Queensland University of Technology

School of Information Systems

Science and Engineering Faculty

IFN 680 *Artificial Intelligence and Machine Learning*

Assignment 1 - Sokoban Planner

Name: Ian Choi, Patrick Choi

Student ID: N10421106, N10240501

Lecturer: Frederic Maire

<Table of Contents>

INTRODUCTION

This report is written to help the understanding of artificial intelligent solver for Sokoban puzzle. The solver is based on python version 3.8. The informed search method is used as approach for the solution algorithm including A* graph search and heuristics required on calculation. The solution is described in the mySokobanSolver.py file and it imports search.py and sokoban.py file. The search.py file is used mainly for Node and Problem classes required for search function. On the other hand, sokoban.py file is for Warehouse class which has the structure for the mapping elements such as positions of objects.

The solution is theoretically divided into three scenarios; elementary actions, macro actions, and weighted boxes. While first two scenarios share the same features, the last scenario has different state representations and heuristics, respectively.

SOKOBAN SOLVER

1. Elementary Actions

The elementary actions scenario is a naïve solution for the puzzle problem. The algorithm structure follows the Problem class inherited from search.py file and defined in SokobanPuzzle class. The solution of algorithm is defined on solve_sokoban_elem function which takes warehouse as parameter and returns the list of elementary actions that solves the provided puzzle. The actions are action moves of the worker to the adjacent cells.

1.1. State Representation

The state in this scenario uses initial state from the warehouse. The state representation includes only the location of the worker and the boxes for optimization purpose. The number of worker is always one. If there are multiple boxes, the boxes are represented in a tuple at the second position of the state. For example, (worker location (1, 3), boxes location ((3, 3), (5, 1)) can be an example.

1.2. Heuristic

In this scenario, the definition of heuristic being used is adding the minimum distance from the worker to the nearest box and the boxes to the nearest target cell. The heuristic is calculated to be used as input when executing informed search. Manhattan distance was used to calculate each distance between two coordinates because the warehouse is designed in grid model. The presented approach satisfies the admissibility and consistency to achieve heuristic optimization.

$$|x_1 - x_2| + |y_1 - y_2|$$

<Figure1 – Manhattan distance calculation>

The distance between worker to box is simply identifying the minimum distance from worker to boxes and add them on to total value. Since there are multiple boxes and target cells, on the other hand, the minimum distance from boxes to the targets combination needs to be identified for optimized computation. Thus, the approach to iterate all the boxes while identifying the closest target cell and append them on total value. When each box-target combination is put on the total list, they are removed from next iteration. The calculation repeats until the all box-target combination is found. Then, the total value will be used as

heuristic for elementary actions scenario.

2. Macro Actions

Likewise to elementary actions, the algorithm structure follows the Problem class inherited from search.py file and defined in SokobanPuzzle class. The solution of algorithm is defined on solve_sokoban_macro function which takes warehouse as parameter and returns the sequence of macro actions to solve the provided puzzle. The actions are action moves of the boxes to adjacent cells instead of action moves of the worker.

2.1. State Representation

Macro actions use the same state presentation of the elementary actions explained in 1.1 state representation section of this report. The state is a tuple of the locations of worker and boxes.

2.2. Heuristic

Heuristic adopted in this scenario is the same as the heuristic for elementary actions represented on 2.1 Heuristic section. The computation is to find the minimum distance between worker to boxes and boxes to targets using Manhattan distance.

3. Weighted Boxes

The third scenario assigns the pushing cost to each box while other scenarios have same pushing cost for all actions. The algorithm structure of the Problem class inherited from search.py file and defined in WeightedSokobanPuzzle class. The solution of algorithm is defined on solve_weighted _sokoban_elem function which takes warehouse and push costs as parameter and returns list of elementary actions that solve the provided puzzle. The actions are action moves of the worker to the adjacent cells.

3.1. State Representation

Weighted boxes use the similar state presentation of the elementary actions explained in 1.1 state representation section of this report. On this scenario, however, the state is a tuple of worker and ordered boxes structure instead of boxes. The order of the boxes are sorted by pushing cost in decrementing way, the box with higher cost comes first and the box with lower cost comes later.

3.2. Heuristic

The heuristic adopted in this scenario is the sum of the pushing cost from the box to the matched target plus the distance cost which is one per move. The difference in this heuristic is that it is set priority to the box with higher push cost to be moved to the closest target from the box. Total distance is the Manhattan distance of box to target and total pushing cost is distance cost multiplied by pushing cost. Therefore, adding total distance cost and total pushing cost will be total cost which will be used as heuristic. The presented approach satisfies the admissibility and consistency to achieve heuristic optimization.

4. Other Features

4.1. Taboo cells

Taboo cells refer to the cells that puzzle becomes impossible to solve when the box is moved on to them. The

solver uses taboo_cells function to define and mark the taboo cells in order to avoid the situation that worker puts the box on them. Function check_corner is used to prove that the cell is at the corner with one wall up or down side and another wall right or left side of it. On the main function, two rules are adopted in selecting taboo cells. The first rule is when the cell is located inside of the warehouse and at the corner which is not target cell. It is taboo cell because there is no function to pull the box but to push. Second rule is for the cells located between two taboo cells in the corner when they are blocked at least one side with the wall. There is no way that the box can be pulled out if the other side is blocked. The second rule is applied for both vertical and horizontal axis of the warehouse.

Moreover, the A* algorithm was applied to identify cells inside the warehouse since outside cells should be excluded from taboo cells. After re-implementing the warehouse from which the boxes were removed, all cells that workers could go to were found and designated as cells inside the warehouse.

4.2. Can go there

The function can_go_there is designed to check if the worker can go to the desired location on the warehouse without moving any box when received warehouse and destination as parameter. It simply returns 'True' if the worker can go to the destination and 'False' if not. It implements CanGoProblem class which inherits Problem class from search.py file. CanGoProblem class includes the instances from warehouse such as walls, boxes, and worker. State representation of this class is defined as the location of the worker since the location of the box cannot be changed. The heuristic uses the Manhattan distance from worker to the destination.

TESTING METHODOLOGY

In order to test the value of the solver, a tester program was developed. The testing program can set the start and end point of warehouse number to test desired amount of warehouses. From mySokoban.py file, three solvers are tested to check and compare the performances. The three solvers are solve_sokoban_elem, solve_sokoban_macro, and weighted_solve_sokoban_elem. The solvers are designed to implement the algorithms discussed above for elementary actions, macro actions, and weighted boxes respectively.

By running the program, the test for each warehouse is repeated for the set amount to calculate average time taken. Also, results of the solve_sokoban_elem function are recorded on the excel file named 'test_result.csv' with the average time taken for each warehouse. In our testing, we tested five warehouses, warehouse_21 to warehouse_29, which are randomly selected.

```
start_num = 1
end_num = 30
repeat_test = 5
result_file = "test_result.csv"

def solver_elem_test(warehouse_num):
    average_time_taken = 0
    result = None
    for test in range(repeat_test):
        warehouse_set = "./warehouses/warehouse_{:02d}.txt".format(warehouse_num)
        wh = Warehouse()
        wh.load_warehouse(warehouse_set)
        start_time = time.process_time()
        result = solve_sokoban_macro(wh)
        average_time_taken += (time.process_time() - start_time) / repeat_test
```

<Figure2 – part of python code of tester>

PERFORMANCES

The tested solver for elementary actions successfully solved the five puzzles from warehouse_21 to warehouse_29. Warehouse puzzles with number 21, 23, 27 are solved quickly in less than 0.04 second, while 25 took around 0.06 second and 29 took around 0.3 second. The gap between times taken for each warehouse is considered appropriate because the difficulty of the puzzle is in relation to the length of the result. The lengths of result for warehouse 25 and 29 were longer than those for 21, 23, and 27.

The solver for macro actions showed similar result to the elementary actions while weighted boxes had different result in terms of time taken. The time taken for warehouse 21 has increased compared to elementary and macro actions, while other warehouse numbers showed decrease overall.

Warehouse Number	Time Taken	Result							
21	0.009375	['Down'	'Left'	'Left'	'Up'	'Down'	'Left'	'Up'	'Left'
23	0.025	['Right'	'Up'	'Up'	'Left'	'Left'	'Up'	'Right'	'Down'
25	0.0625	['Up'	'Right'	'Right'	'Down'	'Left'	'Up'	'Left'	'Left'
27	0.0375	['Down'	'Left'	'Up'	'Left'	'Down'	'Down'	'Left'	'Left'
29	0.31875	['Left'	'Up'	'Left'	'Left'	'Down'	'Right'	'Up'	'Right'

<Figure3 – result of testing for elementary actions>

Warehouse Number	Time Taken	Result							
21	0.009375	[((3	3)	'Up')	((3	2)	'Up')	((2	2)
23	0.028125	[((1	3)	'Right')	((4	2)	'Up')	((3	2)
25	0.075	[((4	4)	'Left')	((2	3)	'Down')	((3	3)
27	0.0375	[((3	4)	'Up')	((3	3)	'Left')	((2	4)
29	0.340625	[((5	4)	'Right')	((5	2)	'Right')	((5	5)

<Figure4 – result of testing for macro actions>

Warehouse Number	Time Taken	Result							
21	0.01875	['Down'	'Left'	'Left'	'Up'	'Down'	'Left'	'Up'	'Left'
23	0.021875	['Right'	'Up'	'Up'	'Left'	'Left'	'Up'	'Right'	'Down'
25	0.025	['Left'	'Up'	'Up'	'Up'	'Right'	'Down'	'Down'	'Right'
27	0.021875	['Down'	'Left'	'Up'	'Left'	'Down'	'Down'	'Left'	'Left'
29	0.134375	['Left'	'Up'	'Left'	'Left'	'Down'	'Right'	'Up'	'Right'

<Figure5 – result of testing for weighted boxes>

Limitation

The limitation of the solver is that the performance can be improved by optimizing the algorithm to increase the speed of the solver. For example, the weighted boxes which used deformed state representation and heuristic for informed search displayed better performance in terms of time taken compared to the rest two scenarios. Even though current heuristic seems to have fair performance to solve the Sokoban puzzle, further research can be made to improve the performance of elementary and macro actions by improving their heuristics.