Group 17 Statement of Contribution

Introduction

Sokoban is a puzzle video game in which a player pushes boxes around a warehouse to a designated destination. Each warehouse comes in different shapes and sizes, with numerous boxes and target location.

Challenges

The gameplay is simple but often poses challenges, as there is a failure condition when a box is pushed into a location that cannot be maneuvered out (apart from target location), rendering the warehouse unsolvable and force player to restart. Therefore, different warehouses require different strategies and planning for workers to push all the boxes into target locations.

Solution

Often, there are several warehouses that are too difficult for human to solve or may take long time to come up with solution, so a proposed solution could be AI auto solver.

The backbone for this solver is pathfinding algorithms, to find ways to bring the boxes to their intended location. We also approach this problem 2 different ways:

1. Elementary Solver – By focusing on the actions of the worker, we explore the steps that the workers can take.
2. Macro Solver – Focus on the movement of the boxes instead, we explore how the boxes can move, of course considering the worker's position.

This report details our experiment with the solver using different algorithms including breath-first search, A* and depth-first search. We will evaluate and choose a final algorithm based on time-efficiency and move-efficiency (number of moves to reach goal state).

Taboo Cell

To optimize our solver, we implemented a taboo_cell function which acts as a heuristic guide for the AI's action. This function defines any cell and zones as taboo where if a box is pushed in then the puzzle becomes unsolvable, essentially the solver shouldn't push a box in there at all costs. The main rules to identify a taboo cell are:

1. If a cell is a corner inside the warehouse and not a target, then it is a taboo cell.
2. All the cells between two corners inside the warehouse along a wall are taboo if none of these cells is a target.

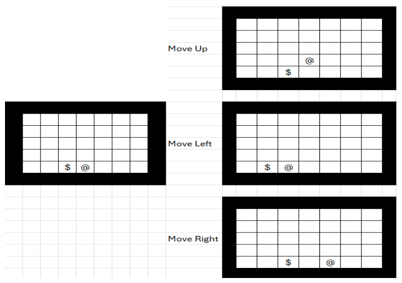
By identifying a taboo zone, it eliminates wasteful moves and optimizes efficiency.

Breath-First Search (BFS)

BFS is an algorithm that starts at a given root node, then explores all its nodes at current depth level, then keeps exploring all the adjacent nodes at next depth level, repeating until found goal state or out of node. In simple words, it aims to explore all possible routes and is guaranteed to identify the shortest route, however, comes at the cost of memory and time efficiency.

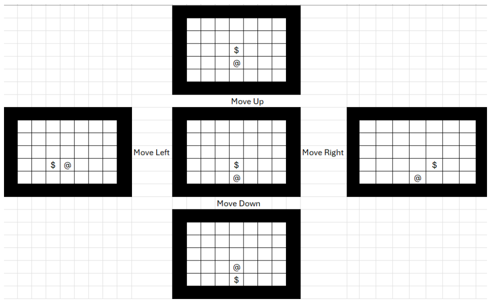
Implementation

Applying BFS to our Sokoban is simple.



For Elementary Solver, we will explore all the directions that a worker can move to at a given position, keep track of the possible positions and continue explore the next possible position. Keep in mind the worker will consider the location of walls and boxes into their movement. An illustration demonstrates idea as follows: with @ as worker and $ as box

The algorithm will repeatedly consider the next possible state, until it finds a state where the location of the boxes matches the target location.



For Macro Solver, the algorithm will explore the direction the boxes can move instead, the following picture demonstrates, where the box can be pushed up, down, left, right and the worker will have to move into the position that can push the box.

Efficiency

Timed Out (Elem): 42/101 warehouses

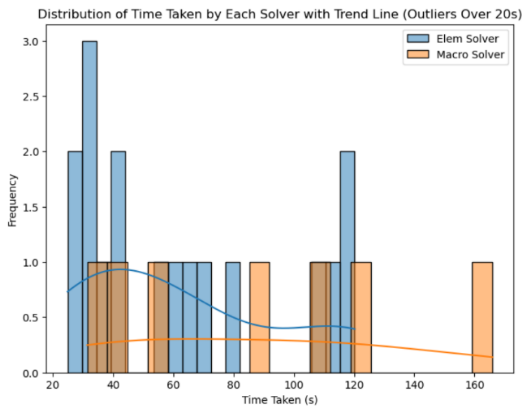
Timed Out (Macro): 35/101 warehouses

Average Time Taken (Elem): 19.615s

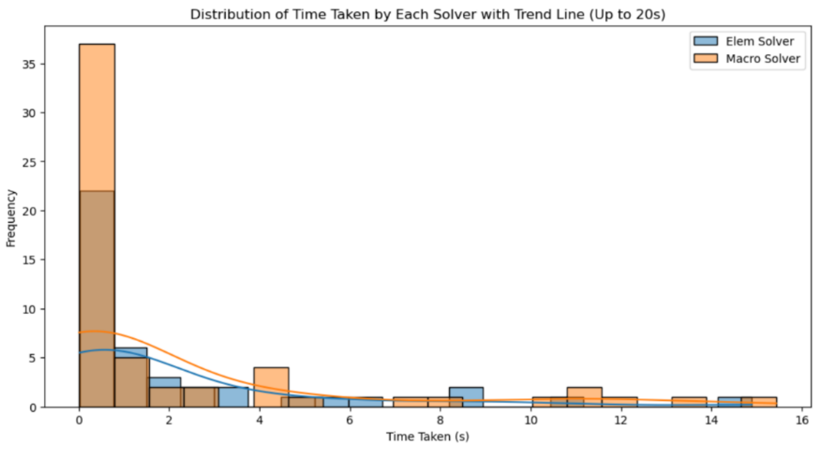
Average Time Taken (Macro): 11.307s

Average Solution Length (Elem): 76.814

Average Solution Length (Macro): 27.091



We excluded any warehouse that got timed out from calculation and visualization. Based on the result, we can see that Macro Solver performance is significantly better than using Elem Solver, with average time to solve a warehouse is 11.3s, while Elem is 19.6s. Macro also has 35 timeout warehouse while Elem is 42. Solution length wise, since Macro only considers movement of box, its hard to compare against Elem solver who considers worker's movement. The histograms also show that Macro Solver managed to solve more warehouses under 1 second than Elem Solver, with Elem Solver struggled and generally takes longer time to solve warehouses as their complexity increases as seen in histogram for warehouse solution over 20s. Overall in BFS, using Macro solver is the more optimized way.

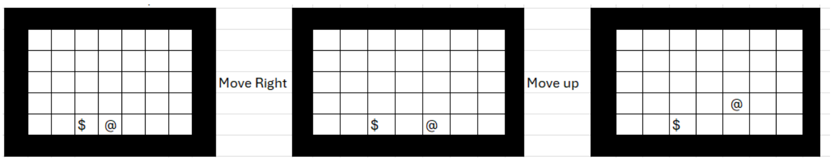


Depth First Search (DFS)

DFS also starts at given root node and will explore the next possible node as far as possible along each branch before backtracking. Depending on the scenario, it may find the goal state at earliest opportunity, quicker than BFS, but does not guarantee an optimal path the same way BFS does.

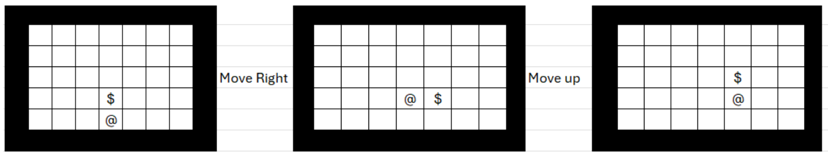
Implementation

Similarly to BFS implementation, the algorithm will explore a possible move and the move right next to it iteratively.



For Elementary Solver, it will be similar to the right picture, moving the worker (@) in a series of actions.

For Macro Solver, which focuses on moving the box ($) only, will be as follows:



Of course, these will move until they reach a goal state, or the box gets stuck in taboo cell.

Efficiency

Timed Out (Elem): 35/101 warehouses

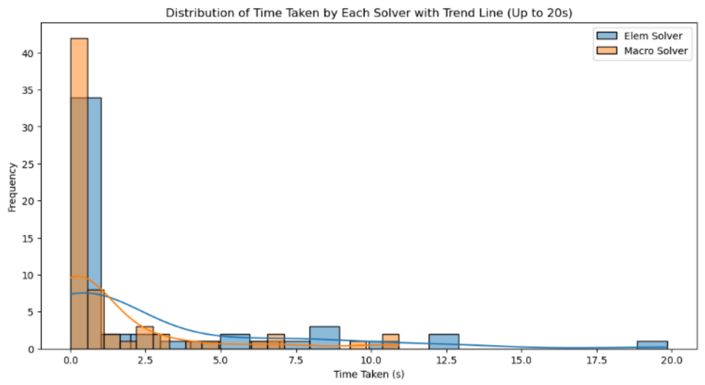
Timed Out (Macro): 29/101 warehouses

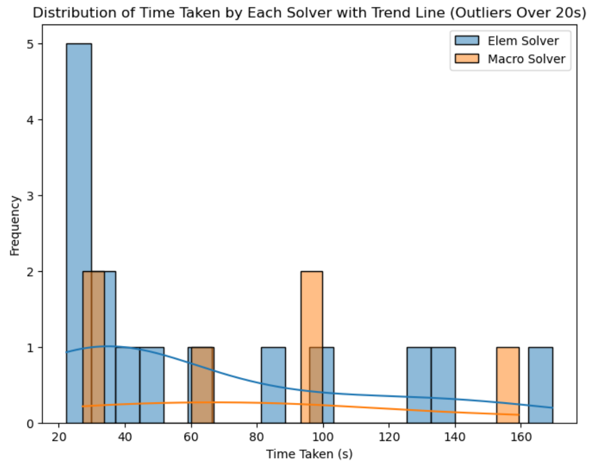
Average Time Taken (Elem): 16.453 seconds

Average Time Taken (Macro): 7.732 seconds

Average Solution Length (Elem): 704.939

Average Solution Length (Macro): 159.375





Using DFS, only 35 and 29 warehouses were timed out respectively by Elem and Macro solver. The average time is also lower than DFS, with Elem achieving 16.4 and Macro achieving 7.7s, both were able to reduce 3-4s. Macro implementation is still the more optimized method according to the histogram, showing Macro solver time concentrated at lower time than Elem solver. However, it is important to note that the solution length (e.g number of action) is significantly higher than DFS. The average Elementary move is a staggering 704, 10 times the length in BFS, while Macro is 159, around 5 times more. This introduces a huge trade-off between time-solution efficiency, optimizing time solve but significantly degrading solution length.

A Star (A*)

A* is a pathfinding and graph traversal algorithm, combining aspects of BFS and DFS using heuristic to guide its search to goal state. A* evaluates each node's cost and uses heuristic to estimate path to the goal. This way it guarantees to find shortest path and if heuristic is admissible. It evaluates each node by calculating the cost using f=g+h:

- g represents path cost from starting node to current node
- h is heuristic estimate from current node to the goal node.

Heuristic

For this puzzle, we tried Manhattan and Euclidean distance based on the following considerations.

Manhattan Distance
It is a metric used to calculate distance between two points in a grid-like path, where it measures the sum of absolute difference between coordinates of points. This is suitable for our Sokoban where the space is based on graph.

Value Method
This calculates the distance between each box to its nearest target using a greedy approach. It iteratively finds the shortest distance between a box and target and pairs them until all boxes are matched. We used Euclidean distance rather than Manhattan as it provides precise estimate of cost. We tried to include worker's distance to nearest box to heuristic encourage worker stay close, but after testing we found this made the solution slower and less optimal.

Path Cost & Taboo Cell
We also considered the number of moves workers made to reach a particular state, and whether it can reach that state. Additionally, by incorporating Taboo Cell, we prune the algorithm from considering those spaces and reduce wasteful movements that would have rendered the puzzle unsolvable.

Result - Efficiency
Overall, we observed that implementing A* with Manhattan distance yields identical solutions length to BFS, however the time taken to solve each warehouse is longer, and in many cases A* even result in timeout for warehouse that otherwise would be solved in time by BFS and DFS. This is an indication that our heuristic is weak and suboptimal for Sokoban Puzzle. While in theory this is a more optimized option, when applied to our work it seems to underperform due to heuristic. Not a good solution-efficiency wise, and due to space limitation and overlapping information with BFS, we couldn't put a graph here.

Recap & Limitation
To recap, we experimented with BFS, DFS and A* (Manhattan) using Macro and Elem Solver. The result shows that BFS provided the most optimal solution in terms of moves, but at slightly longer solve time and higher memory. DFS on the other hand, reduced time solved by exploring different path rapidly, leading to lower average time to solve and managed to solve more warehouse within 180 seconds than BFS, however significantly increase the result sequences of move by average up to 10 times than BFS. A*'s was supposed to consider both path and time efficiency, however, was not fully optimized due to weak heuristic (Manhattan) distance and ended up with backfiring with identical path-efficiency as BFS, but significantly longer solve time and usually ended with timed out.

Solution & Future Recommendations
Our overall recommended solution would be to use **BFS with Macro Solver Approach**, as it provides optimal pathfinding with adequate solve time. While DFS yields better solve time, its poor solution sequence is not ideal for Sokoban and many real-world applications. For future improvements, using A* with a Sokoban-specialized heuristic would be ideal, as it would specifically aim to solve the problem within Sokoban puzzle complex and could give better efficiency, whereas Manhattan does not consider all aspect in Sokoban, and BFS which is a universally used algorithm rather than made for Sokoban.

Contributions of Phi Long Nguyen:

- I was responsible for preliminary code inspection and familiarized in various files such as search.py, sokobanTester.py and mySokobanSolver.py. Identified important functions that needed to be implemented such as SokobanPuzzle, and several other helpers function necessary to ease workflow and debug. Laid down some requirements for our code to match the instructions. Adjust code in sokobanTester for more debug information and record data for visualization
- I completed this sections of the code:
    - Section: Define taboo_cells, half SokobanPuzzle for macro implementation, full solve_sokoban_macro, and can_go_there
    - Function includes:
        - findTaboo()
        - taboo_cells()
        - flip_coordinates()
        - can_go_there()
        - sokobanMacroProblem()
        - solve_sokoban_macro()
- I contributed to the report by
    - From start to DFS

Contributions of Aswin Jayaraman:

• In our project, I was responsible for implementing several core components of the Sokoban puzzle solver, including developing functions for ELEM solver for game, heuristic optimization, and specific pathfinding and deadlock detection mechanisms. My contributions include:

I completed these sections of the code:

•Section: half SokobanPuzzle for elem implementation, full check_action_seq and solve_sokoban_elem , sokobanElemProb

Function includes:

- manhattan_distance()
- h()
- is_deadlocked(()
- box_in_corner()
- Box_stuck_along_wall()
- get_movement()

I contributed to the report by

- Writing A* and conclusion – purposed solution and limitation

Team Signatures (Digital)

Phi Long Nguyen

26/10/2024

Aswin Jayaraman

26/10/2024