Sokoban Solver as a Planning Problem in ASP

Table of Contents

1. Introduction
2. Problem Description
3. Planning in ASP
    - Actions, Preconditions, and Effects
    - Frame Problem Solution
    - Background Knowledge
4. Encoding the Problem
    - Goal Definition
    - Action Selection
    - Defining Actions
    - Inertia
    - Preconditions
    - Positive and Negative Effects
    - Minimizing the Number of Steps
    - Constraints on Walls and Crates
    - Output Actions
5. Implementation
6. Solving the Frame Problem
7. Experience and Reflections
8. Conclusion

Introduction

Author: Havriil Pietukhin

Date: January 20, 2025

Title: Solving Sokoban as a Planning Problem Using Answer Set Programming
Purpose: This document provides a comprehensive explanation of modeling and solving the Sokoban puzzle as a planning problem using Answer Set Programming (ASP).

Problem Description

Sokoban is a puzzle game where the player, known as Sokoban, pushes crates onto designated storage locations within a confined grid. The objective is to move all crates to their respective storage spots with the fewest possible moves. The complexity arises from the need to plan actions that avoid creating unsolvable configurations, making it an ideal candidate for planning algorithms.

Key Components:

- **Grid Layout:** The game area is represented as a grid consisting of walls, empty spaces, crates, storage locations, and Sokoban's position.
- **Crates:** Movable boxes that Sokoban pushes.

- **Storage Locations:** Targets where crates must be placed.
- **Sokoban:** The player character who moves and pushes crates.

Planning in ASP

Answer Set Programming (ASP) is a declarative programming paradigm suited for solving complex combinatorial problems. In the context of Sokoban, ASP efficiently models the planning problem by defining the game's rules, actions, and constraints.

Actions, Preconditions, and Effects

Actions:
The primary actions in Sokoban are:

1. **Move:** Sokoban moves one cell in a specified direction (up, down, left, right) without pushing a crate.
2. **Push:** Sokoban moves one cell in a direction while pushing a crate into the next cell.

Preconditions:
Conditions required for an action to be executed:

- **Move:**
  - The target cell must be empty or a storage location.
  - The target cell must not contain a crate or a wall.
- **Push:**
  - The cell adjacent to Sokoban in the moving direction contains a crate.
  - The cell beyond the crate (in the same direction) is empty or a storage location.
  - Neither the crate nor the destination cell is a wall.

Effects:
Outcomes resulting from executing an action:

- **Move:**
  - Sokoban's position updates to the target cell.
- **Push:**
  - Sokoban's position updates to the target cell.
  - The crate's position updates to the cell beyond.

Frame Problem Solution

The frame problem involves representing what remains unchanged after an action without explicitly stating all non-changes. In Sokoban's ASP encoding:

- **Fluent Facts:** Dynamic facts that can change over time, such as the positions of Sokoban and crates.
- **Static Facts:** Immutable facts like wall positions and storage locations.

To address the frame problem, the encoding employs inertia rules that assume fluents persist unless altered by actions. For example, if a crate is not moved in a step, its position remains the same.

Background Knowledge

The encoding incorporates background knowledge to define the environment and possible actions:

- **Grid Relations:** Defines spatial relationships like adjacency (`leftOf`, `below`) between cells.
- **Entity Definitions:** Identifies entities such as Sokoban and crates.
- **Constraints:** Ensures actions adhere to game rules, preventing movements into walls or pushing crates into occupied spaces.

Encoding the Problem

The ASP encoding translates the Sokoban puzzle into a formal representation that Clingo can process. The encoding encompasses several key components:

Goal Definition

The goal is to ensure that all crates are placed on storage locations at least once during the plan execution. The following ASP rules define the goal and related constraints (you can find it in "GOAL" section):

Explanation:

1. **Mutually Exclusive Goals and Walls:**
   - `:- isgoal(L), isnongoal(L).`
     A location cannot be both a goal and a non-goal.
   - `:- isgoal(L), wall(L).`
     A goal cannot be a wall.
   - `:- clear(L,T), time(T), wall(L).`
     Walls are never clear.
2. **Reaching Goals:**
   - `reachedGoal(C) :- crate(C), at(C,L,T), isgoal(L), time(T).`
     A crate is considered to have reached its goal if it is on a goal location at any time.
   - `:- crate(C), not reachedGoal(C).`
     All crates must reach their goals; otherwise, the plan is invalid.
3. **Entity Position Constraints:**
   - Sokoban cannot be in two places at the same time.
   - A crate cannot occupy multiple locations simultaneously.
   - Two different crates cannot be in the same cell.
   - Sokoban and a crate cannot occupy the same cell.
4. **Deadlock Detection:**

- A cell is a deadlock if it's adjacent to walls on both horizontal and vertical sides and is not a goal.
- Actions that push crates into deadlock cells or walls are prohibited to prevent unsolvable states.

Action Selection

You can find this encoding part in "ACTION SELECTION" section of sokoban.lp To ensure that exactly one action is chosen per time step, the following rule is implemented:

Explanation:

1. **Action Selection Rule:**
   - `0 { do(M,T) : move(M) } 1 :- time(T), T < maxsteps.`
     At each time step `T`, the solver can choose to perform at most one action from the set of possible moves (`move(M)`). The `0` indicates that choosing no action is also permissible, allowing the possibility of skipping a step if desired.
2. **Goal Achievement:**
   - `goal_achieved(T) :- time(T), #count { C : crate(C), not reachedGoal(C) } 0.`
     This rule defines that the goal is achieved at time `T` if there are zero crates that have not reached their goals by that time.
3. **Preventing Sokoban from Staying Idle:**
   - `:- at(sokoban, L, T), at(sokoban, L, T+1), T < maxsteps, not goal_achieved(T).`
     This constraint ensures that Sokoban does not remain in the same location across consecutive time steps unless the goal has already been achieved.

Defining Actions

All possible actions (`move` and `push`) are defined to generate ground actions that can be selected during planning. For detailed info see "6) DEFINING"move(M)" (GENERATING ALL POSSIBLE ACTIONS)" section of sokoban.lp

Explanation:

Each `move` predicate defines a possible action Sokoban can perform:

1. **Movement Actions:**
   - `moveLeft`, `moveRight`, `moveUp`, `moveDown`: Sokoban moves in the respective direction without interacting with crates.
2. **Push Actions:**
   - `pushLeft`, `pushRight`, `pushUp`, `pushDown`: Sokoban pushes a crate in the respective direction.
3. **Parameters:**
   - `S`: Sokoban entity.

- `X`: Current location of Sokoban.
- `Y`: Target location Sokoban moves to.
- `Z`: Target location where the crate is pushed to.
- `C`: Crate entity being pushed.

Inertia

Inertia rules ensure that the state of the game persists over time unless altered by actions. For detailed description of rules from this section see “7) INERTIA” section in sokoban.lp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 7) INERTIA (WITHOUT holds(...), a at(...,T) -> at(...,T+1)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% "If object O was in location L at moment T, and there is no proof
% that it stopped being there at moment T+1, then it remains."
at(O,L,T+1) :-
  at(O,L,T),
  not -at(O,L,T+1),
  time(T).

% Similarly for clear(L,T)
clear(L,T+1) :-
  clear(L,T),
  not -clear(L,T+1),
  time(T).
```

Explanation:

1. **Persistence of Positions:**
   - If an object `O` is at location `L` at time `T` and there is no evidence (`not -at(O,L,T+1)`) that it has moved, then it remains at `L` at time `T+1`.
2. **Persistence of Clear Cells:**
   - Similarly, if a cell `L` is clear at time `T` and there is no evidence that it has become blocked, it remains clear at time `T+1`.

Preconditions

Preconditions ensure that actions are only executed when the necessary conditions are met.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 8) PRECONDITIONS (CONSTRAINTS)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If the action moveLeft(S,X,Y) is selected, then Sokoban S MUST be at X at step T,
% and Y MUST be clear at step T.

% moveLeft
```

```
:- do(moveLeft(S,X,Y), T), not at(S,X,T).
:- do(moveLeft(S,X,Y), T), not clear(Y,T).

% moveRight
:- do(moveRight(S,X,Y), T), not at(S,X,T).
:- do(moveRight(S,X,Y), T), not clear(Y,T).

% moveUp
:- do(moveUp(S,X,Y), T), not at(S,X,T).
:- do(moveUp(S,X,Y), T), not clear(Y,T).

% moveDown
:- do(moveDown(S,X,Y), T), not at(S,X,T).
:- do(moveDown(S,X,Y), T), not clear(Y,T).

% pushLeft
:- do(pushLeft(S,X,Y,Z,C), T), not at(S,X,T), crate(C).
:- do(pushLeft(S,X,Y,Z,C), T), not at(C,Y,T), crate(C).
:- do(pushLeft(S,X,Y,Z,C), T), not clear(Z,T), crate(C).

% pushRight
:- do(pushRight(S,X,Y,Z,C), T), not at(S,X,T), crate(C).
:- do(pushRight(S,X,Y,Z,C), T), not at(C,Y,T), crate(C).
:- do(pushRight(S,X,Y,Z,C), T), not clear(Z,T), crate(C).

% pushUp
:- do(pushUp(S,X,Y,Z,C), T), not at(S,X,T), crate(C).
:- do(pushUp(S,X,Y,Z,C), T), not at(C,Y,T), crate(C).
:- do(pushUp(S,X,Y,Z,C), T), not clear(Z,T), crate(C).

% pushDown
:- do(pushDown(S,X,Y,Z,C), T), not at(S,X,T), crate(C).
:- do(pushDown(S,X,Y,Z,C), T), not at(C,Y,T), crate(C).
:- do(pushDown(S,X,Y,Z,C), T), not clear(Z,T), crate(C).
```

Explanation:

Each precondition rule ensures that specific conditions are met before an action can be executed. If the preconditions are not satisfied, the action is deemed invalid, and the solver will not consider it as part of a valid plan.

1. **Move Actions:**
   - Sokoban must be at the starting location `X` at time `T`.
   - The target location `Y` must be clear at time `T`.
2. **Push Actions:**
   - Sokoban must be at the starting location `X` at time `T`.
   - The crate `C` must be at location `Y` at time `T`.

- The destination location `Z` for the crate must be clear at time `T`.

Positive and Negative Effects

These rules define how actions change the state of the game by updating the positions of Sokoban and crates, as well as the status of cells (clear or blocked).

Move Left

```
%========== moveLeft ==========
% (a) If Sokoban moved from X to Y, then at T+1 Sokoban is in Y
at(S, Y, T+1) :-
  do(moveLeft(S,X,Y), T),
  at(S,X,T),
  clear(Y,T).

% (b) Remove Sokoban from X at T+1
-at(S, X, T+1) :-
  do(moveLeft(S,X,Y), T),
  at(S,X,T).

% (c) The old cell X becomes clear at T+1
clear(X, T+1) :-
  do(moveLeft(S,X,Y), T),
  at(S,X,T),
  clear(Y,T).

% (d) The new cell Y is no longer clear at T+1
-clear(Y, T+1) :-
  do(moveLeft(S,X,Y), T),
  at(S,X,T),
  clear(Y,T).
```

Explanation:

1. **Effect (a):**
   - Sokoban moves to location `Y` at time `T+1` if it was at `X` and `Y` was clear at time `T`.
2. **Effect (b):**
   - Sokoban is no longer at location `X` at time `T+1`.
3. **Effect (c):**
   - The original location `X` becomes clear at time `T+1` since Sokoban has moved out.
4. **Effect (d):**
   - The new location `Y` is no longer clear at time `T+1` because Sokoban has moved into it.

Move Right, Move Up, Move Down Similar rules are defined for `moveRight`, `moveUp`, and `moveDown`, adjusting the positions accordingly based on the direction of movement.

Push Actions Push actions involve moving both Sokoban and a crate. Here's the detailed encoding for `pushLeft`:

```
%========== pushLeft ==========
% pushLeft(S,X,Y,Z,C): Sokoban S pushes crate C from Y to Z, moving from X to Y

% Sokoban moves to Y
at(S, Y, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(S,X,T),
  at(C,Y,T),
  clear(Z,T).

% Sokoban is no longer at X
-at(S, X, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(S,X,T).

% Crate C moves to Z
at(C, Z, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(S,X,T),
  at(C,Y,T),
  clear(Z,T).

% Crate C is no longer at Y
-at(C, Y, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(C,Y,T).

% Original cell X becomes clear
clear(X, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(S,X,T),
  at(C,Y,T),
  clear(Z,T).

% Destination cell Z is no longer clear
-clear(Z, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(S,X,T),
  at(C,Y,T),
```

```
  clear(Z,T).

% Intermediate cell Y is now occupied by Sokoban, so it is no longer clear
-clear(Y, T+1) :-
  do(pushLeft(S,X,Y,Z,C), T),
  at(S,X,T),
  at(C,Y,T),
  clear(Z,T).
```

Explanation:

1. **Effect on Sokoban:**
   - Sokoban moves from `X` to `Y` at time `T+1`.
   - Sokoban is no longer at `X`.
2. **Effect on Crate:**
   - Crate `C` is moved from `Y` to `Z` at time `T+1`.
   - Crate `C` is no longer at `Y`.
3. **Effect on Cells:**
   - Original cell `X` becomes clear since Sokoban has moved out.
   - Destination cell `Z` becomes occupied by crate `C`, so it is no longer clear.
   - Intermediate cell `Y` is now occupied by Sokoban, so it is no longer clear.

Similar rules are defined for `pushRight`, `pushUp`, and `pushDown`, adjusting the positions of Sokoban and crates based on the direction of the push.

Minimizing the Number of Steps

To optimize the solution for the minimal number of steps, the following ASP directives are used:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 10) MINIMIZE THE NUMBER OF STEPS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#minimize{1,T : do(M,T)}.
#minimize{T : do(_,T)}.
```

Explanation:

1. **Minimizing Actions:**
   - `#minimize{1,T : do(M,T)}.`
     This directive minimizes the total number of actions taken across all time steps `T`.
2. **Minimizing Time Steps:**
   - `#minimize{T : do(_,T)}.`
     This directive minimizes the highest time step `T` at which any action is taken, effectively reducing the plan's duration.

These minimization directives guide the ASP solver to find the most efficient plan that achieves the goal in the fewest possible steps.

Constraints on Walls and Crates

To ensure the integrity of the game environment, the following constraints maintain the consistency of walls and crates throughout the plan execution:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Constraints: The number of walls and crates does not change
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Define the total number of walls on the map
total_walls(N) :- N = #count{L : wall(L)}.

% Constraint: For each moment T, the number of walls remains equal to N
:- time(T), total_walls(N), N != #count{L : wall(L)}.

% Define the total number of crates on the map
total_crates(M) :- M = #count{C : crate(C)}.

% Constraint: For each moment T, the number of crates remains equal to M
:- time(T), total_crates(M), M != #count{C : at(C,L,T), crate(C)}.
```

Explanation:

1. **Total Walls:**
   - `total_walls(N) :- N = #count{L : wall(L)}.`
     Counts the total number of wall cells on the map and assigns it to `N`.
   - `:- time(T), total_walls(N), N != #count{L : wall(L)}.`
     Ensures that at every time step `T`, the number of walls remains constant. Any deviation implies an invalid plan.
2. **Total Crates:**
   - `total_crates(M) :- M = #count{C : crate(C)}.`
     Counts the total number of crates on the map and assigns it to `M`.
   - `:- time(T), total_crates(M), M != #count{C : at(C,L,T), crate(C)}.`
     Ensures that the number of crates remains constant across all time steps. Adding or removing crates is prohibited.

These constraints maintain the game's rules by preventing the unintended creation or destruction of walls and crates during the solving process.

Output Actions

To facilitate the extraction of the solution steps, the following directives specify which predicates should be displayed in the output:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 11) OUTPUT ACTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#show do/2.
#show total_crates/1.
#show total_walls/1.
```

Explanation:

1. **Action Display:**
   - `#show do/2.`
     Displays all `do(Action, Time)` predicates, which represent the actions taken at each time step.
2. **Summary Information:**
   - `#show total_crates/1.`
     Displays the total number of crates.
   - `#show total_walls/1.`
     Displays the total number of walls.

These directives ensure that the ASP solver provides a clear and concise output of the actions taken to solve the Sokoban puzzle, along with essential summary information about the game environment.

Background Knowledge

The background knowledge section lays the foundation for accurately modeling the Sokoban environment and its dynamics. It ensures that the solver understands the spatial layout, entity properties, and the rules governing interactions within the grid.

1. Exclusivity of Place   Concept:

In Sokoban, each object—whether Sokoban himself or a crate—must occupy a unique location on the grid at any given moment. This exclusivity ensures that no two objects share the same cell simultaneously, maintaining the game's integrity and preventing ambiguous states.

Details:

- **Sokoban's Unique Position:** Sokoban cannot be in multiple locations at the same time.
- **Crate's Unique Position:** Each crate occupies only one location at any time step.
- **No Overlapping of Crates:** Multiple crates cannot be placed in the same cell simultaneously.
- **Sokoban and Crate Separation:** Sokoban and a crate cannot occupy the same cell.

Rationale:

By enforcing these exclusivity constraints, the encoding eliminates invalid states

where objects overlap, thereby streamlining the solver's search process and reducing computational overhead.

2. Prohibition of Pushing into Occupied Cells Concept:
Crates cannot be pushed into cells that are already occupied by walls, other crates, or Sokoban. This constraint is crucial to prevent the formation of unsolvable configurations and to maintain the logical consistency of the puzzle.

Details:

- **Preventing Push into Deadlocks and Walls:** Actions that would result in pushing a crate into a deadlock cell or a wall are prohibited.

Rationale:
By disallowing pushes into occupied cells, the encoding ensures that every push action maintains the game's solvability and adheres to its physical rules. This prevents the solver from considering infeasible or invalid action sequences.

3. Deadlock Detection Concept:
Deadlocks occur when a crate is pushed into a position from which it cannot reach any goal location, effectively rendering the puzzle unsolvable from that state. Detecting and preventing deadlocks is pivotal for enhancing the solver's performance and ensuring the feasibility of action sequences.

Details:

- **Deadlock Identification:** A cell is a deadlock if it's adjacent to walls on both horizontal and vertical sides and is not a goal.
- **Action Prohibition:** Actions that push crates into deadlock cells or walls are prohibited to prevent unsolvable states.

Rationale:
By proactively identifying and blocking deadlock scenarios, the encoding significantly reduces the search space, allowing the ASP solver to focus on viable action sequences. This not only enhances performance by avoiding futile paths but also ensures that the generated solutions are genuinely solvable.

4. Static Walls and Goal Locations Concept:
Walls and goal locations are immutable features of the Sokoban grid. Their static nature means that their positions do not change over time, regardless of the actions taken by Sokoban. This immutability is fundamental to maintaining the puzzle's structure and defining valid action preconditions.

Details:

- **Static Walls:** Walls are declared as static entities that occupy specific locations on the grid.
- **Static Goal Locations:** Goal locations are similarly declared as static cells where crates must be placed.

- **Consistency Constraints:** Constraints prevent goal locations from being walls and ensure that walls remain consistently blocked across all time steps.

Rationale:
Maintaining walls and goal locations as static elements simplifies the encoding by eliminating the need to account for their movement or alteration. This stability is crucial for accurately defining action preconditions and effects, as well as for enforcing the rules that govern Sokoban's interactions within the grid.

5. Spatial Relations and Grid Geometry Concept:
Understanding the spatial layout of the Sokoban grid is essential for defining valid movements and pushes. Spatial relations such as adjacency (`leftOf`, `below`) facilitate the encoding of movement directions and interactions between objects.

Details:

- **Defining Adjacency:** Predicates like `leftOf` and `below` establish the relative positions of cells, enabling the encoding to determine valid movement directions.
- **Deadlock Detection Based on Adjacency:** Adjacency predicates are utilized to identify deadlock conditions based on surrounding walls.

Rationale:
By explicitly defining spatial relationships, the encoding can accurately model Sokoban's movement mechanics and the spatial constraints that govern valid actions. This foundational knowledge is pivotal for ensuring that actions like moving or pushing are contextually appropriate within the grid's geometry.

6. Clear Cells and Their Dynamics Concept:
A cell is considered clear if it is unoccupied by any object (Sokoban or crates) and not a wall. Clear cells are essential for determining valid movement and push destinations.

Details:

- **Initial Clear Cells:** Initial states declare certain cells as clear based on the map input.
- **Updating Clear Cells Over Time:** Clear cell statuses are updated based on actions that occupy or vacate cells.

Rationale:
Managing the state of clear cells allows the encoding to enforce movement constraints, ensuring that Sokoban and crates can only move into or through unoccupied spaces. This dynamic tracking is crucial for maintaining the game's logical consistency throughout the planning process.

7. Prohibiting Sokoban from Remaining Idle Concept:
To promote progress towards the goal, the encoding includes constraints that prevent Sokoban from staying in the same position across consecutive time steps unless the goal has been achieved. This encourages the solver to actively seek actions that contribute to solving the puzzle.

Details:

- **Idle Movement Constraint:** Constraints ensure that Sokoban does not remain stationary unless all crates are already on their goal locations.

Rationale:
By preventing Sokoban from idling, the encoding ensures that each action contributes towards reaching the goal, thereby enhancing the efficiency of the solver and reducing unnecessary exploration of non-progressive states.

8. Ensuring Consistency of Walls and Crates Concept:
Walls and crates are integral to defining the puzzle's structure and the possible actions. Ensuring their consistency across all time steps is vital for maintaining the puzzle's rules and preventing unintended alterations during the solving process.

Details:

- **Consistent Number of Walls:** Rules enforce that the number of walls remains constant across all time steps.
- **Consistent Number of Crates:** Constraints ensure that the number of crates does not change throughout the plan execution.

Rationale:
Maintaining a consistent number of walls and crates is essential for preserving the puzzle's original configuration and ensuring that the solver operates within the defined boundaries of the game environment.

9. Spatial Constraints and Movement Validity Concept:
Validating the legality of movements and pushes based on the spatial arrangement of walls and other objects is crucial for generating feasible action sequences. Spatial constraints ensure that actions like moving or pushing adhere to the game's physical limitations.

Details:

- **Movement Constraints:** Rules ensure that Sokoban is in the correct starting position and that the target cell is clear before executing a move.
- **Push Constraints:** Constraints validate that Sokoban and the targeted crate are in the correct positions and that the destination cell is clear before executing a push.

Rationale:
By embedding these spatial constraints, the encoding ensures that only legiti-

mate and feasible actions are considered during the planning process, thereby enhancing the solver's accuracy and efficiency.

Background Knowledge

The background knowledge section lays the foundation for accurately modeling the Sokoban environment and its dynamics. It ensures that the solver understands the spatial layout, entity properties, and the rules governing interactions within the grid.

1. Exclusivity of Place Concept:
In Sokoban, each object—whether Sokoban himself or a crate—must occupy a unique location on the grid at any given moment. This exclusivity ensures that no two objects share the same cell simultaneously, maintaining the game's integrity and preventing ambiguous states.

Details:

- **Sokoban's Unique Position:** Sokoban cannot be in multiple locations at the same time.
- **Crate's Unique Position:** Each crate occupies only one location at any time step.
- **No Overlapping of Crates:** Multiple crates cannot be placed in the same cell simultaneously.
- **Sokoban and Crate Separation:** Sokoban and a crate cannot occupy the same cell.

Rationale:
By enforcing these exclusivity constraints, the encoding eliminates invalid states where objects overlap, thereby streamlining the solver's search process and reducing computational overhead.

2. Prohibition of Pushing into Occupied Cells Concept:
Crates cannot be pushed into cells that are already occupied by walls, other crates, or Sokoban. This constraint is crucial to prevent the formation of unsolvable configurations and to maintain the logical consistency of the puzzle.

Details:

- **Preventing Push into Deadlocks and Walls:** Actions that would result in pushing a crate into a deadlock cell or a wall are prohibited.

Rationale:
By disallowing pushes into occupied cells, the encoding ensures that every push action maintains the game's solvability and adheres to its physical rules. This prevents the solver from considering infeasible or invalid action sequences.

3. Deadlock Detection Concept:
Deadlocks occur when a crate is pushed into a position from which it cannot reach any goal location, effectively rendering the puzzle unsolvable from that state. Detecting and preventing deadlocks is pivotal for enhancing the solver's performance and ensuring the feasibility of action sequences.

Details:

- **Deadlock Identification:** A cell is a deadlock if it's adjacent to walls on both horizontal and vertical sides and is not a goal.
- **Action Prohibition:** Actions that push crates into deadlock cells or walls are prohibited to prevent unsolvable states.

Rationale:
By proactively identifying and blocking deadlock scenarios, the encoding significantly reduces the search space, allowing the ASP solver to focus on viable action sequences. This not only enhances performance by avoiding futile paths but also ensures that the generated solutions are genuinely solvable.

4. Static Walls and Goal Locations Concept:
Walls and goal locations are immutable features of the Sokoban grid. Their static nature means that their positions do not change over time, regardless of the actions taken by Sokoban. This immutability is fundamental to maintaining the puzzle's structure and defining valid action preconditions.

Details:

- **Static Walls:** Walls are declared as static entities that occupy specific locations on the grid.
- **Static Goal Locations:** Goal locations are similarly declared as static cells where crates must be placed.
- **Consistency Constraints:** Constraints prevent goal locations from being walls and ensure that walls remain consistently blocked across all time steps.

Rationale:
Maintaining walls and goal locations as static elements simplifies the encoding by eliminating the need to account for their movement or alteration. This stability is crucial for accurately defining action preconditions and effects, as well as for enforcing the rules that govern Sokoban's interactions within the grid.

Implementation

The implementation consists of two primary components:

1. **ASP Encoding (`sokoban.lp`):**
   - Contains the 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.
- **Testing (`test_solver.py` & `conftest.py`):** Implements automated tests using `pytest` to validate the correctness of the encoding and solver.

Workflow

1. **Map Input:** Users provide Sokoban maps in plain text files following predefined encoding rules.
2. **Fact Generation:** The Python solver reads the map and generates corresponding ASP facts.
3. **Solving:** Clingo processes the facts alongside the ASP encoding to find a sequence of actions that solves the puzzle.
4. **Visualization:** The GUI visualizer displays the initial map and animates each step of the solution, allowing users to observe the solving process.

Key Modules

- **`solver.py`:** Handles interaction with Clingo, manages the solving process, and formats the solution.
- **`visualizer.py`:** Implements the Tkinter-based GUI, facilitating user interaction and visualization.
- **`test_solver.py`:** Contains test cases to ensure the solver behaves as expected across different maps.
- **`conftest.py`:** Configures `pytest` to handle test cases and command-line options.

Detailed Implementation Aspects

- **Fact Generation:** The solver parses the input map, identifies walls, crates, storage locations, and Sokoban's initial position, and translates these into ASP facts.
- **Action Modeling:** Each possible action (`move`, `push`) is encoded with its preconditions and effects. The solver iteratively searches for a sequence of actions that transitions the game state from the initial to the goal state.
- **Inertia Rules:** To efficiently handle the frame problem, the encoding includes rules that maintain the state of entities unless explicitly changed by an action.
- **Optimization:** The solver seeks the minimal number of steps required to solve the puzzle, enhancing both performance and solution quality.

Solving the Frame Problem

The frame problem is a fundamental challenge in AI planning, dealing with representing what remains unchanged in the world after an action is performed. In Sokoban's ASP encoding, solving the frame problem ensures that only relevant aspects of the game state are updated while everything else remains consistent.

Approach in Encoding

1. **Inertia Rules:**
   - **Positions Persistence:**
     ```
     at(O,L,T+1) :-
       at(O,L,T),
       not -at(O,L,T+1),
       time(T).
     ```
     If an object `O` is at location `L` at time `T` and there is no evidence that it has moved (`not -at(O,L,T+1)`), then it remains at `L` at time `T+1`.
   - **Clear Cells Persistence:**
     ```
     clear(L,T+1) :-
       clear(L,T),
       not -clear(L,T+1),
       time(T).
     ```
     Similarly, if a cell `L` is clear at time `T` and remains unaffected, it stays clear at time `T+1`.
2. **Negative Effects:**
   - Actions that move Sokoban or crates generate negative effects by explicitly removing their previous positions. For example:
     ```
     -at(S, X, T+1) :-
       do(moveLeft(S,X,Y), T),
       at(S,X,T).
     ```
     This rule ensures that Sokoban is no longer at location `X` after performing `moveLeft`.
3. **Selective Updates:**
   - Only the entities directly involved in an action have their states updated. All other entities and cells remain unchanged due to the inertia rules.
4. **Avoiding Over-specification:**
   - By using `not -at(O,L,T+1)`, the encoding avoids the need to specify all non-effects explicitly, thereby preventing unnecessary constraints and keeping the encoding concise.

Experience and Reflections

Working on this project provided profound insights into the application of declarative programming for solving intricate planning problems. Modeling Sokoban in ASP underscored the elegance and expressiveness of logic programming.

Challenges

- **Frame Problem Handling:** Ensuring that only relevant state changes were encoded required meticulous design of inertia rules.
- **Performance Optimization:** Balancing the expressiveness of the encoding with the solver's efficiency necessitated iterative refinements.
- **GUI Integration:** Seamlessly connecting the ASP solver with a user-friendly interface involved careful synchronization of data and actions.

Key Learnings

- **ASP Proficiency:** Gained proficiency in writing efficient ASP encodings and leveraging Clingo's capabilities for optimization.
- **Problem Decomposition:** Learned to decompose a real-world problem into formal logical representations.
- **Testing and Validation:** Emphasized the importance of comprehensive testing to ensure reliability across various scenarios.

Conclusion

This project successfully demonstrated the feasibility of modeling and solving Sokoban as a planning problem using Answer Set Programming. 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 could explore extending the solver to handle more complex variations of Sokoban, incorporating heuristic-based optimizations, and enhancing the visualization features for a better user experience.