CSC320 Summer 2015

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 are required to work in groups of 3 if at all possible. Only one submission per group is required.

Basic Task

(Worth 12/15 of grade.)

To complete the basic task, you must write a program that will take a Sudoku puzzle (in some specified text format) and convert 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).

You also should write a program which will convert the output of `miniSAT` back into a solved sudoku puzzle (including prettyprinting.)

You should test your SAT-based Sudoku solver on a good set of example. One place to find sample puzzles is norvig.com/sudoku.html, and produce a report which summarizes the results of your experiments.

You may use any language to implement your translator as long as we can test it on a Linux server in the `csc` domain (`Ubuntu 12.04.5 LTS`.) Please include a `README` file with full instructions on how to run your program or produce an executable (including download instructions for any software that is needed.) You should be able to handle a variety of input encodings. 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

Each extended task is worth 1/15 of the final grade.

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 of the following (worth 1/15 each)

1. Further experimentation with alternate encodings
2. Comparison to special-purpose Sudoku solvers
3. Use of SAT-solvers other than `miniSAT`
4. Exploring general, $n \times n$-size puzzles

For (1), you should at least compare the minimal and extended encodings from *Sudoku as a SAT Problem*. For (2), you should at least consider Peter Norvig's special-purpose solver which can be found at norvig.com/sudoku.html. If you can think of any other approached to encoding, feel free to try them. If you are able to find and experiment with other solvers, feel free to do so.

For extended tasks, you should also include the following: a comparison of performance for

- Different encodings

- Non-SAT approaches
- Different SAT solvers

Use the best means at your disposal for determining performance. Many SAT solvers will include generate performance data.

You can find miniSAT at http://minisat.se/, and you can find a list of solvers at www.cs.ubc.ca/ hoos/SATLIB/index-ubc.html.

Your submission should include your code, with documentation on how to use it, a summary report, and a `README` file describing the entire contents of the submission. Submit everything as a single zipped folder.

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 3. You only need to submit once per group. Just make sure you indicate the names of all group members in your `README`.