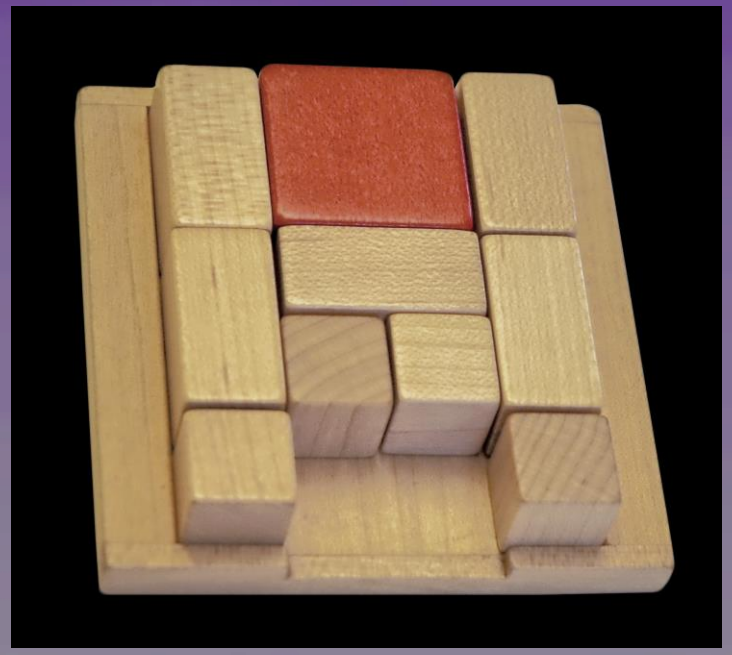
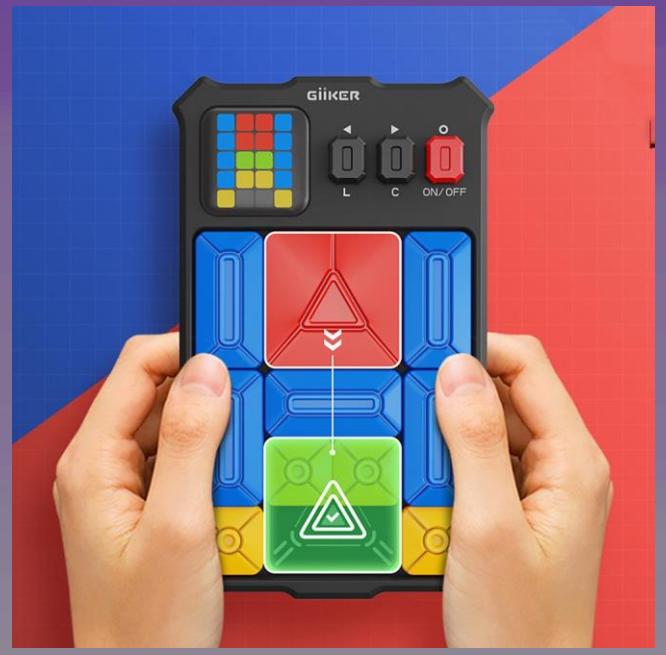
Klotski

Heuristic Search Methods for One Player Solitaire Games



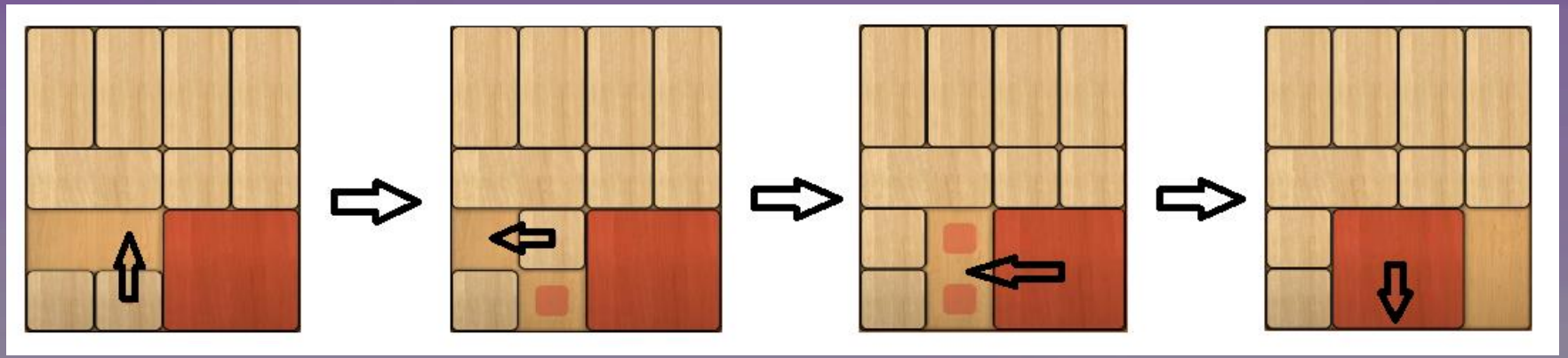


Artificial Inteligence

Group 58_1D

Problem Specification

We are going to implement the Klotski puzzle, a puzzle of sliding blocks in which the goal is to move a specific block to the exit of the board (located on the bottom of the board). The board is a grid made of squares, and each block occupies a determined number of squares. The blocks can only move horizontally or vertically, and they cannot overlap or leave the board. The program should be able to find a solution to any given puzzle in the minimum number of moves.



One exemple of solving this puzzle

Problem Formulation

- State Representation: A 5x4 grid, where each cell contains either a block or an empty space. Each block is represented by its position and size (1x1, 1x2, 2x1 or 2x2) and the number 0 represents the empty spaces. Ex: [[4, 9, 8, 5], [6, 1, 1, 7], [2, 1, 1, 3], [14, 10, 11, 3], [12, 0, 0, 13]]
  - Initial State: [[4, 9, 8, 5], [6, 1, 1, 7], [2, 1, 1, 3], [14, 10, 11, 3], [12, 0, 0, 13]]
  - Objective Test: The objective is to move the largest block (2x2) from its initial position to the exit in the bottom of the grid. It will check if the 2x2 block is in the middle of the bottom row.

```
def is_solved(self):
    return self.board[4][1] == 1 and self.board[4][2] == 1
```

Problem Formulation

- Operators:
  - Move_Up – Moves a block up on the grid.
  - Move_Down – Moves a block down on the grid.
  - Move_Left – Moves a block left on the grid.
  - Move_Right – Moves a block right on the grid.

The preconditions are that a block can't be separated by an empty space and all operators have unitary cost.

```
# Move the piece to the new position
new_board = [row[:] for row in board]
row_delta, col_delta = direction
for r, c in sorted(pieces, reverse=True):
    if new_board[r + row_delta][c + col_delta] == 0:
        new_board[r][c] = 0
        new_board[r + row_delta][c + col_delta] = piece
        if r-row_delta>=0 and r-row_delta<len(board) and c-col_delta>=0 and c-col_delta<len(board[r]) and new_board[r - row_delta][c - col_delta] == piece:
            new_board[r][c] = piece
            new_board[r - row_delta][c - col_delta] = 0
        #print("|"+ str(r) +","+ str(c) + "|\n" + "|"+ str(row_delta) +","+ str(col_delta) + "|\n")

return new_board
```

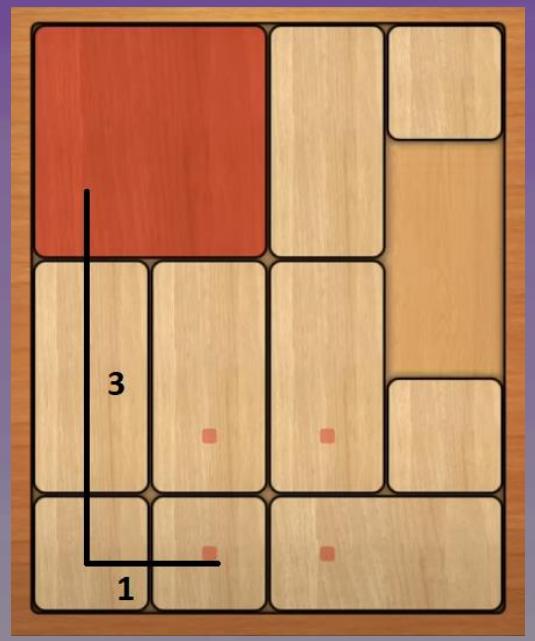
Implemented Heuristics

- We have implemented BFS, DFS and A* Search.
- Our evaluation function, used in A*, calculates the Manhattan Distance between our 2x2 block and the position on the grid where the game is solved.

```
def manhattan_distance(self, goal_state):
    distance = 0
    flat_goal_board = [val for sublist in goal_state.board for val in sublist]
    for i in range(len(self.board)):
        for j in range(len(self.board[0])):
            if self.board[i][j] != 0:
                #print(f'i->{i}, j-> {j}\n')
                #print(f'atual state: {self.board}, goal state: {goal_state.board}')
                # Use the index function on the flattened lists
                index = flat_goal_board.index(self.board[i][j])
                row, col = divmod(index, len(self.board[0]))
                distance += abs(i - row) + abs(j - col)
                #print(row, col)
                #print(distance)
    return distance
```

Approach

- As we mentioned before, our heuristic calculates the Manhattan Distance between our 2x2 block and its position on the grid where the puzzle is solved. We tried to find other admissible heuristics to use in our game but we were not able to find any, so we stayed just with this one.



- In this case, our heuristic has the value 4 (3+1).
- That means our block is located at a Manhattan Distance of 4 from the final position of the puzzle.

Algorithms

- We have implemented the following algorithms: BFS, DFS and A* Search. For hardware reasons, we implemented a maximum depth, making DFS a DFS-Limited.

```python
def bfs(start_state, objective_test, successors, depth_limit=None):
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()

    start_state.depth = 0

    while frontier:
        state = heapq.heappop(frontier)

        if objective_test(state):
            f.write("\nPuzzle Solved!\n")
            return state.move_history

        explored.add(state)

        if depth_limit is None or state.depth <= depth_limit:
            for (successor, _) in successors(state):
                successor.depth = state.depth + 1
                if successor not in explored:
                    heapq.heappush(frontier, successor)

    f.write("No solution found!!")
    return None
```

```python
def dfs(start_state, objective_test, successors, depth_limit):
    frontier = [(start_state, 0)]
    explored = set()

    while frontier:
        state, depth = frontier.pop()

        if depth > depth_limit:
            continue

        if objective_test(state):
            f.write("\nPuzzle Solved!\n")
            return state.move_history

        explored.add(state)

        successors_list = list(successors(state))
        successors_list.reverse()

        for (successor, _) in successors_list:
            if successor not in explored:
                frontier.append((successor, depth + 1))

    f.write("No solution found!!")
    return None
```

```python
def a_star_search(start_state, goal_state, objective_test, successors, heuristic):
    frontier = []
    heapq.heappush(frontier, (heuristic(start_state, goal_state), start_state))
    explored = set()
    #print_board(start_state.board)
    while frontier:
        (cost, state) = heapq.heappop(frontier)

        if objective_test(state):
            f.write("\nPuzzle Solved!\n")
            return state.move_history

        explored.add(state)

        for (successor, action_cost) in successors(state):
            if successor not in explored:
                new_cost = cost - heuristic(state, goal_state) + action_cost + heuristic(successor, goal_state)
                heapq.heappush(frontier, (new_cost, successor))

    f.write("No solution found!!")
    return None
```

Experimental Results

- Initially, we expected that A* algorithm was going to have the best performance, time wise. We also expected the DFS-Limited to have the best performance, space wise. This were our results:

Time	**Level 2 (1 move)**	**Level 3 (27 moves)**	**Level 4 (10 moves)**
BFS	0.203125 s	Too big	0.71875 s
DFS-Limited	0.265625 s	Too big	22.765 s
A*	0.265625 s	0.390625 s	0.287525 s
Space	**Level 2**	**Level 3**	**Level 4**
BFS	23.75 MB	Too big	27.48 MB
DFS-Limited	23.95 MB	Too big	39.43 MB
A*	23.59 MB	23.92 MB	23.55 MB

Conclusions

- As expected, A* algorithm was the best performing in terms of time, except for level 2, where the number of moves needed to finish the level is only 1, and in that case, BFS was the better one.
- For optimal solutions, as expected, A* was the best performing algorithm on both ends.
- DFS is the worst one, time wise, and its optimality depends on its depth limit.
- Unfortunately, we didn't spend the time we should have spend, developing this project, and that affected our results. In a next time, we need to manage our time a lot better.

References and Materials

- Link to the Google Play page;
- All the curricular unit's slides on Moodle;
- Github;

- Python;
- VSCode.