Sokoban solver as a planning problem in ASP

Havriil Pietukhin

January 20, 2025

Contents

1 Introduction

Author: Havriil Pietukhin
Date: January 20, 2025
Title: Solving Sokoban as a planning problem using ASP
Purpose: This is a project that demonstrates an approach for solving planning problems with ASP.

2 Problem description

Sokoban is a grid-based puzzle where the player, Sokoban, pushes crates onto designated storage locations. The objective is to position all crates on their goal positions, navigating around walls and avoiding deadlocks.

3 Planning in ASP

ASP is well-suited for Sokoban due to its declarative nature and expression power, allowing the concise definition of rules and constraints essential for planning. In my encoding, I heavily use all these techniques. In my solution, I am running ASP in iterations, finding plans of different lengths. After a run, Clingo outputs a solution if one exists.

4 Formal description of Sokoban planning problem

4.1 Cells positioning

4.2 Fluents

The following predicates and functions describe the **state** and **properties** of the Sokoban grid at any time T. Using mathematical notation, we define:

- $leftOf(\ell_1, \ell_2)$: "Location ℓ_2 is directly **to the left** of location ℓ_1."
- $below(\ell_1, \ell_2)$: "Location ℓ_2 is directly **below** location ℓ_1."
- $wall(\ell)$: "Location ℓ is a **wall** (permanently impassable)."
- $isgoal(\ell)$: "Location ℓ is a **goal cell**."
- $isnongoal(\ell)$: "Location ℓ is a non **goal cell**."
- $sokoban(S)$: "The object S is **Sokoban** (the player)."
- $crate(C)$: "The object C is a **crate** (box)."
- $at(O, \ell)$: "The object O (either Sokoban or a crate) is **at** location ℓ."
- $clear(\ell)$: "Location ℓ is **free** (unoccupied)."

4.3 Actions

Move: Sokoban moves one cell without pushing. This action is coded using literals *moveLeft*, *moveRight*, *moveUp*, *moveDown* – 1 for each direction:

- **moveLeft**(S, X, Y)
  - $p^+ = \{at(S, X), clear(Y)\}$
  - $p^- = \{wall(Y)\}$
  - $e^+ = \{at(S, Y), clear(X)\}$
  - $e^- = \{at(S, X), clear(Y)\}$
  - X is directly to the left of Y.
- **moveRight**(S, X, Y)
  - $p^+ = \{at(S, X), clear(Y)\}$
  - $p^- = \{wall(Y)\}$
  - $e^+ = \{at(S, Y), clear(X)\}$
  - $e^- = \{at(S, X), clear(Y)\}$
  - Y is directly to the right of X.

- **moveUp**(S, X, Y)
  - $p^+ = \{at(S, X), clear(Y)\}$
  - $p^- = \{wall(Y)\}$
  - $e^+ = \{at(S, Y), clear(X)\}$
  - $e^- = \{at(S, X), clear(Y)\}$
  - X is directly below Y.
- **moveDown**(S, X, Y)
  - $p^+ = \{at(S, X), clear(Y)\}$
  - $p^- = \{wall(Y)\}$
  - $e^+ = \{at(S, Y), clear(X)\}$
  - $e^- = \{at(S, X), clear(Y)\}$
  - Y is directly below X.

Push actions:

- **pushLeft**(S, X, Y, Z, C)
  - $p^+ = \{at(S, X), at(C, Y), clear(Z)\}$
  - $p^- = \{wall(Y), wall(Z)\}$
  - $e^+ = \{at(S, Y), at(C, Z), clear(X)\}$
  - $e^- = \{at(S, X), at(C, Y), clear(Z)\}$
  - X is directly to the left of Y, and Y is directly to the left of Z.
- **pushRight**(S, X, Y, Z, C)
  - $p^+ = \{at(S, X), at(C, Y), clear(Z)\}$
  - $p^- = \{wall(Y), wall(Z)\}$
  - $e^+ = \{at(S, Y), at(C, Z), clear(X)\}$
  - $e^- = \{at(S, X), at(C, Y), clear(Z)\}$
  - X is directly to the right of Y, and Y is directly to the right of Z.
- **pushUp**(S, X, Y, Z, C)
  - $p^+ = \{at(S, X), at(C, Y), clear(Z)\}$
  - $p^- = \{wall(Y), wall(Z)\}$
  - $e^+ = \{at(S, Y), at(C, Z), clear(X)\}$
  - $e^- = \{at(S, X), at(C, Y), clear(Z)\}$
  - X is directly below Y, and Y is directly below Z.
- **pushDown**(S, X, Y, Z, C)

- $p^+ = \{at(S, X), at(C, Y), clear(Z)\}$
- $p^- = \{wall(Y), wall(Z)\}$
- $e^+ = \{at(S, Y), at(C, Z), clear(X)\}$
- $e^- = \{at(S, X), at(C, Y), clear(Z)\}$
- Y is directly below X, and Z is directly below Y.

Push: Sokoban moves and pushes a crate into the next cell. This action is also coded separately for 4 different directions

4.4 Preconditions

- **Move:**
  - Target cell is empty or a storage location.
  - No crate or wall in the target cell.
- **Push:**
  - Adjacent cell contains a crate.
  - Destination cell is empty or a storage location.
  - No wall blocking.

4.5 Effects

- **Move:** Updates Sokoban's position.
- **Push:** Updates positions of Sokoban and the crate.

5 Preconditions (now formally)

5.1 Move Actions

moveLeft: $\neg do(moveLeft(S, X, Y), T) \vee at(S, X, T) \wedge clear(Y, T)$
moveRight: $\neg do(moveRight(S, X, Y), T) \vee at(S, X, T) \wedge clear(Y, T)$ moveUp: $\neg do(moveUp(S, X, Y), T)$
moveDown: $\neg do(moveDown(S, X, Y), T) \vee at(S, X, T) \wedge clear(Y, T)$

5.2 Push Actions

pushLeft: $\neg do(pushLeft(S, X, Y, Z, C), T) \vee at(S, X, T) \wedge at(C, Y, T) \wedge clear(Z, T)$
pushRight: $\neg do(pushRight(S, X, Y, Z, C), T) \vee at(S, X, T) \wedge at(C, Y, T) \wedge clear(Z, T)$
pushUp: $\neg do(pushUp(S, X, Y, Z, C), T) \vee at(S, X, T) \wedge at(C, Y, T) \wedge clear(Z, T)$
pushDown: $\neg do(pushDown(S, X, Y, Z, C), T) \vee at(S, X, T) \wedge at(C, Y, T) \wedge clear(Z, T)$

6 Effects (now formally)

6.1 Move Actions

$$
\begin{aligned}
&at(S, Y, T+1) \colon do(moveLeft(S, X, Y), T) \land at(S, X, T) \land clear(Y, T) \\
&-\, at(S, X, T+1) \colon do(moveLeft(S, X, Y), T) \land at(S, X, T) \\
&clear(X, T+1) \colon do(moveLeft(S, X, Y), T) \land at(S, X, T) \land clear(Y, T) \\
&-\, clear(Y, T+1) \colon do(moveLeft(S, X, Y), T) \land at(S, X, T) \land clear(Y, T)
\end{aligned}
$$

6.2 Push Actions

$$
\begin{aligned}
&at(S, Y, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(S, X, T) \land at(C, Y, T) \land clear(Z, T) \\
&-\, at(S, X, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(S, X, T) \\
&at(C, Z, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(C, Y, T) \land clear(Z, T) \\
&-\, at(C, Y, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(C, Y, T) \\
&clear(X, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(S, X, T) \land clear(Z, T) \\
&-\, clear(Z, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(C, Y, T) \land clear(Z, T) \\
&-\, clear(Y, T+1) \colon do(pushLeft(S, X, Y, Z, C), T) \land at(S, X, T) \land clear(Z, T)
\end{aligned}
$$

Preconditions and effects are using time T, $0 <= T <= n$, where n is the current planning horizon.

7 Frame Problem Solution

I utilized inertia rules to assume states persist unless altered by actions, thereby avoiding the need to specify all non-changes explicitly.

8 Inertia Rules

Definition: If an object O is in location L at time T, and there is no proof that it stopped being there at time $T+1$, then it remains in L.

8.1 Inertia for Object Positions

$$at(O, L, T+1) \iff at(O, L, T) \land \neg\neg at(O, L, T+1) \quad \forall O, L, T$$

Explanation:

- If O is in L at time T, and it is not explicitly proven that O moved at $T+1$, then O remains in L.
- This rule ensures the persistence of states unless an action explicitly changes them.

8.2 Inertia for Cell Clarity

$$clear(L, T+1) \iff clear(L, T) \land \neg\neg clear(L, T+1) \quad \forall L, T$$

Explanation:

- If L is clear at time T, and it is not explicitly proven that L becomes blocked at $T+1$, then L remains clear.

8.3 General Inertia Rule for Fluents

For any fluent ϕ:

$$\phi(T+1) \iff \phi(T) \land \neg\neg\phi(T+1)$$

Explanation:

- This general rule applies to all fluents, ensuring their persistence unless explicitly altered.

9 Background Knowledge

The encoding incorporates spatial relations, entity properties, and constraints to accurately model the Sokoban environment and ensure adherence to game rules. More details can be found in the next chapter or in the encoding itself, `sokoban.lp`.

10 Implementation

The implementation comprises ASP encoding and a Python interface.

10.1 ASP Encoding (`sokoban.lp`)

- Formal representation of the Sokoban problem.
- Defines entities, actions, state transitions, and goal conditions.

10.2 Python Interface

- **Solver (`solver.py`):** Interfaces with Clingo to solve the ASP-encoded Sokoban puzzle. It translates map files into ASP facts, invokes the solver, and processes the solution steps.
- **Visualizer (`visualizer.py`):** Provides a GUI for users to select maps, run tests, and visualize the solution steps interactively.
- **Map Backend (`sokoban_map.py`):** Supports step-by-step map generation.

- **Testing (`test_solver.py` & `conftest.py`):** Implements automated tests using `pytest` to validate the encoding and solver.

Additional Note: I intended to develop an incremental planner (`sokInc3.py`) to dynamically adjust the planning horizon. However, this approach was unsuccessful. Consequently, I implemented a classical planner where the planning horizon is incrementally increased using a constant `maxsteps`.

11 Workflow

1. **Map Input:** Users provide maps in text files.
2. **Fact Generation:** Solver converts maps to ASP facts.
3. **Solving:** Clingo finds action sequences.
4. **Visualization:** GUI displays the solution steps.

12 Detailed Implementation Aspects

- **Fact Generation:** Parses maps to identify walls, crates, storage locations, and Sokoban's initial position.
- **Action Modeling:** Encodes actions with preconditions and effects.
- **Inertia Rules:** Maintains state unless actions change it.
- **Optimization:** Seeks minimal-step solutions.

13 Solving the Frame Problem

I addressed state persistence using inertia rules, ensuring only relevant state changes occur.

13.1 Approach in Encoding

1. **Inertia Rules:** Maintain object positions and cell statuses unless altered.
2. **Negative Effects:** Explicitly remove previous positions when objects move.
3. **Selective Updates:** Only affected entities are updated.
4. **Conciseness:** Avoids over-specification with `not -` constructs.

13.2 Example Scenario

Sokoban performs a `pushRight`:

1. **Preconditions:** Sokoban at X, crate at Y, Z clear.
2. **Execution:** Sokoban moves to Y, crate to Z.
3. **Frame Handling:** Updates only affected positions, preserving others.

14 Experience and Reflections

Working on this project provided profound insights into declarative programming for solving intricate planning problems. Modeling Sokoban in ASP underscored the elegance and expressiveness of logic programming. **Note:** On basic maps like #1, #8, #4-#6, the encoding works efficiently, finding optimal plans within fractions of a second. However, maps #2, #3, and #7 require significantly more time, making them less convenient. Despite this, the encoding can still find suboptimal solutions for these challenging maps.

14.1 Challenges

- **Frame Problem Handling:** Ensuring only relevant state changes were encoded required meticulous inertia rule design.
- **Performance Optimization:** Balancing encoding expressiveness with solver efficiency needed iterative refinements. Each horizon increment (+1 step) doubles solving time, a limitation inherent to Sokoban's NP nature.
- **GUI Integration:** Seamlessly connecting the ASP solver with a user-friendly interface involved careful data synchronization.

14.2 Key Learnings

- **ASP Proficiency:** Enhanced skills in writing efficient ASP encodings and leveraging Clingo for optimization.
- **Problem Decomposition:** Learned to break down real-world problems into formal logical representations.
- **Testing and Validation:** Recognized the importance of comprehensive testing to ensure reliability across various scenarios.

15 Conclusion

This project successfully modeled and solved Sokoban using ASP, demonstrating ASP's capability in AI planning. The integration of an interactive GUI enhances

usability, making the solution accessible to users and serving as a valuable educational tool for those studying AI planning and logic programming.

Future Work: Explore various solver optimizations, handle more complex puzzles, and improve visualization features for a better user experience.