**大 连 海 事 大 学**

**大数据原理与技术实验报告**

**学生： 杨显鹏**

**学号： 1120211433**

**专业： 电子信息**

**日期： 2021.11.12**

# 游戏介绍

贪食蛇（也叫贪吃蛇）是一款经典的小游戏。初始是像素版本，后来又衍生出3D版本、多人对战版本等。

玩家使用方向键操控一条长长的蛇不断吞下豆子，同时蛇身随着吞下的豆子不断变长，当蛇头撞到蛇身或障壁时游戏结束。贪吃蛇最初为人们所知的是诺基亚手机附带的一个小游戏，它伴随着诺基亚手机走向世界。现在的贪吃蛇出现了许多衍生版本，并被移植到各种平台上。

本游戏共有两方，每个玩家分别在n\*m的网格中操纵三条蛇。n为垂直高度，m为水平宽度。蛇是一系列坐标构成的有限不重复有顺序的序列，序列中相邻坐标均相邻，即两坐标的x轴坐标或y轴坐标相同，序列中第一个坐标代表蛇头。玩家只能通过控制蛇头的朝向(上、下、左和右)来控制蛇。

蛇以恒定的速度前进(前进即为序列头插入蛇头指向方向下一格坐标，并删除序列末尾坐标)。蛇的初始位置为网格中的随机位置，初始长度为3格。每吃掉一个豆子长度加1。在游戏中豆子数量维持一个常数（5）不变，即被吃掉则重新随机生成等数量的豆子。

蛇头在自己蛇的身体(即序列重复)、其他蛇的身体(即与其他序列有相同坐标)均判定为死亡。当输入了非法方向时(如正在向左前进，无法选择向右)，随机选择一个合法方向。与传统贪吃蛇不同的是，蛇头超过边界时可以穿越到对称位置，不算做死亡。蛇死亡时可以按照初始化的设置重生。

到达指定步数时游戏结束。游戏结束后，蛇身长度总和大的一方获胜。

# 算法逻辑

本算法会依次判断给定条件是否满足, 从而做出下列五个动作之一:

1. defensive move: 衔尾

2. claim move: 以最短路径吃豆子

3. safe move: 最大化生存空间

4. attack move: 侵略性的动作

5. random move: 没有操作空间, 随机动作

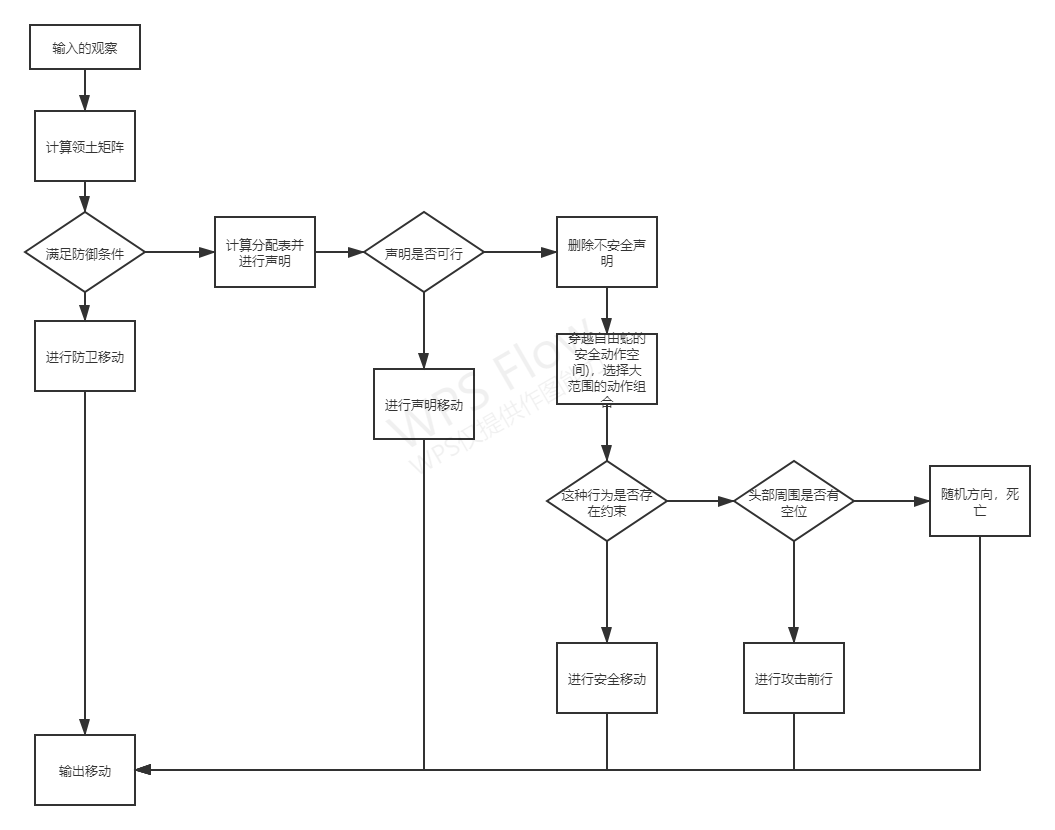


图1

计算领土矩阵的一个例子如图1所示. 简单说明如下:

领土的申明由两个循环构成:

外循环：对每条蛇, 维护一个待探列表, 外循环直至该列表为空才结束, 比如在循环开始时, 该列表即为六条蛇的蛇头.

每次循环开始首先去掉蛇尾 (若蛇尾存在的话), 这样做的目的是为了使得领土的计算更加精确, 否则将有很多蛇身未被申明.

内循环：即每条蛇按顺序申明领土, 比如下面的例子的顺序为:深绿→浅绿→青→红→粉→褐.

同一层外循环中, 短的蛇覆盖长的蛇的领土, 这可以保证短的蛇做更多攻击性的动作, 而长的蛇趋向防守.

同一层外循环中, 若蛇长度相同, 则将领土视为有争议的区域, 均放入待探列表, 并存储信息放入争议列表.

同一层外循环中, 若短的蛇可以申明一块争议区域, 且产生争议的两条蛇的长度都比其大, 则短的蛇直接申明该点, 并将该点移出争议列表。

# 代码

submission.py

class HiddenPrints:

def \_\_enter\_\_(self):

self.\_original\_stdout = sys.stdout

sys.stdout = open(os.devnull, 'w')

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

sys.stdout.close()

sys.stdout = self.\_original\_stdout

def get\_direction(x\_h, y\_h, x, y, height, width): # from (x\_h, y\_h) to (x, y)

if (x\_h + 1) % height == x and y\_h == y:

return [0, 1, 0, 0]

elif (x\_h - 1) % height == x and y\_h == y:

return [1, 0, 0, 0]

elif x\_h == x and (y\_h + 1) % width == y:

return [0, 0, 0, 1]

elif x\_h == x and (y\_h - 1) % width == y:

return [0, 0, 1, 0]

else:

assert False, 'the start and end points do not match'

def connected\_count(matrix, pos):

height, width = matrix.shape

x, y = pos

sign = matrix[x, y]

unexplored = [[x, y]]

explored = []

\_connected\_count = 1

while unexplored:

x, y = unexplored.pop()

explored.append([x, y])

for x\_, y\_ in [((x + 1) % height, y), # down

((x - 1) % height, y), # up

(x, (y + 1) % width), # right

(x, (y - 1) % width)]: # left

if matrix[x\_, y\_] == sign and [x\_, y\_] not in explored and [x\_, y\_] not in unexplored:

unexplored.append([x\_, y\_])

\_connected\_count += 1

return \_connected\_count

class Snake:

def \_\_init\_\_(self, snake\_positions, board\_height, board\_width, beans\_positions):

self.pos = snake\_positions # [[2, 9], [2, 8], [2, 7]]

self.len = len(snake\_positions) # >= 3

self.head = snake\_positions[0]

self.beans\_positions = beans\_positions

self.claimed\_count = 0

displace = [(self.head[0] - snake\_positions[1][0]) % board\_height,

(self.head[1] - snake\_positions[1][1]) % board\_width]

print('creat snake, pos: ', self.pos, 'displace:', displace)

if displace == [board\_height - 1, 0]: # all action are ordered by left, up, right, relative to the body

self.dir = 0 # up

self.legal\_action = [2, 0, 3]

elif displace == [1, 0]:

self.dir = 1 # down

self.legal\_action = [3, 1, 2]

elif displace == [0, board\_width - 1]:

self.dir = 2 # left

self.legal\_action = [1, 2, 0]

elif displace == [0, 1]:

self.dir = 3 # right

self.legal\_action = [0, 3, 1]

else:

assert False, 'snake positions error'

positions = [[(self.head[0] - 1) % board\_height, self.head[1]],

[(self.head[0] + 1) % board\_height, self.head[1]],

[self.head[0], (self.head[1] - 1) % board\_width],

[self.head[0], (self.head[1] + 1) % board\_width]]

self.legal\_position = [positions[\_] for \_ in self.legal\_action]

def get\_action(self, position):

if position not in self.legal\_position:

assert False, 'the start and end points do not match'

idx = self.legal\_position.index(position)

return self.legal\_action[idx] # 0, 1, 2, 3: up, down, left, right

def step(self, legal\_input):

if legal\_input in self.legal\_position:

position = legal\_input

elif legal\_input in self.legal\_action:

idx = self.legal\_action.index(legal\_input)

position = self.legal\_position[idx]

else:

assert False, 'illegal snake move'

self.head = position

self.pos.insert(0, position)

if position in self.beans\_positions: # eat a bean

self.len += 1

else: # do not eat a bean

self.pos.pop()

class Board:

def \_\_init\_\_(self, board\_height, board\_width, snakes, beans\_positions, teams):

print('create board, beans\_position: ', beans\_positions)

self.height = board\_height

self.width = board\_width

self.snakes = snakes

self.snakes\_count = len(snakes)

self.beans\_positions = beans\_positions

self.blank\_sign = -self.snakes\_count

self.bean\_sign = -self.snakes\_count + 1

self.board = np.zeros((board\_height, board\_width), dtype=int) + self.blank\_sign

self.open = dict()

for key, snake in self.snakes.items():

self.open[key] = [snake.head] # state 0 open list, heads, ready to spread

# see [A\* Pathfinding (E01: algorithm explanation)](https://www.youtube.com/watch?v=-L-WgKMFuhE)

for x, y in snake.pos:

self.board[x][y] = key # obstacles, e.g. 0, 1, 2, 3, 4, 5

# for x, y in beans\_positions:

# self.board[x][y] = self.bean\_sign # beans

self.state = 0

self.controversy = dict()

self.teams = teams

print('initial board')

print(self.board)

def step(self): # delay: prevent rear-end collision

new\_open = {key: [] for key in self.snakes.keys()}

self.state += 1 # update state

# if self.state > delay:

# for key, snake in self.snakes.items(): # drop tail

# if snake.len >= self.state:

# self.board[snake.pos[-(self.state - delay)][0]][snake.pos[-(self.state - delay)][1]] \

# = self.blank\_sign

for key, snake in self.snakes.items():

if snake.len >= self.state:

self.board[snake.pos[-self.state][0]][snake.pos[-self.state][1]] = self.blank\_sign # drop tail

for key, value in self.open.items(): # value: e.g. [[8, 3], [6, 3], [7, 4]]

others\_tail\_pos = [self.snakes[\_].pos[-self.state]

if self.snakes[\_].len >= self.state else []

for \_ in set(range(self.snakes\_count)) - {key}]

for x, y in value:

print('start to spread snake {} on grid ({}, {})'.format(key, x, y))

for x\_, y\_ in [((x + 1) % self.height, y), # down

((x - 1) % self.height, y), # up

(x, (y + 1) % self.width), # right

(x, (y - 1) % self.width)]: # left

sign = self.board[x\_][y\_]

idx = sign % self.snakes\_count # which snake, e.g. 0, 1, 2, 3, 4, 5 / number of claims

state = sign // self.snakes\_count # manhattan distance to snake who claim the point or its negative

if sign == self.blank\_sign: # grid in initial state

if [x\_, y\_] in others\_tail\_pos:

print('do not spread other snakes tail, in case of rear-end collision')

continue # do not spread other snakes' tail, in case of rear-end collision

self.board[x\_][y\_] = self.state \* self.snakes\_count + key

self.snakes[key].claimed\_count += 1

new\_open[key].append([x\_, y\_])

elif key != idx and self.state == state:

# second claim, init controversy, change grid value from + to -

print(

'\tgird ({}, {}) in the same state claimed by different snakes '

'with sign {}, idx {} and state {}'.format(

x\_, y\_, sign, idx, state))

if self.snakes[idx].len > self.snakes[key].len: # shorter snake claim the controversial grid

print('\t\tsnake {} is shorter than snake {}'.format(key, idx))

self.snakes[idx].claimed\_count -= 1

new\_open[idx].remove([x\_, y\_])

self.board[x\_][y\_] = self.state \* self.snakes\_count + key

self.snakes[key].claimed\_count += 1

new\_open[key].append([x\_, y\_])

elif self.snakes[idx].len == self.snakes[key].len: # controversial claim

print(

'\t\tcontroversy! first claimed by snake {}, then claimed by snake {}'.format(idx, key))

self.controversy[(x\_, y\_)] = {'state': self.state,

'length': self.snakes[idx].len,

'indexes': [idx, key]}

# first claim by snake idx, then claim by snake key

self.board[x\_][y\_] = -self.state \* self.snakes\_count + 1

# if + 2, not enough for all snakes claim one grid!!

self.snakes[idx].claimed\_count -= 1 # controversy, no snake claim this grid!!

new\_open[key].append([x\_, y\_])

else: # (self.snakes[idx].len < self.snakes[key].len)

pass # longer snake do not claim the controversial grid

elif (x\_, y\_) in self.controversy \

and key not in self.controversy[(x\_, y\_)]['indexes'] \

and self.state + state == 0: # third claim or more

print('snake {} meets third or more claim in grid ({}, {})'.format(key, x\_, y\_))

controversy = self.controversy[(x\_, y\_)]

pprint.pprint(controversy)

if controversy['length'] > self.snakes[key].len: # shortest snake claim grid, do 4 things

print('\t\tsnake {} is shortest'.format(key))

indexes\_count = len(controversy['indexes'])

for i in controversy['indexes']:

self.snakes[i].claimed\_count -= 1 / indexes\_count # update claimed\_count !

new\_open[i].remove([x\_, y\_])

del self.controversy[(x\_, y\_)]

self.board[x\_][y\_] = self.state \* self.snakes\_count + key

self.snakes[key].claimed\_count += 1

new\_open[key].append([x\_, y\_])

elif controversy['length'] == self.snakes[key].len: # controversial claim

print('\t\tcontroversy! multi claimed by snake {}'.format(key))

self.controversy[(x\_, y\_)]['indexes'].append(key)

self.board[x\_][y\_] += 1

new\_open[key].append([x\_, y\_])

else: # (controversy['length'] < self.snakes[key].len)

pass # longer snake do not claim the controversial grid

else:

pass # do nothing with lower state grids

self.open = new\_open # update open

# update controversial snakes' claimed\_count (in fraction) in the end

for \_, d in self.controversy.items():

controversial\_snake\_count = len(d['indexes']) # number of controversial snakes

for idx in d['indexes']:

self.snakes[idx].claimed\_count += 1 / controversial\_snake\_count

def claim2action(self, claim\_position, snake\_idx, step\_count, output\_type):

# claim e.g. [2 ,3 ,4 ,-9] bean 2 is claimed by snake 3 within 4 steps

x, y = claim\_position

x\_h, y\_h = self.snakes[snake\_idx].head # head position

while step\_count > 1:

step\_count -= 1

temp = []

for x\_, y\_ in [((x + 1) % self.height, y), # down

((x - 1) % self.height, y), # up

(x, (y + 1) % self.width), # right

(x, (y - 1) % self.width)]: # left

sign = self.board[x\_][y\_]

if sign == self.blank\_sign:

continue # snake too long, board not spread completely!! see example 20210815 0:41:48

state = sign // self.snakes\_count if sign > 0 else -(sign // self.snakes\_count)

indexes = [sign % self.snakes\_count] if sign >= 0 else self.controversy[(x\_, y\_)]['indexes']

if step\_count == state and snake\_idx in indexes:

temp.append([x\_, y\_])

x, y = random.choice(temp)

if output\_type == 'action':

return get\_direction(x\_h, y\_h, x, y, self.height, self.width)

elif output\_type == 'position':

return [x, y]

else:

assert False, 'unknown output\_type {}'.format(output\_type)

def state2claims(state\_array, max\_state, priority):

# state\_array:

# array([[ 1, 30, 30, 30, 30, 30],

# [30, 30, 3, 30, 30, 30],

# [30, 1, 30, 5, 30, 30],

# [30, 30, 30, 6, 5, 30],

# [30, 30, 30, 30, 30, 6]])

beanCount, snakeCount = state\_array.shape # (5, 6)

horiz = [list(state\_array[\_]).count(max\_state) for \_ in range(beanCount)] # [5, 5, 4, 4, 5]

vert = [list(state\_array[:, \_]).count(max\_state) for \_ in range(snakeCount)] # [4, 4, 4, 3, 4, 4]

claim\_order = []

for b in range(beanCount):

for s in range(snakeCount):

if state\_array[b][s] < max\_state:

claim\_order.append([b, s, state\_array[b][s], -horiz[b] - vert[s]])

# priority rule: smaller state, larger horizontal or vertical cover number of max\_state

if not claim\_order:

return []

if priority == 'state':

temp = min(claim\_order, key=operator.itemgetter(2, 3)) # [0, 0, 1, -9]

elif priority == 'cover':

temp = min(claim\_order, key=operator.itemgetter(3, 2))

else:

assert False, 'unknown priority'

# update

for b in range(beanCount):

state\_array[b, temp[1]] = max\_state + 1

for s in range(snakeCount):

state\_array[temp[0], s] = max\_state + 1

return [temp] + state2claims(state\_array, max\_state, priority)

def my\_controller(observation\_list, action\_space\_list, is\_act\_continuous):

with HiddenPrints():

# detect 1v1, 3v3, 2p or 5p

observation\_len = len(observation\_list.keys())

teams = None

if observation\_len == 7:

teams = [[0], [1]] # 1v1

# teams = [[0, 1]] # 2p

elif observation\_len == 10:

teams = [[0, 1, 2, 3, 4]] # 5p

elif observation\_len == 11:

teams = [[0, 1, 2], [3, 4, 5]] # 3v3

assert teams is not None, 'unknown game with observation length {}'.format(observation\_len)

teams\_count = len(teams)

snakes\_count = sum([len(\_) for \_ in teams])

# read observation

obs = observation\_list.copy()

board\_height = obs['board\_height'] # 10

board\_width = obs['board\_width'] # 20

ctrl\_agent\_index = obs['controlled\_snake\_index'] - 2 # 0, 1, 2, 3, 4, 5

# last\_directions = obs['last\_direction'] # ['up', 'left', 'down', 'left', 'left', 'left']

beans\_positions = obs[1] # e.g.[[7, 15], [4, 14], [5, 12], [4, 12], [5, 7]]

snakes = {key - 2: Snake(obs[key], board\_height, board\_width, beans\_positions)

for key in obs.keys() & {\_ + 2 for \_ in range(snakes\_count)}} # &: intersection

team\_indexes = [\_ for \_ in teams if ctrl\_agent\_index in \_][0]

init\_board = Board(board\_height, board\_width, snakes, beans\_positions, teams)

bd = copy.deepcopy(init\_board)

with HiddenPrints():

while not all(\_ == [] for \_ in bd.open.values()): # loop until all values in open are empty list

bd.step()

print(bd.board)

defense\_snakes\_indexes = [] # save defensive or claimed snakes, to calculate safe move for ctrl snake

# define defensive move

defensive\_claim\_list = [] # [pos, snake\_idx, step]

# first check win side

snakes\_lens = [snake.len for snake in snakes.values()]

snakes\_claimed\_counts = [snake.len for snake in snakes.values()]

print('snakes\_lens: ', snakes\_lens)

print('snakes\_claimed\_counts: ', snakes\_claimed\_counts)

# design defense threshold

# defense\_threshold = 0.5 \* math.pow(board\_height \* board\_width, 1.1) / snakes\_count \* \

# math.sqrt(4 / (4 \* teams\_count + 1))

defense\_threshold = board\_height \* board\_width \* teams\_count / (teams\_count + 1) / snakes\_count

for idx in team\_indexes:

if snakes\_lens[idx] > defense\_threshold:

# 3: player count + 1, 2: player count, 6: snake count

for \_ in range(1, min(bd.state, snakes\_lens[idx] // 2)):

# range should be designed more carefully!!

x, y = snakes[idx].pos[-\_]

if bd.board[x, y] == idx + \_ \* snakes\_count: # claim a loop in step \_

defense\_snakes\_indexes.append(idx)

defensive\_claim\_list.append([[x, y], idx, \_])

if idx == ctrl\_agent\_index:

action = [bd.claim2action(claim\_position=[x, y],

snake\_idx=idx,

step\_count=\_,

output\_type='action')]

print('the controlled agent {} make a defensive move {} within {} step(s)'

.format(idx, action[0], \_))

print(

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*' +

' defensive move ' +

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

return action

# calculate state\_array

# e.g.

# array([[ 1, 30, 30, 30, 30, 30],

# [30, 30, 3, 30, 30, 30],

# [30, 1, 30, 5, 30, 30],

# [30, 30, 30, 6, 5, 30],

# [30, 30, 30, 30, 30, 6]])

max\_state = board\_height + board\_width # 30

state\_array = np.zeros((len(beans\_positions), len(snakes)), dtype=int) + max\_state

for i, (x, y) in enumerate(beans\_positions):

sign = bd.board[x][y]

if sign >= snakes\_count: # bean claimed by one snake

idx = sign % snakes\_count # 0, 1, 2, 3, 4, 5

state = sign // snakes\_count # 1, 2, ...

state\_array[i][idx] = state

elif sign < 0 and sign % snakes\_count in [\_ for \_ in range(snakes\_count) if \_ > 0]: # [2, 3, 4, 5]

state = -(sign // snakes\_count)

for idx in bd.controversy[(x, y)]['indexes']:

state\_array[i][idx] = state

elif sign == bd.blank\_sign: # bean not reachable for any snakes! see example: 20210815, 1:10:23

pass

else:

assert False, 'unknown sign when calculating state\_array'

# calculate claim list

# e.g. [[2, 3, 4, -9], [1, 1, 4, -6], [3, 2, 4, -4], [0, 4, 5, -2]]

claim\_list\_byState = state2claims(state\_array.copy(), max\_state, 'state')

claim\_list\_byCover = state2claims(state\_array.copy(), max\_state, 'cover')

print('claim\_list\_byState: ', len(claim\_list\_byState), claim\_list\_byState)

print('claim\_list\_byCover: ', len(claim\_list\_byCover), claim\_list\_byCover)

claim\_list = claim\_list\_byState if len(claim\_list\_byState) >= len(claim\_list\_byCover) else claim\_list\_byCover

print('claim\_list: ', claim\_list)

claim\_snakes\_indexes = []

# for agent claiming a bean safely, simply return its action

for c in claim\_list:

if ctrl\_agent\_index == c[1]: # the controlled agent claim a bean

print('the controlled agent {} claim a bean {} within {} step(s)'.format(c[1], c[0], c[2]))

action = [bd.claim2action(claim\_position=bd.beans\_positions[c[0]],

snake\_idx=c[1],

step\_count=c[2],

output\_type='action')]

print('and play a action', action[0])

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* claim move \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

return action

claim\_snakes\_indexes.append(c[1])

else:

# not claim any beans,

# traverse all possible action combination (at most 27),

# choose one that claim most grids

# calculate free team snakes indexes and safe positions list

free\_team\_snakes\_indexes = [\_ for \_ in team\_indexes

if \_ not in claim\_snakes\_indexes and \_ not in defense\_snakes\_indexes]

safe\_positions\_list = []

for idx in free\_team\_snakes\_indexes:

safe\_positions = [] # may be empty list

x\_h, y\_h = snakes[idx].head

for x, y in [((x\_h + 1) % bd.height, y\_h), # down

((x\_h - 1) % bd.height, y\_h), # up

(x\_h, (y\_h + 1) % bd.width), # right

(x\_h, (y\_h - 1) % bd.width)]: # left

if bd.board[x][y] == idx + snakes\_count:

safe\_positions.append([x, y]) # should be further tested if exists breath!!

safe\_positions\_list.append(safe\_positions)

# delete snakes whose safe positions are [], which means they are dying

check\_list = [\_ != [] for \_ in safe\_positions\_list]

free\_team\_snakes\_indexes = list(np.array(free\_team\_snakes\_indexes)[check\_list]) # [idx1, idx2]

safe\_positions\_list = [\_ for \_ in safe\_positions\_list if \_] # [[pos1, pos2, pos3], [pos1, pos2, pos3]

print('free\_team\_snakes\_indexes: ', free\_team\_snakes\_indexes)

print('safe\_positions\_list: ', safe\_positions\_list)

# create new snakes

snakes\_next = copy.deepcopy(snakes)

for c in claim\_list: # claimed snake make one move

idx = c[1]

if idx in defense\_snakes\_indexes:

continue # defense is prior to claim

position = bd.claim2action(claim\_position=bd.beans\_positions[c[0]],

snake\_idx=idx,

step\_count=c[2],

output\_type='position')

snakes\_next[idx].step(position)

for c in defensive\_claim\_list: # claimed snake make one move

idx = c[1]

position = bd.claim2action(claim\_position=c[0],

snake\_idx=idx,

step\_count=c[2],

output\_type='position')

snakes\_next[idx].step(position)

# traverse and find the action combination with most grids claimed

max\_claimed\_counts\_sum = 0

best\_pos\_comb = None

for pos\_comb in itertools.product(

\*safe\_positions\_list): # calculate cartesian product of safe positions list

# initiate claimed\_count

for i, idx in enumerate(free\_team\_snakes\_indexes): # unclaimed and undead snake make one move

snakes\_next[idx] = copy.deepcopy(snakes[idx])

snakes\_next[idx].step(pos\_comb[i])

for snake in snakes\_next.values():

snake.claimed\_count = 0 # reset after deep copy !!

bd\_next = Board(board\_height, board\_width, snakes\_next, beans\_positions, teams)

with HiddenPrints():

while not all(

\_ == [] for \_ in bd\_next.open.values()): # loop until all values in open are empty list

bd\_next.step()

print(bd.board)

claimed\_counts = np.zeros(len(team\_indexes))

for i, idx in enumerate(team\_indexes): # not free, consider all team snakes!!

claimed\_counts[i] = snakes\_next[idx].claimed\_count

claimed\_counts\_sum = sum(claimed\_counts)

if claimed\_counts\_sum > max\_claimed\_counts\_sum:

max\_claimed\_counts\_sum = claimed\_counts\_sum

best\_pos\_comb = pos\_comb # one-to-one with free\_team\_snakes\_indexes

print('claimed\_counts\_sum: ', claimed\_counts\_sum)

print('pos\_comb: ', pos\_comb)

print('max\_claimed\_counts\_sum: ', max\_claimed\_counts\_sum, 'best\_pos\_comb: ', best\_pos\_comb)

if best\_pos\_comb:

for i, idx in enumerate(free\_team\_snakes\_indexes):

if ctrl\_agent\_index == idx:

action = [[0, 0, 0, 0]]

direction = snakes[idx].get\_action(best\_pos\_comb[i])

action[0][direction] = 1

print('the controlled agent {} make a safe move'.format(idx), action[0])

print(

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* safe move \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

return action

# todo: design attack moves

# no claim move, no safe move, no attack or die

action = [[0, 0, 0, 0]]

direction = snakes[ctrl\_agent\_index].legal\_action[0]

action[0][direction] = 1

print('the controlled agent {} play a random action and is dying'.format(ctrl\_agent\_index), action[0])

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* random move \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

return action

# 评价指标

本游戏共有两方，每个玩家分别在n\*m的网格中操纵三条蛇。n为垂直高度，m为水平宽度。

蛇是一系列坐标构成的有限不重复有顺序的序列，序列中相邻坐标均相邻，即两坐标的x轴坐标或y轴坐标相同，序列中第一个坐标代表蛇头。玩家只能通过控制蛇头的朝向(上、下、左和右)来控制蛇。

蛇以恒定的速度前进(前进即为序列头插入蛇头指向方向下一格坐标，并删除序列末尾坐标)。蛇的初始位置为网格中的随机位置，初始长度为3格。每吃掉一个豆子长度加1。在游戏中豆子数量维持一个常数（5）不变，即被吃掉则重新随机生成等数量的豆子。

蛇头在自己蛇的身体(即序列重复)、其他蛇的身体(即与其他序列有相同坐标)均判定为死亡。当输入了非法方向时(如正在向左前进，无法选择向右)，随机选择一个合法方向。与传统贪吃蛇不同的是，蛇头超过边界时可以穿越到对称位置，不算做死亡。蛇死亡时可以按照初始化的设置重生。

到达指定步数时游戏结束。游戏结束后，蛇身长度总和大的一方获胜。

观测为一个字典，字典的键为1至7的正整数和"board\_width"、"board\_height"、"last\_direction"、"controlled\_snake\_index"。其中1表示豆子，2至7表示蛇；字典的中的值为一个以[h,w]坐标为元素的列表，表示豆子的位置或蛇身的位置，其中h表示距左上角原点的垂直距离，w表示距原点的水平宽度；在2至7对应的值当中，列表元素从左至右表示对应蛇的蛇头至蛇尾的位置；"board\_width"的值为地图的宽，"board\_height" 的值为地图的高，"last\_direction"的值为上一步各个蛇的方向，"controlled\_snake\_index"的值为控制蛇的序号。

动作空间为长度为n\_action\_dim的列表，其中n\_action\_dim=1，每个元素为Gym当中的Discrete类（ Discrete Link），如[Discrete(4)]。

# 实验结果

实验再及第平台进行提交，排名为14名。从提交至写作日期由系统进行多次测试，最终结果如图2所示。

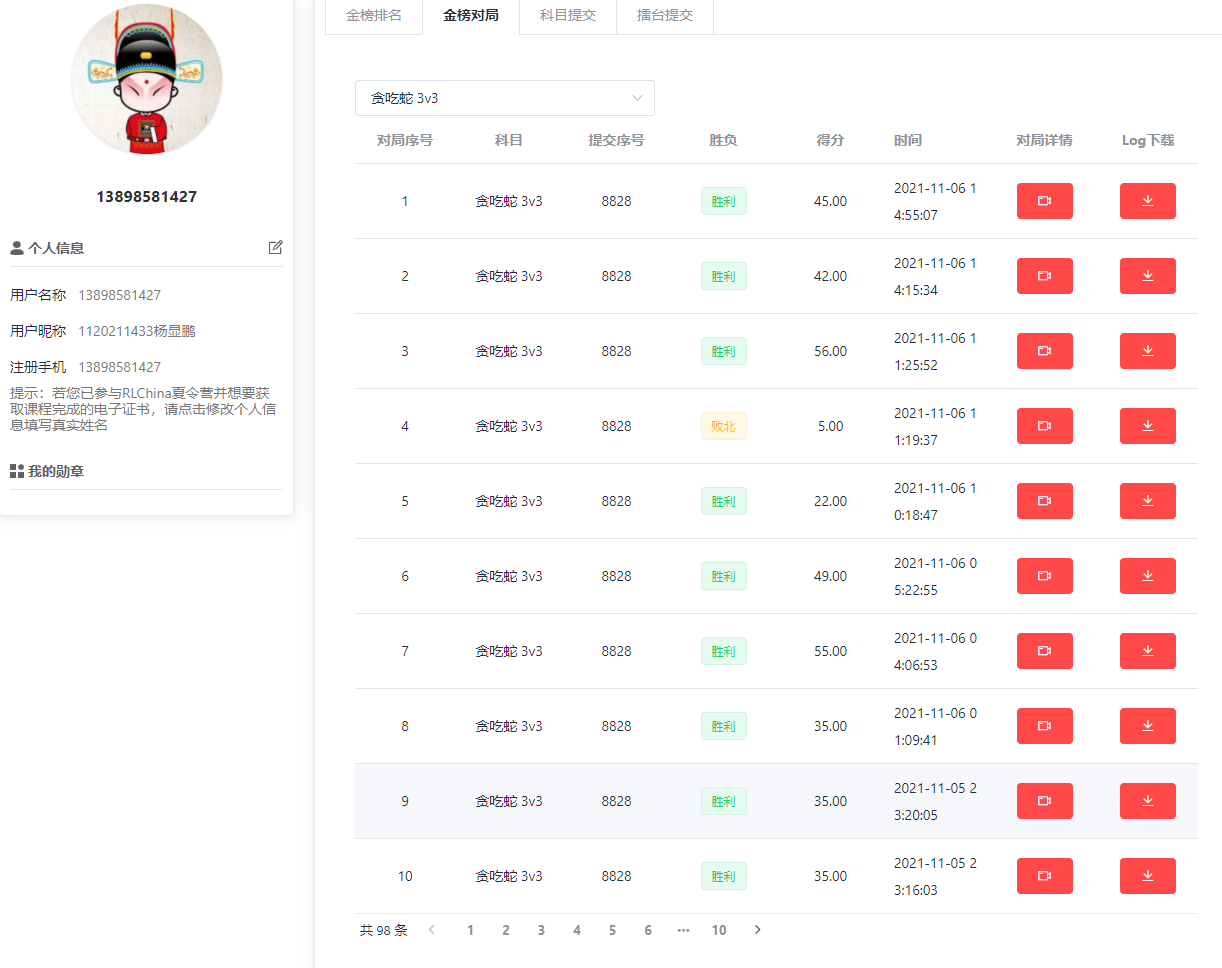


图2

可见，胜利的情形占据了多数。说明本游戏方法，性能优越。

# 比赛截图

