CS228 Assignment 1 report

Arnav Baranwal - 24B1099
Mayank Jain - 24B1024

August 2025

Contents

1 Question 1

1.1 Problem Overview

In this problem we were asked to develop a Sudoku-solver using propositional logic and SAT solvers. It required us to map the problem to propositional logic variables and also to map the constraints required for the problem into formulae in Conjunctive Normal Form. Once we are able to encode these formulae, a satisfying assignment can be found using SAT solvers from pysat library.

1.2 Approach overview

We assigned each cell a unique propositional variable based on its row number, column number and the value it contains. Then we translated rules and restrictions of solving Sudoku to CNF clauses and used SAT solver to find a possible solution. If a satisfiable assignment was found, we translated the propositional variables which were assigned true to find the solution in that case for the Sudoku puzzle, and returned the solved grid.

1.3 Variable encoding

1.3.1 what each variable refers to?

Each cell of solution of a Sudoku puzzle has unique triplet of row number(say i), column number(say j) and value in it(say k). We assigned the value $81 * i + 9 * j + k + 1$ to each cell which is a unique encoding for each cell, (we have taken 0 based indexing for rows and columns while encoding which can easily be converted to 1 based indexing while decoding, and decremented values of the cell by 1 for the sake of simplicity of encoding, and incremented all the cell values

by 1 after getting a solution and before returning the solved Sudoku grid).

1.3.2 Initial Constraints

We first encoded the values already present in the given puzzle by looping over cells of the grid and whenever we come across any non zero value we take its value after decrementing 1 (for sake of simplicity in mapping). So we can straightaway append these quantities to our CNF formula as these propositional variables must be true.

1.3.3 Constraints on each cell

We need to take care of two important constraints for each cell, that is, for each pair of i and j. First is that there is atleast some value in each cell. We achieved this by looping over i and j and for each value of i and j we made a list having values corresponding to each value of k, and appended it to our CNF. Second is that there are no two values in the same cell. This was achieved by looping over i and j, and introduced 2 variables k and k1, and looping over k and k1, we added the clause that atleast one of the triplets (i,j,k) and (i,j,k1) should we false whenever $k \neq k1$ and appended these clauses to the CNF.

1.3.4 Row constraints

Each row contains the numbers from 1 to 9 once and only once. For each value of i, we looped over k and for a chosen value of i and k, we looped over all j and made a list and appended this list to our CNF.

1.3.5 Column constraints

Each column contains the numbers from 1 to 9 once and only once. For each value of j, we looped over k and for a chosen value of j and k, we looped over all i and made a list and appended this list to our CNF.

1.3.6 Subgrid constraints

The grid can be seen as 9 blocks of size $3 * 3$. Each such block contains the numbers from 1 to 9 once and only once. The condition that each sub-block has all numbers from 1 to 9 will be sufficient given our constraints on the cell. We achieved this by moving in steps of 3 for each i and j and thus reaching a top left corner. Now for that subgrid and a chosen value of k(by looping on k), we looped over di and dj from 0 to 2 and added these values to i and j to cover whole subgrid for which we added a list to our CNF.

1.4 Decoding

On getting a satisfiable assignment of the input CNF translated from the sudoku puzzle, we decoded the variables which were assigned true to get the value stored at each cell of the grid of the solution of sudoku puzzle. We assigned these values to a grid named result and returned the grid as the solution.

2 Question 2

2.1 Problem Overview

This problem involved learning about Sokoban encoder in which a sequence of player's moves along a grid have to be encoded and the

problem is solved using a SAT solver and then to be decoded in order to get the player's moves.

2.2 Approach Overview

2.3 Parsing logic

By looping through row number and column number of initial grid, we updated a list of pairs of coordinates of goals, a list of triplets (box number, row number, column number) of boxes, and assigned the initial encoded value of player to "self.player_start".

2.4 Variable Encoding

Walls, empty cells and goals were not required to be encoded because they were fixed, so it could be known at any time if a cell contained a wall or goal by checking the character stored at the cell, and a cell which didn't have wall, goal, player or a box was an empty cell.

2.4.1 Player

A player can be represented by the row and column numbers (say, x and y respectively) of his position, and the time passed since beginning of game (say t). We assigned each such condition the value $M * N * t + M * x + y + 1$ where M is the total number of columns and N is the total number of rows. This value indicates that at time t, the player is at the cell (x,y).

2.4.2 Box

The boxes were indexed from 1 to N (N = number of boxes) to distinguish the boxes and encode them uniquely. Any arbitary box can be represented by its box number (say b), the row and column numbers

(say, x and y respectively) of its position, and the time passed since beginning of the game (say t). We assigned each such condition the value $(M * N * (T + 1)) * b + M * N * t + M * x + y + 1$ where T is the maximum time in which we have to finish the game, M is the total number of columns and N is the total number of rows. This value indicates that at time t, box number b is at the cell (x,y).

2.5 Constraints Encoding

2.5.1 Initial Constraints

For these, the initial box and player positions which are parsed by iterating over the grid are straightaway encoded for time 0 and appended to CNF.

2.5.2 Player Movement

When we look for the movement of the player, it can happen in all four directions and we are also assuming that the player may choose to stay at rest. So, we need a clause which ensures that if a player is at i, j at time t it implies that the player should be at i,j at t+1 or i+di, j+dj at t+1 where i+di and j+dj are increments for each direction which we can get by looping over the already given list of directions. One thing that needs to be taken care of is that player cannot move on a wall (we are allowing movement on a box as box pushing is handled later) and the cell on which player moves on is in the grid. Now, the implied statement can be converted into a disjunction by taking negation of the left side(which has only one literal representing that player is at i, j at time t) and adding it with rest of the literals on the right side of implication.

2.5.3 Box Movement

Box can't move unless pushed :-
Looping over b (index of box), for each b, we loop over t from 0 to T-1, and for each b and t, we loop over i and j and append the clause that either b'th box is not at (i,j) at time t, or player is at (i,j) at t+1 or b'th box is at (i,j) at time t+1 which basically says that until the player arrives at the box's position after 1 unit time, the box will remain at the same position after 1 unit time.

Movement of box when pushed :-
Looping over b (index of box), for each b, we loop over t from 0 to T-1, and for each b and t, we loop over i and j, and for all of b,t,i,j fixed we loop over the directions (dx,dy) from DIRS. Now that b,t,i,j,dx,dy are fixed, we initialise an empty list. If (i+dx,j+dy) is within the grid, we append literals representing player not being at (i,j) at time t, player not being at (i+dx,j+dy) at t+1, b'th box not being at (i+dx,i+dy) at time t and additionally we add the literal representing b'th box being at (i+2*dx,j+2*dy) at time t+1 if (i+2*dx,j+2*dy) is within grid. Now if list is not empty, we add this as a clause to CNF.
What we did actually represents that if player is at (i,j) at time t, and at (i+dx,j+dy) at t+1 then either no box was at (i,j) at t, else if a box was there which can be moved to (i+2*dx,j+2*dy), then it will be moved to (i+2*dx,y+2*dy) at time t+1. Note that non overlap constraints ensure that a box isn't pushed into a wall or another box.

2.5.4 Non Overlap Constraints

There are certain constraints which prevent the overlapping of objects at same place at same time and are very important to take care of.

They are as follows:

- box and wall cannot be in same cell.
- player and wall cannot be in same cell.
- Two boxes cannot be in same cell at same time.
- box and player cannot be in same cell at same time.

For the first two conditions that involve the wall, we just find out the wall in the grid by iterating over all positions and then append negation of the box or player being at that position at all times to our CNF. For the third condition, we iterate over all positions at all times and iterate on two different boxes and append the clause which has negation of 1st box being at that place or negation of 2nd box being at that place which will not let two boxes to be at same place at same time. For the last one, iterate over row number,column number,time and boxes, and append the clause having negation of player being at that place on that time or negation of box being at that place at that time.

2.5.5 Goal Conditions

Now, there are two constraints which help us in checking if the box reaches goal before T. If a box gets to a goal, we do not to need to move it further and this condition can be encoded by taking i and j from the parsed value of goals and then iterate over boxes and time and if a box is at the goal's position at time t it implies that it will be there at t+1 also. This can be appended in CNF as negation of box being at i,j at t or box being at i,j at t+1.
The other constraint is just this that every box must be at some goal at time T. We iterate over all the boxes and for each box, we make

a clause representing that the box is at some goal at time T and append such clauses for all boxes.

2.5.6 Other Conditions

Condition that a player can only be at one position at a given time: Looping over t from 0 to T, for each value of t, we loop 2 positions (i1,j1) and (i2,j2) over the whole grid and whenever the positions are not same, we append the clause that either player is not at (i1,j1) at time t, or player is not that (i2,j2) at time t.
Condition that a box can only be at one position at a given time: Looping over t from 0 to T and b over 1 to (number of boxes), for each value of t and b, we loop 2 positions (i1,j1) and (i2,j2) over the whole grid and whenever the positions are not same, we append the clause that either b'th box is not at (i1,j1) at time t, or b'th box is not at (i2,j2) at time t.

2.6 Decoding

Our task was to map player positions at each timestep to movement directions, if a solution was obtained. For this, we first translated the player_start from encoder back to coordinates to get initial position of player. We also initialised an empty string result to store the whole path. Because there's a solution we know that after each unit of time till T, player will either remain at same position or move 1 unit in 1 direction. If player doesn't move then we don't need to add anything to result. Looping t over 0 to T, for each value of t, we loop over (dx,dy) in DIRS, and if the literal corresponding to player being at (i+dx,j+dy) at t+1 is true and (i+dx,j+dy) is within grid, then we update i to i+dx, j to j+dy, and append the character corresponding to the movement to result. After exiting the loop, return result which

is the path to be followed to win the game.

3 Contribution

Both of us contributed equally overall. Division is as follows :

3.1 Question 1

Encoding clauses of initial values: Mayank
Encoding that each cell contains atleast 1 value: Mayank
Encoding that no cell contains 2 values: Arnav
Encoding clauses denoting that each row,column and the 9 3*3 sub-grids contained each value: Arnav and Mayank
Decoding the literals back to sudoku convention of number being stored in a cell: Arnav

3.2 Question 2

Parsing the grid : Arnav
Variable encoding of player and boxes : Arnav
Encoding initial positions for player and boxes : Mayank
Encoding player movement : Arnav
Encoding box movements : Mayank
Non overlap constraints : Arnav and Mayank
Encoding goal conditions : Mayank
Encoding that player can't be at 2 positions at the same time : Arnav
Encoding that no box can be at 2 positions at the same time : Mayank
Decoding : Mayank

3.3 Report

Written equally by both of us.