University of Queensland

COMP7702 Assigment 1 - Report

Sokoban - Discrete Search

Author:
Duy Long Tran

Student Number:
44992305

August 22, 2019

1 Define the Agent

The agent for a Sokoban solver is defined as follows:

1. Action space (A): The player moves lelf (L), right (R), up (U), down (D)

2. Percept space (P): The position combinations of boxes, player, targets, and obstacles

3. State space (S): The position combinations of boxes and player only, since the targets and obstacles are static, this can be represented as a sequence of 2-D coordinates

4. World dynamics (T): The change from one state to another, given a movement in action space

5. Percept function ($\mathbf{Z = S \rightarrow P}$): An identity mapping of P

6. Utility function (U): +1 for goal state, 0 for all other state

Thus, the agent is discrete (since the number of possible states is countable), fully observable (since it can percept all the world dynamics, given it's current state), deterministic (since one movement lead to only one state), and static (since the only state of the agent changes when it take a movement, the rest of the map doesn't change). Having the definition in hand, we can design a solver under the following structures:

1. State representation: Since the targets and obstacles do not change, then the state includes only the positions of the boxes and the player, which is stored in a tuple of all boxes' coordinates, followed by the player's coordinates. Here I put the position of player to the last element for easier process. For example, the map including 2 boxes at (1,2) and (3,4), and the player at (5,6) is represented as ((1,2),(3,4),(5,6)).

2. Neighborhood: The map_neighbors() function returns a list of the next states given the current state and the obstacles map. The function runs through 4 cells (according to 4 possible moves), and check if the cells are blocked. The new states are in the form of the "State representation" described above.

3. Goal test: The finish() function takes a state and a list of targets, and check if all element in the state, (except for the last element which represents the player), are in the list of targets, then returns True. Otherwise, returns False.

4. Cost function: The cost function for Uniform cost search is:

$$g(\text{new state}) = g(\text{current state}) + 1$$

The cost function for A* search is :

$$f(\text{new state}) = g(\text{current state}) + h(\text{new state}) + 1$$

where $h(x)$ is the hamming distance of the state x, which is the number of out-of-target boxes. Here the hamming distance is not a well-performed heuristic, because the number of boxes and targets are all small, so the difference of the heuristic values with respect to different states is not much. However, this heuristic is admissible, since if we have n boxes being out-of-target, so we need at least n moves to correct all of them. Thus, the actual cost will be always greater than or equal to the heuristic value.

2 Uniform Cost Search and A* Search

Regarding the performance of the two search algorithms, we can expect that the difference is not substantial since the heuristic is quite simple. The following tables will confirm the prediction.

Test Case	UCS			A*		
	Time (s)	Explored	Frontier	Time (s)	Explored	Frontier
1box_m3	0.009	417	20	0.009	417	20
2box_m3	1.57	26845	596	1.50	26884	596
3box_m2	75.95	185326	6909	76.53	186679	6910
4box_m1	9.28	66621	2145	8.82	66650	2068

Table 1: Performance of the algorithms on 4 test cases, including the elapsed time (Time) in seconds, number of explored nodes (Explored), and maximum size of fringe when the algorithm terminates (Frontier)

The table shows the results on 4 test cases, including 1box_m3, 2box_m3, 3box_m2, and 4box_m1. As predicted, the performance of the two algorithm is quite similar. The A* search seems to be more efficient on the less number of explored nodes, while the Uniform cost search is better on the large number of nodes. There are two reasons behind this, first, the hamming distance is not a good heuristic, and second, the largest number of nodes is on the case 3box_m2, where the boxes cannot be moved to any target, except the correct one. Thus, the counting of out-of-target boxes is wasteful and add time to the processing time.

3 Deadlocks

In Sokoban game world, there are several types of deadlock. Here in this solver, I tried to implemented the simplest deadlock, where an out-of-target box cannot be moved to any direction. For example, if there are obstacles at the (adjacent) left cell, and the up cell of the box, it will be blocked, and the state is a deadlock.

For implementation, the function obs_deadlock() checks the adjacent cells of a box, and the function deadlocks() run through the state and return True if there is a blocked box.

The following table shows the number of deadlocks in 3 out of 4 test cases above:

Test Case	UCS	A*
2box_m3	453	453
3box_m2	5580	5605
4box_m1	3573	3574

Table 2: Number of deadlocks found on 3/4 test cases

Similarly to the performance, the deadlocks found by two algorithm is almost equal. One obvious observation is the number of deadlocks is positively related to the number of explored nodes, and it truly help to improve the algorithm.