CS228 : Assignment -1

Kona Pavan Sai Subhash
Roll No. 24B0955

Shaik Suraj
Roll No. 24B0907

August 24, 2025

Contents

1 SAT-Based Sudoku Solver **2**
- 1.1 Approach Overview 2
- 1.2 Variable Encoding 3
- 1.3 Individual Contribution 3

2 Grading Assignments Gone Wrong **4**
- 2.1 Understanding the Problem 4
- 2.2 Variable Encoding 4
  - 2.2.1 Player Encoding 4
  - 2.2.2 Boxes Encoding 4
- 2.3 Approach Overview 5
- 2.4 Decoding the model 6
- 2.5 Individual Contribution 7

Chapter 1

SAT-Based Sudoku Solver

1.1 Approach Overview

- We encoded the Sudoku using propositional logic and then found its solution using the PySAT library in Python.
- Our approach is similar to the one discussed in class. We created a propositional variable $\mathbf{P(i, j, n)}$ which is true when the number n is at the position (i, j).
- We divided the CNF encoding into 5 parts:

  1. **Set the initial conditions**
     - In the Sudoku grid, the number given initially in any position should always be the same. Therefore, we added that propositional variable as a CNF clause to ensure it is true every time.
     - $\mathbf{P(i, j, n)}$ is added for every initial (i, j) where a number n is given.

  2. **Every row has all the numbers**
     - We made a clause that is true when each number is present in at least one position of every row.
     - It works like this: number n is in the 1st position of row x or the 2nd position of row x or ...

  3. **Similarly, every column has all the numbers**
     - Similar to the row

  4. **All 3x3 squares given has all the numbers**
     - There are nine 3×3 squares in the classic Sudoku, which also should contain every number.
     - We use the same logic as we did for the row. In a row or column, it was a straight 9 positions with 1 increment in either i or j, but in the case of a square, it's not a direct increment. We need to be careful in that.

  5. **Every position has exactly one number**
     - This condition should be implemented because the Solver may produce solutions with multiple numbers in a position.
     - This constraint is encoded as follows :
       * If a number n is present at a position (i, j), i.e., $\mathbf{P(i, j, n)}$ is true, then all other numbers shouldn't be present at that position, i.e., $\forall n' \neq n,\ \mathbf{P(i, j, n')}$ is false.

1.2 Variable Encoding

- The variable $\mathbf{P(i, j, n)}$ is encoded as $100i + 10j + n$, which is a unique integer for all $\forall i, j, n \in \{1, \ldots, 9\}$.

1.3 Individual Contribution

- Each of us solved this question separately, and both of us used almost the same variable and CNF encoding with only minor differences.

Chapter 2

Grading Assignments Gone Wrong

2.1 Understanding the Problem

The given problem is a variation of the classical Sokoban puzzle. In this context:

- Boxes represent stacks of grading assignments,
- Walls correspond to the obstructions placed by Yuvaraj, and
- Goals denote the submission desks.

Thus, the task effectively reduces to implementing a SAT-based Sokoban solver.

2.2 Variable Encoding

2.2.1 Player Encoding

The propositional variable of the player at position (i, j) in the grid and at time t is represented as:

$$P(i, j, t) = 17 \cdot 17 \cdot 17 \cdot t \; + \; 17 \cdot (i + 1) \; + \; (j + 1).$$

2.2.2 Boxes Encoding

The propositional variable of a box b at position (i, j) in the grid and at time t is represented as:

$$B(b, i, j, t) = 17 \cdot 17 \cdot 17 \cdot t \; + \; 17 \cdot 17 \cdot b \; + \; 17 \cdot (i + 1) \; + \; (j + 1).$$

We proved that no two variables can overlap with the given constraints.

The constraints are defined as $N, M \leq 10$, $T \leq 10$, and $B \leq 5$.

Note : Position (i, j) means $(i + 1)^{th}$ row and $(j + 1)^{th}$ column.

2.3 Approach Overview

- We encoded the Sokoban Puzzle using propositional logic and then found its solution using the PySAT library in Python.
- To simplify the encoding, we added two borders of walls around the grid and accordingly update the number of rows and columns
  For example, if the initial grid is given as:

$$\begin{matrix} P & . & . \\ . & B & . \\ \# & . & G \end{matrix}$$

  then the transformed grid with the extra border becomes:

$$\begin{matrix} \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# \\ \# & \# & P & . & . & \# & \# \\ \# & \# & . & B & . & \# & \# \\ \# & \# & \# & . & G & \# & \# \\ \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# \end{matrix}$$

- The following constraints are used to get the correct CNF formula:

  1. **Initial Conditions**
     The initial player position (i_1, j_1) is determined by traversing the grid. This is encoded as $P(i_1, j_1, 0)$ being true. Similarly, the initial positions of the boxes are obtained by traversing the grid. Each box is assigned a unique identifier, which makes it possible to track individual boxes across different time steps.
  2. **Goal Conditions**
     A box is considered correctly placed if it occupies at least one goal position. Since each box has a unique position at any time, it can only be on a single goal. To prevent multiple boxes from occupying the same goal, we additionally enforce that no two boxes can share the same position.
  3. **Wall Conditions**
     Neither the player nor any box may occupy a wall position. For every cell (i, j) containing a wall symbol '#', and for all times t, we encode

$$\neg P(i,j,t), \quad \neg B(b,i,j,t) \;\; \forall b.$$

  4. **Player Movement**
     The following restrictions are imposed on the player's movement:
     - At time t, if the player is at (i, j), then at time $t+1$ the player may be at one of

$$(i,j),\ (i+1,j),\ (i-1,j),\ (i,j+1),\ (i,j-1).$$

     - The player cannot move into a wall position. This is already enforced by the wall constraints.
     - The player cannot push a box into an invalid position. Specifically, if the player attempts to move into a box, then the box must be pushed into the adjacent cell in the same direction. This move is only valid if the target cell is not occupied by another box or a wall. Otherwise, it would create a contradiction with the previously defined constraints.
  5. **Box Movement**
     The movement of a box b from time t to $t+1$ can be described as follows:
     - At time $t+1$, if a box is located at position (i, j), then either:
       (a) The box was already at (i, j) at time t, or

(b) A valid push condition at time t caused the box to move into (i,j).

– Formally, for all b,i,j,t:

$$B(b,i,j,t{+}1) \;\Rightarrow\; B(b,i,j,t) \vee \Phi_{\mathrm{R}}(b,i,j,t) \vee \Phi_{\mathrm{L}}(b,i,j,t) \vee \Phi_{\mathrm{U}}(b,i,j,t) \vee \Phi_{\mathrm{D}}(b,i,j,t)$$

– The push conditions are defined as:

$$\Phi_{\mathrm{R}}(b,i,j,t) := B(b,i,j+1,t) \;\wedge\; P(i,j+2,t) \;\wedge\; P(i,j+1,t+1) \qquad \text{(pushed from the right)}$$
$$\Phi_{\mathrm{L}}(b,i,j,t) := B(b,i,j-1,t) \;\wedge\; P(i,j-2,t) \;\wedge\; P(i,j-1,t+1) \qquad \text{(pushed from the left)}$$
$$\Phi_{\mathrm{U}}(b,i,j,t) := B(b,i+1,j,t) \;\wedge\; P(i+2,j,t) \;\wedge\; P(i+1,j,t+1) \qquad \text{(pushed from below)}$$
$$\Phi_{\mathrm{D}}(b,i,j,t) := B(b,i-1,j,t) \;\wedge\; P(i-2,j,t) \;\wedge\; P(i-1,j,t+1) \qquad \text{(pushed from above).}$$

6. **Player Uniqueness**
At every time step t, the player must occupy exactly one position.
Formally, for all $(i,j) \neq (x,y)$,

$$\neg\big(P(i,j,t) \wedge P(x,y,t)\big).$$

7. **Box Uniqueness**
At every time step t, each box b must occupy exactly one position.
Formally, for all $(i,j) \neq (x,y)$,

$$\neg\big(B(b,i,j,t) \wedge B(b,x,y,t)\big).$$

8. **Collision Avoidance**
No two boxes may occupy the same position at any time. Likewise, the player and a box cannot occupy the same position simultaneously.
Formally, for all boxes b_1, b_2 with $b_1 \neq b_2$, and for all positions (i,j) at time t:

$$\neg\big(B(b_1,i,j,t) \wedge B(b_2,i,j,t)\big)$$

Additionally, for all boxes b and positions (i,j) at time t:

$$\neg\big(P(i,j,t) \wedge B(b,i,j,t)\big)$$

2.4 Decoding the model

Since we extended the grid by adding additional rows and columns, we must also update the decoding process accordingly.
The decoding procedure is as follows:

1. **Identify the Initial Player Position**
Traverse the grid and determine the initial position (i_1, j_1) of the player by checking whether $P(i,j,0)$ is satisfied in the model.

2. **Trace Player Movement**
Starting from (i_1, j_1) at time $t = 0$, check which of the neighboring positions

$$(i_1+1, j_1), \quad (i_1-1, j_1), \quad (i_1, j_1+1), \quad (i_1, j_1-1)$$

is true at time $t = 1$ in the model.

3. **Update Position and Record Direction**
If a neighboring position (i', j') is satisfied at time $t+1$, update the player position as $(i_1, j_1) \leftarrow (i', j')$. Append the corresponding movement direction (up, down, left, or right) to the move sequence.

4. **Iterate Until Termination**
Increment $t \leftarrow t+1$ and repeat the above step until t reaches the given maximum time T.

2.5 Individual Contribution

- Pavan proposed the idea of adding two layers of boundary walls around the grid in order to eliminate edge cases.
- The encoding of the initial conditions, goal conditions, wall conditions, and player movements was relatively straightforward and was implemented collaboratively by both of us.
- For the box movement rules, we initially developed multiple approaches, many of which contained flaws. Combining our individual ideas, we arrived at a correct formulation.
- We experimented with different strategies for variable encoding before finalizing the most efficient one:
  - Pavan encoded the variables as integers, assigning two positions in the integer to each of the i, j, t, b.
  - Suraj encoded the variables as $p \cdot p \cdot p \cdot t + p \cdot p \cdot b + p \cdot i + j$, where p is a prime number. This scheme treats the variable as a player if $b = 0$, and as a box otherwise. Compared to Pavan's encoding, this approach significantly reduced the size of the integers.
  - We finalized Suraj's encoding scheme, fixing the value of p to 17 in accordance with the given constraints.
- The constraints for **Player Uniqueness**, **Box Uniqueness**, and **Collision Avoidance** were implemented by Pavan and thoroughly reviewed by both of us before execution.
- The **Decoder function** was implemented by Suraj and similarly verified by both of us prior to running the code.