SOKOBAN SOLVER

Computational Logic 2024

Formalization of the Sokoban game as a planning problem. Representation of game states and actions using Answer Set Programming (ASP). Development of a solver to compute solutions for the Sokoban game.

Roman Božik, 18.12.2024

1. Introduction to the Problem

Sokoban is a logic-based puzzle game originating from Japan first introduced in 1982. The word Sokoban translates to warehouse keeper (thus reflecting game problem).The player controls a character (Sokoban) who must push crates onto defined storage positions. Importantly, the player can only push crates, and nothing else. The game board consists of walls, crates, target storage positions and space emptyness. The challenge lies in determining a sequence of moves that successfully solves the level without trapping crates in unsolvable positions or trapping Sokoban in some weird position.

The goal of this project is to develop an application that can automatically solve Sokoban puzzles and present the solution in a user-friendly and visually appealing way. The solution will include step-by-step animations, allowing users to observe and understand the sequence of moves performed by the solver.To achieve this the project will use knowledge acquired from the Computational Logic course. The Sokoban game will be formalized as a planning problem and implemented using Answer Set Programming (ASP)and the Clingo solver.

2. Gameplay

Game is played on grid like board where the player must push crates to designed storage position. The player can move in four directions: up, down, left,right. A crate can only be pushed if it is adjacent to the player and the space directly behind the crate in the direction of the push is empty respectively contains storage position. The player can only interact with crates so that he pushes them, cannot pull them and also can not move through walls, or pass through other crates. A crate is successfully placed when it is placed onto a storage position. When all craters ale placed onto a storage position, the level is solved successfully. The primary challenge of the game lies in avoiding situations where crates are pushed onto positions that make them immovable which makes the level unsolvable.

3. Formalization

3.1.Formalization of the Game Board

Game board is represented as a set of facts created directly from an input file. Each input file contains a grid representation of the game, where specific characters correspond to different elements of the board. The representation is as follows:

- # – wall
- C – crate
- X – storage position
- S – player position
- s – player on a storage position
- c – crate on a storage position

According to this representation will be make predicates representing major game objects:

- wall(X, Y) – Represents the presence of a wall at coordinates (X, Y) on the game board.
- direction(Direction, X, Y) – Defines movement directions with their respective coordinate changes.
- crate(T, X, Y) – Represents the position of a crate at coordinates (X, Y) at time step T.
- storage(X, Y) – Represents a storage position at coordinates (X, Y).

3.2.Actions

The actions reflects the available moves that can be performed when gameplay which are push and move. These actions represent the player's ability to move in the game board and interact with crates.To accurately describe actions, the concept of time muse be introduced. Time is used to define the order of things on the board. All movable elements, such as the player's movements (move) and crate interactions (push), are labeled with a specific time (T) that represents the exact moment an action fires
Thus actions will be represented by following predicates:

- time(T) – Represent the time
- move(T,X ,Y) – Represents the position of Sokoban when time elapse
- push(T, X1, Y1, X2, Y2, X3, Y3) – Pushes crate from X2/Y2 position into X3/Y3 position and move sokoban from X1/Y1 into X2/Y2 position in heading direction. All happening at the same time.

3.3. Defining rules

For defining exact ASP rules used for solving sokoban solver we must first figure out how the player can move across the game board, so that he didn't broke any game rules (walking through walls for example).

The first step is to define directions. Directions specify the possible moves the player can make, such as up, down, left, and right. These directions are represented as coordinate changes on the grid as follows:

- direction(up, 0, -1).
- direction(down, 0, 1).
- direction(left, -1, 0).
- direction(right, 1, 0).

After that we need to enforce the constraints that prevent the player from moving into unwanted positions, so, we need rules that block movement if a wall or crate is present at the target position in perspective of future sokoban position (thus T+1):

- :- move(T+1, X, Y), wall(X, Y).
- :- move(T+1, X, Y), crate(T, X, Y).

We cannot forget to define our maximum step clausule. We will do it by creating a generator, that will generate exact amount of time predicates, so it mirrors our expected count of steps:

- #const maxSteps={times}.
  time(0..maxSteps).

For defining move action correctly, we need to check if move respects the limits of the maximum allowed steps, then, it is necessary to verify whether the Sokoban is currently located at the starting position of the move that means that there must exist a record of the player's position at the previous time step. Once this is confirmed, the direction of movement must be defined, then a command to execute the move should be issued. Next a check is performed to verify that the future position does not contain any walls or crates, and move must be performed after push, therefore:

- move(T+1, X2, Y2) :-time(T)
  ,move(T,X1 ,Y1),
  direction(Smer, DX ,DY),
  vykonaj(T,Smer),
  X2=X1+ DX,
  Y2=Y1+DY,
  not wall(X2,Y2),
  not crate(T, X2, Y2).
- move(T+1, X2, Y2) :- push(T, _, _, X2, Y2, _, _).

Push action is very similar, only real difference is, that you check for a existing crate for this time:

- push(T, X1, Y1, X2, Y2, X3, Y3) :- time(T),
  move(T, X1, Y1),
  direction(Smer, DX, DY),
  vykonaj(T, Smer),
  X2 = X1+ DX,
  Y2 = Y1 + DY,
  crate(T, X2, Y2),
  X3 = X2 + DX, Y3 = Y2 +DY,
  not wall(X3, Y3),not crate(T, X3, Y3).

When push is performed, it is necessary to update positions of unpushed crates to actual time and create a new crate for exchange for old pushed:

- crate(T+1, X, Y) :- time(T), crate(T, X, Y), not push(T, _, _, X, Y, _, _).
- crate(T+1, X3, Y3) :- push(T,_, _, _, _, X3, Y3).

3.4.Finding solution using generator

Now we have everything we need to begin searching for a solution. For finding one we use generators. Generators allow produce actions at each time step so that the Sokoban can perform exactly one move in a defined direction, provided that the game has not yet reached its end state. This guarantees that the solver considers all valid moves step by step, coming toward the solution. We also need to define final state, so its state when all crates at one time period are at storage position:

- {{ vykonaj(T, Smer) : direction(Smer, _, _) }} = 1 :- time(T),not theend(T).
- boks_out_of_storidz(T) :- time(T), crate(T,X,Y), not storage(X,Y).
- theend(T) :- time(T), not boks_out_of_storidz(T).

4. Conclusion

This file presented a solution to the Sokoban game using Answer Set Programming. During the formalization, I aimed to divide each step into the smallest possible components to ensure that the rules and logic were as simple as possible to understand and implement.The current solution utilizes a single-step generator that iterates through all possible directions at each time. However, the solution could be further improved by exploring alternative search strategies, such as more efficient methods for generating moves or optimizing the search all direction traversal. This could allow for faster and more efficient solution to complex Sokoban levels solution finding.