Assignment 1 Report

Jovita Bhasin (24B0939)
Rohan Jadhav (24B1012)

23 August 2025

Question 1. SAT Based Sudoku Solver

Approach Overview

Our approach was to represent the rules of Sudoku as a Boolean SAT problem and then solve it using a SAT solver. Each rule of Sudoku was translated into a set of SAT clauses, and the conjunction of all these clauses formed the final CNF formula. We designed the solution to work for a general $n \times n$ Sudoku (with $n \leq 9$). For this, we used Python's `math.sqrt` function to compute $\sqrt{n}$ which was required for the subgrid constraint. **The problem was solved collaboratively by both of us.**

To convert the Sudoku puzzle into an SAT problem, we first define our variable $x(i, j, k)$ to represent the proposition that cell in column i and row j contains the number k where $i, j, k \epsilon \{1, 2, \ldots, n\}$. A value of true for this variable means that the cell (i, j) contains the number k and false means that it does not. This three-dimensional variable was mapped to a unique positive integer for the SAT solver using the formula $100 \times i + 10 \times j + k$.

Next we translated the fundamental rules of Sudoku in Conjunctive Normal Form (CNF) as follows:

Variable Encoding

1. **Cell Constraints:**

   - **At most one number in each cell:** For two distinct numbers $k1$ and $k2$, the clause $(\neg x_{i,j,k_1} \vee \neg x_{i,j,k_2})$ ensures that the cell cannot hold two different numbers simultaneously.
   - **At least one number in each cell:** For each cell (i, j), the clause $(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,n})$ means that the cell is not empty. We did not encode this explicitly because this is directly implied by our subgrid, row and column constraints.

   Here is the code snippet that ensures the at most one number in each cell condition

```
# each cell can contain only one number
for row in range(1,n+1):
    for column in range(1,n+1):
        for num1 in range(1,n+1):
            for num2 in range(num1+1,n+1):
                cnf1.append([-(100*row+10*column+num1),
                    -(100*row+10*column+num2)])
```

2. **Row, Column and Subgrid Constraints:** We must ensure that for each region and number k, the region contains at least one instance of k. For example, for each row i and number k, we add the clause $(x_{i,1,k} \vee x_{1,2,k} \vee \cdots \vee x_{i,n,k})$

   Here are the code snippets that ensure these conditions for rows, columns and subgrids respectively:

```
    # each row must contain all numbers
    for row in range(1,n+1):
        for num in range(1,n+1):
            cnf1.append([100*row+10*column+num for
                column in range(1,n+1)])
```

```
    # each column must contain all numbers
    for column in range(1,n+1):
        for num in range(1,n+1):
            cnf1.append([100*row+10*column+num for row
                in range(1,n+1)])
```

```
    # each 3x3 subgrid must contain all numbers
    for rownums in range(1,n+1,sqrt_n):
        for colnums in range(1,n+1,sqrt_n):
            for num in range(1,n+1):
                cnf1.append([100*(rownums+dr)+10*(
                    colnums+dc)+num for dr in range(
                    sqrt_n) for dc in range(sqrt_n)])
```

3. **Initial Conditions:** The pre filled cells were encoded as unit clauses, this means that for each given cell (i, j) with a number $k \neq 0$, we added the clause $(x_{i,j,k})$, which forces this variable to be True in any satisfying assignment.

```
    # initial conditions
    for row in range(0,n):
        for column in range(0,n):
            if grid[row][column] != 0:
                cnf1.append([100*(row+1)+10*(column+1)+
                    grid[row][column]])
```

Question 2. Grading Assignments Gone Wrong

In the `__init__` function, a list `self.walls` is created to store the coordinates of the walls. In the `_parse_grid` function, the initial ($t = 0$) positions of the goals, boxes, and player are extracted from the input grid and stored in `self.goals`, `self.boxes`, and `self.player_start`, respectively. The wall positions are also stored in `self.walls` during this stage.

Implementation (By Jovita):

```
    def _parse_grid(self):
        """Parse grid to find player, boxes, and goals."""
        # TODO: Implement parsing logic
        for i in range(self.N):
            for j in range(self.M):
                if self.grid[i][j]=="G":
                    self.goals.append((i, j))
                elif self.grid[i][j]=="B":
                    self.boxes.append((i, j))
                elif self.grid[i][j]=="P":
                    self.player_start = (i, j)
                elif self.grid[i][j]=="#":
                    self.walls.append((i, j))
        return
```

Variable Encoding

For the variable encoding, we defined unique propositional variables to represent the positions of the player and the boxes at any given time step.

Player Encoding

The player's position at coordinates (x, y) at time t (with $0 \leq x, y \leq 9$ and $0 \leq t \leq 10$) was encoded as:

$$\text{var\_player}(x, y, t) = 1 + 1000 \cdot x + 100 \cdot y + t$$

The offset of +1 ensures that the SAT solver only receives positive variable IDs (since 0 is not allowed). A multiplier of 1000 was chosen for x instead of 100, because t can take the value 10, and using only three digits for encoding could have caused collisions.

Box Encoding

For boxes, where $0 \leq b \leq 5$ represents the box identifier and $0 \leq x, y \leq 9$, $0 \leq t \leq 10$, the position was encoded as:

$$\text{var\_box}(b, x, y, t) = 10000 \cdot (b + 1) + 1000 \cdot x + 100 \cdot y + t$$

Here, $(b + 1)$ was used instead of b so that even when $b = 0$, the encoding yields a 5-digit number, thus avoiding overlap with the player's encoding scheme. The extra spacing in the 10's place was introduced for the same reason as in the player encoding: to prevent collisions when $t = 10$.

Summary

This scheme guarantees that:

- Player positions are encoded as distinct 4-digit numbers.
- Box positions are encoded as distinct 5-digit numbers.
- There are no collisions within the player encoding, within the box encoding, or between the two.

The variable encoding was ideated and implemented by Jovita.

The Encode Function

The `encode` function is responsible for generating the CNF clauses that represent the Sokoban problem as a SAT instance.
As a first step, we construct the list `valid_positions`, which stores all coordinates that are not occupied by walls. These positions represent the cells on which the player and the boxes are allowed to move. **This step was done by Rohan.**
Code snippet:

```
    # storing the valid positions (coordinates where wall
        is not there)
    valid_positions = []
    for i in range(self.N):
        for j in range(self.M):
            if (i, j) not in self.walls:
                valid_positions.append((i, j))
```

Initial Conditions

The initial positions are encoded into the CNF to capture the starting configuration. Specifically, the player's position at $t = 0$ is appended to `self.cnf` using the value stored in `self.player_start`, and the initial positions of all boxes are appended to `self.cnf` using the values stored in `self.boxes`. **This was done by Jovita.**
Code snippet:

```
    # initial conditions
    # initial position of player
    self.cnf.append([self.var_player(self.player_start[0],
        self.player_start[1], 0)])

    # initial positions of boxes
    for b in range(0,self.num_boxes):
        self.cnf.append([self.var_box(b, self.boxes[b][0],
            self.boxes[b][1], 0)])
```

Player position and movement constraints

At all times, the player's position must satisfy two important constraints:

1. **Existence constraint:** The player must be located at *some* valid position at each time step. **This was done by Jovita.**
2. **Uniqueness constraint:** The player can only occupy *one* valid position at each time step. **This was done by Rohan.**

Existence constraint: The player must always occupy *some* valid position at each time step. The following code enforces that for each t, at least one propositional variable corresponding to a valid position is true:

```
# coordinates of player lie within valid positions
for t in range(0, self.T+1):
    self.cnf.append([self.var_player(x, y, t) for (x, y) in
        valid_positions])
```

Uniqueness constraint: The clause

$$\neg P_{p_1,t} \vee \neg P_{p_2,t}$$

ensures that the player cannot simultaneously occupy both positions p_1 and p_2 at time t.

Since this clause is generated for every distinct pair of valid positions, the overall effect is that **at most one player position variable can be true at each time step**. This encodes the *uniqueness constraint* for the player.

```
# at a given time, player is at at most one cell
for t in range(self.T+1):
    for i in range(len(valid_positions)):
        for j in range(i+1, len(valid_positions)):
            p1 = valid_positions[i]
            p2 = valid_positions[j]

            self.cnf.append([
                -self.var_player(p1[0], p1[1], t),
                -self.var_player(p2[0], p2[1], t)
            ])
```

Movement constraint. Finally, the player is allowed to *either stay in place or move to one of the four adjacent valid positions* at the next time step. Formally, for every $(x, y) \in$ `valid_positions` and each $t < T$:

$$P_{x,y,t} \;\rightarrow\; \big(P_{x,y,t+1} \;\vee \bigvee_{(dx,dy)\in \texttt{DIRS}} P_{x+dx,y+dy,t+1}\big)$$

Implementation (By Rohan):

```
# if player is at some position at t then he can be at the
    same position
# or any adjacent valid position at time t+1
for t in range(self.T):
    for x, y in valid_positions:
        possible_next_positions = [
            -self.var_player(x, y, t),
            self.var_player(x, y, t+1)
        ]

        for dx, dy in DIRS.values():
            npx, npy = x + dx, y + dy
            if (npx, npy) in valid_positions:
                possible_next_positions.append(
                    self.var_player(npx, npy, t+1)
                )

        self.cnf.append(possible_next_positions)
```

Box position constraints

At all times, each box's position must satisfy two important constraints:

1. **Existence constraint:** Every box must be located at *some* valid position at each time step.

2. **Uniqueness constraint:** A box can only occupy *one* valid position at each time step.

This encoding is similar to that used for the player. **This part was done by Jovita.**

Existence constraint: The following code ensures that, for each time step t and for each box b, at least one of the propositional variables corresponding to valid positions is true:

```
# box coordinates lie within valid positions
for t in range(0,self.T+1):
    for b in range(0, self.num_boxes):
        self.cnf.append([self.var_box(b, x, y, t) for x, y
            in valid_positions])
```

Uniqueness constraint: The clause

$$\neg B_{b,p_1,t} \vee \neg B_{b,p_2,t}$$

ensures that box b cannot simultaneously occupy both positions p_1 and p_2 at time t. Since this clause is generated for every distinct pair of valid positions, the overall effect is that each box can be in *at most one* position at each time step. This correctly enforces the uniqueness constraint.

```
# at a given time, box is at at most one position
for t in range(self.T+1):
    for b in range (self.num_boxes):
        for i in range(len(valid_positions)):
            for j in range(i+1, len(valid_positions)):
                p1 = valid_positions[i]
                p2 = valid_positions[j]

                self.cnf.append([-self.var_box(b, p1[0], p1
                    [1], t),
                                 -self.var_box(b, p2[0], p2
                                     [1], t)])
```

Overlapping constraints

To ensure the validity of the game state at all times, overlapping of entities is forbidden. Specifically:

1. The player and any box cannot occupy the same cell at the same time step.

2. Two different boxes cannot occupy the same cell at the same time step.

This was implemented by both of us.

Player-Box Overlap: For every valid position (x, y) and time t, the clause

$$\neg P_{x,y,t} \vee \neg B_{b,x,y,t}$$

is added for each box b, preventing the player and the box from simultaneously being at (x, y).

```
# player and box can't overlap
for t in range(0, self.T+1):
    for x, y in valid_positions:
        for b in range(self.num_boxes):
            self.cnf.append([-self.var_player(x, y, t),
                             -self.var_box(b, x, y, t)])
```

Box-Box Overlap: For every valid position (x, y) and pair of distinct boxes (b_1, b_2), the clause

$$\neg B_{b_1,x,y,t} \vee \neg B_{b_2,x,y,t}$$

is added at each time t, ensuring that no two boxes occupy the same cell.

```
# boxes can't overlap
for t in range(0, self.T+1):
    for x, y in valid_positions:
        for b1 in range(0, self.num_boxes-1):
            for b2 in range(b1+1, self.num_boxes):
                self.cnf.append([-self.var_box(b1, x, y, t),
                                 -self.var_box(b2, x, y, t)
                                    ])
```

Box movement constraints

A box can only move if it is pushed by the player. Formally, if at time t the player is at position (x, y) and a box b is in an adjacent cell $(x + dx, y + dy)$, with the cell $(x + 2dx, y + 2dy)$ empty and valid, then two possibilities exist for time $t + 1$:

1. The box remains in its current position $(x + dx, y + dy)$, while the player moves to some other adjacent position.
2. The player pushes the box into the empty cell $(x + 2dx, y + 2dy)$, and the player moves into the box's previous cell $(x + dx, y + dy)$.

To encode this, we add the precondition that the player must be at (x, y) and the box must be at $(x + dx, y + dy)$ at time t. From this precondition, we generate two possible future states:

- **Clause1:**

  $\neg\text{player}(x, y, t) \vee \neg\text{box}(b, x{+}dx, y{+}dy, t) \vee \text{box}(b, x{+}dx, y{+}dy, t{+}1) \vee \text{player}(x{+}dx, y{+}dy, t{+}1)$

  This means: if the precondition holds, then at $t + 1$ the box stays at its position, or the player moves into the box's cell.

- **Clause2:**

  $\neg\text{player}(x, y, t) \vee \neg\text{box}(b, x{+}dx, y{+}dy, t) \vee \text{box}(b, x{+}dx, y{+}dy, t{+}1) \vee \text{box}(b, x{+}2dx, y{+}2dy, t{+}1)$

  This means: if the precondition holds, then at $t+1$ either the box remains in place, or it is pushed forward into the empty cell.

In other words, `clause1` ensures the correct movement of the **player** when interacting with the box, while `clause2` ensures the correct movement of the **box** itself.

Implementation (by Jovita):

```
# if player-box-empty at t then at t+1 either box at same
    position and player at some other adjacent position
# or box goes in the empty cell and player occupies box's
    position at t+1
for t in range(0, self.T):
    for b in range(0, self.num_boxes):
        for x, y in valid_positions:
            for (dx, dy) in DIRS.values():
                bx, by = x+dx, y+dy
                bx2, by2 = x+2*dx, y+2*dy

                if (bx, by) in valid_positions and (bx2, by2
                    ) in valid_positions:

                    # precondition: player at (x,y) and box
                        at (bx,by) at time t
                    precond = [-self.var_player(x, y, t), -
                        self.var_box(b, bx, by, t)]

                    clause1 = [self.var_box(b, bx, by, t+1),
                         self.var_player(bx, by, t+1)]
                    clause2 = [self.var_box(b, bx, by, t+1),
                         self.var_box(b, bx2, by2, t+1)]

                    self.cnf.append(precond + clause1)
                    self.cnf.append(precond + clause2)
```

Box stability constraint

If a box is not pushed by the player at time t, then it must remain in the same cell at time $t + 1$. This ensures that boxes only move when an explicit push action is possible.

Formally, for a box b at position (x, y) at time t, the following clause is added:

$$\neg\text{box}(b, x, y, t) \ \vee \ \text{box}(b, x, y, t+1) \ \vee \bigvee_{(dx,dy)\in DIRS} \text{player}(x - dx,\, y - dy,\, t)$$

The meaning of this clause is:

- If the box is at (x, y) at time t, then it must either:
  1. Stay at (x, y) at time $t + 1$, or
  2. Have a player positioned at $(x - dx, y - dy)$ at time t, which allows a possible push in the (dx, dy) direction.
- If no player is in an adjacent pushing position, the box is forced to remain in the same cell.
- If a player is present in a valid pushing position, the stability clause does not prevent the box from moving, and its movement is governed by the *Box movement constraints*.

Implementation (by Rohan):

```
# if box is not pushed then it stays in the same place
for t in range(0, self.T):
    for b in range(0, self.num_boxes):
        for x, y in valid_positions:
            stay_clause = [-self.var_box(b, x, y, t),
                self.var_box(b, x, y, t + 1)]

            for dx, dy in DIRS.values():
                px, py = x - dx, y - dy
                nx, ny = x + dx, y + dy
                if (px,py) in valid_positions and (nx,ny
                    ) in valid_positions:
                    stay_clause.append(self.var_player(
                        px, py, t))
            self.cnf.append(stay_clause)
```

Goal constraint

At the final time step $t = T$, all boxes must occupy some goal cell. This ensures that the puzzle is only considered solved if every box has been pushed onto a designated goal position.

Formally, for each box b, we add the clause:

$$\bigvee_{(g_x, g_y) \in \text{Goals}} \text{box}(b, g_x, g_y, T)$$

This clause enforces that at time T, box b must be located in at least one of the goal cells. Since this constraint is applied to every box, the solver guarantees that all boxes end up on goals at the final step.

Implementation (by Jovita):

```
# goal condition
for b in range(0,self.num_boxes):
    self.cnf.append([self.var_box(b,gx,gy,self.T) for (gx,gy
        ) in self.goals])
```

The Decode Function

Decoding the SAT model into moves

Once the SAT solver returns a satisfying assignment, we must interpret it back into a valid sequence of player moves. This decoding step identifies the player's positions over time and converts the movement between consecutive positions into the standard move symbols `U` (up), `D` (down), `L` (left), and `R` (right).

- For each timestep $t \in [0, T]$, we check which grid cell (x, y) satisfies `player`(x, y, t). This gives us the sequence of player positions.
- Next, we compare consecutive positions (x, y) and (x', y') to determine the direction of movement:

$$\begin{aligned} x' = x - 1 &\quad \Rightarrow \quad \texttt{U} \quad \text{(Up)} \\ x' = x + 1 &\quad \Rightarrow \quad \texttt{D} \quad \text{(Down)} \\ y' = y - 1 &\quad \Rightarrow \quad \texttt{L} \quad \text{(Left)} \\ y' = y + 1 &\quad \Rightarrow \quad \texttt{R} \quad \text{(Right)} \end{aligned}$$

- Finally, we append the corresponding move symbol to the solution sequence.

This ensures that the satisfying assignment is translated back into a human-readable solution.

Implementation (By Jovita):

```
def decode(model, encoder):

    N, M, T = encoder.N, encoder.M, encoder.T

    # TODO: Map player positions at each timestep to
        movement directions
    moves = []
    positions = []

    # Extract player positions
    for t in range(0, T+1):
        found = False
        for x in range(0,N):
            for y in range(0,M):
```

```
            if (encoder.var_player(x, y, t)) in model:
                positions.append((x,y))
                found = True
                break
        if found:
            break

# Convert position differences into moves
for t in range(0,T):
    x, y = positions[t]
    nx, ny = positions[t+1]
    if nx == x - 1:
        moves.append("U")
    elif nx == x + 1:
        moves.append("D")
    elif ny == y - 1:
        moves.append("L")
    elif ny == y + 1:
        moves.append("R")

return moves
```