IFN680 Project 2 Report

Introduction

Sokoban, a classic computer puzzle game, tasks players with navigating a robot through a maze to push boxes into designated locations. Originally published in 1982 for platforms like the Commodore 64 and IBM-PC, Sokoban has since been adapted to numerous other platforms and video game consoles.

On the surface, it might seem like a simple game. However, the challenges posed by Sokoban are many, making it an interesting subject for artificial intelligence and automated planning problem enthusiasts.

While Sokoban remains a beloved game, its real-world implications extend beyond mere entertainment. The game effectively models a robot's task of moving boxes within a warehouse, a scenario that modern industries often encounter. As automation becomes increasingly integral to supply chains and logistics, the ability to efficiently plan and navigate space becomes crucial. In this context, Sokoban is not just a game but a representation of real-world automation and space optimization challenges.

One of the primary challenges in Sokoban is the identification of "taboo cells"—cells that, if occupied by a box, would render the puzzle unsolvable. Correctly identifying these cells is paramount, as it can prevent the player (or automated solver) from making irreversible mistakes. This report will delve into the methodology employed to detect these taboo cells, ensuring a more informed and strategic approach to solving the Sokoban puzzle.

Methodology

‘taboo_cells’ function

The **taboo_cells** function is designed to identify "taboo cells" in a Sokoban warehouse layout. These are specific positions where, if a box is placed, the puzzle becomes unsolvable. The identification is based on two primary criteria:

Corner Taboo Cells:
- If a cell is a corner inside the warehouse and isn't a designated target, it's marked as taboo.
- The function checks for four types of corners (top-left, top-right, bottom-left, bottom-right) and labels the qualifying cells.
-

Cells Between Two Corners:
- The function identifies cells that lie between two corner cells along a wall. If none of these intermediary cells are targets, they're marked as taboo.

After identifying these cells, the function returns a representation of the warehouse where taboo cells are marked with an 'X' and walls are marked with '#'. All other markings, like the worker, boxes, and targets, are omitted from this representation.

This function was tested and the results were successful.

Sokoban Puzzle Solver Methods

actions(self, state)

- Determines valid actions that the worker can take from a given state.
- Considers the following actions: 'Left', 'Down', 'Right', 'Up'.
- Checks if the worker is blocked by walls or boxes. If there's a box in the action's direction, it ensures the box can be pushed without hitting another box, wall, or taboo cell.

result(self, state, action)

- Applies a given action to a state and returns the resulting state.
- Updates the position of the worker and, if a box is in the path, the position of the box.
- Ensures that only valid actions, as determined by the **actions** method, are applied.

goal_test(self, state)

- Checks if all boxes are on their target positions.
- Returns **True** if the state meets the goal criteria, otherwise returns **False**.

h(self, node)

- A heuristic function that estimates the cost to reach the goal from a given state.
- Calculates the sum of the Manhattan distances from each box to its closest target position.
- Used to guide search algorithms to find solutions more efficiently.

check_action_seq(warehouse) function

The function determines whether a sequence of actions can be applied to a Sokoban warehouse without violating any rules (e.g., pushing a box into a wall, moving a box into a taboo cell, etc.).

Steps:
Initialisation:
- Convert the input warehouse into a SokobanPuzzle instance.
- Extract the worker's position, box positions, and taboo cells from the puzzle instance.

Action Sequence Validation:
- For each action in the provided sequence:

- Check if the action is valid for the current state using the **actions** method of the SokobanPuzzle instance.
- If the action is not valid, return "Failure".
- If valid, apply the action to the current state to get the new state using the **result** method.
- Check if any box has been moved into a taboo cell. If so, return “Failure".

Return Final State:
  - If all actions in the sequence are valid and no boxes are in taboo cells, convert the final state back into a warehouse representation and return it.

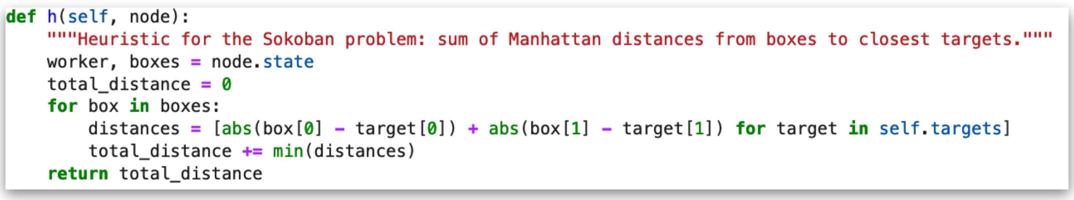
Solver Algorithm and Use of Heuristics.

Manhattan_distance(point1, point2)

Calculates and returns the Manhattan distance between two given points. Manhattan distance is the distance between two points in a grid-based path (like Manhattan street grids) and is computed as the sum of the absolute differences of their coordinates.

Sokoban_heuristic(node, problem)

The Sokoban puzzle is solved using a heuristic function that is specially designed for it. This function calculates for each box in the state, the minimum Manhattan distance to any target in the warehouse. The heuristic's value is obtained by summing up these minimum distances for all boxes. Essentially, this heuristic estimates the cost required to move each box to its nearest target.



```
def h(self, node):
    """Heuristic for the Sokoban problem: sum of Manhattan distances from boxes to closest targets."""
    worker, boxes = node.state
    total_distance = 0
    for box in boxes:
        distances = [abs(box[0] - target[0]) + abs(box[1] - target[1]) for target in self.targets]
        total_distance += min(distances)
    return total_distance
```

Solve_sokoban_elem (warehouse) function

To solve the Sokoban puzzle, the program takes the following steps:

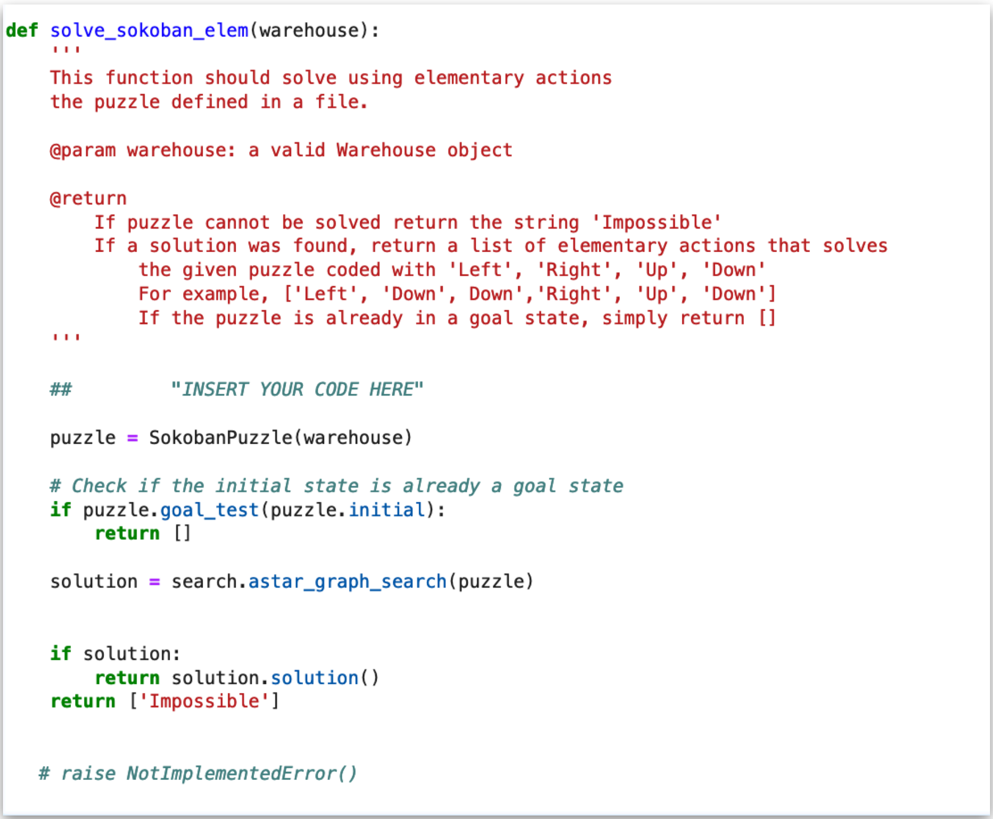
1. It converts the input warehouse into a SokobanPuzzle instance.
2. It checks if the initial state of the puzzle is already a goal state (i.e., all boxes are on targets). If yes, it returns an empty list.
3. If not, the program uses the A* search algorithm to find a solution to the puzzle.
4. The heuristic for the A* search is defined in sokoban_heuristic.
5. If a solution is found, the program returns a list of actions that solve the puzzle.
6. If no solution is found, it returns the string 'Impossible'.

By following these steps, the program can solve the Sokoban puzzle using elementary actions (i.e., basic moves without macros or compound actions).

These functions provide tools to estimate the cost of reaching the goal state in the

Sokoban puzzle and to find a solution using elementary actions with the help of the A* search algorithm.

The tests were successful for this method.



```
def solve_sokoban_elem(warehouse):
    '''
    This function should solve using elementary actions
    the puzzle defined in a file.

    @param warehouse: a valid Warehouse object

    @return
        If puzzle cannot be solved return the string 'Impossible'
        If a solution was found, return a list of elementary actions that solves
            the given puzzle coded with 'Left', 'Right', 'Up', 'Down'
            For example, ['Left', 'Down', Down','Right', 'Up', 'Down']
            If the puzzle is already in a goal state, simply return []
    '''

    ##         "INSERT YOUR CODE HERE"

    puzzle = SokobanPuzzle(warehouse)

    # Check if the initial state is already a goal state
    if puzzle.goal_test(puzzle.initial):
        return []

    solution = search.astar_graph_search(puzzle)


    if solution:
        return solution.solution()
    return ['Impossible']


   # raise NotImplementedError()
```

Comparison of Search Algorithms for Sokoban Puzzle Solving

Breadth-First Search (BFS)

Breadth-First Search, commonly referred to as BFS, is a fundamental graph traversal method. Its systematic approach ensures every node at a particular depth is explored before delving deeper into subsequent levels.

Application to Sokoban:

In the context of the Sokoban puzzle, BFS systematically explores all feasible move sequences. The primary strength of BFS lies in its ability to guarantee the shortest path in terms of the sheer number of actions executed. However, BFS doesn't take into account the specific costs associated with individual actions. This oversight can render BFS computationally intensive, especially when solutions necessitate a multitude of steps.

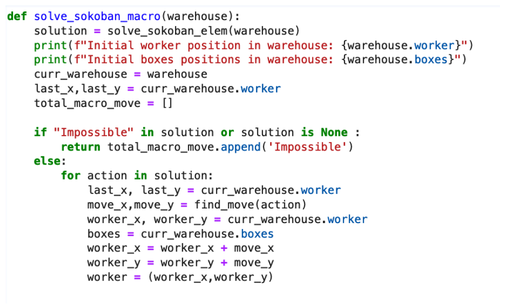
A* Search

A* search is an informed strategy, distinguished by its dual reliance on the actual cost to access a node and a heuristic estimate of the remaining distance to the objective.

Application to Sokoban:

For the Sokoban challenge, A* search is instrumental in narrowing down potential solution paths. By estimating the "distance" required to align boxes with their respective target positions, A* search can traverse the puzzle landscape more efficiently than BFS. The algorithm's effectiveness hinges on the heuristic's ability to be both admissible (never overestimating the true cost) and consistent, ensuring an optimal solution pathway.

Challenges with Sokoban Macro Actions



```python
def solve_sokoban_macro(warehouse):
    solution = solve_sokoban_elem(warehouse)
    print(f"Initial worker position in warehouse: {warehouse.worker}")
    print(f"Initial boxes positions in warehouse: {warehouse.boxes}")
    curr_warehouse = warehouse
    last_x,last_y = curr_warehouse.worker
    total_macro_move = []

    if "Impossible" in solution or solution is None :
        return total_macro_move.append('Impossible')
    else:
        for action in solution:
            last_x, last_y = curr_warehouse.worker
            move_x,move_y = find_move(action)
            worker_x, worker_y = curr_warehouse.worker
            boxes = curr_warehouse.boxes
            worker_x = worker_x + move_x
            worker_y = worker_y + move_y
            worker = (worker_x,worker_y)
```

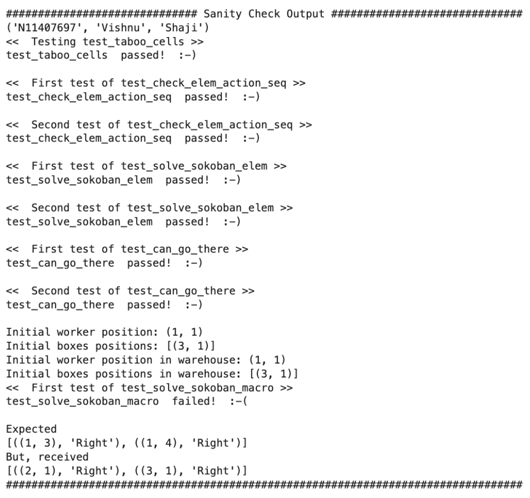
One main challenge was the potential misreading of the worker's initial position. This inconsistency can lead to both algorithms failing to find a solution.

Future work.

Through Sokoban Macro action experiments, I have identified certain challenges that demand attention for improvement. These key areas include:

1. Ensuring initial state accuracy
2. Enhancing algorithm robustness to accommodate various layouts
3. Refining the heuristic approach for A Search*

Tackling these issues head-on will not only bolster the reliability of the Sokoban puzzle solver but also improve its overall efficiency.



```
############################## Sanity Check Output ##############################
('N11407697', 'Vishnu', 'Shaji')
<<  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!  :-)

Initial worker position: (1, 1)
Initial boxes positions: [(3, 1)]
Initial worker position in warehouse: (1, 1)
Initial boxes positions in warehouse: [(3, 1)]
<<  First test of test_solve_sokoban_macro >>
test_solve_sokoban_macro  failed!  :-(

Expected
[((1, 3), 'Right'), ((1, 4), 'Right')]
But, received
[((2, 1), 'Right'), ((3, 1), 'Right')]
#################################################################################
```