The *Treasure Hunt* game

1 Context

Treasure hunt is a 1 vs 1 video game created by Clara Thomas and Luc Vignolles (ECN Robotics major 2020-2021). The game is played on a grid were each player controls a boat. The goal is to find the treasure while avoiding hitting the rocks. At each turn your boat is able to retrieve the Euclidean distance to the treasure, without knowing where it is exactly.

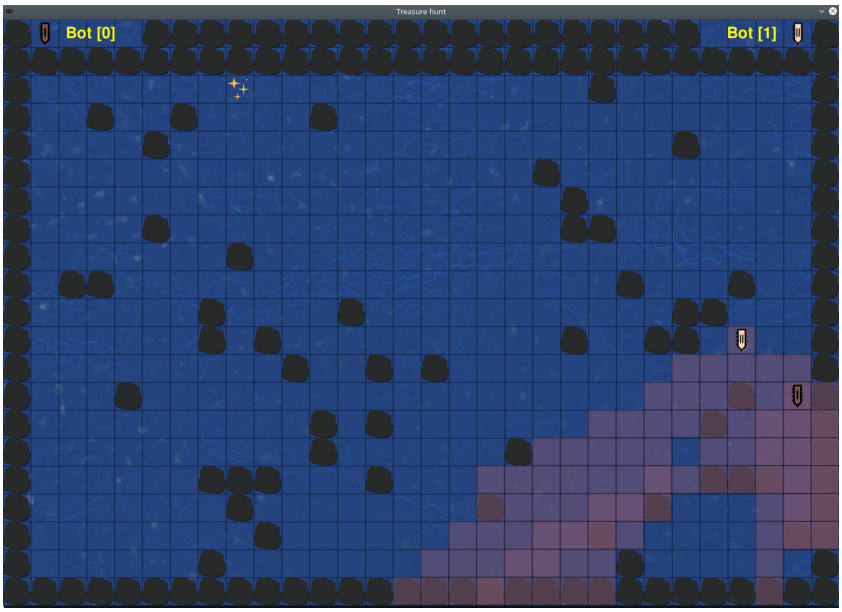

Figure 1: Game interface where the first player (Bot 0) bas the brown boat while second player (Bot 1) has the white one and uses the sonar. The treasure here is around the top-left corner.

To get the code template, go to your favorite coding folder and type:

```
cp -r /opt/duels/games/treasure_hunt .
cd treasure_hunt
gqt # configure QtCreator
```

The files are then ready to use in the `treasure_hunt` folder.

2 Game description

The initial code is an infinite loop where the *feedback* variable is sent by the game. You have 100 ms to compute your *input* and send it to the game. You can fight various AI levels (from 0 to 3) depending on your confidence in your own AI. The code should be compiled according to the included `CMakeLists.txt` file.

2.1 Input rules

The input to send to the game is simply the action you want to take now. Your boat may turn left or right, move forward or use its sonar to detect the rocks ahead:

- `input.action (Action)`: which action you want to take:
  - `Action::TURN_LEFT`
  - `Action::TURN_RIGHT`
  - `Action::MOVE`
  - `Action::SONAR`

The feedback you get from the game is composed of the pose of your boat, a list of positions of detected rocks if you have used the sonar, and the current distance to the treasure.
From a programming point of view:

- `feedback.pose (Pose)` is a 2D pose which is a structure composed of `(x,y,theta)` where `theta` can be:
  - `Orientation::RIGHT`
  - `Orientation::LEFT`
  - `Orientation::UP`
  - `Orientation::DOWN`
- `feedback.scan (std::vector<Scan>)` is a list of `Scan`, which is a structure composed of `(x,y,occupied)`. It is empty if you have not used `Action::SONAR` as the previous action.
- `feedback.treasure_distance (float)` is the Euclidean distance to the treasure

2.2 Game rules

You lose the game if you hit a rock, you win if you go to the treasure before the opponent. Draws are possible when both players make a winning / losing move at the same turn.

3 Expected work

3.1 A class for your AI

An AI class is already created and plays at line 23 in the initial `treasure_hunt.cpp`:

```
input.action = ai.computeFrom(feedback);
```

Your goal is to write this function that should return what is the best action to reach the goal and not hit any rocks. In the next section, the algorithm to implement is presented.

3.2 A smart algorithm

The algorithm uses two member variables that are initialized at the top of `treasure_hunt_ai.cpp`

- `map`: a 20×30 grid

  - `map.cell(x, y)` or `map.cell(pose)` allows accessing and modifying the cell at given coordinates
  - Aliases have been defined as `FREE`, `ROCK` or `UNKNOWN`. At the beginning the map is filled with `UNKNOWN`.

- `candidates`: a vector of 2D positions corresponding to all possible positions of the treasure

These variables are already set in the constructor of `TreasureHuntAI`.

The algorithm to reach the treasure is then composed of four steps that need to be implemented in `TreasureHuntAI::computeFrom(const Feedback &feedback)`:

3.2.1 Update the map from the sonar (line 31)

From `feedback.scan`, update `map` (without sonar, the scan is empty)

3.2.2 Identify the treasure positions (line 39)

Use `feedback.treasure_distance` and the current pose to remove the impossible treasure positions in `candidates`. After a few moves only one position should remain. It is done by defining the lambda function called by `pruneCandidate`.

3.2.3 Identify the first move from the A* path (line 50)

The next pose in the path is at `path[1]`. You have to find how to go from the current pose to this one:

- if their `(x,y)` position is not the same then the action is `Action::MOVE`
- otherwise the orientation can be found by using the `leftFrom` and `rightFrom` functions that return the left and right orientation from an initial one.

If the resulting action is `Action::MOVE` then you have to check if the next cell is indeed `FREE`. If not, use the sonar instead.

3.2.4 BONUS Find the path to a candidate (line 47)

The given A* algorithm will find a way from any pose to any other position:

```
std::vector<Point> Astar(Point start, Point goal)
```

The goal can be any remaining candidate. In the given code the picked candidate is just the first one in the list. Try to find the path to the closest candidate instead.

N.B. in practice this is really useless as by definition, after parsing `feedback.treasure_distance` all remaining candidates have the same distance to the boat!

4 Provided tools

4.1 A* algorithm

The same A* algorithm as in the `Maze` project is provided. The syntax is:

```
const auto path = Astar(start, goal)};
```

The output variable `path` is of size 0 is no path was found. Otherwise, it begins with `start`, contains the intermediary path and ends with `goal`.

4.2 Boat Node

The file `boat_node.h` implements a class that is compatible with the A* algorithm. In particular, a node is composed of `(x,y,orientation)` and can generate the reacheable positions:

- Same position but rotated left or right (always possible)
- Same orientation but moved forward (only if the front cell is free or unknown)

4.3 Orientation

Two functions are given to help playing with orientation:

```
Orientation leftFrom(const Orientation &orientation);
Orientation rightFrom(const Orientation &orientation);
```

They simply return the left or right orientation from the given one.