U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Klotski

Assignment 1 Final Delivery

L.EIC - AI - T2 - Group 9

André Tomás da Cunha Soares
Diogo Alexandre da Costa Melo Moreira da Fonte
Jorge Carlos Baptista Duarte

Specification

The objective for this project is to develop a problem solving Artificial Intelligence Agent capable of finding a solution to the game Klotski using different search algorithms.
Klotski is a sliding block puzzle game, with different sized blocks being placed usually inside a 4x5 frame. The goal of the game is to move a special 2x2 block to a designated area without removing blocks, only sliding the blocks horizontally or vertically.

Formulation

State Representation:
State is represented through a RxC numpy array where R and C are the dimension of the board. Every array position has a number corresponding to a block. Number 1 corresponds to blue pieces, 2 corresponds to yellow pieces, 3 represents the red piece, and 0 are empty spots on the board.

Initial State:
Any as long as Special Block is not already in goal coordinates.

Objective State:
Special Block at goal coordinates **(XG, YG): [SpecialBlock, XG, YG]**

Operators:
Each block can be moved **Up, Down, Right, Left** as long as there's free space equal to dimensions of the block.
[Block, direction]
Each move has the same cost.

```
np.array([[1,1,1,1],
          [1,1,1,1],
          [0,1,3,3],
          [0,1,3,3],
          [2,2,2,2]])
```

Figure 1 - Example numpy array for a game board

Implementation

Programming Language:
Python, using pygame library.

The game has a PC Mode and Player Mode available for every level of varying difficulty. PC Mode will solve the puzzle using the selected algorithm, and Player Mode allows player to solve the puzzle with the help of an hint system. Players will receive a score based on the amount of moves and completion time. The game reads levels from .txt files and recognizes boards of different sizes.

Algorithms to Implement:
BFS, DFS, Iterative Deepening Search, A*, Greedy Search.

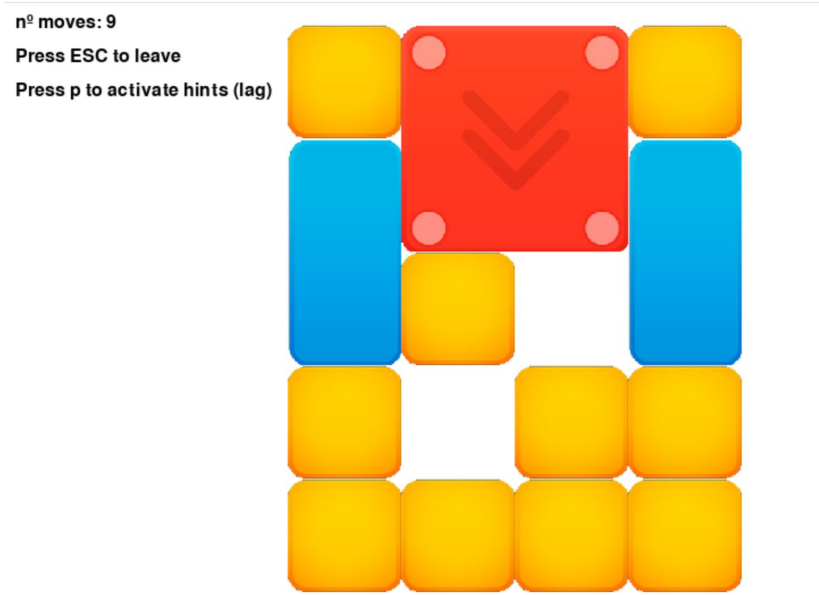


Figure 2 - Implemented interface

Implementation

Data Structures:
State: Numpy array that stores what piece is on each position..
TreeNode: generic definition of a tree node holding a state of the problem.

```
self.state.blocks = np.loadtxt("levels/lvl6.txt", dtype='i',delimiter=',')
```

Figure 2 - Example of a how state is initialized for a level. A numpy array is created from a .txt file.

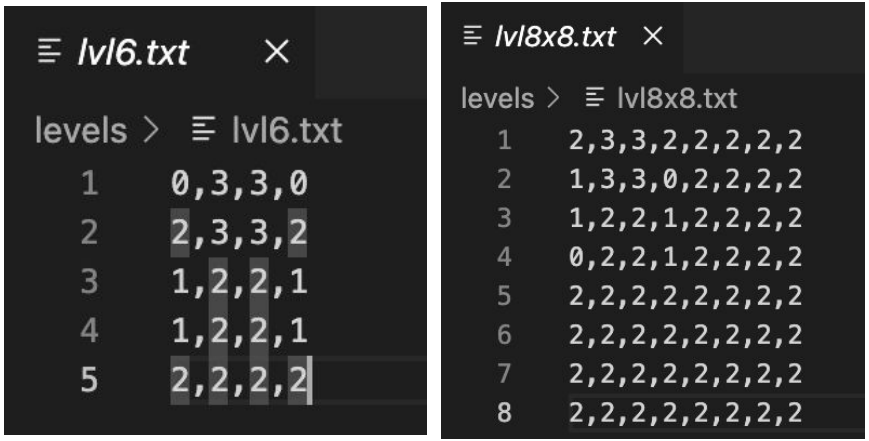


Figure 3 - Example of .txt files with the block disposition for different sized boards

Implementation

Implemented algorithms:
We implemented the uninformed algorithms BFS, DFS, IDS.

```python
def breadth_first_search(initial_state, goal_state_func, operators_func):

    root = TreeNode(initial_state)   # create the root node in the search tree
    queue = deque([root])   # initialize the queue to store the nodes
    tracemalloc.start()
    while queue:
        node = queue.popleft()   # get first element in the queue
        if goal_state_func(node.state):   # check goal
            peak_memory = tracemalloc.get_traced_memory()[1]
            tracemalloc.stop()
            return (node, peak_memory)


        for state in operators_func(node.state):   # go through next states
            # create tree node with the new state
            tempNode = TreeNode(state,node)

            # link child node to its parent in the tree

            node.add_child(tempNode)

            # your code here

            # enqueue the child node

            queue.append(tempNode)


    print('failed')
    return None
```

Figure 4 - Breadth First Search Algorithm

```python
def depth_first_search(initial_state, goal_state_func, operators_func):

    root = TreeNode(initial_state)   # create the root node in the search tree
    stack = [root]  # initialize the queue to store the nodes
    visited =  []  # keep track of visited nodes
    tracemalloc.start()
    while stack:
        node = stack.pop()   # get the last element in the stack
        visited.append(node.state.blocks)  # mark the node as visited
        if goal_state_func(node.state):   # check goal state
            peak_memory = tracemalloc.get_traced_memory()[1]
            tracemalloc.stop()
            return (node, peak_memory)
        for state in operators_func(node.state):   # go through next states
            if not any(np.array_equal(state.blocks, arr) for arr in visited):
                # create tree node with the new state
                tempNode = TreeNode(state, node)
                # link child node to its parent in the tree
                node.add_child(tempNode)
                # enqueue the child node
                stack.append(tempNode)
    return None
```

Figure 5 - Depth First Search Algorithm

```python
def iterative_deepening_search(initial_state, goal_state_func, operators_func, depth_limit):

    for depth in range(depth_limit + 1):
        root = TreeNode(initial_state)   # create the root node in the search tree
        stack = [(root, 0)]   # initialize the stack to store the nodes and their depths
        visited = set()  # keep track of visited nodes
        tracemalloc.start()
        while stack:
            node, node_depth = stack.pop()   # get the last element in the stack
            #visited.add(node.state)  # mark the node as visited
            if goal_state_func(node.state):   # check goal state
                peak_memory = tracemalloc.get_traced_memory()[1]
                tracemalloc.stop()
                return (node, peak_memory)
            if node_depth < depth:  # explore nodes within the depth limit
                for state in operators_func(node.state):   # go through next states
                    if state not in visited:  # explore only unvisited nodes
                        # create tree node with the new state
                        tempNode = TreeNode(state, node)
                        # link child node to its parent in the tree
                        node.add_child(tempNode)
                        # enqueue the child node
                        stack.append((tempNode, node_depth + 1))
    return None
```

Figure 6 - Iterative Deepening Search Algorithm

Implementation

Implemented algorithms:
We implemented the informed algorithms Greedy and A* using two different heuristics.

```python
def greedy_search(problem, heuristic, operators_func):

    # heuristic (function) - the heuristic function that takes a board (matrix), and returns an integer
    setattr(TreeNode, "__lt__", lambda self, other: heuristic(self.state) < heuristic(other.state))
    root = TreeNode(problem)
    states = [root]
    visited = [] # to not visit the same state twice
    tracemalloc.start()
    while states:

        node = heapq.heappop(states)   # get first element in the queue
        visited.append(node.state.blocks)

        if goal_piece_state(node.state):   # check goal state
            peak_memory = tracemalloc.get_traced_memory()[1]
            tracemalloc.stop()
            return (node, peak_memory)


        for child in operators_func(node.state): # go through next states
            if not any(np.array_equal(child.blocks, arr) for arr in visited):
                tempNode = TreeNode(child, node)
                node.add_child(tempNode)


                heapq.heappush(states,tempNode)


    return None
```

Figure 7 - Greedy Search Algorithm

```python
def A_star_search(problem, h1, h2, operators_func):

    root = TreeNode(problem)
    g_score = {tuple(root.state.blocks.flatten()): 0}
    f_score = {tuple(root.state.blocks.flatten()): h1(root.state) + h2(root.state)}

    setattr(TreeNode, "__lt__", lambda self, other: f_score[tuple(self.state.blocks.flatten())] < f_score[tuple(other.state.blocks.flatten())])

    states = [root]
    visited = []
    tracemalloc.start()
    while states:
        node = heapq.heappop(states)
        visited.append(node.state.blocks)

        if goal_piece_state(node.state):
            peak_memory = tracemalloc.get_traced_memory()[1]
            tracemalloc.stop()
            return (node, peak_memory)

        for child_state in operators_func(node.state):
            if any(np.array_equal(child_state.blocks, visited_state) for visited_state in visited):
                continue

            tentative_g_score = g_score[tuple(node.state.blocks.flatten())] + 1  # cost of moving to child is always 1

            if tuple(child_state.blocks.flatten()) not in g_score or tentative_g_score < g_score[tuple(child_state.blocks.flatten())]:
                g_score[tuple(child_state.blocks.flatten())] = tentative_g_score
                f_score[tuple(child_state.blocks.flatten())] = h1(child_state) + h2(child_state)

                child = TreeNode(child_state, node)
                node.add_child(child)

                if not any(np.array_equal(child_state.blocks, visited_state) for visited_state in visited):
                    heapq.heappush(states, child)

    return None
```

Figure 8 - A Search Algorithm*

Approach

Heuristics:

We explored the use of 3 different heuristics

H1- Number of pieces between the red block and the goal.

H2- Manhattan Distance between the red block and the goal. We calculate this by adding the number of columns and number of rows that the red block must pass until it reaches the goal.

H3- Number of blocks surrounding the red block. This heuristic proved inadmissible, since we could have a situation where the red block is surrounded by all sides except the goal.

Evaluation Function:

Our evaluation function consists of checking if the block is in the desired final position, which is in the center of the bottom part of the board

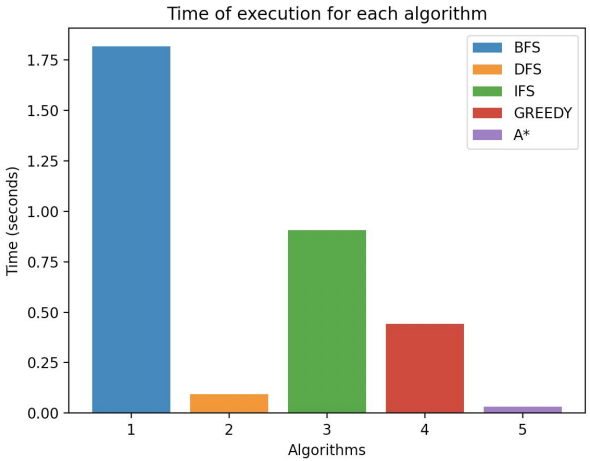
```
def goal_piece_state(state):

    row = state.blocks.shape[0]
    col = state.blocks.shape[1]
    row = row -1
    col = int(math.ceil(col/2)) -1

    return state.blocks[row-1][col] == 3 and state.blocks[row,col+1] == 3
```

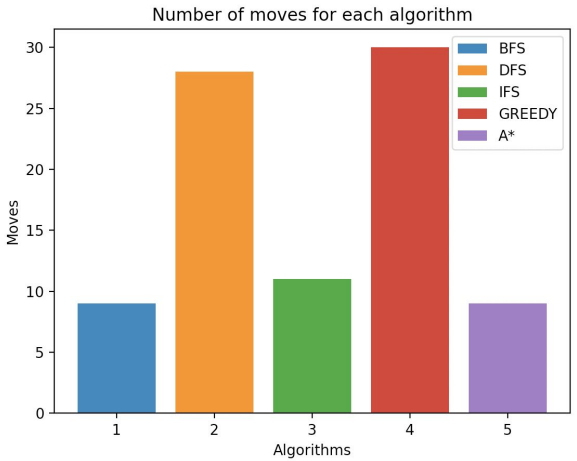
Experimental Results

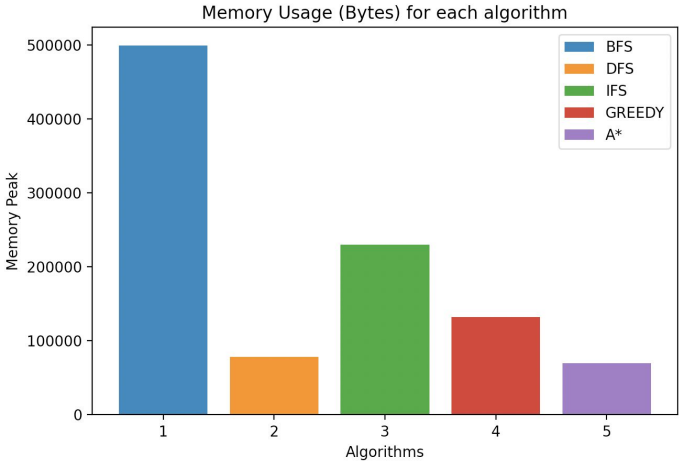
These results were obtained for level 0 of the game. Although DFS was able to run in a very short time, the solution took quite a number of moves, which makes the solution worse.

BFS can reach a great solution, but it takes a lot of time and memory to reach this solution.

We can conclude that, with the increase of complexity of the board, only Greedy and A Star algorithms can reach a solution, but they need good heuristics to reach good solutions.







Conclusions

With this practical work we were able to understand how to use search in solving games. By formulating it as a search problem, that is, with representation of states, operators and preconditions, we managed to use the algorithms that we implemented to search for the various possible solutions.

We were able to realize the importance of choosing good heuristics for the use of algorithms such as A Star and Greedy, as it makes all the difference in the results obtained and in the performance of the algorithm. As the level complexity increased, only the informed algorithms were capable of consistently finding solutions for all levels. That said, for levels where uninformed algorithms were able to find a solution, they could outperform the Greedy algorithm, showing the importance of choosing a good heuristic when designing informed algorithms.

References

Klotski Wikipedia Article - Information about the rules of the game Klotski

Klotski Game Google App Store Page - We used this app as general inspiration for our game and level layouts

Pygame Documentation

OpenAI's ChatGPT - Used to help with automation of extensive similar code