Sokoban Solver as a Planning Problem in ASP

Table of Contents

Introduction

Author: Havriil Pietukhin

Date: January 20, 2025

Title: Solving Sokoban as a Planning Problem Using Answer Set Programming
Purpose: This is a project that demonstrates an approach for solving planning problems with ASP.

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 targets with the fewest moves, navigating around walls and avoiding deadlocks.

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.

Actions, Preconditions, and Effects

When defining Sokoban as a planning problem, we should describe key elements of every planning problem - actions, preconditions, and effects. In case of Sokoban we can define next elements:

Actions:

1. **Move:** Sokoban moves one cell without pushing.
2. **Push:** Sokoban moves and pushes a crate into the next cell.

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.

Effects:

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

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.

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 you can find in next chapter or in the encoding itself, "sokoban.lp"

Encoding the Problem

Below I outline how Sokoban is translated into ASP, covering goals, actions, inertia, and constraints.

Goal Definition

The goal was defined as ensuring all crates reach storage locations at least once during the plan execution.

Key Rules:

- Goals and walls are mutually exclusive.
- Crates must reach goals.

- Prevent overlapping entities.
- Detect and prohibit deadlocks.

Explanation:

1. **Mutually Exclusive Goals and Walls:**
   - A location cannot be both a goal and a wall.
   - Walls are never clear.
2. **Reaching Goals:**
   - A crate on a goal location is considered to have reached its goal.
   - All crates must reach their goals for the plan to be valid.
3. **Entity Position Constraints:**
   - Sokoban cannot occupy multiple locations simultaneously.
   - Crates cannot overlap or share a cell with Sokoban.
4. **Deadlock Detection:**
   - Identifies cells that become deadlocks and prohibits actions that would push crates into them.

Action Selection

This section ensures that one action is selected per timestep and prevents Sokoban from idling unless all goals are achieved.

Explanation:

- At each time step, the solver selects at most one action.
- Sokoban must move unless all goals are achieved.

Defining Actions

I enumerated all possible move and push actions with necessary parameters.

Explanation:

Defines actions like `moveLeft`, `moveRight`, `pushLeft`, etc., specifying the involved entities and target locations.

Inertia

Inertia rules maintain state persistence for object positions and cell statuses unless changed by actions.

Explanation:

- Objects remain in their current positions unless acted upon.
- Cell statuses (clear or blocked) persist unless altered by actions.

Preconditions

This section ensures actions are executed only when required conditions are met, such as Sokoban's position and cell clarity.

Explanation:

Ensures Sokoban is correctly positioned and target cells are clear before performing moves or pushes.

Positive and Negative Effects

Defines how actions modify the game state by updating positions and cell statuses.

Explanation:

Specifies the resulting state after an action, such as updating Sokoban's position and crate locations.

Minimizing Steps

I employed ASP directives to minimize the number of actions and the overall plan duration for efficiency.

Explanation:

Directives guide the solver to find the most efficient sequence of actions to achieve the goal.

Constraints on Walls and Crates

Ensures walls and crates remain consistent throughout the plan, preventing unintended alterations.

Explanation:

Maintains the fixed number of walls and crates, ensuring the puzzle's integrity.

Output Actions

Specifies which predicates to display, focusing on actions and summary information.

Explanation:

Displays the actions taken and provides summary details about crates and walls.

Implementation

The implementation comprises ASP encoding and a Python interface.

1. **ASP Encoding (`sokoban.lp`):**
   - Formal representation of the Sokoban problem.
   - Defines entities, actions, state transitions, and goal conditions.
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`.

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.

Key Modules

- **`solver.py`:** Manages the solving process.
- **`visualizer.py`:** Handles the user interface.
- **`test_solver.py`:** Ensures solver correctness.
- **`conftest.py`:** Configures the testing environment.

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.

Solving the Frame Problem

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

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.

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.

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.

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.

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.

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.