|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления

КАФЕДРА Системы обработки информации и управления\_\_\_\_\_\_\_\_\_\_\_

**ОТЧЕТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4**

**По дисциплине «Методы поддержки принятия решений»**

**Решение оптимизационных задач с помощью генетического алгоритма**

Студент ИУ5-75б  А.Е. Сорокин

(Группа) (Подпись, дата) (И.О.Фамилия)

Преподаватель  Ю.Н. Кротов

(Подпись, дата) (И.О.Фамилия)

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва – 2023

**Цель работы:** ознакомиться с методами и этапами обучения генетических алгоритмов, изучить несколько моделей из направления эволюционных вычислений.

**Вариант:** 9

**Задание:**

I. Использовать генетический алгоритм для оптимизации раскладки клавиатуры для заданных слов: контрпродуктивно, многомерное, многопоточное. Визуализировать макет клавиатуры.

II. Использовать генетический алгоритм для обучения интеллектуальных агентов. Генетический алгоритм будет обучаться под игру Ping Pong, реализованную на JavaScript.

1. Обучить интеллектуального агента с помощью генетического алгоритма в среде без графического интерфейса на Python. В случае необходимости модифицировать код для получения лучших результатов.
2. Проверить качество агента в первоначальной среде на JavaScript.

Критерием выполнения II части лабораторной работы является продолжительность жизни интеллектуального агента по результатам 3 партий. Среднее значение должно равняться не менее 30 секунд. **При этом агент НЕ должен бездействовать**.

**1) Выполнение работы**

**Часть 1: Раскладка клавиатуры**

# ЗДЕСЬ ВПИСЫВАЕТЕ СЛОВА ПО ВАРИАНТУ

messages = [np.random.choice(['контрпродуктивно', 'многомерное', 'многопоточное'], size=1000)]

# тут необходимая подготовка, чтобы быстро считать функцию приспособленности

df = pd.DataFrame(dict(msg=np.concatenate(messages)))

df.msg = df.msg.str.lower()

df.msg = df.msg.str.replace('ё', 'е').str.replace(u'\xa0', u' ').str.replace(u'\-', u' ')

df.msg = df.msg.str.replace('[^a-zа-я0-9\s?,.!]', '')

sequence = list(''.join(df[~df.msg.isna()].msg.values))

charmap = np.unique(sequence)

diffs\_mask = df[~df.msg.isna()].msg.str.len().cumsum().values.copy()[:-1] - 1

diffs\_boolean\_mask = np.ones(len(sequence)-1, dtype=np.bool8)

diffs\_boolean\_mask[diffs\_mask] = 0

sequence = list(''.join(df[~df.msg.isna()].msg.values))

bisequence = (pd.Series(sequence[:-1])+pd.Series(sequence[1:]))

BISEQUENCE\_FREQS = bisequence.loc[diffs\_boolean\_mask].value\_counts().reset\_index()

BISEQUENCE\_FREQS.columns = ['biseq', 'freq']

BISEQUENCE\_FREQS

Этот код генерирует DataFrame (BISEQUENCE\_FREQS), который отражает частоты последовательных пар символов в предварительно обработанных сообщениях. Каждая строка в DataFrame показывает биграмму (biseq) и её частоту (freq) в сообщениях.

biseq: Последовательная пара символов (биграмма).

freq: Частота соответствующей биграммы в сообщениях.

Самая часто встречающаяся биграмма - "но" с частотой 1662.

Вторые по частоте биграммы - "мн", "ог", "го", "ое", каждая с частотой 662.

Остальные биграммы следуют в порядке убывания.

Здесь указываем раскладку клавиатуры по строкам:

FIRST\_ROW = [list('1234567890')]

SECOND\_ROW = [list('йцукенгшщзх')]

THIRD\_ROW = [list('фывапролджэ')]

FOURTH\_ROW = [list('ячсмитьбю')]

FIFTH\_ROW = [list(', .\n')]

ROWS = [FIRST\_ROW, SECOND\_ROW, THIRD\_ROW, FOURTH\_ROW, FIFTH\_ROW]

Здесь указывается расположение (координатами X, Y) клавиш из строк, указанных выше на макете клавиатуры.

KEYBINDS = \

[

# first row

[(68,68), (174, 68), (280, 68), (385, 68), (485, 68),

 (585, 68), (685, 68), (790, 68), (905, 68), (1010, 68)],

# second row

[(60, 201),(155, 201),(255, 201),(345, 201),(445, 201),

 (540, 201),(635, 201),(730, 201),(820, 201),(920, 201),(1015, 201)],

# third row

[(60, 350),(155, 350),(255, 350),(345, 350),(445, 350),

 (540, 350),(635, 350),(730, 350),(820, 350),(920, 350),(1015, 350)],

# fourth row

[(155, 500),(255, 500),(345, 500),(445, 500),

 (540, 500),(635, 500),(730, 500),(820, 500),(920, 500)],

# fifth row

[(224, 645),(530, 645),(855, 645),(980, 645)],

]

def generate\_one():

    # mapper это наша хромосома одной особи

    mapper = {}

    for k, row in enumerate(ROWS):

        for chng, row\_lang in enumerate(row):

            for i, s in enumerate(row\_lang):

                mapper[s] = (KEYBINDS[k][i][0]//10, KEYBINDS[k][i][1]//10)

    return mapper

def plot\_keyboard(mapper, generation=0):

    keyboard\_img = Image.open('keyboard.png').convert('RGB')

    d = ImageDraw.Draw(keyboard\_img)

    font = ImageFont.truetype("Roboto-Bold.ttf", 30)

    for s, v in mapper.items():

        if s=='\n':

            s = 'Ent'

        if s==' ':

            s = '\_\_'

        x, y = v[0]\*10, v[1]\*10

        d.text((x, y), s, font=font, fill=(255, 255, 255, 255))

    return keyboard\_img.resize((500, 250))

plot\_keyboard(generate\_one())



Этот код тестирует вывод стандартной раскладки клавиатуры и настраивает расположение клавиш на макете, чтобы в дальнейшем можно было поменять лишь строки клавиш (Порядок).

# функция приспособленности должна минимизировать расстояние,

# требуемое для написания слов

def get\_scores(population):

    scores = []

    for new\_mapper in population:

        cache = {}

        for i in charmap:

            for j in charmap:

                cache[i+j] = \

                    np.square(np.array(new\_mapper.get(i, [-100, -100])) -

                              np.array(new\_mapper.get(j, [-100, -100]))).sum()\*\*0.5

        weights = BISEQUENCE\_FREQS.biseq.map(cache)

        scores.append(BISEQUENCE\_FREQS['freq'].dot(weights))

    return scores

def mutation(thingey, mutation\_rate=0.05):

    # мутация меняет местами mutation\_rate\*100 % клавиш друг с другом

    keys = list(thingey.keys())

    values = list(thingey.values())

    mutation = (np.random.random(size=len(values))<mutation\_rate).nonzero()[0]

    swap\_index\_to = np.random.choice(len(values),

                                     size=len(values),

                                     replace=False)

    new\_values = np.array(values)

    for i, j in zip(mutation, swap\_index\_to[mutation]):

        new\_values[[i,j]] = new\_values[[j,i]]

    return {k:v for k,v in zip(keys, new\_values)}

def crossover(thingeyA, thingeyB):

    # скрещивание между двумя особями заменяет n-ое количество клавиш 1ой особи

    # n-ым количеством клавиш второй особи с сохранением

    # порядка их встречаемости на раскладке (приоритет сверху-вниз, слево-направо)

    keysA = list(thingeyA.keys())

    valuesA = list(thingeyA.values())

    keysB = list(thingeyB.keys())

    valuesB = list(thingeyB.values())

    ranksA = np.argsort(keysA)

    keysA = np.array(keysA)[ranksA].copy()

    keysB = np.array(keysB)[ranksA].copy()

    valuesA = np.array(valuesA)[ranksA].copy()

    valuesB = np.array(valuesB)[ranksA].copy()

    offset = np.random.randint(1, max(2, len(valuesA)-1))

    offspring = {k:v for k, v in zip(keysA[:offset], valuesA[:offset])}

    keysO = list(offspring.keys())

    valuesO = list(offspring.values())

    keys\_rest = list(filter(lambda k: k not in offspring, thingeyB.keys()))

    valuesRest = valuesA[offset:]

    values = valuesB[offset:]

    ranking = (

        values +

        (np.max(values)+1)\*\*np.arange(len(values[0])).reshape(1, -1)

               ).sum(axis=1).argsort()

    for k, v in zip(keys\_rest, valuesRest[ranking]):

        offspring[k] = v

    # проверка что клавиши не потерялись и не задублировались

    assert set(offspring.keys()

    ).symmetric\_difference(thingeyA.keys()).\_\_len\_\_()==0

    assert set(map(tuple, offspring.values())

    ).symmetric\_difference(set(map(tuple, thingeyA.values()))).\_\_len\_\_()==0

    return offspring

def generate\_initial(POPULATION\_SIZE, mrate=0.0):

    # первая раскладка неизменённая (может быть лучшее решение)

    population = [mutation(generate\_one(), mutation\_rate=0.0)]

    # остальные со случайными перестановками клавиш

    for \_ in range(max(0, POPULATION\_SIZE - 1)):

        population.append(mutation(generate\_one(), mutation\_rate=mrate))

    return population

def generate\_new\_population(population, scores,

                            population\_size=10,

                            elite\_topk=1,

                            random\_size=2,

                            ):

    new\_population = []

    # элитизм

    for topK in range(elite\_topk):

        new\_population.append(population[np.argsort(scores)[topK]].copy())

    # трансформация функции приспособленности

    # оценка -> ранг -> экспоненцирование -> нормализация

    scores\_ranks = scores.copy()

    for i, r in enumerate(np.argsort(scores)):

        scores\_ranks[r] = i

    scores\_ranks = np.exp(-np.array(scores\_ranks)/len(scores\_ranks)\*4)

    scores\_ranks /= scores\_ranks.sum()

    # рулеточный отбор особей для продолжения потомства

    for \_ in range(max(0, population\_size - elite\_topk - random\_size)):

        parentA, parentB = np.random.choice(len(scores), size=2, replace=True, p=scores\_ranks)

        new\_population.append(mutation(crossover(population[parentA], population[parentB])))

    # добавление особей со стороны

    for \_ in range(max(0, random\_size)):

        if np.random.random() < 0.5:

            # полностью случайная раскладка с перемешанными клавишами

            new\_population.append(mutation(generate\_one(), 1.0))

        else:

            # случайно изменённая лучшая (hill climbing)

            new\_population.append(mutation(new\_population[0]))

    return new\_population

POPULATION\_SIZE = 200

ELITISM\_TOPK = 10

RANDOM\_SIZE = 100

NUM\_GENERATIONS = 25

NUM\_RESTARTS = 10

bscore = np.inf

stats = []

for restart in range(NUM\_RESTARTS):

    print('Рестарт: %d' % (restart+1))

    population = generate\_initial(POPULATION\_SIZE, 1.0)

    for generation in range(NUM\_GENERATIONS):

        scores = get\_scores(population)

        population = generate\_new\_population(population, scores,

                                            POPULATION\_SIZE,

                                            ELITISM\_TOPK,

                                            RANDOM\_SIZE)

        stats.append((generation, min(scores), max(scores), np.mean(scores)))

        if min(scores) < bscore:

            bscore = min(scores)

            bimg = plot\_keyboard(population[0], generation)

            print(('Поколение: %d\tЛучшее расстояние: %.1f\t'

                   'Худшее расстояние: %.1f\t'

                   'Среднее расстояние в популяции: %.1f')%stats[-1])

Выше представлена реализация генетического алгоритма для оптимизации клавиатурной раскладки с целью минимизации расстояния между символами, используемыми при печати слов.

Этапы:

Определение клавиатуры и раскладки:

Задаются строки символов на клавиатуре (FIRST\_ROW, SECOND\_ROW, и так далее) и их соответствующие привязки к координатам на клавиатуре (KEYBINDS).

Генерация начальной популяции:

generate\_one создает случайное отображение символов на клавишах для представления начальной раскладки.

generate\_initial генерирует начальную популяцию, включая одну особь без изменений и остальные со случайными перестановками клавиш.

Определение функций мутации и скрещивания:

mutation выполняет мутацию особи, меняя местами клавиши с определенной вероятностью.

crossover выполняет скрещивание между двумя особями, сохраняя порядок встречаемости клавиш.

Вычисление функции приспособленности:

get\_scores вычисляет оценку приспособленности для каждой особи в популяции на основе расстояния между символами.

Генерация новой популяции:

generate\_new\_population создает новую популяцию с учетом элитизма, рулеточного отбора и случайного добавления особей.

Основной цикл оптимизации:

Производится несколько рестартов для повышения вероятности нахождения лучшей раскладки.

В каждом рестарте выполняется несколько поколений, где особи приспосабливаются к раскладке клавиатуры.

За каждое поколение сохраняются лучшее, худшее и среднее расстояние в популяции.

В конце выполнения алгоритма выводится лучшая найденная раскладка клавиатуры.

Таким образом, проводится оптимизация раскладки клавиатуры с использованием генетического алгоритма, стремясь минимизировать расстояние между самыми используемыми символами для повышения эффективности набора текста.

Как можно увидеть, все буквы слов "контрпродуктивно", "многомерное", "многопоточное" собрались в одном месте. Кроме того, ГА расположил "н" по центру, таким образом обход по буквам, например, слова к->о->н->т->р->п->р->о->д->у->к->т->и->в->н->о минимизируется. После 10 повторных запусков ГА минимальное найденное расстояние оказалось равным 187046.0. Это в два раза меньше, чем у стандартной раскладки клавиатуры.

**2) Часть 2: Генетический алгоритм для игры в PingPong, написанной на JavaScript.**

Загрузим необходимые библиотеки.

Инициализируем глобальные переменные.

Опишем логику игры и возможные действия агента.

Приступим к объявлению функций для обучения алгоритма:

Признаки для интеллектуального агента:

def get\_features(leftPaddle, rightPaddle, ball):

    sensors = [

        np.sign(leftPaddle['y'] - ball['y']),

        np.abs(leftPaddle['y'] - ball['y']) / height,

        np.abs(leftPaddle['x'] - ball['x']) / width,

        np.sign(rightPaddle['y'] - ball['y']),

        np.abs(rightPaddle['y'] - ball['y']) / height,

        np.abs(rightPaddle['x'] - ball['x']) / width,

        np.sign(leftPaddle['dy']),

        np.sign(leftPaddle['dy'])==0,

        np.sign(rightPaddle['dy']),

        np.sign(rightPaddle['dy'])==0,

        np.sign(ball['dx']),

        np.sign(ball['dy']),

        np.sign(ball['x'] - width//2),

        np.sign(ball['y'] - height//2),

        1 # что это?

    ]

    return sensors

%timeit get\_features(leftPaddle, rightPaddle, ball)

def get\_one():

    W = np.random.normal(size=(15, 6))

    return W

def getAction(leftPaddle, rightPaddle, ball, W):

    return (W.T.dot(get\_features(leftPaddle, rightPaddle, ball))).argmax()

getAction(leftPaddle, rightPaddle, ball, get\_one())

**Функция приспособленности:**

def get\_score(W, patience=100):

    restart(leftPaddle, rightPaddle, ball)

    maxScore\_patience = patience

    maxScore\_prev = ball['score']

    action = getAction(leftPaddle, rightPaddle, ball, W)

    for \_ in range(int(2e4)):

        if loop(leftPaddle, rightPaddle, ball) == -1:

            break

        # симуляция запоздалой реакции агента

        if np.random.random() < 0.5:

            action = getAction(leftPaddle, rightPaddle, ball, W)

        apply\_action(leftPaddle, rightPaddle, action)

        if  ball['score'] > maxScore\_prev:

            maxScore\_prev = ball['score']

            maxScore\_patience = patience

        maxScore\_patience -= 1

        if maxScore\_patience < 0:

            break

    return ball['score']

**Этапы генетического алгоритма:**

1) def mutate(W, mutation\_rate=0.02):

    dW = get\_one()

    dM = get\_one() > 0

    return W + dW \* dM \* mutation\_rate

def crossover(W1, W2):

    maskW = np.random.random(W1.shape) < 0.5

    return W1 \* maskW + W2 \* (~maskW)

2) def generate\_random(population, size):

    new\_population = []

    for \_ in range(size):

        if np.random.random() < 0.5:

            new\_population.append(get\_one())

        else:

            new\_population.append(mutate(population[0]))

    return new\_population

def selection(population, scores, topK=2):

    scores = np.array(scores)\*1.

    scores /= scores.sum()

    elitismTopK = np.argsort(scores)[::-1][:topK//2]

    roulleteTopK = np.random.choice(len(scores),

                                    p=scores,

                                    size=topK//2)

    new\_population = [population[i].copy() for i in elitismTopK] + \

                     [population[i].copy() for i in roulleteTopK]

    return new\_population

def breed(population, scores, nChilds=10):

    scores = np.array(scores)\*1.

    scores /= scores.sum()

    parents = np.random.choice(len(scores),

                               p=scores,

                               size=(nChilds, 2))

    new\_population = []

    for parentA, parentB in parents:

        new\_population.append(mutate(crossover(population[parentA],

                                               population[parentB])))

    return new\_population

def get\_new\_population(population, scores, topK=4, randomNum=10):

    return (

    selection(population, scores, topK) + \

    breed(population, scores,

          nChilds=max(0, len(population) - randomNum - topK)) + \

    generate\_random(population, randomNum)

    )

3) def get\_scores(population, patience=100):

    scores = []

    for W in population:

        scores.append(get\_score(W, patience))

    return scores

**Основной блок обучения интеллектуального агента:**

POPULATION\_SIZE = 256

RANDOM\_SIZE = 20

ELITE\_SIZE = 75

NUM\_GENERATIONS = 100

NUM\_REPEATS = 3 # зачем? - повторы теста одного агента

NUM\_RESTARTS = 1

PATIENCE = lambda x: 1000\*((x+2)//2)

best\_thingey = None

best\_score = 0

for n\_restart in range(NUM\_RESTARTS):

    print('='\*50)

    print('Cтарт перезапуска №%d'%(n\_restart+1))

    print('Лучшая пока что: %.1f'%best\_score)

    print('='\*50)

    population = [get\_one() for \_ in range(POPULATION\_SIZE)]

    for generation in range(NUM\_GENERATIONS):

        scores = 1e-10

        for \_ in range(NUM\_REPEATS):

            scores += np.array(get\_scores(population, PATIENCE(generation)))

        scores /= NUM\_REPEATS

        bscore = max(scores)

        scores \*\*= 4 # зачем?

        population = get\_new\_population(population, scores,

                                        topK=ELITE\_SIZE,

                                        randomNum=RANDOM\_SIZE)

        if bscore > best\_score:

            best\_score = bscore

            best\_thingey = np.array(population[0])

            print('Рестарт: %d\tПоколение: %d\tЗначение: %.1f'%(n\_restart+1,

                                                                generation,

                                                                bscore))

            if bscore > 80:

              with open('pingpong\_weights\_%.1f.js'%bscore, 'w') as f:

                f.write('var W = %s;\n'%(json.dumps([[int(1e3\*w)/1e3 for w in W] for W in best\_thingey])))

Параметры обучения:

**Population\_Size** = 256 – размер популяции, количество агентов в популяциях каждого поколения. (Что-то схожее с batch\_size при обучении однослойного персептрона)

**Random\_Size** = 20 – количество новых агентов с полностью случайным генотипом (Весами модели), эти агенты возможно даже не существовали в предыдущих поколениях, они генерируются случайно. Они нужны для выхода генетического алгоритма из «Тупиковой ветки эволюции». Если убрать случайных особей, модель упрется в локальный минимум или локальный максимум.

**Elite\_Size** = 75 – количество лучших агентов из предыдущего поколения переходят в следующее поколение, чтобы сохранить лучшие генотипы.

**Num\_Generations** = 100 – количество поколений при обучении алгоритма (Схожий параметр при обучении однослойного персептрона – количество шагов)

**Num\_Repeats** = 3 – количество повторов тестирования одного агента. Если тестировать каждого агента по 1 разу, неизбежно будут происходить случаи, когда плохой агент, удачно пройдет тест один раз, и наоборот, хороший агент может неудачно пройти тест единожды – это все называется выбросами. Если проводить тестирование агентов по 3 раза и брать среднее количество очков за каждый тест, сглаживаются и отбрасываются выбросы. Однако увеличение количество тестов сильно повышает время обучения алгоритма.

**Num\_Repeats** = 1 – количество запусков генетического алгоритма, когда сбрасываются уже обученные поколения и все рассчитанные веса, сохраняя лишь одного единственного лучшего агента.

**Patience** – терпимость, параметр, определяющий элитизм. Он описывает то, насколько алгоритм будет терпим к агентам, которые были лучшими в предыдущих поколениях, но в текущем проявили себя неудачно. Это дает шанс элитным агентом переживать неудачные поколения и передавать генотип дальше.

**Параметры, которые были изменены и почему:**

**Population\_Size** было 128, увеличили вдвое, что увеличивает вероятность на появления удачных генов вдвое, но и увеличивает время обучения.

**Random\_Size** не был изменен, несмотря на то что размер популяции увеличился вдвое. Вследствие чего пропорция была уменьшена и вероятность застрять в локальных экстремумах обучения. Решение заключается в том, что их видно на первых (20) поколениях обучения, в случае их обнаружения этап обучения просто перезапускаем.

**Elite\_Size** был равен 5, что необоснованно мало для популяции размером в 128 агентов. Было выбрано их количество равное 75, что в пропорции больше в 7 раз от исходного. Чем больше количество элитных агентов в популяции, тем больше вероятность распространения лучших генотипов. Но эти генотипы являются лучшими лишь на данном этапе поколения, увеличение количество элитных агентов до половины (и более) от всего размера популяции может привести к застреванию в локальном максимуме.

**Num\_Restarts** изменено с 5 на 1 – удобнее перезапускать обучение вручную.

**В код добавлено:**

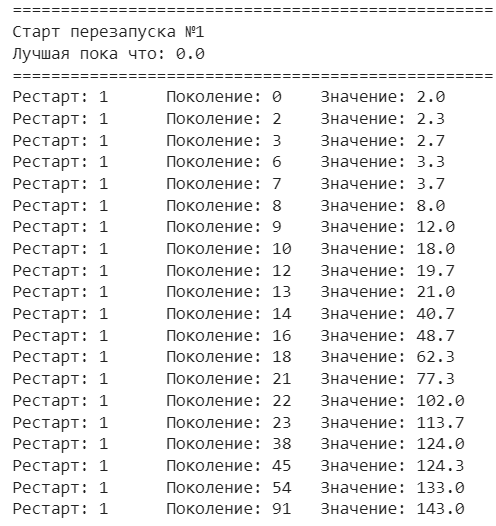
            if bscore > 80:

              with open('pingpong\_weights\_%.1f.js'%bscore, 'w') as f:

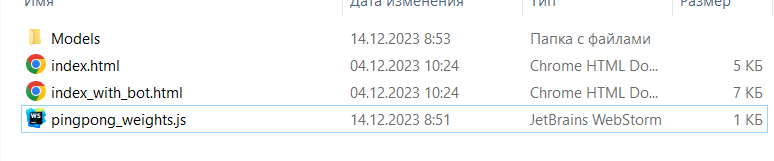
                f.write('var W = %s;\n'%(json.dumps([[int(1e3\*w)/1e3 for w in W] for W in best\_thingey])))

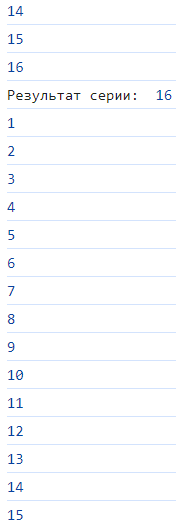
Что позволяет сохранять веса сразу, как только значение агента превысит 80, не дожидаясь завершения обучения.

**Запуск и результаты обучения модели:**



Скачиваем файл расставленными весами агента с Google Colab, добавляем его в ту же папку, в которой хранится игра PingPong:

Запускаем index\_with\_bot.html в браузере и смотрим результаты обученного агента:



Этот вывод показывает количество отскоков в течение игры.

Средний показатель: 16 отскоков за серию

**Вывод:** ознакомился с методами и этапами обучения генетических алгоритмов, изучил несколько моделей из направления эволюционных вычислений.

Использовал генетический алгоритм для оптимизации раскладки клавиатуры, а также для обучения интеллектуальных агентов под игру Ping Pong, реализованную на JavaScript.