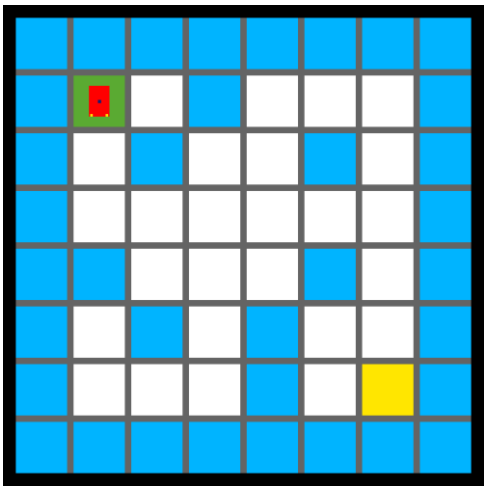
Assignment 3: Frozen Lake

Introduction

We present to you a variation of the Grid Explorer problem: the problem of the Frozen Lake. You want to be able to place your robot in the middle of a frozen lake, and train it so it learns to maneuver its way from the start point to a destination point you set. The robot is even pre-calibrated, so the task sounds easy enough, right? Unfortunately, the problem is not as easy as it seems, as some sections of the lake are not completely frozen; too much weight will break the ice within those sections. Your robot is a heavy one, so a misstep will result in emotional and financial ruin, with your non-waterproof and very expensive robot plunging into the depths below. Mistakes are not inevitable though; your robot comes equipped with sensors that detect whether the ice on some of the blocks adjacent to it is too thin.

You want to train your robot with a simulator so that it can learn how to solve the frozen lake problem and avoid such sections while managing to reach the destination point. Isn't it great then that we have just the right simulator for you to train this robot? In this assignment, you will be required to implement Q-learning for the robot so that it can solve the problem. Just like the Grid Explorer assignment, there are three files that will be provided to you: **student.py**, **simulator_hidden.py**, and **simulator.py**. **simulator_hidden.py** serves as the backbone of the simulator; this file creates the simulator and gets it running. **student.py** is where you will code your algorithm. Last but not least, **simulator.py** serves as a middleman between **simulator_hidden.py** and **student.py;** it gives access of some useful functions implemented in **simulator_hidden.py** to **student.py**.

Figure 1. Simulator initialization



When we start the simulator, a randomized grid containing 36 blue cells, a green cell, and a yellow cell will be shown to you. The green cell indicates the start point, and the yellow cell indicates the goal point. These two cells are fixed. Blue cells indicate sections with thin ice, which you should avoid. 28 out of the 36 blue cells serve as the border of the grid and are also fixed; the remaining 8 will be placed into the middle randomly.

simulator.py provides you with the following functions: move_forward(), turn_left(), turn_right(), reset_map(), set_speed(), show_animation(), and test().

reset_map() is the function you should call when you want to re-initialize the map. Unlike the Grid Explorer assignment, the simulator by default is in "training mode", and will not stop the program even if the robot gets wasted (by courtesy of *Grand Theft Auto*) by going to a thin ice block, and no notification will pop up as well. The simulator will still understand that the robot is "dead", however, despite the fact that the robot is still shown on the map.

Therefore, whenever you want to reset the map and/or revive your robot, you must call reset_map() yourself.

reset_map() will also return the initial state of the robot in the new map as a 4-dimensional tuple. The first element is a 2-dimensional tuple and is the coordinates of the robot. Like the Grid Explorer assignment, the top left corner of the grid is (0,0), and horizontal movements change the first coordinate, while vertical movements change the second coordinate. The second element is the current orientation, which is represented by the strings 'N', 'E', 'S', and 'W', and denotes where the robot is facing. The third element is a list of sensor values of length 3. Element 0 corresponds to the block to the robot's left, element 1 is the block in front of the robot, and element 2 corresponds to the block to the robot's right, all with respect to its orientation. Sensor values are 1 if the block it corresponds to is a thin ice block, and 0 otherwise. Last but not least, the final element is the robot's status as a boolean: True if the robot either is wasted or has reached the goal point, and False otherwise. *The robot's initial state is set as (1, 1), facing south, and False (alive and not at the goal).* There may or may not be a thin ice block in front of it (at block (1, 2)).

The functionality of move_forward(), turn_left(), and turn_right() are all self-evident. Note that turn_left() and turn_right() ONLY TURNS the robot in the corresponding direction, and does not move it to different coordinates. These movement functions now also return the current state AFTER the action is completed, in the same format as described above. If these functions are called after the robot is dead, it will not move the robot and will just return the state at which it died, indicating that you probably want to call reset_map().

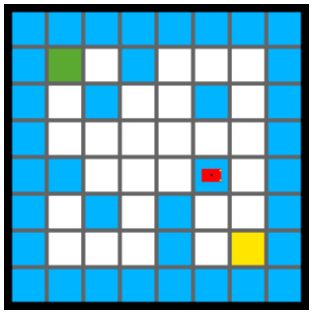


Figure 2a. Getting wasted

By calling move_forward() at (4, 4) while facing east, the robot would move to a thin ice block, and the function will return the following state: ((5, 4), 'E', [0, 0, 0], True). The robot is dead as indicated by the True boolean, but it'll be stuck there until you call reset_map().

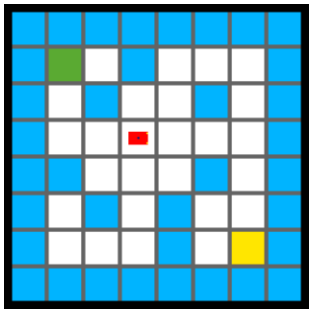


Figure 2b. A normal case

By calling move_forward() at (2, 3) while facing east, the robot would move to a normal block, and the function will return the following state: ((3, 3), 'E', [0, 0, 0], False). The robot is still alive and there is nothing in front of it.

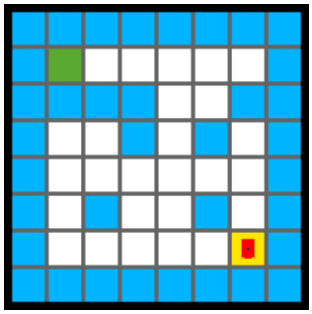


Figure 2c. Winner winner chicken dinner

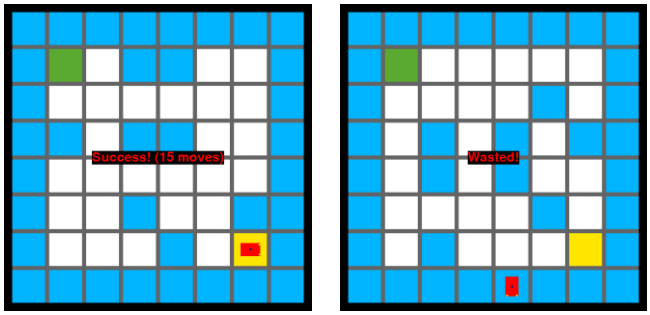
By calling move_forward() at (6, 5) while facing south, the robot would move to the goal block, and the function will return the following state: ((6, 6), 'S', [1, 1, 0], True). The robot is still alive and it has reached the goal state!

set_speed() and show_animation() are provided to you for QoL (quality of life) changes. show_animation() is a function that sets a boolean that indicates whether the simulator will be visually updated as it processes new frames (True) or it won't be updated (False; instead, it'll just be stuck on the initial frame, and you will not see the car move). set_speed() allows you to set the frame rate of the simulator when animations are enabled in units of frames/second; giving the function a higher number means that the simulator will process more frames a second and will therefore update the frames faster. By default, the speed is initialized to 5 frames/sec. It is recommended that you disable animations or increase the frame rate if you want to lower training time, and that you set the speed low and enable animations before testing (so you can actually see what the robot is doing, which might help you debug).

test() will set the simulator into test mode. You should change to test mode and then call reset_map() when you think the Q-table is done converging and want to test how well your robot is trained on a new test map. The program will give you a "Success" or "Wasted!" notification if your robot reaches either the destination or a thin ice block, respectively. Once a notification is displayed, the program will shut itself down. It is also possible for your robot to get stuck in an infinite loop. In this case, the code will never end, and you will have to close the program manually. test() also automatically enables animations.

Figure 3. Running the code

These are examples of what your code should look like once it calls test() and goes through the test map.



Assignment 3

You are required to fill in the blank areas in **student.py**. Do not touch the other two files, and do not import **simulator_hidden** in **student.py**. Implement Q-learning with the following structure (steps that are provided to you in the template are denoted as orange):

1. Training
   a. (Optional but recommended once you know how the simulator works) Set the frame rate high or disable animations.
   b. Implement steps 1 to 3 of Q-learning as shown on slide 13 of the Lab 6 presentation to learn the Q-table.
   c. Save the Q-table as a file.
2. Testing
   a. (Optional but recommended) Set the frame rate low.
   b. Call test() and reset_map().
   c. Implement step 4 of Q-learning as shown on slide 13 of the Lab 6 presentation to get the robot to use the optimal policy.

After implementation, you can run the code to see if you did a good job:

a. Run the code and watch the robot as it trains (or not if you disabled animations)
b. Watch during testing mode to see whether the robot succeeded the test map or not.

Obviously, just because the robot succeeded in the test map doesn't mean it will succeed in all cases. We have therefore also provided to you a file called **test.py** in which you can set your own thin ice blocks and test out maps on your own with your saved Q-table. Feel free to modify **test.py** however you like.

Also, your saved Q-table file will be overwritten every time you complete training. If there is a Q-table that you think solves the task pretty well but you still want to try out new things, please change the name of that file so that it isn't overwritten!

Several fixed test maps will be used for grading. These test maps will not be provided. Grading will take into consideration the following:

1. The number of test maps succeeded
2. The number of steps it takes for each success

Submission Guidelines

Please submit a zip file named "studentid_studentname.zip" containing the **logs** folder, the Q-table file named "q_table.npy", and **student.py** to KLMS by **May 25th, 23:59 pm**.

Hints:

1. Remember to think of the problem as a Markov Decision Process! You have many choices to make on how to structure this problem. What are the states? What are the actions? What are the rewards?
2. What should the choices for your hyperparameters — alpha, gamma, epsilon, and the number of epochs — approximately be? Play around with them and think about how your choices can affect the learning process.
3. Some useful numpy functions/modules include but are not limited to: zeros(), ones(), where(), max(), argmax(), and random.