Цель

Создать программу, реализующую игру «The Snake» и умеющую играть в эту игру самостоятельно при помощи искусственного интеллекта.

Задачи

1. Разработать примитивную игру «The Snake», работающую в консоли.
2. Сделать GUI (графический интерфейс).
3. Разработать генетический алгоритм для «змейки».

Реализация

Использованные библиотеки

NumPy (Numerical Python) – это библиотека Python, которая предлагает возможность использования мощных N-мерных массивов, высокоуровневых функций, интеграции кода C/C++ и Fortran, использование линейной алгебры, преобразований Фурье и различных возможностей случайных чисел. Она также предлагает эффективный многомерный контейнер общих данных. С ее помощью можно определять произвольные типы данных.

Tkinter – это пакет для Python, предназначенный для работы с библиотекой Tk. Библиотека Tk содержит компоненты графического интерфейса пользователя. Эта библиотека написана на языке программирования Tcl.

Time – модуль для работы со временем в Python.

Random – модуль, позволяющий генерировать случайные числа.

Game – библиотека, предназначенная для разработки мультимедийных приложений с графическим интерфейсом, например, игр.

Math – библиотека, позволяющая упростить запись сложных математических функций и предоставляющая обширный функционал работы с числами.

Импорт библиотек

Для импорта библиотек необходимо воспользоваться командной строкой. Таким образом, можно запустить командную строку от имени администратора, перейти в каталог библиотек (в моем случае C:\Program Files\Python39\Scripts) и использовать команду *pip install <имя_библиотеки>*.

Разработка кода «The Snake»

Решение будет содержать 4 класса: main, game, players, test. Задача main – запускать игру на воспроизведение, game – основной код, будет содержать логику игры, players – имитация игрока, будет содержать в себе генетический алгоритм, test – класс для генерации положения «яблок».

1. Игра в консоли

Змея может ползать вверх, вниз, вправо, влево. Возможности движения змеи за один ход отражаются в глобальных переменных UP, DOWN, LEFT, RIGHT. Все эти возможности объединены в массив MOVES. Также нужно задать переменные, показывающие наличие «яблок».

```
UP = (-1, 0)
DOWN = (1, 0)
LEFT = (0, -1)
RIGHT = (0, 1)
MOVES = [UP, DOWN, LEFT, RIGHT]
EMPTY = 0
FOOD = 99
```

Заведем класс Game и опишем функцию __init__, запускающуюся по обращении к классу. Size – размер доски (в этом случае 10x10 яблок), num_snakes – число змей на доске (здесь будет одна змея), players – информация об игроке, gui – графический интерфейс, display – отображение графического интерфейса, max_turns – количество попыток игры. Перечисленные параметры должны передаваться в функцию __init__. Внутри функции объявим переменные

num_food – количество яблок на поле, turn – начальная точка игры, snake_size – размер змеи. Для случая, когда на поле несколько змей, обозначим массив snakes. В food внесены положения первых четырех яблок, которые будут отображаться при запуске. Каждой змее соответствует игрок», поэтому сколько змей, столько и игроков - запишем id игроков в player_ids. В случае рассмотрения игры на несколько игроков это бы пригодилось для исключения проигравших игроков и их змей с поля. Board – собственно поле, изначально представляет собой пустой массив size x size (здесь 10x10). Затем инициализируем элементы board: змея/яблоко. Начальный индекс яблока - 0.

Далее нужно определить положения появляющихся в процессе игры «яблок». Для этого напишем класс test, в котором сможем генерировать случайные координаты яблок. Выглядеть он будет следующим образом:

```
import random as rand
print([(rand.randint(0, 9), rand.randint(0, 9)) for _ in range(200)])
```

Как можно заметить, он представляет собой просто генератор последовательности случайных координат для поля 10x10, выводящий результат генерации в консоль. Для разнообразия, если есть желание, можно генерировать координаты несколько раз и менять их в Game. Итак, сгенерируем координаты и запишем их в food_xy.

```
class Game:
    def __init__(self, size, num_snakes, players, gui = None, display = False, max_turns = 100):
        self.size = size
        self.num_snakes = num_snakes
        self.players = players
        self.gui = gui
        self.display = display
        self.max_turns = max_turns
        self.num_food = 4
        self.turn = 0
```

```
        self.snake_size = 3
        self.snakes = [[((j + 1) * self.size//(2 * self.num_snakes), self.size//2 + i) for i in range (self.snake_size)]
                       for j in range (self.num_snakes)]
        self.food = [(self.size//4, self.size//4),
                     (3 * self.size//4, self.size//4),
                     (self.size//4, 3 * self.size//4),
                     (3 * self.size//4, 3 * self.size//4)]
        self.player_ids = [i for i in range (self.num_snakes)]
        self.board = np.zeros([self.size, self.size])
        for i in self.player_ids:
            for tup in self.snakes[i]:
                self.board[tup[0]][tup[1]] = i + 1
        for tup in self.food:
            self.board[tup[0]][tup[1]] = FOOD
        self.food_index = 0
        self.food_xy = [(8, 3), (9, 7), (9, 3), (8, 6), (8, 2), (0, 8), (6, 1), (5, 2), (6, 3), (1, 3), (6, 2), (1, 7), (8, 5), (9, 4 ), (1, 4), (3, 8),
(3, 6), (8, 0), (3, 7), (7, 6), (2, 8), (0, 1), (8, 8), (0, 4), (4, 3), (7, 1), (3, 5), (1, 4), (9, 6), (3, 3), (6, 2), (1, 8), (5, 3), (1, 9), (8, 2),
(6, 1), (2, 0), (9, 9), (0, 7), (7, 0), (3, 3), (8, 9), (1, 4), (9, 0), (5, 0), (9, 1), (4, 9), (5, 8), (3, 1), (4, 1), (0, 6 ), (9, 1), (8, 7), (5, 9),
(5, 9), (4, 6), (5, 9), (1, 8), (3, 2), (1, 1), (1, 3), (2, 0), (2, 9), (5, 6), (1, 3), (9, 9), (5, 2), (9, 9), (2, 7), (6, 2), (8, 0), (8, 0), (8, 5),
(5, 8), (7, 7), (9, 5), (9, 6), (8, 9), (8, 4), (3, 8), (4, 7), (3, 0), (4, 2), (4, 4), (2, 2), (7, 4), (4, 6), (5, 4), (4, 8 ), (8, 9), (8, 8), (8, 6),
(1, 2), (7, 6), (1, 2), (7, 5), (4, 2), (0, 0), (7, 4), (9, 2), (0, 5), (8, 5), (5, 1), (4, 5), (9, 0), (7, 9), (7, 6), (6, 6), (8, 4), (7, 4 ), (1, 1),
(4, 2), (6, 5), (9, 4), (7, 0), (3, 9), (3, 2), (7, 2), (1, 9), (5, 5), (2, 5), (1, 6), (5, 1), (7, 9), (7, 4), (4, 0), (4, 0), (4, 1), (1, 1), (2, 5),
(5, 5), (9, 5), (0, 0), (7, 8), (2, 5), (9, 3), (2, 2), (8, 0), (7, 5), (1, 9), (5, 4), (8, 0), (8, 8), (7, 5), (4, 9), (6, 1 ), (0, 4), (9, 5), (8, 7),
(2, 3), (9, 1), (6, 6), (7, 3), (9, 9), (7, 7), (7, 2), (2, 6), (2, 8), (6, 5), (8, 0), (1, 7), (1, 6), (6, 2), (4, 0), (1, 4), (2, 6), (4, 7), (8, 7),
(5, 7), (5, 1), (7, 4), (2, 2), (8, 3), (6, 3), (3, 2), (6, 4), (4, 7), (2, 0), (3, 6), (1, 9), (5, 1), (8, 3), (1, 6), (9, 9 ), (0, 0), (1, 2), (7, 7),
(9, 4), (8, 8), (8, 4), (0, 5), (9, 0), (6, 9), (4, 7), (1, 5), (1, 7), (5, 3), (4, 2), (5, 8), (7, 5)]
```

Далее опишем функцию move, которая будет обрабатывать движение змеи.

```
    def move(self):
        moves = []
        for i in self.player_ids:
            snake_i = self.snakes[i]
            move_i = self.players[i].get_move(self.board, snake_i)
            moves.append(move_i)
            new_square = (snake_i[-1][0] + move_i[0], snake_i[-1][1] + move_i[1])
            snake_i.append(new_square)
```

Во фрагменте выше показана обработка движения головы змеи. Выполняется проверка, жива ли змея, и если ответ положительный, тогда со стороны головы

добавляется соседний «квадратик». Но так змея удлинится на единицу, что будет неверным, если змея не скушала яблоко. Тогда нужно «урезать» ей хвост.

```
for i in self.player_ids:
    head_i = self.snakes[i][-1]
    if head_i not in self.food:
        self.board[self.snakes[i][0][0]][self.snakes[i][0][1]] = EMPTY
        self.snakes[i].pop(0)
    else:
        self.food.remove(head_i)
```

Так выглядит проверка, не вышла ли «змея» за пределы игрового поля:

```
for i in self.player_ids:
    head_i = self.snakes[i][-1]
    if head_i[0] >= self.size or head_i[1] >= self.size or head_i[0] < 0 or head_i[1] < 0:
        self.player_ids.remove(i)
    else:
        self.board[head_i[0]][head_i[1]] = i + 1
```

В случае нахождения нескольких змей на поле, можно изобразить функцию, обрабатывающую столкновения:

```
for i in self.player_ids:
    head_i = self.snakes[i][-1]
    for j in range (self.num_snakes):
        if i == j:
            if head_i in self.snakes[i][:-1]:
                self.player_ids.remove(i)
        else:
            if head_i in self.snakes[j]:
                self.player_ids.remove(i)
```

Если змея избавится от какого-то «яблока» на поле, необходимо сгенерировать новое на каком-нибудь свободном месте:

```
        while len(self.food) < self.num_food:
            x = self.food_xy[self.food_index][0]
            y = self.food_xy[self.food_index][1]
            while self.board[x][y] != EMPTY:
                self.food_index += 1
                x = self.food_xy[self.food_index][0]
                y = self.food_xy[self.food_index][1]
            self.food.append((x, y))
            self.board[x][y] = FOOD
            self.food_index += 1
        return moves
```

Теперь можно создать функцию play, которая будет контролировать игру. Можно, например, автоматически избавляться от змеи, если она не съест яблоко, скажем, за 10 ходов.

```
    def play(self, display, termination = False):
        if display:
            self.display_board()
        while True:
            if termination:
                for i in self.player_ids:
                    if len(self.snakes[0]) - self.turn/20 <=0:
                        self.player_ids.remove(i)
                        # remove return if more than 1 snakes
                        return -2
            if self.turn >= self.max_turns:
                return 0
```

Также нужна проверка, есть ли в принципе живые змеи на поле. Если их нет, возвращаем -1.

```
            if len(self.player_ids) == 0:
                return -1
```

Обработка движений представлена ниже. Чтобы отслеживать движения, добавлены выводы. Также здесь прописано обновление поля в GUI. GUI будет написан в следующей части данной работы. Time.sleep – это функция притормаживания выполнения программы. Без нее программа может перегрузить процессор и зависнуть.

```
        moves = self.move()
        self.turn += 1
        if display:
            for move in moves:
                if move == UP:
                    print('UP')
                elif move == DOWN:
                    print('DOWN')
                elif move == LEFT:
                    print('LEFT')
                else:
                    print('RIGHT')
            self.display_board()
            if self.gui is not None:
                self.gui.update()
            time.sleep(3)
```

Теперь опишем функцию display_board. Эта функция отвечает за представление игрового поля игры в консоли. После описания классов player и main будет представлен пример вывода.

```
def display_board(self):
    for i in range(self.size):
        for j in range(self.size):
            if self.board[i][j] == EMPTY:
                print('|_', end = '')
            elif self.board[i][j] == FOOD:
                print('|#', end = '')
            else:
```

```
                print('|' + str(int(self.board[i][j])), end = '')
            print('|')
```

Затем необходимо создать случайного игрока, который будет случайным образом генерировать движения змеи, просто выбирая случайное число от 0 до 3. Сделаем это в классе player. В дальнейшем будет написан искусственный интеллект вместо случайного игрока. Этому будет посвящена третья часть данной работы.

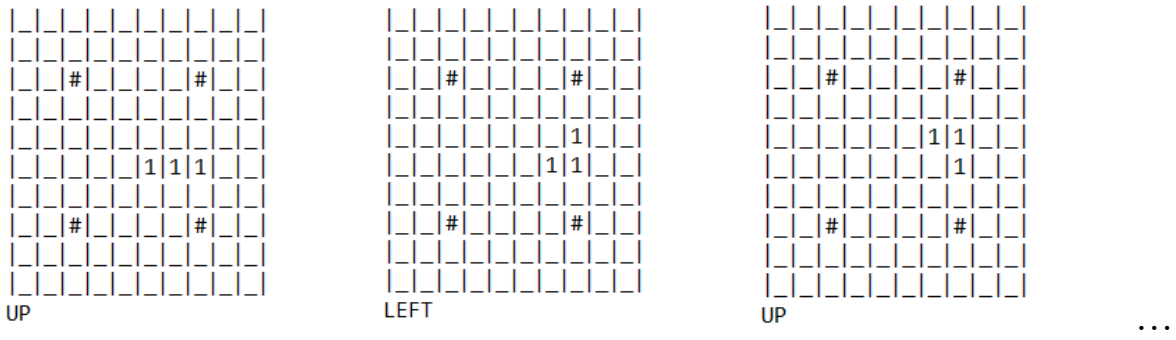
```
from game import *
import math


class RandomPlayer:
    def __init__(self, i):
        self.i = i
    def get_move(self, board, snake):
        r = rand.randint(0, 3)
        return MOVES[r]
```

Для тестирования на данном этапе в классе main достаточно написать:

```
from game import *
size = 10
num_snakes = 1
players = [RandomPlayer(0)]
game = Game(size, num_snakes, players, gui=None, display=True, max_turns=100)
game.play(True, termination=False)
```

Итак, несколько первых ходов змеи при запуске представлены ниже.



...

2. GUI

Теперь, когда положено начало змейке, можно создать графический интерфейс.

Объявим класс Gui в game.py, который будет принимать на вход game, size. Определим ширину и высоту окна. Также, если змеек на поле несколько, нужно каждой задать свой цвет, а также цвет головы должен отличаться от цвета тела. Таким образом, будут создаваться змейки из цветных квадратных блоков, размер которых тоже необходимо прописать.

```
class Gui:
    def __init__(self ,game, size):
        self.game = game
        self.game.gui = self
        self.size = size
        self.ratio = self.size/self.game.size
        self.app = tk.Tk()
        self.canvas = tk.Canvas(self.app, width = self.size, height = self.size)
        self.canvas.pack()
        for i in range (len(self.game.snakes)):
            color = '#' + '{0:03X}'.format((i + 1) * 500)
            snake = self.game.snakes[i]
            self.canvas.create_rectangle(self.ratio * (snake[-1][1]),
                                         self.ratio * (snake[-1][0]),
                                         self.ratio * (snake[-1][1] + 1),
                                         self.ratio * (snake[-1][0] + 1), fill = color)
            for j in range (len(snake) - 1):
                color = '#' + '{0:03X}'.format((i + 1) * 123)
                self.canvas.create_rectangle(self.ratio * (snake[j][1]),
                                             self.ratio * (snake[j][0]),
                                             self.ratio * (snake[j][1] + 1),
                                             self.ratio * (snake[j][0] + 1), fill = color)
```

Что касается яблок, они будут представлять собой квадратные блоки черного цвета.

```
        for food in self.game.food:
                self.canvas.create_rectangle(self.ratio * (food[1]),
                                             self.ratio * (food[0]),
                                             self.ratio * (food[1] + 1),
                                             self.ratio * (food[0] + 1), fill = '#000000000')
    def update(self):
        self.canvas.delete('all')
        for i in range (len(self.game.snakes)):
            color = '#' + '{0:03X}'.format((i + 1) * 500)
            snake = self.game.snakes[i]
            self.canvas.create_rectangle(self.ratio * (snake[-1][1]),
                                         self.ratio * (snake[-1][0]),
                                         self.ratio * (snake[-1][1] + 1),
                                         self.ratio * (snake[-1][0] + 1), fill = color)
            for j in range (len(snake) - 1):
                color = '#' + '{0:03X}'.format((i + 1) * 123)
```

Если запустить программу на данном этапе, появится игровое поле, на нем – 4 яблока и змейка. Но змейка в основном будет кушать себя, а не яблоки, поскольку сейчас она настроена на случайные ходы, а не на интеллектуальные.

Теперь стоит задача прописать змейке такой алгоритм, следуя которому она смогла бы совершать ходы в направлении яблок и продлить время своей жизни.

3. Генетический алгоритм

Для начала разберемся, что же такое генетический алгоритм. Говоря простыми словами и основываясь на самом названии алгоритма, можно сказать, что его идея основана на комбинировании. Путем перебора и отбора получаются необходимые выгодные комбинации. Алгоритм делится на три этапа: скрещивание, отбор, формирование нового поколения. Можно повторить эти три этапа подряд столько раз, сколько нужно, пока результат не будет удовлетворять требованиям. Или же пока количество поколений (циклов) не достигнет заранее заданного максимума или не закончится время на мутацию.

Шаги алгоритма:

1. Создается первая популяция – некоторое множество brains, каждый входящий в него brain играет в змейку и набирает какое-то количество очков.
2. Происходит отбор – из brains выбираются лучшие 25%, которые «выводят потомство» в количестве двух штук.
3. Brains перезаписывается: новое множество будет включать в себя полученных на предыдущем этапе потомков и заново сгенерированные brain-ы.

Приведенные шаги выполняются нужное количество раз (этот параметр будет необходимо задать, чем его значение больше, тем выше успех). После последней итерации выбирается лучший brain – тот, который способен набрать наибольшее число очков. Процесс набора им очков как раз и будет показан нам в интерфейсе при запуске итоговой программы.

Итак, перейдем к реализации. Создадим класс GeneticPlayer в players.py, объявим в нем параметры генерации поколений.

```
class GeneticPlayer:

    def __init__(self, pop_size, num_generations, num_trails, window_size, hidden_size, board_size, mutation_chance = 0.1,
mutation_size = 0.1):

        self.pop_size = pop_size

        self.num_generations = num_generations

        self.num_trails = num_trails

        self.window_size = window_size

        self.hidden_size = hidden_size

        self.board_size = board_size

        self.mutation_chance = mutation_chance

        self.mutation_size = mutation_size

        self.display = False

        # brain selected to play games

        self.current_brain = None

        self.pop = [self.generate_brain(self.window_size**2, self.hidden_size, len(MOVES)) for _ in range(self.pop_size)]
```

Затем создадим функцию генерации brain-ов. Каждый brain будет заключать в себе три слоя матрицы: два скрытых и слой вывода. Первый скрытый слой производит «потомков» и передает их второму слою, так же поступает второй по отношению к слою вывода. Слой вывода будет как бы являться итогом отбора из скрытых слоев.

```
    def generate_brain(self, input_size, hidden_size, output_size):
        hidden_layer1 = np.array([[rand.uniform(-1, 1) for _ in range(input_size + 1)] for _ in range(hidden_size)])
        hidden_layer2 = np.array([[rand.uniform(-1, 1) for _ in range(hidden_size + 1)] for _ in range(hidden_size)])
        output_layer = np.array([[rand.uniform(-1, 1) for _ in range(hidden_size + 1)] for _ in range(output_size)])
        return [hidden_layer1, hidden_layer2, output_layer]
    def get_move(self, board, snake):
        input_vector = self.process_board(board, snake[-1][0], snake[-1][1], snake[-2][0], snake[-2][1])
        hidden_layer1 = self.current_brain[0]
        hidden_layer2 = self.current_brain[1]
        output_layer = self.current_brain[2]
        hidden_result1 = np.array([math.tanh(np.dot(input_vector, hidden_layer1[i])) for i in range (hidden_layer1.shape[0])] +
[1]) # [1] for bias
        hidden_result2 = np.array([math.tanh(np.dot(hidden_result1, hidden_layer2[i])) for i in range (hidden_layer2.shape[0])] +
[1]) # [1] for bias
        output_result = np.array([np.dot(hidden_result2, output_layer[i]) for i in range (output_layer.shape[0])])
        max_index = np.argmax(output_result)
        return MOVES[max_index]
    def process_board(self, board, x1, y1, x2, y2):
        # x and y are positions of the snake
        input_vector = [[0 for _ in range(self.window_size)] for _ in range(self.window_size)]
        for i in range (self.window_size):
            for j in range (self.window_size):
                ii = x1 + i - self.window_size//2
                jj = y1 + j - self.window_size//2
                # if window out of bounds, snake can't move to that position
                if ii < 0 or jj < 0 or ii >= self.board_size or jj >= self.board_size:
                    input_vector[i][j] = -1
                elif board[ii][jj] == FOOD:
                    input_vector[i][j] =1
```

```
            elif board[ii][jj] == EMPTY:
                input_vector[i][j] = 0
            else: # it is another snake
                input_vector[i][j] = -1
    if self.display: print(np.array(input_vector))
    input_vector = list(np.array(input_vector).flatten()) + [1]
    return np.array(input_vector)
```

Process_board – функция, обрабатывающая все позиции на доске вокруг головы змеи (берется область размером с доску, в центре которой – голова змеи). Выполняется проверка, выходит ли эта область за пределы доски, присутствует ли рядом другая змея (отрицательный исход), и проверка на наличие еды (положительный исход).

Переходим к «репродукции» – о ней говорилось ранее (генерируется поколение, выбирается 25% лучших особей и происходит повторная генерация, т.д.).

```
def reproduce(self, top_25):
    new_pop = []
    for brain in top_25:
        new_pop.append(brain)
    for brain in top_25:
        new_brain = self.mutate(brain)
        new_pop.append(new_brain)
    # spawn new random brains
    for _ in range(self.pop_size//2):
        new_pop.append(self.generate_brain(self.window_size**2, self.hidden_size, len(MOVES)))
    return new_pop
def mutate(self, brain):
    new_brain = []
    for layer in brain:
        new_layer = np.copy(layer)
        for i in range(new_layer.shape[0]):
            for j in range(new_layer.shape[1]):
                if rand.uniform(0, 1) < self.mutation_chance:
```

```
                    new_layer[i][j] += rand.uniform(-1, 1) * self.mutation_size
            new_brain.append(new_layer)
        return new_brain


    def one_generation(self):
        scores = [0 for _ in range(self.pop_size)]
        max_score = 0
        for i in range(self.pop_size):
            for j in range(self.num_trails):
                self.current_brain = self.pop[i]
                game = Game(self.board_size, 1, [self])
                outcome = game.play(False, termination = True)
                score = len(game.snakes[0]) #for single player variation
                scores[i] += score
                if outcome == 0:
                    print('Snake', i, 'made it to the last turn')
                if score > max_score:
                    max_score = score
                    print(max_score, 'at ID', i)
        top_25_indexes = list(np.argsort(scores))[3 * (self.pop_size//4):self.pop_size]
        print(scores)
        top_25 = [self.pop[i] for i in top_25_indexes][::-1]
        self.pop = self.reproduce(top_25)
    def evolve_pop(self):
        for i in range(self.num_generations):
            self.one_generation()
            print('gen', i)
        key = input('Enter any key character to display boards ')
        for brain in self.pop:
            self.display = True
            self.current_brain = brain
            game = Game(self.board_size, 1, [self], display = True)
            gui = Gui(game, 800)
            game.play(True, termination=True)
            print('Snake length', len(game.snakes[0]))
```

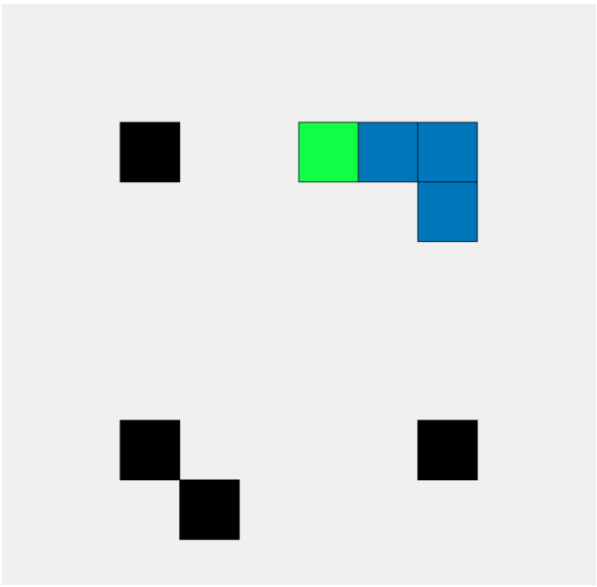
One_generation описывает действия одного brain-a. Evolve_pop – проходит по поколениям и обновляет популяцию согласно результатам поколений.

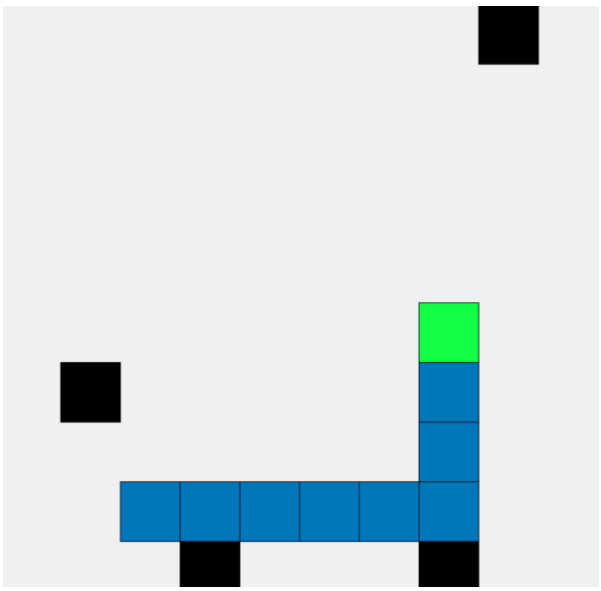
Результат

Конечный фрагмент вывода в консоль (отладочные выводы):

```
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 6, 10, 8, 10, 9, 10, 5, 10, 8, 10, 10, 10, 10, 5, 7, 10, 6, 10, 8
gen 793
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 5, 10, 10, 10, 7, 8, 10, 10, 10, 10, 10, 10, 7, 10, 10, 1
gen 794
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 7, 10, 10, 10, 8, 10, 5, 7, 7, 10, 8, 8, 10, 10, 7, 10, 10, 10, 10, 10, 6, 1
gen 795
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 4, 10, 10, 4, 6, 10, 8, 7, 10, 8, 10, 6, 10, 5, 6, 8, 10, 4, 10, 10, 10,
gen 796
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 9, 10, 10, 6, 10, 7, 10, 10, 6, 10, 7, 10, 10, 7, 10, 10, 7, 10, 10, 10, 10,
gen 797
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 7, 10, 10, 8, 10, 10, 10, 5, 8, 10, 8, 10, 8, 9, 10, 9, 10, 10, 10, 10, 10, 1
gen 798
10 at ID 0
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 7, 10, 8, 10, 10, 5, 10, 8, 7, 10, 5, 6, 10, 10, 10, 8, 10, 10, 8, 1
gen 799
Enter any key character to display boards
```

Вводим любой символ, открывается интерфейс:





Вывод

В результате выполнения данного расчетно-графического задания была реализована игра «The Snake» с помощью генетического алгоритма, имеющая графический интерфейс. В настоящем отчете описан алгоритм решения и приведен итоговый код с пояснениями.