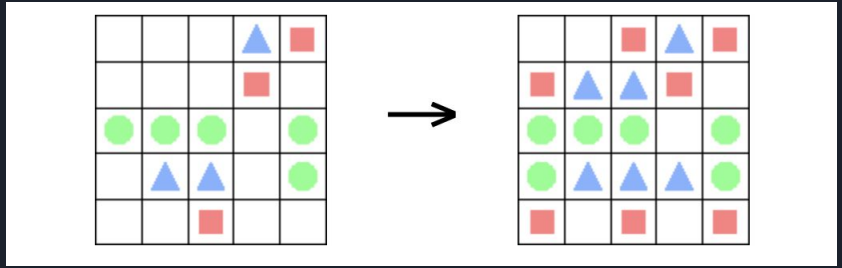
Symmetry puzzles

up201908253-Francisco Nunes
up201907156-António Santos
up201906576-José Silva

Project specification

The objective of Symmetry puzzles is to form palindromes in each row and column, by adding shapes, in this case, squares,circles and triangles in some of the empty cells in the puzzle. Each puzzle has a unique solution.



Formulation of the problem

State Representation:

- Array with 2 dimensions , some of which contain shapes(circles,triangles,squares) and some of which are blank.
- Each shape in the array with 2 dimensions is associated with a row and a column index.

Initial State:

- The puzzle with a random display of the shapes.

Objective State:

- Ignoring the blank spaces, a valid state where, the shapes in each row and column should be palindromes.

Operators:

- Add a triangle in the puzzle

Precondition: the cell that will be added must be empty.
Effect: Transition to a new state of the puzzle where the position added is filled with a shape.
Cost: 1

- Add a square in the puzzle

Precondition: the cell that will be added must be empty.
Effect: Transition to a new state of the puzzle where the position added is filled with a shape.
Cost: 1

- Add a circle in the puzzle

Precondition: the cell that will be added must be empty.
Effect: Transition to a new state of the puzzle where the position added is filled with a shape.
Cost: 1

Heuristics/Evaluation function

The heuristics that we think to implement:

- The heuristic that we use as function give us the sum of the rows and columns that are not palíndromes.

```python
def h1(state):
    # count non-palindromic rows
    non_palindromic_rows = 0
    for row in state.board:
        row_trimmed = [c for c in row if c != 0]
        if len(row_trimmed) > 1 and row_trimmed != row_trimmed[::-1]:
            non_palindromic_rows += 1
    # count non-palindromic columns
    non_palindromic_cols = 0
    for j in range(len(state.board[0])):
        col = [state.board[i][j] for i in range(len(state.board))]
        col_trimmed = [c for c in col if c != 0]
        if len(col_trimmed) > 1 and col_trimmed != col_trimmed[::-1]:
            non_palindromic_cols += 1
    # return the sum of non-palindromic rows and columns
    return (non_palindromic_rows + non_palindromic_cols)/2
```

Implementation work already carried out

Programming Language: Python, with visualization using pygame package.

Development environment: Visual studio code, Github.

Data structures: Array with 2 dimensions.

Implemented algorithms

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

```python
def bfs(problem):
        queue = deque([problem]) # can also also use array and pop(0)
        visited = set() # to not visit the same state twice

        while queue:

            node = queue.popleft()

            if node.objective_state():
                return node.move_history

            for child in node.children():
                if(child not in visited):
                    count = count + 1
                    print(child.board)
                    visited.add(child)
                    queue.append(child)
        return None
```

```python
def dfs(problem):
        stack = deque([problem]) # can also also use array and pop(0)
        visited = set() # to not visit the same state twice
        while stack:
            node = stack.pop();

            if node.objective_state():
                return node.move_history

            for child in node.children():
                if(child not in visited):
                    visited.add(child)
                    stack.append(child)
        return None
```

```python
def greedy_search(problem, heuristic):
    setattr(Symmetry_puzzle,"__lt__", lambda self, other: heuristic(self) < heuristic(other))
    states = [problem]
    visited = set()


    while states:
        node = heapq.heappop(states)
        visited.add(node)

        if node.objective_state():
            return node.move_history

        for child in node.children():

            if child not in visited:
                print(child.board)
                heapq.heappush(states,child)
        heapq.heapify(states)
    return None
```

```python
def a_star_search(problem, heuristic):
    setattr(Symmetry_puzzle, "__lt__", lambda self, other: heuristic(self) + len(self.move_history) < heuristic(other) + len(other.move_history))
    states = [problem]
    visited = set()

    while states:
        node = heapq.heappop(states)
        visited.add(node)

        if node.objective_state():
            return node.move_history

        for child in node.children():
            if child not in visited:
                print(child.board)
                heapq.heappush(states, child)
        heapq.heapify(states)


    return None
```

Experimental results

For all algorithms using board 3x3 dimensions we can verify that the time performance is around 1-2 seconds and the steps to perform the algorithms is around 3-4 steps.

For the first example of statement of the project, with greedy algorithm , the time performance was 8.25 seconds and the steps to solve the problem was 8 steps.

Conclusions

We managed to implement the algorithms BFS,DFS,GREEDY,A*. We were able to conclude that for larger matrices, both for BFS and for DFS, it would take a long time to find the solution. As for the Greedy Algorithm, we were able to find the solution for 3x3 and 4x4 matrices and for some concrete examples we were not always able to find the optimal solution.

For the a* algorithm, the same thing happened to the Greedy algorithm.

We faced some challenges in locating information related to the game and comprehending the reason why we were unable to achieve the desired solution. At times, this hindered our progress with the project.

References and Materials

- Python, with pygame package.
- Visual studio code, Github