Final Project: AlphaRobot

CS3317

October 11, 2023

1 About this Project

- For this project, students need to work in groups of 3 to implement the AI (the control policy) for a mobile robot.
- Please use Python for this project.
- Every group will be assigned an ID number such as 1,2,3... Please remember your ID number.
- Please submit a zip file of a folder named Robot_[ID], where [ID] is the ID number of your group. For example, if your ID is 3, then your folder name should be Robot_3.
- Please refer to Section 4 for the details of what to include in the folder.

1.1 Auto-Grading

- Basic scores: If your submission is able to outperform the baseline that we prepared, your group will get a basic score of 60%.
- Additional scores: you will get additional scores according to your ranking among all 10 groups.

Table 1: Ranking

Ranking	Score
1	+40%
2	+40%
3	+40%
4	+30%
5	+30%
6	+30%
7	+20%
8	+20%
9	+10%
10	+10%

Take it easy! The baseline is easy to outperform.

- Along with the submisssion, you also need to provide a contribution table based on mutal agreement. The sum of all of contributions must be 300% (if there are 3 members), but individual contributions can exceed 100%. Equal contribution (100%, 100%, 100%) is allowed. In the example below, if the total points obtained by the group is 80, then student 1 will get a score of 80* 100%=80, student 2 will get 80×110%=89, and student 3 will get 80×90%=72. Overflow of scores (the part exceeding 100, if it happens) will be regarded as bonus to the total grading.

Table 2: An example of the contribution table

Name	Contribution
Student 1	100%
Student 2	110%
Student 3	90%

1.2 Presentation

You are required to present your method and results as a group in Week 16. To get higher grades:

- Make the presentation structured and clear.
- Present collaboratively.
- Present your exploration path (different approaches you tried, the iteration process of a single method, heuristics you tried, hyper-parameter tuning, ...) and discuss the insights gained from it.
- Discuss the limitations of your method and possible future work to address them.

1.3 Deadline

This project is due on Friday, Week 15 (**23:59, December 22nd**).

2 Mobile Robot Navigation

The navigation environment setup is shown in Figure 1.

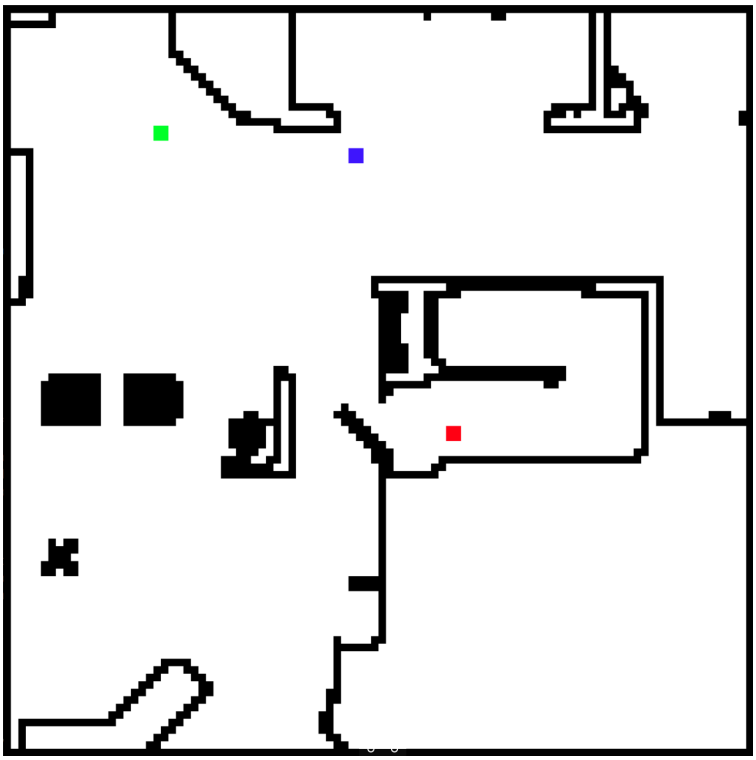

Figure 1: The task environment of robot navigation

2.1 Task description

- The map: a map is a top-down occupancy grid of a household environment. It has a size of 100*100, with black denoting obstacles and white denoting free spaces for the robot to move around. There are in total five maps, representing five different houses. The blue block represents the start position of the robot and the red block represents the goal position.
- The robot: a square-shaped mobile robot navigates on the map. It has a size of 2*2 and is colored green. It can move towards four directions (left, up, right, down) with four possible velocities (0,1,2,3). A robot taking velocity 1 can move one pixel per step.
- The robot's action: what we can control is the robot's acceleration and rotation. At each step, the robot can either increase the speed by 1 (accelerate), retain the speed, or decrease the speed by 1 (decelerate). Also, it can choose to turn left by 90 degrees, turn right by 90 degrees, or continue forward to control its facing directions. However, the robot can't simultaneously change its speed and direction (to prevent turning over). So, it can only rotate when the moving speed is 0.
- The robot's observation: the robot can observe the map. It can also observe its position, speed, and facing direction to make decisions accordingly.
- Control noise: there is noise in both speed and rotation control. With a small probability, the robot can achieve incorrect velocities or reach wrong directions when controlling the acceleration or rotation. The corresponding probabilities are unknown.
- The task: a task is specified as a pair of start and goal positions on the map. The robot has to start from the start location, navigate to the goal, and stop at the goal precisely at zero speed. Passing the goal at a non-zero speed is not regarded as success.
- Collision constraints: when the robot's shape overlaps with the map's walls, it causes a collision. In the case of collision, the robot will be forced to stop or bounce off the wall depending on the velocity of the robot, possibly causing some damage to itself. If the damage accumulates, the robot will be destroyed.

2.2 Performance measure

To evaluate the performance of your controller, we will measure the success-weighted path length (SPL) [1]. SPL is an evaluation metric that rewards agents who reach their goals and do so efficiently with respect to the length of their path. Here, path length is defined as the number of time steps.

Particularly, the SPL of an episode is defined as follows:

$$\frac{1}{N}\sum_{i=1}^{N} S_i \frac{l_i}{\max(p_i, l_i)} \tag{1}$$

where:

- N is the number of total evaluation episodes.
- S_i is a binary number indicating whether the robot succeeds in the ith episode. An episode succeeds if the robot stops at the goal precisely at zero speed. It fails if the robot tries to execute an illegal action in any step, does not succeed within the allowed amount of steps (1000), or collides more than five times (which would totally damage the robot). Moreover, if the robot fails to return an action within the given thinking time (1s) in any step, it also fails.

- p_i is the actual path length (the number of time steps) that the robot actually takes to reach a successful or failing state in the ith episode.
- l_i is the ideal path length for optimally solving the task in the ith episode. We estimate it using the following relaxations (this part has been taken care of in our evaluator):
  - There is no obstacle in the map and no noise in the robot's control.
  - In an empty space, the robot travels at the maximum speed except when making turns.
  - When coming near walls, the robot maintains a low speed of 1 and navigates cautiously.

In effect, a successful episode with an actual path length smaller or equal to l_i would get an SPL score of 1; a successful episode with an actual path length of $2l_i$ would get an SPL score of 0.5; a failed episode would get an SPL score of 0;

2.3 Performance evaluation

To perform auto-grading, each team's robots will be tested with 20 tasks on each of the 5 maps, where each task will be repeated 10 times. So, we will run a total of 5 * 20 * 10 episodes for each submitted policy, report the average SPL, and rank your robots accordingly.

3 Python APIs

We have provided a package of Python scripts for you. Please use it as the code base to complete your project.

3.1 Setting up the environment

The project is tested under Python 3.8. Therefore, if your local Python version is not 3.8, we recommend using an Anaconda virtual environment.

```
conda create -n AlphaRobot python=3.8 # create the conda environment

# launch the conda environment
conda activate AlphaRobot # for linux
activate AlphaRobot # for windows

# do your work

# exit the conda environment when you are done
conda deactivate
```

Inside the conda environment, you can install the required packages and validate the installation:

```
cd alpha_robot
pip install -r requirements.txt
python run_once.py # you should see a robot moving on the map as in Figure 1.

```

3.2 Implement your robot policy

You policy should be implemented in a python script named policy_[ID].py, where [ID] is your group id. You can refer to alpha_robot/policy.py as an example. Here is what it looks like:

```python
from grid_map_env.classes.action import Action
from grid_map_env.utils import *

import random

class Policy:
    def __init__(self) -> None:
        pass

    def get_action(self, house_map, robot_state):
        '''
        Calculate a legal action.
        Here we demonstrate a very simple policy that
        does not perform any from of search.
        '''

        if robot_state.speed < 2:
            acc = 1  # accelerate
        else:
            acc = -1  # decelerate
        action = Action(acc=acc, rot=0)  # construct an instance of the Action class

        next_state = self.transition(robot_state=robot_state, action=action) #
                                                    predict the transition
        # collision checking and response
        if is_collision(house_map=house_map, robot_state=next_state):
            # change the action due to collision in the predicted next state
            if robot_state.speed > 0: # decelerate to stop
                action = Action(acc=-1, rot=0)
            else: # choose random action
                random_action = random.choice([(0, 1), (0, -1)])
                action = Action(acc=random_action[0], rot=random_action[1])

        return action  # return the action for execution

    def transition(self,robot_state, action):
        '''
        a simple example for transition function
        '''
        next_state = robot_state.copy() #deep copy the robot state

        # update the robot's speed
        next_state.speed += action.acc
        next_state.speed = max(min(next_state.speed, 3), 0)

        #update the robot's position
        if next_state.speed != 0:
            if next_state.direction == 0:
                next_state.col -= next_state.speed
            elif next_state.direction == 1:
                next_state.row -= next_state.speed
            elif next_state.direction == 2:
                next_state.col += next_state.speed
            elif next_state.direction == 3:
```

```
                next_state.row += next_state.speed

        #update the robot's direction
        else:
            next_state.direction = (next_state.direction+action.rot) % 4

        return next_state # return the predicted next state

```

- Methods:
  - get_action: it's the core method for you to implement. It should calculate an action for the robot to execute, according to the given map and robot state. It must return a legal action, represented as an instance of the Action class (see Section 3.3.2).
    - Args:
      - house_map (list of list): a 2D list of integers representing the house map. Please refer to Table 6 for its encoding.
      - robot_state (RobotState): an instance of the RobotState class representing the current state of the robot. See Section 3.3.1 for more about the RobotState class.
    - Returns:
      - action (Action): an instance of Action class representing the action for execution. See Section 3.3.2 for more about the Action class.
  - transition: it's a simple example of transition function used by get_action. It won't check whether the action is legal, nor does it respond to collision.
    - Args:
      - robot_state (RobotState): an instance of the RobotState class representing the current state of the robot.
      - action (Action): an instance of Action class representing the action for execution.
    - Returns:
      - next_state (RobotState): an instance of the RobotState class representing the predicted state of the robot.

You can freely add your own member variables and methods or other attributes. But keep in mind that the get_action method is the main interface to be called by the robot. Therefore, please don't change the input and output format of it.

3.3 Interfaces

3.3.1 The RobotState Class

alpha_robot/grid_map_env/classes/robot_state.py contains the RobotState class that encodes a robot state.

```
class RobotState:

    def __init__(self, row=0, col=0, direction=0, speed=0):
        """
        Constructor for the robot class.
        """

```

```
        self.row = row
        self.col = col
        self.direction = direction
        self.speed = speed

    def copy(self):
        """
        Deep copy function for RobotState.
        """
        return RobotState(row=self.row,
                          col=self.col,
                          direction=self.direction,
                          speed=self.speed)
```

- Member variables:
  - row (int): an integer representing the row coordinate of the robot.
  - col (int): an integer representing the column coordinate of the robot.
  - direction (int): an integer representing the robot's direction, see Table 3 for its encoding.

Table 3: Encoding of robot direction

Encoding	Meaning
0	Facing left
1	Facing up
2	Facing right
3	Facing down

  - speed (int): an integer indicating the robot's speed, chosen from {0,1,2,3}.
- Methods:
  - copy: it deep copys an instance of the RobotState class.

3.3.2 The Action Class

alpha_robot/grid_map_env/classes/action.py contains the Action class that represents an action.

```
class Action:
    def __init__(self, acc, rot):
        self.acc = acc
        self.rot = rot

    def is_legal(self, robot_state):
        ...
```

- Member variables:
  - acc (int): an integer representing acceleration. See Table 4 for its encoding.

Table 4: Encoding of acceleration

Encoding	Meaning
-1	Decelerate by 1
0	Retain current speed
1	Accelerate by 1

- rot (int): an integer representing rotation direction. See Table 5 for its encoding.

Table 5: Encoding of rotation

Encoding	Meaning
-1	Turn left
0	Remain in the direction
1	Turn right

- Methods
  - is_legal: check the legality of an action. In a legal action:
    - acc can only be -1, 0, or 1.
    - rot can only be -1, 0, or 1.
    - rotation is allowed only when the robot's speed is 0, and no acceleration or deceleration is allowed when rotation is performed.

    This function will return True if the action is legal and False otherwise.

3.3.3 Utility functions

alpha_robot/grid_map_env/utils.py (please do not change this file!) provides useful utility functions. The most important ones are:

- **sample_start_and_goal**: sample a pair of feasible start and goal positions.

```
def sample_start_and_goal(map_file_path):
    ...
    return (start_row, start_col), (goal_row, goal_col)
```

  - Args:
    - map_file_path (string): a string representing the path to path file (text).
  - Returns:
    - start_pos (tuple): a tuple of two integers representing the sampled start (row, col).
    - goal_pos (tuple): a tuple of two integers representing the sampled goal (row, col).
  - A feasible task (specified by the start and goal positions) satisfies:
    - Both the start position and goal position are collision-free.
    - There exists at least one feasible path from the start position to the goal position.
    - The Manhattan distance between the start and the goal should be no smaller than 50.
- **is_collision**: check the collision between the robot and the map.

```
def is_collision(house_map, robot_state):
    ...
    return collision
```

  - Args:
    - house_map (list of list): a 2D list of integers representing the house map.
    - robot_state (RobotState): an instance of the RobotState class representing the current state of the robot.
  - Returns: a Boolean that represents whether any collision will occur.
- **is_goal**: the utility function to check whether the robot has stopped at the goal successfully.

```python
def is_goal(house_map, robot_state):
    ...
```

- Args:
  - house_map (list of list): a 2D list of integers representing the house map.
  - robot_state (RobotState): an instance of the RobotState class representing the current state of the robot.
- Returns: a Boolean representing whether the robot has stopped at the goal position.

3.4 How do I run an episode of the task using my policy?

To simulate an episode using your policy, you need to use our wrapped Gym environment, implemented as the GridMapEnvCompile class.

3.4.1 The GridMapEnvCompile Class

alpha_robot/grid_map_env/classes/grid_map_env_compile.py contains the GridMapEnvCompile class. It is a wrapped Gym environment with the following interfaces:

```python
class GridMapEnvCompile(gym.Env):
    def __init__(self, n, map_file_path, start_pos, goal_pos, headless=False):
        self.observation_space = spaces.Dict(
            {
                "map": spaces.Box(low=0, high=GOAL, shape=(n, n), dtype=int),
                "robot_pos": spaces.Box(low=0, high=n-1, shape=(2,), dtype=int),
                "robot_speed": spaces.Discrete(4),
                "robot_direction": spaces.Discrete(4)
        )
        ...

    def step(self, action):
        ...

    def reset(self):
        ...

    def render(self):
        ...

    def close(self):
        ...
```

- Member variables:
  - observation space (gym.spaces.Dict): observation_space specifies the format of observations. It is a dictionary of Gym-supported spaces. The dictionary keys include:
    - "map" (gym.spaces.Box): an $n \times n$ matrix of integers representing the occupancy grid of the map. Table 6 shows the map encoding.
    - "robot_pos" (gym.spaces.Box): a list of two integers representing the robot position (row, col) on the map.
    - "robot_speed" (gym.spaces.Discrete): one integer indicating the speed of the robot.

Table 6: Encoding of map elements

Encoding	Meaning
0	Empty space
1	Obstacle
2	Start position
3	End position

    - "robot_direction" (gym.spaces.Discrete): one integer indicating the facing direction of the robot. See Table 3 for the encoding of directions.
- Methods:
  - __init__: initialize the gym environment
    - Args:
      - n (int): an integer indicating the side length of the map.
      - map_file_path (str): the path to the text file storing the map.
      - start_pos (tuple): the robot's start position (row, col) on the map.
      - goal_pos (tuple): the robot's goal position (row, col) on the map.
      - headless (bool): whether to run the environment in headless mode (without rendering). The default value is False (with rendering).
  - step: execute a given action, transit the robot state, and feed the observation and reward back to the robot.
    - Args:
      - action (Action): an instance of Action class representing the action to be executed.
    - Returns:
      - observation (gym.spaces.Dict): the observation received after executing the action. It follows the format specified in self.observation_space.
      - step_number (int): the current time step.
      - terminated (bool): whether the task has terminated (due to success or failure).
      - is_goal (bool): whether the robot has stopped at the goal.
      - info (dict): a dummy dictionary here, used to match the Gym interface.
  - reset: reset the robot state to the initial configuration and return an observation after resetting.
  - render: render the environment, including the map, the start and goal positions, and the robot.
  - close: stop and quit the rendering window.

3.4.2 Simulating an episode with GridMapEnvCompile

alpha_robot/run_once.py provides a function "run_once" that runs one episode using your policy and the Gym environment defined in the GridMapEnvCompile class.

```
def run_once(map_file_path, policy, start_pos=None, goal_pos=None, store=False,
                                        store_path="~/", step_limit=1000,
                                        time_limit=1.0, headless=False):
    ...
    env = gym.make("grid_map_env/GridMapEnv-v0",  # name of the registered Gym
```

```
                                                environment of the GridMapEnvCompile
                                                class
                    n=100, # load an 100*100 map
                    map_file_path=map_file_path, # location of the map file
                    start_pos=start_pos, # start
                    goal_pos=goal_pos, # goal
                    headless=headless # whether to use rendering
                    )

    initial_observation, _ = env.reset() # Reset the environment
    map = initial_observation["map"] # retrieve the map from the state dictionary
    episode_length = 0

    # construct the initial robot state
    robot_state = RobotState(row=initial_observation["robot_pos"][0],
                             col=initial_observation["robot_pos"][1],
                             speed=initial_observation["robot_speed"],
                             direction=initial_observation["robot_direction"])
    ...
    for _ in range(step_limit): # set step limit
        # run your policy (simplified code with timing thread removed)
        action = policy.get_action(map, robot_state)

        # execute the action in the Gym environment
        observation, curr_step, terminated, is_goal, _ = env.step(action)

        # update the robot state by observation
        robot_state.row = observation["robot_pos"][0]
        robot_state.col = observation["robot_pos"][1]
        robot_state.speed = observation["robot_speed"]
        robot_state.direction = observation["robot_direction"]
        ...
        if terminated: # stop when the task is finished
            episode_length = curr_step
            print("episode terminated! total step number: ", episode_length)
            env.close()
            break

        if not headless:
            env.render() # render the environment
    ...
    return is_goal, episode_length
```

- Args:
  - map_file_path (str): a string indicating the path to the text file storing the map.
  - policy (Policy): the policy to test. It should be an instance of your Policy class.
  - start_pos (tuple): a tuple of 2 integers representing the start position (row, col).
  - goal_pos (tuple): a tuple of 2 integers representing the goal position (row, col). When either the start or goal is set to None, both will be randomly sampled.
  - store (bool): whether to store data in the file system as a JSON file.
  - store_path (str): the path of the file to store the data (if store is set to True).
  - step_limit (int): the maximum number of time steps allowed. No limit if set to None.

- time_limit (float): the thinking time limit (in seconds) for each step. Breaking the time limit leads to immediate failure.
- headless (bool): whether to run the episode in headless mode.

- Returns:
  - is_goal (bool): whether the robot has stopped at the goal.
  - episode_length (int): the episode's total number of time steps.

3.5 How do I collect data using my policy?

Simply call:

```
python run_once.py
```

This command will execute the main function of alpha_robot/run_once.py, which again calls the run_once function with store=True to run one episode and record the data in a .JSON file. The main function looks like:

```
from policy import Policy

MAP_NAME="Wiconisco"  # Collierville, Corozal, Ihlen, Markleeville, or Wiconisco

if __name__ == "__main__":
    # prepare map file name
    current_directory = os.path.dirname(__file__)
    map_file_path = os.path.join(current_directory, "grid_maps", MAP_NAME, "
                                                occ_grid_small.txt") #

    # path for storing data
    store_path = os.path.join(current_directory, "replay_data")

    # prepare the policy
    policy = Policy()

    # run an episode with randomly sampled start and goal, and store the data
    run_once(map_file_path=map_file_path,
             policy=policy,
             store=True,
             store_path=store_path)
```

You can change the following parameters in the script:

```
MAP_NAME = 'Wiconisco' # Collierville, Corozal, Ihlen, Markleeville, or Wiconisco
```

History of the episode will be saved as a .JSON file (named in the format of "data_[DATE]_[TIME]") inside the "replay_data". It consists of the following information:

- map_file_path (str): the map file path used.
- start_pos (tuple): the start position (row, col).
- goal_pos (tuple): the goal position (row, col).
- robot_rows (list): a list of row indices of the robot in all time steps.
- robot_cols (list): a list of column indices of the robot in all time steps.
- robot_speeds (list): a list of robot speeds in all time steps.

- action_accs (list): a list of accelerations in all time steps (except the last).
- action_rots (list): a list of rotations in all time steps (except the last).
- steps (int): the total number of time steps.
- is_goal (bool): whether the robot has stopped at the goal.

Additionally, alpha_robot/run_batch.py provides the function to run a batch of episodes on a given map in headless mode (without rendering) and collect a dataset. Data from each episode will be saved into a separate .JSON file. You can change the following parameters in the script:

```
TASK_NUM = 10 # number of tasks to run
MAP_NAME = 'Wiconisco' # Collierville, Corozal, Ihlen, Markleeville, or Wiconisco
```

After setting the parameters, simply call:

```
python run_batch.py
```

You will see the saved JSON files in the "replay_data" folder.

3.6 How do I evaluate a policy?

alpha_robot/evaluator.py gives an example of evaluating a policy using randomly sampled tasks on the five maps. It mocks our auto-grading script, except that the tasks used for auto-grading are pre-sampled and kept confidential until finishing the submissions. You can control the following parameters in the script:

```
TASK_NUM = 20  # The number of tasks for each map.
RUN_TIME = 10  # The number of times to run each task.
STEP_LIMIT = 1000  # The maximum number of steps allowed for each run.
TIME_LIMIT = 1.0  # The maximum thinking time in seconds for each step.
```

Then you can evaluate your policy just by running the evaluator.py:

```
python evaluator.py
```

Upon finishing all runs, the script will report the SPL score of your policy.

3.7 How do I replay a past episode in my dataset?

alpha_robot/replay.py provides the function to replay a saved episode:

```
python replay.py --data_file /path/to/your/json/file
```

3.8 What if I myself want to control the robot (e.g., for collecting data)?

alpha_robot/keyboard.py contains a "play_with_keyboard" function for you to control the robot yourself. You can execute it and store the episode data by running:

```
python keyboard.py --store True --store_path /path/to/store/folder
```

Encoding of keyboard input:

Table 7: Keyboard command encoding

Keyboard input	**Action of robot**
w	Accelerate by 1
s	Decelerate by 1
a	Turn left
d	Turn right
space	Remain in the direction and retain the speed

4 Submission

You need to submit the following:

- policy_[ID] folder (zip)
  - requirements.txt (the python packages that your program depends on, such as "Scipy", etc..)
  - policy_[ID].py (your implementation of the Policy class)
  - Your own utility scripts (scripts that your Policy class depends on)

5 Tips

- You are obligated to ensure that your program is bug-free and exception-free.
- Please do not touch the files we provide except example scripts.
- We will contact you if and only if your program has dependency issues during the final games.
- If you have any questions or have found any bug in the code base we provide, contact us as soon as possible.

References

[1] P. Anderson, A. Chang, D. S. Chaplot, A. Dosovitskiy, S. Gupta, V. Koltun, J. Kosecka, J. Malik, R. Mottaghi, M. Savva *et al.*, "On evaluation of embodied navigation agents," *arXiv preprint arXiv:1807.06757*, 2018.

Good Luck to All Groups!