Jahonoro Tojieva

Solving Sokoban as a Planning Problem Using SAT Encoding

Computational Logic

FACULTY OF MATHEMATICS,
PHYSICS AND INFORMATICS
Comenius University
Bratislava

2026

Abstract

This project addresses the classical Sokoban puzzle by modeling it as a deterministic planning problem and solving it using propositional satisfiability (SAT). The game environment, actions, and goal conditions are encoded into conjunctive normal form (CNF) and solved using the Minisat solver. The solution incrementally searches for the shortest plan and provides both a command-line interface and a graphical visualization of the resulting plan. Several optimizations are introduced to improve scalability on larger maps.

1. Problem Description

Sokoban is a classical single-agent puzzle game played on a two-dimensional grid consisting of walls, free cells, movable crates, and storage locations (goals). The player occupies exactly one free cell and can move or push crates in order to place all crates onto goal cells.

The game is deterministic, fully observable, and static: the layout of the grid does not change, and all action outcomes are known in advance. Each crate occupies exactly one cell, multiple crates cannot share a cell, and neither the player nor crates may enter wall cells.

The player can perform two types of actions. A *move* action allows movement to an adjacent empty cell in one of the four cardinal directions. A *push* action allows the player to push a single adjacent crate into a free cell behind it; crates cannot be pulled.

The goal is reached when all crates are placed on goal cells, regardless of the player's final position. Due to irreversible actions and tight spatial constraints, incorrect moves may lead to unsolvable configurations.

In this project, Sokoban is modeled as a classical planning problem and solved by encoding its dynamics and constraints into propositional logic. A SAT solver is then used to compute a shortest valid plan.

2. Planning Model

In order to solve Sokoban using automated planning techniques, the game is formalized as a classical planning problem with discrete time steps. A planning instance is defined by a finite set of states, actions with preconditions and effects, background constraints, and a goal condition.

State Representation

A state represents a complete configuration of the Sokoban grid at a given discrete time step $t \in \mathbb{N}$. The state is described using propositional predicates that capture the positions of the player and all crates, as well as auxiliary information about goal satisfaction.

The following predicates are used to represent states:

- `playerAt(x, y, t)`: the player is located at cell (x, y) at time step t.
- `boxAt(b, x, y, t)`: crate b is located at cell (x, y) at time step t.
- `emptyCell(x, y, t)`: cell (x, y) is empty at time step t.
- `goal(x, y)`: cell (x, y) is a storage (goal) cell. This predicate is static.
- `reachedGoal(b, t)`: crate b has been placed on a goal cell at or before time step t.

At every time step, exactly one player position is true, and each crate occupies exactly one cell. Walls are not explicitly represented as predicates; instead, the set of valid coordinates excludes wall cells.

Actions

The planning model includes three types of actions, corresponding to the legal moves in Sokoban. All actions are parameterized by a time step t and affect the transition from state $t-1$ to state t.

Move

The `move` action represents a simple movement of the player to an adjacent free cell:

```
move(fromX, fromY, toX, toY, t)
```

Preconditions

- The player is at cell (`fromX`, `fromY`) at time $t-1$.
- The target cell (`toX`, `toY`) is empty at time $t-1$.
- The two cells are adjacent in one of the four cardinal directions.

Effects

- The player is at (`toX`, `toY`) at time t.
- The source cell (`fromX`, `fromY`) becomes empty at time t.

Push

The `push` action models pushing a crate into a non-goal cell:

```
push(b, px, py, bx, by, tx, ty, t)
```

Preconditions

- The player is at position (px, py) at time $t-1$.
- Crate b is located at the adjacent cell (bx, by).
- The target cell (tx, ty) behind the crate is empty at time $t-1$.
- The target cell is not a goal cell.

Effects

- The player moves to the former crate position (bx, by).
- Crate b moves to the target cell (tx, ty).
- The player's original cell becomes empty.
- The crate is not marked as having reached a goal.

Push to Goal

The `pushToGoal` action represents pushing a crate onto a goal cell:

```
pushToGoal(b, px, py, bx, by, tx, ty, t)
```

Preconditions

- The player is at position (px, py) at time $t-1$.
- Crate b is located at the adjacent cell (bx, by).
- The target cell (tx, ty) is empty at time $t-1$.
- The target cell is a goal cell.
- The crate has not previously reached a goal.

Effects

- The player moves to the crate's former position.
- Crate b is moved onto the goal cell.
- The crate is marked as having reached a goal.

Goal Condition

The planning goal is to place all crates onto goal cells. Formally, the goal condition requires that for every crate b, the predicate `reachedGoal(b, t)` is true at the final time step t. The final position of the player is irrelevant.

Frame Problem and State Persistence

The frame problem arises from the need to specify which aspects of the state remain unchanged after an action is executed. In this model, frame axioms are used to ensure that objects not affected by the chosen action preserve their positions and goal status between consecutive time steps.

Specifically:

- A `move` action does not change the positions of any crates or their goal status.
- A `push` or `pushToGoal` action affects only a single crate; all other crates remain unchanged.
- The goal status of unaffected crates persists across time steps.

Background Knowledge

Several background constraints are enforced implicitly in the planning model:

- At most one action may occur at each time step.
- Each cell may contain at most one object (player or crate).
- The player and crates can only occupy non-wall cells.

These constraints significantly restrict the search space and ensure that only valid Sokoban configurations are considered.

3. SAT Encoding

The planning model is reduced to propositional satisfiability by introducing time-indexed Boolean variables for states and actions. For a fixed plan horizon T, the encoding produces a CNF formula Φ_T such that Φ_T is satisfiable if and only if there exists a valid Sokoban plan of length T. The final solution is obtained using iterative deepening over T, and the first satisfiable instance yields a shortest plan.

Variables

Let C be the set of free (non-wall) cells and $B = \{1, \ldots, m\}$ the set of crates. For each time step $t \in \{0, \ldots, T\}$, the encoding introduces Boolean variables corresponding to the following predicates:

- $\texttt{playerAt}(x, y, t)$ for $(x, y) \in C$,
- $\texttt{boxAt}(b, x, y, t)$ for $b \in B$ and $(x, y) \in C$,
- $\texttt{emptyCell}(x, y, t)$ for $(x, y) \in C$,
- $\texttt{reachedGoal}(b, t)$ for $b \in B$.

In addition, action variables are created for each time step $t \in \{1, \ldots, T\}$:

- $\texttt{move}(\texttt{fromX}, \texttt{fromY}, \texttt{toX}, \texttt{toY}, t)$,
- $\texttt{push}(b, px, py, bx, by, tx, ty, t)$,
- $\texttt{pushToGoal}(b, px, py, bx, by, tx, ty, t)$.

The predicate $\texttt{goal}(x, y)$ is static and is treated as background knowledge derived from the input map.

Initial State

The initial state is encoded by fixing all relevant predicates at time $t = 0$. For example, if the player starts at (x_0, y_0), the encoding includes:

$$\texttt{playerAt}(x_0, y_0, 0)$$

and for all other $(x, y) \neq (x_0, y_0)$, the corresponding literals are set to false. Similarly, each crate position $\texttt{boxAt}(b, x, y, 0)$ is fixed according to the map, and $\texttt{emptyCell}(x, y, 0)$ is set consistently with object occupancy.

State Validity Constraints

To ensure that each time step describes a valid Sokoban configuration, the following constraints are enforced.

At least one occupancy label per cell. For each $(x, y) \in C$ and each t, the cell must be either empty, contain the player, or contain a crate:

$$\texttt{emptyCell}(x, y, t) \ \vee \ \texttt{playerAt}(x, y, t) \ \vee \ \bigvee_{b \in B} \texttt{boxAt}(b, x, y, t).$$

Mutual exclusion. No cell can contain multiple objects. In particular:

$$\neg\texttt{playerAt}(x, y, t) \ \vee \ \neg\texttt{boxAt}(b, x, y, t) \quad \text{for all } (x, y) \in C,\ b \in B,\ t,$$

and analogous pairwise constraints prevent two different crates from occupying the same cell. Furthermore, the player must be in exactly one cell at each t, and each crate must occupy exactly one cell at each t.

Action Constraints

At least one action. At every step $t \in \{1, \ldots, T\}$, at least one action variable is true:

$$\bigvee_{a \in \mathcal{A}_t} a,$$

where $\mathcal{A}_t$ is the set of all action variables generated for time step t.

At most one action. To ensure sequential execution, at most one action may occur per time step. In the basic version, this can be expressed using pairwise mutex constraints $\neg a_i \vee \neg a_j$. In the optimized version, a sequential (ladder) encoding is used, which reduces the number of required clauses from quadratic to linear in $|\mathcal{A}_t|$ (see **??**).

Preconditions and Effects

Action semantics are encoded as implications from an action variable to its preconditions and effects, which are then transformed into CNF.

Move. For a move action $\texttt{move}(f_x, f_y, t_x, t_y, t)$, we encode:

$$\texttt{move}(f_x, f_y, t_x, t_y, t) \ \rightarrow \ \texttt{playerAt}(f_x, f_y, t-1) \ \wedge \ \texttt{emptyCell}(t_x, t_y, t-1),$$

and the effects:

$$\texttt{move}(f_x, f_y, t_x, t_y, t) \ \rightarrow \ \texttt{playerAt}(t_x, t_y, t) \ \wedge \ \texttt{emptyCell}(f_x, f_y, t).$$

Push and pushToGoal. Push actions require that the player is positioned behind a crate and that the target cell is free. For $\texttt{push}(b, px, py, bx, by, tx, ty, t)$, the encoding enforces:

$$\texttt{push}(\cdot, t) \ \rightarrow \ \texttt{playerAt}(px, py, t-1) \ \wedge \ \texttt{boxAt}(b, bx, by, t-1) \ \wedge \ \texttt{emptyCell}(tx, ty, t-1),$$

with corresponding effects moving the player to (bx, by) and the crate to (tx, ty). The action `pushToGoal` is treated similarly, with the additional requirement that $\texttt{goal}(tx, ty)$ holds and the goal-status predicate $\texttt{reachedGoal}(b, t)$ becomes true.

Frame Axioms

To handle the frame problem, persistence constraints are added so that facts not affected by the chosen action remain unchanged between consecutive steps. For example, if no action moves crate b, then its position persists:

$$\texttt{boxAt}(b, x, y, t-1) \ \wedge \ \neg \mathit{MovesBox}(b, t) \ \rightarrow \ \texttt{boxAt}(b, x, y, t),$$

where $\mathit{MovesBox}(b, t)$ is a shorthand for the disjunction of all actions at time t that move crate b. Analogous persistence constraints are used for $\texttt{reachedGoal}(b, t)$.

Goal Encoding

The goal of Sokoban is satisfied when all crates are placed onto storage cells. In the encoding, this is expressed by requiring that every crate has reached a goal at the final time step:

$$\bigwedge_{b \in B} \texttt{reachedGoal}(b, T).$$

Iterative Deepening and Optimality

To obtain a shortest plan, the solver constructs and solves Φ_T for $T = 1, 2, \ldots$ until a satisfiable instance is found. Because each horizon corresponds to plans of exactly that length, the first satisfiable T yields a plan that is optimal with respect to the number of time steps.

4. Implementation Overview

The project is implemented in Python and is structured in a modular way in order to clearly separate the planning model, the SAT encoding, and the user interface.

At the core of the system is a single solver class, responsible for coordinating the encoding process, invoking the SAT solver, and extracting the resulting plan. The encoding itself is divided into several specialized components. State-related constraints are handled by a state encoder, action semantics by an action encoder, and mutual exclusivity constraints by a dedicated exclusivity encoder. This separation improves readability and allows individual parts of the encoding to be modified or optimized independently.

The planning problem is translated into a CNF formula and written to a file, which is then converted into DIMACS format. The external SAT solver Minisat is invoked as a separate process, and its output is parsed to reconstruct the executed actions. The solver uses an iterative deepening strategy over the plan length, incrementally increasing the time horizon until a satisfiable instance is found.

In addition to a command-line interface that prints the resulting action sequence, the project also includes a graphical user interface. The GUI allows the user to select an input map, run the solver, and visually observe the execution of the computed plan step by step on the grid.

5. Experimental Results and Observations

The implementation was tested on several Sokoban maps of varying complexity, including simple debugging instances and moderately sized puzzles. Specifically, maps labeled 1, 4, 5, 6, and 8 were used for evaluation.

Small maps are typically solved almost instantly, often within a fraction of a second. As the map size and the number of crates increase, the solving time grows significantly. For more complex instances, solving times of up to approximately 20 seconds were observed, depending on the required plan length.

A key observation is that most of the computation time is spent solving unsatisfiable instances during iterative deepening. As the plan horizon increases, the number of propositional variables and clauses grows rapidly, leading to a noticeable slowdown. Nevertheless, the introduced optimizations reduce the size of the encoding sufficiently to allow the solver to handle maps that would otherwise be infeasible.

6. Discussion

Although SAT-based planning provides a declarative and conceptually clean solution to Sokoban, it also exhibits several limitations. The size of the SAT encoding grows quickly with both the number of grid cells and the plan horizon, which directly impacts solver performance. Sokoban is particularly challenging due to the presence of irreversible actions and dead-end configurations, which are difficult to detect purely at the propositional level.

Another limitation is the lack of domain-specific heuristics in a pure SAT formulation. While the solver guarantees optimality in terms of plan length, it does not exploit higher-level structural properties of the puzzle, such as symmetry or dead squares, unless they are explicitly encoded.

Despite these limitations, the SAT-based approach offers strong guarantees of correctness and optimality and serves as a solid baseline for comparing different planning paradigms.

7. Conclusion and Personal Experience

In this project, Sokoban was successfully modeled as a classical planning problem and solved using a SAT-based approach. The project demonstrates how complex game dynamics can be captured using propositional logic and solved with a general-purpose SAT solver.

The most challenging aspect of the project was designing and integrating the overall system architecture, including the interaction between different encoders, the SAT solver, and the visualization component. Debugging the encoding and ensuring the correctness of constraints proved to be more time-consuming than initially expected.

One important lesson learned is that SAT-based planning is a powerful but computationally demanding technique. It is particularly effective for problems with well-defined constraints, but scalability quickly becomes an issue for larger instances. If more time were available, further optimizations could be explored, and alternative paradigms such as Answer Set Programming or Prolog-based planning could be investigated and compared experimentally.

Overall, the project provided valuable insight into automated planning, SAT solving, and the practical challenges of implementing complex AI systems.