Institut für Informatik
der Universität München
Dr. J. Johannsen

WS 2026/27

27. Oktober 2026

Practical course
SAT Solving

Problem set 1

Exercises are meant to be solved collaboratively during the tutorial. Projects have to be handed in.

Exercise 1: Setting up CaDiCaL and using the DIMACS CNF format.

1. Download and compile CaDiCaL[1] fromhttps://github.com/arminbiere/cadical.

2. The *DIMACS CNF file format* is the standard format for feeding SAT solvers with formulas. DIMACS CNF files are simple text files looking like this:

   ```
   c comment
   p cnf 3 2
   1 -3 0
   2 3
   -1 0
   ```

   Comments start with `c` and are only allowed at the top. Afterwards, the compulsory header begins with `p cnf` and indicates the number n of variables and m of clauses, respectively. Variables are numbered from 1 to n.

   The following lines specify the m clauses: positive literals are denoted by their corresponding variable number, and negative literals by the additive inverse of that number. Each clause is terminated with a zero, and can potentially extend over several lines.

   Hence the above formula is equivalent to $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$.

3. Let's use CaDiCaL to find a satisfying assignment:

   ```
   > cadical -q simple.cnf
   s SATISFIABLE
   v -1 -2 -3 0
   ```

   CaDiCaL deduced, that setting all variables to false satisfies the formula.

[1] While we recommend using CaDiCaL, you are free to use a different SAT solver and experiment yourself.

Exercise 2: Devise an encoding of standard 9×9 Sudoku puzzles into SAT formulas.

1. Each cell has to contain an integer between 1 and 9, and each row, column and none overlapping 3×3 block (see standard Sudoku rules) has to contain all values exactly once.

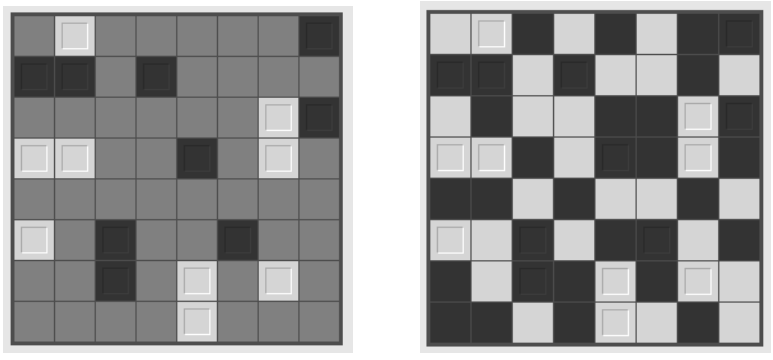
2. Write a script that encodes a given Sudoku instance, e.g.:

```
...546..9
.2......7
..39....4
9.5....7.
7......2.
....93...
.56..8...
.1..39...
......8.6
```

3. Find a solution by invoking a SAT solver and reconstruct the solution from the satisfying assignment.

Project 1: The goal of this project is to build a solver for the puzzle *Unruly*, by encoding it as a SAT problem. The game is played on an $n \times m$ grid. Initially, each cell can either be empty or colored black or white. The empty cells should be colored black or white, such that:

- No three consecutive squares, horizontally or vertically, are the same color.
- Each row and column contains the same number of black and white squares.



(a) Input (b) Solved puzzle

Figure 1: Example puzzle

You can play Unruly online at: https://puzzles-game.com/en/game/unruly/

We will use the following format to represent puzzles as inputs to the program: The first line contains n and m, the number of rows and columns of the puzzle, respectively. Note that due to the rules, n and m must be even. Each of the following n lines contains an m character long string, describing the grid. A period `.` indicates an empty cell, while the character `B` marks a black cell, and `W` marks a white cell. The puzzle in figure 1, (a) corresponds to the following input:

```
8 8
.W.....B
BB.B....
......WB
WW..B.W.
........
W.B..B..
..B.W.W.
....W...
```

This project consists of two parts. The first part is necessary to pass, while the other part will get you a better grade. Keep in mind that quality is more important than quantity, though.

1. Compulsory part: Implement a program that reads puzzles in the above format, converts them into a SAT instance, runs a SAT solver to find a solution, and outputs the result in the same format. A set of sample inputs are attached to the problem set, more can be generated on the website linked above. We will verify your program against different inputs as well and the provided ones are only meant to help you test your program.

2. Bonus part: How well does you encoding scale with increasing grid sizes? Measure the performance and devise a more efficient encoding.

Solutions can be submitted until Friday, 27. November 2026 in groups of 3 students. The programming languages C/C++, Rust, Java, Python and Haskell may be used. Other languages might be permitted as well, just ask beforehand.

Projects should be stored in the institute's GitLab https://gitlab2.cip.ifi.lmu.de/. Upload your project files and add a short README file, explaining how to build and run your program. Also upload the generated DIMACS files for the solved instances of the puzzle to the repository. To submit your project, please fork your repository and share the forked version with the instructor (`jan.johannsen@ifi.lmu.de`).