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.

```
class State:
    def __init__(self, pirate_ships, treasures, marine_ships, grid):
        self.pirate_ships = pirate_ships
        self.treasures = treasures
        self.marine_ships = marine_ships
        self.timestamp=0
        self.grid = grid
        self.pirate_treasure = None
        self.pirate_base=None
        self.precomputed={}
        self.treasures_collected=0
```

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.

```
def __eq__(self, other):
    """Check if two states are equal."""
    return (self.pirate_ships,
                self.treasures,
                self.pirate_treasure,
                self.timestamp) ==(other.pirate_ships,
                other.treasures,
                other.pirate_treasure,
                other.timestamp)
def __lt__(self,other):
    return (len(self.treasures)+self.treasures_collected)<(len(other.treasures)+other.trea

def __hash__(self):
    """Hash function for the state."""
    return hash((tuple(self.pirate_ships.values()),
                frozenset(self.treasures.values()),
                tuple(map(lambda x: tuple(x),self.pirate_treasure.values())),
                self.timestamp))
```

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.

```
def actions(self, state):
    actions = []


    pirate_id ="pirate_ship_1"
    pirate_pos =state.pirate_ships[pirate_id]
    if len(state.pirate_treasure[pirate_id])<2:
        for dx, dy in orientations:
            island_x, island_y = pirate_pos[0] + dx, pirate_pos[1] + dy
            if (0 <= island_x < len(state.grid) and
                    0 <= island_y < len(state.grid[0]) and
                    state.grid[island_x][island_y] == 'I'):
                for treasure_id, treasure_pos in state.treasures.items():
                    if ((island_x==treasure_pos[0])and(island_y==treasure_pos[1])):
                        actions.append(('collect_treasure', pirate_id, treasure_id))
    if state.grid[pirate_pos[0]][pirate_pos[1]] == 'B' and len(state.pirate_treasure[pirate_id])!=0:
        actions.append(('deposit_treasure', pirate_id))
    for dx, dy in orientations:
        new_x, new_y = pirate_pos[0] + dx, pirate_pos[1] + dy
        if (0 <= new_x < len(state.grid) and
                0 <= new_y < len(state.grid[0]) and
                state.grid[new_x][new_y] in  'SB'):
            actions.append(('sail', pirate_id, (new_x, new_y)))
    actions.append(('wait', pirate_id))



    return actions
```


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.

```
def h_1(self, node):
    """
    Heuristic function for the One Piece problem.
    Estimates the remaining cost to reach the goal state from the given node.
    """
    state = node.state
    num_uncollected_treasures = len(state.treasures)+state.treasures_collected
    num_pirates = len(state.pirate_ships)
    if num_pirates == 0:
        return 0  # To avoid division by zero if there are no pirate ships
    return num_uncollected_treasures // num_pirates
```

h_2 estimates minimal distance from adjacent treasures, if a treasure has not adjacent treasures will return infinity.

```python
def h_2(self, node):
    """
    Heuristic function for the One Piece problem.
    Estimates the remaining cost to reach the goal state from the given node.
    """
    state = node.state
    pirate_ships = state.pirate_ships
    treasures = state.treasures
    grid = state.grid
    # Find the position of the pirate base
    pirate_base=state.pirate_base

    if not pirate_base:

        return float('inf')  # If there's no pirate base, return infinity
    num_pirates = len(state.pirate_ships)
    if num_pirates == 0:
        return 0  # To avoid division by zero if there are no pirate ships
    total_distance = sum([state.precomputed[treasure_id] for treasure_id in treasures])
    # Divide the total distance by the number of pirate ships
    return total_distance // num_pirates
```

```python
        self.pirate_treasure={p:{} for p in pirate_ships}

        for marine in self.marine_ships:
            self.marine_ships[marine]=self.marine_ships[marine]+list(reversed(self.marine_ships[marine][1:-1]))
        for i in range(len(self.grid)):
            for j in range(len(self.grid[i])):
                if self.grid[i][j] == 'B':
                    self.pirate_base = (i, j)
                    break
            if self.pirate_base:
                break
        for treasure_id,treasure_pos in treasures.items():
            # Check adjacent cells to the treasure
            adjacent_cells = [(treasure_pos[0] + dx, treasure_pos[1] + dy) for dx, dy in orientations]
            is_adjacent_to_island = all(grid[x][y] == 'I' for x, y in adjacent_cells if 0 <= x < len(grid) and 0 <= y < len(grid[0]))
            if is_adjacent_to_island:
                self.precomputed[treasure_id]= 10e10
                continue  # If all adjacent cells are islands, return infinity
            # Calculate the distance from the pirate base to the closest sea cell adjacent to the treasure
            min_distance_to_sea = min(distance(self.pirate_base,(x,y)) for x, y in adjacent_cells if 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] == 'S
```

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.

```python
def astar_search(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""

    if h is None:
        h = problem.h  # Use the default heuristic function from the problem

    # Initialize the frontier using a priority queue
    frontier = []
    # Add the initial node to the frontier
    frontier.append((0,Node(problem.initial)))

    # Initialize an empty dictionary to keep track of the best path cost to each state
    best_path_cost = {problem.initial: 0}
    depth=60
    # Iterate until the frontier is empty
    while frontier:
        # Pop the node with the lowest f(n) value from the frontier
        c,node = heapq.heappop(frontier)

        # If the node contains a goal state, return the solution
        if problem.goal_test(node.state):
            return node

        # Expand the current node and add its children to the frontier
        for child in node.expand(problem):
            # Calculate the child's path cost
            new_path_cost = node.path_cost + problem.path_cost(node.path_cost, node.state, child.action, child.state)
```

```python
        # Expand the current node and add its children to the frontier
        for child in node.expand(problem):
            # Calculate the child's path cost
            new_path_cost = node.path_cost + problem.path_cost(node.path_cost, node.state, child.action, child.state)

            # If the child state has not been visited or has a better path cost, update it
            if child.state not in best_path_cost or new_path_cost < best_path_cost[child.state]:
                best_path_cost[child.state] = new_path_cost
                child.path_cost = new_path_cost
                try:
                    heapq.heappush(frontier,(int(new_path_cost + h(child)),child))
                except OverflowError:
                    return None

    # If the frontier is empty and no goal state is found, return None (no solution)
    return None
```