CS228 Assignment-1
Project Report

Boda Prabanjan Jadav
24B1001
Abhinav V
24B1007

Contents

1 Introduction

This pdf contains report for the assignment given under course CS228: Logic in Computer Science; We were given two questions which were solved using PySat (Library in Python).We learnt basics of how to use PySat in solving Puzzle questions : Sudoku,Sokoban. These questions were solved collectively by both of us:

Abhinav V - 24B1007
Boda Prabanjan Jadav - 24B1001

2 Question-1 Sudoku Solver

This is an introduction to using PySAT for solving a puzzle like Sudoku (9x9). We have a total of 500 test cases, each a different Sudoku. In the input, the given numbers are provided, and the unknown positions are represented by ".". Using the methodology taught in class, we implement the following important conditions:

Conditions:
Condition 1: *Each row must contain all 1-9 values.*
Condition 2: *Each column must contain all 1-9 values.*
Condition 3: *Each **sub-matrix (3×3)** must contain all 1-9 values.*
Condition 4: *Placing the values which are already provided and performing operations accordingly.*
Condition 5: *Each cell must contain only one value.*

As explained in class, we created a clause with variables representing the row, column, and value, denoted as i, j, n respectively. Let us represent as P(i,j,n) which when it is true the number n is present in the i^{th} row and j^{th} column of the grid. The `index_map` list stores the mapping between (i,j,n) to unique values. Key of index_map is the string f"ijn" and its value is $i * 100 + j * 10 + n$, this is unique as all are single digit. List value_index is the reverse bijection of index_map

2.1 Condition 1

Checking correctness across all rows is done using the loop where we assign a new list `row_fill` for every row, which fills all possible values in the row.So this is basically conjuncting of cases involving such that the

$$\bigwedge_{i=0}^{8}\left(\bigwedge_{n=1}^{9}\left(\bigvee_{j=0}^{8} P(i,j,n)\right)\right)$$

It is doing like this First we fix a value of n like 1 or something and we iterate over all j such that n must be as some element of the column by iterating over n we make sure all numbers are also present in that row and iterating over i makes sure this for every row in the grid. To be in short we can say for any row i and number n, there exists some j where P(i,j,n) is true. This makes sure that a row contains all numbers from 1 to 9.We add these encoding of variable to list row_fill(row_fill.append(index_map[f"ijn"]) then append this to cnf for every row after iterating from j=0 to 9.

2.2 Condition 2

Checking correctness across all columns is done similarly with `column_fill`, which collects all possible values in a column.like first we fix a value like n=1 or something and j such a way that n=1 must be in either of i=0 to 8.later we apply the same logic for all n .And we iterate over all columns by iterating over j and by conjuncting we get

$$\bigwedge_{j=0}^{8}\left(\bigwedge_{n=1}^{9}\left(\bigvee_{i=0}^{8}P(i,j,n)\right)\right)$$

To be in short we can say for any column j and number n, there exists some i where P(i,j,n) is true. This makes sure that a column contains all numbers from 1 to 9. We add these encoding of variable to list column_fill(column_fill.append(index_map[f"ijn"]) then append this to cnf for every column after iterating from i=0 to 9.

2.3 Condition 3

For the each 3×3 sub-matrices in the grid, we define two variables k and l to represent the location of the sub-matrix. That is we divide the grid as 3×3 submatrices of 3×3. For example, $k = 0, l = 0$ represents the first sub-matrix (i.e., $i = 0$ to $i = 2$ and $j = 0$ to $j = 2$). We then define `box_fill`, which stores variables encoding for any number n that can lie on any cell of that sub-matrix.

$$\bigwedge_{k=0}^{2}\left(\bigwedge_{l=0}^{2}\left(\bigwedge_{n=1}^{9}\left(\bigvee_{j=0}^{3}\left(\bigvee_{i=0}^{3}P(i+3*k,j+3*l,n)\right)\right)\right)\right)$$

At first we fix the k , l values later we iterate over every element of box by the same process we did above .we fix some n value and make sure it is present in the sub matrix by iterating over i,j and later iterate over all n .Finally iterating over k,l makes sure it occurs for all sub matrices . All these encodings are appended to list box_fill$(box_fill.append(index_map[f"(i+3*k)(j+3*l)(n)"]))$ for each sub-matrix then appended to cnf.

2.4 Condition 4

Next, we check the existing values from the grid. For example, `grid[0][0]` represents the first element of the first row in the input case. **Since our range of valid values is 1–9 *Only Valid ones are taken*** , this corresponds to the first number.

$$\bigwedge_{i,j,n}P(i,j,n)$$

Like these are for provided values of n (given in question), we iterate over the grid and append encoding of P(i,j,n) to cnf if n is equal to grid[i][j].

2.5 Condition 5

Now we ensure that each cell must have only one number. For this, we define a new list `atmost_one`. In this case, for each possible pair of values (n_1, n_2), if n_1 is set to True in the CNF, then n_2 must be False. This ensures that only one value can be assigned to a cell(i,j). **Note :** *"-" is used to represent negation in the code*

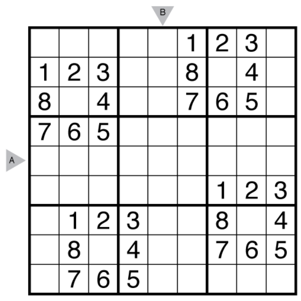
$$\bigwedge_{i=0}^{8}\left(\bigwedge_{j=0}^{8}\left(\bigwedge_{n1=1}^{9}\left(\bigwedge_{n2=n1+1}^{9}\neg P(i,j,n1,n2)\right)\right)\right)$$

Finally, we solve the CNF using the `solve()` function. The result is stored in the variable `model`. For each element in the model, if it is > 0, we map it back using `value_index` to obtain the corresponding i, j, n values by converting them back to int from string. We then construct the answer(List of list of integers, $9 * 9$) grid by setting `answer[i][j] = n`.

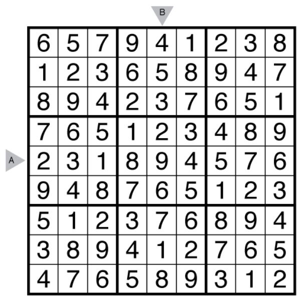
$$C1 \bigwedge C2 \bigwedge C3 \bigwedge C4 \bigwedge C5$$

Note: For clear references of what each list stores, refer to the table below.

List Name	What it stores
`index_map`	Mapping of (i, j, n) triples to unique integer variables
`value_index`	Reverse mapping of integer variables back to (i, j, n)
`row_fill`	Ensures every row contains all 9 values
`column_fill`	Ensures every column contains all 9 values
`box_fill`	Ensures every 3×3 box contains all 9 values
`atmost_one`	Ensures each cell has only one value



					1	2	3	
1	2	3			8		4	
8		4			7	6	5	
7	6	5						
						1	2	3
	1	2	3			8		4
	8		4			7	6	5
	7	6	5					



6	5	7	9	4	1	2	3	8
1	2	3	6	5	8	9	4	7
8	9	4	2	3	7	6	5	1
7	6	5	1	2	3	4	8	9
2	3	1	8	9	4	5	7	6
9	4	8	7	6	5	1	2	3
5	1	2	3	7	6	8	9	4
3	8	9	4	1	2	7	6	5
4	7	6	5	8	9	3	1	2

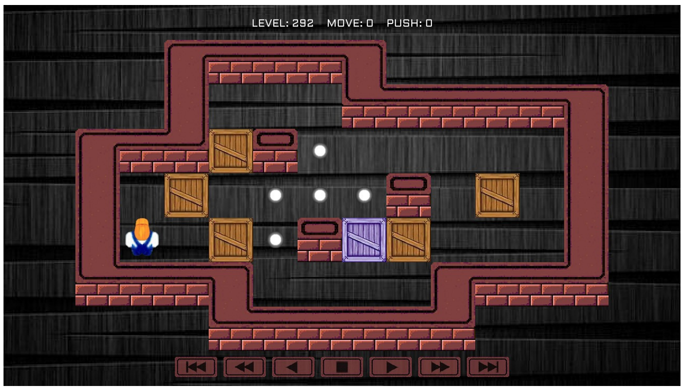
3 Question-2 (Sokoban Puzzle)

This is a puzzle usually played by children as a video game. The rules of the game are simple, there will be a Player('P') who can move only one unit per move like Up('U'), Down('D'), Left('L'), Right('R') on a given rectangular Grid($N * M$). The game expects the player to push the Boxes('B') to the marked places also called as Goals('G') within a specified number of Moves('T'). There are also constraints on the game like some places on the grid has obstacles like Wall('#') thus player can't go there hence the boxes too. Here the player can't push two boxes at a move. Let us denote the empty spaces where player can move as '.' and solve this puzzle using our SAT solvers by giving clauses to it. Number of moves is given as first line of input and all other lines represent the each row of grid using above encoding.

3.1 Sokoban Encoder

SokobanEncoder is the class in q2.py. It has some members like

1. **grid** : This is a list of list of strings. Where each list are the rows of the grid and each list contains strings like ('P', 'G', 'B', '.', '#'). Thus, grid[i][j] will say the condition of i^{th} row and j^{th} column.
2. **T** : This say the maximum number of moves that a player can move to push the boxes to the goal.



3. **N** : It is the number of rows in the grid.
4. **M** : It is the number of columns in the grid.
5. **goals** : It is the list of tuples containing the coordinates of the goal in the grid.
6. **boxes** : It is the list of tuples containing the box number and the coordinates of the goal in the grid.
7. **player_start** : It is a tuple saying the position of the player at time 0.
8. **num_boxes** : It is the number of boxes in the grid.
9. **max_player_value** : It is the maximum possible value of the encoding to represent player's variable.

   self.max_player_value $= \big(((self.N-1)(self.N+1)+(self.M-1))(self.T+1)+(self.T-1)\big)+1$

Other than these the class also have some useful functions, they are

- def __init__(self, grid, T):
- def _parse_grid(self):
- def var_player(self, x, y, t):
- def var_box(self, b, x, y, t):
- def encode(self):

The function def __init__(self, grid, T): initialises the member variables said above. We handle all members of class by outside two functions def decode(model, encoder): and def solve_sokoban(grid, T):(Given)

3.2 def _parse_grid(self):

This function parses the given grid and update members of class like boxes, goals, player_start. We traverse the grid by iterating i from 0 to N and j from 0 to M, in nested for loops to cover all places in the grid. Thus, grid[i][j] will say the condition of i^{th} row and j^{th} column.
Cases :

- **grid[i][j]** $=='$ **B**$'$: Thus in i^{th} row and j^{th} column there is a box hence we append the Tuple(b,i,j) {coordinates with box number(b)} to the list boxes. Then we update the count of boxes(b) by incrementing 1 to it.

- **grid[i][j] ==' G'** : Thus in i^{th} row and j^{th} column there is a goal hence we append the Tuple(i,j) {coordinates of the goal} to the list goals.
- **grid[i][j] ==' P'** : Thus player starts from i^{th} row and j^{th} column. Thus, player_start = (i,j).

3.3 def var_player(self, x, y, t):

The player can be in x^{th} row and y^{th} column at time t, hence we can represent this as a propositional variable P(x,y,t). If P(x,y,t) is true then the player is at x^{th} row and y^{th} column at time t and if it is false then the player is not at that position at time t. This statement is also either way true(Viceversa). To represent these player variables in the clauses we want a specific and unique encoding bijection from P(x,y,t) to a unique number. This number is returned by this function.

$$\text{Player ID} = ((x * (self.N + 1) + y) * (self.T + 1) + t) + 1$$

This is mixed-radix encoding based on the range of the variables x,y and t. As we know $x < N$, $y < M$ and $t < T$, hence we made a unique encoding based on their ranges. Thus the maximum possible encoding will be:

$$\text{self.max\_player\_value} = \big(((self.N-1)(self.N+1)+(self.M-1))(self.T+1)+(self.T-1)\big)+1$$

3.4 def var_box(self, b, x, y, t):

A box b(box number) can be in x^{th} row and y^{th} column at time t, hence we can represent this as a propositional variable P(b,x,y,t). If P(b,x,y,t) is true then the box b is at i^{th} row and j^{th} column at time t and if it is false then the box is not at that position at time t. This statement is also either way true(Viceversa). To represent these box variables in the clauses we want a specific and unique encoding bijection from P(b,x,y,t) to a unique number. This number is returned by this function.

$$\text{Box ID} = (((x*(self.N+1)+y)*(self.T+1)+t)*(self.num\_boxes+1)+b)+self.max\_player\_value+1$$

This is mixed-radix encoding based on the range of the variables b,x,y and t. As we know $b <= num\_boxes$, $x < N$, $y < M$ and $t < T$, hence we made a unique encoding based on their ranges. Here we add max_player_value so that the minimum value of box ID lie out of the range of the player ID to ensure difference between both. This is the reason for using a extra variable in the class.

3.5 def encode(self):

This function returns self.cnf which contain clauses to represent the game rules and constraints. These contraints include:

- Initial conditions
- Player movement
- Box movement (push rules)
- Non-overlap constraints
- Goal conditions
- Other conditions

3.5.1 Initial conditions

1. This includes adding the player, box and goal positions at time 0. Initially at time t=0, player is at position at player_start. Thus, we find its encoding and append its value to cnf. Its encoding can be found by self.var_player(self.player_start[0],self.player_start[1],0).

$$P(self.player\_start[0], self.player\_start[1], 0)$$

2. Then we set the every box b positions at time t=0 to cnf by appending their encodings to cnf. This we do for every tuple in boxes i.e for every boxes. Thus, their encoding will iterate through boxes and append clause [self.var_box(b[0],b[1],b[2],0)].

$$\bigwedge_{\text{b in boxes}} P(b[0], b[1], b[2], k)$$

3.5.2 Player movement

Conditions for player movement by game rules are:

1. As we know there is only one player throughout the game, the sat solver must not make duplicate player by making more than one variables of players at time t=k to be true. Thus for any two pair of coordinates (i,j) at any time t=k, only one player variable must be true. This logic is similar to atleast_player and atmost_player one variable must be true at any time t=k. Atleast cases is a repeated constraint of the following condition hence we add atmost case here. This could be represented as:

$$\bigwedge_{i=0}^{N-1}\bigwedge_{j=0}^{M-1}\bigwedge_{k=0}^{T}\left(\bigwedge_{p=0}^{N-1}\bigwedge_{\substack{q=0\\(p,q)\neq(i,j)}}^{M-1}(\neg P(i,j,k)\vee\neg P(p,q,k))\right)$$

Thus above formula can be done by nested 5 for loops over i,j,k,p and q for the above said limits. Replacing the variables with encoding gives the clause as [(-self.var_player(i,j,k)), (-self.var_player(p,q,k))].

2. By rules of game we know that a player can move only one square at time t, thus if a player is at (i,j) at time t=k-1 he will be at any one of [(i,j),(i-1,j),(i,j-1),(i+1,j),(i,j+1)] at time t=k for all $k >= 1$. This can be written as:

$$\bigwedge_{i=0}^{N-1}\bigwedge_{j=0}^{M-1}\bigwedge_{k=1}^{T}(\neg P(i,j,k-1)\vee P(i,j,k)\vee P(i-1,j,k)\vee P(i,j-1,k)\vee P(i+1,j,k)\vee P(i,j+1,k))$$

The above said clause gives that atleast one of that 5 possible moves is true. As by condition 1 above, a player can't be in two boxes at time t=k, the above equation makes only one out of that 5 to be true. The clause for this is made case wise based on the current value of i and j to ensure that the coordinates doesn't go outside of the grid. This could be handled by making a list next_mov which contains only the possible moves for the given (i,j) and then append this to cnf. For any (i,j) a player can stay there for next instant also so,
next_mov = [(-(self.var_player(i,j,k-1))), (self.var_player(i,j,k))]

- If $i < N-1$: `next_mov.append`(var_player$(i+1, j, k)$)
- If $i > 0$: `next_mov.append`(var_player$(i-1, j, k)$)
- If $j > 0$: `next_mov.append`(var_player$(i, j-1, k)$)
- If $j < M-1$: `next_mov.append`(var_player$(i, j+1, k)$)

3.5.3 Box movement (push rules)

Based on the rules of the game, any box b at (i,j) at time t=k-1 will move to (i+dx,j+dy) at time t=k only when the player moves from (i-dx,j-dy) at time t=k-1 to (i,j) at time t=k.This statement is also true in backward direction (Viceversa). Here dx=(-1,0,1) dy=(-1,0,1) where their both absolute values are not 1. Based on (i,j) position of the box we have box movement contraints which is handled by if cases, we can also handle this by adding walls around the grid in that case we need to ensure the encoding is not negative or index the grid from 1 hence I choose this for minimum changes in my code.
Cases :

1. (i,j)==(0,0) or (i,j)==(0,M-1) or (i,j)==(N-1,0) or (i,j)==(N-1,M-1)
   The above coordinates are the dead end corner for boxes in the grid as the player can't push box out of grid and can move only in x or y direction. Thus if any box b is at (i,j) at time t=k-1 stays there (i,j) at time t=k also regardless of whatever player movement is. This is represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee P(b,i,j,k))\right)\right)\right)$$

   Thus the clause will be [(-(self.var_box(p,i,j,k-1))), (self.var_box(p,i,j,k))]

2. $i > 0$ and $i < self.N - 1$
   For all above values of i any box b can move up or down without moving out of grid.

   - Box moving Up
     If any box b is at (i,j) at time t=k-1 moves to (i+1,j) at time t=k if and only if player moves from (i-1,j) at time t=k-1 to (i,j) at time t=k. Which can be said as If B is at (i,j), P is at (i-1,j) at time t=k-1 and P is at (i,j) at time t=k then B will be at (i+1,j) at time t=k. which is represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i-1,j,k-1)\vee\neg P(i,j,k)\vee P(b,i+1,j,k))\right)\right)\right)$$

     Applying 4 nested for loops for i,j,k and b we append clause [(-self.var_box(p,i,j,k-1)), (-self.var_player(i-1,j,k-1)), self.var_box(p,i+1,j,k), (-self.var_player(i,j,k))] to cnf.

   - Box moving Down
     If any box b is at (i,j) at time t=k-1 moves to (i-1,j) at time t=k if and only if player moves from (i+1,j) at time t=k-1 to (i,j) at time t=k. Which can be said as If B is at (i,j), P is at (i+1,j) at time t=k-1 and P is at (i,j) at time t=k then B will be at (i-1,j) at time t=k. which is represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i+1,j,k-1)\vee\neg P(i,j,k)\vee P(b,i-1,j,k))\right)\right)\right)$$

     Applying 4 nested for loops for i,j,k and b we append clause [(-self.var_box(p,i,j,k-1)), (-self.var_player(i+1,j,k-1)), self.var_box(p,i-1,j,k), (-self.var_player(i,j,k))] to cnf.

3. $j > 0$ and $j < self.M - 1$
   For all above values of j any box b can move right or left without moving out of grid.

   - Box moving Right
     If any box b is at (i,j) at time t=k-1 moves to (i,j+1) at time t=k if and only if player

moves from (i,j-1) at time t=k-1 to (i,j) at time t=k. Which can be said as If B is at (i,j), P is at (i,j-1) at time t=k-1 and P is at (i,j) at time t=k then B will be at (i,j+1) at time t=k. which is represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i,j-1,k-1)\vee\neg P(i,j,k)\vee P(b,i,j+1,k))\right)\right)\right)$$

Applying 4 nested for loops for i,j,k and b we append clause [(-self.var_box(p,i,j,k-1)), (-self.var_player(i,j-1,k-1)), self.var_box(p,i,j+1,k), (-self.var_player(i,j,k))] to cnf.

- Box moving Left
  If any box b is at (i,j) at time t=k-1 moves to (i,j-1) at time t=k if and only if player moves from (i,j+1) at time t=k-1 to (i,j) at time t=k. Which can be said as If B is at (i,j), P is at (i,j+1) at time t=k-1 and P is at (i,j) at time t=k then B will be at (i,j-1) at time t=k. which is represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i,j+1,k-1)\vee\neg P(i,j,k)\vee P(b,i,j-1,k))\right)\right)\right)$$

  Applying 4 nested for loops for i,j,k and b we append clause [(-self.var_box(p,i,j,k-1)), (-self.var_player(i,j+1,k-1)), self.var_box(p,i,j-1,k), (-self.var_player(i,j,k))] to cnf.

We do have some contraints like the any box b shouldn't jump from(i,j) at time t=k-1 to any other square at t=k without player coming to the square (i,j) at time t=k. Which is represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee P(b,i,j,k)\vee P(i,j,k))\right)\right)\right)$$

The above clause says for any box b at (i,j) at time t=k-1, if P doesn't come to square (i,j) at time t=k then box remains there i.e (i,j) at time t=k. This would avoid unnecessary jumping of box from a square to next.

But this doesn't handle corner case if box was at the edges of grid, for ex: If B was at (0,1), P was at (1,1) and if P comes to (0,1) then B must persist there but above clause allows it to jump. However it puts constraint over middle boxes, hence we have to introduce conditions only for below edge cases:

- i==0 and j!=0 and j!=M-1
  When any box b at (i,j) at time t=k-1, P at (i+1,j) at time t=k-1, if P comes to (i,j) at t=k then it allows box to jump to avoid this we add:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i+1,j,k)\vee\neg P(i,j,k)\vee P(b,i,j,k))\right)\right)\right)$$

  This clause puts constraint over jumping of box if it was on the corner of the grid, thus we add clause [(-(self.var_box(p,i,j,k-1))), -(self.var_player(i+1,j,k-1)),(self.var_box(p,i,j,k)), -(self.var_player(i,j,k))] to the cnf.

- j==0 and i!=0 and i!=N-1
  When any box b at (i,j) at time t=k-1, P at (i,j+1) at time t=k-1, if P comes to (i,j) at t=k then it allows box to jump to avoid this we add:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i,j+1,k)\vee\neg P(i,j,k)\vee P(b,i,j,k))\right)\right)\right)$$

This clause puts constraint over jumping of box if it was on the corner of the grid, thus we add clause [(-(self.var_box(p,i,j,k-1))), -(self.var_player(i,j+1,k-1)),(self.var_box(p,i,j,k)), -(self.var_player(i,j,k))] to the cnf.

- i==N-1 and j!=0 and j!=M-1
  When any box b at (i,j) at time t=k-1, P at (i-1,j) at time t=k-1, if P comes to (i,j) at t=k then it allows box to jump to avoid this we add:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i-1,j,k)\vee\neg P(i,j,k)\vee P(b,i,j,k))\right)\right)\right)$$

  This clause puts constraint over jumping of box if it was on the corner of the grid, thus we add clause [(-(self.var_box(p,i,j,k-1))), -(self.var_player(i-1,j,k-1)),(self.var_box(p,i,j,k)), -(self.var_player(i,j,k))] to the cnf.

- j==0 and i!=0 and i!=N-1
  When any box b at (i,j) at time t=k-1, P at (i,j-1) at time t=k-1, if P comes to (i,j) at t=k then it allows box to jump to avoid this we add:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=1}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(b,i,j,k-1)\vee\neg P(i,j-1,k)\vee\neg P(i,j,k)\vee P(b,i,j,k))\right)\right)\right)$$

  This clause puts constraint over jumping of box if it was on the corner of the grid, thus we add clause [(-(self.var_box(p,i,j,k-1))), -(self.var_player(i,j-1,k-1)),(self.var_box(p,i,j,k)), -(self.var_player(i,j,k))] to the cnf. **Note :** We can make walls around grid but current encoding would give negative values to represent them.

3.5.4 Non-overlap constraints

1. A player and a box can't be in a same square at any time t=k. This could be done by iterating through all squares at time t=0 to time t=T for every pair of player and the box. So for given i,j and k the player and any box can't be together. This could be represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=0}^{T}\left(\bigwedge_{b=0}^{\text{num\_boxes}-1}(\neg P(i,j,k)\vee\neg P(b,i,j,k))\right)\right)\right)$$

   This could be done by nested 4 for loops iterating for each rows, columns, boxes and time. The variables could be found by the encoding of it by the above functions. Thus the clause will be [(-(self.var_player(i,j,k))), (-(self.var_box(p,i,j,k)))]

2. More than one boxes can't be in a same square at any time t=k. This could be done by iterating through all squares at time t=0 to time t=T for every pair of boxes. So for given i,j and k box (p,q) can't be together. This could be represented as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{j=0}^{M-1}\left(\bigwedge_{k=0}^{T}\left(\bigwedge_{p=0}^{\text{num\_boxes}-1}\left(\bigwedge_{q=p+1}^{\text{num\_boxes}-1}(\neg P(p,i,j,k)\vee\neg P(q,i,j,k))\right)\right)\right)\right)$$

   This could be done by nested 5 for loops iterating for each rows, columns, pair of boxes and time. The variables could be found by the encoding of it by the above functions. Thus the clause will be [(-(self.var_box(p,i,j,k))), (-(self.var_box(q,i,j,k)))]

3.5.5 Goal conditions

By rules of the game at time t=T or before the boxes have to be in any one of the goal. Thus for a box b to be in any one of the goals which can be represented as:

$$\bigwedge_{b=0}^{\text{num\_boxes}-1}\left(\bigvee_{i=0}^{\text{len(goals)}-1} P(b,\ \text{goals}[i][0],\ \text{goals}[i][1],\ T)\right)$$

This says a box b has to be in atleast one the goal. We have made constraints for more than one boxes can't be in one place at a time t and a box can't be at two squares hence it will ensure that a box will be at only one goal at the end of time t=T.Thus we iterate through b in boxes and for every b we iterate through goals to make above clause. For a box b, we append the encoding of the variable to list atleast_goal list and at end of goals iteration we append it to the cnf.

3.5.6 Other conditions

1. The given SAT can make duplicate boxes for any box b. To avoid this if for some (i,j), box b is at (i,j) at t=k then for any other pair of coordinates (p,q) the variable P(b,p,q,k) must be false. It can be written as:

$$\bigwedge_{i=0}^{N-1}\bigwedge_{j=0}^{M-1}\bigwedge_{k=0}^{T}\bigwedge_{b=0}^{num\_boxes-1}\left(\bigwedge_{p=0}^{N-1}\bigwedge_{\substack{q=0\\(p,q)\neq(i,j)}}^{M-1}(\neg P(b,i,j,k)\vee\neg P(b,p,q,k))\right)$$

   This say at any time t=k, for box b it will be present in atmost one square of the grid to avoid its duplicates. By the box-movements clauses we can see that a box can't disappear if it was at time t=0. Thus we append clause [(-self.var_box(b,i,j,k)), (-self.var_box(b,p,q,k))] to the cnf with iterating for every variable by for loop.

2. The player or box cannot move into a square where wall('#') is present.
   This could be done by iterating through the grid for wall and add constraint for them. This can be said as:

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{\substack{j=0\\grid[i][j]='\#'}}^{M-1}\left(\bigwedge_{k=0}^{T}(\neg P(i,j,k))\right)\right)$$

$$\bigwedge_{i=0}^{N-1}\left(\bigwedge_{\substack{j=0\\grid[i][j]='\#'}}^{M-1}\left(\bigwedge_{k=0}^{T}\left(\bigwedge_{b=0}^{num\_boxes-1}(\neg P(b,i,j,k))\right)\right)\right)$$

3.6 def decode(model, encoder)

Based on the solution of the SAT solver we get a list of integers i.e model here. If the integer is positive then the variable corresponding to it is assigned true else false otherwise. As we want the order of moves in which the player P move we filter the list model for the element m, $m >= 1$ and $m <= max\_player\_value$ it goes to list mov.
From this we need to order the moves based on the value of time t, from 1 to T and give a list of strings ('U', 'D', 'R', 'L') as the returned value. For this we start from player_start (t=0) then check whether the encodings of below variables are there in mov or not:

- P(i+1,j,1): Append 'U' to list moves.
- P(i-1,j,1): Append 'D' to list moves.

- P(i,j+1,1): Append 'R' to list moves.
- P(i,j-1,1): Append 'L' to list moves.
- P(i,j,1): Append nothing.

This process is repeated till t=T, then list moves is returned.

4 Contributions & Credits :

4.1 Prabanjan(24B1001) :

- Question-1 Sudoku Solver:
  - Made constraints over the row checking on the case such that each row must contain all elements (1-9)
  - Made constraints over the Column checking on the case such that each Column must contain all the elements (1-9)
  - Made constraints over for the checking thing like no box contains 2 elements that is if location is i,j then we must have only a unique value of n
  - Made constraints over for the given values by extracting them from the given Input test cases of the grid which has list of lists
  - Tried for some new input cases and checked them
  - Wrote most of the Latex Report for this Question
- Question-2 Sokoban Solver:
  - Designed unique encoding for player and box variables i.e P(i,j,k) and P(b,i,j,k), by completing var_player and var_box function.
  - Introduced conditions like no movement over wall, box must be on goal on time t=T. Added clauses that include the initial positions of boxes and player.
  - Checked the code thoroughly and tested it for corner and many testcases.

4.2 Abhinav(24B1007) :

- Question-1 Sudoku Solver:
  - Designed unique variable encoding P(i,j,n) where it is a bijection between string and a number. Also completed parse_grid function which fills list boxes,goals and player_start.
  - Made constraints over submatrices of the grid that it must contain numbers from 1 to 9.
  - Checked the code thoroughly and tested it for corner and many testcases.
- Question-2 Sokoban Solver:
  - Made constraints over player movements and box movements that they can move only a unit square.
  - Designed clauses that introduce constraints over non overlapping things i.e (Player,Box) and (Box,Box).
  - Introduced conditions that doesn't allow sat solver to make duplicate players and boxes, also ensured that they won't disappear.

- Applied sat solver to the cnf, extracted the move sequences by completing function decode.
- Designed the solver that it could run for any arbitary numbers of rows, columns, boxes and goals. Optimized the code to run and tested regressive cases
- Wrote most of the Latex Report for this Question