AI Snake Design

作者：陈利瑄 沈昭正 钱丽玥 井文可 杨贝媛

I. Introduction

贪吃蛇作为一款经典的智力游戏，是很多人小时候的记忆。游戏中玩家通过上下左右控制蛇的方向来寻找食物。在游戏中蛇每吃到一个食物就会长一单位，由于蛇在运动的过程中不能碰墙和自己的身体，因此随着蛇吃到的食物越来越多，蛇的身子越来越长，游戏的难度也越来越大。在我们自己设计的贪吃蛇游戏中，界面中一直保持 5 个食物的存在，蛇每吃掉一个食物，就会随机地另外生成一个食物。

游戏分为两个阶段，第一个阶段为一条蛇自己吃食物，我们分别训练了运用 greedy 算法和 MDP 算法的两条蛇。第二个阶段为两条蛇互相对抗的阶段，一条蛇采用 greedy 算法，一条蛇采用 MDP 算法，两条蛇每走一步，游戏的状态都会被更新，直到一条蛇死亡时，游戏终止。

II. Motivation

选择训练一个 AI agent 来获得贪吃蛇游戏高分作为我们 project 的初衷是其可以和我们课程中所学的 search 算法，MDP 算法，Q-learning 算法等有很好地结合。但在搜索资料的过程中，我们发现对于传统的贪吃蛇游戏，贪吃蛇只要跟随一个固定策略（即哈密顿回路）就能一直吃到食物直至占满整个屏幕获得胜利。所以我们将传统的贪吃蛇游戏进行改进，将简单的一颗食物变为多颗食物，将一条蛇变为两条存在竞争和协作关系的贪吃蛇，从而使得游戏场景变得更加复杂，增加更多可能的状态，提高游戏的难度。

III. Method

我们采用了 Greedy 和 Reinforcing learning 的方法来解决这个问题，但在实现 Reinforcing learning 的过程中我们遇到了一些问题，并提出了用 MDP 来解决这个问题的新思路。

1. Greedy 实现方法

我们首先尝试使用贪心算法来探索贪吃蛇的搜索问题，即在避开敌人、墙和自己身体的情况下向着最近的食物前进。我们定义到食物的最近距离为一个位置到所有食物的曼哈顿距离的最小值，我们使用曼哈顿距离是因为在贪吃蛇游戏中蛇并不能走斜线，所以使用曼哈顿距离比使用欧几里得距离更符合实际情况。我们分别计算蛇的头部上下左右四个位置到食物的最近距离，然后选取值最小的位置作为蛇的头部的下一个位置。

但由于贪吃蛇无法倒着走，所以事实上对于贪食蛇的头部来说只有三个可能的位置。考虑到这种情况，我们规定如果在当前状态下向后是最优的策略，那我们则随机采取向左或者向右的操作，使得贪吃蛇可以向后转以靠近食物。

为了使得贪吃蛇能够避开敌人、墙和自己身体，所以我们将所有的敌人、墙和自己身体的位置设置为不合法的位置，这些位置将不会被纳入且不会计算到食物的最近距离。

2. Q-learning 实现方法

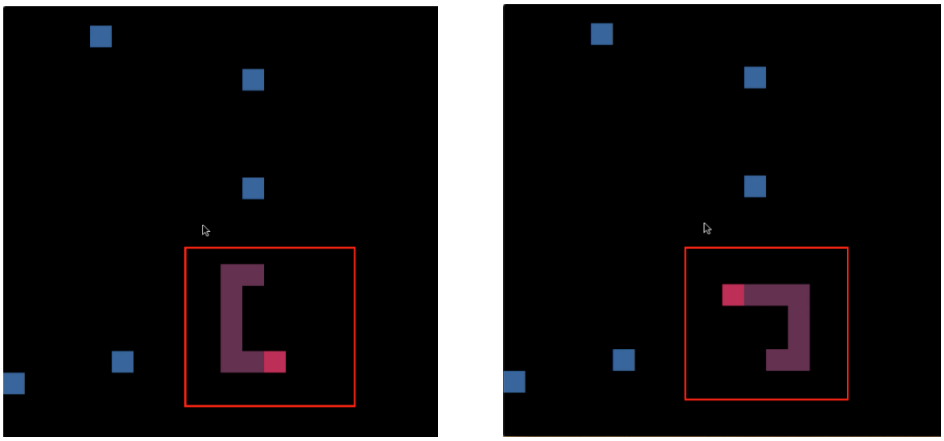
我们采用的第二种实现方法是 Q-learning，尝试通过采取 policy 得到样本，并通过观察得到的结果(reward)来学习，从而试图学会采取能够最大化 reward 的 policy。

$$sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$

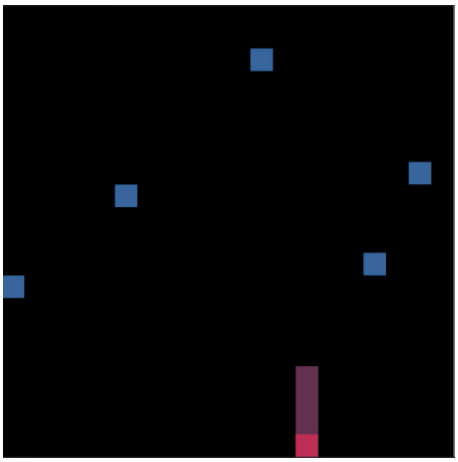
$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[sample]$$

但在实现结果之后我们发现，贪吃蛇总是在局部地区徘徊，而不会 explore 所有的 state；而正因如此，在 train 很多次之后，它还是会出现撞墙或不愿意吃食物的情况。

实现结果如下图所示：



（图一，结果显示，贪吃蛇总是在局部地区徘徊，整个 game state 只是 explore 了一小部分）



（图二，在训练很多次以后，贪吃蛇还是容易出现自杀的情况）

我们分析了其中的问题，总结了其中的问题。由于我们将 state 定义为了{坐标：类型}，如 state = {(x1, y1): snake1.head, (x2, y2): food1.position, ...}，贪吃蛇必须经过所有的 state（即 snake, food 的位置经

历过所有的坐标），其 Q(s, a)才能最终收敛。而我们的贪吃蛇总是在局部地区徘徊，整个 game state 无法充分 explore，这就不能保证 Q(s, a)收敛到真实值。

3. MDP 实现方法

在上述 Q-learning 方法中，game state 过于复杂导致难以充分 explore 所有情况，在这种情况下，我们想到了另一种方式来解决这个问题。

考虑到现在的游戏情况，转移概率和奖惩制度都是已知的，因此我们选择了一种 offline 的方式。通过这种 offline 的方式，我们可将坐标作为 state 来储存 Q(s, a)，这解决了 Q-learning 因状态空间过大而无法充分探索和储存的问题。

$$Q_{k+1}(s,a) \leftarrow R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

在游戏开始前设定好食物、墙以及两条蛇坐标处的 reward，然后以循环迭代的方式在蛇采取行动之前计算好整个游戏空间每一个坐标处的四个 Q(s, a)，然后通过比较与蛇的头部相邻的四个位置的 Q(s, a)来确定下一个位置。

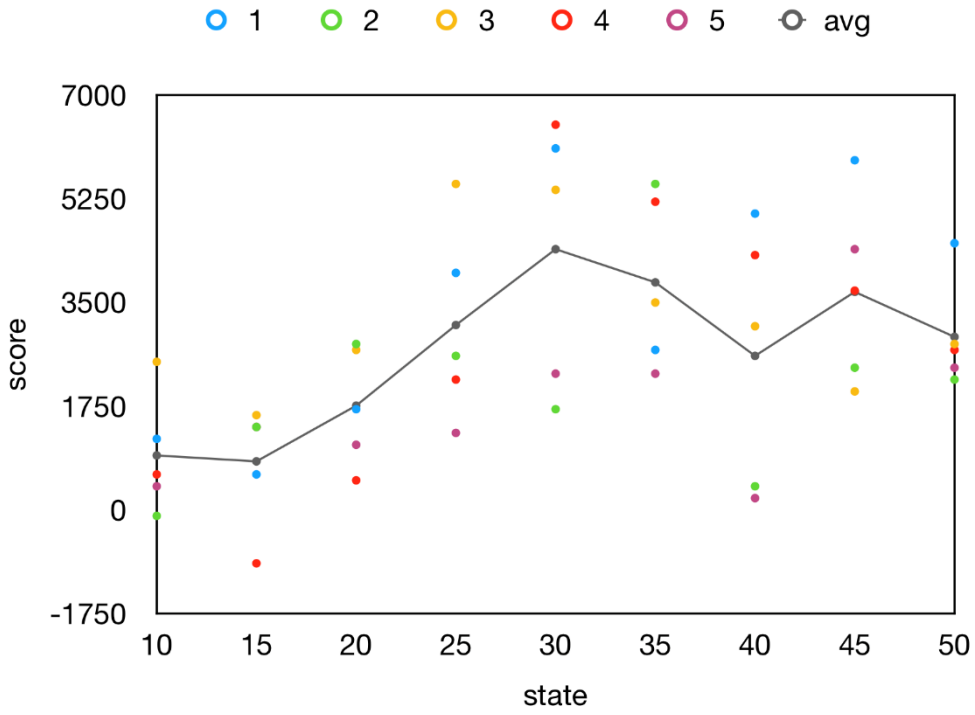
然而这种做法也有一定的不足之处。在游戏空间很大的时候，我们在每采取一个行动前都要计算所有的 Q(s, a)值。如果我们用 m 来表示游戏空间的横坐标，n 为纵坐标，k 是迭代的循环次数，那每一步的移动的复杂度为 O(4kmn)，计算量与前面的方法相比有显著的增大，可能会出现蛇在采取行动时动作缓慢的问题。

IV. Experiment

以下是我们根据上述 method 进行大量实验得出的结果。

1. 对 Greedy 的结果分析

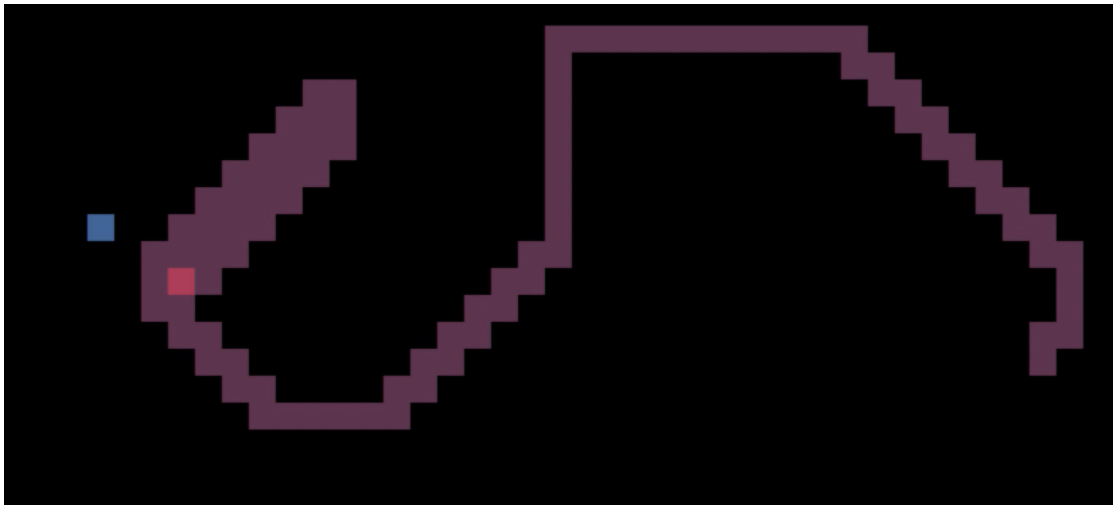
我们通过改变 state 的 size 来调整 state space，我们以 5 为间隔统计了从 10x10 ~ 50x50 之间的通过 Greedy 算法得到的贪吃蛇的分数，同时我们通过记录使用 greedy 算法计算一个 action 的时间来衡量我们的算法时间复杂度，对于每一个间隔我们实验了五次并取其平均。实验结果如图三和图五所示：



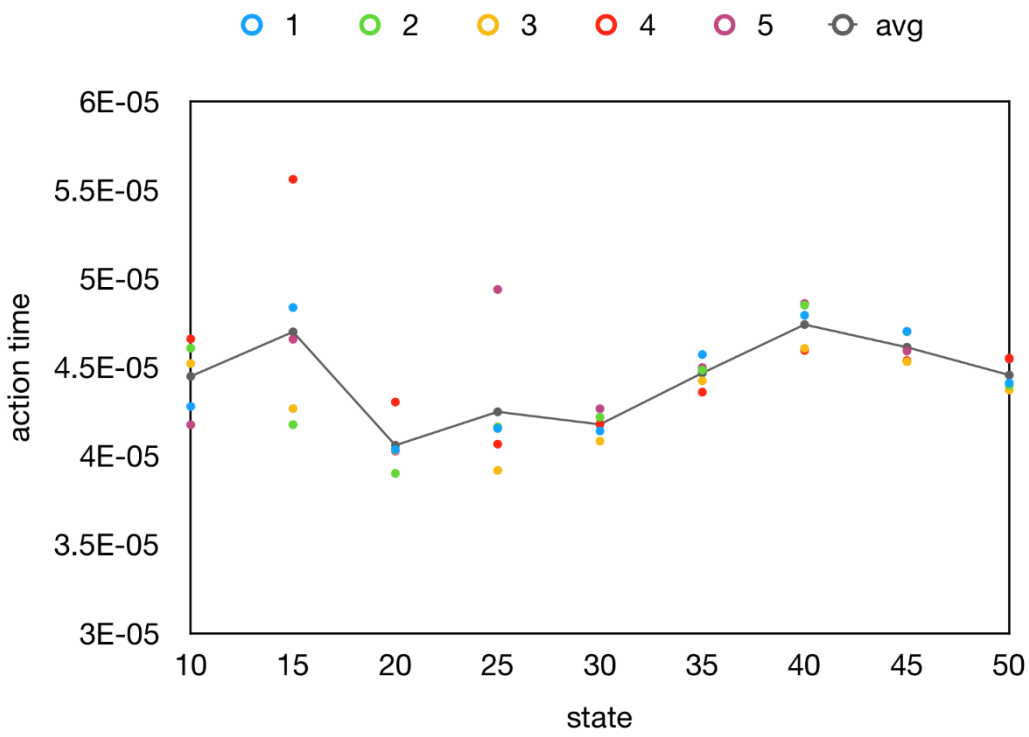
（图三显示了 state 复杂度和 score 之间的关系）

通过图三可以看到，当 state size 比较小的时候，贪吃蛇比较容易撞到自己的身体，所以这个时候的 score 比较小；当 state size 逐渐变大的时候，贪吃蛇有了更多的活动空间，score 也随之升高，当 state size 增大到一定程度的时候，state size 几乎不会限制贪吃蛇的活动，主要限制贪吃蛇分数的是贪

吃蛇的头部会比较容易被自己的身体围住（如图四所示），所以分数不会一直上升。因为贪吃蛇的分数受到运气的影响比较大，所以分数最后会维持在一个区间中。



（图四显示了贪吃蛇的头部会比较容易被自己的身体围住）

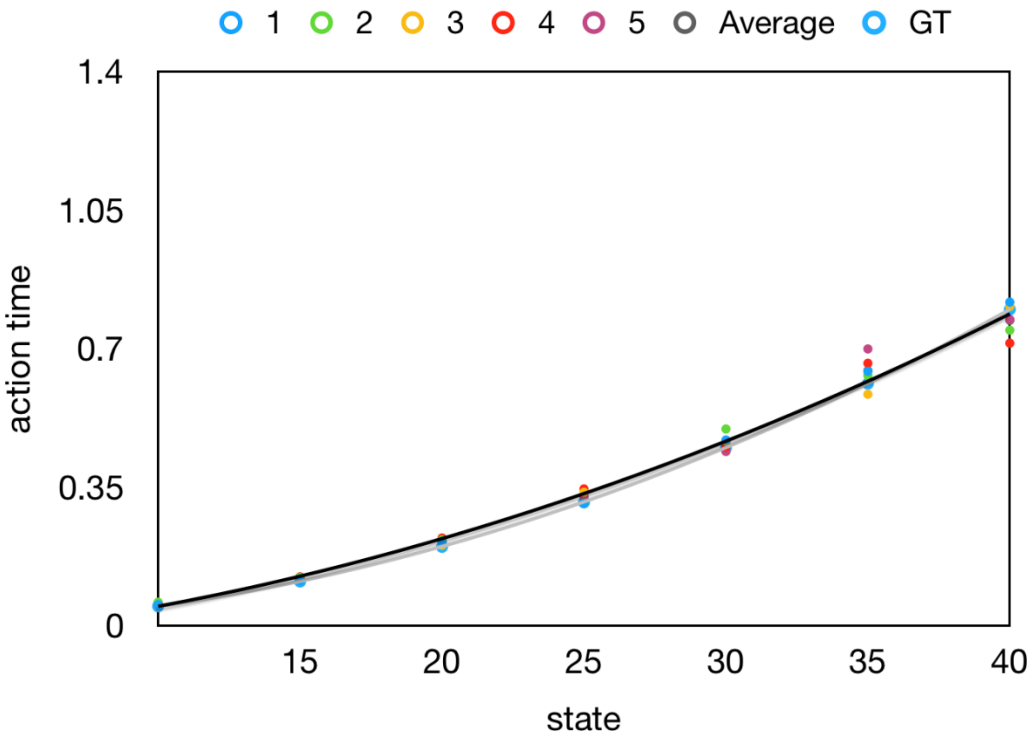


（图五显示了 state 复杂度和 time per action 之间的的关系）

而从图五中我们可以看到，greedy 算法的时间复杂度和 state size 并无大的关系，随着 state size 的增大，greedy 算法的时间复杂度变化始终维持在一个很小的区间内。

2. 对 MDP 的结果分析

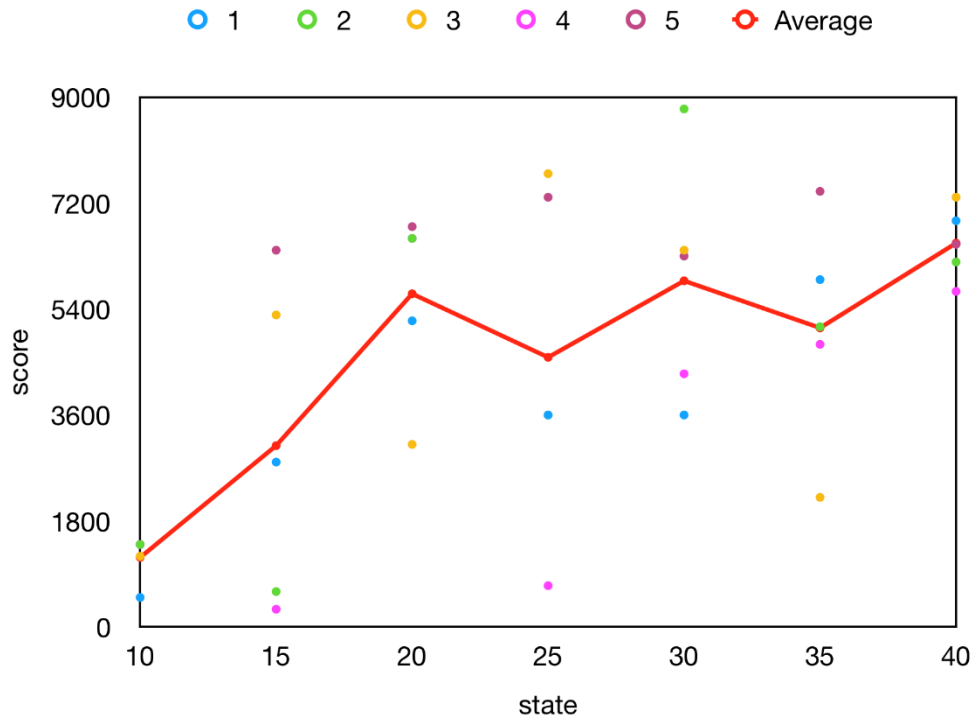
我们通过改变 state 的 size 来调整 state space，我们以 5 为间隔统计了从 10x10～40x40 之间的通过 MDP 算法得到的贪吃蛇的分数，同时记录了决策一个 action 的时间来衡量我们的算法复杂度，实验结果如图六和图七所示：



(图六 action time 随 state 的变化图表)

从分数与 state 的图表（图六）中我们可以看出，在 state 比较小时，随着 state 的增长，MDP 蛇的分数是有增长趋势的。state 的增长意味着蛇的活动范围的增大，所以它活的时间会相应增加。然而随着 state 的继续增大，蛇

越来越长，它把自己绕进去的可能性也越来越大，于是蛇的分数就稳定在了一个区间。



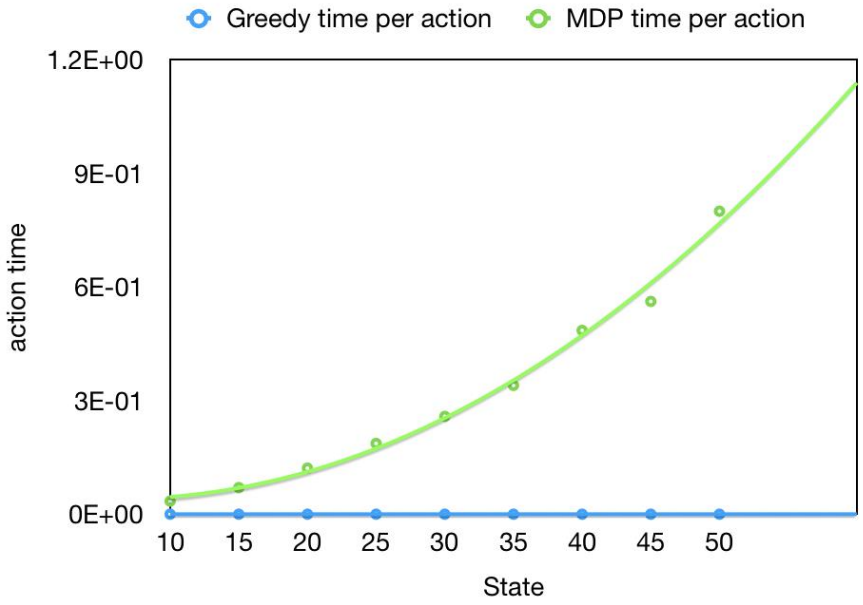
(图七 score 随 state 的变化图表)

从算法复杂度分析图表（图七）中我们可以看出，随着 state 的增长，MDP 蛇采取下一步行动的耗时是增加的。经过拟合，我们发现 action time 随 state 边长呈现二项式增长变化，所以当 state 增大到一定大小之后，贪吃蛇选取 action 变得非常耗时，所以我们只选取到 state space = 40x40 来记录数据。耗时的原因在于，我们每次采取行动前需要计算整个 state 的所有 Q(s, a) 值，state 越大需要计算的值就越多，耗时也就越长。

我们可以看出，MDP 算法的表现还是不错的，不过由于 state size 大的时候，每计算一步 action 耗时太大，这种方法不适用于实际应用中。

3. 对两条蛇对抗结果的分析

图八、图九是两条蛇（greedy 与 MDP）相互对抗的结果。



图八：比较不同state下greedy蛇和MDP蛇每采取一步动作所需时间

图八是我们以 5 为间隔统计了从 10x10 ~ 50x50 之间 greedy 蛇和 MDP 蛇每采取一次 action 所需的时间。可以看出，MDP 蛇每采取一步所需的时间远远高于 greedy 蛇，其采取一个 action 所需时间与 greedy 蛇至少相差三个数量级。这符合我们之前的预测，因为在我们的代码实现中，greedy 蛇每采取一步的时间复杂度是 O(1)，而 MDP 蛇每采取一步所需的时间复杂度为 O(4kmn)，其中 mn 表示图表是 mxn 的网格，4 是因为每次要考虑蛇往 4 个方向，k 表示每次要循环 k 次才能算出最终的 Q(s, a)值。

图九：不同state下greedy蛇和MDP蛇的分数

图九是我们以 5 为间隔统计了从 10x10 ~ 50x50 之间两条蛇相互对抗，每次对最终的得分取平均值后作出的图。在 20x20 的网格中，我们一共测试了 35 组数据（如下表所示），其中 greedy 赢了 18 次，mdp 赢了 15 次，平局 2 次。两种算法优势相当。综合其他大小的网格，greedy 算法稍微胜出一点。

Win(MDP better)	Loss(greedy better)	Tie
15	18	2

V. Future Work

通过本次的课题，我们对贪吃蛇的探索算法有了一定的了解，但是在这个过程中也暴露出了一系列问题，以下是我们小组在这个项目中接下来的思考研究方向。

1. 在之前的探索中，我们发现 Q-learning 在解决贪吃蛇问题中暴露出了蛇会倾向于在界面中心区域探索和容易撞墙的问题，这是由于 state 太大，蛇在短时间内无法充探索导致的。为了解决此问题，我们希望将算法换成 MDP 或者 approximate Q-learning 来进行尝试。在上述探索中，我们发现 MDP 出现了时间复杂度过高，在游戏域变大时效果很差，不适用于实际情况等问题。因此，我们决定采用 approximate Q-learning 来进行下一步尝试，这种基于 feature 的探索减少了 state 的可能情况，减少充分探索的时间。
2. 在 greedy 的实现中，我们希望蛇在避开敌人、墙和自己身体的情况下向着最近的食物前进。但是只遵循这个原则会出现将自己绕入死胡同、食物在身体绕城的圈里椅子找不到路线等问题。由于我们知道，当蛇一直追着自己尾巴前行时，蛇可以确保活着。因此我们的未来的改进方向是，在蛇确保可以找到到自己尾巴的路径再去吃食物，以增大蛇的存活率。

VI. Code Implementation

我们代码的 Github 链接为： https://github.com/catnip0823/AI-project，且我们的具体代码如下：

Greedy implementation

```
def mainGreedy(world_state, food, snake2):
    # 定义行动方向
  actions = {K_LEFT, K_RIGHT, K_DOWN, K_UP}
    # 参数设置
    snake_head = snake2.head
    snake_body = snake2.body
    headx = snake_head[0]/LEN
    heady = snake_head[1]/LEN
    food_pos = food.pos
    distance = 999999
    action = None
    unvalid = ['enemy_head', 'my_body', 'enemy_body', 'wa
ll']


    # 避免撞墙、撞到自己、撞到其他蛇的问题
    if world_state.state[(max(0,headx-
1),heady)] in unvalid:
        actions.discard(K_LEFT)
    if world_state.state[(min(headx+1,COLNUM-
1),heady)] in unvalid:
        actions.discard(K_RIGHT)
    if world_state.state[(headx,min(heady+1,ROWNUM-
1))] in unvalid:
        actions.discard(K_DOWN)
    if world_state.state[(headx,max(0,heady-
1))] in unvalid:
```

```
        actions.discard(K_UP)

    for i in actions:
        # 遍历食物以寻找离蛇距离最近的食物
        if i == K_LEFT:
            for x2,y2 in food_pos:
                temp = cul_dis(headx-,heady,x2/LEN,y2/LE)
                if(temp < distance):
                    distance = temp
                    action = K_LEFT
        elif i == K_RIGHT:
            for x2,y2 in food_pos:
                temp = cul_dis(headx+1,heady,x2/LEN,y2/LEN)
                if(temp < distance):
                    distance = temp
                    action = K_RIGHT
        elif i == K_DOWN:
            for x2,y2 in food_pos:
                temp = cul_dis(headx,heady+1,x2/LEN,y2/LEN)
                if(temp < distance):
                    distance = temp
                    action = K_DOWN
        elif i == K_UP:
            for x2,y2 in food_pos:
                temp = cul_dis(headx,heady-1,x2/LEN,y2/LEN)
                if(temp < distance):
                    distance = temp
                    action = K_UP
    # 当蛇找到最近的食物在其后方的时候，先随机向左、右转弯，再向后走
        if snake2.head[0] == snake2.body[0][0]:
            if snake2.head[1] < snake2.body[0][1]:
                if action == K_DOWN:
```

```
                    temp_actions = actions.copy()
                    temp_actions.discard(K_DOWN)
                    temp_actions.discard(K_UP)
                    if temp_actions:
                        action = random.sample(temp_actio
ns,1)[0]
            else:
                if action == K_UP:
                    temp_actions = actions.copy()
                    temp_actions.discard(K_DOWN)
                    temp_actions.discard(K_UP)
                    if temp_actions:
                        action = random.sample(temp_actio
ns,1)[0]
        if snake2.head[1] == snake2.body[0][1]:
            if snake2.head[0] < snake2.body[0][0]:
                if action == K_RIGHT:
                    temp_actions = actions.copy()
                    temp_actions.discard(K_RIGHT)
                    temp_actions.discard(K_LEFT)
                    if temp_actions:
                        action = random.sample(temp_actio
ns,1)[0]
            else:
                if action == K_LEFT:
                    temp_actions = actions.copy()
                    temp_actions.discard(K_RIGHT)
                    temp_actions.discard(K_LEFT)
                    if temp_actions:
                        action = random.sample(temp_actio
ns,1)[0]
    return action
```

MDP implementation

```
class MDP:
  def __init__(self, snake1, snake2, food):
      self.mdp_learning_state = {}
      for i in range(COLNUM):
          for j in range(ROWNUM):
              for k in range(4):
                 self.mdp_learning_state[((i, j), k)] = 0
      for i in range(-1, COLNUM+1):
          for k in range(4):
              self.mdp_learning_state[((-1, i), k)] = -
99999999
              self.mdp_learning_state[((ROWNUM, i), k)] =
 -99999999
      for i in range(-1, ROWNUM+1):
          for k in range(4):
              self.mdp_learning_state[((i, -1), k)] = -
99999999
              self.mdp_learning_state[((i, COLNUM), k)] =
 -99999999

    # Reward
    def get_reward(self, state, action, food, snake1, sna
ke2):
        next_state = self.get_next_state(state, action)
        next_state = [LEN*next_state[0], LEN*next_state[1
]]
        if next_state in food.pos:
            return 10000
        if next_state[0] < 0 or next_state[1] < 0 or next
_state[0] > LEN*COLNUM-LEN or next_state[1] > LEN*ROWNUM-
LEN:
            return -100000
        if next_state in snake1.body:
```

```
            return -100000
        if next_state in snake2.body:
            return -100000
        if next_state in snake1.head:
            return -50000
        return 0


    # 退出条件判断
  def judge_exit(self, state, action, food):
      next_state = state
      next_state = [LEN*next_state[0], LEN*next_state[1]]
      if next_state in food.pos:
          return -1
      if next_state[0] == -LEN or next_state[1] == -
LEN or next_state[0] == LEN*COLNUM or next_state[1] == LE
N*ROWNUM:
          return -1
      return 0
    # iteration
  def iteration(self, food, snake1, snake2):
      for k in range(20):
          previos_state = self.mdp_learning_state
          for state, action in self.mdp_learning_state.ke
ys():
              reward = self.get_reward(state, action, foo
d, snake1, snake2)
              next_state = self.get_next_state(state, act
ion)
              if (next_state[0] >= - and next_state[1] >=
 - and next_state[0] <= COLNUM and next_state[1] <= ROWNU
M):
                  max_q_next = self.get_max_q(next_state,
previos_state)
```

```
                    self.mdp_learning_state[(state, action)
] = reward + DF * max_q_next
    # Next State
     def get_next_state(self, state, action):
         if action == 0: # up
             state = (state[0], state[1] - 1)
         if action == 1: # down
             state = (state[0], state[1] + 1)
         if action == 2: # left
             state = (state[0] - 1, state[1])
         if action == 3: # right
             state = (state[0] + 1, state[1])
         return state


   # 得到最大的 Q-value 值
  def get_max_q(self, state, previos_state):
      ret_value = None
      for i in range(4):
          if ret_value == None:
              ret_value = previos_state[(state, i)]
          else:
              ret_value = max(ret_value, previos_state[(s
tate, i)])
      return ret_value


    # Policy
  def get_policy(self, start_state, food, last_action):
      start_state = (start_state[0]//LEN, start_state[1]/
/LEN)
      policy_list = []
      for _ in range(100):
          policy = None
          max_value = -999999999999
```

```
            for i in range(4):
                if policy_list != []:
                    if policy_list[-1] == 0 and i == 1:
                        continue
                    if policy_list[-1] == 1 and i == 0:
                        continue
                    if policy_list[-1] == 2 and i == 3:
                        continue
                    if policy_list[-1] == 3 and i == 2:
                        continue
                elif last_action != None:
                    if last_action == 0 and i == 1:
                        continue
                    if last_action == 1 and i == 0:
                        continue
                    if last_action == 2 and i == 3:
                        continue
                    if last_action == 3 and i == 2:
                        continue
                if max_value < self.mdp_learning_state[(tup
le(start_state), i)]:
                    max_value = self.mdp_learning_state[(tu
ple(start_state), i)]
                    policy = i
            return policy
            policy_list.append(policy)
            if self.judge_exit(tuple(start_state), policy,
food) != 0:
                break

            start_state = self.get_next_state(tuple(start_s
tate), policy)
            print('start state', start_state)
```

```
        return policy_list
```

Q-learning implementation

```
class Q_learning:
    def __init__(self):
        self.Q = {}
   # Reward
    def get_reward(self, food, snake2):
        ret_value = 0
        for i in food.pos:
            ret_value += abs(i[0] - snake2.head[0]) + abs
(i[1] - snake2.head[1])
        return ret_value
   # Predict
    def next_action(self, state):
        self.prevstate = state
        if(state not in self.Q.keys()):
            self.Q[state] = np.array([0, 0, 0, 0])
          random_action = np.random.rand(4)  ##only one s
tate
          self.action = np.argmax(random_action)
          return self.action
      self.action = np.argmax(self.Q[state])
      return self.action
  # Update
  def update(self, reward, state):
      if(state not in self.Q.keys()):
          self.Q[state] = np.array([0, 0, 0, 0])
      self.Q[self.prevstate][self.action] = LF * (reward
+ DF * np.max(self.Q[state]))
```

State declaration

```
class State():
    def __init__(self, snake1, snake2, food):
        self.state = {}
        self.q_learning_state = []

  # 定义每个位置对应的状态
    for i in range(-1,COLNUM+1):
        for j in range(-1,ROWNUM+1):
            if [LEN*i, LEN*j] == snake1.head:
                self.state[(i, j)] = 'my_head'
                self.q_learning_state.append(((i, j), 'my
_head'))
            elif [LEN*i, LEN*j] == snake2.head:
                self.state[(i, j)] = 'enemy_head'
                self.q_learning_state.append(((i, j), 'en
emy_head'))
            elif [LEN*i, LEN*j] in snake1.body:
                self.state[(i, j)] = 'my_body'
                self.q_learning_state.append(((i, j), 'my
_body'))
            elif [LEN*i, LEN*j] in snake2.body:
                self.state[(i, j)] = 'enemy_body'
                self.q_learning_state.append(((i, j), 'en
emy_body'))
            elif [LEN*i, LEN*j] in food.pos:
                self.state[(i, j)] = 'food'
                self.q_learning_state.append(((i, j), 'fo
od'))
            elif i == -1:
                self.state[(i, j)] = 'wall'
                self.q_learning_state.append(((i, j), 'wa
ll'))
            elif i == COLNUM:
```

```
                self.state[(i, j)] = 'wall'
                self.q_learning_state.append(((i, j), 'wa
ll'))
            elif j == -1:
                self.state[(i, j)] = 'wall'
                self.q_learning_state.append(((i, j), 'wa
ll'))
            elif j == ROWNUM:
                self.state[(i, j)] = 'wall'
                self.q_learning_state.append(((i, j), 'wa
ll'))
            else:
                self.state[(i, j)] = 'None'
    self.q_learning_state = tuple(self.q_learning_state)
  # 得到状态
   def get_state(self, food, snake1, snake2):
        for i in range(5):
            x = food.pos[i-1]
        y_1 = snake1.head
        z_1 = snake1.body
        if y_1[0] == z_1[0][0]:
            if y_1[1] > z_1[0][1]:
                direction_1 = 3
            if y_1[1] < z_1[0][1]:
                direction_1 = 1
        if y_1[1] == z_1[0][1]:
            if y_1[0] > z_1[0][0]:
                direction_1 = 0
            if y_1[0] < z_1[0][0]:
                direction_1 = 2
        y_2 = snake2.head
        z_2 = snake2.body
        if y_2[0] == z_2[0][0]:
```

```
            if y_2[1] > z_2[0][1]:
                direction_2 = 3
            if y_2[1] < z_2[0][1]:
                direction_2 = 1
        if y_2[1] == z_2[0][1]:
            if y_2[0] > z_2[0][0]:
                direction_2 = 0
            if y_2[0] < z_2[0][0]:
                direction_2 = 2
        size = LEN
        return (x[0] <= y_2[0], x[1] >= y_2[1], x[0] == y
_2[0], x[1] == y_2[1], direction_1, direction_2, [x[0] -
size, x[1]] in z_2, [x[0] + size, x[1]] in z_2, [x[0], x[
1] - size] in z_2, [x[0], x[1] + size] in z_2)
```

Food Declaration

```
class Food():
    #对食物位置的初始化
    def __init__(self, screen):
        self.pos = []
        for i in range(FOOD_NUM):
            self.pos.append([])
            self.pos[i]=[random.randint(0, COLNUM-
1)*LEN,random.randint(0, ROWNUM-1)*LEN]
        self.screen = screen

    #通过 pygame 库将食物画出
    def draw_food(self):
        for i in range(FOOD_NUM):
            pygame.draw.rect(self.screen,FOOD_COLOR,(self
.pos[i][0],self.pos[i][1],LEN,LEN))
    #在某一食物被吃时，随机生成另一食物
```

```
    def be_eatten(self, index):
        self.pos[index]=[random.randint(0, COLNUM-
1)*LEN,random.randint(0, ROWNUM-1)*LEN]
```