🏴☠️

Project 2

REPORT PROJECT 2 - TREASURE ISLAND

A. Group members

- **20125049 - Nguyễn Xuân Quang**
- 20125096 - Phan Vĩnh Khang
- 20125113 - Nguyễn Quang Tân
- 20125109 - Trương Kiến Quốc

B. Assignment Plan

No.	Task	Performer	Completion Time
1	Implement game rule, 5 hints (11-15)	Nguyễn Xuân Quang	1 week
2	Implement agent	Nguyễn Xuân Quang	1 week
3	Implement pirate, 5 hints (6-10)	Nguyễn Quang Tân	1 week
4	Make 5 maps, implement 5 hints (1-5)	Trương Kiến Quốc	1 week
5	Implement map generator	Phan Vĩnh Khang	1 week
6	Implement visualization tools	Phan Vĩnh Khang	1 week
7	Report	Whole team	2 days

C. Self-assessment

No.	Criteria	Completion Level
1	Create 5 maps: - 3 maps with sizes: 16x16, 32x32, 64x64 - 2 maps larger than 64x64	100%
1.1	Implement a map generator	100%
2	Implement the game rule	100%
3	Implement the logical agent	100%
4	Implement the visualization tools	100%
6	The agent can handle a complex map with large size, many of regions, prison, mountains	100%
7	Report your implementation with some reflection or comments.	100%
8	Contest*	0%

D. Work

▼ I. Map generator

1. Algorithm description

The map is divided into two halves, an upper half and a lower half. This is done to determine the general location of each region. For example, if the map has 4 regions, they will be split 2-2; if it has 5 regions, they will be split 2-3; if it has 6 regions, they will be split 3-3, and so on (the sea is excluded).

A root is generated at the center of each region, and a breadth-first search (BFS) algorithm is applied to randomly expand the region to its surrounding areas. The sea region is randomly generated from the border of the map, extending inward towards the land surface.

▼ 2. Pseudo code

`generate_mountain`

```
FUCNTION: GENERATE-MOUNTAIN()
OUTPUT: numpy array of cells containing groups of mountains
  # Randomize the number of mountains
  self.num_mountain = random

  # Generate the mountain groups
  self.mountains = []
  for each i in the range from 0 to self.num_mountain do
      # Randomize the size of the mountain
      size = random

      # Generate the mountain cells
      mountain = []
      randomize x, y to a valid coordinate that is inside the land surface
      and assign it to mountain

     for each i in the range from 1 to size do:
          # Mountain can be diagonal adjacent to each other
          direction = list of 8 possible directions
          randomize one among direction and
              assign new_x and new_y to the sum of x, y and random direction
              if new_x and new_y in are inside the map and not on the sea do:
                mountain.append((new_x, new_y))
                break
          ENDWHILE
      ENDFOR

      # Add the mountain to the list
      self.mountains.append(mountain)
```

`generate_prison`

```
FUCNTION: GENERATE-PRISON()
OUTPUT: numpy array of cells containing prisons
  # Randomize the number of prisons
  self.num_prison = random

  # Generate the prison cells
  for each i in the range from 1 to self.num_prison do:
      randomize x, y to a valid coordinate that is inside the land surface
          if x and y in are inside the map and not on the sea and
          not overlapped by mountain do:
            assign that prison location x, y to self.prisons
   ENDFOR
```

`generate_treasure`

```
FUCNTION: GENERATE-TREASURE()
OUTPUT: (x, y) represents treasure location
  # Generate the prison cells
  randomize x, y to a valid coordinate that is inside the land surface
  if x and y in are inside the map and not on the sea and
  not overlapped by mountain and prison do:
    assign that treasure location x, y to self.treasure
```

`generate_map`

```
FUCNTION: GENERATE-MAP()
INPUT: none
OUTPUT: numpy array of map consists of region indicator and tile types

COMPUTE:
# Determine the size of the sea region based on the width of the map
sea_size = determine sea size based on width of map

for each region do:
  # Initialize an empty queue
      queue = []

      # Get the center of the region and the area
      center_x, center_y, area = get_region_center(sea_size, region)
      queue.append((center_x, center_y))

      # Set the center of the region as the starting point
      map[center_x, center_y] = region

      # Use BFS to expand the region
      while queue is not empty:
          x, y = pop first element from queue
          region = map[x, y]
          ENDWHILE

      # Check the adjacent cells and assign them the same region if they are empty
      for each i in the range from -x to area do:
          for each j in the range from -y to area do:
              new_x, new_y = x + i, y + j
              if new_x and new_y in are inside the map and there is the same
                  region cells surround it do:
                  map[new_x, new_y] = region
              ENDFOR
      ENDFOR

      # Generate mountains:
      CALL GENRATE-MOUNTAIN()
      # Generate prisons:
      CALL GENRATE-PRISON()
      # Generate prisons:
      CALL GENRATE-TREASURE()

      # Traverse the whole map to make sure that there is no uncontiguous cells
      # having the same region.
      for each row in the range from 0 to self.height do:
          for each col in the range from 0 to self.width do:
              if surround current cell doesn't have the same region cell do:
                  assign it to the region value of a random adjacency cell
          ENDFOR
      ENDFOR

      # Make sea from the border goes random
      for loop:
        traverse every rows and assign a random number of near-sea cells to 0

      # Initialize the output map
      output_map = []

      # Iterate through the map and add the mountain, prison, and treasure symbols
      for loop:
        traverse every rows and assign each cell to its corresponding region value
        and tile type
        if cell is a mountain do: output_map[] += 'M'
        if cell is a prison do: output_map[] += 'P'
        if cell is a treasure do: output_map[] += 'T'
      ENDFOR
```

3. Explanation

Main function:

The `generate_map()` function is used to generate the map. It takes no input and returns a numpy array representing the map, which consists of region indicators and tile types.

First, the size of the sea region is determined based on the width of the map. Then, for each region, an empty queue is initialized, and the center of the region as well as the area is obtained using the `get_region_center()` function. The region's center is set as the starting point in the map, and a breadth-first search (BFS) algorithm is used to expand the region. This involves checking the adjacent cells and assigning them the same region value if they are empty.

Next, the `generate_mountain()`, `generate_prison()`, and `generate_treasure()` functions are called to generate mountains, prisons, and treasure on the map. The whole map is then traversed to ensure that there are no uncontiguous cells with the same region. If there are, it assigns them the region value of a random adjacency cell.. The border cells are set to region 0 (sea region), and a random number of near-sea cells are set to 0. The resulting map is returned as the output.

Side functions:

The root is located by `get_region_center` , which receives `region` as input parameter, and return the center coordinate according to that `region` .

`generate_prison()` is used to randomly place prisons on the map. The number of prisons is first randomly determined. Then, the function iterates through the number of prisons that are to be placed. For each prison, a pair of random coordinates `x` and `y` is chosen such that they are inside the land surface of the map and are not on the sea or overlapped by a mountain. If these conditions are met, the location `x, y` is added to the list of prison locations `self.prisons`.

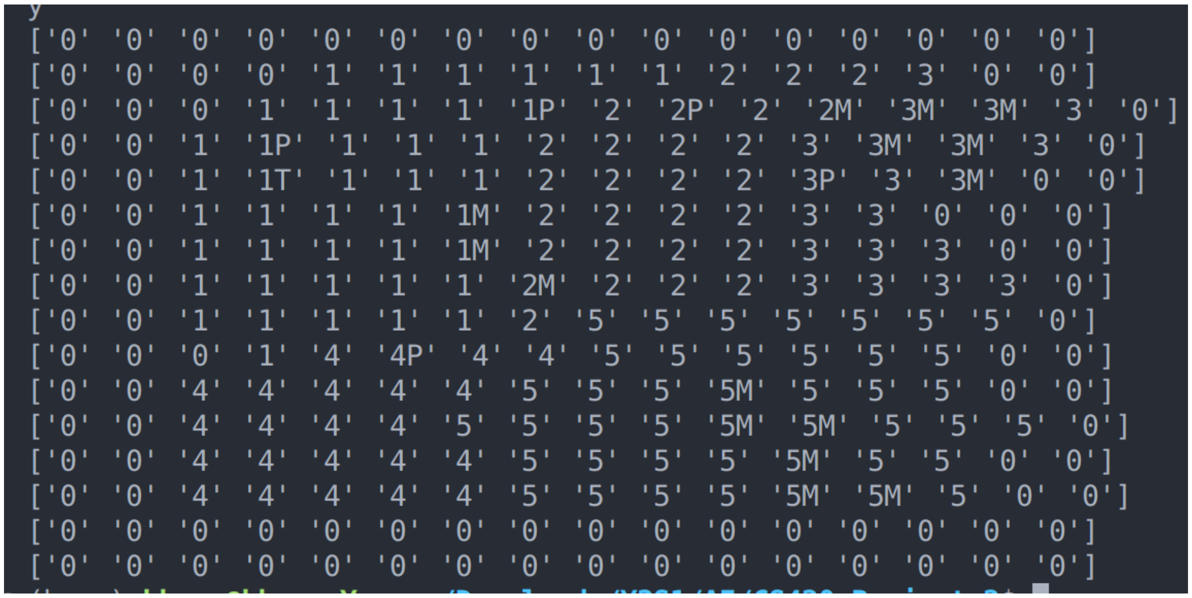
`generate_mountain()` generates a list of mountain groups for a given map. The number of mountain groups is first randomized, and then each group is created iteratively.

For each mountain group, a size is randomly determined, and a starting position is randomly chosen within the map's land surface. The mountain group is then generated by expanding it with additional cells in random directions, ensuring that each cell is within the map's bounds and not on the sea. Once all cells for a mountain group have been chosen, the group is added to the list of mountain groups. The function returns this list of mountains.

▼ 4. Visualization & Test cases

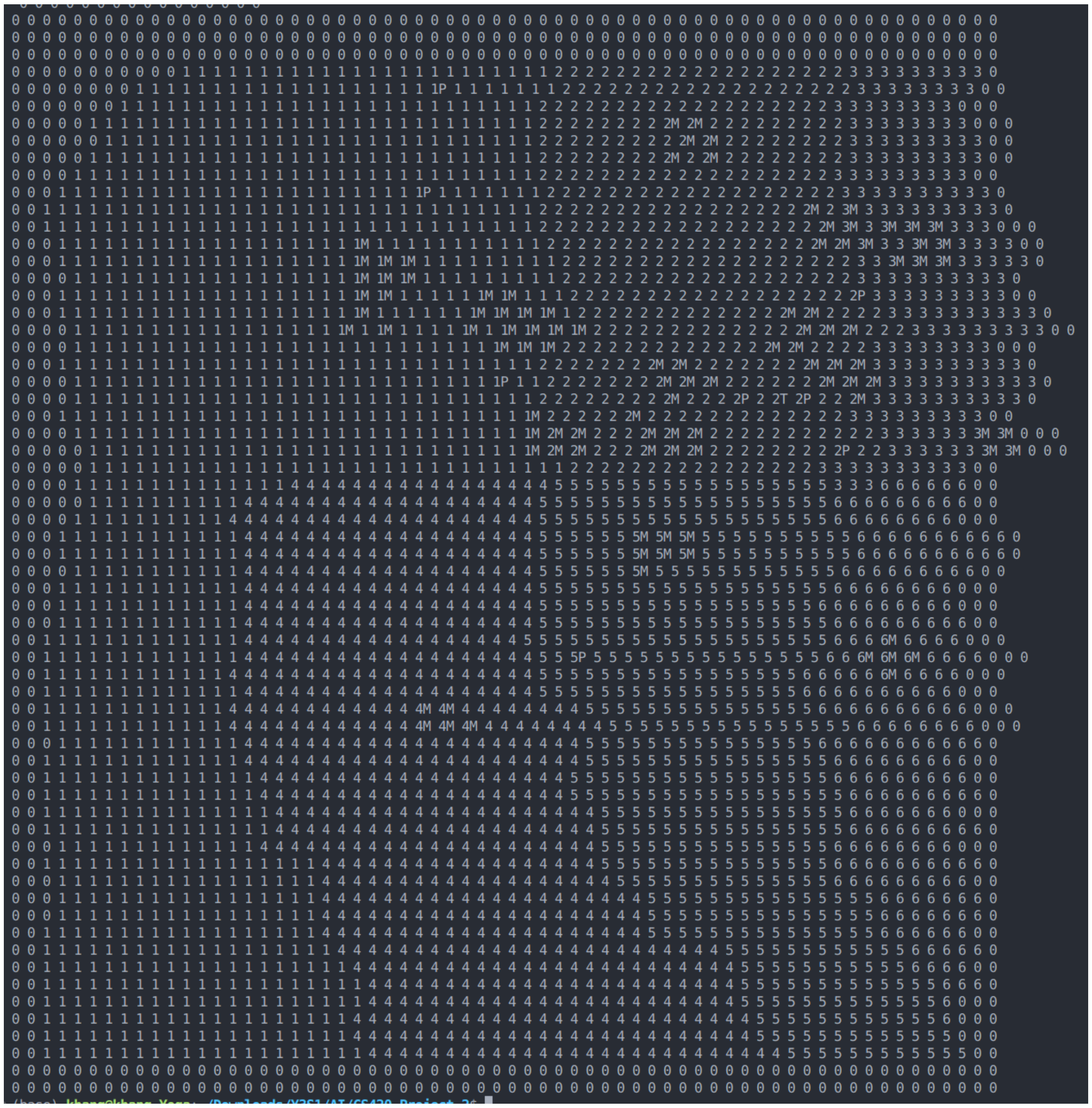
Output examples:

16x16 map



```
 y
 ['0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '1' '1' '1' '1' '1' '1' '2' '2' '2' '3' '0' '0']
 ['0' '0' '0' '1' '1' '1' '1' '1P' '2' '2P' '2' '2M' '3M' '3M' '3' '0']
 ['0' '0' '1' '1P' '1' '1' '1' '2' '2' '2' '2' '3' '3M' '3M' '3' '0']
 ['0' '0' '1' '1T' '1' '1' '1' '2' '2' '2' '2' '3P' '3' '3M' '0' '0']
 ['0' '0' '1' '1' '1' '1' '1M' '2' '2' '2' '2' '3' '3' '0' '0' '0']
 ['0' '0' '1' '1' '1' '1' '1M' '2' '2' '2' '2' '3' '3' '3' '0' '0']
 ['0' '0' '1' '1' '1' '1' '1' '2M' '2' '2' '2' '3' '3' '3' '3' '0']
 ['0' '0' '1' '1' '1' '1' '1' '2' '5' '5' '5' '5' '5' '5' '5' '0']
 ['0' '0' '0' '1' '4' '4P' '4' '4' '5' '5' '5' '5' '5' '5' '0' '0']
 ['0' '0' '4' '4' '4' '4' '4' '5' '5' '5' '5M' '5' '5' '5' '0' '0']
 ['0' '0' '4' '4' '4' '4' '5' '5' '5' '5' '5M' '5M' '5' '5' '5' '0']
 ['0' '0' '4' '4' '4' '4' '4' '5' '5' '5' '5' '5M' '5' '5' '0' '0']
 ['0' '0' '4' '4' '4' '4' '4' '5' '5' '5' '5' '5M' '5M' '5' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0' '0']
```

64x64 map



```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1P 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 0 0
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2M 2M 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2M 2M 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2M 2 2M 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1P 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2M 2 3M 3 3 3 3 3 3 3 3 3 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2M 3M 3 3M 3M 3M 3 3 3 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2M 2M 3M 3 3 3M 3M 3 3 3 3 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1M 1M 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3M 3M 3M 3 3 3 3 3 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1M 1M 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1M 1 1 1 1 1 1M 1M 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2P 3 3 3 3 3 3 3 3 3 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1 1 1 1 1 1 1M 1M 1M 1M 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2M 2M 2 2 2 2 3 3 3 3 3 3 3 3 3 3 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1 1M 1 1 1 1 1M 1 1M 1M 1M 1M 2 2 2 2 2 2 2 2 2 2 2 2 2 2M 2M 2M 2 2 2 3 3 3 3 3 3 3 3 3 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 1M 1M 2 2 2 2 2 2 2 2 2 2 2 2 2 2M 2M 2 2 2 2 3 3 3 3 3 3 3 3 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2M 2M 2 2 2 2 2 2 2 2M 2M 2M 3 3 3 3 3 3 3 3 3 3 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1P 1 1 2 2 2 2 2 2 2 2M 2M 2M 2 2 2 2 2 2 2M 2M 2M 3 3 3 3 3 3 3 3 3 3 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2M 2 2 2 2P 2 2T 2P 2 2 2M 3 3 3 3 3 3 3 3 3 3 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 2 2 2 2 2 2M 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 2M 2M 2 2 2 2M 2M 2M 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3M 3M 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1M 2M 2M 2 2 2 2M 2M 2M 2 2 2 2 2 2 2 2 2P 2 2 3 3 3 3 3 3 3M 3M 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 3 3 3 6 6 6 6 6 6 6 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5M 5M 5M 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5M 5M 5M 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5M 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6M 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5P 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6M 6M 6M 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6M 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4M 4M 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4M 4M 4M 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 6 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(base) khang@khang-Yoga:~/Downloads/Y3S1/AI/CS420-Project-2$
```

II. Map visualization

1. Description

The user interface (UI) for the game was implemented using the Tkinter library. Customized libraries, such as `sv_ttk` and `customtkinter`, were used to create the main grids for displaying the map and side information, including region labels, game logs, and notes. A button was also implemented for user interaction, allowing for navigation between game states. The following classes were implemented in order to create the UI:

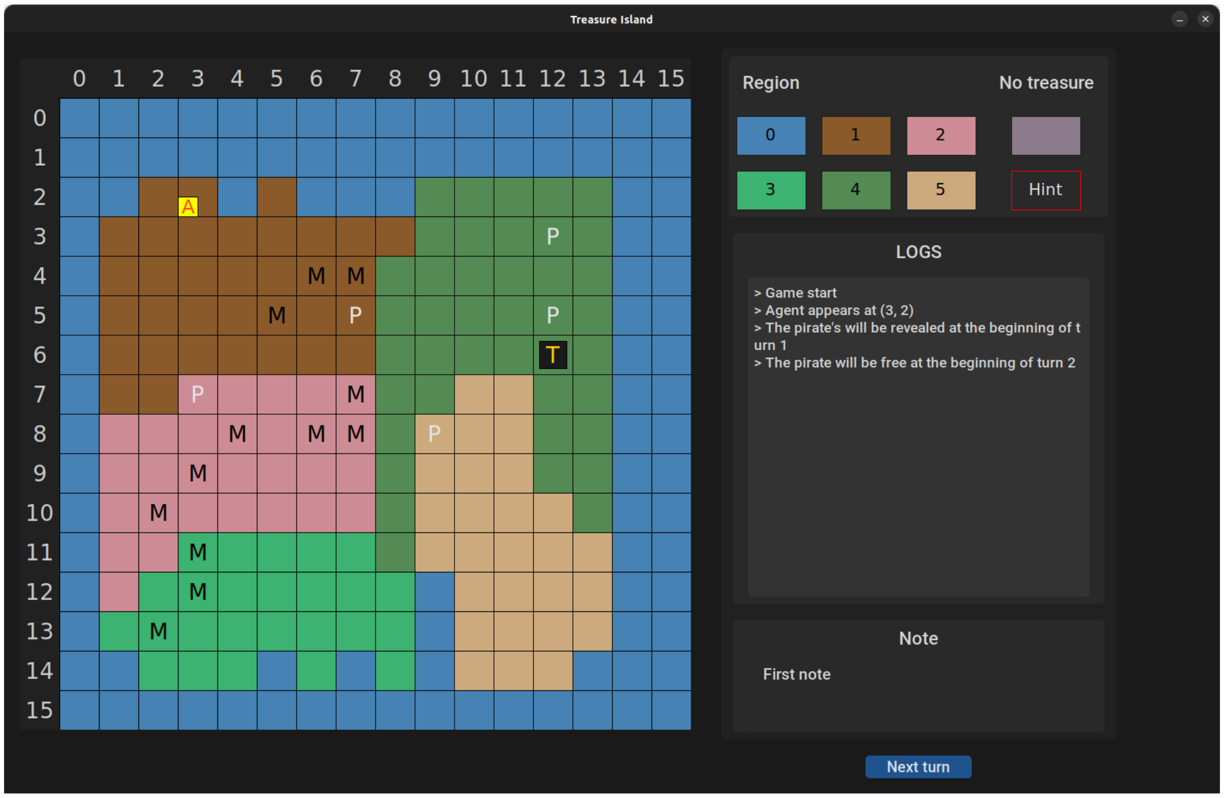
`App`: The root application, responsible for managing the main grids, components, and button.
`SideInformation`: This class handles the right half of the window and manages the displays for logs, regions, and notes.
`MapDisplay`: This class is responsible for displaying the map, as well as the movement of the agent and pirate.
`LogDisplay`: This class displays the logs, which are passed to the insert_log function as a string of content from the agent and game rule.
`NoteDisplay`: This class displays the notes, which are passed to the insert_note function as a string of content from the game

rule.

`RegionDisplay`: This class displays the region labels and the non-treasure area label.

2. Visualization

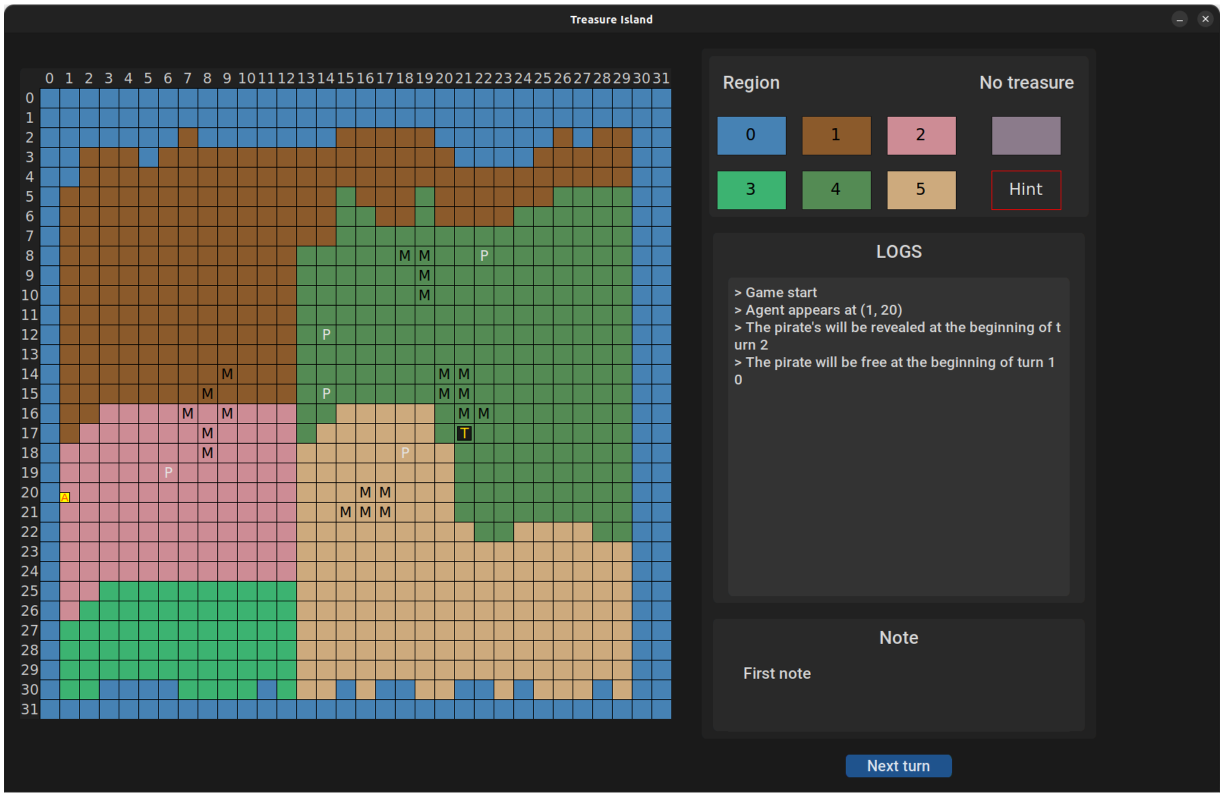
16x16 map



```
> Game start
> Agent appears at (3, 2)
> The pirate's will be revealed at the beginning of t
urn 1
> The pirate will be free at the beginning of turn 2
```

First note

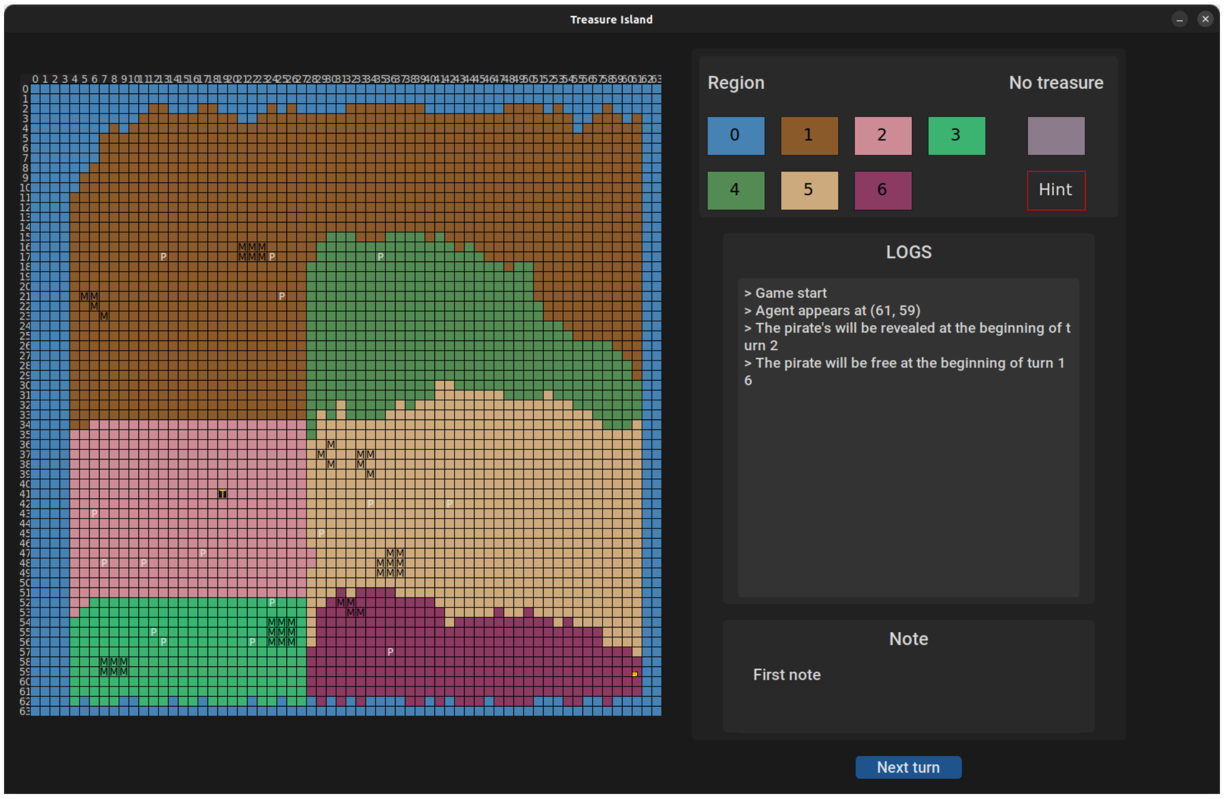
32x32 map



```
> Game start
> Agent appears at (1, 20)
> The pirate's will be revealed at the beginning of t
urn 2
> The pirate will be free at the beginning of turn 1
0
```

First note

64x64 map



```
> Game start
> Agent appears at (61, 59)
> The pirate's will be revealed at the beginning of t
urn 2
> The pirate will be free at the beginning of turn 1
6
```

First note

III. Game rule

1. Algorithm description and Explanation

This code is for the game mechanics. It handles the turns of the game and the actions taken by the agent and the pirate.

On each turn, it checks if the current turn is the turn when the pirate should be revealed or when the pirate should be freed. If it is, it updates the hint weights and the state of the pirate accordingly.

It then generates a hint using the hint manager and adds it to the list of hints the agent has received. If it is the first turn, it sets the first hint as verified and truth.

Next, the code allows the agent to make up to 2 moves. If the agent knows the location of the treasure, it either teleports to the treasure and takes a large scan, takes a large scan at its current location, or moves one step closer to the treasure and takes a small scan. If the agent does not know the location of the treasure, it either takes a small scan at its current location or moves to a random location and takes a small scan.

After each move, the code checks if the agent has found the treasure. If the agent has found the treasure, the game ends and the loop is broken. If the agent has not found the treasure, the loop continues for another move (if there are still moves left) or ends the turn.

2. Pseudo code

```
FUNCTION  run():
        RETURNS // NONE
        // Init pirate pos here because it must be after the init of map (and prison)
        IF self.pirate.cur_pos is None
            THEN
            self.pirate.set_pos(self.map_manager.prisons[random.randint(
                0, self.map_manager.num_prisons - 1)])
            self.pirate_prev_pos<-self.pirate.cur_pos
        ENDIF
    ENDFUNCTION

        IF not self.is_win
            THEN
            self.turn_idx += 1
            OUTPUT 'TURN: {}'.format(self.turn_idx
            self.logs.put('START TURN {}'.format(self.turn_idx))
        ENDIF

            IF self.turn_idx = self.prison_revealTurn
                THEN
                self.logs.put(
                    "The Pirate is at the prison {}".format(self.pirate.cur_pos))
                self.hint_weights<-np.array(
                    [1, 1, 1, 1, 1, 0, 0.5, 1, 1, 1, 1, 0.5, 1, 0.5, 1])
            ENDIF

            IF self.turn_idx = self.pirate_freeTurn
                THEN
                self.logs.put("The pirate is free")
                self.hint_weights<-np.array(
                    [1, 1, 1, 1, 1, 1, 0.5, 1, 1, 1, 1, 0.5, 1, 0.5, 1])
                self.pirate_isFree<-True
                self.pirate.find_shortest_path()
            ENDIF

            IF self.turn_idx = 1
                THEN
                hint_type, log, truth, data<-self.hint_manager.gen_first_hint(
                    self.agent.cur_pos, self.pirate.cur_pos, self.hint_weights)
            ELSE
                hint_type, log, truth, data<-self.hint_manager.generate(
                    self.agent.cur_pos, self.pirate.cur_pos, self.hint_weights)
            ENDIF
            self.truth_list.append(truth)
            array_of_tiles, binary_mask<-self.agent.refactor_hint(
                hint_type, data)

            IF array_of_tiles is not None
                THEN
                self.hint_tiles.put(array_of_tiles)
            ENDIF
            self.agent.add_hint(self.turn_idx, hint_type, binary_mask)

            self.logs.put('HINT {}: {}'.format(self.turn_idx, log))
            self.logs.put(f"ADD HINT {self.turn_idx} TO HINT LIST")
            IF self.turn_idx = 1
                THEN
                self.logs.put('HINT 1: is_verified<-TRUE, is_truth<-TRUE')
                mask<-self.agent.hints[0][2]
                self.agent.verify(0, True, mask)
            ENDIF

            step<-0
            WHILE step < 2 and not self.is_win DO
                IF self.known_treasure
                    THEN
                    // IF tele is still available, agent tele to the treasure and takes a large scan
                    THEN
                    ENDIF
                    IF self.can_tele
                        THEN
                        self.can_tele<-False
                        self.agent.teleport(self.map_manager.treasure_pos)
                        self.logs.put(
                            'Agent teleport to position: {}'.format(self.agent.cur_pos))
                        self.logs.put(
                            'Agent takes a large scan at {}'.format(self.agent.cur_pos))
                        self.is_win<-True
                    ENDIF
                ENDIF
            ENDWHILE

                    // IF there is no tile in path, agent arrives the target, just takes a large scan and return true
                    THEN
                    ENDIF
                    IF len(self.agent.path) = 0 and not self.is_win
                        THEN
                        self.logs.put(
                            'Agent takes a large scan at {}'.format(self.agent.cur_pos))
                        self.is_win<-True
                    ENDIF

                    // Else, agent move the next step to the treasure
                    next_pos<-self.agent.path.pop(0)
                    IF cal_manhattan_distance(self.agent.cur_pos, next_pos) <= 2
                        THEN
                        // Small move
                        self.logs.put(
                            'Agent moves from {} to tile {} and takes a small scan'.format(self.agent.cur_pos, next_pos))
                        self.agent.update_pos(next_pos)
                        self.is_win<-self.agent.small_scan()
                    ELSE
                        // Large move
                        self.logs.put(
                            f"Agent moves from {self.agent.cur_pos} to tile {next_pos}")
                        self.agent.update_pos(next_pos)
                    ENDIF
                    step += 1

                ELSE
                    // Treasure position is unknown
                    // Get action of the turn
                    next_action<-self.agent.get_action(
                        self.pirate_isFree, self.can_tele, self.pirate.cur_pos, self.pirate_prev_pos)
                    IF next_action[1] = 0
                        THEN
                        // Verification
                        (idx, turn, mask)<-next_action[2]
                        truth<-self.truth_list[turn-1]
                        self.agent.verify(idx, truth, mask)
                        self.logs.put(
                            f"Agent has verified the hint {turn}, it is {truth}!!")
                        step += 1
                        self.logs.put(
                            'HINT {}: is_verified<-TRUE, is_truth<-{}'.format(turn, truth))
                    ENDIF

                    elif next_action[1] = 1:  //You will need to change this to CASE OF
                        // Move 1-2 tiles and small scan
                        next_move<-next_action[2]
                        prev_pos<-self.agent.cur_pos[:]
                        self.agent.move(next_move)
                        self.logs.put('Agent moves a small steps from {} to {} and takes a small scan'.format(
                            prev_pos, self.agent.cur_pos))
                        has_treasure<-self.agent.small_scan()
                        IF has_treasure
                            THEN
                            self.logs.put('WIN')
                            self.is_win<-True
                        ENDIF
                        step += 1

                    elif next_action[1] = 2:  //You will need to change this to CASE OF
                        // Move 3-4 tiles
                        next_move<-next_action[2]
                        prev_pos<-self.agent.cur_pos
                        self.agent.move(next_move)
                        self.logs.put('Agent moves a large steps from {} to {}'.format(
                            prev_pos, self.agent.cur_pos))
                        step += 1

                    elif next_action[1] = 3:  //You will need to change this to CASE OF
                        // Large scan 5x5
                        has_treasure<-self.agent.large_scan()
                        self.logs.put('Agent takes a large scan')
                        IF has_treasure
                            THEN
                            self.logs.put('WIN')
                            self.is_win<-True
                        ENDIF
                        step += 1

                    elif next_action[1] = 4:  //You will need to change this to CASE OF
                        // Teleport
                        pos<-next_action[2]
                        self.agent.teleport(pos)
                        self.can_tele<-False
                        self.logs.put(
                            "Agent teleports to tiles {}".format(pos))
                        // this action is not counted as a step

                    IF np.count_nonzero(self.agent.knowledge_map) = 1
                        THEN
                        // If there is only 1 tile available, it's the treasure
                        self.known_treasure<-True
                        self.agent.bfs_fastest_path()
                        // pop the current position of agent
                        self.agent.path.pop(0)
                    ENDIF

            IF self.turn_idx >= self.pirate_freeTurn
                THEN
                // OUTPUT 'Pirate cur pos: {}'.format(self.pirate.cur_pos
                // OUTPUT 'Pirate prev pos: {}'.format(self.pirate_prev_pos
                self.pirate_prev_pos<-self.pirate.cur_pos
                IF not self.pirate.path.empty()
                    THEN
                    (move, log)<-self.pirate.path.get()
                    self.logs.put(log)
                    self.pirate.set_pos(move.tolist() IF isinstance(move, np.ndarray) else list(move))
                    THEN
                    ENDIF
                ENDIF
            ENDIF

            IF self.pirate.reach_treasure()
                THEN
                self.logs.put('LOSE')
                self.is_lose<-True
            ENDIF
```

▼ IV. Agent

1. Algorithm description:

The agent can perform several actions, including verifying a hint, staying in place and taking a 5x5 scan, making a small move and taking a 3x3 scan, or making a large move without taking a scan. These estimated values are calculated based on certain

conditions and heuristic i.e. based on the agent's current position and the availability of certain resources (e.g. teleporting ability, knowledge base). The function `get_best_action` returns the best action for the agent to take based on the agent's current knowledge of the map represented as a binary map. The function `cal_heuristic` calculates a score for a given position on the map based on how many tiles in a given area around the position are known. The function `get_action` returns the best action for the agent to take based on whether or not the pirate (a rival player) is free and the availability of the agent's teleporting ability. The function `bfs_fastest_path`
 finds the shortest path to the treasure using breadth first search. The function `cal_manhattan_distance` calculates the Manhattan distance between two positions on the map.

▼ 2. Pseudo code

`refactor_hint`

```
FUNCTION  refactor_hint(self, hint_type, data):
        RETURNS // the array of tiles represents hints, the binary mask that
                  represents tiles with and without treasure in the map.
        array_of_tiles<-None
        binary_mask<-None
        IF hint_type = 1
            THEN
            // Hint type 1, data is an array of tiles do not contain treasure
            binary_mask<-arrayTiles_to_binaryMask(
                data, self.width, self.height, flip=True)
        ENDIF
ENDFUNCTION

        elif hint_type in [2, 15]:  //You will need to change this to CASE OF
            // Hint type 2 and 15, data is list of regions
            binary_mask<-np.isin(self.map_manager.map, data)

        elif hint_type = 3:  //You will need to change this to CASE OF
            // Hint type 3, data is list regions, which do not contain treasure
            binary_mask<-np.logical_not(np.isin(self.map_manager.map, data))

        elif hint_type = 4:  //You will need to change this to CASE OF
            // Hint type 4, data is an array of tiles
            binary_mask<-arrayTiles_to_binaryMask(
                data, self.width, self.height, flip=False)

        elif hint_type = 5:  //You will need to change this to CASE OF
            // Hint type 5, data is a rectangle does not contain treasure
            binary_mask<-arrayTiles_to_binaryMask(
                data, self.width, self.height, flip=True)

        elif hint_type = 6:  //You will need to change this to CASE OF
            pass

        elif hint_type = 7:  //You will need to change this to CASE OF
            // Hint type 7, data is column or/and row contain treasure
            col, row<-data
            binary_mask<-np.zeros((self.width, self.height), dtype=bool)
            IF col
                THEN
                binary_mask[col,]<-1
            ENDIF
            IF row
                THEN
                binary_mask[:, row]<-1
            ENDIF

        elif hint_type = 8:  //You will need to change this to CASE OF
            // Hint type 8, data is column/row do not contain treasure
            col, row<-data
            binary_mask<-np.ones((self.width, self.height), dtype=bool)
            IF col
                THEN
                binary_mask[col,]<-0
            ENDIF
            IF row
                THEN
                binary_mask[:, row]<-0
            ENDIF

        elif hint_type = 9:  //You will need to change this to CASE OF
            // Hint type 9, data is indices of 2 regions
            rid_1, rid_2<-data
            bound_1, bound_2<-self.map_manager.get_two_regions_boundary(
                rid_1, rid_2)
            array_of_tiles<-np.concatenate((bound_1, bound_2), axis=0)
            binary_mask<-arrayTiles_to_binaryMask(
                array_of_tiles, self.width, self.height, flip=False)

        elif hint_type = 10:  //You will need to change this to CASE OF
            // Hint type 10, data is None
            array_of_tiles, binary_mask<-self.map_manager.get_all_boundaries()

        elif hint_type = 11:  //You will need to change this to CASE OF
            // Hint type 11, data is a binary map represents tiles with distance from sea
            binary_mask<-data

        elif hint_type = 12:  //You will need to change this to CASE OF
            // Hint type 12, data is a rectangle noted by top_left and bot_right
            top_left, bot_right<-data
            binary_mask<-np.ones((self.width, self.height), dtype=bool)
            binary_mask[top_left[0]:bot_right[0],
                        top_left[1]:bot_right[1]]<-False

        elif hint_type = 13:  //You will need to change this to CASE OF
            binary_mask<-data

        elif hint_type = 14:  //You will need to change this to CASE OF
            outer_rec, inner_rec<-data
            col_0, row_0, col_1, row_1<-outer_rec
            col_2, row_2, col_3, row_3<-inner_rec
            binary_mask<-np.zeros((self.width, self.height), dtype=bool)
            binary_mask[col_0:(col_1+1), row_0:(row_1+1)]<-True       // outer
            binary_mask[col_2:(col_3+1), row_2:(row_3+1)]<-False      // inner

        IF hint_type in [1, 4, 5]
            THEN
            array_of_tiles<-data
        ENDIF
        elif hint_type in [2, 7, 11, 13, 14, 15]:  //You will need to change this to CASE OF
            x, y = np.where(binary_mask = 1)
            array_of_tiles<-np.vstack((x, y)).T
        elif hint_type in [3, 8, 12]:  //You will need to change this to CASE OF
            x, y = np.where(binary_mask = 0)
            array_of_tiles<-np.vstack((x, y)).T

        return array_of_tiles, binary_mask  // turn_idx, hint_type, mask
```

`get_best_action`

```
FUNCTION  get_best_action(self, knowledge):
        RETURNS // the best possible move
        actions<-[]
        movable<-np.ones(4, dtype=bool)
        num_available<-np.count_nonzero(self.knowledge_map)
ENDFUNCTION

        // Estimate verify action
        FOR i in range(len(self.hints)):
            IF self.hints[i][1] = 6
                THEN
                continue
            ENDIF
            turn, hint_type, binary_mask<-self.hints[i]
            // turn, hint_type, binary_mask<-self.refactor_hint_data(i)
            temp<-np.logical_and(binary_mask, self.knowledge_map)
            count<-np.count_nonzero(temp)
            heapq.heappush(
                actions, (count, 0, (i, turn, binary_mask)))

        // Estimate stay & 5x5 scan
        heuristic<-self.cal_heuristic(self.cur_pos, size=5)
        heapq.heappush(actions, (heuristic, 3, (0, 0)))

        // Estimate small move & 3x3 scan
        temp<-999999
        selected_move<-None
        count<-0
        FOR i, move in enumerate([(1, 0), (-1, 0), (0, 1), (0, -1), (2, 0), (-2, 0), (0, 2), (0, -2)]):  //Pseudocode can't handle t
            x<-self.cur_pos[0] + move[0]
            y<-self.cur_pos[1] + move[1]
            IF x < 0 or x >= self.width or y < 0 or y >= self.height or movable[i % 4] = False
                THEN
                continue
            ENDIF
            IF self.map_manager.is_sea((x, y)) or self.map_manager.is_mountain((x, y))
                THEN
                movable[i % 4]<-False
                continue
            ENDIF

            dist_2_mean<-self.cal_dist_to_mean((x, y), knowledge)
            heuristic<-self.cal_heuristic((x, y), size=3)
            IF temp > dist_2_mean - heuristic
                THEN
                temp<-dist_2_mean - heuristic
                selected_move<-move
                count<-heuristic
            ENDIF
        IF selected_move
            THEN
            heapq.heappush(
                actions, (count + min(num_available+1, 6), 1, selected_move))
        ENDIF

        // Estimate large move without scan
        selected_move<-None
        FOR i, move in enumerate([(3, 0), (-3, 0), (0, 3), (0, -3), (4, 0), (-4, 0), (0, 4), (0, -4)]):  //Pseudocode can't handle t
            x<-self.cur_pos[0] + move[0]
            y<-self.cur_pos[1] + move[1]
            IF x < 0 or x >= self.width or y < 0 or y >= self.height or movable[i % 4] = False
                THEN
                continue
            ENDIF
            IF self.map_manager.is_sea((x, y)) or self.map_manager.is_mountain((x, y))
                THEN
                movable[i % 4]<-False
                continue
            ENDIF
            dist_2_mean<-self.cal_dist_to_mean((x, y), knowledge)
            IF temp > dist_2_mean
                THEN
                temp<-dist_2_mean
                selected_move<-move
            ENDIF
        IF selected_move
            THEN
            heapq.heappush(
                actions, (min(num_available+2, 8), 2, selected_move))
        ENDIF

        return actions[-1]
```

`phase_2_action`

```
FUNCTION  phase_2_action(self, pirate_cur_pos, pirate_prev_pos):
        RETURNS // best possible action
        x<-pirate_cur_pos[0] - pirate_prev_pos[0]
        y<-pirate_cur_pos[1] - pirate_prev_pos[1]
ENDFUNCTION

        knowledge_clone<-np.copy(self.knowledge_map)
        IF y = 0 and x in [1, 2]
            THEN
            // Pirate moved right
            knowledge_clone[:pirate_cur_pos[0],]<-0
        ENDIF

        elif y = 0 and x in [-1, -2]:  //You will need to change this to CASE OF
            // Pirate moved left
            knowledge_clone[(pirate_cur_pos[0]+1):,]<-0

        elif x = 0 and y in [1, 2]:  //You will need to change this to CASE OF
            // Pirate moved down
            knowledge_clone[:, :pirate_cur_pos[1]]<-0
        elif x = 0 and y in [-1, -2]:  //You will need to change this to CASE OF
            // Pirate moved up
            knowledge_clone[:, (pirate_cur_pos[1]+1):]<-0

        elif x = 1 and y = 1:  //You will need to change this to CASE OF
            // Pirate moved down right
            knowledge_clone[:pirate_cur_pos[0],]<-0
            knowledge_clone[:, :pirate_cur_pos[1]]<-0

        elif x = 1 and y = -1:  //You will need to change this to CASE OF
            // Pirate moved up right
            knowledge_clone[:pirate_cur_pos[0],]<-0
            knowledge_clone[:, (pirate_cur_pos[1]+1):]<-0

        elif x = -1 and y = 1:  //You will need to change this to CASE OF
            // Pirate moved down left
            knowledge_clone[(pirate_cur_pos[0]+1):,]<-0
            knowledge_clone[:, :pirate_cur_pos[1]]<-0

        ELSE
            // Pirate moved up left
            knowledge_clone[(pirate_cur_pos[0]+1):, ]<-0
            knowledge_clone[:, (pirate_cur_pos[1]+1):]<-0
```

`bfs_fatest_path`

```
FUNCTION  bfs_fastest_path(self):
        RETURNS // None or the best path
        paths<-[[self.cur_pos]]
        visited<-[self.cur_pos]
ENDFUNCTION

        WHILE self.paths DO
            path<-self.paths.pop(0)
            IF path[-1] = self.map_manager.treasure_pos
                THEN
                self.path<-path
                return self.path
            ENDIF
        ENDWHILE

            movable<-np.ones(4, dtype=bool)
            list_adj<-np.array(
                [(-1, 0), (1, 0), (0, -1), (0, 1), (-2, 0), (2, 0), (0, -2), (0, 2), (-3, 0), (3, 0), (0, -3), (0, 3), (-4, 0), (4,
            FOR i, move in enumerate(list_adj):  //Pseudocode can't handle this
                x<-path[-1][0] + move[0]
                y<-path[-1][1] + move[1]
                IF x < 0 or x >= self.width or y < 0 or y >= self.height
                    THEN
                    continue
                ENDIF
                IF (x, y) in visited
                    THEN
                    continue
                ENDIF
                IF self.map_manager.is_sea((x, y)) or self.map_manager.is_mountain((x, y)) or not movable[i % 4]
                    THEN
                    movable[i % 4]<-False
                    continue
                ENDIF
                paths.append(path + [(x, y)])
                visited.append((x, y))
        return None
```

`cal_heuristic`

```
FUNCTION  cal_heuristic(self, pos, size=3):
        RETURNS // h (heuristic value)
        IF size = 3
            THEN
            col_0, row_0<-max(pos[0]-1, 0), max(pos[1]-1, 0)
            col_1, row_1<-min(pos[0]+1, self.width -
                               1), min(pos[1]+1, self.height)
        ELSE
            col_0, row_0<-max(pos[0]-2, 0), max(pos[1]-2, 0)
            col_1, row_1<-min(pos[0]+2, self.width -
                                1), min(pos[1]+2, self.height)
        ENDIF
        region<-self.knowledge_map[col_0:(col_1+1), row_0:(row_1+1)]
        h<-np.count_nonzero(region)
        return h
ENDFUNCTION
```

3. Explanation:

The `get_best_action` function determines the best action for the agent to take given its current knowledge of the game map. The function first estimates the value of verifying each of the hints the agent has received. It then estimates the value of staying in place and taking a large scan, as well as the value of making a small move and taking a small scan. Finally, it estimates the value of making a large move without a scan. The function returns the action with the highest estimated value.

The `cal_heuristic` function calculates a heuristic value for a given position on the map. This value represents the estimated number of tiles that the agent will learn about if it takes a scan of a certain size (either 3x3 or 5x5) at that position.

The `get_action` function determines the action that the agent should take on the current turn. If the pirate is not yet free, the function returns the best action determined by `get_best_action`. If the pirate is free, the function first checks if the agent has any teleportation available. If it does, the function returns a teleportation action to the current position of the pirate. If the pirate has not moved from its previous position, the function returns a large scan action. Otherwise, the function returns an action determined by the `phase_2_action` function.

The `bfs_fastest_path` function uses a breadth-first search algorithm to find the shortest path for the agent to take to reach the treasure. The `cal_manhattan_distance` function calculates the Manhattan distance between two positions on the map. This is simply the sum

V. Hint

We create the class `HintManager` to take responsibility of generating a hint at each turn. It has 15 method `gen_hint_x` $(1 \leq x \leq 15)$ to generate 15 types of hint which are suggested by teachers, we do not add or define any additional hint, the detail of those 15 method for 15 hints would be provided later. The `HintManager` contains the method generate which will randomly choose one of 15 types of hint and generate the hint at each turn. It also contains the method called `gen_first_hint` in which we do the same things as genarate but we will generate the hint until we get the truth of that hint is True for the first turn since the game requires to give the agent a true hint at the first turn.

We implement `HintManager` with Numpy along with a “vectorize” coding style to take advantages of the fast speed of Numpy array, therefore, we could optimize the whole game’s operations.

1. The 1st typed hint

This hint stated that “A list of random tiles that don't contain the treasure (1 to 12).”

We implement this hint by first random how many tiles from 1-12 should we give as a hint. Then we random the corresponding number of tiles by random the ordinal number of tiles from `0` to `width * height - 1` and convert its to the corresponding coordinate (we need to do this to utilize the speed of numpy array).

This function returns `(1, log, truth, array_of_tiles)` where 1 is the type of hint, `log` is for visualization for example “A list of tiles [[1, 2], [3, 6]] doesn't contain the treasure”, `truth` is True if the treasure is not in the list of the tiles that have been chosen and False otherwise, `array_of_tiles` is the list of the chosen tiles’ coordinates.

2. The 2nd typed hint

This hint stated that “2-5 regions that 1 of them has the treasure.”

We implement this hint by first random how many regions from 2-5 should we give as a hint. Then we randomize the corresponding number of regions. After that, we check if the treasure position belongs to the regions that are chosen.

This function returns `(2, log, truth, list_regions)` where 2 is the type of hint, `log` is for visualization for example “One of the regions 1, 4, 5 has the treasure”, `truth` is True if the treasure is in one of the chosen regions and False otherwise, `list_regions` contains the list of regions that are chosen.

3. The 3rd typed hint

This hint stated that “1-3 regions that do not contain the treasure.”

This hint is kinda similar to the 2nd typed hint, it is just different from the number of regions is from 1 to 3 and this hint is the opposite case of the 2nd typed hint.

This function returns `(3, log, truth, list_regions)`

4. The 4th typed hint

This hint stated that “A large rectangle area that has the treasure.”

We implement this hint by first getting a rectangle area with its top left and bottom right coordinates. We do that by random 2 `x` coordinates and 2 `y` coordinates from `0 - width - 1` and `0 - height - 1` respectively and sort them ascendingly. It can be seen that the top left coordinate would be the 2 biggest `x` and `y` coordiantes while the bottom right is the smallest. We must also ensure that a large rectangle is a rectangle that has area from 50% - smaller than 70% of the map’s area.

Then, we check if the treasure is in the chosen rectangle area by check if the treasure’s `x` and `y` coordinates are between the range of 2 chosen `x` and `y` coordinates respectively.

This function returns `(4, log, truth, rectangle)` where 4 is the type of hint, `log` is for visualization for example “A large rectangle area has the treasure. Top-Left-Bottom-Right = [1, 3, 15, 13]”, `truth` is True if the treasure is lie in the chosen rectangle and False otherwise, `rectangle` contains the `top left` and `bottom right` vertices of the chosen rectangle.

5. The 5th typed hint

This hint stated that “A small rectangle area that doesn't have the treasure.”

This hint is kinda similar to the *4th* typed hint, it is just different from constraint that the area of the rectangle in this hint must be from 20% - smaller than 50% of the map’s area.

This function returns `(5, log, truth, rectangle)`

6. The 6th typed hint

This hint stated that “He tells you that you are the nearest person to the treasure (between you and the prison he is staying).”

We implement this hint by first calculate the manhattan distance between treasure postition and agent position as well as between treasure position and pirate position. Then we check if the manhattan distance of the agent to treasure is smaller than the pirate’s.

This function returns `(6, log, truth, None)` where 6 is the type of hint, `log` is for visualization for example “The agent is nearer than the pirate to the treasure”, `truth` is True if the agent is nearer than the pirate to the treasure based on manhattan distance and False otherwise, `None` is a None typed object.

7. The 7th typed hint

This hint stated that “A column and/or a row that contain the treasure.”

We implement this hint by first random whether to give the agent a column or a row or both column and row. The chance for getting both column and row hint is 10% while getting only column or row is 45% each. After that, we just simply choose randomly which column or row or both and then check if they contain the treasure.

Finally, we return a tuple of `(7, log, truth, col_row)` where 7 is the type of hint, `log` is for visualization for example “Column 6 and row 5 contain the treasure”, `truth` is True if col or row contain the treasure and False otherwise, `col_row` is the ordinal number of the col or row or both that have been chosen.

8. The 8th typed hint

This hint stated that “A column and/or a row that do not contain the treasure.”

This hint is kinda similar to the 7th typed hint, it is just the opposite case.

This function returns a tuple of `(8, log, truth, col_row)`

9. The 9th typed hint

This hint stated that “2 regions that the treasure is somewhere in their boundary.”

We implement this hint by first random the first region to include in this hint. After that, we get a list of all the neighbored regions of the first region and the we just need to choose randomly a second region from that list to complete giving this hint to the agent.

Finally, we return a tuple of `(9, log, truth, region1_region2)` where 9 is the type of hint, log is for visualization for example “Treasure is somewhere in the boundary of regions 2 and 5”, truth is True if the treasure is somewhere in their boundary and False otherwise, region1_region2 is the ordinal number of the 2 regions that have been chosen.

10. The 10th typed hint

This hint stated that “The treasure is somewhere in a boundary of 2 regions.”

We implement this hint by checking the boundaries of all pairs of 2 adjacent if one of them contains the treasure. We check it by getting the treasure position and then move 1 step in 4 directions up, down, left, right to see if the position that we moved to is belonged to another region.

This function returns a tuple of `(10, log, truth, None)` where 10 is the type of hint, `log` is for visualization for example “The treasure is somewhere in a boundary of 2 regions”, `truth` is True if the treasure is somewhere in a boundary of 2 regions and False otherwise, `None` means there are no data provided.

11. The 11th typed hint

This hint stated that “The treasure is somewhere in an area bounded by 2 - 3 tiles from sea.”

We implement this hint by first random a number which decides how many tiles should bound the sea. After that, we create a bitmask of the whole map. In that bitmask, tiles that are sea be 0 and the other regions would be 1. Then we move that bit mask 2, 3 tiles up, down, left and right, we would get a new bitmask in which 0 is the sea and the area bounded by 2 - 3 tiles from sea, 1 is the value of the other regions. Finally, we map the bitmask one more time to get rid of the tiles which are sea.

This function returns a tuple of `(11, log, truth, bitmask)` where 11 is the type of hint, `log` is for visualization for example “The treasure is somewhere in an area bounded by 3 tiles from sea”, `truth` is True if the treasure is in an area bounded by 2 - 3 tiles from sea and False otherwise, `bitmask` has the same shape as the array of map, in which tile would be 1 if the tile is in an area bounded by 2 - 3 tiles from sea (not included sea) and 0 for the others.

12. The 12th typed hint

This hint stated that “A half of the map without treasure.”

We implement this hint by choose random randomly which half of the map should be returned which are top half, bottom half, left half and right half and then simply return the top left and bottom right coordiantes of the half of the map.

This function returns a tuple of `(12, log, truth, top_left_bottom_right)` where *12* is the type of hint, `log` is for visualization for example “The top half of the map does not contains treasure”, `truth` is True if the treasure is not in the half that has been chosen and False otherwise, `top_left_bottom_right` contains the top left and bottom right coordinates of the chosen half.

13. The 13th typed hint

This hint stated that “From the position that he's staying, he tells you a direction that has the treasure (W, E, N, S or SE, SW, NE, NW)” (The shape of area when the hints are either W, E, N or S is triangle)

We decide to return the rectangle area if the direction is SE, SW, NE, NW and from the position of the pirate instead of the center of the map. So we implement this hint by first random which direction we should suggest for the agent. For the SE, SW, NE, NW directions, since they return a rectangle area, it is kinda easy to implement, for example the direction SE would return all tiles whose $x > pirate_x$ and $y > pirate_y$ .The W, E, N, S directions is more complicated but overall we just need to use the first and second diagonal.

This function returns a tuple of `(13, log, truth, mask)` where 13 is the type of hint, `log` is for visualization for example “Treasure is in the South of the pirate's position [1, 2]”, `truth` is True if the treasure in the area of the direction returned and False otherwise, `mask` is a bitmask with the map’s shape in which 1 is the area of the direction returned and 0 for the others.

14. The 14th typed hint

This hint stated that “2 squares that are different in size, the small one is placed inside the bigger one, the treasure is somewhere inside the gap between 2 square.”

We first choose randomly 2 big and small rectangles by getting the top left and bottom right coordinates of big and small rectangles. We do that by random 4 x coordinate and 4 y coordinate from a sequence from `0 - map_width - 1` and `0 - map_height -1` respectively with replacement and sort them ascendingly. It can be seen that the big rectangle has two biggest and smallest x and y coordinates while the small rectangle owns the others. After that, we must ensure that the small rectangle must have an area from 10% - smaller than 30% of the map and the big rectangle's area is 50% - smaller than 70% of the map. If the conditions are not satisfied, the small and big rectangles would be chosen again.

Next, we check whether the treasure is somewhere inside the gap between 2 squares or not by getting all tiles' coordinates that lie inside the big and small rectangles. Then we keep the tiles that are inside the big rectangle but outside the small rectangle and check if the treasure position is in those tiles or not.

This function returns a tuple of `(14, log, truth, big_square_small_square)` where 14 is the type of hint, `log` is for visualization for example "The treasure is somewhere in the gap between 2 squares: *S1 = [1, 3, 15, 13], S2 = [3, 5, 10, 10]*", **truth** is True if the treasure is somewhere inside the gap between 2 squares and False otherwise, `big_square_small_square` contains the top left and bottom right coordinates of big and small rectangle respectively.

15. The 15th typed hint

This hint stated that "The treasure is in a region that has mountain."

We implement this hint by first getting all regions that contain mountain and then check if the treasure position is belonged to those regions.

This function returns a tuple of `(15, log, truth, list_mountain_region)` where 15 is the type of hint, log is for visualization for example "The treasure is in one of the regions 1, 2, 3 which have mountain", truth is True if the treasure is in a region that has mountain and False otherwise, list_mountain_region contains the regions that has mountain.

VI. Pirate

1. Algorithm description

First, this class generates a list of possible next moves for the pirate based on its current position, one step in four directions (up, down, left, right). Then, for each of these moves, it generates a list of next moves by moving the pirate one more step in four directions. Finally, the function sorts the list of all next moves by their total path cost (the sum of the path cost from the start to the current position and the heuristic cost to the treasure) and their direction. The direction is used as a tiebreaker when two next moves have the same total path cost. The resulting list of next moves is then returned.

We use A* search algorithm for finding the shortest path between two positions on a map. The algorithm is used to find the shortest path for the pirate to reach the treasure. The `find_shortest_path` function takes as input the current position of the pirate and the position of the treasure, and returns the shortest path from the current position of the pirate to the treasure.

Pseudo code

`move_4_direction`

```
FUNCTION  move_4_directions(self, cur_move):
        RETURNS // best possible next_moves
        cur_pos<-cur_move[2:4]
        prev_direction<-cur_move[4]
        IF np.array_equal(cur_pos, self.cur_pos)
            THEN
            step<-1
            cost<-cur_move[6] + 1
            prev_direction<--1
        ELSE
            step<-2
            cost<-cur_move[6]
        ENDIF
ENDFUNCTION

        offset_pos<-np.array([[0, -1], [0, 1], [-1, 0], [1, 0]])
        prev_pos<-np.full((4, 2), cur_pos)
        next_pos<-cur_pos + offset_pos
        IF prev_direction = 0
            THEN
            directions<-np.array([0, 1, 4, 5])
        ENDIF
        elif prev_direction = 1:  //You will need to change this to CASE OF
            directions<-np.array([0, 1, 6, 7])
        elif prev_direction = 2:  //You will need to change this to CASE OF
            directions<-np.array([8, 9, 2, 3])
        elif prev_direction = 3:  //You will need to change this to CASE OF
            directions<-np.array([10, 11, 2, 3])
        ELSE
            directions<-np.array([0, 1, 2, 3])
        directions<-directions.reshape((4, 1))
        num_tiles<-np.full((4, 1), step)
        path_costs<-np.full((4, 1), cost)
        heuristics<-np.apply_along_axis(
            self.cal_manhattan_dist, 1, next_pos).reshape((4, 1))
        next_moves<-np.concatenate(
            (prev_pos, next_pos, directions, num_tiles, path_costs, heuristics), axis=1)

        // submask_1 checks whether the next move of pirate is still in the map or not
        // submask_2 checks whether the next move of pirate is not to the mountain, sea or not
        // submask_3 checks whether the next move is different from the cur_pos of the pirate or not
        // mask check all conditions stated in submask_1 and submask_2
        col<-next_moves[:, 2]
        row<-next_moves[:, 3]
        pos<-next_moves[:, 2:4]
        submask_1<-np.logical_and(np.logical_and(col >= 0, col < self.hint_manager.map.width), np.logical_and(
            row >= 0, row < self.hint_manager.map.height))
        submask_2<-np.logical_and(self.hint_manager.map.map[col, row] != 0, np.array(
            [not np.any(np.equal(p, self.hint_manager.map.mountains).all(1)) FOR p in pos]))  //Pseudocode can't handle this
        submask_3<-~np.equal(pos, self.cur_pos).all(1)
        mask<-np.logical_and(np.logical_and(submask_1, submask_2), submask_3)
        // remove all the next move of pirate that is not satisfied the above conditions
        next_moves<-next_moves[mask]

        return next_moves
```

`get_next_moves`

```
FUNCTION  get_next_moves(self, cur_move):
RETURNS // next_moves
ENDFUNCTION

        // move up, down, left, right by 1 step first
        next_moves<-self.move_4_directions(cur_move)

        // now from each tile that we have moved to, we mobe by 1 step to create 2-step moves
        FOR move in next_moves:  //Pseudocode can't handle this
            next_2_tiles_moves<-self.move_4_directions(move)
            next_moves<-np.concatenate(
                (next_moves, next_2_tiles_moves), axis=0)

        // perform sorting so that all next moves are ordered by path_cost + heuristics and then by the direction since mr.Thuyen sa
        // base on the shortest path but prioritize the straight path with largest steps
        ind<-np.lexsort(
            (next_moves[:, 4], next_moves[:, 6] + next_moves[:, 7]))
        next_moves<-next_moves[ind]

        return next_moves
```

`find_shortest_path`

```
FUNCTION  find_shortest_path(self):
        RETURNS // pirate.path consists of best path and log
        '''
            Find shortes path to treasure using A* search
        '''
        initial_move<-np.array(
            [-1, -1, self.cur_pos[0], self.cur_pos[1], -1, 0, 0, self.cal_manhattan_dist(self.cur_pos)])
ENDFUNCTION

        frontier<-np.array([initial_move])

        explored<-None
```

```
    WHILE np.size(frontier) DO
        // choose 1 and move and do not left anything
        cur_move<-frontier[0]
        frontier<-frontier[frontier.shape[0] + 1:]
    ENDWHILE

        self.cur_pos<-cur_move[2:4]
        log<-self.get_log(cur_move)
        self.path.put((self.cur_pos, log))

        // goal test
        IF np.array_equal(self.cur_pos, self.treasure_pos)
            THEN
            break  //This might be better as a repeat loop
        ENDIF

        // add node to explored
        IF explored is None
            THEN
            explored<-np.array([self.cur_pos])
        ELSE
            explored<-np.concatenate(
                (explored, self.cur_pos[None, :]), axis=0)
        ENDIF

        // get successors and remove successors which are in explored
        next_moves<-self.get_next_moves(cur_move)
        mask<-np.array([not np.any(np.equal(move, explored).all(1))
                        FOR move in next_moves[:, 2:4]])  //Pseudocode can't handle this
        next_moves<-next_moves[mask]

        // add to frontier and sort again
        frontier<-np.concatenate((frontier, next_moves), axis=0)
        ind<-np.lexsort((frontier[:, 4], frontier[:, 6] + frontier[:, 7]))
        frontier<-frontier[ind]

    self.cur_pos<-self.initial_pos
    return
```

3. Explanation

The `move_4_directions` function receives the current move of the pirate as an input and returns a list of possible next moves that the pirate can take in four different directions: up, down, left, and right. Each move consists of the previous position, the next position, the direction of the move, the number of tiles moved, the path cost (the total cost from the initial position to the next position), and the heuristics (the estimated distance from the next position to the treasure). The function also filters out any next moves that are outside the map, on mountains or seas, or on the same position as the current position of the pirate.

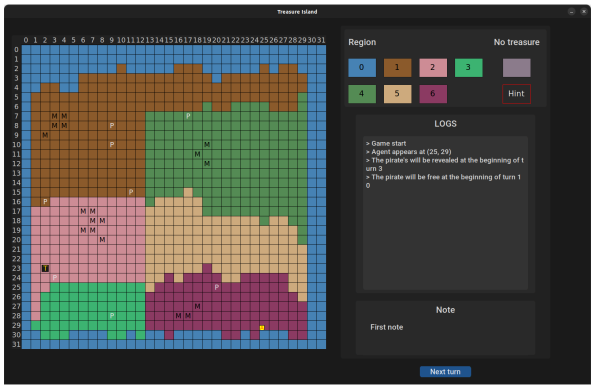
The `get_next_moves` function receives the current move of the pirate as an input and returns a list of all possible next moves by calling the `move_4_directions` function on the current move and on each of the four one-step moves returned by the `move_4_directions` function. The function then filters out any next moves that have already been explored (visited) and sorts the list of next moves by the path cost + heuristics and then by the direction, so that the moves with the lowest cost are prioritized.

The `find_shortest_path` function implements the A* search algorithm to find the shortest path from the current position of the pirate to the treasure. It initializes the search with an initial move that consists of the current position of the pirate and sets up a frontier (a list of moves that have not yet been explored) with the initial move. The function then enters a loop that continues until the frontier is empty. In each iteration, it chooses the move with the lowest cost in the frontier, adds the position of the move to the list of explored positions, and checks if the position is the treasure. If it is, the function breaks out of the loop. If it is not, the function calls the `get_next_moves` function to get the list of all possible next moves, filters out the moves that have already been explored, and adds them to the frontier. The frontier is then sorted by the path cost + heuristics and then by the direction. Once the loop is finished, the function resets the current position of the pirate to the initial position and returns the path from the initial position to the treasure.

▼ VI. Game visualization

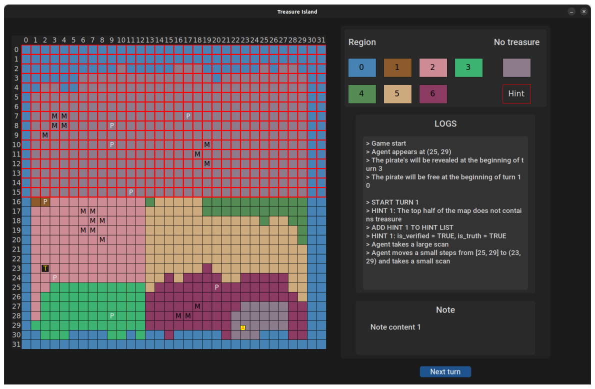
Init state:

1st turn:



```
> Game start
> Agent appears at (25, 29)
> The pirate's will be revealed at the beginning of t
urn 3
> The pirate will be free at the beginning of turn 1
0
```

First note

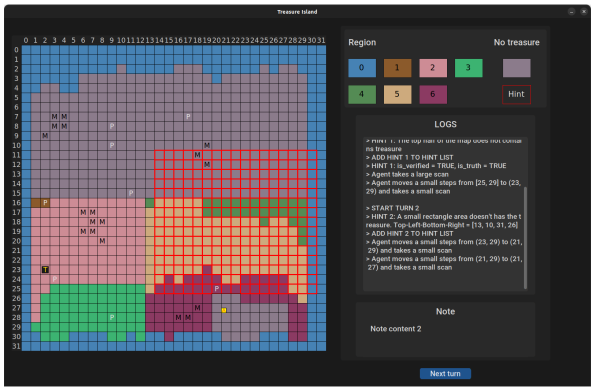


```
> Game start
> Agent appears at (25, 29)
> The pirate's will be revealed at the beginning of t
urn 3
> The pirate will be free at the beginning of turn 1
0

> START TURN 1
> HINT 1: The top half of the map does not contai
ns treasure
> ADD HINT 1 TO HINT LIST
> HINT 1: is_verified = TRUE, is_truth = TRUE
> Agent takes a large scan
> Agent moves a small steps from [25, 29] to (23,
29) and takes a small scan
```

Note content 1

2nd turn:

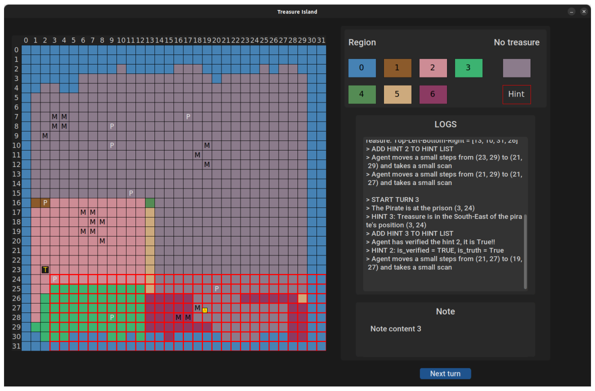


```
> HINT 1: The top half of the map does not contai
ns treasure
> ADD HINT 1 TO HINT LIST
> HINT 1: is_verified = TRUE, is_truth = TRUE
> Agent takes a large scan
> Agent moves a small steps from [25, 29] to (23,
29) and takes a small scan

> START TURN 2
> HINT 2: A small rectangle area doesn't has the t
reasure. Top-Left-Bottom-Right = [13, 10, 31, 26]
> ADD HINT 2 TO HINT LIST
> Agent moves a small steps from (23, 29) to (21,
 29) and takes a small scan
> Agent moves a small steps from (21, 29) to (21,
 27) and takes a small scan
```

Note content 2

3rd turn:

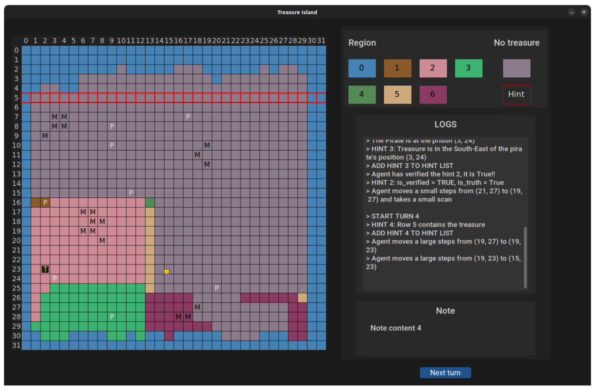


```
reasure. Top-Left-Bottom-Right = [13, 10, 31, 26]
> ADD HINT 2 TO HINT LIST
> Agent moves a small steps from (23, 29) to (21,
 29) and takes a small scan
> Agent moves a small steps from (21, 29) to (21,
 27) and takes a small scan

> START TURN 3
> The Pirate is at the prison (3, 24)
> HINT 3: Treasure is in the South-East of the pira
te's position (3, 24)
> ADD HINT 3 TO HINT LIST
> Agent has verified the hint 2, it is True!!
> HINT 2: is_verified = TRUE, is_truth = True
> Agent moves a small steps from (21, 27) to (19,
 27) and takes a small scan
```

Note content 3

4th turn:

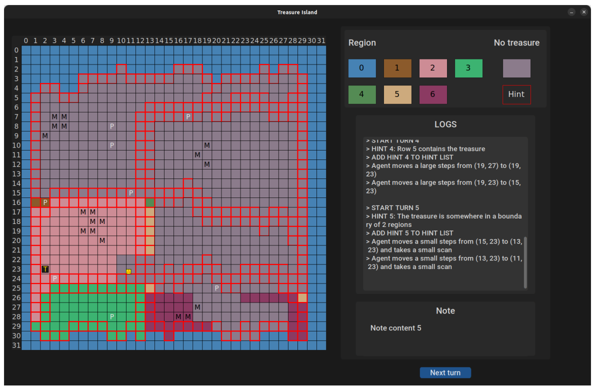


```
> The Pirate is at the prison (3, 24)
> HINT 3: Treasure is in the South-East of the pira
te's position (3, 24)
> ADD HINT 3 TO HINT LIST
> Agent has verified the hint 2, it is True!!
> HINT 2: is_verified = TRUE, is_truth = True
> Agent moves a small steps from (21, 27) to (19,
 27) and takes a small scan

> START TURN 4
> HINT 4: Row 5 contains the treasure
> ADD HINT 4 TO HINT LIST
> Agent moves a large steps from (19, 27) to (19,
23)
> Agent moves a large steps from (19, 23) to (15,
23)
```

Note content 4

5th turn:



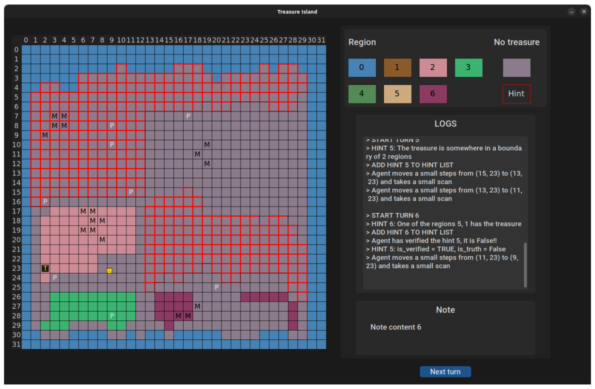
```
> START TURN 4
> HINT 4: Row 5 contains the treasure
> ADD HINT 4 TO HINT LIST
> Agent moves a large steps from (19, 27) to (19,
23)
> Agent moves a large steps from (19, 23) to (15,
23)

> START TURN 5
> HINT 5: The treasure is somewhere in a bounda
ry of 2 regions
> ADD HINT 5 TO HINT LIST
> Agent moves a small steps from (15, 23) to (13,
 23) and takes a small scan
> Agent moves a small steps from (13, 23) to (11,
 23) and takes a small scan
```

Note content 5

6th turn:

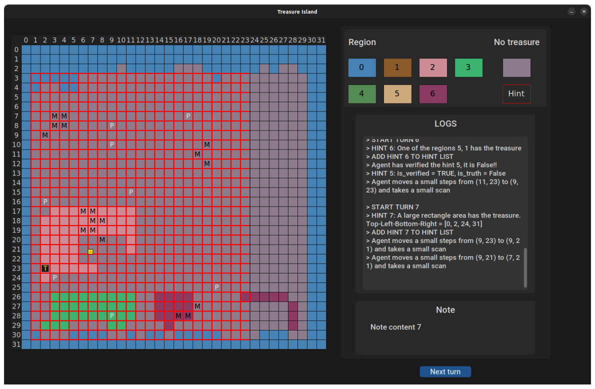
7th turn:8th turn:



```
> START TURN 5
> HINT 5: The treasure is somewhere in a bounda
ry of 2 regions
> ADD HINT 5 TO HINT LIST
> Agent moves a small steps from (15, 23) to (13,
 23) and takes a small scan
> Agent moves a small steps from (13, 23) to (11,
 23) and takes a small scan

> START TURN 6
> HINT 6: One of the regions 5, 1 has the treasure
> ADD HINT 6 TO HINT LIST
> Agent has verified the hint 5, it is False!!
> HINT 5: is_verified = TRUE, is_truth = False
> Agent moves a small steps from (11, 23) to (9,
23) and takes a small scan
```

Note content 6



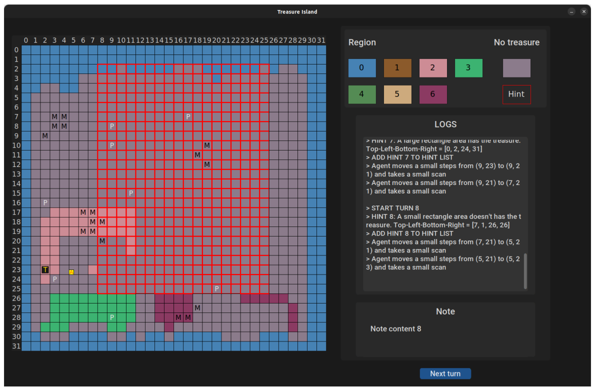
```
> START TURN 6
> HINT 6: One of the regions 5, 1 has the treasure
> ADD HINT 6 TO HINT LIST
> Agent has verified the hint 5, it is False!!
> HINT 5: is_verified = TRUE, is_truth = False
> Agent moves a small steps from (11, 23) to (9,
23) and takes a small scan

> START TURN 7
> HINT 7: A large rectangle area has the treasure.
Top-Left-Bottom-Right = [0, 2, 24, 31]
> ADD HINT 7 TO HINT LIST
> Agent moves a small steps from (9, 23) to (9, 2
1) and takes a small scan
> Agent moves a small steps from (9, 21) to (7, 2
1) and takes a small scan
```

Note content 7

8th turn:

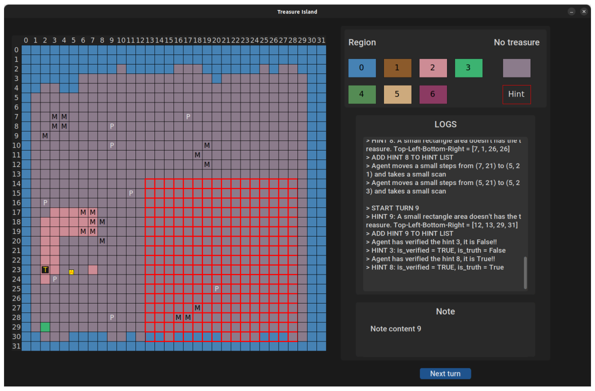
9th turn:



```
> HINT 7: A large rectangle area has the treasure.
Top-Left-Bottom-Right = [0, 2, 24, 31]
> ADD HINT 7 TO HINT LIST
> Agent moves a small steps from (9, 23) to (9, 2
1) and takes a small scan
> Agent moves a small steps from (9, 21) to (7, 2
1) and takes a small scan

> START TURN 8
> HINT 8: A small rectangle area doesn't has the t
reasure. Top-Left-Bottom-Right = [7, 1, 26, 26]
> ADD HINT 8 TO HINT LIST
> Agent moves a small steps from (7, 21) to (5, 2
1) and takes a small scan
> Agent moves a small steps from (5, 21) to (5, 2
3) and takes a small scan
```

Note content 8

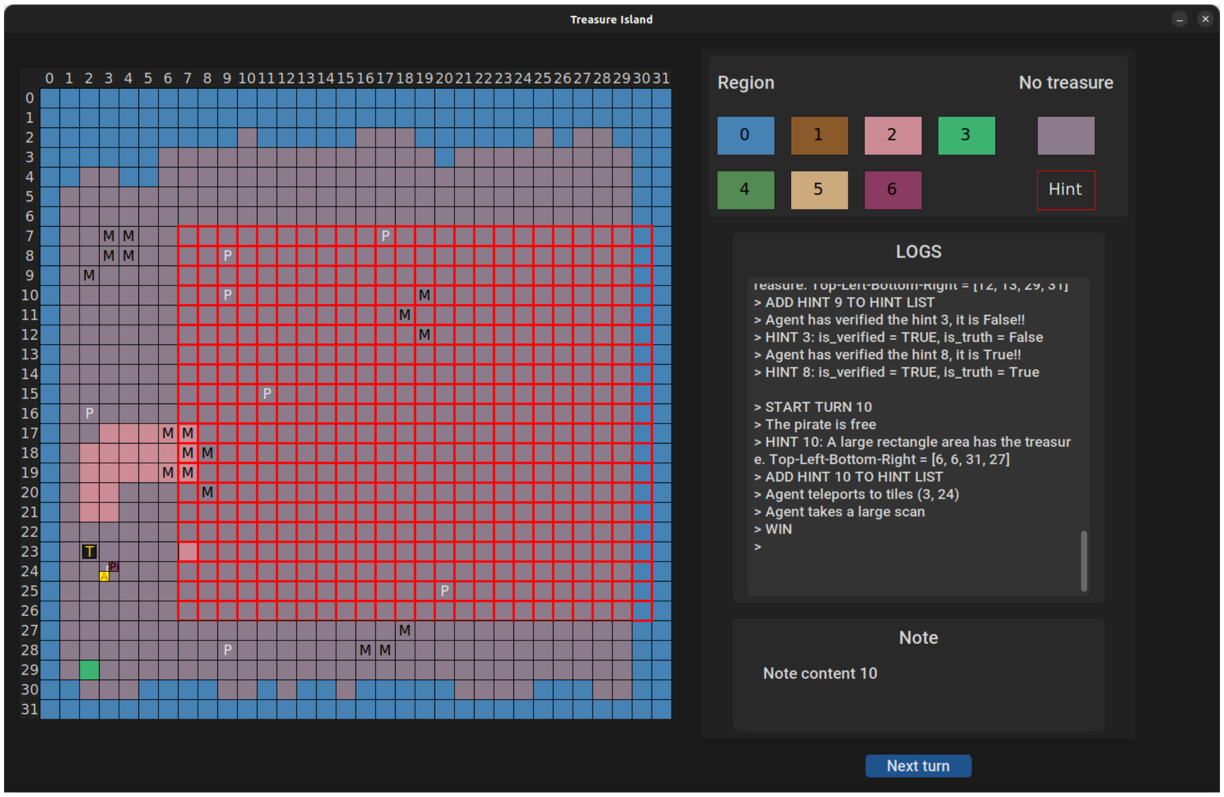


```
> HINT 8: A small rectangle area doesn't has the t
reasure. Top-Left-Bottom-Right = [7, 1, 26, 26]
> ADD HINT 8 TO HINT LIST
> Agent moves a small steps from (7, 21) to (5, 2
1) and takes a small scan
> Agent moves a small steps from (5, 21) to (5, 2
3) and takes a small scan

> START TURN 9
> HINT 9: A small rectangle area doesn't has the t
reasure. Top-Left-Bottom-Right = [12, 13, 29, 31]
> ADD HINT 9 TO HINT LIST
> Agent has verified the hint 3, it is False!!
> HINT 3: is_verified = TRUE, is_truth = False
> Agent has verified the hint 8, it is True!!
> HINT 8: is_verified = TRUE, is_truth = True
```

Note content 9

10th turn:



```
reasure. Top-Left-Bottom-Right = [12, 13, 29, 31]
> ADD HINT 9 TO HINT LIST
> Agent has verified the hint 3, it is False!!
> HINT 3: is_verified = TRUE, is_truth = False
> Agent has verified the hint 8, it is True!!
> HINT 8: is_verified = TRUE, is_truth = True

> START TURN 10
> The pirate is free
> HINT 10: A large rectangle area has the treasur
e. Top-Left-Bottom-Right = [6, 6, 31, 27]
> ADD HINT 10 TO HINT LIST
> Agent teleports to tiles (3, 24)
> Agent takes a large scan
> WIN
>
```

Note content 10

VI. Comment and evaluation

To summarize, the intelligence of the agent can be divided into two stages. The first stage happens before the pirate is freed. At this point, since the agent knows nothing about the exact position of the treasure, we decide to let the agent move as greedily as he can. Before taking an action, we have him consider carefully which action can give as much information (a.k.a the highest number of unknown tiles that the agent can scan). Of course, this amount of information cannot cover the entire map, but it will be very useful since it helps narrow the search space to the next stage. The agent remembers every useful information in the form of a bitmask (binary map), which we refer to as the agent's “knowledge”.

The second stage starts right after the pirate is freed from his prison. At the pirate's free turn, no matter where the agent is, he must teleport immediately to the position of the pirate. Although this move sounds a bit stupid, it can work effectively in most cases. This stage also applies the greedy strategy (try to explore as much information) *(1)* but is more intelligent since the agent tends to move to the “mean” tile of the rest available tiles *(2)*. We found this strategy is very effective when the number of available tiles decreased. We also propose another technique to take the direction of moves of the pirate into account: after every pirate's move, we temporarily mask tiles lying on the opposite side of the map. It helps the agent focus only on tiles lying in front of the pirate's face, thanks to this, the “mean” of the tiles becomes more accurate than ever *(3)*. From *(1)*, *(2)*, and *(3)*, the agent is now intelligent enough to take advantage of his prior knowledge as well as follow the pirate's steps, or even look further to predict the more feasible tiles for treasure.

The final source code and instructions are attached in this repository: https://github.com/nxquang2002/CS420-Project-2.git

END.