Sokoban Puzzle

CAB320 Assignment 1

Liam Percy | N9959807 | 26/4/2020

1 Introduction

Sokoban is a puzzle computer game in which a player pushes boxes around a warehouse with the goal of placing them on designated target locations. The game is played on a board of squares, where each square represents either a floor tile, box, target, or wall. The player may move horizontally or vertically on to empty floor tiles. They can move a box by walking up to it and pushing it to the square beyond. These boxes cannot be pulled, nor can they be pushed into walls or other boxes. The puzzle is solved once all boxes are pushed onto target locations.

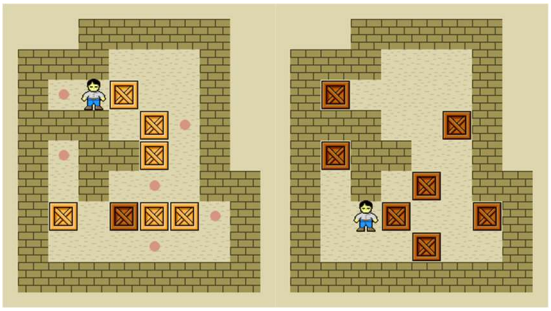


Figure 1: A Sokoban warehouse initial (RIGHT) and solved (LEFT). Target squares are denoted by red dots. Boxes on target squares are shaded.

Solving Sokoban is often compared to automated planning problems faced by autonomous robots, as the game essentially models a robot moving boxes in a warehouse. The game poses as a challenge for AI systems, not because of its branching factor however, but because of its *large search tree depth*. This essentially means that many actions (box pushes) are needed to reach a goal state.

This report aims to outline how an intelligent search agent was designed, implemented, and optimised in order to not only solve the Sokoban puzzle, but also minimise the amount of time taken to reach this solution state.

2 Code Structure

2.1 Overview

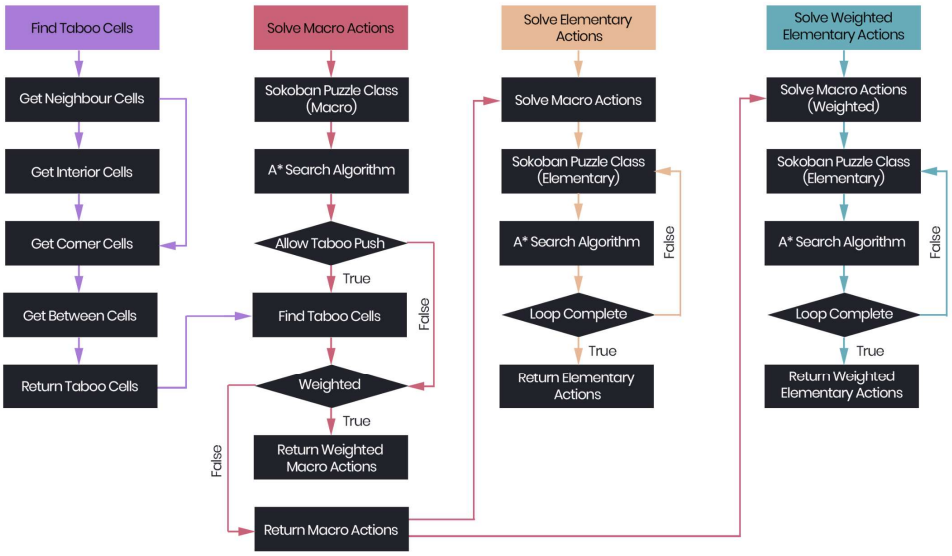


Figure 2: Flow chart of four main functions making up our solution

2.2 Sokoban Puzzle Class

The Sokoban Puzzle class is a subclass of the Problem class in search.py. It acts as the *problem instance* for each of our solver functions (section 2.3, 2.4, 2.5, 2.6) when calling a search algorithm. Upon initialisation, the class requests several arguments, each of which slightly change the way the search algorithm operates:

Argument	Function	Default
initial	Initial state of the puzzle	Required
goal	Goal state of the puzzle	None
macro	Returns macro actions if True, returns elementary actions if False	False
allow_taboo_push	Actions that result in the box being on a taboo cell are allowed if True	False
weighted	Each box will cost a certain value defined by *push_costs* if True	False
push_costs	List of values that each box will now cost to push	None

Table 1: Sokoban Puzzle class arguments

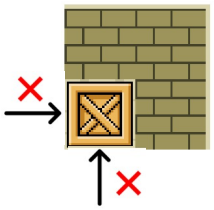
The class then defines several attributes based on these arguments, which is uses throughout the class to solve the problem.

2.3 Find Taboo Cells

A taboo cell describes any location in which, if a player were to move a box, the game would be unsolvable. There are two types of taboo cells within a Sokoban puzzle: a corner cell, and a cell along the wall between two corners that does not contain a target.

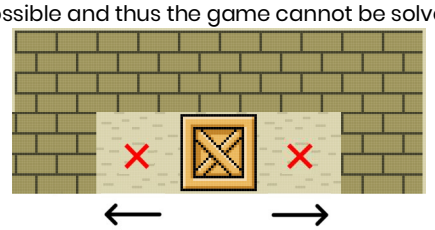
Corner Cell

If a player moves a box into a corner cell, the box can no longer be pushed in any direction except towards a wall. No further actions are possible and thus the game cannot be solved



Between Corner Cell

If a player moves a box into a cell along a wall and between two corners, it can now only be pushed into a corner cell. Once in a corner cell, no further actions are possible and thus the game cannot be solved



To find our corner cells, we first use the *get_interior_cells()* function to find all cells within the walls of the warehouse. For each interior cell, we find its neighbours using *get_neighbours()*. If there is at least one wall to the left/right of the cell, and at least one wall above/below the cell, then that cell is a corner cell. We can then check each combination of corner cells to see if they are in the same row or column. If this condition is met, we move between the two corners, ensuring that each cell in between them is along a wall and not an object. We then combine these cells into a list and return them using *get_taboo_cells()*.

Finding taboo cells allows us to optimise our searching algorithm by preventing it from exploring new branches that result in an impossible game state. This significantly reduces the computing time required to solve the puzzle

2.4 Solve Macro Actions

2.4.1 State

When solving a Sokoban puzzle, there are only two dynamic items: the player's location and the box location(s). A state in our macro actions solver will be represented using these dynamic items, and hence take the form *((player location (x, y)), (box locations (x, y)))*. All other information within our warehouse will be stored in our *problem instance* (Sokoban Puzzle object).

2.4.2 Heuristics

In order to solve the puzzle, each box must be moved a certain distance such that it reaches a target square. As the boxes can only be moved horizontal or vertically, the minimum distance between a single box and a single target is the Manhattan distance (distance measured along the axes at right angles) between these two objects. Given that a puzzle can contain multiple boxes and multiple targets, we will define our macro heuristic as the Manhattan distance from *all* boxes to their closest target

2.4.3 Process

The macro action solver creates a Sokoban Puzzle object with the *macro* argument set to True. The *__init__()* function defines the initial state as the current locations of the warehouse player and boxes, and the goal state as the warehouse target locations. The *actions()* function finds the neighbours of each box. If a player can reach a given neighbour and push the box, the action is accepted. If *allow_taboo_push* is set to False, the function rejects boxes pushed into taboo cells. The *result()* function updates the current player and box locations based on the above action, and then the *goal_test()* function checks to see if the box locations match the goal state.

2.5 Solve Elementary Actions

2.5.1 State

The elementary action solver first solves the problem on a macro level, and then for each macro action, returns the elementary actions taken to move between these macro actions. For this reason, the only dynamic item in the elementary action solver is the player, and hence our state takes the form (player location (x, y), direction).

2.5.2 Heuristics

The minimum distance between a player and their goal location is the Manhattan distance between these two points. In this scenario, because we only ever consider one player location and one goal location, we can simply use this distance as our heuristic.

2.5.3 Process

As explained above, the elementary action solver first finds all macro actions required to solve the puzzle. For each macro action, the solver creates a Sokoban Puzzle object, defining the initial state of the player and their goal. The goal is defined as the location of the player before completing the next macro action, and is found using the *move_player()* function. The solver uses an A* searching algorithm to find the elementary actions between these two points, and then updates the state of the warehouse. These steps are completed for all macro actions until all elementary actions are found.

2.6 Solve Weighted Elementary Actions

2.6.1 Process

The weighted elementary actions solver has the same state representation, the same heuristic and works the same way as the standard elementary actions solver, however each box push now incurs a specific cost. These costs are defined when we run the macro actions solver, by setting *weighted* to True and defining a *push_cost* argument. This argument is a list where the [i]th element in the list corresponds to the push cost of the [i]th box in the warehouse (at position warehouse.boxes[i]). By setting the *weighted* function, the macro solver will now use the calculate the cost of moving the player to a box, and add this value to the boxes push cost.

3 Performance

3.1 Overview

Warehouse	Find Taboo Cells (ms)	Solve Macro Actions (s)	Solve Elementary Actions (s)	Solve Weighted Elementary Actions (s)
Warehouse 07	1	44.13	44.14	78.42-212.56
Warehouse 09	0.5	0.03	0.04	0.05
Warehouse 11	1	0.64	0.67	0.98
Warehouse 47	1	0.68	0.73	1.62
Warehouse 81	1	0.79	0.81	2.54
CAB320 Warehouse 5	1	6.68	6.69	10.7

Table 2: Solver function performance

The key metric chosen to rate the performance of our intelligent agent is the computing time required to solve the puzzle. As we can observe, regardless of our warehouse, the agent finds taboo cells incredibly efficiently. By using our macro actions to find elementary actions, our elementary solver takes mere milliseconds longer than our macro solver. This should not be considered a limitation either, as there is not way to solve the game without completing macro actions. It is difficult to judge the performance of the weighted elementary action solver, as it

is largely determined by the push costs of each box, however it can still be said that this solver maintains a high level of performance after testing.

3.2 Optimisation

There are several design decisions that serve to reduce the solution's time cost of execution and therefore increase its performance.

3.2.1 Graph Search vs A*

The *can_go_there()* function is used each time we look for new actions we can complete for a given state. In order to reduce the computational time of this function, a typical graph searching algorithm was used to implement this function, rather than an A* search algorithm, which removes the need to calculate heuristics.

Warehouse	Graph Search Time (s)	A* Search Time (s)
Warehouse 07	44.14	96.95
Warehouse 09	0.04	0.23
Warehouse 11	0.67	3.74
Warehouse 47	0.73	4.16
Warehouse 81	0.81	5.41
CAB320 Warehouse 5	6.69	40.59

Table 3: Solving elementary actions using graph search vs A search for can_go_there() function*

While a graph search algorithm may not return the optimal path, we can clearly observe the improvements in performance when compared to the A* searching algorithm

3.2.2 Assignment vs Append

Another method used to increase the performance of our solver is the utilisation of assignment instead of Pythons *.append()* function. Append uses temporary memory allocation to store the appended variable and then add it to the end of a list. This is slightly slower than assigning a value directly to a list index. This technique was used in the result function in order to improve the performance of the agent, especially for warehouses with deeper search trees.

Warehouse	Assignment Solve Time (s)	Append Solve Time (s)
Warehouse 07	44.14	215.41
Warehouse 81	0.81	1.09
CAB320 Warehouse 5	6.69	13.64

Table 4: Solving elementary actions using assignment vs append

4 Limitations

4.1 Macro Representation

Elements within a Sokoban warehouse are expressed as coordinates in the (column, row). The macro action solver returns actions in the form ((row, column), direction). In order to complete certain operations, we often must re-arrange tuples. This significantly reduces the performance of the solution.

4.2 Defensive Programming

Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances, such as incorrect or missing arguments. Some functions within our solver lack this. As an example, while our weighted solver does ensure a push cost exists for each box, there is no statement preventing too many push costs from being entered, and therefore our solution will not function if the user inputs more push costs for boxes, than there are boxes themselves.

Conclusion

Over the course of developing this intelligent agent, several careful design decisions have been made to optimise our solution and ensure high performance when solving the search problem. The solution is cable of returning multiple different sets of actions based on user input, and can do so in an incredibly timely fashion, without branching to impossible game states. With that said, there are still some limitations present in the design, that if removed may serve to both increase the performance and reliability of the solution.