Logic Assignment

CS 228

Submitted by:
Tamanna Kumari, 24B1015
Sagar V, 24B1021

Question 1

Approach towards encoding Sudoku rules as a CNF

We categorised 5 constraints which should be encoded.

Variable Encoding

Suppose P(i,j,num) stands true if num occupies row i and column j.

$$1 \leq num \leq 9$$

$$0 \leq i, j \leq 8$$

Now we encoded P(i,j,num) as follows:

$$81 * i + 9 * j + num$$

This gives us numbers from 1 to 729 each corresponding to a unique P(i,j,num).

Constraints

```
    # Encoding all given numbers
    for i in range(9):
        for j in range(9):
            if (grid[i][j] != 0 ):
                box_no = i*9 + j
                cnf.append([box_no*9+grid[i][j]])
```

This appends to our cnf that the initial marking of numbers that we have to solve must be true.

```
    # Encoding rows, columns and 3x3 grid condition
    for num in range(9):
        cell = 0
        for i in range(9):
            row_encode = []
            col_encode = []
            box_encode = []

            if(i%3==0):
                cell = 9*i+1
            else:
                cell = cell - 17

            for j in range(9):
                row_encode.append((9*i+j)*9+num+1)
                col_encode.append((i+9*j)*9+num+1)

                box_encode.append((cell-1)*9+num+1)

                if j==8:
                    continue
```

```
            if((j+1)%3==0):
                cell += 7
            else:
                cell += 1

        cnf.append(row_encode)
        cnf.append(col_encode)
        cnf.append(box_encode)
```

Each row must contain atleast one occurence of 1,2,...9

$$\bigwedge_{\text{num}=1}^{9} \bigwedge_{\text{row}=0}^{8} \left(\bigvee_{\text{col}=0}^{8} P(\text{row}, \text{col}, \text{num}) \right)$$

Each column must contain atleast one occurence of 1,2,...9

$$\bigwedge_{\text{num}=1}^{9} \bigwedge_{\text{col}=0}^{8} \left(\bigvee_{\text{row}=0}^{8} P(\text{row}, \text{col}, \text{num}) \right)$$

Each 3x3 block must contain atleast one occurence of 1,2,..,9

$$\bigwedge_{\text{num}=1}^{9} \bigwedge_{\text{box\_row}=0}^{2} \bigwedge_{\text{box\_col}=0}^{2} \left(\bigvee_{i=0}^{2} \bigvee_{j=0}^{2} P(3 \cdot \text{box\_row} + i, 3 \cdot \text{box\_col} + j, \text{num}) \right)$$

Row and Column Encoding Explanation

We use three nested loops over `num`, `i`, and `j`. Here, `num` represents the number 1–9, `i` the row index, and `j` the column index.

For each row `i` and number `num`, we maintain a list `row_encode` which represents a single clause, i.e., the disjunction (OR) of all boxes in that row for the given number. Using `j`, we iterate over all 9 columns/boxes in the row and add the corresponding variable $P(i, j, num)$ to the clause, ensuring that the number `num` appears in at least one box of the row.

Similarly, we use `col_encode` and iterate using j over all rows/boxes in a particular column and add a clause for every number and every column ensuring that the number num appears in atleast one box of every column.

For encoding atleast one appearance of each number in a 3x3 box,

Box Encoding Explanation

The list `box_encode` represents a single clause that ensures a given number `num` appears at least once in a 3×3 box. The variable `cell` is used to traverse all 9 cells of the current box in a linearized SAT variable numbering. For each column index `j`, we append the corresponding SAT variable $(cell - 1) \cdot 9 + num + 1$ to `box_encode`. The arithmetic on `cell` ensures that after each step, we correctly move to the next cell within the 3×3 box, handling the jumps between rows of the box. After iterating over all columns, `box_encode` contains all variable indices for the number in that box, and adding it to `cnf` enforces the "at least one occurrence per box" constraint.

- At the start of a new set of three rows (`i % 3 == 0`), `cell` is initialized to the first cell of the top row of the current box:

$$\texttt{cell} = 9 \cdot i + 1.$$

  This sets the starting point for the box in the flattened 9×9 grid numbering.

- For the second and third rows of the 3×3 box, `cell` is adjusted backwards by 17:

$$\texttt{cell} = \texttt{cell} - 17.$$

  This moves the pointer from the end of the previous row back to the start of the next row of the box, correctly positioning for the next set of three cells.

- Within a row of the box, after each column `j`, `cell` is incremented to move to the next cell:
  - If `(j+1)%3 == 0` (i.e., at the end of a mini-row within the box), `cell` is incremented by 7. This jumps to the first cell of the next mini-row within the 3×3 box.
  - Otherwise, `cell` is incremented by 1 to move to the next column in the same mini-row.

No two numbers accupy the same box

We encoded the atleast one rule for each row, column and cell. But it can also be the case that our SAT solver returns true for all P(i,j,num). To avoid such a scenario we encode the condition that for any two numbers they cannot be in the same box.

```
for i in range(81):
    for j in range(9):
        for k in range(j+1,9):
            cnf.append([-(i*9+j+1), -(i*9+k+1)])
```

$$\bigwedge_{i=0}^{80} \bigwedge_{0 \le j < k \le 8} (\neg P(i,j) \ \vee \ \neg P(i,k))$$

The given code generates the clauses that enforce the condition that each cell of the Sudoku grid contains at most one number. The loop variable i ranges from 0 to 80, representing the 81 cells of the board. For each cell, the indices j and k range over the possible numbers 0 to 8, with the restriction $j < k$ to avoid repetition. For every such pair, the clause

$$\neg P(i, j+1) \ \vee \ \neg P(i, k+1)$$

is added to the CNF. This guarantees that no two distinct numbers can be simultaneously assigned to the same cell.

Since we have an atleast one occurence in each row, col and cell for all numbers between 1-9 and we have an atmost one number per box condition we have the suffucuent conditions and need not att the condition of atleast one number per box.

Member Contributions

This question was independently solved and coded by both team members with similar logic and runtime but just slight differences. Then we discussed and optimized the encoder to give the final submission.

Question 2

1.1 Padding used and `self.grid` was modified as follows:

```
self.grid = grid
arr = np.array(grid)
padded = np.pad(arr, pad_width=1, mode='constant',
   constant_values='#')
self.grid = padded.tolist()
```

In our sokoban encoder class, we have padded the grid with walls (#) on all sides of the grid. This is done to account for boundary checks while checking for valid moves. This allowed us to club out of bounds and wall checks into a single check for walls (#).

Consequently, the number of rows and columns were also updated to account for the padded walls.

```
self.rows = len(self.grid)  # here I changed the definition of
   rows from grid to self.grid so that the padded rows are taken
   into account.
self.cols = len(self.grid[0])
```

Also, I changed variables `N` and `M` to `rows` and `cols` respectively for better readability.

1.2 Variable Encoding used

We have used the following variable encoding for the given problem:

In `q2.py` file, we defined 3 new class members:

```
self.c1 = self.num_boxes + 1
self.c2 = self.c1 * self.rows
self.c3 = self.c2 * self.cols
```

where `num_boxes` = number of boxes, `rows` = number of rows, `cols` = number of columns.

To encode a player at position (x, y) at time t:

```
return self.num_boxes + 1  + self.c1*x + self.c2*y + self.c3 * t
```

In the rest of the report, we will use $P(x, y, t)$ to denote the above encoding.

To encode a box with index b at position (x, y) at time t:

```
return b  + self.c1*x + self.c2*y + self.c3 * t
```

This ensures no overlap between player and box encodings. In the rest of the report, we will use $B(b, x, y, t)$ to denote the above encoding.

Grid parsing is done by checking the grid cell values and updating `self.goals`, `self.boxes` and `self.player` accordingly.

1.3 Explaining the constraints

```
# Constraints for initial conditions:
self.cnf.append([self.var_player(self.grid_player_start[0], self.
    grid_player_start[1],0)])
for index in range(self.num_boxes):
    self.cnf.append([self.var_box(index+1, self.boxes[index][0],
        self.boxes[index][1], 0)])
```

These constraints ensure that the initial positions of the player and boxes are correctly represented in the CNF formula at time step 0.

We add `P(player_start[0], player_start[1], 0)` to the CNF to indicate the player's starting position.

For each box, we add `B(index+1, boxes[index][0], boxes[index][1], 0)` to represent the initial position of each box.

```
# Condition for player movement:
for t in range(self.T):
    for x in range(1, self.rows-1):
        for y in range(1, self.cols-1):
            self.cnf.append([
                -self.var_player(x, y, t),
                self.var_player(x+1, y, t+1),
                self.var_player(x-1, y, t+1),
                self.var_player(x, y+1, t+1),
                self.var_player(x, y-1, t+1),
                self.var_player(x, y, t+1)
            ])
```

This constraint ensures that if the player is at position (x, y) at time t, then at time $t+1$, the player must move to one of the adjacent positions $(x+1, y), (x-1, y), (x, y+1), (x, y-1)$ or remain in the same position (x, y).

Using logical notation, we can express this as:

$$P(x,y,t) \implies P(x{+}1,y,t{+}1) \vee P(x{-}1,y,t{+}1) \vee P(x,y{+}1,t{+}1) \vee P(x,y{-}1,t{+}1) \vee P(x,y,t{+}1)$$

Encoded in CNF form, this becomes:

$$\neg P(x,y,t) \vee P(x{+}1,y,t{+}1) \vee P(x{-}1,y,t{+}1) \vee P(x,y{+}1,t{+}1) \vee P(x,y{-}1,t{+}1) \vee P(x,y,t{+}1)$$

This ensures that the player moves to a valid adjacent cell or stays in place at each time step.

```
# Condition that if there is a wall, player and boxes can't move
   through it
for t in range(self.T + 1):
    for x in range(self.rows):
        for y in range(self.cols):
            if self.grid[x][y] == '#':
                self.cnf.append([-self.var_player(x, y, t)])
                for b in range(self.num_boxes):
                    self.cnf.append([-self.var_box(b+1, x, y, t)
                       ])
```

This constraint ensures that neither the player nor any box can occupy a wall cell at any time step t.

- For each wall cell (x, y) where grid[x][y] is a '#', we add

$$\neg P(x, y, t)$$

  to the CNF, indicating that the player cannot be at that position at time t.

- Similarly, for each box b, we add

$$\neg B(b, x, y, t)$$

  to indicate that box b cannot be at the wall position (x, y) at time t.

Since the grid has been padded with walls, this also automatically handles out-of-bounds checks.

```
# Condition that at a given time, a player and box can be at most
    in one place
for t in range(self.T + 1):
    atleast_one_place_player = []  # condition that at a given
       time player must be in at least one place
    for x in range(self.rows):
        for y in range(self.cols):
            atleast_one_place_player.append(self.var_player(x, y,
                t))
            for xi in range(self.rows):
                for yi in range(self.cols):
                    if (x, y) != (xi, yi):
                        self.cnf.append([-self.var_player(x, y, t
                           ), -self.var_player(xi, yi, t)])
                        for b in range(self.num_boxes):
                            self.cnf.append([-self.var_box(b+1, x
                               , y, t), -self.var_box(b+1, xi, yi
                               , t)])
    self.cnf.append(atleast_one_place_player)

# Condition that at a given time, a box must be in at least one
   place
for t in range(self.T + 1):
    for b in range(self.num_boxes):
        atleast_one_place_box = []
        for x in range(self.rows):
            for y in range(self.cols):
                atleast_one_place_box.append(self.var_box(b+1, x,
                    y, t))
        self.cnf.append(atleast_one_place_box)
```

This ensures that at any given time step t, the player and each box occupy exactly one position on the grid.

Logical Representation

Player must be in at least one place:

$$\forall t, \quad P(x_1, y_1, t) \vee P(x_2, y_2, t) \vee \cdots \vee P(x_n, y_n, t)$$

where $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ are all valid positions on the grid.

Each box must be in at least one place:

$$\forall t \; \forall b, \quad B(b, x_1, y_1, t) \vee B(b, x_2, y_2, t) \vee \cdots \vee B(b, x_n, y_n, t)$$

where $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ are all valid positions on the grid for box b.

Player must be in at most one place:

$$\forall t, \forall (x_i, y_i) \neq (x_j, y_j), \quad \neg P(x_i, y_i, t) \vee \neg P(x_j, y_j, t)$$

Each box must be in at most one place:

$$\forall t, \forall b, \forall (x_i, y_i) \neq (x_j, y_j), \quad \neg B(b, x_i, y_i, t) \vee \neg B(b, x_j, y_j, t)$$

These constraints guarantee that a player and a box can occupy exactly one place at a time.

```
# Condition that no cell can have more than one box or both a
   player and a box
for t in range(self.T + 1):
    for x in range(self.rows):
        for y in range(self.cols):
            for b1 in range(self.num_boxes):
                # Player and box cannot occupy the same cell
                self.cnf.append([-self.var_player(x, y, t), -self
                   .var_box(b1+1, x, y, t)])
                for b2 in range(b1 + 1, self.num_boxes):
                    # Two boxes cannot occupy the same cell
                    self.cnf.append([-self.var_box(b1 + 1, x, y,
                       t), -self.var_box(b2 + 1, x, y, t)])
```

This constraint ensures that no cell on the grid can contain more than one box or both a player and a box at the same time step t.

- For each cell (x, y) and each box b_1, we add:

$$\neg P(x, y, t) \vee \neg B(b_1, x, y, t)$$

to indicate that if the player is at (x, y) at time t, then box b_1 cannot occupy the same position at the same time.

- For each pair of boxes b_1 and b_2, we add:

$$\neg B(b_1, x, y, t) \vee \neg B(b_2, x, y, t)$$

to ensure that two boxes do not occupy the same cell simultaneously.

Logical Notation

$$\forall t, \forall (x, y), \forall b_1 : \quad \neg P(x, y, t) \vee \neg B(b_1, x, y, t)$$

$$\forall t, \forall (x, y), \forall b_1 \neq b_2 : \quad \neg B(b_1, x, y, t) \vee \neg B(b_2, x, y, t)$$

```
# Condition about moving boxes
for t in range(self.T):
    for b in range(self.num_boxes):
        for x in range(1, self.rows+1):
            for y in range(1, self.cols+1):
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x+1, y, t), self.var_box(b+1, x
                    +1, y, t+1), self.var_box(b+1, x+2, y, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x+1, y, t), self.var_box(b+1, x
                    +1, y, t+1), self.var_player(x+1, y, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x-1, y, t), self.var_box(b+1, x
                    -1, y, t+1), self.var_box(b+1, x-2, y, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x-1, y, t), self.var_box(b+1, x
                    -1, y, t+1), self.var_player(x-1, y, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x, y+1, t), self.var_box(b+1, x,
                     y+1, t+1), self.var_box(b+1, x, y+2, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x, y+1, t), self.var_box(b+1, x,
                     y+1, t+1), self.var_player(x, y+1, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x, y-1, t), self.var_box(b+1, x,
                     y-1, t+1), self.var_box(b+1, x, y-2, t+1)])
                self.cnf.append([-self.var_player(x, y, t), -self
                    .var_box(b+1, x, y-1, t), self.var_box(b+1, x,
                     y-1, t+1), self.var_player(x, y-1, t+1)])
                self.cnf.append([-self.var_box(b+1, x, y, t),
                    self.var_player(x+1, y, t), self.var_player(x
                    -1, y, t), self.var_player(x, y-1, t), self.
                    var_player(x, y+1, t), self.var_box(b+1, x, y,
                     t+1)])
```

This constraint ensures that a box can only be moved if the player is adjacent to it and moves into its position, pushing the box to the next cell in the same direction.

Logical Notation

$$B(b,x+1,y,t) \wedge P(x,y,t) \implies \big(B(b,x+2,y,t+1) \wedge P(x+1,y,t+1)\big) \vee B(b,x+1,y,t+1)$$

This means if box b is at $(x+1,y)$ at time t and the player is at (x,y) at time t, then at time $t+1$, either: - The box moves to $(x+2,y)$ and the player moves to $(x+1,y)$, or - The box stays at $(x+1,y)$.

Encoding this in CNF gives:

$$\neg P(x,y,t) \vee \neg B(b,x+1,y,t) \vee \big(B(b,x+2,y,t+1) \wedge P(x+1,y,t+1)\big) \vee B(b,x+1,y,t+1)$$

Expanding the ANDs:

$$\neg P(x,y,t) \lor \neg B(b,x+1,y,t) \lor B(b,x+2,y,t+1) \lor B(b,x+1,y,t+1)$$

$$\neg P(x,y,t) \lor \neg B(b,x+1,y,t) \lor P(x+1,y,t+1) \lor B(b,x+1,y,t+1)$$

Similarly, we encode the other directions:

$$\begin{aligned}
B(b,x-1,y,t) \land P(x,y,t) &\implies \big(B(b,x-2,y,t+1) \land P(x-1,y,t+1)\big) \lor B(b,x-1,y,t+1) \\
B(b,x,y+1,t) \land P(x,y,t) &\implies \big(B(b,x,y+2,t+1) \land P(x,y+1,t+1)\big) \lor B(b,x,y+1,t+1) \\
B(b,x,y-1,t) \land P(x,y,t) &\implies \big(B(b,x,y-2,t+1) \land P(x,y-1,t+1)\big) \lor B(b,x,y-1,t+1)
\end{aligned}$$

Finally, if the player is not adjacent to the box, the box remains in the same position:

$$B(b,x,y,t) \land \neg\big(P(x+1,y,t) \lor P(x-1,y,t) \lor P(x,y+1,t) \lor P(x,y-1,t)\big) \implies B(b,x,y,t+1)$$

In CNF:

$$\neg B(b,x,y,t) \lor P(x+1,y,t) \lor P(x-1,y,t) \lor P(x,y+1,t) \lor P(x,y-1,t) \lor B(b,x,y,t+1)$$

```
# Box should be at some goal at t = T
for b in range(self.num_boxes):
    possible = []
    for goal_index in self.goals:
        possible.append(self.var_box(b+1, goal_index[0],
           goal_index[1], self.T))
    self.cnf.append(possible)
```

This constraint ensures that at the final time step T, each box must be on a goal position.

For each box b, we create a clause that is the **disjunction of all possible goal positions** for that box at time T. This means that box b must be at least one of the goal positions at time T.

Logical Notation

$$\forall b \bigvee_{(x_g,y_g)\in \text{Goals}} B(b,x_g,y_g,T)$$

where (x_g, y_g) are the coordinates of all goal positions.

1.4 Decoding the model to get the sequence of moves

```
    sequence = []
    positions = []
    for literal in model :
        if literal > 0 :
            value = literal % encoder.c1
            if value == 0:
                value = encoder.c1
            row = int(((literal - value)%encoder.c2)/encoder.c1)
            col = int(((literal - encoder.c1*row - value)%encoder
               .c3)/encoder.c2)
            # time = int((literal - encoder.c2*col - encoder.c1 *
                row - value)/encoder.c3)
            if value == encoder.num_boxes + 1:
                positions.append([row,col])
                if len(positions) > 1:
                    for key in DIRS.keys():
                        if DIRS[key] == ((positions[-1][0] -
                           positions[-2][0]),(positions[-1][1] -
                           positions[-2][-1])):
                            sequence.append(key)
                            break
    #print(sequence)
    return sequence
```

This function decodes the model returned by the SAT solver to extract the sequence of moves made by the player. This basically finds out which literals are true and extracts the player/box positions at each time step. looking at the player's movements it decides the sequence of moves. If our model is UNSAT we return -1 and never call decode.

Explanation of the Code

1. We initialize an empty list `sequence` to store the moves and an empty list `positions` to track the player's positions over time.
2. We iterate through each literal in the model. If the literal is positive,we decode it first to find out of it's a player or box variable by taking remainder with our first constant `c1`.
3. Important thing to note here is that if remainder = 0 that means our value was `c1` so we set value = `c1`.
4. Now once we know it's a player or box variable we decode the row and column using the constants `c1`, `c2`, and `c3`.
5. If the value corresponds to the player variable (`num_boxes + 1`), we append the (row, col) position to the `positions` list.
6. If there are at least two positions in the `positions` list, we calculate the difference between the last two positions to determine the direction of movement.
7. We use a predefined dictionary `DIRS` to map the position difference to a move direction ('Up', 'Down', 'Left', 'Right') and append to `sequence`.
8. Finally, we return the `sequence` list containing the series of moves made by the player.

1.5 Member Contributions

Firstly, we both individually derived the CNF clauses for the constraints and then discussed them together to finalize the constraints and their logical representations. Then we distributed the coding work as follows:

- Sagar V implemented the variable encoding, initial conditions, player movement constraints,single occupancy constraints, and wall constraints.
- Tamanna Kumari implemented the grid padding, wall constraints, no overlap constraints, box movement constraints, goal conditions, and the decoding function.
- Both of us collaborated on testing and debugging, coming up with additional testcases and experimenting with different values of rows, cols, T etc.