IOAI20. Финал. Проектный тур

Легенда

«Змейка» -- целый класс компьютерных игр, в которых игрок управляет длинным, тонким существом, напоминающим змею, которое ползает по плоскости, ограниченной стенками, собирая еду, избегая столкновения с собственным хвостом и краями игрового поля. Каждый раз, когда змея съедает кусок пищи, она становится длиннее, что постепенно усложняет игру.

Существует разновидность игры, где на игровом поле сразу несколько игроков. Это позволяет выбрать несколько стратегий поведения игрока для победы – добывать еду или делать ловушки для змеек других игроков.

В финальной задаче предлагается разработать программу управления змейкой в том варианте игры, когда на игровом поле существует две змейки. Помимо змеек на поле появляются яблоки, при съедании которых змейка получает 1 балл. При длительности игры в 180 игровых тактов, выиграет та змейка, которая наберет больше баллов.

Условия проведения

1. Участники во время проектного тура финального этапа могут использовать интернет и заранее подготовленные библиотеки для решения задачи.
2. Для решения задачи проектного тура участники должны следовать техническим требованиям, диктуемых особенностями проверочной системы, в которой будут запускаться решения.
3. Участники не могут пользоваться помощью тренера, наставника или привлекать третьих лиц для решения задачи.
4. Проектный тур финального этапа длится 2 дня
5. Финальная задача формулируется участникам в первый день проектного тура.
6. В начале первого дня состязаний участники получают доступ к репозиторию на серверах GitLab.com. Каждый участник имеет свой собственный репозиторий и не имеют доступ к репозиториям других участников.
7. В течение соревновательных дней не ведется учет количества изменений, которые команды регистрируют в Git-репозитории.
8. В конце каждого дня команды должны сформировать запрос на слияние (Merge Request) из своей ветки исходного кода в основную ветку (master) в Git-репозитории.
9. Команда ответственна за то, чтобы в запросе на слияние не должно быть конфликтов. Запрос на слияние с конфликтами может не рассматриваться жюри для запуска в проверяющей системе.
10. В конце первого дня проектного тура жюри проводит предварительный турнир, используя решения участников. Турнир проводится по круговой системе, когда составляются всевозможные пары участников, а затем решения каждой пары участников запускаются на одном игровом поле. После проведения турнира участникам предоставляется доступ к детальным результатам проведенных игр. На основании анализа результатов турнира участники могут доработать свои решения во второй день проектного тура.
11. Во конце второго дня жюри производить запуск решений участников в проверяющей системе для начисления баллов согласно критериям оценки.
12. После выставления баллов, участникам предоставляется доступ к проверяющей системе, так что участники могут ознакомиться с логикой проверки и подать апелляцию, если не согласны с корректностью проведения оценивания.
13. После рассмотрения сути апелляции, жюри вправе провести оценивание еще раз и назначить участнику дополнительные баллы.
14. Описанные выше условия могут быть изменены членами жюри. Все изменения в условиях объявляются участникам перед началом каждого дня состязаний.

Технические требования

Решение задачи должно быть написано на Python3 и должно содержать реализацию класса `Bot`, расширяющего класс `IBot` из модуля `src.bot`. Самым первым действием в конструкторе класса должно быть инициализация алгоритма управления змейкой.

```
from src.bot import IBot
class Bot(IBot):

  def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)
```

Конструктор может быть расширен дополнительным функционалом. Например, здесь может быть инициализировано изначальное состояние алгоритма управления.

Класс `Bot` должен реализовывать метод `chooseDirection`:

```
def chooseDirection(self, snake, opponent, mazeSize, apple)
```

Данный метод будет вызываться в процессе игры в начале каждого игрового такта. Подразумевается, что метод должен запланировать действия игрока в зависимости от ситуации на игровом поле и вернуть следующее действие, которое должна выполнить змейка в текущем такте. Ситуация на игровом поле передается в метод через следующие аргументы:

- `snake` – координаты элементов тела змейки игрока, объект класса `Snake` из модуля `src.snake`;
- `opponent` – координаты элементов тела змейки противника, объект класса `Snake` из модуля `src.snake`;
- `mazeSize` – кортеж, который содержит размеры игрового поля, класс `src.geometry.Coordinate`, для проектного тура закреплен постоянный размер 14 на 14 ячеек);
- `apple` – кортеж с координатами ячейки с яблоком (`apple.x` и `apple.y`), класс `src.geometry.Coordinate`.

Метод не должен выполняться дольше одной секунды, иначе игра будет завершена с проигрышем игрока.

Метод должен вернуть одну из констант `src.geometry.DOWN`, `src.geometry.UP`, `src.geometry.LEFT`, `src.geometry.RIGHT`, определяющих в каком направлении смещается передняя часть змейки в текущем такте.

Больше информации о классах и методах, доступных для алгоритма управления змейкой можно найти в README.md файле визуализатора игры, который может использоваться для локального тестирования решения.

Пример простого алгоритма управляющего змейкой первые три такта, а потом завершающего игру:

```
from src.bot import IBot
from src.geometry import Direction, Coordinate, RIGHT
from src.snake import Snake

STEPS_TO_MOVE = 3

class Bot(IBot):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.i = 0

    def chooseDirection(self, snake: Snake,
                        opponent: Snake,
                        mazeSize: Coordinate,
                        apple: Coordinate) -> Direction:
        self.i += 1

        if self.i > STEPS_TO_MOVE:
            head = snake.body[0]
            firstBodyElement = snake.body[1]
            directionToDeath = head.getDirection(firstBodyElement)

            return directionToDeath
        else:
            return RIGHT
```

Игра между двумя змейками продолжается пока обе змейки живы.

Игра завершается, если, как минимум, одна змейка погибает.

Змейка считается погибшей, если выполняется одно из следующих условий:

- количество игровых тактов превышает 180;
- продолжительность исполнения метода `chooseDirection` превышает 1 секунду;
- в ходе исполнения `chooseDirection` происходит исключение (exception);
- метод `chooseDirection`: вернул значение отличающееся от `src.geometry.DOWN`, `src.geometry.UP`, `src.geometry.LEFT`, `src.geometry.RIGHT`;
- в результате смещения в ту или иную сторону змейка попадает за границы поля;
- в результате смещения в ту или иную сторону змейка попадает в ячейку, занятую телом змейки.

Критерии оценки

1. Змейка продолжает перемещаться после третьего игрового такта – 1 балл;
2. В каждой из 10 игр при наличии доступных перемещений змейка не смещается в ячейку, занятую телом этой же змейки – 10 баллов;
3. В каждой из 10 игр при наличии доступных перемещений змейка не смещается за границы поля – 10 баллов;
4. В каждой из 10 игр при наличии доступных перемещений змейка не смещается в ячейку, занятую телом змейки противника – 10 баллов;
5. В каждой из 10 игр змейка съедает, минимум, одно яблоко – 15 баллов;
6. Как минимум, в 3 из 10 игр змейка выполняет такую серию перемещений, что змейка противника вынуждена сместиться в ячейку, занятую телом змейки, или за пределы поля – 5 баллов при условии, если получены баллы за пункты 2, 3 и 4.

Для проверки согласно перечисленным критериям, обе змейки на игровом поле управляются решением от одного и того же участника.

Критерии определения победителя проектного тура

- Сумма баллов, набранных за решение задачи проектного тура финального этапа, определяет итоговую результативность участника (измеряемую в баллах).
- Участники ранжируются по результативности.
- Победитель – участник с максимальной результативностью.

Решение

```
from src.bot import IBot
from src.geometry import Direction, Coordinate, directions
from src.snake import Snake


def create_map(snake, opponent, mazeSize):
    lst = [[0 for s in range(mazeSize.x)] for i in range(mazeSize.x)]
    for i in snake.body:
        lst[i.y][i.x] = 1
    for i in opponent.body:
        lst[i.y][i.x] = 1

    return lst


def get_good_coords(map_lst):
    color = 2
    while True:
        colored = False

        color_coords = []

        for i in range(len(map_lst)):
```

```
            for s in range(len(map_lst[i])):
                if map_lst[i][s] == 0:
                    map_lst[i][s] = color
                    color_coords.append((i, s))
                    colored = True
                    break
            if colored:
                break

        if not color_coords:
            break

        while color_coords:
            s = color_coords.pop(0)
            y, x = s
            #print(x, y)
            if y > 0:
                if map_lst[y - 1][x] == 0:
                    map_lst[y - 1][x] = color
                    color_coords.append((y - 1, x))
            if y < len(map_lst) - 1:
                if map_lst[y + 1][x] == 0:
                    map_lst[y + 1][x] = color
                    color_coords.append((y + 1, x))
            if x > 0:
                if map_lst[y][x - 1] == 0:
                    map_lst[y][x - 1] = color
                    color_coords.append((y, x - 1))
            if x < len(map_lst) - 1:
                if map_lst[y][x + 1] == 0:
                    map_lst[y][x + 1] = color
                    color_coords.append((y, x + 1))

        color += 1

    color_dict = {}

    for i in map_lst:
        for s in i:
            if s in color_dict.keys():
                color_dict[s] += 1
            else:
                color_dict[s] = 1

    return map_lst, color_dict


def get_occupied_cells(snake, opponent, apple):
    lst = []

    lst += snake.body

    if snake.body[0].getDistance(apple) > 1:
        lst.pop(-1)

    lst += opponent.body

    if opponent.body[0].getDistance(apple) > 1:
        lst.pop(-1)

    return lst


def check_move(direction, snake_head: Coordinate, occupied_cells, mazeSize):
    snake_head = snake_head.moveTo(direction)

    if snake_head.x < 0 or snake_head.x >= mazeSize.x:
        return False
```

```
    if snake_head.y < 0 or snake_head.y >= mazeSize.x:
        return False

    if snake_head in occupied_cells:
        return False

    return True


def get_good_moves(snake_head: Coordinate, occupied_cells, mazeSize):
    lst = []
    for direction in directions:

        if check_move(direction, snake_head.clone(), occupied_cells, mazeSize):
            lst.append(direction)

    return lst


def get_best_move(good_moves, snake_head, apple, map_lst, snake_len):
    start_good_moves = good_moves.copy()
    if not good_moves:
        good_moves = directions

    color_map, color_dict = get_good_coords(map_lst)

    colors = list(color_dict.keys())
    colors.remove(1)

    if len(colors) > 1:
        colors.sort(key=lambda s: color_dict[s])
        move_colors = []

        for move in good_moves:
            coord = snake_head.moveTo(move)
            color = color_map[coord.y][coord.x]
            move_colors.append(color)
        new_good_moves = good_moves.copy()

        for move in good_moves:
            coord = snake_head.moveTo(move)
            color = color_map[coord.y][coord.x]
            if color_dict[color] < snake_len:
                new_good_moves.remove(move)

        good_moves = new_good_moves

        if not good_moves:
            good_moves = start_good_moves

    best_move = False
    best_move_n = 10000

    for move in good_moves:
        coord = snake_head.moveTo(move)
        n = coord.getDistance(apple)
        if n < best_move_n:
            best_move = move
            best_move_n = n

    return best_move


class Bot(IBot):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def chooseDirection(self, snake: Snake, opponent: Snake, mazeSize: Coordinate,
```

```
                        apple: Coordinate) -> Direction:

        map_lst = create_map(snake, opponent, mazeSize)

        snake_head = snake.body[0]
        opponent_head = opponent.body[0]

        occupied_cells = get_occupied_cells(snake, opponent, apple)

        good_moves = get_good_moves(snake_head, occupied_cells, mazeSize)
        good_opponent_moves = get_good_moves(opponent_head, occupied_cells,
mazeSize)

        best_move = get_best_move(good_moves, snake_head, apple, map_lst,
len(snake.body))
        best_opponent_move = get_best_move(good_opponent_moves, opponent_head,
apple, map_lst,
                                           len(opponent.body))

        coord_after_move = snake_head.moveTo(best_move)
        coord_after_opponent_move = opponent_head.moveTo(best_opponent_move)

        if coord_after_move == coord_after_opponent_move:
            if coord_after_move == coord_after_opponent_move == apple:
                if len(snake.body) == len(opponent.body):
                    sum1 = snake.body[0].x * 100 + snake.body[0].y
                    sum2 = opponent.body[0].x * 100 + opponent.body[0].y
                    if sum1 > sum2:
                        return best_move
            if len(snake.body) > len(opponent.body):
                return best_move
            else:
                if len(good_moves) == 1:
                    return best_move
                else:
                    good_moves.remove(best_move)
                    return get_best_move(good_moves, snake_head, apple, map_lst,
len(snake.body))

        return best_move
```