USI

Theory of Computation

SAT Lab

Practical Homework 2

Sudoku Solver

Andrea Frachini, Davide Bucher, Lorenzo Spoleti, Milo Wroblewski,
Samuele Bischof

Application

The web application is bundled with Electron. The application can be installed on MacOS, but requires python to be installed.

The application can be found here on GitHub.

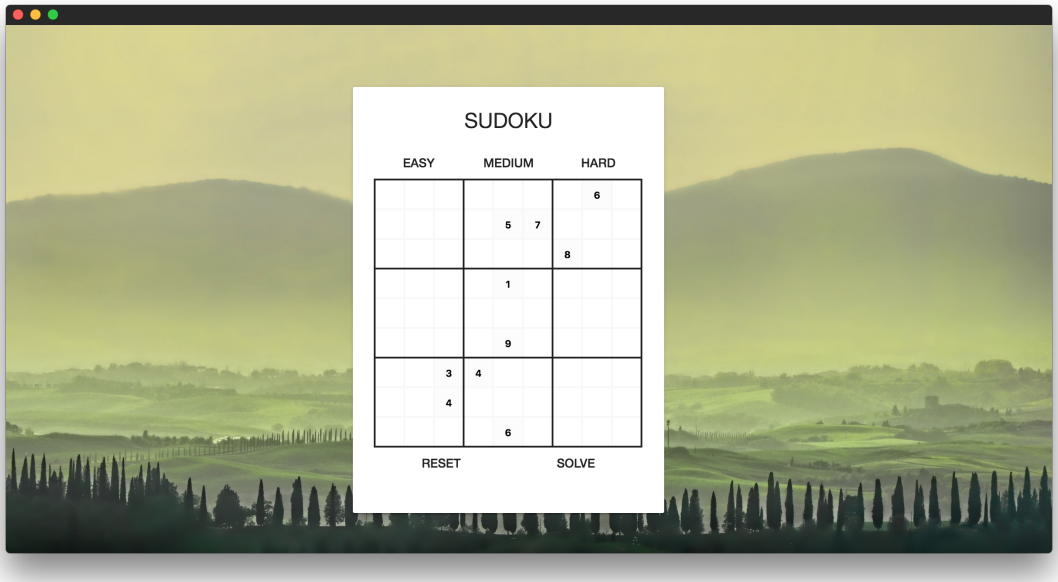

Figure 1: Screenshot of the game

When the application is started, the user can choose the difficulty level, electron then randomly generates a sudoku board. When a board is generated, it is possible that it has no solution. The board is then sent to the sudoku solver to check if it has a solution. Until a board with solution is found, the application randomly tries to generate a new board.

The application runs some simple check on the input, but also if no number is red, it does not mean that the sudoku is correct. To check the correctedness you have to press the solve button. The application will warn the user if the board is not satisfiable.

How does it work

First of all, the electron application generates a file sudoku.in with a total of 81 lines. On every line there will be a value from 1 to 9 or the letter n, which stands for null value.

Then, electron calls the `./sat/sudoku.py` script.

The python script then imports the data. The following image represents the function that reads the sudoku.in file and stores the values in an array of arrays called grid.

```
grid = []
  sudoku = open(sudokuInFile, "r")
  counter = 0
  tempLine = []
  for line in sudoku:
    for word in line:
      if word.isdigit():
        tempLine.append(word)
        if (counter == 8):
          counter = 0
          grid.append(tempLine)
          tempLine = []
        else:
          counter += 1
      if word == 'n':
        tempLine.append(None)
        if (counter == 8):
          counter = 0
          grid.append(tempLine)
          tempLine = []
        else:
          counter += 1
```

Figure 2: The function that imports the grid from the sudoku.in file

When the data is read, the next function that is run is the following:

```
def gen_vars(rows, columns, values):

  varMap = {}

  # Save the 9 * 9 * 9 possible variables
  # every cell can have 9 possibility, there are 9 * 9 cells
  for row in range(0, rows):
    for col in range(0, columns):
    for val in range(1, values + 1):
        var_name=cl(row,col,val)
        varMap[var_name] = gvi(var_name)

  return varMap
```

Figure 3: This function defines the variables. The idea is simple, for every cell there can be a value from 1 to 9. This means we have $9 \times 9 \times 9$ different variables, as we have 9×9 different cells and every single one can contain 9 different values.

After this function, the script runs `genSudokuConstr` This function is divided in four different parts. Each of them adds some clauses.

```
# Add fixed variable
for row in range(0, 9):
  for col in range(0, 9):
    # for every field if there is already a number fix it
    val = grid[row][col]
    if val:
      clauses.append([vars[cl(row,col,val)]])
```

Figure 4: This part of code adds a clause for every value existing in grid. This means that if $\text{cell}_{i,j}$ contains the value 7, this is added as a clause. So the SAT solver knows that in that cell there has to be the value 7.

After this, the SAT solver is executed. The grid is then written back to file and the electon application reads it back in and displays the result if successful and a warning otherwise.

```
# Iterate over every cell
for row in range(0, 9):
  for col in range(0, 9):
    # Every cell has a number from 1 to 9
    clauses.append([vars[cl(row, col, val)] for val in range(1, 10)])
    # Every cell contains only one number
    for valA in range(1, 10):
      for valB in range(valA + 1, 10):
        clauses.append([-vars[cl(row, col, valA)], -vars[cl(row, col, valB)]])
```

Figure 5: This part of code states that in every column and every row there has to be one of the values from 1 to 9. Starting from line 6 in states that in every cell there can only be one possible value.

```
# Check for rows and columns with different values
for i in range(0,9):
  for jA in range(0,9):
    for val in range(1,10):
      for jB in range(0,9):
        if (jA < jB):
          clauses.append([-vars[cl(i,jA,val)], -vars[cl(i,jB,val)]]) # rows
          clauses.append([-vars[cl(jA,i,val)], -vars[cl(jB,i,val)]]) # columns
```

Figure 6: This part of code states that in every column and every row there can be only one value.

Encountered problems

We did not encounter any problem as sudoku has predefined constraints, thus it is straightforward to transform into a SAT problem.

Alternative solutions

For the same reason as above, sudoku does not seem to have alternative solutions.

```
# Check for 3 x 3 squares with different values
for i in 0, 3, 6:
  for j in 0, 3 ,6:
    for iA in 0, 1, 2:
      for jA in 0, 1 ,2:
        for val in range(1,10):
          for iB in 0, 1, 2:
            for jB in 0, 1 ,2:
              if not (iA == iB and jA == jB):
                clauses.append([-vars[cl(i+iA,j+jA,val)], -
vars[cl(i+iB,j+jB,val)]])
```

Figure 7: This part of code states that in 3×3 block there are no repetitions on the same number.

```
# print solution to file
  out = open(sudokuOutFile, "w")
  for i in range(0, len(solution)):
    print solution[i]
    for j in range(0, len(solution[i])):
      if (solution[i][j]):
        out.write(str(solution[i][j]) + "\n")
      else:
        out.write("n\n")
```

Figure 8: Output result to file `sudoku.out`