Assignment 1 :CS228

Hemant Godve
24B1004

Sreeraj Sasidharan N
24B0982

August 2025

Question 1

Approach Overview

This report describes how to construct a Boolean Satisability (SAT) problem from a Sudoku gameso that the game can be solved with SAT methods. The method consists in a reduction of the game into a set of propositional variables modelling the state of the grid and expressing game rules by a set of logical constraints in CNF. The SAT solver tries to satisfy this CNF using the given rule.

Variable Encoding Scheme

The underlying 9x9 Sudoku grid is encoded by **729 propositional variables** in total. The variable is equal to the digit in a particular cell.

Weak variables are defined by
$v_{r,c,d} \mid 0c,\ 0r,\ 0d,\ 1 \leq d \leq d(r,c)$

- r is the index of the row and $r \in \{1, 2, ..., 9\}$.
- c the column number, $c \in \{1, 2, ..., 9\}$.
- d is the digit, $d \in \{1, 2, \ldots, 9\}$.

The intended interpretation of a variable is as follows: "$v_{r,c,d}$ is **true** precisely when the cell in row r and column c is filled with the number d".

Constraint Encoding in CNF

The Sudoku rules are being translated into a set of logical clauses – a conjunctive normal form (CNF) formula. An accept assignment for this formula is directly equivalent to a solution to the puzzle. The following five sets of constraints are encoded here using the implementation.

Cell Constraints

These are the clauses that guarantee that each cell of the grid contains exactly one digit.

- **Not Threatened by Any One Digit:** For each cell (r, c), a clause is added to ensure that this cell does not contain any digit from 1 to 9. This gives a disjunction (an OR) of 9 variables:

$$(v_{r,c,1} \vee v_{r,c,2} \vee \cdots \vee v_{r,c,9})$$

- **At most One digit:** For each cell (r, c), clauses are created that disallow it to have two different digits together. For every pair (d_1, d_2) that are not the same, we add a clause:

$$(\neg v_{r,c,d_1} \vee \neg v_{r,c,d_2})$$

  I.e., if v_{r,c,d_1} is true then v_{r,c,d_2} is false and vice-versa.

Row and Column Constraints

These conditions guarantee that all digits occur exactly once in any row, any column.

- **At Least One Appearance:** A clause is generated for every row r and digit d that states that d appears somewhere in the row. One such faxt clause set is produced for each column.

  Row: $(v_{r,1,d} \vee v_{r,2,d} \vee \ldots \vee v_{r,9,d})$

Sreeraj Sasidharan N
24B0982

Hemant Godve
24B1004

Column: $(v_{1,c,d} \vee v_{2,c,d} \vee \ldots \vee v_{9,c,d})$

- **At Most One Appearance:** For each row r and each digit d, we add pairwise clauses for d not being in two different columns.

Columns are done by the same logic.

Row: $(\neg v_{r,c_1,d} \vee \neg v_{r,c_2,d})$ *for every permutation of columns* (c_1, c_2) *with* $c_1 < c_2$.

Column: $(\neg v_{r_1,c,d} \vee \neg v_{r_2,c,d})$ for all pairs of rows (r_1, r_2) with $r_1 < r_2$.

Subgrid (Box) Constraints

These blocks make certain that there is precisely one instant of each number within each 3x3 sub-square. The reasoning is the same as the row and column constraints, just with the nine cells within each of the nine small squares.

Puzzle-Specific Constraints (Clues)

The numbers on the grid at the beginning of the puzzle are set as givens which are non-modifiable. If the cell (r, c) is to contain the digit d, a **unit clause** of this form is added.

$$(v_{r,c,d})$$

There is only one positive variable in this clause, so any satisfying model will have to make this variable **true**.

Note on Constraint Redundancy

The encoding contains restrictions which limit a digit to appear only once in each row and column and box. These constraints represent valid logical conditions that exist but remain unnecessary. The cell constraints which define single digit values per cell together with the appearance constraints which require at least one digit per row column and box eliminate duplicate occurrences automatically. The pigeonhole principle demonstrates that nine distinct digits cannot fit into nine individual boxes because this arrangement leaves no space for duplicates. The formula should eliminate unnecessary clauses because doing so will both decrease its size and enhance solver efficiency.

Solving and Decoding Process

After all constraints become available the entire set of clauses enters a SAT solver for processing.

1. **Solving:** The solver attempts to discover a model which assigns true/false values to all 729 variables that fulfill all clauses within the formula.
2. **Result:** A puzzle is considered solvable when the solver finds a model. The solver determines that no model exists when it proves this condition.
3. **Decoding:** A model that represents a solvable puzzle contains variables which have been assigned true values. The list from the model is processed to reconstruct the 9x9 solution grid. For each cell (r, c), the digit d is found such that $v_{r,c,d}$ is in the model.

Sreeraj Sasidharan N
24B0982

Hemant Godve
24B1004

Question 2

The answer explains the process of modeling and solving Sokoban puzzles by implementing a SAT solver. A system for encoding game states at different times must be developed along with a method to convert Sokoban rules into CNF constraints.

Variable Encoding Scheme

A SAT solver requires Sokoban representation through unique integer variables which track every possible game state across multiple time points. The problem is discretized into a maximum of **T** time steps.

There are two main types of variables:

Player Position Variables

A variable named $P_{x,y,t}$ functions to express the statement that the player stands at coordinate (x, y) during time step t."

- x is the row.
- y is the column.
- t is the time step, from 0 to T.

Box Position Variables

The variable $B_{b,x,y,t}$ denotes the statement that box number b exists at position (x, y) during time step t.

- Each box receives its own unique index number b which ranges from 0 to `num_boxes` - 1.
- Each box has its own unique row position represented by x.
- Each box has its own unique column position represented by y.
- Each box has its own unique time step t which ranges from 0 to T.

The function `var_box(b, x, y, t)` serves as the implementation for this concept in the code.

A large set of variables exists because each proposition maps to a unique positive integer which enables the complete description.

Translation of Rules into CNF Constraints

Sokoban rules exist in the form of logical clauses which use Conjunctive Normal Form as their encoding. The formula's satisfying assignment shows a valid sequence of game states which solves the puzzle.

Initial and Goal Conditions

- **Initial State (t=0):** At the beginning of the game the player along with every box are assigned as **unit clauses**. Variables which represent these initial positions must become **true**.
- **Goal State (t=T):** Each box receives a clause which requires it to be positioned on one of the goal squares during the final time step T. The clause for box b appears as:

$$(B_{b,g_1,T} \vee B_{b,g_2,T} \vee \dots)$$

where $g_1, g_2, \dots$ represent the positions of the goal squares.

Sreeraj Sasidharan N
24B0982

Hemant Godve
24B1004

Grid and Object Constraints

- **Uniqueness of Position:** Multiple restrictions are employed to guarantee that at time t:
  - The player maintains a single valid position at each time step.
  - All boxes maintain singular valid positions at each time.

  The method works by joining a statement that objects need to be in *atleast* one place with pairs of statements about being *atmost* one place.
- **Non-Overlapping Boxes:** Clauses exist which stop two different boxes from sharing the same square at any given moment. The logic states: "b_1 and b_2 cannot share position (x, y)." The resulting clause simplifies to:

$$(\neg B_{b_1,x,y,t} \vee \neg B_{b_2,x,y,t})$$

Movement and Box-Pushing Logic

The main part of the encoding defines how states transition between different time steps.

- **Player Movement:** The movement rules are encoded "backwards." At time $t + 1$ the player stands at (n_x, n_y) so they must have been at one of the valid adjacent positions during time t. The system guarantees that the player executes single-step movements.
- **Box Pushing:** This is the most complex constraint. The condition requires that if the player stands on (x, y) during time t and a box exists next to the player on (n_x, n_y) during t then the player will occupy (n_x, n_y) at $t + 1$ while the box will occupy (n_{nx}, n_{ny}) which represents the square behind it at $t + 1$. The CNF transformation requires two separate clauses for this implication.
- **The Frame Problem (Inertia):** A fundamental rule states that objects remain stationary until an action causes them to move. The "Box Frame Axiom" serves to enforce this rule through the following condition: a box at position (x, y) during time t without being pushed must remain at the same position (x, y) during time $t + 1$. Boxes cannot move independently because of this rule.

Decoding the Solution

The SAT solver outputs an integer variable list which contains all the true variables after finding a satisfying assignment. The model stores every step of the game starting from $t = 0$ up to $t = T$. The solution path decoding process uses this model to reconstruct the solution:

1. **Filter and Organize:** The decoder first creates a set of all the true variables for quick lookups.
2. **Trace the Player:** The decoder scans through each time step starting from $t = 0$ up to T to determine which player position variable $P_{x,y,t}$ holds true at that time. The player's route is reconstructed through a series of coordinate positions.
3. **Convert Coordinates to Moves:** The player coordinate series is converted into move sequences using `'U'`, `'D'`, `'L'`, `'R'`. When the player's coordinates show $(3, 3)$ at $t = 0$ followed by $(2, 3)$ at $t = 1$ the move is determined to be `'U'` (Up).
4. **Final Output:** The final Sokoban puzzle solution consists of this sequence of moves. When no model exists the puzzle cannot be solved within T steps.

Sreeraj Sasidharan N
24B0982

Hemant Godve
24B1004

Individual Contribution

- **Sriraj:**
  - Developed the SAT-based Sudoku logic solution.
  - Debugged the Sudoku implementation.
  - Devised the logic for Sokoban.
  - Jointly designed and implemented Sokoban logic with Hemant.
- **Hemant:**
  - Wrote the logic for the Sudoku code.
  - Helped write the Sokoban code.
  - Contributed in debugging of the Sokoban implementation.

  We tried hard to solve the Sokoban, but we were unable to get all the test cases to pass

Sreeraj Sasidharan N
24B0982

Hemant Godve
24B1004