# CS-228 : Assignment 1

Ansh Agrawal (24B0909), Saksham Khandelwal (24B0965)

## Contents

# 1 SAT-Based Sudoku Solver

## 1.1 Approach Overview

To solve sudoku puzzle using a SAT Solver, we need to make variables and add the constraints such that any satisfyable assignment corresponds to solved puzzle. That is we need to define the variables with their condition such that there exists a bijection between value of these variables and state of filled sudoku puzzle and add constraints on them such that only such assignment are possible that are related to solved sudoku puzzle from the given conditions.

Variable Conditions

- One boolean variable for a particular grid cell having one particular value
- One grid cell should have exactly one such variable true

Due to this, we define $P_{rcm}$ as follows:

$$P_{rcm} = \begin{cases} 1 & \text{if the grid cell } (r, c) \text{ contains number } m, \\ 0 & \text{otherwise.} \end{cases}$$

where (r,c) represents the grid cell where both r and c $\in \{0, 1, \ldots, n-1\}$ (0 based indexing of rows and columns) and m $\in \{1, 2, \ldots, n\}$ representing the number inside the cell

Sudoku Constraints

- Row: All digits should appear in each row
- Column: All digits should appear in each column
- SubGrid: All digits should appear in each subgrid
- Given: All given digits should not change

If we are able to encode the above things, then we can use a SAT solver to get one such satisfyable assignment then we can decode to get the solved sudoku.

## 1.2 Variable Encoding

### 1.2.1 Defining Variables

As the Python-SAT SAT Solver requires the variables as integers we encode the variable $P_{rcm}$ as follows:

$$P_{rcm} = (r * n + c) * n + m$$

And as both c and m can take n distinct values, this integer would be distinct for different tuples $(r, c, m)$

```
n = len(grid)
sub_n = int(n**0.5)

def prop(row: int, col: int, num: int) -> int:
    return (row * n + col) * n + num
```

### 1.2.2 Constraints

#### 1.2.2.1 Cell: Each cell must contain exactly one number.

$$\bigwedge_{r=0}^{n-1}\bigwedge_{c=0}^{n-1}\left(\bigvee_{m=1}^{n} P_{rcm}\right) \quad \text{and} \quad \bigwedge_{r=0}^{n-1}\bigwedge_{c=0}^{n-1}\left(\bigwedge_{1\le m_1<m_2\le 9} \neg P_{rcm_1} \vee \neg P_{rcm_2}\right)$$

```python
for r in range(n):
    for c in range(n):
        clause = [prop(r, c, num) for num in range(1, n + 1)]
        cnf.append(clause)

        for i in range(n):
            for j in range(i + 1, n):
                cnf.append([-clause[i], -clause[j]])
```

#### 1.2.2.2 Row: Each number must appear in each row.

$$\bigwedge_{r=0}^{n-1}\bigwedge_{m=1}^{n}\left(\bigvee_{c=0}^{n-1} P_{rcm}\right)$$

```python
for r in range(n):
    for num in range(1, n + 1):
        clause = [prop(r, c, num) for c in range(n)]
        cnf.append(clause)
```

#### 1.2.2.3 Column: Each number must appear in each column.

$$\bigwedge_{c=0}^{n-1}\bigwedge_{m=1}^{n}\left(\bigvee_{r=0}^{n-1} P_{rcm}\right)$$

```python
for c in range(n):
    for num in range(1, n + 1):
        clause = [prop(r, c, num) for r in range(n)]
        cnf.append(clause)
```

#### 1.2.2.4 SubGrid: Each number must appear in each subgrid($s \times s$) where $s = \sqrt{n}$

$$\bigwedge_{i=0}^{s-1}\bigwedge_{j=0}^{s-1}\bigwedge_{m=1}^{n}\left(\bigvee_{a=0}^{s-1}\bigvee_{b=0}^{s-1} P_{is+a,\, js+b,\, m}\right).$$

```python
for br in range(0, n, sub_n):
    for bc in range(0, n, sub_n):
        for num in range(1, n + 1):
            clause = [prop(r, c, num)
                      for r in range(br, br + sub_n)
                      for c in range(bc, bc + sub_n)]
            cnf.append(clause)
```

**1.2.2.5 Initial Conditions:** Propositions representing given values should be true. Let S be the set of all initial value tuples (row, column, number) then

$$\bigwedge_{(r,c,m)\in S} P_{rcm}$$

```
for r in range(n):
    for c in range(n):
        if grid[r][c] != 0:
            cnf.append([prop(r, c, grid[r][c])])
```

### 1.2.3 Output

Encoding by the above scheme and applying the constraints, if the SAT solver is now able to have an satisfyable assignment of this cnf formula, that means there exist a solved state of the given input sudoku. And thus we can get the solved sudoku by checking all the variables that are true in the assignment.

```
with Solver(name='glucose42', bootstrap_with=cnf.clauses) as solver:
    solution = grid

    if solver.solve():
        model = solver.get_model()

        for r in range(n):
            for c in range(n):
                for num in range(1, n + 1):
                    if model[prop(r, c, num) - 1] > 0: # proposition is
                        1 indexed whereas model array is 0 indexed
                        solution[r][c] = num
                        break

    return solution
```

## 1.3 Contribution

Ideation was done independently and discussed afterwards. As this example was done in class, the ideas were more or less the same. The code work was not as big and tedious, so one of us wrote it, while the other helped sitting beside him. We debugged together for the errors and generalised for any size n.

# 2 Grading Assignments Gone Wrong

## 2.1 Approach Overview

Similar to the above problem, here also we need to define the variables in such a way that all the properties of game and a particular state can be represented by these boolean variables, that are:

- Constant Properties:
  - Goal Positions
  - Wall Positions
- State Properties(dependent on time instant t):
  - Player Position
  - Player Move
  - Box Positions

And after forming such variables, we need to add all the constraints(explained in next section) to get a satisfyable assignment from which we can decode back the moveset played in those $T$ turns.

## 2.2 Variable Declarations

**• Input Parsing $\implies$**

**goals:** List of goal locations ('G') in form of tuples (x,y)

**boxes:** List of initial locations of box('B') in form of tuples (x,y), index represent box id

**walls:** List of wall locations ('#') in form of tuples (x,y)

**player_start:** Tuple pointing to the starting position of player('P')

These variables were updated just once during reading of input.

**• Proposition Variables $\implies$**

Taking, grid to be of size $N * M$ and 0-indexed, and $T$ = number of total states

**Note:** Instead of taking $T$ to be the number of steps, it is the number of states, so there is one extra state for initial conditions.

$*$**Goals** Goal Presence at that position

$$G_{x,y} = x * M + y + 1$$

$*$**Player** Player Presence at that position, time

$$P_{x,y,t} = [M * N] + t * M * N + x * M + y + 1$$

∗**Moves** Move done to reach t from t-1

$$\text{move} \in \begin{Bmatrix} 1 : \text{Move Up},\ 2 : \text{Move Right},\ 3 : \text{Move Down},\ 4 : \text{Move Left}, \\ 5 : \text{Push Up},\ 6 : \text{Push Right},\ 7 : \text{Push Down},\ 8 : \text{Push Left}, \\ 9 : \text{Stay} \end{Bmatrix}$$

$$M_{move,t} = [T * M * N + M * N] + 9 * t + move$$

∗**Boxes** Particular Box Presence at that position, time

$$B_{b,x,y,t} = [T * M * N + M * N + 9 * T] + B * t * M * N + (b-1) * M * N + x * M + y + 1$$

• **Indexing, Grouping, and Ranges** $\implies$

The integer ranges of variables and their grouping structure are as follows:

**Goals** ($G_{x,y}$) :
Range: 1 to $M * N$
Grouping: indexed only over grid cells.

**Players** ($P_{x,y,t}$) :
Range: $M * N + 1$ to $T * M * N + M * N$
Grouping: grouped by time $t$. For each $t$, all player positions $(x, y)$ are enumerated consecutively, then the next $t$ begins.

**Moves** ($M_{move,t}$) :
Range: $T * M * N + M * N + 1$ to $T * M * N + M * N + 9 * T$
Grouping: grouped by time $t$. For each $t$, all 9 possible moves are listed together, then the next $t$ begins.

**Boxes** ($B_{b,x,y,t}$) :
Range: $T * M * N + M * N + 9 * T + 1$ to $T * M * N + M * N + 9 * T + B * T * M * N$
Grouping: grouped first by time $t$, then by box identifier $b$, and finally by grid position $(x, y)$.

Thus, the integer assignment is consistent: first by time and then by the ordering of sub-indices (player positions, moves, or boxes).

By definition of these variables, each one takes distinct value and represent all the properties of the game.

## Notation regarding Boxes

The list boxes contains tuples of initial indices (x,y) of boxes and different boxes are distinguished based on their indices in the list. So,
$(x, y) \in boxes$ represent (x,y) is one of the element in this list.
$b : boxes$ represent b is one of the index (or equivalently box id) of this list of boxes.

## 2.3 Constraints

### 2.3.1 Initial Conditions

**2.3.1.1 Player Starting Position** Proposition representing player initial position (i.e., at time 0) must be true. Position saved in player_start variable during input parsing.

$$P_{player\_start_x, player\_start_y, 0}$$

**2.3.1.2 Box Initial Positions** Similar for the boxes, the initial condition variables must be true. $b$ here is the index of that $(x, y)$ representing the box id.

$$\bigwedge_{(x,y) \in boxes} B_{b,x,y,0}$$

**2.3.1.3 Move 0** As move at time t represent move done at instant t-1 to reach t, hence we need to put move at time 0 to stay.

$$M_{9,0}$$

### 2.3.2 Wall Blocking

At any time, player or box cant be at wall positions, so we just need to make those propositions false.

$$\bigwedge_{(x,y) \in walls} \bigwedge_{t=0}^{T-1} \neg P_{x,y,t} \quad , \quad \bigwedge_{(x,y) \in walls} \bigwedge_{t=0}^{T-1} \bigwedge_{b:boxes} \neg B_{b,x,y,t}$$

### 2.3.3 Player Movement (walk rules)

If the player performs a walk move from $(x, y)$, then at the next time step he must occupy the adjacent cell corresponding to that move, provided that the cell lies within the grid and is not occupied by a box.

The new position based on move relation is defined as

$$(x_{\text{new}}, y_{\text{new}}) = \begin{cases} (x-1, y), & \text{if move } = 1 \text{ (Up)} \\ (x, y+1), & \text{if move } = 2 \text{ (Right)} \\ (x+1, y), & \text{if move } = 3 \text{ (Down)} \\ (x, y-1), & \text{if move } = 4 \text{ (Left)} \end{cases}$$

**2.3.3.1 Invalid Moves** If $(x_{\text{new}}, y_{\text{new}})$ lies outside the grid boundary, then the corresponding move cannot be applied:

$$\bigwedge_{x,y} \bigwedge_{\substack{\text{move} \in \{1,2,3,4\} \\ (x_{\text{new}}, y_{\text{new}}) \notin grid}} \bigwedge_{t=1}^{T-1} \neg P_{x,y,t-1} \vee \neg M_{\text{move},t}$$

**2.3.3.2 Box Stability** As no box movement is involved in this type of move, the box positions should not change during this turn.

$$\bigwedge_{x,y} \bigwedge_{\text{move}\in\{1,2,3,4\}} \bigwedge_{t=1}^{T-1} \bigwedge_{b:boxes} (\neg M_{\text{move},t} \vee \neg B_{b,x,y,t-1} \vee B_{b,x,y,t})$$

**2.3.3.3 Player Position Update** If $(x_{\text{new}}, y_{\text{new}})$ lies inside the grid, then

$$\bigwedge_{x,y} \bigwedge_{\text{move}\in\{1,2,3,4\}} \bigwedge_{t=1}^{T-1} \neg M_{\text{move},t} \vee \neg P_{x,y,t-1} \vee P_{x_{\text{new}},y_{\text{new}},t}$$

### 2.3.4 Box Movement (push rules)

If the player performs a push move from $(x, y)$ into an adjacent cell $(x_{\text{new}}, y_{\text{new}})$ that contains a box, then the box must be pushed further in the same direction to a new position $(x_{\text{box}}, y_{\text{box}})$ at time $t$ if its empty(already handled by overlap conditions) and inside grid.

The player position updates $(x_{new}, y_{new})$ are same as above.

and

$$(x_{\text{box}}, y_{\text{box}}) = \begin{cases} (x_{\text{new}} - 1, y_{\text{new}}), & \text{if move } = 5 \\ (x_{\text{new}}, y_{\text{new}} + 1), & \text{if move } = 6 \\ (x_{\text{new}} + 1, y_{\text{new}}), & \text{if move } = 7 \\ (x_{\text{new}}, y_{\text{new}} - 1), & \text{if move } = 8 \end{cases}$$

**2.3.4.1 Invalid pushes** If either $(x_{\text{new}}, y_{\text{new}})$ or $(x_{\text{box}}, y_{\text{box}})$ lies outside the grid, then the corresponding push move is not allowed:

$$\bigwedge_{x,y} \bigwedge_{\substack{\text{move}\in\{5,6,7,8\} \\ (x_{\text{new}},y_{\text{new}})\notin grid \\ \textbf{or} \\ (x_{\text{box}},y_{\text{box}})\notin grid}} \bigwedge_{t=1}^{T-1} \neg P_{x,y,t-1} \vee \neg M_{\text{move},t}$$

**2.3.4.2 Valid pushes** If the move is valid, then these four conditions must hold:

1. There must be a box at $(x_{\text{new}}, y_{\text{new}})$ at $t-1$ into which player moves:

$$\bigwedge_{x,y} \bigwedge_{\text{move}\in\{5,6,7,8\}} \bigwedge_{t=1}^{T-1} (\neg M_{\text{move},t} \vee \neg P_{x,y,t-1} \vee \bigvee_{b:boxes} B_{b,x_{\text{new}},y_{\text{new}},t-1})$$

2. The player moves into the box's old position:

$$\bigwedge_{x,y} \bigwedge_{\text{move}\in\{5,6,7,8\}} \bigwedge_{t=1}^{T-1} \bigwedge_{b:boxes} \neg M_{\text{move},t} \vee \neg P_{x,y,t-1} \vee \neg B_{b,x_{\text{new}},y_{\text{new}},t-1} \vee P_{x_{\text{new}},y_{\text{new}},t}$$

3. The box is pushed into the next cell:

$$\bigwedge_{x,y} \bigwedge_{\text{move}\in\{5,6,7,8\}} \bigwedge_{t=1}^{T-1} \bigwedge_{b:boxes} \neg M_{\text{move},t} \vee \neg P_{x,y,t-1} \vee \neg B_{b,x_{\text{new}},y_{\text{new}},t-1} \vee B_{b,x_{\text{box}},y_{\text{box}},t}$$

4. All other boxes preserve their positions:

$$\bigwedge_{\substack{x,y \\ (x,y)\neq(x_{new},y_{new})}} \bigwedge_{\text{move}\in\{5,6,7,8\}} \bigwedge_{t=1}^{T-1} \bigwedge_{b:boxes} \neg M_{\text{move},t} \vee \neg P_{x,y,t-1} \vee \neg B_{b,x,y,t-1} \vee B_{b,x,y,t}$$

### 2.3.5 Stay Conditions

If player stayed at last turn, then neither the player nor any of the boxes should move.

$$\bigwedge_{t=1}^{T-1} \bigwedge_{x,y} (\neg M_{9,t} \vee \neg P_{x,y,t-1} \vee P_{x,y,t})$$

and

$$\bigwedge_{t=1}^{T-1} \bigwedge_{x,y} \bigwedge_{b:boxes} (\neg M_{9,t} \vee \neg B_{b,x,y,t-1} \vee B_{b,x,y,t})$$

### 2.3.6 Non Overlap

At any time, no box with other boxes or with player can overlap with each other on any point.

$$\bigwedge_{x,y} \bigwedge_{b:boxes} \bigwedge_{t=0}^{T-1} (\neg P_{x,y,t} \vee \neg B_{b,x,y,t})$$

and

$$\bigwedge_{x,y} \bigwedge_{\substack{b,b':boxes \\ b<b'}} \bigwedge_{t=0}^{T-1} (\neg B_{b,x,y,t} \vee \neg B_{b',x,y,t})$$

### 2.3.7 Entities unique position per time

For player, it should be at some unique position at any fixed time t for all t.

$$\bigwedge_{t=0}^{T-1} \bigvee_{x,y} P_{x,y,t} \quad , \quad \bigwedge_{t=0}^{T-1} \bigwedge_{(x,y)\neq(x',y')} \neg P_{x,y,t} \vee \neg P_{x',y',t}$$

Similarly for boxes, a particular box should be at some unique position at any fixed time t for all t.

$$\bigwedge_{t=0}^{T-1} \bigwedge_{b:boxes} \bigvee_{x,y} B_{b,x,y,t} \quad , \quad \bigwedge_{t=0}^{T-1} \bigwedge_{b:boxes} \bigwedge_{(x,y)\neq(x',y')} \neg B_{b,x,y,t} \vee \neg B_{b,x',y',t}$$

### 2.3.8 Exactly one move at a time

At one time instant, the player can only play a unique move once. Atleast one move

$$\bigwedge_{t=0}^{T-1} \bigvee_{m\in\{1,2,\ldots,9\}} M_{m,t}$$

Atmost one move

$$\bigwedge_{t=0}^{T-1} \bigwedge_{\substack{m,m'\in\{1,2,\ldots,9\} \\ m<m'}} (\neg M_{m,t} \vee \neg M_{m',t})$$

### 2.3.9 Final State (Goal Conditions)

At time T, all the boxes should be at goal positions.
NOTE: We have assumed number of goals to be independent of number of boxes. Hence to win, we need to have all the boxes on goals and not otherwise.

$$\bigwedge_{x,y} \bigwedge_{b:boxes} (\neg B_{b,x,y,T-1} \vee G_{x,y})$$

## 2.4 Decoding

Decoding in our case is straightforward, as the variables for moves are defined explicitly. We just need to iterate over them and identify the positive ones in the solver output model. Since the encoding guarantees that exactly one move is selected at each time step, the decoding procedure reduces to locating that variable and mapping it back to the corresponding move direction.

Note: Our variables are defined to be 1-indexed, while the solver output list is 0-indexed. Therefore, a shift of 1 is required. The move variables occupy the range

$$(T+1) \cdot M \cdot N + 1 \quad \text{to} \quad (T+1) \cdot M \cdot N + 9T,$$

which corresponds in the solver output to array indices

$$(T+1) \cdot M \cdot N \quad \text{to} \quad (T+1) \cdot M \cdot N + 9T - 1.$$

To get the move for each time step $t$, we identify the positive variable and apply the modulus operator with respect to 9, which uniquely determines the move type (up, right, down, left, push-up, push-right, push-down, push-left, or stay). The result is then appended to a sequence of moves.

```
moves = []
for index in range((T+1)*N*M, (T+1)*N*M + 9*T):
    if (model[index]>0):
        match ((index - (T+1)*N*M)%9):
            case 0 : moves.append('U')
            case 1 : moves.append('R')
            case 2 : moves.append('D')
            case 3 : moves.append('L')
            case 4: moves.append('U')
            case 5: moves.append('R')
            case 6: moves.append('D')
            case 7: moves.append('L')
            # case 8: moves.append('S')
return moves
```

## 2.5 Contribution

Ideation was done together. The thought process of mapping our variables with numbers was mostly done by Saksham while the constraints for movement were thought by Ansh. However, both of us complemented each others ideas for better results. Debugging was done together. We used ”codeshare.io” for writing the code in parallel. We distributed the code writing work in the following way:

Ansh -

1. Player movement constraints
2. Box movement constraints
3. Unique move constraint
4. Stay constraints

Saksham -

1. Variable Encoding and Decoding
2. Initial Constraints
3. Goal condition
4. Uniqueness per time Condition
5. Non-overlap constraint