Homework 1 - Deterministic Search

Introduction

You're the captain of a pirate fleet in the world of "One Piece". Your goal is to find treasure chests scattered across various islands and return them to your base, all while avoiding the patrolling Marine ships. To achieve this most efficiently, you must make use of the search algorithms shown in class, with the first task being modeling the problem precisely.

Environment

The environment is a representation of the “Grand Line” and is modelled as a rectangular grid, with details provided in the “Input” section as a dictionary. Each cell within this grid symbolizes a different region in the “Grand Line” and can have different properties. A region can either be a sea, an island (which may contain treasures), the Pirate Base, or a cell occupied by a Marine Ship.
Here’s a detailed breakdown of the regions:

1. **Sea (S):** A navigable, passable region.
2. **Island (I):** A location where the ship can dock to collect possibly available treasures, the ship can dock and collect the treasure only if it is on an adjacent cell to an Island.
3. **Pirate Base (B):** The starting and returning point for all ships, where collected treasures are deposited. The ships can dock at the base to deposit treasures and start/continue its journey from there.

The core objective is to navigate strategically to collect ALL of the treasures.
In addition:

- A pirate ship can hold up to two treasures at any turn.
- If a pirate ship encounters a Marine Ship, it forfeits any collected treasure but can continue the journey to recollect the treasures.
- The treasures are not finite; they don’t disappear after being collected and can be gathered multiple times if needed. However, the objective (as seen later on), is not to collect the most treasures but all of them.
- Each action in this environment occurs in discrete turns, altering the state of the environment and depicting the current positions of the pirate ship and Marine Ships.

Actions

You assume control of the pirate ships, which can perform various actions to navigate the seas, collect treasures, and avoid Marine ships:

1. **Sail**: The ship can move up to one tile vertically or horizontally on the grid (the ship cannot move diagonally), and it cannot sail into an impassable tile. The action’s syntax is (“sail”, pirate_ship, (x, y)). For example, if you want to move the ship “ship_1” to a tile (0,2), the correct syntax for this action is (“sail”, “ship_1”, (0, 2)).
2. **Collect Treasure:** The ship can collect treasures if it is in an adjacent tile to an island with a treasure. The ship can hold up to two treasures at once. The syntax for this action is (“collect_treasure”, pirate_ship, “treasure_name”).
3. **Deposit Treasure:** If the ship is at the base or sails into the base, it can dock to deposit the collected treasures. The syntax for this action is (“deposit_treasures”, “pirate_ship”).
4. **Wait**: The ship can choose to wait, which does not change anything about its state or position. The syntax for this action is (“wait”, “pirate_ship”).

Each of the above actions is performed in turns or timestamps, allowing the environment and other entities like Marine ships to react or change positions based on the chosen action.
These four actions are known as atomic actions, as they relate to a single ship. Since you can control multiple ships, you can command them to do things simultaneously. The syntax of a full action is a tuple of atomic actions. Each ship must do something during any given turn, even if it is waiting.
The actions are performed synchronously.

Examples of a Valid Atomic Action:

- If you want the ship "pirate_1" to sail to a tile (1,3): ("sail", "pirate_1", (1, 3))
- If you want the ship "pirate_1" to dock and collect a treasure: ("collect_treasure", "pirate_1", "treasure_1")
- If you want the ship "pirate_1" to return to the base and deposit treasures: ("deposit_treasure", "pirate_1")
- If you want the ship to wait: ("wait", "pirate_1")

Note:

- Ensure that each ship does not attempt to sail into impassable tiles, such as those occupied by islands.
- Keep track of the ship's carried treasures and make sure not to exceed the carrying capacity, which is up to two treasures at any turn.

Examples of a Valid Action:
If you have one ship: (("wait", "pirate_1"),)
If you have 2 ships: (("wait", "pirate_1"),("move", "pirate_2", (1, 2)))

Additional clarification

1. **Collecting Treasures**: Treasures do not disappear upon collection and can be collected multiple times. To collect a treasure, a pirate ship must dock at an island in an adjacent tile. A pirate ship can hold up to two treasures at any turn. At each turn a pirate ship can collect at most one treasure.
2. **Encounter with Marine Ships**: If a pirate ship ends its turn on an a tile with a Marine ship (after they both moved), the Marine ship will confiscate any treasures the pirate ship is carrying. However, the pirate ship can continue its journey unimpeded, and it can collect the same treasures again.
   In case a pirate ship encounters a marine ship in the same turn it collects a treasure, the treasure is also confiscated.
   In case a pirate ship encounters a marine ship in the same turn it deposits a treasure, the treasure is deposited and not confiscated.
3. **Marine Ships' Movement**: Marine ships move only on sea regions. They move along a pre-determined path, which is provided in the input. They can move one tile vertically or horizontally (no diagonal moves) per turn in the sea tiles. If a marine ship has a path of A→B→C→D, instead of returning to point A after reaching point D, it will move to point C, and continue retracing the path in reverse, i.e., D→C→B→A, and then again A→B→C→D, and so forth. So, the path will be A→B→C→D→C→B→A→B…
4. **Pirate Base**: The pirate base is the starting location of all ships, and it is also the place where collected treasures are to be deposited. Each ship can pass and dock at the pirate base to deposit the treasures.

5. **Island Docking**: A pirate ship must be in an adjacent cell to the island to perform the "dock_collect" action to collect treasures.
6. **Objective**: The primary objective is to avoid the Marine ships while collecting treasures and returning them to the pirate base. Strategically planning the ship's actions, taking into account the Marine ships' paths, is crucial to avoid losing the collected treasures while navigating towards the islands and the pirate base.

Goal

The overall goal is to ensure that every treasure is successfully collected and returned to the Pirate Base while minimizing the number of turns.

Input and the task

As an input, you will get a dictionary that describes the initial environment.
"**map**": A list of lists of strings, where for example 'S' represents sea terrain that is passable for ships, and 'I' represents islands, which are impassable.
"pirate_ships": Contains the initial position of your pirate ships in the problem.
"marine_ships": Details the paths for every marine ship, representing their pre-determined movement pattern through the grid.
Full example of a dictionary:

```
{
    "map": [
        ['S', 'S', 'I', 'S'],
        ['S', 'S', 'S', 'S'],
        ['B', 'S', 'S', 'S'],
        ['S', 'S', 'S', 'S']
    ],
    "pirate_ships": {"pirate_ship_1": (2, 0)},
    "treasures": {'treasure_1': (0, 2)},
    "marine_ships": {'marine_1': [(1, 1), (1, 2), (2, 2), (2, 1)]}
},
```

This input is provided to the constructor of the class **OnePieceProblem** as the variable initial. This variable is then used to create the root node of the search, hence you will need to transform it into your representation of the state before the **search.Problem.__init__(self, initial)** line.
Implementation Requirements:
You will need to implement the following functions in the **OnePieceProblem** class:

1. **def actions(self, state)**:
    - Returns all available actions from a given state.
2. **def result(self, state, action)**:
    - Returns the next state, given a previous state and an action.
3. **def goal_test(self, state)**:
    - Returns 'True' if a given state is a goal, 'False' otherwise.
4. **def h(self, node)**:
    - Returns a heuristic estimate of a given node.
5. **def h_1(self, node):**
    - number of uncollected treasures divided by the number of pirates.
6. **def h_2(self, node):**
    - Sum of the distances from the pirate base to the closest sea cell adjacent to a treasure - for each treasure, divided by the number of pirates. If there is a treasure which all the adjacent cells are islands – return infinity.

Note on State Representation:

You are free to choose your own representation of the state. However, it must be Hashable. We strongly suggest sticking to the built-in data types and not creating a new class for storing the state. Nevertheless, you might find unhashable data structures useful, so consider functions that transform your data structure to a hashable one and vice versa.

Evaluating your solution

Having implemented all the functions above, you will launch the A* search by running check.py. It is your job to implement the A* algorithm by completing it in the search.py file. Your code is expected to finish the task in 60 seconds.

Output

You may encounter one of the following outputs:

- A bug - self-explanatory
- (-2, -2, None) - No solution was found. This output can stem either from the insolvability of the problem or from a timeout (it took more than 60 seconds of real-time for the algorithm to finish).
- A solution of the form (list of actions taken, run time, number of turns)

Code handout

Code that you receive has 4 files:

1. ex1.py - the main file that you should modify, implements the specific problem.
2. check.py - the file that includes some wrappers and inputs, the file that you should run.
3. search.py - a file that has the implementation of the A* function.
4. utils.py - the file that contains some utility functions. You may use the contents of this file as you see fit.

Note: we do not provide any means to check whether the solution your code provided is correct, so, it is your responsibility to validate your solutions.

Submission and grading

You are to submit only the files named ex1_ID1_ID2.py and search_ID1_ID2.py as python files (no zip, rar, etc.). We will run check.py with our own inputs, and your ex1_ID1_ID2.py and search_ID1_ID2.py.
IMPORTANT, replace ID1 and ID2 in the names of the file with YOUR IDs.
If you do not have a partner, the name should be ex1_ID.py and search_ID.py.
The check is fully automated, so it is important to be careful with the names of functions and classes. The grades will be assigned as follows:

- 20 points – creating the h_1 heuristic.
- 25 points – creating the h_2 heuristic.
- 35 points – Solving all the problems with only one pirate ship, in under 60 seconds.
- 25 points – competitive part. Computing the optimal plans (minimal number of turns) in under 60 seconds, for as many problems as possible. The grade will be relative to the other students in the class. The problems which will be tested might be bigger than in the non-competitive part.
- Notice, you can get more than 100 points in this HW. Scores above 100 will **not** be rounded down to 100.
- The submission is due on the 14.2.2024, at 23:59
- Submission in pairs/singles only.
- Write your ID numbers in the appropriate field ('ids' in ex1.py) as strings. If you submit alone, leave only one string.

Important notes

- You are free to add your own functions and classes as needed, keep them in the ex1.py file or search.py file.
- Note that you can access the state of the node by using node.state (for calculating h for example)
- We encourage you to start by implementing a working system without a heuristic and add the heuristic later.
- We encourage you to double-check the syntax of the actions as the check is automated.
- More inputs will be released a week after the release of the exercise.
- You may use any package that appears in the standard library, however, the exercise is built in a way that most packages will be useless.
- We encourage you not to optimize your code prematurely. Simpler solutions tend to work best.
- Do not alter 'memoize' function in "search.py" since it is crucial for performance. Do not modify it when implementing A*.