CSC320 Spring 2017

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 paper *Sudoku as a SAT Problem*, and course notes to get ideas about how to encode puzzles as CNF formulas.

You should work in groups of 4 members. Only one submission per group is required.

Basic Task

(Worth 7/10 of grade.)

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

- `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.) 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 (suitable for printing)

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 experiments. First of all, you should include the solved puzzles (all 50 of them) in your report. Also provide information on the time required to solve each puzzle, as well as average time for all puzzles and any other relevant statistics you choose to include.
You may use any language to implement your translator as long as we can test it on (`Ubuntu 16.04 LTS 64bit`.)

Background

Basically, 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. 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
```

If you decide to produce more than one output format, your translation program should have a command-line option that allows the format to be specified.

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., $(i, j, k) \rightarrow 81 \times (i-1) + 9 \times (j-1) + (k-1) + 1$ (OK, 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.
If you decide to try using `GSAT`, you will have to use a different format. A CNF is just represented as a list with one clause on each line of the file. Literals are represented by numbers, but there is no terminating `0`. Instead, `(` and `)` are used as delimiters. So for the example above we would have.

```
( 1 3 4 )
( -1 2 )
( -3 -4 )
```

Extended Tasks

The basic task is worth 7. Each extended task is worth an extra 1 point. You may complete up to 3 extended tasks.

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.

For the first extended task, you should try at least one alternate to the minimal encoding, and consider how it impacts the problem, e.g., with respect to the size of the encoding, solution time, etc.

For the remaining, do two from the following list:

1. Try at least one alternate to the minimal encoding, and consider how it impacts the problem, e.g., with respect to the size of the encoding, solution time, etc.

2. Comparison to special-purpose Sudoku solvers

3. Use of SAT-solvers other than `miniSAT`

4. Exploring general, $n \times n$-size puzzles

You can find a list of solvers at www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html.

Getting Access to miniSAT

For the grading procedure we'll be using a version of Ubuntu 16.04 LTS 64bit, installed as a virtual machine in VirtualBox. You can create your own virtual machine using the following steps:

1. Download and install the VirtualBox software, which can be found at https://www.virtualbox.org/wiki/Downloads

2. Download and install the VirtualBox Extension Pack, found at the same location

3. Download the Ubuntu installation file "ubuntu-16.04.02-desktop-amd64.iso", found at https://www.ubuntu.com/download/desktop

4. In VirtualBox create a new virtual machine, with the default settings, except for the virtual hard disk size, use 10GB or more, instead of the recommended 8GB.

5. Select the new virtual machine and open Settings→Storage

6. Select the "Empty" cdrom field, under "Controller: IDE". On the right side, click on the cdrom icon and select the Ubuntu iso file.

7. Start the virtual machine and install Ubuntu

8. Add the universe repository, which can be done using this command:

   ```
   sudo add-apt-repository universe
   ```

9. Update repository and install minisat:

   ```
   sudo apt-get update
   sudo apt-get install minisat
   ```

10. A minimal use of minisat would look like this: (first.in is a CNF format)

    ```
    minisat first.in first.out
    ```

Deliverables and Detailed Grade Breakdown

Your submission should include

1. Your code, with documentation on how to use it. Your code should produce *two executables*. The first should be called `sud2sat` and the second should be called `sat2sud`.

2. A report (please submit this as a PDF file) described above for the basic task. If there is anything you would like to add, feel free to do so.

3. 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.

Basic Task	
Code	4
Report	2
`README`	1
Extended Task	
Task 1	1
Task 2	1
Task 3	1

Table 1: Grade Breakdown

Submit everything as a single zipped folder. DO NOT submit any executable of miniSAT. If you are using a Ubuntu virtual machine, DO NOT include the image as part of your submission.

The grader will test your submission as follows: after unzipping your submission, the grader should be able execute `make clean`, followed by `make target`. This should produce the executables from item (1). Which will then be tested on the basic samples. If you are not able to use `make`, you should give clear instructions on how to build your commands.

For each extended task, you should submit your code, as well as a description of your approach in the summary report. Make sure you document the extended tasks in your report, as well as documenting the associated files in your `README`.

You only need to provide one submission for your group.

The amount of programming required here is pretty minimal. This is mainly meant to be fun, and give you some exposure to using a SAT-solver for actual problem solving. However, due to our constraints on grading resources, I would ask you to work in groups of 4. You only need to submit once per group. Make sure you indicate the names of all group members in your `README`.