CS 228 Assignment 1 Report

Y Harsha Vardhan (Roll No: 24b1069)
M Rishi Vardhan (Roll No: 24b0969)

August 24, 2025

Contributions

Task	**Harsha Vardhan**	**Rishi Vardhan**
Question-1	Came up with the encoding scheme and constraints for rows and coloumns in the process of formalizing the problem helped in typing code for the program and made the report	Came up with the constraints for the subgrids, coding the program and helped in making report for the qsn
Question-2	came up with constraints for player movement, non-overlapping and goal conditions,made the report and helped in typing the code for the program	came up with constraints for box movement, player and box's uniqueness and existence, coded the program and helped in making the report.

Table 1: Division of contributions between team members.

1 Question 1: SAT-Based Sudoku Solver (30 Marks)

1.1 Problem Restatement

We need to implement a solver for a standard 9x9 Sudoku puzzle. Given a partially filled grid, where empty cells are marked with 0, the goal is to find a valid solution by modeling the puzzle as a SAT problem and using the PySAT library to solve it by finding a satisfying assignment (valid solution for the sudoku puzzle)

1.2 Approach Overview

We begin our approach by translating the Sudoku problem: the grid, the numbers, and the rules — into the language of propositional logic. We will use boolean variables such that we can represent any state of the grid using them.
We then formulate the rules of Sudoku as a series of clauses in Conjunctive Normal Form (CNF). This CNF formula is then passed to a SAT solver. If the formula is satisfiable, the solver returns a "model," which is an assignment of true/false values to our variables. We will then decode the model to get back to the solved Sudoku grid.

1.3 Variable Encoding

To represent the state of the grid, we define a propositional variable for each possible digit in each cell. The variable is $x_{i,j,k}$, where:

- i is the row index, from 0 to 8.
- j is the column index, from 0 to 8.
- k is the digit, from 1 to 9.

The variable $x_{i,j,k}$ is **True** if the cell at (row i, column j) {represented as cell(i, j)} contains the digit 'k', and **False** otherwise. This results in a total of $9 \times 9 \times 9 = 729$ variables. In the Python code, we map each variable $x_{i,j,k}$ to a unique integer using the formula (100*i + 10*j + k).

1.4 Constraints

We encode the rules of Sudoku as clauses that constrain the possible values of our variables (xij_k). We ensure that each cell, row, column, and 3x3 subgrid contains each digit from 1 to 9 exactly once.

- **Cell Constraints:** Each cell(i, j) must contain exactly one digit.
  1. *At least one digit:* For each cell(i, j), the clause is the disjunction of all possible digits for that cell.
  $$\bigvee_{k=1}^{9} xij\_k$$
  2. *At most one digit:* For any two distinct digits k_1 and k_2, a cell cannot contain both.
  $$\neg xij\_k_1 \vee \neg xij\_k_2 \quad \forall k_1 \neq k_2$$

Listing 1: Snippet for Cell Constraints

```
# each cell must have at least one digit(1-9)
for i in range(9):
    for j in range(9):
        # (xij_1 V xij_2 V ... V xij_9)
        sudoku.append([x(i,j,k) for k in range(1,10)])

# each cell must have at most one digit(1-9)
for i in range(9):
    for j in range(9):
        for k1 in range(1,10):
            for k2 in range(k1 + 1,10):
                # (~xij_k1 V ~xij_k2)
                sudoku.append([-x(i,j,k1),-x(i,j,k2)])
```

- **Row Constraints:** Each digit ‘k‘ must appear exactly once in each row ‘i‘.
    1. *At least one of each digit:*
    $$\bigvee_{j=0}^{8} xij\_k$$
    2. *At most one of each digit:*
    $$\neg xij_1\_k \vee \neg xij_2\_k \quad \forall j_1 \neq j_2$$
- **Column Constraints:** Each digit ‘k‘ must appear exactly once in each column ‘j‘.
    1. *At least one of each digit:*
    $$\bigvee_{i=0}^{8} xij\_k$$
    2. *At most one of each digit:*
    $$\neg xi_1j\_k \vee \neg xi_2j\_k \quad \forall i_1 \neq i_2$$

Listing 2: Snippet for Row and Coloumn Constraints

```
# each row and column should have all the digits (At least one of
    each k)
for i in range(9):
    for k in range(1,10):
        sudoku.append([x(i,j,k) for j in range(9)])
for j in range(9):
    for k in range(1,10):
        sudoku.append([x(i,j,k) for i in range(9)])

# each row and column should not have any repeating digit (At
   most one of each k)
```

```
for i in range(9):
    for k in range(1, 10):
        for j1 in range(9):
            for j2 in range(j1 + 1, 9):
                sudoku.append([-x(i, j1, k), -x(i, j2, k)])
for j in range(9):
    for k in range(1, 10):
        for i1 in range(9):
            for i2 in range(i1 + 1, 9):
                sudoku.append([-x(i1, j, k), -x(i2, j, k)])
```

- **Subgrid Constraints:** Each digit ‘k‘ must appear exactly once in each 3x3 subgrid. For each subgrid S and each digit k:

  1. *At least one of each digit:*

  $$\bigvee_{(i,j)\in S} xij\_k$$

  2. *At most one of each digit:* For any two distinct cells (i_1, j_1) and (i_2, j_2) in S:

  $$\neg xi_1j_1\_k \vee \neg xi_2j_2\_k$$

- **Pre-filled Cell Constraints:** The initial numbers in the puzzle must be respected. For each cell (i, j) containing a given digit $k \neq 0$ in the input grid, we add a unit clause to force the corresponding variable to be true.

  $$(xij\_k)$$

Listing 3: Snippet for Subgrid and Pre-filled Cell Constraints

```
# each 3X3 subgrid should have all digits (At least one of each k
   )
for t1 in range(0,9,3):
    for t2 in range(0,9,3):
        for k in range(1,10):
            sudoku.append([x(i+t1,j+t2,k) for i in range(3) for j
                in range(3)])

# each 3X3 subgrid should not have any repeating digit (At most
   one of each k)
for k in range(1, 10):
    for t1 in range(0, 9, 3):
        for t2 in range(0, 9, 3):
            cells = [x(i + t1, j + t2, k) for i in range(3) for j
                in range(3)]
            for c1 in range(len(cells)):
                for c2 in range(c1 + 1, len(cells)):
                    sudoku.append([-cells[c1], -cells[c2]])
```

```
# for the given numbers in the input grid
for i in range(9):
    for j in range(9):
        if grid[i][j] != 0:
            sudoku.append([x(i, j, grid[i][j])])
```

1.5 Decoding

If the SAT solver finds the formula to be satisfiable (i.e., if the given partially filled sudoku input is valid), it provides a model containing a list of integer literals. We only require the positive literals because they correspond to variables that are assigned True (we get the correct digit at all possible 81 positions).
We loop through this list, filtering for positive literals, and for each literal we reverse our mapping (100*i + 10*j + k) to find the corresponding row ‘i‘, column ‘j‘, and digit ‘k‘. For each such true variable xij_k, we place the digit k into the cell(i,j) of our solution grid.

1.6 Discussion

This encoding approach is both correct and complete, as it captures all the rules of Sudoku. Whenever a solution exists, the SAT solver is guaranteed to find it. An interesting point to note here is that, some constraints: such as the “at most one” rule for rows and columns — are technically redundant because they can already be inferred from other rules through the pigeonhole principle. Still, they play an important practical role. Adding these constraints makes the solver’s job easier by reducing the search space, which significantly speeds up the solving process.

Question 2: Grading Assignments Gone Wrong (70 Marks)

Problem Restatement

We need to solve a Sokoban-like puzzle by modeling it as a Satisfiability (SAT) problem. We are given an $\boldsymbol{N} \times \boldsymbol{M}$ grid representing an office, a maximum number of moves $\boldsymbol{T}$, and the initial position of a player, several boxes and several goal locations.
The objective is to find if a sequence of moves exists to push every box into a goal cell within the given time limit of $\boldsymbol{T}$ minutes.

Variable Encoding

We need to choose propositional variables which help us do the following task: track the location of every object at every point in time.
Given that the grid has $\boldsymbol{N}$ rows and $\boldsymbol{M}$ columns and let there be $\boldsymbol{K}$ boxes, so we can define the following as our propositional variables for each time step t from 0 to $\boldsymbol{T}$:

- $var\_player(x, y, t)$**:** A function that assigns an integer to the variable that is encoded from the inputs x, y and t. It is used to represent the position of player at (x, y) at time t.
- $var\_box(i, x, y, t)$: A function that assigns an integer to the variable that is encoded from the inputs x, y, t. It is used to represent the position of the box (it is given an index i) at (x, y), at time t.

In the code, these variables are mapped to unique positive integers. This encoding was done in the following manner because it is efficient.

Listing 4: Variable Encoding Used

```python
# ---------------- Variable Encoding ----------------
def var_player(self, x, y, t):
    """
    Variable ID for player at (x, y) at time t.
    """
    # TODO: Implement encoding scheme
    return 1 + (t * self.N * self.M) + (x * self.M) + y

def var_box(self, b, x, y, t):
    """
    Variable ID for box b at (x, y) at time t.
    """
    # TODO: Implement encoding scheme
    k = (self.T + 1) * self.N * self.M
    return k+1+ (b*k) + (t * self.N * self.M) + (x * self.M) + y
```

Constraints

The rules of the game are translated into a set of clauses in Conjunctive Normal Form (CNF).

Initial State:

We use simple unit clauses to define the starting configuration of the puzzle at $t = 0$.

- For the player's starting position (x_p, y_p), we add the clause: $P(x_p, y_p, 0)$
- For each box b starting at (x_b, y_b), we add the clause: $B(b, x_b, y_b, 0)$

Transition Rules (Player Movement and Box Pushing):

These constraints define how the state can evolve from time t to $t + 1$.

- **Player Movement**: The player must move to an adjacent valid cell or stay at the same cell at each time step. A player at (x, y) must have a valid destination at $t + 1$.

$$P(x, y, t) \implies \bigvee_{(new\_x, new\_y) \in \mathrm{adj}(x,y) \cup \{(x,y)\}} P(new\_x, new\_y, t + 1)$$

Listing 5: Player Movement Logic

```
# 2. Player movement
#Adding rules for player movement
for t in range(self.T):
    for x in range(self.N):
        for y in range(self.M):
            if self.grid[x][y] == '#': continue
            v = self.var_player(x, y, t)
            poss_moves = []
            for d, (dx, dy) in DIRS.items():
                new_x, new_y = x + dx, y + dy
                if 0 <= new_x < self.N and 0 <= new_y < self.M
                   and self.grid[new_x][new_y] != '#':
                    poss_moves.append(self.var_player(new_x,
                        new_y, t + 1))
            if poss_moves: self.cnf.append([-v] + poss_moves)
                #(-v) or (poss_move1) or (poss_move2) ....
```

- **Box Movement Logic**: A box's position at $t + 1$ is determined by its position at t and the player's actions.The logic states that if a player moves in a particular way (from an adjacent cell to the cell where the box is) then the box moves or the fact that the box has moved must mean that the player has moved from an adjacent cell to the cell where the box was.

$$B(b, new\_pos, t + 1) \wedge B(b, old_pos, t) \implies P(player\_start, t) \wedge P(old\_pos, t + 1)$$

  Furthermore, for a box to be at a location, it must either have been pushed there or it must already have been there and not pushed away.

Non-Overlap Constraints:

These clauses ensure that the following holds:

- player and wall don't overlap
- box and wall don't overlap
- two boxes don't overlap
- player and box don't overlap

Listing 6: Non-Overlap Constraints for Encode

```
# 4. Non-overlap constraints
for t in range(self.T + 1):
    for x in range(self.N):
        for y in range(self.M):
            if self.grid[x][y] == '#':
                self.cnf.append([-self.var_player(x,y,t)])  #
                   making sure player and wall dont overalp
                for b in range(self.num_boxes):
                    self.cnf.append([-self.var_box(b,x,y,t)]) #
                       making sure box and wall dont overlap

            a = [self.var_box(b, x, y, t) for b in range(self.
               num_boxes)]
            for i in range(len(a)):
                for j in range(i + 1, len(a)):
                    self.cnf.append([-a[i], -a[j]]) #making sure
                       2 boxes dont overlap

            for b in range(self.num_boxes):
                self.cnf.append([-self.var_player(x, y, t), -self
                   .var_box(b, x, y, t)])# making sure box and
                   player dont overlap
```

Goal State:

These clauses ensure that the puzzle is considered solved at the final time step T.

- **All Boxes in Goals**: For each box b, it must be in one of the goal cells.

$$\bigvee_{(gx,gy)\in \text{Goals}} B(b, gx, gy, T)$$

- **Unique Goal Occupancy**: A single goal cell cannot be occupied by more than one box. For each goal (gx, gy) and each pair of boxes i, j:

$$\neg B(i, gx, gy, T) \vee \neg B(j, gx, gy, T)$$

Listing 7: Goal State Condition

```
# 5. Goal conditions
# Each box should be on some goal
for i in range(self.num_boxes):
    self.cnf.append([self.var_box(i,goal_x,goal_y,self.T) for
        goal_x, goal_y in self.goals])
```

Other Conditions:

These constraints must hold true for all time steps $t \in [0, T]$ to ensure the game's rules are always followed.

- **Exactly One Position**: The player must occupy exactly one valid (non-wall) cell at all times. This is enforced with "at-least-one" and "at-most-one" clauses. For the player:

$$\bigvee_{(x,y)\in \text{floor}} P(x,y,t) \wedge \bigwedge_{(x1,y1)\neq(x2,y2)} (\neg P(x1,y1,t) \vee \neg P(x2,y2,t))$$

- A Box must occupy exactly one valid cell just like the Player

$$\bigvee_{(x,y)\in \text{floor}} B(b,x,y,t) \wedge \bigwedge_{(i,x1,y1)\neq(j,x2,y2)} (\neg B(i,x1,y1,t) \vee \neg B(j,x2,y2,t))$$

Decoding

The satisfying assignment (the 'model') returned by the SAT solver represents a complete set of positions of all objects at all times, made in a valid game solution. The 'decode' function converts this model into a sequence of moves.

1. It first identifies all the true $P(x, y, t)$ variables in the model. This is done by mathematically inverting the variable encoding formula to find the (x, y, t) tuple for each true variable ID.
2. This information is then used to build a complete path of the player's coordinates for each time step, from $t = 0$ to $t = T$.
3. The function then iterates from $t = 0$ to $T - 1$ and compares the player's position at time t with their position at $t + 1$.
4. The difference in coordinates, $(\Delta x, \Delta y)$, corresponds directly to a move: $(-1, 0)$ is 'U', $(1, 0)$ is 'D', etc. moves where the player stays put are filtered out.
5. These characters are concatenated to form the final solution string. If the solver returns no model (UNSAT), it means no solution exists, and -1 is returned.