Assignment One: Sokoban Solver

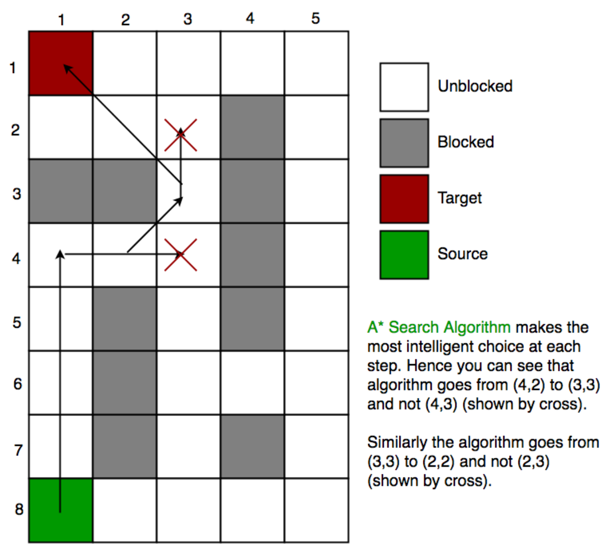
Members: Kevin Duong, n9448977; Nicholas Havilah, n10469231; Connor McHugh, 10522662

A* Graph Search

The search algorithm that was chosen to be used to implement the Sokoban solver was the A* graph search. The goal of the A* search is to find the path to the goal node with the shortest cost. The formula it uses is:

$$f(n) = g(n) + h(n)$$

Where $f(n)$ is the estimated cost of the cheapest solution through n, $g(n)$ is the cost to reach the node, and $h(n)$ is the cost to get from the node to the goal. A* search works by finding the node with the lowest value of $g(n) + h(n)$, and expanding it. As it is a graph search, it also keeps track of all nodes that have been expanded. For A* graph search to work, it requires an admissible heuristic to calculate $h(n)$. An admissible heuristic is one that never overestimates the cost to reach to goal. The heuristic utilises Manhattan distance, as outlined in the following section.



A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Figure 1: A Search Algorithm*

Heuristic

The heuristic chosen to implement the A* graph search is Manhattan distance, as it finds a distance from a given point to a target without making use of diagonal lines. This is ideal, as the player is unable to move diagonally in Sokoban.

The formula to find the Manhattan distance from some point (x_1, y_1) to some other point (x_2, y_2) is:

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

That is, the absolute value of x_1 minus x_2, added to the absolute value of y_1 minus y_2. For example, to find the Manhattan distance between the points $(1,1)$ and $(8,4)$, the equation would be:

$$\begin{aligned} |1-8| + |1-4| &= 7 + 3 \\ &= 10 \end{aligned}$$

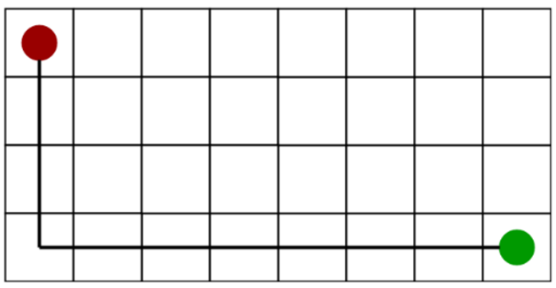


Figure 2: Manhattan Distance demonstration

The solvers implement this Manhattan distance by finding the average of the distance between the player and the closest box, and the average of distances between boxes and their closest targets, all of these distances are calculated using

Manhattan distance. This design for the heuristic was chosen because it would allow the solver to check how far from the goal state the game currently is(i.e. how far the boxes need to move on average to find a solution). It also helps the solver determine how close the player is to a box, which will help it determine the next viable move.

State Representations

Sokoban puzzles have several key features that need to be identified and monitored in order to solve the puzzle:

- the worker position,
- the box positions,
- the wall locations, and
- the target locations

Each of these elements can be categorised as being either static or dynamic. Static elements are those that at no point need to be updated in order to complete the puzzle and dynamic elements are those that do need to be updated for the puzzle to be completed.

There are two locations that the coordinates of elements can be stored: the problem instance; and the state. Initially, all elements' locations are stored in the problem instance, but the locations of the dynamic elements are moved into the state so that they can be modified.

The elements that are categorised as static are the wall locations and the target locations as their coordinates do not need to be modified. These are kept in the problem instance(walls are marked as the bricked blocks, and the targets are the black blocks in figure 3).

The elements that are categorised as dynamic are the worker position and the box positions as their coordinates need to be modified to solve the puzzle. These are moved in the state(the player is denoted as a green circle and the boxes are the brown blocks in figure 3).

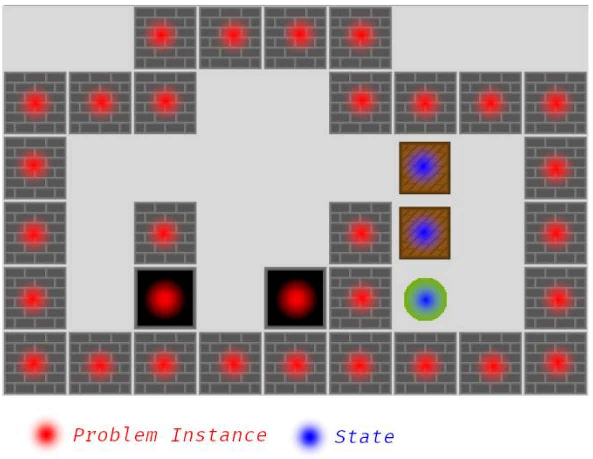


Figure 3: State representation of the elements

The program also identifies cells that would result in the game being left in a doomed state where it cannot be solved, called taboo cells. These are cells in corners that are not goals, and cells between other taboo_cells along a wall that are not targets.

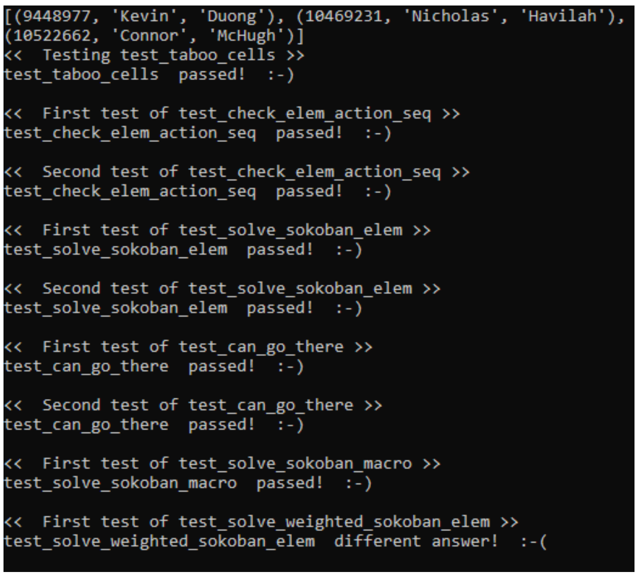
Other key features in the program that are worth noting is the Booleans that define which solvers are to be used and solver behaviour. self.macro is a Boolean used to call the macro solver, which solves the problem focusing on the moves of the boxes rather than the player(though it still has to ensure the player can push the boxes). If this Boolean is set false it calls the elementary solver, which solves the puzzle based on the moves of the player instead. self.allow_taboo_push is a Boolean that allows a move to go into a taboo cell, which was set to false to prevent boxes being pushed into taboo cells. The final Boolean labelled self.weighted allowed for the use of movement costs in the weighted solver. Unlike the other solvers, each move in the puzzle has a different cost depending on if it is pushing a box or not. This was set true only for the weighted solver.

Performance and Limitations

For the assessment, auxiliary files were given alongside the skeleton code; one of which is '*sanity_check.py*'. This script is used to perform basic tests on the solution written inside '*mySokobanSolver.py'* to ensure functionality and homogenisation within the interface. The '*sanity_check.py'* script tests several functions within *'mySokobanSolver.py*', which include:

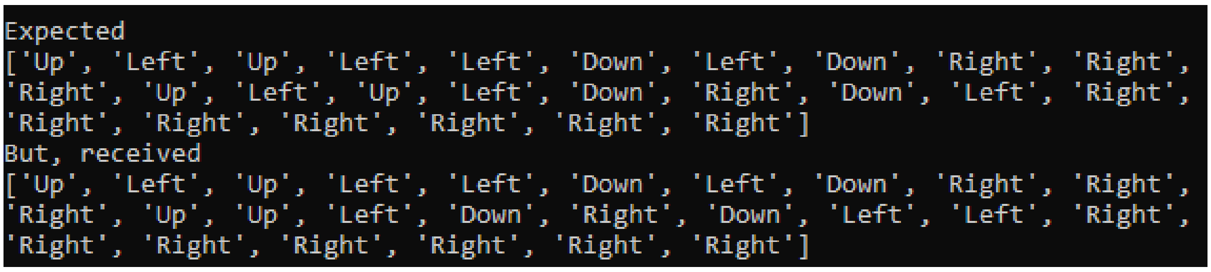
- taboo_cells,
- check_elem_action_seq,
- solve_sokoban_elem,
- can_go_there,
- solve_sokoban_macro, and
- solve_weighted_sokoban_elem

The tests are designed to check the functionality of the solution by returning an expected answer, a solution, or a Boolean value (true or false). The solution implemented passes all the tests. However, it receives a different answer for the solve_weighted_sokoban_elem function, as seen at the bottom of Figure 4. In Figure 5, due to a different solution being produced the expected route is shown first and the generated route shown second. Both routes were manually tested and although there is a small variation in the route, the route costs are the same and will both complete successfully.



```
[(9448977, 'Kevin', 'Duong'), (10469231, 'Nicholas', 'Havilah'),
(10522662, 'Connor', 'McHugh')]
<<  Testing test_taboo_cells >>
test_taboo_cells  passed!  :-)

<<  First test of test_check_elem_action_seq >>
test_check_elem_action_seq  passed!  :-)

<<  Second test of test_check_elem_action_seq >>
test_check_elem_action_seq  passed!  :-)

<<  First test of test_solve_sokoban_elem >>
test_solve_sokoban_elem  passed!  :-)

<<  Second test of test_solve_sokoban_elem >>
test_solve_sokoban_elem  passed!  :-)

<<  First test of test_can_go_there >>
test_can_go_there  passed!  :-)

<<  Second test of test_can_go_there >>
test_can_go_there  passed!  :-)

<<  First test of test_solve_sokoban_macro >>
test_solve_sokoban_macro  passed!  :-)

<<  First test of test_solve_weighted_sokoban_elem >>
test_solve_weighted_sokoban_elem  different answer!  :-(
```

Figure 4: Generated output from sanity_check.py



```
Expected
['Up', 'Left', 'Up', 'Left', 'Left', 'Down', 'Left', 'Down', 'Right', 'Right',
'Right', 'Up', 'Left', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Right',
'Right', 'Right', 'Right', 'Right', 'Right', 'Right']
But, received
['Up', 'Left', 'Up', 'Left', 'Left', 'Down', 'Left', 'Down', 'Right', 'Right',
'Right', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Left', 'Right',
'Right', 'Right', 'Right', 'Right', 'Right', 'Right']
```

Figure 5: The expected and generated solutions for test_solve_weighted_sokoban_elem

This behaviour could potentially be due to incorrectly optimised code or it has not been implemented in the same way as the expected in the tester, which could also be due to the design of the heuristic, which could be changed in future revisions to make a more viable heuristic.

With the solution being compatible with the tests within *"sanity_check.py,"* ten warehouses were chosen to further test the model solution. The warehouses are tested for their completion times for elementary and macro solvers. The number of boxes are also recorded as there seems to be correlation to the elapsed times of the tests. The results from the ten warehouses are recorded in Table 1, the Elem and Macro columns are measured in seconds. Half of the warehouses were chosen from the assessment's F.A.Q. page so they may be compared to the elapsed times there.

Warehouse	# of Boxes	Elementary Solver	Macro Solver
warehouse_03_impossible.txt	2	0.025023	0.03404
warehouse_09.txt	2	0.005005	0.006005
warehouse_47.txt	2	4.390582	1.385139
warehouse_81.txt	2	0.953696	0.62961
warehouse_11.txt	2	0.380347	0.41839
warehouse_15.txt	2	0.050041	0.063066
warehouse_45.txt	3	0.245223	0.445396
warehouse_73.txt	3	389.613149	128.600677
warehouse_69.txt	3	173.455258	27.107415
warehouse_131.txt	4	N/A	N/A

Table 1: Time elapsed and number of boxes from the ten warehouses tested

Warehouses 9, 47, 81, 11 and 3 were the ones chosen from the page with Warehouse 03 being used to test returning an impossible response. However, warehouse 9, 47, 81 and 11 under the proposed solution method, generally resulted in greater times. Warehouse 47 being an outlier in being extraordinarily greater in run time in comparison to their 0.15 seconds to solve elem. The other warehouses were chosen as they were seemingly an increase in difficulty judged on the number of boxes and goals. Although Warehouse 45 with three boxes took less than a second to solve for both elem and macro, 73 and 69 with the same number of boxes took several minutes each to complete. Finally, Warehouse 131, with four boxes took over three hours without completion and was forcefully stopped without receiving a time. More tests would be required for a trend to be identified; however, it can be observed that with an increase in the number of boxes, the difficulty will increase and will significantly increase the required time to generate a solution. Though this behaviour could also be influenced by the positions of the walls, as more moves will need to be generated to navigate around them, which would therefore increase solver runtimes. The run times in comparison to those in the F.A.Q. section may indicate optimization issues or inefficient code which will hinder the performance overall.

Conclusion

To summarise, all of the solvers implemented managed to provide solutions for most of the warehouses provided, with only some performance issues, that, given time could be identified and addressed to create a more efficient set of Sokoban puzzle solvers.