1}^{3}\bigvee_{v=1}^{3}x_{(3a+u)(3b+v)k}$$

   The encoding given in the lecture slides is the *minimal encoding*. Adding all of 1-4 to the minimal encoding gives the *extended encoding*. Adding just 1 gives the *efficient encoding*.

   For the second extended task, consider the efficient encoding, and for the third extended task the extended encording..

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. 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`. If you do the extended tasks, produce separate commands for each task, named as follows: `sud2sat1`, `sat2sud1` for extended task 1, `sud2sat2`, `sat2sud2` for extended task 2 (efficient encoding), `sud2sat3`, `sat2sud3` for extended task 3 (extended encoding)

2. A `README.md` 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. If you do extended tasks, be sure to describe them here.

Submit everything as a single `.tar.gz` file. The name of the file should be the Brightspace 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 Brightspace ID of the submitter, as described above.)

The grader will test your submission as follows. After executing `tar xvzf subid.tar.gz; cd subid`

- If your executables are non-binary (e.g., shell scripts) the grader should find them at the top level of the directory; otherwise
- The grader should be able to execute `make clean`, followed by `make target` to create the executables. If you are not able to use `make` you should give clear instructions on how to build your commands in the `README`

DO NOT submit any executable of `minisat`. **DO NOT** submit binary executables for your solutions. If the source files need to be compiled, include a `MAKEFILE` as described above.

You only need to provide one submission for your group.

Basic Task	
Code	4
Report (general)	1
Report (performance evaluation)	1
`README`	1
Extended Task	
Task 1	1
Task 2	1
Task 3	1

Table 1: Grade Breakdown

The breakdown for the code grade: (2) for following the specification for two executable files with the required input/output behaviour as described in the **Interfacing** section above. (1) for correctness – this will be determined by testing your commands on some subset of the sample puzzles linked to above. (1) for clarity, style and economy of code.