Sokoban Assignment

John Pase, n9998616
Lasni Hakmanage, n10182403
Phuong Nam Ly, n10098097

Queensland University of Technology

Introduction

Sokoban is a Japanese computer game created in 1982 for the Commodore 64 and IBM-PC. It's a puzzle game that requires the players to maneuver crates from their starting positions to their proper storage locations. A square grid is used where each of those squares can be either a floor part or a wall. Floor squares may also take the form of target squares. The player has the ability to push the boxes and move the crates to reach the goal square. However, the player can only move up, down, left and right, and also can only push one box at a single time.

This report aims to discuss the state representations, heuristics, performance and limitations of the Sokoban solver.

Notations

The puzzles and their initial state are coded as follows,

- **space**, a free square
- '#', a wall square
- '**$**', a box
- '**.**', a target square
- '@', the player
- '**!**', the player on a target square
- '*****', a box on a target square

Class

SokobanPuzzle()
An instance of this class represents a Sokoban puzzle. The constructor contains attributes like the walls, the targets, the boxes and the worker. The method "actions" returns the list of actions that can be executed in the current state. The method "results" returns the resultant state after executing an action in the given state.

Functions

Taboo_cells()
This function returns the taboo cells within a warehouse. A “taboo” cell is a cell inside a warehouse and makes the puzzle unsolvable if a box is pushed on the cell. There are two criteria when determining taboo cells. Firstly, if a cell is a corner and not a target, it is taboo. Secondly, if none of the cells along a wall, between two corners is a target, all of them are taboo.

Check_elem_action_seq()
This function determines whether a sequence of actions is legal.
The function gets the resultant state after executing actions and checks this result, whether a worker has gone into a wall, a box has gone into a wall or two boxes are pushed at the same time. If there is any collision, the string “Impossible” will be returned and the function will end. Otherwise, the function returns the state of the puzzle after applying the sequence of action.

Solve_sokoban_elem()
This function solves the puzzle using A* graph search algorithm. In this scenario, all actions have the same cost. An elementary action is an action that moves the worker to an adjacent cell.
Within the search algorithm, the position of the worker, boxes and targets can be considered as nodes. The factors that influence the movement of the worker depends on the list of valid moves, and the minimum distance between the box to the target and the minimum distance between the worker to box. The worker moves until it reaches the goal. The function then returns the actions that led to the node, or “Impossible” if the search fails.

can_go_there()
This function determines whether the worker can walk to the cell without pushing any box. The function uses A* graph search to find the optimal path from the current cell to the box, if there is one.
Within the search algorithm, the factors that choose the optimal path include choosing the minimal distance between the worker and the target.

solve_sokoban_macro()

This function determines the macro actions needed to push a specific box to a targeted cell. The implementations to a macro-based Sokoban solver are as follows.

1. Is the worker next to the box?
2. Can the box be pushed?
3. If the box is pushed, does it cause deadlock?

Deadlock state means when the boxes are pushed into a place where a solution to the puzzle is impossible. For example, if we push a box into a gap between two boxes against a wall, then it can cause all three boxes to be locked.

The self.allow_taboo and self.macro are to consider the above rules and help return a list of actions that would solve the puzzle. Following this, a heuristic used in the A* search is used to get the distance from boxes to the targeted cells, and from worker to the boxes. Finally, these macro actions lead to a sequence of elementary moves by the worker. The result of this heuristic search helps to complete the solve_sokoban_macro function.

solve_weighted_sokoban_elem()
This function solves the puzzle using A* graph search algorithm, with each box assigned a respective weight. In this scenario, the actions might have different costs.
In the last two scenarios, the actions have the same cost. By default, the path cost increases by one per step. In this scenario, when the weights of the boxes are considered, the path cost increases by the weight of a respective box, if a box is pushed.

Heuristics

As mentioned previously, the search algorithm used in the code is A* heuristic search. This is based on the breadth-first tree search with the cost of path_cost and the heuristic function. For this application, ‘’manhattan distance = dist” is used between boxes and targets. By using this, we calculated the minimum distance between a box and the closest target.
Dist = abs(box(x) – target(x)) + abs(box(y) – target(y))
This helps find which objects are closer to the goals and gives the shortest path for the solution.
The A* algorithm shows that it can determine low cost pathway. Unlike depth-first search, nodes are explored from left to right and the heuristics detects which nodes in the frontier are expanded first. Once it expands, the result is passed through and it updates the worker with the positions of the boxes it has pushed, and also recalculate the current available actions to form the frontier and continue the search. When the solution is found, the result is return with actions (series of nodes).
It is important to note that this solver doesn’t consider about the locations of the wall and other obstacles when calculating the heuristics distance.

Performance and Limitations

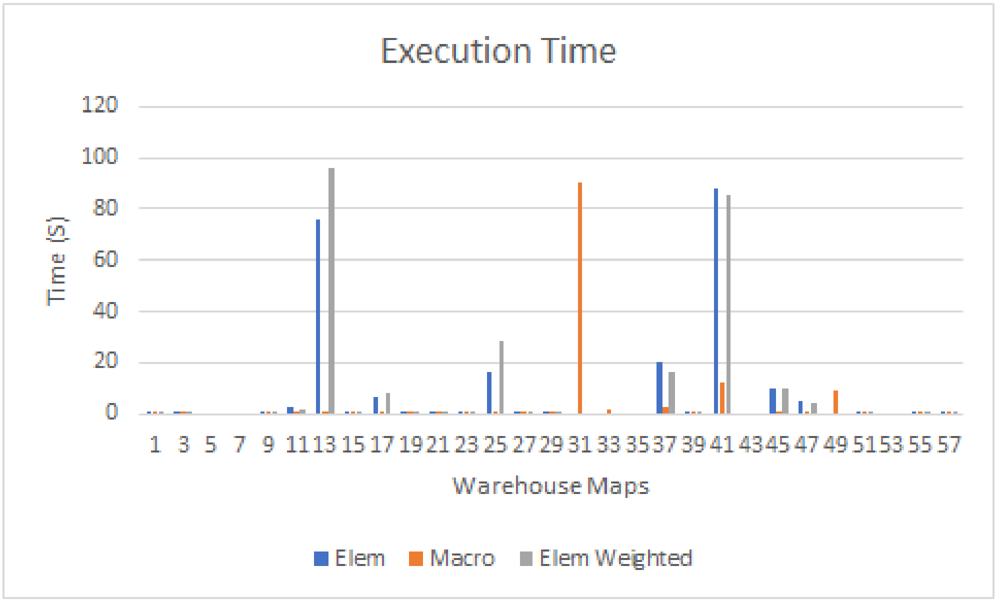

Figure 1. The bar graph comparing the execution time of the three methods over a range of warehouse maps (with allow_taboo = true)

Regarding performance, the solver performs well because on average, the puzzle is solved within seconds. There are some cases that took the algorithm longer to solve or too long to record data: warehouse 5, 7, 35, 43, 53. In addition, warehouse 31, 33 and 49 took too long for elem and elem_weighted methods to be completed. Macro methods take less time to finish compared to the other two methods.

Regarding limitations, when there are three or more boxes, the computation time increases and the algorithm might take too long to complete. For example, warehouse 5, 7 and 31 all contain three or more boxes and that increases the computational time significantly. Changes that can be made to the algorithm includes using list comprehension for heuristics.

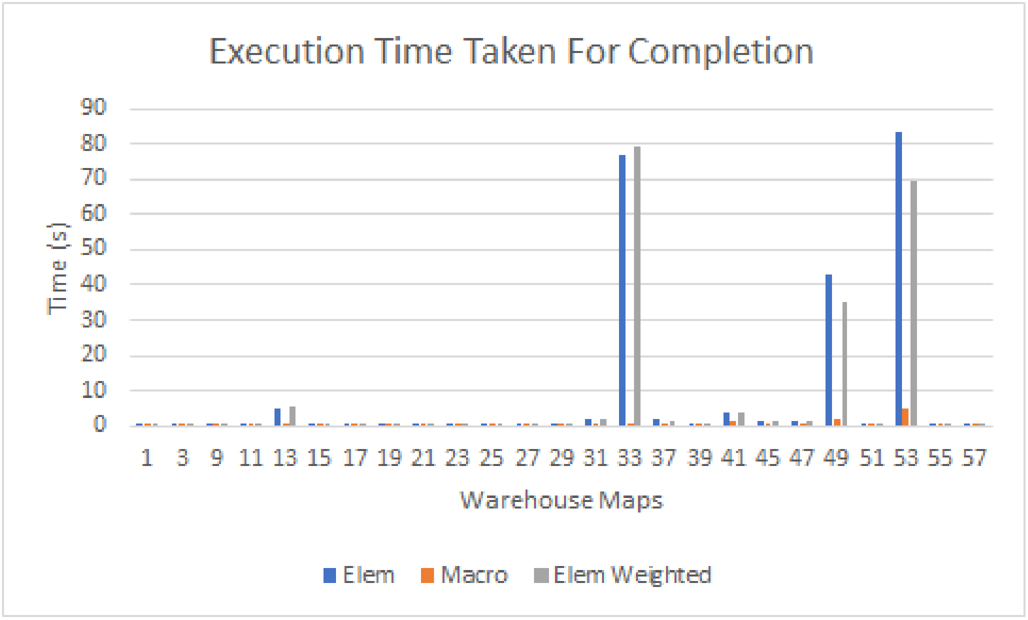

Figure 2. The bar graph comparing the execution time of the three methods over a range of warehouse maps (with allow_taboo = false)

With allow_taboo = false, meaning that moving a box onto a taboo cell is not allowed, performance drastically improves compared to allow_taboo = true. This limits the number of available spaces to move therefore limiting the number of moves that the search algorithm checks, increasing computational speed. Macro methods still take less time to finish compared to the other two methods. There are still cases that take a long time to solve like warehouse 33 and 53 for elem and elem weight methods.

Conclusion

The Sokoban solver greatly utilises A* search algorithm in the three methods used. The solver performs well on average, especially when allow_taboo = false, with a couple of exceptions. When there are three boxes or more, the computation time is increased dramatically. Solutions to this problem includes using list comprehension for heuristics.