CS271 Project: Sokoban Solver

### Team

###### Name

Emmm

###### Students

|  |  |  |
| --- | --- | --- |
| Name | Email | Student ID |
| Kush Dave | kkdave@uci.edu | 91443886 |
| Yinan Zhou | yinanz17@uci.edu | 69964523 |
| Sriram Rao | srirar1@uci.edu | 17567152 |

### Outline

We aim to implement the Q-learning algorithm to build our Sokoban Solver. For this, we first define the states, termination condition, actions and rewards. Then, we describe our proposed algorithm with some detail. We follow with a brief explanation of the algorithm and steps we intend to take to implement it in Python.

### Environment Setup

In this section, we define the state, termination conditions, permitted actions, and rewards.

We also specify the initialisation for the Q-value matrix.

###### State

We define a state as an arrangement of the person and block locations in the map.

###### Termination condition

The goal is to place all boxes in one of the storage locations each. This forms a favourable termination condition.

Another state that is terminated is when a box (any box) is not in a storage location but cannot be moved. This is an unfavourable termination condition.

We intend to differentiate between these two types by assigning appropriate rewards for these termination/goal states: a favourable reward for placing a box in a storage location, and a reward to represent negative infinity for ending up with a box that cannot be moved.

###### Actions

Defining an action well in our Sokoban environment is important. We could define each movement by the person/player (up, down, left, right) as an action. We choose not to do this because it would cause our state space to become unnecessarily large. We need our solver to move the blocks to storage locations. Therefore, we define an action as moving a box in a direction it is allowed to move in.

A box is allowed to move when there is no obstruction in the desired direction. An obstruction could be a wall or another box itself.

###### Rewards

We first explain the reasoning and idea behind choosing rewards and then define the computation for the reward of an action.

Since we need our Sokoban solver to complete the puzzle in as few moves/operations as possible, our reward will have a constant cost for each movement of the person/player. Moving a box is desirable. So, we want to define a higher reward for moving/pushing a box than for moving without a box.

Moving a box into a storage location is desirable. Hence, we must define a better reward for this action than moving a box around outside a storage location.

From initial instances of the game, we see that there are realistic situations where a box must be moved out of a storage location in order to clear the way for another box to reach a desired storage location. To allow this to happen in our algorithm, moving a box out of the storage location must simply undo it’s high reward so far (while maintaining the negative reward convention for the number of moves), and nothing more.

With these ideas in mind, we define the reward for an action a in a state s as:

R(s, a) = Distance(current location to the box and one move) \* (Reward for movements: R1) +   
 (Reward for one movement of the box in the desired direction: R2)

Here, the *Distance* term represents the number of moves needed to reach the box on the required adjacent location, and one move to push the box. We explain “required” with an example: to move a box one step to the right side, the player must be directly adjacent to the box to its left side.

Reward R1 is the fixed cost of movement as explained earlier.

Reward R2 depends on whether the current location of the box and its location after this action is a storage location, and whether we reached an unfavourable terminal state.   
It encapsulates five types of movements and we can assign a reward for each type:

1. Any box not free to move after this action (Unfavourable termination).
2. Non-storage location to non-storage location with all boxes free to move.
3. Non-storage location to storage location with all boxes free to move.
4. Storage location to non-storage location with all boxes free to move.
5. All boxes in a storage location each (Favourable termination).

Rewards R1 and R2 are hyperparameters that will be assigned and tuned.

###### Q-value initialisation

We initialise the quality values for actions to show a tendency to move a box to the nearest storage location after it is moved in the direction of the action.

### Solver Algorithm

##### Data Structures

###### State

Map representation of the environment

Implemented as a dictionary of dictionaries in Python to represent rows and columns. The first level dictionaries are rows which index second level dictionaries that are columns which index a sokoban grid object.

Each point on the grid is associated with the following information:

1. Coordinates: (x,y)

2. Type: (Empty, Terminal, Player, Box or Wall).

3. Certain quick check functions (See grid object algorithm section below)

The locations of boxes on the grid as well as terminal locations will be stored as a set of tuples.

###### Q-value

Hashmap from (state, action) to numerical quality value.

Depends on the current state and on the state the action leads to:

###### Action

Action encapsulates the following:

1. The location of the box, and
2. The direction to move the box.

##### Algorithm

###### Initialisation step

*Algorithm*:

Initialise Q-values

Read the input map with starting location

*Explanation:*

Setting up the program with the input map and initial Q-values.

The Q-values are initialised to an estimate of that state.

**~~Options~~**~~:~~

1. ~~Shortest distance from a box to its closest storage location~~

###### Learning stage

*Algorithm:*

while(time < time allotted for learning):

Reset state, reward, terminated.

while (not terminated):

action ← chooseAction(state)

next\_state, reward, terminated ← perform(state, action)

best\_action ← next\_action with best q-value(next\_state, next\_action).

new\_value ← old\_value + learning \* [R + discount \* q-value(next\_state,

best\_action) - old\_value]

q-value(state, action) ← new\_value

state ← new\_state

*Explanation:*

This is the straightforward Q-learning algorithm.

The value update step is done using the best action of the next step.

SARSA: value update step is done using the action actually taken in the next step.

###### Choosing Actions

*Algorithm:*

chooseAction(state):

e = random value between 0 and 1

if (e < epsilon): // explore

action = random permitted action of current state

else: // exploit

action = best known action of current state

*Explanation:*

This is the *f function* as referred to in the textbook.

What we have here is a simplistic function to choose actions based on a chosen constant epsilon value. This epsilon value decides whether to explore or exploit in this step.

The *f function* be replaced by one that takes into consideration the frequency of chosen actions to make the exploration/exploitation decision better informed.

###### Performing Action

*Algorithm:*

perform(state, action):

next\_state ← move block (update box set) and update board

terminal ← isTerminal(next\_state)

reward ← calculateReward(state, action, next\_state)

*Explanation:*

We call other helpers to find the parameters needed to update the Q-value and to progress in the learning process.

The “next state” is the state of the environment after moving to the desired location adjacent to the block and then pushing the box.

###### Checking Termination

*Algorithm:*

isTerminal(state):

if (boxes\_set intersection terminal\_set = boxes\_set)

return favourable termination

return not terminal

*Explanation:*

Here, we check whether the current state is terminal.

Favourable termination: All boxes are in a storage location each

If this condition is not met, the state is not terminal.

###### Calculate Reward

*Algorithm:*

calculateReward(state, action, next\_state):

reward ← movement cost till action.location // R1

reward ← reward + reward for moved box location (estimatedValueOf(new\_state)) // R2

return reward

*Explanation:*

Since our action is defined as moving a box, we include the cost of player movements to the box in the reward (R1).

‘The second component (R2) of the reward depends on the location of the box after movement. It will be estimated using “estimatedValueOf” function. This differentiates between the resting location of the box after movement as explained earlier in Rewards section.

### Provide assessment/evaluation of the time/space complexity of your algorithms

**[To be filled]**

### References

1. Wikipedia: <https://en.wikipedia.org/wiki/Sokoban>
2. Q-learning example: <https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/>
3. Artificial Intelligence - A Modern Approach (4th Edition) by Stuart Russell, Peter Norvig
4. Class presentation on Markov Decision Processes and Reinforcement Learning
5. Online Sokoban game: <https://www.mathsisfun.com/games/sokoban.html>

Helper function:

‘’’

We use this function to decide if an action is illegal

‘’’

isDeadPosition(x:int, y:int) -> bool:

Check if board[x][y] will be a dead end position for a box on [x][y]

Return true if yes

Global variables : learning rate alpha, discount rate r

We need location of boxes to decide what actions we can perform

Also need it to calculate

Box locations data structure : list

Terminal locations: set

Algorithm action sequence: