CS228 Assignment 1 Report

Ajinkya Chandak (24B1003)
Atharva Bhutada (24B0996)

Date: August 24, 2025

1 Problem 1: Sudoku Solver using SAT

1.1 Understanding and Encoding the Problem Statement

We are given a partially filled 9×9 Sudoku grid (in which 0 denotes an empty cell), and we have to produce a completed grid that satisfies Sudoku rules. The solver is implemented by encoding the problem as a CNF formula and solving it using PySAT.

1.2 Approach Overview

We introduce a propositional variable $P(r, c, d)$ meaning "cell at row r, column c contains digit d." We then add CNF clauses to enforce the Sudoku constraints:

- Every cell contains at least one digit.
- No cell contains two different digits simultaneously.
- Every digit appears at most once in each row.
- Every digit appears at most once in each column.
- Every digit appears at most once in each 3×3 subgrid.
- Every digit appears at least once in each row, column and subgrid.
- Clauses for digits initially present in the grid.

1.3 Variable Encoding

We map each triple (r, c, d) with $0 \leq r, c \leq 8$ and $1 \leq d \leq 9$ to a unique integer:

$$P(r, c, d) = r \times 81 + c \times 9 + d.$$

1.4 Constraints Listing

The CNF clauses enforce the Sudoku rules. Each type of clause is written formally and then explained in words.

Cell Constraints

$$P(r, c, 1) \vee P(r, c, 2) \vee \cdots \vee P(r, c, 9)$$

This clause ensures that every cell (r, c) must contain at least one digit between 1 and 9.

Additionally, to prevent a cell from containing two digits simultaneously, for every pair of digits $d_1 \neq d_2$:

$$\neg P(r, c, d_1) \vee \neg P(r, c, d_2).$$

Row Constraints

For every digit d and every pair of distinct columns $c_1 \neq c_2$ in a row r:

$$\neg P(r, c_1, d) \vee \neg P(r, c_2, d).$$

To guarantee that each digit appears at least once in a row:

$$P(r, 0, d) \vee P(r, 1, d) \vee \cdots \vee P(r, 8, d).$$

Column Constraints

For every digit d and every pair of distinct rows $r_1 \neq r_2$ in a column c:

$$\neg P(r_1, c, d) \vee \neg P(r_2, c, d).$$

To enforce presence:

$$P(0, c, d) \vee P(1, c, d) \vee \cdots \vee P(8, c, d).$$

Subgrid Constraints

For each digit d and for each pair of cells (r_1, c_1) and (r_2, c_2) within the same 3×3 subgrid:

$$\neg P(r_1, c_1, d) \vee \neg P(r_2, c_2, d).$$

And for presence: at least one literal per digit inside each subgrid.

Pre-filled Cell Constraints

For every cell (r, c) that is pre-filled in the puzzle with digit d, we add:

$$P(r, c, d).$$

1.5 Solving using PySAT

PySAT is invoked using:

```
with Solver(name="glucose3") as solver:
    solver.append_formula(cnf.clauses)
```

where `cnf` stores all the clauses for solving the problem.

2 Problem 2: Sokoban Solver using SAT

2.1 Understanding the Problem Statement

We are given a Sokoban grid with characters:

- `P` = player `B` = box `G` = goal `#` = wall `.` = empty cell

The goal is to push all boxes onto goal cells within time T (timesteps $t = 0, \ldots, T$). A push is valid only if the player moves to the position of the box and the square beyond the box (in the same direction) is free (i.e., inside the grid and not a wall or another box).

2.2 Grid and State Representation

The Sokoban grid has N rows and M columns. We divide the cells into two types:

$\texttt{cellswithoutwalls} = \{(i, j) \mid \text{cell is not a wall}\}, \qquad \texttt{cellswithwalls} = \{(i, j) \mid \text{cell is a wall}\}.$

From the input we also record: the player's starting position (x_P, y_P), the list of box positions $\{(x_b, y_b)\}$, and the set of goals $\mathcal{G}$.

2.3 Variable Encoding

Player variable.

$$\texttt{var\_player}(x, y, t) = t \cdot (NM) + x \cdot M + y + 1$$

This encodes the integer variable id for the player at (x, y) and time t. To represent the player, we have to have (T+1)NM variables because total timesteps are T+1 and at a given time, grid positions possible are NM in number. Now x.M + y is for the grid position of the player. t.(NM) is for the timesteps and adding 1 ensures that the variable cannot be zero.
Let us take an example M=4 N=3 and T=5. Now we have to represent 6 timesteps 0,1,2,3,4,5 and 12 grid positions at each timestep. This given encoding encodes such that 1 to 12 are for grid positions at time 0, 13 to 24 are for grid positions at time 1 and so on. Total 72 integers that is 1 to 72 are required for the player. Hence, we can conclude the uniqueness of the encoding scheme.

Box variable.

$$\texttt{var\_box}(b, x, y, t) = (1 + b) \cdot ((T + 1)NM) + t \cdot (NM) + x \cdot M + y + 1$$

This represents the clause that box b is at (i, j) at time t. Let suppose we have B number of boxes. For each box, we will need the same number of variables as the player but since the variables cannot coincide we have to shift the indexing. So, what we do is for the first box, we use 73 to 144, for the second box we use 145 to 216 and so on. Now as the boxes b are numbered from 0 to B-1, the term which we have to add for shifting is (1 + b) · ((T + 1)NM).

2.4 CNF Constraints (aligned with the encoder code)

1. Initial conditions (time $t = 0$)

- **Player start:** the player is fixed to its given start cell. In code:

$$\texttt{self.cnf.append([var\_player}(x_P, y_P, 0)\texttt{])}$$

- **Boxes start:** every box is fixed to its start cell. For each box b:

$$\texttt{self.cnf.append([var\_box}(b, x_b, y_b, 0)\texttt{])}$$

- **Exactly-one player position at $t = 0$:** the player must occupy one of the valid cells in `cellswithoutwalls` and not two at once. The encoder adds the at-least-one clause and pairwise exclusion clauses:

$$\bigvee_{(i,j)\in\texttt{cellswithoutwalls}} \texttt{var\_player}(i, j, 0),$$

$$\neg\texttt{var\_player}(i_a, j_a, 0) \vee \neg\texttt{var\_player}(i_b, j_b, 0) \quad \text{for every distinct } (i_a, j_a), (i_b, j_b).$$

- **Exactly-one position per box at $t = 0$:** for each box b the encoder adds

$$\bigvee_{(i,j)\in\texttt{cellswithoutwalls}} \texttt{var\_box}(b, i, j, 0)$$

and pairwise exclusion clauses to prevent one box from being in two places at once.

- **No occupancy on walls at $t = 0$:** for every $(i, j) \in$ `cellswithwalls`:

$$\neg\texttt{var\_player}(i, j, 0), \qquad \neg\texttt{var\_box}(b, i, j, 0).$$

2. Overlap constraints (for every timestep)

1. **Player exactly-one:** at each t the encoder enforces the player is in exactly one valid cell:

$$\bigvee_{(i,j)\in\texttt{cellswithoutwalls}} \texttt{var\_player}(i, j, t),$$

and pairwise exclusion clauses $\neg\texttt{var\_player}(\cdot) \vee \neg\texttt{var\_player}(\cdot)$.

2. **Box exactly-one (per box):** each box b must be at exactly one valid cell at time t (same pattern as the player).

3. **No occupancy on walls at time t:**

$$\neg\texttt{var\_player}(i, j, t), \qquad \neg\texttt{var\_box}(b, i, j, t) \quad \text{for } (i, j) \in \texttt{cellswithwalls}.$$

4. **No player–box overlap:** for every valid cell and box:

$$\neg\texttt{var\_player}(i, j, t) \ \vee \ \neg\texttt{var\_box}(b, i, j, t).$$

5. **No two boxes share a cell:**

$$\neg\texttt{var\_box}(b_1, i, j, t) \ \vee \ \neg\texttt{var\_box}(b_2, i, j, t) \quad \text{for } b_1 \neq b_2.$$

3. Player movement

- **Reachable next positions:** if the player is at (i, j) at time t, then at time $t + 1$ the player must be in one of the neighbouring valid cells (the code uses `DIRS` = {U,D,L,R}):

$$\neg\texttt{var\_player}(i, j, t) \ \vee \bigvee_{(i',j')\in\mathsf{Adj}_{\texttt{cellswithoutwalls}}(i,j)} \texttt{var\_player}(i', j', t + 1).$$

This is added for every $(i, j) \in$ `cellswithoutwalls` and every $t \in \{0, \ldots, T - 1\}$.

- In the code this is implemented exactly as:

```
self.cnf.append([-var_player(i,j,t)] + validnextpos).
```

4. Box push constraints

- **Valid push:** if the player moves from (i, j) at time t to $(i + dx, j + dy)$ at time $t + 1$, and box b occupies $(i + dx, j + dy)$ at time t, and the push destination $(i + 2dx, j + 2dy)$ is a valid cell, then box b must be at $(i + 2dx, j + 2dy)$ at $t + 1$. The encoder adds one combined clause per such configuration:

$$[-\texttt{var\_player}(i, j, t),\ -\texttt{var\_player}(i + dx, j + dy, t + 1),\ -\texttt{var\_box}(b, i + dx, j + dy, t),\ \ldots,\ \texttt{var\_box}(b, i + 2dx, j + 2dy, t + 1)]$$

  where the "..." denotes the negative literals forbidding other boxes at the push destination (`nototherboxesatpushdest` in the code).

- **Restricted push when destination blocked/wall:** if the destination $(i + 2dx, j + 2dy)$ is not valid, the encoder restricts the player move into $(i + dx, j + dy)$ when box b is present:

$$\neg\texttt{var\_player}(i, j, t)\ \vee\ \neg\texttt{var\_player}(i + dx, j + dy, t + 1)\ \vee\ \neg\texttt{var\_box}(b, i + dx, j + dy, t).$$

- **Box stays or moves to adjacent cell:** for each box b and cell $(i, j) \in$ `cellswithoutwalls` the encoder appends:

$$\neg\texttt{var\_box}(b, i, j, t)\ \vee\ \texttt{var\_box}(b, i, j, t + 1)\ \vee \bigvee_{(i',j')\in\mathsf{Adj}(i,j)} \texttt{var\_box}(b, i', j', t + 1).$$

- **Box movement requires a push:** if a box moves from (i, j) at time t to $(i + dx, j + dy)$ at $t + 1$, then the encoder requires the player at $(i - dx, j - dy)$ at time t and at (i, j) at time $t + 1$. These are encoded as:

$$\neg\texttt{var\_box}(b, i, j, t)\ \vee\ \neg\texttt{var\_box}(b, i + dx, j + dy, t + 1)\ \vee\ \texttt{var\_player}(i - dx, j - dy, t),$$

  and

$$\neg\texttt{var\_box}(b, i, j, t)\ \vee\ \neg\texttt{var\_box}(b, i + dx, j + dy, t + 1)\ \vee\ \texttt{var\_player}(i, j, t + 1).$$

  If the pushing position $(i - dx, j - dy)$ does not exist (wall/outside), the move cannot happen:

$$\neg\texttt{var\_box}(b, i, j, t)\ \vee\ \neg\texttt{var\_box}(b, i + dx, j + dy, t + 1).$$

6. Goal conditions (final time $t = T$)

For every box b, the encoder appends that the box must be at some goal at time T:

$$\bigvee_{(g_i,g_j)\in\texttt{goals}} \texttt{var\_box}(b, g_i, g_j, T).$$

2.5 Decoding the Model

The solver outputs a propositional assignment `model`, represented as a list of signed integers (positive = variable true, negative = variable false). Our function `decode(model, encoder)` interprets this assignment to reconstruct the player's path.

- The encoder ensures that at each timestep exactly one `var_player(i,j,t)` is true. Thus, for every $t = 0, \ldots, T$, the decoder scans through `cellswithoutwalls` (the list of valid non-wall positions) and checks which ID `encoder.var_player(i,j,t)` appears positively in `model`. The matching (i, j) is recorded in `playerpositions`.

- After this loop, we obtain a sequence

$$\texttt{playerpositions} = [(i_0, j_0), (i_1, j_1), \ldots, (i_T, j_T)],$$

  representing the player's location at each step.

- To convert these positions into moves, the decoder computes

$$(\Delta i, \Delta j) = (i_{t+1} - i_t,\ j_{t+1} - j_t),$$

  and maps it using the direction dictionary `DIRS`: $(-1, 0) \mapsto U$, $(1, 0) \mapsto D$, $(0, -1) \mapsto L$, $(0, 1) \mapsto R$. Each symbol is appended to the list `moves`.

If no satisfying assignment exists, the function returns `-1`. Otherwise, the result is a list of move characters, directly extracted from the sequence of `var_player` literals found in `model`.

NOTE: Our code is correct because it will always find out that whether the boxes can be pushed into goals in the given time T. The decoder always finds a sequence of exactly T moves which push the boxes into the goals if possible. But, if we give a greater time T than the optimal time in which it can be done, it will still generate the sequence of T moves which is correct. For getting the optimal sequence, you have to input the optimal time. So, it will always be correct in terms of whether the task can be done or not but it may not always generate the optimal sequence of moves.
NOTE: We have put self-explanatory comments in our code. Please go through them for better understanding of our implementations.

2.6 Solving with PySAT

PySAT is invoked using:

```
with Solver(name='g3') as solver:
    solver.append_formula(cnf)
```

3 Contributions

Although most of the part is done collaboratively sitting together, the distribution of work to be more precise was similar to:
Atharva: Major contribution in Problem 2 implementation and encoding details.
Ajinkya: Major contribution in Problem 1 implementation and report writing.