Autonomous Systems

H. Geffner
G. Francès

Universitat Pompeu Fabra
2019-2020 Term 2

Exercise Sheet A

Due: Feb. 17, 2020

Some of the files required for this exercise are in the directory `lab2` *of the course repository (*`https://github.com/aig-upf/miis-autonomous-systems-19-20`*). You can always update your clone of the repository with* `git pull` *to see new files.*

In this session, we will work with SAT solvers from a "user" point of view. SAT is the problem of telling whether a given propositional formula ϕ in conjunctive normal form is *satisfiable*, that is, if there is an assignment of truth values to the propositions of the formula that makes ϕ true. Although the SAT problem is NP-complete, current SAT solvers are capable of computing satisfying assignments for propositional formulas with hundreds of thousands of variables and clauses.
Interestingly, the fact that the SAT problem is NP-complete implies that any problem in NP can be polynomially reduced to it, which opens up the door of tackling a large number of challenging combinatorial problems by *transforming* them to a satisfiability problem, and then using a SAT solver to compute a solution. In a way, you can think of this approach as coming up with a propositional formula ϕ_P that "describes" the solutions of your original problem P, in the sense that any assignment that satisfies ϕ_P can be easily mapped into a solution of P.

Exercise A.1 (Sudoku)

Sudoku is a famous puzzle[1] where the objective is to to fill a 9×9 grid with integer digits from $N = \{1, 2, \ldots, 9\}$, so that the resulting grid satisfies two basic constraints:

- Each digit in N appears exactly once in every row and column of the grid.
- Each digit in N appears exactly once in every one of the nine 3×3 subgrids that partition the main grid.

Sudoku boards come with certain pre-filled grid cells, which ensures that there is a single solution, as in Fig. 1. We will represent Sudoku boards with single strings made up of either "." characters or 1-9 digits. A "." denotes an empty cell, and a digit denotes a cell with that value. See the figure for an example of such encoding.
In this exercise, your objective is to model Sudoku as a propositional satisfiability problem, and develop a Sudoku solver based on a SAT solver.

(a) Model the Sudoku problem as a propositional satisfiability problem. More precisely, this means that you need to think and specify what are the propositional variables of your model, and what are the Boolean constraints between these variables that will ensure that any satisfying assignment corresponds to a valid solution to the original Sudoku problem. Remember that your constraints need to be in conjunctive normal form (CNF). Your formalization needs to be valid for any possible initial configuration of the Sudoku board.

For this section you're only asked to submit the formal description of the variables and constraints of your SAT model.

(b) Write a Sudoku solver that makes use of the above compilation to propositional satisfiability. In other words, implement the compilation that you formally described. Your program should be called with one single command-line argument that specifies an initial 9×9 Sudoku board in a single string, and prints a solution in some readable format, e.g.:

[1] `https://en.wikipedia.org/wiki/Sudoku`.

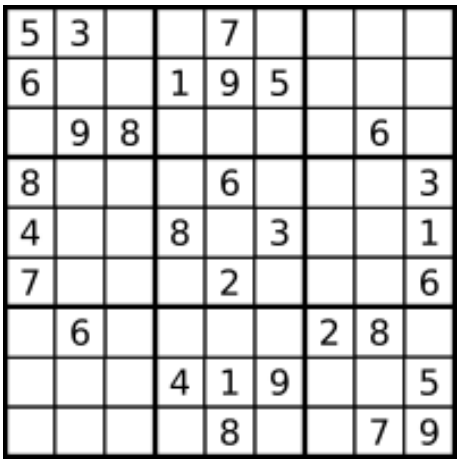

Figure 1: An initial Sudoku configuration (by Tim Stellmach, Public domain). We'll represent such a board with the string "53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8..79" that *reads* the board row by row, top to bottom, for a total of $9 \times 9 = 81$ characters.

```
$./sudoku.py \
.......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........8.6...
Solution: 693784512487512936125963874932651487568247391741398625319475268856129743274836159
Solution in board form:
6 9 3 | 7 8 4 | 5 1 2
4 8 7 | 5 1 2 | 9 3 6
1 2 5 | 9 6 3 | 8 7 4

9 3 2 | 6 5 1 | 4 8 7
5 6 8 | 2 4 7 | 3 9 1
7 4 1 | 3 9 8 | 6 2 5

3 1 9 | 4 7 5 | 2 6 8
8 5 6 | 1 2 9 | 7 4 3
2 7 4 | 8 3 6 | 1 5 9
```

Your program needs to generate a SAT problem that corresponds to the given Sudoku board and feed it to a SAT solver to do the hard work for you. Please see below for some hints on how to get started with the coding.

For this section you need to submit the source code of your program. Remember that code comments usually help in the correction when something doesn't work as expected.

(c) Write another program that *counts* the number of possible solutions (i.e., completions) of a given Sudoku board. Your program should be invoked like:

```
$./sudoku.py -c \
.....54.........8.8.19....3....1.6........34....6817.2.4...6.39......2.53.2.....
Number of solutions: 54
```

For this program, you can use the following technique commonly used to enumerate all satisfying assignments of a SAT problem. Say that $\mathcal{T}$ is a propositional sentence over Boolean variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$. Assume that you have already found a first solution, that is, an assignment σ that maps each variable v_i to a truth value $\sigma(v_i) \in \{True, False\}$. From this assignment, you can create a second SAT problem $\mathcal{T}'$ that will be satisfiable by exactly the same assignments than $\mathcal{T}$, *except for* σ. Think how that problem $\mathcal{T}'$ should look like, and implement the transformation within a loop that will keep looking for solutions that are different to previously-found solutions, until no more such solutions can be found.

Play a bit with your new "solution counter" and some of the instances you can find in the `benchmarks` directory. How many solutions do these instances usually have? What happens if you remove some of the pre-filled digits? How many of them can you remove and still have your solver report the number of solutions under a reasonable amount of time, e.g. one minute?

For this section you need to submit the source code of your program and a brief text answering the questions above.

(d) What is the size of your SAT problem (number of variables, number of CNF clauses)? How does that size depend on the size n of the Sudoku board side (assuming hypothetically that your program could work for arbitrary sizes of the Sudoku board!).

Have a look at the simple `benchmark.py` script that is provided for you, which can be used to run your program against the benchmark sets in directory `benchmarks` (retrieved from the benchmark repository from the Tdoku solver.[2] Run your solver against a large number of instances from any of the two provided benchmarks. What are the runtimes you observe? What conclusions can you draw from them?

For this section you need to submit a brief text answering the questions above.

Getting Started with the Code

Most SAT solvers are able to read input in the simple DIMACS CNF format.[3] The fastest way to proceed for this lab session is:

(a) Install any SAT solver e.g. from some recent competition,[4]

(b) Generate the propositional formula that describes the Sudoku instance you are dealing with, in CNF.

(c) Print your representation of the formula into a file that follows the DIMACS format.

(d) Invoke the SAT solver you installed on the above file.

(e) Parse the output of the SAT solver (an assignment of truth values to propositions printed in some form or another) and convert it back into a solution to the original Sudoku problem.

In the interest of time, we provide you with some code skeleton so that you can focus on the interesting part of the submission, implementing the propositional theory generator. Check the `readme.md` file under directory `lab2` in the course Github repository (`https://github.com/aig-upf/miis-autonomous-systems-19-20/tree/master/lab2`).

The exercise sheets should be submitted in groups of two or three students. Please submit one single copy of the exercises per group (only one member of the group does the submission), and provide all student names on the submission.

[2] See a description of the benchmarks on `https://github.com/t-dillon/tdoku/blob/master/benchmarks/README.md`.

[3] `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`

[4] `http://www.satcompetition.org/`