SOLVING WEIGHTED SOKOBAN BY A* GRAPH SEARCH

IFN680 Assignment 1

Zhipeng He
School of Information Systems
Queensland University of Technology
n10599070
zhipeng.he@connect.qut.edu.au

September 19, 2021

1 Introduction

Sokoban is a popular puzzle game created by Hiroyuki Imabayash in 1981. In this game, the player, as a warehouse worker, must push all the boxes into a collection of storage locations (targets) by four directions (Up, Down, Left and Right) successfully without any blocks by walls or boxes. Constantly, the solution of a warehouse in this game is not unique. Some variants of sokoban games add extra rules to limit the total costs of moving, and the player can only win when finding the shortest path. This project is based on the idea of these variants and also assigns each box with a weight, which represents the cost of pushing this box. In order to achieve the success of the sokoban, it requires to find the *optimal solution with lowest path costs*.

2 Approaches

2.1 Algorithm

From related works [1], a set of search algorithms can be applied to solve the traditional sokoban puzzle, including *deep first search, breadth first search, uniform cost search, A* search*, etc. For the variant of sokoban in this report, the primary task is to find the path of lowest costs to complete a warehouse, rather than solving a warehouse by any solutions. According to [2], in an ideal condition, A* search algorithm is a complete and optimal approach for minimizing the total solution cost. Consequently, A* search is chosen as the fundamental method for the sokoban solver.

A* graph search A* search is a type of best-first search algorithm, which guides search to the right path by calculating the sum $f(n)$ of current path cost $g(n)$ and estimated path cost $h(n)$ to the target for each node (shown in Equ. 1).

$$f(n) = g(n) + h(n) \tag{1}$$

So as to achieve the minimization of total solution cost, A* search always tries to find the node with lowest value of $g(n) + h(n)$ firstly and expend it to child nodes then. As the previous mentioned, the optimal A* search needs an ideal condition or requirement — $h(n)$ must be an **admissible** heuristic or an **consistent** heuristic. For sokoban puzzle, the optimality of the heuristic should be admissible and consistent. Generally, "the tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent" [2]. As a consequence, A* graph search are used for implementing the solver.

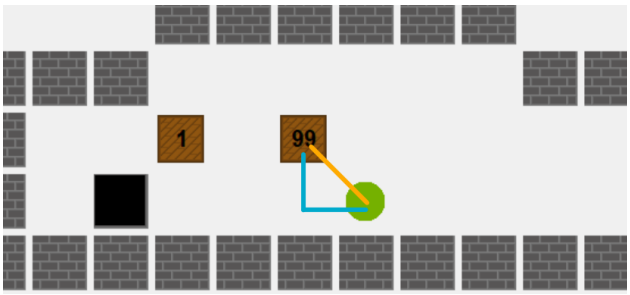

Figure 1: An example of Euclidean distance (orange) and Manhattan distance (blue).

Heuristic Considering that worker in the sokoban game can only move in four directions, some actions, like moving through diagonal, are impossible. Hence, *Euclidean distance* in Equ. 2, which calculates the diagonal distance between two point, might be not suitable for solving sokoban.

$$E_{dist} = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2} \tag{2}$$

Manhattan distance, also known as *taxicab geometry*, shows the distance from $P(p_1, p_2)$ to $Q(q_1, q_2)$ in Equ. 3, which represents the sum of the absolute differences of their Cartesian coordinates. Comparing with Euclidean distance, Manhattan distance can perfectly match the path of worker in the warehouse (shown in Fig. 1).

$$M_{dist} = |q_1 - p_1| + |q_2 - p_2| \tag{3}$$

For estimating the optimal value of $h(n)$, it is easy to calculate the Manhattan distance for only one box. However, sokoban warehouses may have more than one box for pushing, and a method for combining Manhattan distance for multiple boxes is required here. In this project, the heuristic function will apply *Simple Lower Bound* method [3] for warehouses having more than one box, which computes and returns the sum of Manhattan distance between each box to the nearest target. In addition, the sokoban warehouses in this project also assign boxes with weights, which means the heuristic function needs to take the weight differences for boxes into consideration.

Based on these requirements, if a warehouse has the number of n box(es) and target(s), my proposed heuristic function is that:

$$h(n) = \sum_{i=1}^{n} \left[(|b_{i1} - p_1| + |b_{i2} - p_2|) \times c + \min_{j \in [1,\ldots,n]} (|b_{i1} - t_{j1}| + |b_{i2} - t_{j2}|) \times (w_i + c) \right], \tag{4}$$

where p and c represent the of player's coordinate position and the path cost of each movement, $i, j \in [1, \ldots, n]$ are the index of box b_i, box weight w_i and target t_j.

2.2 State and problem instance

When implementing A* graph search algorithm, the heuristic function $h(n)$ takes node n as the input and only depends on the *state* of that node [2]. It indicates that the choice of state would take a important role in building the solver of sokoban. In this project, a sokoban warehouse is represented by a number of characters, and it is unrealistic to consider the whole warehouse as the state of node in search algorithm. Because not each part of sokoban warehouse would be modified in the progress of searching. As a result, the nodes in A* search only require dynamic features in the state. For all the static features, they should be saved in a persistent problem instance.

A sokoban is consisted of several types of cells, such as empty spaces, worker, boxes, etc. Despite the cells of free space always change when worker moves or push a box, they will not be taken into account here. Apart from the free spaces, a warehouse of sokoban can be devided into five major features, which are:

- location coordinates for all wall cells (static),
- location coordinates for all target cells (static),
- location coordinates for all box cells (dynamic),
- location coordinates for the worker (dynamic),
- value of weights for each box (static).

State The positions of the worker and all boxes will be treated as the state in each node of A* star graph search algorithm. When implementing the sokoban solver, the initial state of the warehouse `self.initial` is represented by a tuple `(self.warehouse.worker, tuple(self.warehouse.boxes))`. The follow-up states share the same structure with the initial state.

Problem instance The positions of all wall and targets and the weights of boxes will be stored in the problem instance. In the implementation, they are represented by `self.walls`, `self.goal` and `self.weights` respectively. They will not change during the game.

3 Evaluation

3.1 Tasks

Three main tasks of the assignment will be solved in this project: `taboo_cells()` function, in which finds all taboo cells (simple deadlocks) in a warehouse; `check_elem_action_seq()` function, in which checks if a given sequence of actions in the warehouse is legal or not; `solve_weighted_sokoban()` function, in which analyses the given warehouse and finds the solution with optimisation of path and cost.

Task 1: find taboo cells Besides the function `taboo_cells()`, some extra functions are also be used for search all the taboo cells in the experiment. The two rules of checking taboo cells are implemented into two inner nested functions `_rule_1()` and `_rule_2()` in `taboo_cells()`. These inner functions also call a set of auxiliary functions, such as `_move_in_2d_coordinate()`, in which calculates the final location in the 2D coordinate after a moving transformation; `_check_wall()`, in which checks if a cell having wall marks in the up, down, left and right of itself; `_check_corner()`, in which checks if a cell is a corner of the warehouse.

Task 2: validate action sequences Function `check_elem_action_seq()` is the main function for checking if a given sequence is legal or not, and function `_move_in_2d_coordinate()` is also applied for calculating the movement.

Task 3: solve weighted sokoban In order to solve the weighted sokoban, function `solve_weighted_sokoban()` will be as the interface that creates the problem instance `SokobanPuzzle(warehouse)`, calls the search algorithm function `search.astar_graph_search()` and returns the solution of sequence and cost for current warehouse.

3.2 Test Methodology

The test designs of this project is corresponding to the previous three tasks. Here is the test methodology for each task respectively in detail.

Test design for task 1 The method for testing `taboo_cells()` is based on the test function `test_taboo_cells()` in `sanity_check.py` and creates extra test cases from warehouses, which contains rule 1 taboo cells and rule 2 taboo cells.

Test design for task 2 Similar with task 1 design, the test function `test_check_elem_action_seq()` is also from `sanity_check.py`. Then, some legal and illegal action sequences are created for testing.

Test design for task 3 Function `solve_weighted_sokoban()` is the focus of overall testing. The whole test for task 3 are combined with three steps, which are:

1. A toy test by `sanity_check.py` with example warehouses on assignment description for ensuring no syntax and functional errors.
2. A benchmark on all given warehouse by function `test_solve_weighted_sokoban_all()`, which can automatically timeout the test longer than 5 minutes and record the names of unfinished warehouse and the results of finished warehouse into a csv file.

3.3 Performance

My sokoban solver successfully passes all the designed tests for task 1 and task 2. As Fig. 2a shown that, the solver correctly and quickly returns the answers as expected. The results show that the solver is sufficient for dealing with

```
<<  Testing test_taboo_cells >>
test_taboo_cells  passed!  :-)

<<  Testing test_taboo_cells >>
test_taboo_cells  passed!  :-)

<<  check_elem_action_seq, test 1>>
Test 1 passed!  :-)

<<  check_elem_action_seq, test 2>>
Test 2 passed!  :-)
```

(a) Task 1 and Task 2

```
Testing 8a
<<  test_solve_weighted_sokoban >>
 Answer as expected!  :-)

Your cost = 431, expected cost = 431
It took 1.495342 seconds

Testing 09
<<  test_solve_weighted_sokoban >>
 Answer as expected!  :-)

Your cost = 396, expected cost = 396
It took 0.008508 seconds
```

(b) Task 3

Figure 2: A screenshot for outputs by running `sanity_check.py`

tasks for finding taboo cells and identifying the valid action sequences. For function `solve_weighted_sokoban()`, my solver passed all the phase 1 testing and some results of phase 1 testing are presented in Fig. 2b. It proves that there is no simple functional or syntax errors in my codes All the tests in Fig. 2 are based on the `sanity_check.py`.

After finishing the functional testing on particular cases, I conduct a benchmark on all given warehouses to figure out the total number of cases the solver can handle with in five minutes. The whole benchmark takes five and a half hours to process all cases. As a result, the benchmark indicates that the implemented approach can solve 39 out of 108 warehouses in the limit of five minutes and, still, 69 out of all 108 cases cannot be solved in five minutes. Table 1 presents eight examples of warehouses passing the benchmark in five minutes. It suggests that the model needs more time to solver the sokoban when increasing the number of boxes and targets. Generally, my solver works efficiently in the warehouse with only two boxes, and there might be some optimisation problems for handling multiple boxes.

Table 1: Running time, number of boxes and costs for part of tests

Case	03_im	09	33	47	49	71	103	155
Box No.	2	2	3	2	3	2	4	11(1)[1]
Time	0.029	0.013	72.28	0.496	180.5	7.632	115.6	2.576
Cost	None	396	41	179	82	48	35	282

3.4 Limitations

Based on the architecture and test results, there are some limitations for the proposed solver:

- The time complexity of the proposed heuristic function is $O(n^2)$, which means this solver cannot deal with warehouse with a large amount of boxes.
- This solver can only detect some simple static deadlocks for sokoban, such as tabbo cells in this report. However, some deadlocks happen in the process of pushing box, and my solver does not have ability of find dynamic deadlocks.

4 Conclusion

Sokoban is a game in which the player tries to push all the boxes in a warehouse onto goal squares. It is hard for humans and computers alike. In this report, it applied A* star graph search algorithm with Manhattan distance as the heuristic fuction to solve a weighted variant of the traditional sokoban puzzle. From the results of the evaluation, it can solve some two-boxes or three-boxes sokoban with efficiency, but it will take much more time if the warehouse is more complex.

References

[1] Anand Venkatesan, Atishay Jain, and Rakesh Grewal. Ai in game playing: Sokoban solver, 2018. arXiv:1807.00049.

[2] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson, third edition, 2010.

[3] Timo Virkkala. Solving sokoban. Master's thesis, University of Helsinki, Helsinki, Finland, April 2011.

[1] The are 10 boxes already in targets, and only one box needs to be solved.