Ball Sort Puzzle

Heuristic Search Methods for One Player Solitaire Games

Artificial Intelligence

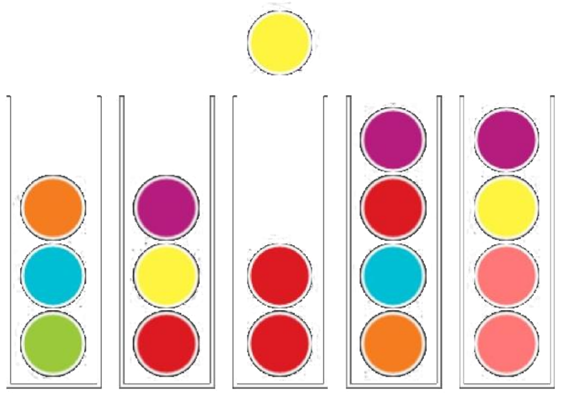
Master in Informatics and Computing Engineering

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Gonçalo Alves – up201806451
Gustavo Mendes – up201806078
Pedro Seixas – up201806227

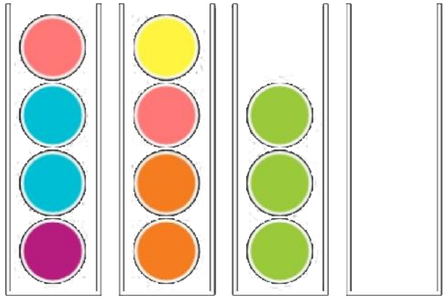
Specification

The objective of the Ball Sort Puzzle is to sort the colored balls in the tubes until all balls with the same color stay in the same tube.



The player can only move a ball at the top of a tube to:

- an empty tube;
- another tube that has a ball with the same color on top and has enough space;



The player can also undo his moves if he finds himself without moves.

Formulation of the problem as a search problem

State Representation:

- Ball: [1..n] (n = the number of colors)
- Tube: [Ball, Ball, Ball, Ball] or []
- Game: [Tube,Tube,…]

Initial State:

- Game: [Tube,Tube,…], where every Tube doesn't contain 4 equal numbers

Objective State:

- Game: [Tube,Tube,…], where every Tube either contains 4 equal numbers or is empty

Operators:

- Move(X,Y):
  - PreCond: Tube Y must be empty or have a Ball, on top, just like the one that is moved
  - Effect: move Ball from Tube X to Tube Y
  - Cost: 1 is the general case, it represents a move; C2, evaluation of the number of wrongly placed Balls, based on the Ball at the bottom of a Tube

Implemented Heuristics

We have implemented BFS, DFS, IDS, Greedy Search and A*.

Our evaluation function, used in A*, calculates the number of wrongly placed Balls (Balls with different color of the Ball at the bottom) in a Tube.

Code:

```python
def greedy(state: Node, a_star: bool = False, max_depth: int = 5000):
    graph = Graph()
    state.setDist(0)
    stack = [state]
    graph.new_depth()
    graph.add_node(state, 1)

    depth = 1
    while depth != max_depth and len(stack) != 0:
        if a_star:
            stack.sort()
        else:
            stack.sort(key = lambda x: x.cost)
        graph.new_depth()
        node = stack.pop(0)
        if node.gamestate.finished():
            print("Found Goal. Depth:", node.dist)
            return graph, node
        else:
            graph.visit(node)
            expanded = new_states(node, a_star)
            [graph.add_node(x, depth + 1) for x in expanded]
            for children in expanded:
                if children in graph.visited:
                    continue
                if children in stack:
                    if (children.cost + node.dist + 1 < children.cost + children.dist) and a_star:
                        children.setParent(node)
                        children.setDist(node.dist + 1)
                else:
                    stack.append(children)

        print(depth)
        depth += 1
```

```python
def better_nWrong_heuristics(self):
    cost = 0

    for tube in self.gamestate.tubes:
        balls = tube.balls.copy()
        idx = next((i for i, v in enumerate(balls) if v != balls[0]), -1)
        if idx == -1:
            continue

        balls = balls[idx:]
        cost += len(balls) * 2

    return cost
```

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Implementation work already carried out

Programming language: Python, with visualization using pygame package

Development environment: VSCode/IntelliJ

Data Structures:

- Lists, for representing the Tubes and the Game
- Nodes and Graphs

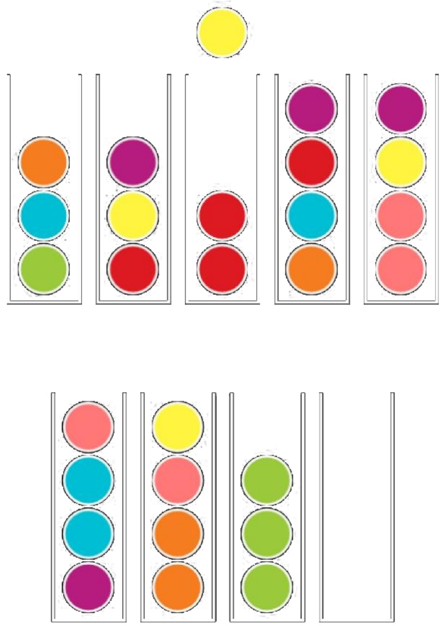
File Structure: The Project's Repository is available at Github

Implemented Work: All algorithms lectured; Graphical Interface, playable game with hints

Approach

As described before, our first heuristic counted the number of wrongly placed balls in a tube, assuming that it will be needed at least one move per ball to put them in the correct place. For this second part of the project, we developed two new heuristics. The first one calculates the maximum number of consecutive balls of the same color for each color and estimates the cost by calculating how many are needed to have the 4 balls of the same color in the same tube. This heuristic calculates a score based on the number of consecutive balls of the same color and the number of empty tubes.

Ex:



1st heuristic: 2+2+1+3+2+3+2 = 15

2nd heuristic: (4-3)+(4-2)+(4-2)+(4-2)+(4-1)+(4-1) = 13

3rd heuristic: 5+5+5+5+5+5+5+15+10 = 60

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Algorithms

We have implemented all lectured algorithms: BFS, DFS, IDS, Greedy and A*. For hardware reasons, we implemented a maximum depth, making DFS a DFS-Limited.

We developed a generic “solver” function, that changes the way the nodes are expanded depending on the algorithm. For IDS, it was necessary to make a separate function, since it uses DFS. (See below)

```python
def solver(start_node: Node, algorithm: Algorithm, max_depth: int = 5000):
    graph = Graph()
    stack = [start_node]

    graph.new_depth()
    graph.add_node(start_node, 1)

    graph.new_depth()

    while len(stack) != 0:
        node = stack.pop(0)

        visited_node = get_stack_item(graph.visited, node)
        if visited_node is not None and node.dist >= visited_node.dist:
            continue

        graph.visit(node)
        graph.add_node(node, node.dist + 1)

        if algorithm != Algorithm.IDS and node.gamestate.finished():
            print("Found goal!")
            return graph, node

        expanded = expand_node(node, graph.visited, algorithm)

        if node.dist < max_depth - 1:
            stack = add_states_to_stack(stack, expanded, algorithm, node)
        else:
            solution = check_final_depth_solution(expanded)
            if solution is not None:
                return graph, solution
```

```python
def ids(start_node: Node, max_depth: int = 5000):
    graph = None
    for depth in range(1, max_depth):
        graph, node = solver(start_node, Algorithm.IDS, depth)
        if node is not None:
            return graph, node

    return graph, None
```

```python
def add_states_to_stack(stack, new_states, algorithm: Algorithm, node: Node):
    if algorithm == Algorithm.BFS:
        stack = stack + new_states
    elif algorithm == Algorithm.DFS or algorithm == Algorithm.IDS:
        stack = new_states + stack
    elif algorithm == Algorithm.GREEDY:
        stack = stack + new_states
        stack = sorted(stack, key=lambda x: x.cost, reverse=True)
        #stack.sort(key=lambda x: x.cost)
    elif algorithm == Algorithm.A_STAR:
        for children in new_states:
            stack_node = get_stack_item(stack, children)
            if stack_node is not None:
                if children.getTotalCost() < stack_node.getTotalCost():
                    stack_node.setParent(node)
                    stack_node.setDist(node.dist + 1)
            else:
                stack.append(children)

        stack = sorted(stack, key=lambda x: x.getTotalCost(), reverse=True)
        #stack.sort(key=lambda x: x.getTotalCost())

    return stack
```

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Experimental Results

Initially, we expected that Greedy algorithm was going to have the best performance, time wise, since it tries to find a solution, based on a heuristic, and does not care for optimality.

Space wise, we expected that DFS-Limited would be the best, based on the table below.

Name	Time	Space	Optimal
BFS	$O(b^{d+1})$	$O(b^{d+1})$	YES
DFS-Limited	$O(b^l)$	$O(bl)$	NO
IDS	$O(b^d)$	$O(bd)$	YES
Greedy	$O(b^m)$	$O(b^m)$	NO
A*	$O(b^m)$	$O(b^m)$	YES

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Experimental Results

To analyze and compare the different algorithms, we used 25 pre-existing levels. Below are the results of such tests.

Name	Completed Levels	Expanded Nodes - Avg	Time Execution (s) - Avg
BFS	6	246	0.14152
DFS-Limited	25	54	0.02335
IDS	4	1162	2.77309
Greedy (Wrong Colours)	25	34	0.01437
Greedy (Minimum Moves)	23	84	0.04154
A* (Wrong Colours)	25	852	1.62916
A* (Minimum Moves)	25	1013	2.18084

Notes:
- The algorithms with less completed levels did not find a solution with the given max depth or took too much time
- While the 25 levels have different sizes, size has no correlation with solution size or difficulty and therefore, was not taken into consideration

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Conclusions

As expected, Greedy was the best performing in terms of time, but surprisingly it was also the one that expanded the least nodes, making it the best performing algorithm overall, for non-optimal solutions.

For optimal solutions, as expected, A* was the best performing algorithm on both ends.

Between our heuristic functions, Wrong Color was the best one. Minimum Moves is better suited for small solutions or when the puzzle is almost finished.

Unfortunately, after analyzing the results for our third heuristic function, we found out it was not admissible, since it was overestimating the cost. As such, its results were not included, but the code can still be found in the graph.py file.

References and Materials

- Link to the Google Play page;
- Link of all existing levels;

We used the curricular unit's presentations to guide our implementation of the various algorithms.

- Python, with visualization using pygame package and xltw for spreadsheet generation;
- VSCode/IntelliJ