# Part 1: State Representation

## Components:

Pirate Ships: Represents the positions of pirate ships on the grid. This is a dictionary where it’s key is pirate\_id and value is the pirate position.

**Treasures**: Represents the positions of treasures on the grid. This is a dictionary where it’s key is treasure\_id and value is the treasure position

**Marine Ships:** Represents the path of marine ships on the grid. This is a dictionary where it’s key is marine\_id and value is a list where is the path.

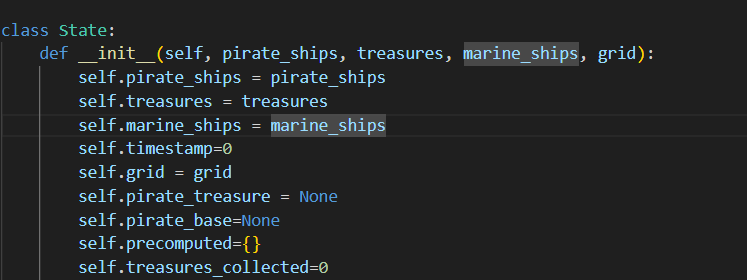
**Timestamp:** Represents the current timestamp or time step. This is an integer each step increases by 1, this is important for marine positions, increases by 1 in the end of the action.

**Grid:** Represents the layout of the grid, containing information about land ('I'), sea ('S'), and pirate bases ('B'). This is a matrix this will be unchanged in all the problem.

**Pirate Treasure:** Represents the treasures collected by each pirate ship. This is a dictionary where it’s key is a pirate\_id its value is other dictionary where stores marines\_id-> marine\_position.

**Precomputed:** Stores precomputed distances for optimization purposes. This will be important for heuristic.

**Treasures Collected:** Keeps track of the total number of treasures collected.



## Hashing:

- The `\_\_hash\_\_` method in the `State` class computes a hash value based on the state's components (pirate ships, treasures, pirate treasure, and timestamp). It uses the `hash` function to hash a tuple containing these components. Some values like dictionaries or list must be transformed in immutable structures like tuples or frozensets.

## Comparison "Equal":

- The `\_\_eq\_\_` method in the `State` class checks whether two states are equal by comparing their components. It returns `True` if all corresponding components are equal, and `False` otherwise.

Texto

Descripción generada automáticamente

# Part 2: Problem Representation

## How to Expand:

The `actions` method in the `OnePieceProblem` class defines the possible actions that can be taken in a given state. It returns a list of actions such as sailing to adjacent sea cells, collecting treasures, depositing treasures, and waiting.

Something important to notice it is not necessarily to move all ships all problems can be solve optimally with just one ship, this helps for priority queue reducing searching nodes.

There are for actions.

Sail: moves a pirate ship in an available adjacent cell

**Collect:** collect a treasure if the pirate is adjacent to the island with a treasure just can move 2 treasures at time

**Deposit:** deposit all treasures in base.

**Wait:** pirate ship doesn’t move.

In all cases timestamp for new\_state increases

This function returns all possible actions from this state.

- The `result` method in the `OnePieceProblem` class defines how the state changes when an action is taken. It modifies the state based on the action performed, updating the positions of pirate ships, treasures, and marine ships accordingly.

This function applies changes for the action in this state creating a new state.

Texto

Descripción generada automáticamente

## How to Manage Marine Positions:

- Marine positions are managed by checking if the position of a pirate ship coincides with the position of a marine ship during an action. If they coincide, treasures collected by the pirate ship are returned to the map, simulating a loss to the marine ship. This is checked in collect or sail action but not in deposit.

## How Heuristics Work:

- The `h` method in the `OnePieceProblem` class defines heuristic functions for estimating the remaining cost to reach the goal state from a given node in the search tree.

- Two heuristic functions (`h\_1` and `h\_2`) are implemented, each estimating the remaining cost differently. These functions consider factors such as the number of uncollected treasures, the number of pirate ships, the total distance to remaining treasures, and the presence of a pirate base.

h\_1 estimates which uncollected treasures are.

Texto

Descripción generada automáticamente

h\_2 estimates minimal distance from adjacent treasures, if a treasure has not adjacent treasures will return infinity.

Texto

Descripción generada automáticamente

Imagen que contiene Texto

Descripción generada automáticamente

Part 3: A\*

- A\* search is implemented in the `astar\_search` function. It uses a priority queue implemented with heap to manage the frontier of nodes, where the priority is based on the sum of the path cost and the heuristic value.

- The function iterates until the frontier is empty, expanding nodes and updating the best path cost to each state along the way.

- A goal test is performed on expanded nodes, and if a goal state is found, the solution is returned.

- The heuristic function (`h`) is used to estimate the remaining cost to reach the goal state from each node, guiding the search towards promising paths.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente