CS 228: Logic in Computer Science

October 3, 2025

Contents

1 Question 1: Sudoku SAT Solver **3**
1.1 Objective . . . 3
1.2 Variable Encoding . . . 3
1.3 CNF Constraint Encoding . . . 3
1.3.1 Exactly One Number per Cell . . . 3
1.3.2 Uniqueness Constraints . . . 3
1.3.3 Initial State . . . 4
1.4 Decoding the Solution . . . 4

2 Question 2: Sokoban Puzzle **5**
2.1 Our Approach . . . 5
2.2 Propositional Variable Encoding . . . 5
2.2.1 Player Variables . . . 5
2.2.2 Box Variables . . . 5
2.3 CNF Constraint Encoding . . . 6
2.3.1 Static Constraints . . . 6
2.3.2 Transition Constraints (Frame Axioms) . . . 7
2.4 Goal Condition . . . 8
2.5 Decoding the Solution . . . 8

1 Question 1: Sudoku SAT Solver

1.1 Objective

The objective is to implement a Sudoku solver by encoding the puzzle as a Satisfiability (SAT) problem. The Sudoku rules and the initial grid configuration are converted into a Conjunctive Normal Form (CNF) formula. A SAT solver is then used to find a satisfying assignment, which, if it exists, represents the unique solution to the puzzle.

1.2 Variable Encoding

A Sudoku grid consists of $9 \times 9 = 81$ cells. Each cell can be filled with a digit from 1 to 9. We represent the state of the puzzle using propositional variables. Let the variable $x_{i,j,k}$ be true if and only if the cell at row i and column j contains the digit $k+1$. The indices range from 0 to 8 for all three dimensions.

To map each unique state (i, j, k) to a single positive integer variable ID, we use the following linear formula:

$$\text{var}(i, j, k) = i \cdot 81 + j \cdot 9 + k + 1$$

This formula guarantees that each possible cell-digit combination maps to a unique integer, which is required by SAT solvers. The '+1' ensures the variables are positive integers.

1.3 CNF Constraint Encoding

We translate the fundamental rules of Sudoku into a set of CNF clauses.

1.3.1 Exactly One Number per Cell

This is a combination of two sub-rules:

1. **At Least One:** For each cell (i, j), we must assert that it contains at least one number. This is done by a single clause for each cell:

$$(x_{i,j,0} \vee x_{i,j,1} \vee \cdots \vee x_{i,j,8})$$

2. **At Most One:** For each cell (i, j), we must assert that it cannot contain two different numbers. We add a binary clause for every pair of distinct digits k and l:

$$(\neg x_{i,j,k} \vee \neg x_{i,j,l})$$

1.3.2 Uniqueness Constraints

The rules of Sudoku require that each digit appears exactly once in each row, each column, and each 3×3 subgrid (box). We enforce these using a series of "at most one" constraints.

- **Rows:** For each row i and each digit k, we ensure that $k+1$ does not appear in two different columns j_1 and j_2.

$$(\neg x_{i,j_1,k} \vee \neg x_{i,j_2,k})$$

- **Columns:** For each column j and each digit k, we ensure that $k+1$ does not appear in two different rows i_1 and i_2.

$$(\neg x_{i_1,j,k} \vee \neg x_{i_2,j,k})$$

- **Boxes:** For each 3×3 box and each digit k, we ensure that $k+1$ does not appear in two different cells within the same box.

$$(\neg x_{i_1,j_1,k} \vee \neg x_{i_2,j_2,k})$$

1.3.3 Initial State

The numbers already present in the initial Sudoku grid are facts that must be true in the final solution. We encode each pre-filled number as a unit clause. For a number N at grid position (i, j), we add the clause:

$$[x_{i,j,N-1}]$$

1.4 Decoding the Solution

If the SAT solver finds a satisfying assignment, it returns a 'model' (a list of true literals). We parse this model to reconstruct the solved Sudoku grid. The process is as follows:

1. We initialize an empty 9×9 grid.
2. We iterate through each positive literal 'v' in the model.
3. For each 'v', we apply the inverse of our encoding function to determine the row, column, and digit:
   - Row: $i = (v-1) \div 81$
   - Column: $j = ((v-1) \pmod{81}) \div 9$
   - Digit: $k = (v-1) \pmod 9$
4. The number is then $k+1$. We place this number in the grid at position (i, j).

This process fills the grid with the solved numbers. If the SAT solver reports the formula is unsatisfiable, no solution exists for the given puzzle.

2 Question 2: Sokoban Puzzle

2.1 Our Approach

Our approach is to model the Sokoban puzzle over a fixed number of time steps, T. We create a vast set of propositional variables, where each variable represents a specific fact about the game, such as "the player is at coordinate (r,c) at time t". We then translate all the rules of Sokoban—how objects move, what constitutes a collision, and the winning condition—into a single, large CNF formula.

If the SAT solver finds a satisfying assignment for this formula, that assignment represents a complete, valid sequence of game states from the start to a successful finish. If the solver proves the formula is unsatisfiable, it means no solution exists within the given time limit T.

This report details the three main stages of this process:

1. **Variable Encoding:** Systematically mapping every possible game state to a unique integer variable.
2. **Constraint Encoding:** Translating the game's rules into a series of CNF clauses.
3. **Solution Decoding:** Interpreting the solver's output to reconstruct the sequence of moves.

2.2 Propositional Variable Encoding

The first step in translating a real-world problem to SAT is to define what a propositional variable represents. In this time-based model of Sokoban, a variable represents a fact about a specific game object at a specific location at a specific time. Our grid has dimensions $N \times M$.

2.2.1 Player Variables

We define a variable for every possible location the player can be at, for every time step from 0 to T. Let the proposition $P_{r,c,t}$ be true if and only if the player is at row r and column c at time t.

To map this three-dimensional state (r, c, t) to a single, unique integer, we "flatten" it using the following formula:

$$\text{var\_player}(r, c, t) = (t \cdot N \cdot M) + (r \cdot M) + c + 1$$

The '+1' is necessary because SAT variable IDs must be positive integers. This formula ensures that every unique combination of (r, c, t) maps to a unique integer.

2.2.2 Box Variables

The logic for boxes is similar, but we have an additional dimension: the box index b, since there can be multiple boxes. Let the proposition $B_{b,r,c,t}$ be true if and only if box b is at row r and column c at time t.

To prevent any overlap with player variables, we first calculate the total number of possible player variables and use this as an offset.

$$\text{player\_offset} = (T+1) \cdot N \cdot M$$

The formula for box variables then becomes:

$$\text{var\_box}(b, r, c, t) = \text{player\_offset} + (b \cdot \text{player\_offset}) + \text{time\_space\_offset}$$

$$\text{time\_space\_offset} = (t \cdot N \cdot M) + (r \cdot M) + c + 1$$

where 'time_space_offset' is the same flattening formula used for the player. This scheme guarantees that player variables and variables for different boxes all occupy distinct, non-overlapping numerical ranges.

2.3 CNF Constraint Encoding

With a clear variable scheme, we translate the rules of Sokoban into Conjunctive Normal Form (CNF). These rules are divided into two main categories: **Static Constraints**, which define a valid board at any single moment, and **Transition Constraints**, which govern valid changes between time steps.

2.3.1 Static Constraints

These rules must hold true for any given time step t, from 0 to T.

Initial State We must assert the starting configuration of the puzzle at time $t = 0$. This is achieved with **unit clauses** (clauses with a single literal). If the player starts at (r_p, c_p) and a box b starts at (r_b, c_b), we add the following clauses to our CNF formula:

- $[P_{r_p,c_p,0}]$ — Asserts the player is at their start position.
- $[B_{b,r_b,c_b,0}]$ — One such clause is added for each box b.

"Exactly One" Location Constraint This is a fundamental constraint: at any time t, every object (the player and each box) must be in exactly one non-wall location. This is encoded by combining two sub-rules:

1. **At Least One:** For each object, we create a large clause representing a disjunction of all its possible positions. For the player at time t, this clause is:

   $$(P_{r_1,c_1,t} \vee P_{r_2,c_2,t} \vee \cdots \vee P_{r_k,c_k,t})$$

   This ensures the object exists somewhere on the board.

2. **At Most One:** We must forbid an object from being in two different places at once. This is done by creating a negative clause for every pair of distinct locations (r_i, c_i) and (r_j, c_j):

   $$(\neg P_{r_i,c_i,t} \vee \neg P_{r_j,c_j,t})$$

   This is logically equivalent to $\neg(P_{r_i,c_i,t} \wedge P_{r_j,c_j,t})$, stating that the player cannot be at both locations simultaneously. This is repeated for every pair of locations and for every object.

Collision Constraint No two objects can occupy the same cell at the same time. For every cell (r, c) and time t, we enforce:

- **Player-Box Collision:** A player and a box cannot collide. For each box b:

$$(\neg P_{r,c,t} \vee \neg B_{b,r,c,t})$$

- **Box-Box Collision:** Two different boxes cannot collide. For each pair of boxes b_1, b_2:

$$(\neg B_{b_1,r,c,t} \vee \neg B_{b_2,r,c,t})$$

2.3.2 Transition Constraints (Frame Axioms)

These crucial rules connect the state at time t to the state at time $t+1$, defining the physics of movement.

Player Movement (Causality) To prevent the player from teleporting across the board, we must enforce that any player position at time $t+1$ is reachable from an adjacent position at time t.

- **Logic:** If the player is at location B at time $t+1$, they must have been at an adjacent location A at time t.

$$P_{B,t+1} \implies \bigvee_{A \in \text{Adjacent}(B)} P_{A,t}$$

- **CNF Clause:** This translates to a single clause for each possible destination cell B:

$$(\neg P_{B,t+1} \vee P_{A_1,t} \vee P_{A_2,t} \vee \dots)$$

The Push Rule (Consequence) This is the central mechanic of Sokoban. It defines what happens when a push is successful.

- **Logic:** If the player at A moves into the adjacent cell B (which contains a box), and the cell C (on the other side of B) is free, then the box at B is pushed to C.

$$(P_{A,t} \wedge B_{b,B,t} \wedge P_{B,t+1}) \implies B_{b,C,t+1}$$

- **CNF Clause:** This becomes a single clause with four literals:

$$(\neg P_{A,t} \vee \neg B_{b,B,t} \vee \neg P_{B,t+1} \vee B_{b,C,t+1})$$

The Inertia Rule (Frame Problem) We must explicitly state that objects that are not acted upon do not change their state. A box that is not pushed must stay in its place.

- **Logic:** If a box is at 'pos' at time 't' AND it was NOT pushed, THEN it must still be at 'pos' at time 't+1'.

- **Implementation:** We handle this using **helper variables**. We define a new variable, 'push_action_another_var', which is true if and only if the player at an adjacent cell A moves to 'pos'.

$$\text{push\_action\_another\_var}_{A \to \text{pos},t} \iff (P_{A,t} \wedge P_{\text{pos},t+1})$$

This equivalence is encoded into CNF using three clauses. With these helper variables, the final inertia clause for a box at 'pos' is:

$$(\neg B_{\text{pos},t} \vee \text{push\_action}_{A_1 \to \text{pos},t} \vee \dots \vee B_{\text{pos},t+1})$$

Forbidding Impossible Pushes The logic is incomplete if we only state what happens on a successful push; we must also forbid impossible pushes.

- **Logic:** If a player at A is next to a box at B, and the space behind at C is an obstacle (a wall or another box), then the player is forbidden from moving from A to B.

$$(P_{A,t} \land B_{B,t} \land \text{Obstacle}_{C,t}) \implies \neg P_{B,t+1}$$

- **CNF Clauses:** This translates to clauses that forbid the action:
  - **Into a wall:** $(\neg P_{A,t} \lor \neg B_{B,t} \lor \neg P_{B,t+1})$
  - **Into another box:** $(\neg P_{A,t} \lor \neg B_{1,B,t} \lor \neg B_{2,C,t} \lor \neg P_{B,t+1})$

2.4 Goal Condition

Finally, we must state the winning condition.

- **Logic:** At the final time step T, every box must be on a goal square.
- **CNF:** For each box b, we create one large clause listing all possible goal locations:

$$(B_{b,G_1,T} \lor B_{b,G_2,T} \lor \dots)$$

2.5 Decoding the Solution

If the SAT solver reports that the formula is satisfiable, it returns a 'model'—a list of literals that are true in the satisfying assignment. Our final task is to translate this list of numbers back into a sequence of moves.

The process is a direct reversal of the encoding:

1. **Reconstruct the Player's Path:** We initialize an empty path array. We then iterate through the 'model'. For each positive literal (true variable), we check if it falls within the numerical range we reserved for player variables. If it does, we apply the inverse of our encoding formula (using integer division '//' and modulo '
2. **Convert Path to Moves:** After reconstructing the full path, we iterate through it from $t = 0$ to $T-1$. For each step, we take the player's position at time t, (r_1, c_1), and at time $t+1$, (r_2, c_2).
3. **Calculate Direction:** We calculate the difference vector: $(dr, dc) = (r_2 - r_1, c_2 - c_1)$.
4. **Look up Move:** This vector corresponds to a unique direction. For example, $(-1, 0)$ is 'U' (Up), and $(0, 1)$ is 'R' (Right). We look up this vector in our 'DIRS' dictionary to find the corresponding move character and append it to our list of moves.

The final list of moves is the solution to the puzzle. If the solver finds no solution, we return '-1' to indicate the puzzle is unsatisfiable within the given time limit.