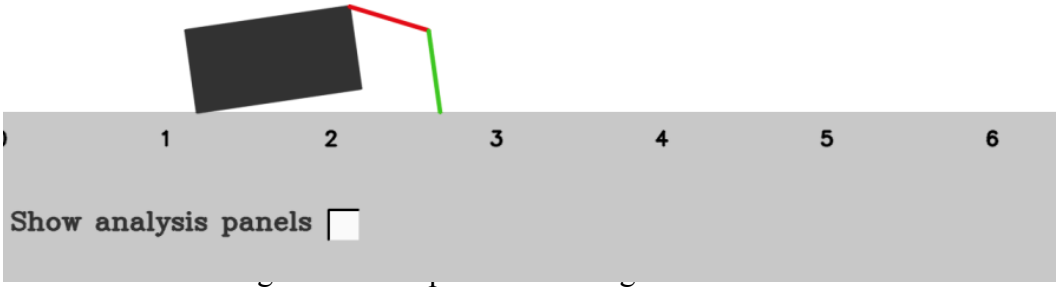
Instruction:

For the *Part I* of this project, you will implement the Temporal Difference (TD) algorithm to achieve the optimal walking for a crawler bot (agent). As the figure shows, the agent can move forward by efficiently using its two arms: top arm (red) and forearm (green).



We form the problem by using two variables to represent a "state": **angle 1** and **angle 2**. Angle 1 is the angle between the top arm and the horizontal line, while angle 2 is the angle between arm 1 and arm 2 as Figure 2 shows:

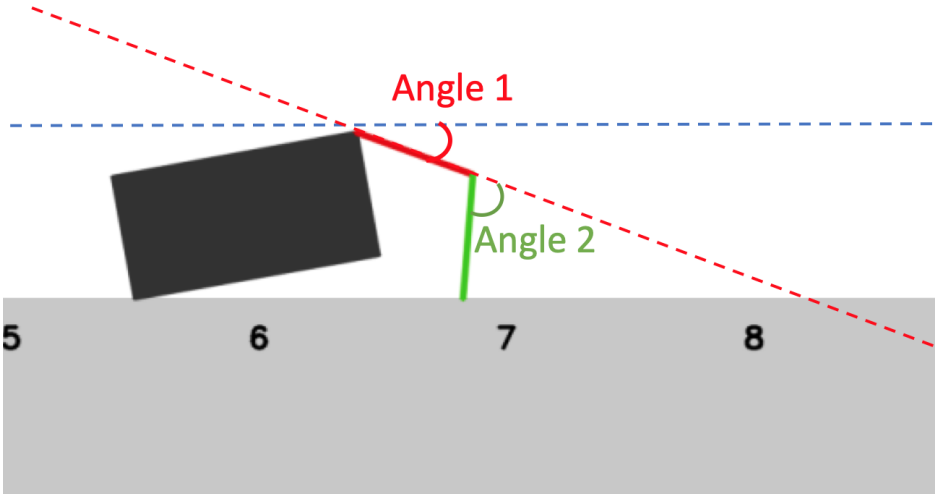


Figure 2. A state is made of two variables

So your task is to find a good policy by using TD-algorithm to let the agent learn how to act efficiently from one state to another.

How does it play?

After installing OpenCV library, you should be able to run the code. The framework allows you to manually moving the crawler's arms by using the W, S, A, D keys to move the arm 1 and arm 2.

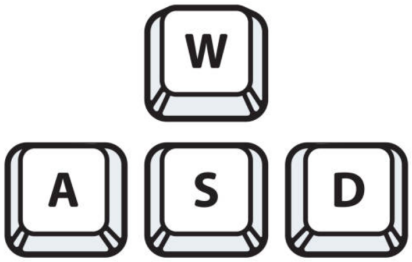
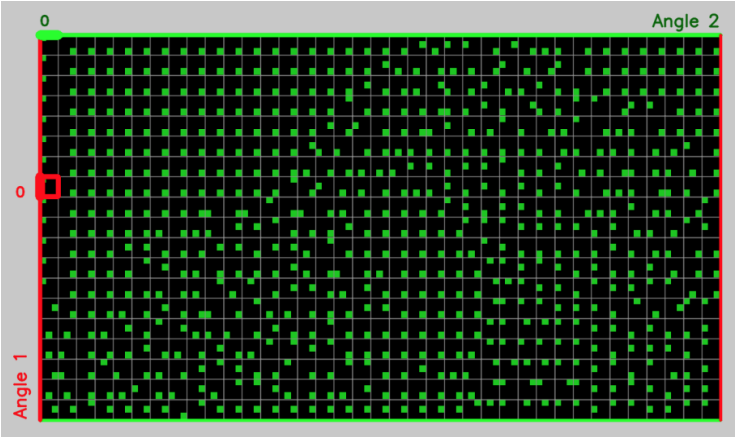


Figure 3. Keys to enable the crawler navigate manually

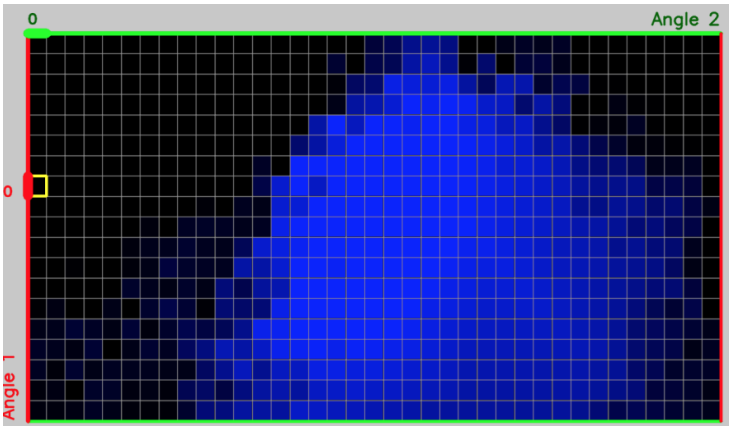
W: decrease angle1 by 5-degrees each time
S: increase angle 1 by 5-degrees each time
A: increase angle 2 by 5-degrees each time
D: decrease angle 2 by 5-degrees each time

In addition to the keys above, you will also use another key often to switch between Q-value and V-value mode to see the analyze your policy learning.

Q: switch between "max Q-value" and "state-value" in the panel. As Figure 4 shows,



(a) max-Q indicator, where the little green square represents the max Q-action for the unit



(b) State value indicator, the brighter of the blue unit, the higher state value of the state

Figure 4. Panel for Q-value and state-value analysis

How do you start coding?

The framework has only two files:

"CrawlerSimulator.py" - (Read-only) It has the GUI and crawler simulator.

"RLearning.py" - (Need your input) Your implementation of TD SARSA should be here.

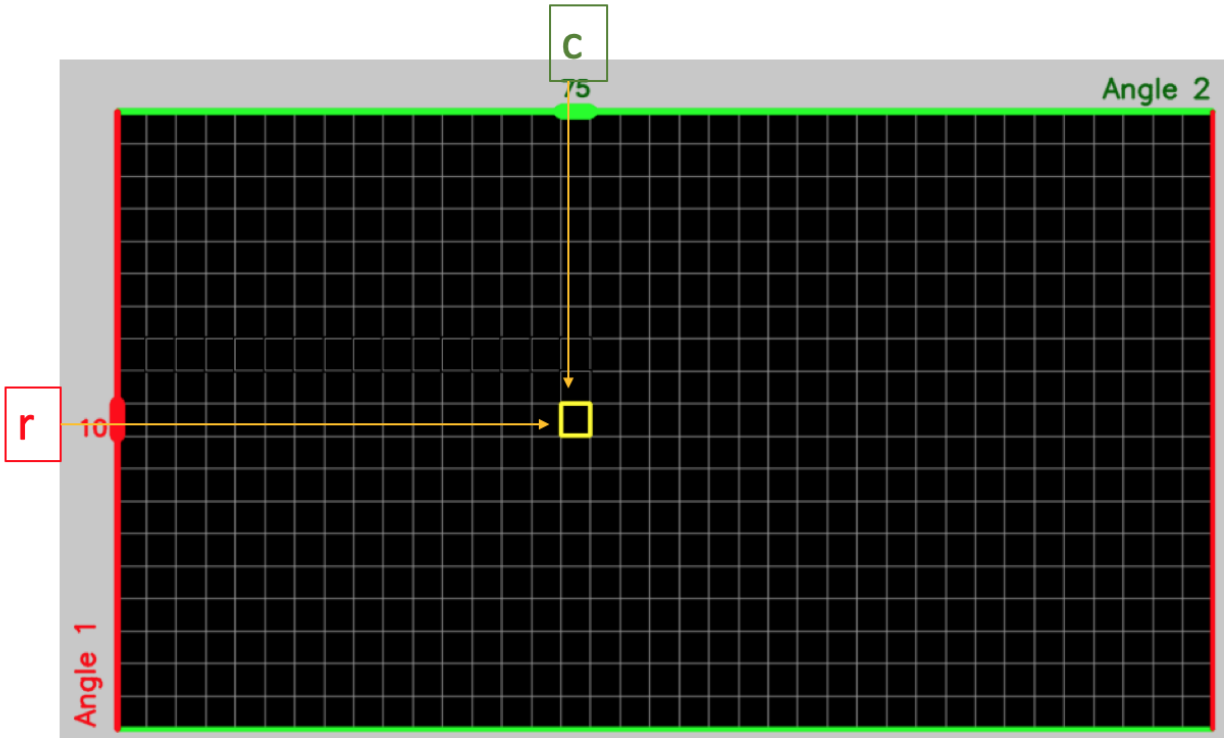
You can find the code of "RLearning.py" by simply using the keyword "***Needed:*** ", which shows in the comment of the code that provides you the information about variables and functions you will need to run the code.

Two functions you will implement:

(1) `def chooseAction(self, r, c):`

This is the function responding to the "play" button of the GUI.

The job of this function is to select an action based on the current state "r, c" using the "ε-greedy" probability model. The two integer variables r and c, indicating the "row" index and "column" index of a 2D array.



As the figure above shows, "r" is the rows counting from the top row. Since the angle 1 is "10 degree". And from the code, you will see that the range of angle1 is [-35, 55]. You can compute it as below:

r = (10 - (-35)) / 5 = 9

Similarly, you can compute the column c based on the angle 2 range [0, 180] as:

c = (75 - 0) / 5 = 15

(2) `def onTDLearning(self):`

This is the second function you have to implement, which performs the "ε-greedy" TD learning to update the Q-value through iterations of randomly generated trajectories. This function will triggered by the "Skip … steps" buttons as the figure shows below. The number of steps is adjustable by using the first trackbar "iterations".

Make sure when you trigger the function, the radio button for "TD SARSA" is selected. It is suggested that you implement the "chooseAction(..)" function first before implementing the "onTDLearning(...)" function, as it can call chooseAction().

Key variables and functions:

For this framework, you will only focus on the "RLearning.py" file only, which contains a few important variables and functions that you will use to achieve the project.

1. self.Qvalue = [] #A 2D list (array) with the dimension 19 x (37x9) as explained below

   This is the most important data for the project. Your TD SARSA algorithm just is just update its value. It represents the Q-value of all the states with 9 actions for each state. Initially, it is assigned as "0.0" for all the actions of all the states. It is a 2D list with the number of rows and columns corresponding to the angle1 and angle2's range:

   self.angle1_range = [-35, 55] #19 rows

   self.angle2_range = [0, 180] #37 colos

   Noted each angle has only a change by +5 or -5 degrees, which is stored in the "self.unit" variable. Also, each state has 9 possible actions, which are lined up in each row. So the actual column is "37 * 9". In other words, every 9 columns represent one state's Q-values.

2. self.crawler

   This "crawler" variable refers to the agent bot. The most important two members are: "self.crawler.angle1" and "self.crawler.angle2"

   These two members represent the two angles described above in Figure 2. If you change the values it will cause the agent to move. But it is not a simple matter of changing angles, which many consider the physic collision detection. So it is not suggested to modify their values directly but instead using the function below.

3. def setBotAngles(self, ang1, ang2):

   This function is used right after you modify the "angle1" and "angle2" above to recalculate the status of the agent in the space. Example:

   angle1 = self.crawler.angle1 # get the current angle 1 of the agent and store it in a temporal variable "angle1"

   angle2 = self.crawler.angle2 # similarly for angle2

   angle1 += self.unit # update the local variable angle1 by increasing 5 degrees

   angle2 -= self.unit # update the local variable angle1 by decreasing 5 degrees

   setBotAngles(self, angle1, angle2) # This function will be a proxy for you to update self.crawler.angle1and self.crawler.angle2 values and do the geometry calculations.

4. After a bot moves, you can get the new location by:
   new_loc = self.crawler.location
   new_loc is a 2D tuple. To access the x value, you can use new_loc[0], since you only consider the x coordinate changes for reward. To compute the trajectory utility or reward, you can record the "initial" location before the trajectory and then get the final location after the trajectory termination:

   ```
   init_loc = self.crawler.location
   for i in range(len(trajectory)):
           …
   old_loc = self.crawler.location
   G = old_loc[0] - init_loc[0]  # The total return of the trajectory. Then you can use it to update the utility for each state that is involved during the trajectory.
   ```

5. In addition to the above variables, there are several other variables that you will need as commented in the code. They are controlled by the "trackbars":

   ```
   # Number of iterations for learning
   self.steps = 1000

   # learning rate alpha (This one is not needed for the "part I" )
   self.alpha = 0.2

   # Discounting factor
   self.gamma = 0.95

   # E-greedy probability
   self.epsilon = 0.1
   ```

Rubrics:

1. Free of compilation error (10%)
2. Correctly using the "self.steps" to repeatedly let the agent learn multiple episodes and generate a random length trajectory for each episode. (25%)
3. Correctly update the Qvalue[] for each state involved during the trajectory by finding the right state and action index within the 2D list. (25%)

4. Correctly implement the “ε-greedy” algorithm for the “chooseAction()” (30%)
5. Provide a document discussing your result that which parts are well implemented and which ones are not. You can also provide some snapshots. (10%)