МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по учебной практике

Тема: Генетические алгоритмы

Студент гр. 0303 _____ Болкунов В.О.

Руководитель _____ Жангиров Т.Р.

Санкт-Петербург

2022

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студент Болкунов В.О. группы 0303

Тема практики: Генетические алгоритмы

Задание на практику:

Разработать и реализовать программу, решающую одну из оптимизационных задач с использованием генетических алгоритмов (ГА), а также визуализирующую работу алгоритма.

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: 00.07.2020

Дата защиты отчета: 00.07.2020

Студент _____ Болкунов В.О.

Руководитель _____ Жангиров Т.Р.

АННОТАЦИЯ

Цель работы заключается в изучении и применении оптимизационного метода решения задач – Генетических алгоритмов. В ходе работы было описан генетический алгоритм решающий поставленную задачу и написана программа, реализующая решение данной задачи с помощью генетического алгоритма.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задание.

Разработать и реализовать программу, решающую одну из оптимизационных задач (файл “Варианты”) с использованием генетических алгоритмов (ГА), а также визуализирующую работу алгоритма. В качестве языков программирования можно выбрать: C++, C#, Python, Java, Kotlin. Все параметры ГА необходимо определить самостоятельно исходя из выбранного варианта. Разрешается брать один и тот же вариант разным бригадам, но при условии, что будут использоваться разные языки программирования.

Задача.

Вариант 8: Задача коммивояжёра.

Входные данные:

- Количество городов
- Координаты городов.

Расстояние между городами равно евклидову расстоянию между точками

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1. Наличие графического интерфейса (GUI)
2. Возможность ввода данных через GUI или файла
3. Пошаговая визуализация работы ГА. С возможностью перейти к концу алгоритма.
4. Настройка параметров ГА через GUI. Например, размер популяции.
5. Одновременное отображение особей популяции с выделением лучшей особи
6. Визуализация кроссинговера и мутаций в популяции
7. Наличие текстовых логов с пояснениями к ГА
8. Программа после выполнения не должна сразу закрываться, а должна давать возможность провести ГА заново, либо ввести другие данные

2. ПЛАН РАЗРАБОТКИ

2.1 Параметры генетического алгоритма.

1. В качестве **особей** возьмём предполагаемое решение задачи – гамильтонов цикл (города образуют полный граф), а конкретнее перестановку множества городов. Например, есть города *1*, *2*, *3*, тогда массивы [1, 2, 3], [1, 3, 2], [3, 2, 1] будут являться хромосомами в данном ГА.

2. Целевой функцией, или же **приспособленностью** будет длина цикла образуемого вершинами в хромосоме. Соответственно эту функцию требуется минимизировать.

3. Начальную популяцию можно построить, выбрав N случайных перестановок множества городов (N – размер популяции)

4. Выбор родителей может осуществляться с помощью нескольких операторов: **панмиксия**, **турнирный отбор** и **метод рулетки**, так как они обеспечивают приемлемую сходимость к предполагаемым оптимальным решениям, при этом оставляя возможность выжить менее оптимальным решениям, которые потенциально могут дать лучшее решение. По этой причине было решено не использовать *селекцию*, *инбридинг* и *аутбридинг*, так как данная задача многомерная с достаточно сложным ландшафтом, и эти методы могут сойтись к квазиоптимальному решению, либо же наоборот (в случае аутбридинга) сильно осциллировать вокруг оптимального решения.

5. Рекомбинацию в данной задаче нельзя производить классическими методами, так как структура хромосом ограничивается условием задачи (перестановка N вершин графа), и проводя классическую рекомбинацию мы рискуем получить цепочки, в которых вершины повторяются.

- Первым способом рекомбинации для перестановок является метод (оператор) **PMX** (partially mapped crossover). Сначала выбираются аллели, которые будут меняться местами (аналогично двуточечному кроссинговеру). Далее в дочерние хромосомы вставляются эти участки. Остальные гены добавляются по принципу: если гена нет в рекомбинируемом участке, то он остаётся на месте, иначе рекурсивно берётся тот ген, который был заменён геном в рекомбинируемом участке. Рассмотрим этот метод на примере.

Возьмём двух родителей (чертами отделён выбранный для рекомбинации участок)

P_1 = (4 1 8 | 2 7 3 | 5 6)

P_2 = (1 3 7 | 6 5 2 | 8 4)

Создадим детей и вставим аллели, которыми обмениваются родители.

O_1 = (_ _ _ | 6 5 2 | _ _)

O_2 = (_ _ _ | 2 7 3 | _ _)

Далее поставим на место те гены, которые не конфликтуют с новой аллелью.

O_1 = (4 1 8 | 6 5 2 | _ _)

O_2 = (1 _ _ | 2 7 3 | 8 4)

Теперь поставим на пропуски оставшиеся гены. Например, у первой особи в 8ом локусе значение гена было 6, ген с таким же значением был поставлен на место гена 2 (4ый локус), но ген 2 тоже присутствует в разрезе (6ый локус) и заменил собой ген со значением 3 в итоге на 8ом локусе будет ген со значением 3.

По такому же принципу расставим оставшиеся гены.

O_1 = (4 1 8 | 6 5 2 | 7 3)

O_2 = (1 6 5 | 2 7 3 | 8 4)

Существует ещё несколько операторов рекомбинации перестановок: **OX** (order crossover) и **CX** (cycle crossover).

- Рассмотрим принцип работы **OX** метода.

Дочерним особям передаются выбранные аллели. Далее у первого родителя цепочка генов сдвигается таким образом, чтобы она начиналась сразу после выбранной аллели, из этой цепочки вычёркиваются гены выбранной аллели второго родителя, и эта цепочка циклично вставляется в хромосому первой особи сразу после выбранной аллели. Для второй особи проводятся аналогичные действия.

Рассмотрим на примере.

P_1 = (4 1 8 | 2 7 3 | 5 6)

P_2 = (1 3 7 | 6 5 2 | 8 4)

Создадим детей.

O_1 = (_ _ _ | 6 5 2 | _ _)

O_2 = (_ _ _ | 2 7 3 | _ _)

Сдвинем цепочку генов родителей, так чтобы они начинались после разреза.

P_1 : 5 6 4 1 8 | 2 7 3

P_2 : 8 4 1 3 7 | 6 5 2

Удалим из них гены, которые присутствуют в разрезе другого родителя.

P_1 : 4 1 8 7 3

P_2 : 8 4 1 6 5

Вставим цепочки в пустые локусы дочерних особей.

O_1 = (8 7 3 | 6 5 2 | 4 1)

O_2 = (1 6 5 | 2 7 3 | 8 4)

- Рассмотрим работу оператора **CX**.

Сначала выбирается первый ген у какого-либо родителя. Далее в особь вставляется ген второго родителя, находящийся на этом же локусе (вставляется на его позицию в первом родителе), и так далее до тех пор, пока не образуется цикл. В оставшиеся локусы вставляются гены второго родителя. В данном методе получается, что хромосомы обмениваются своими циклами (родители относительно друг друга образуют что-то вроде подстановки).

Рассмотрим пример

P_1 = (4 1 8 2 7 3 5 6)

P_2 = (1 3 7 6 5 2 8 4)

O_1 = (1 _ _ _ _ _ _ 4) – здесь мы выбрали первый ген второго родителя. Т.к. у первого родителя значение гена в этом локусе 4, ставим этот ген на его место у выбранного родителя.

O_1 = (1 _ _ 6 _ _ _ 4) – далее гену 4 второго родителя соответствует ген 6 первого.

O_1 = (1 _ _ 6 _ 2 _ 4) – продолжим построение цикла.

O_1 = (1 3 _ 6 _ 2 _ 4) – мы получили цикл 1 4 6 2 3

Дополним генами первого родителя

O_1 = (1 3 8 6 7 2 5 4)

В методе **CX** существует возможность, что дочерние особи будут в точности копировать родительские хромосомы, это возникнет в случае если циклом будет являться вся подстановка. Например

P_1 = (4 1 8 2 7 3 5 6)

P_2 = (1 3 7 6 4 2 8 5)

Тогда потомки будут следующие:

O_1 = (4 1 8 2 7 3 5 6)

O_2 = (1 3 7 6 4 2 8 5)

6. Возможные **мутации** для перестановки:

- **Мутация вставкой**. Выбираются два случайных локуса *i*, *j*. И ген в локусе *j* вставляется сразу после *i*-го. При этом остальные гены сдвигаются.

Например: (**1** 2 3 **4** 5) -> (1 4 2 3 5)

Либо если сдвигать вправо: (**1** 2 3 **4** 5) -> (2 3 1 4 5)

- **Мутация обменом**. Два гена на случайных локусах просто меняются местами.

Например: (1 **2** 3 4 **5**) -> (1 5 3 4 2)

- **Мутация инверсией**. Аллель между двумя выбранными локусами переворачивается.

Например: (**1** 2 3 4 **5**) -> (1 4 3 2 5)

7. Отбор в новую популяцию можно проводить классическими методами. Например, **отбор усечением** и **элитарный** отбор (отбор вытеснением сложно применять из-за особенностей сравнения цикличных перестановок).

8. Помимо канонического ГА, для данной задачи можно применить **модификации,** например **генитор**, **метод прерывистого равновесия**.

2.1 План взаимодействия с программой и GUI

Примерный скетч графического интерфейса представлен на рисунке 1.

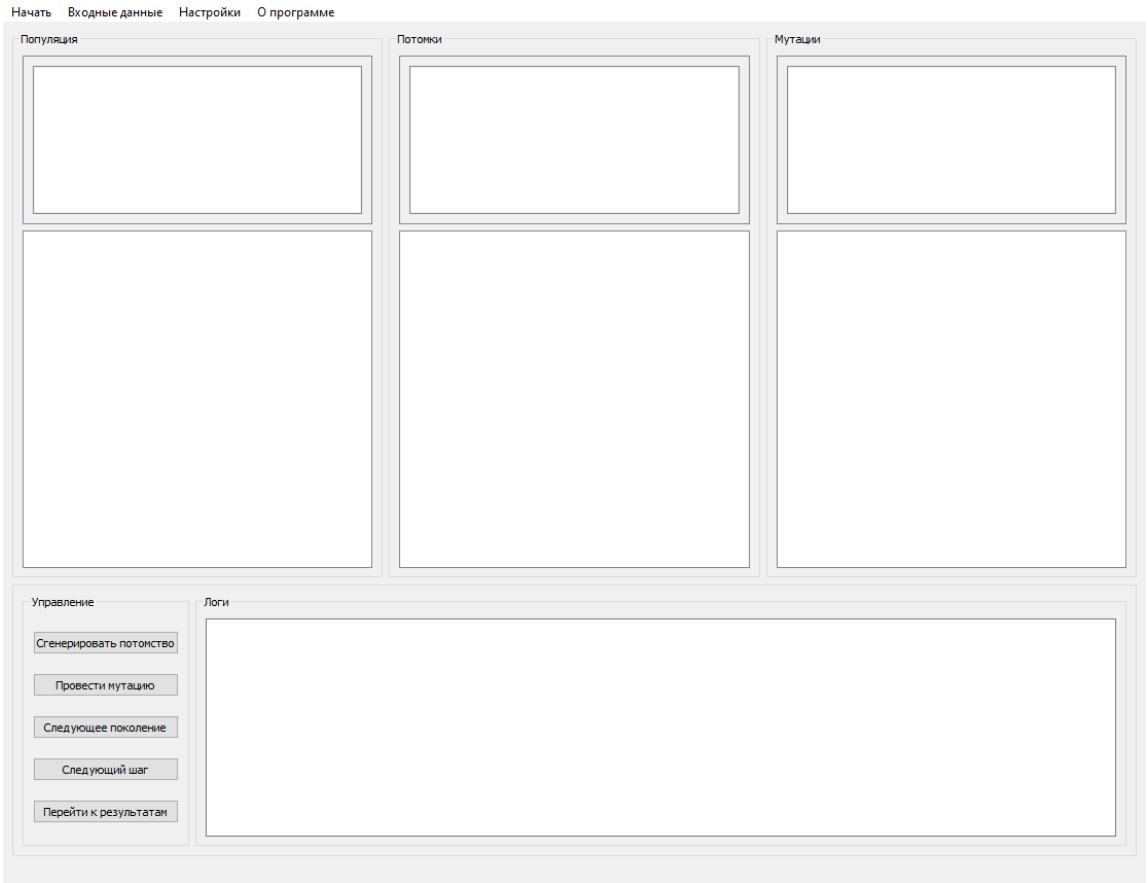

Рисунок 1: скетч GUI

Входные данные можно будет ввести через список во всплывающем окне, либо выбрав файл с данными в нужном формате.

В разделе «настройки» пользователь может выбрать модификацю ГА, и различные параметры: операторы отбора родителей, рекомбинации, мутации и отбора в новую популяцию. Для некоторых операторов также будут настройки параметров.

В разделах «популяция», «потомки» и «мутации» находятся списки с соответствующими особями, а под ними визуальное отображение выбранной особи – построенный граф.

Для управления используются кнопки на панели управления.

В разделе логов выводится информация о выполнении алгоритма.

3. РЕАЛИЗАЦИЯ

3.1 Утилитарный модуль

Содержит вспомогательные классы и фукнции.

Функция ввода данных из файла

```
def file_input(path: str) -> list[Town]
```

Логгер

Класс-синглтон для логгирования. Наследуется от QObject чтобы иметь возможность отправлять сигналы с логами в присоединённые объекты.

```
class Logger(QObject)
```

Свойства (статические)

- Ссылка на объект логгера

```
_instance = None
```

- Мьютекс логгера

```
_lock = threading.Lock()
```

- Сигнал лога

```
logSignal = pyqtSignal(str)
```

Методы

- Конструктор

```
def __init__(self)
```

- Получение объекта логгера (статический приватный метод)

```
def _get() -> Logger
```

- Метод вывода лога. Испускает сигнал со строкой лога.

```
def log(msg: str) -> None
```

- Метод присоединения слота к сигналу логгера

```
def connect(slot: Callable[[str], None]) -> None
```

3.2 Модель

Кодовая база модели задачи условно разделена на:

- Ядро: классы, представляющие точку маршрута, регион (набор городов), решение и популяцию (набор решений).
- Операторы генетического алгоритма: функции для выбора родителей, рекомбинации, мутации, выбора потомства.
- Сами алгоритмы: абстрактный ГА, Классический, Генитор, и Метод прерывистого равновесия.

3.2.1 Классы ядра.

Город: представление точки маршрута (вектор).

```
class Town
```

Свойства

- x-координата

```
x: float
```

- y-координата

```
y: float
```

Методы

- Конструктор

```
def __init__(self, x: float, y: float)
```

- Метод для вычисления расстояния с другим городом

```
def dist(self, t: Town) -> float
```

- Преобразования к строке

```
def __str__(self) -> str
def __repr__(self) -> str
```

Регион: образует список городов и рассчитывает матрицу расстояний

class Region(list[Town])

Свойства

- Матрица расстояний

dists: np.ndarray

Методы

- Конструктор

def __init__(self, towns: list[Town])

- Преобразования к строке

def __str__(self) -> str

def __repr__(self) -> str

Решение: класс для особей – наследуется от списка, хранит решение в виде последовательности номеров городов.

class Solution(list[int])

Свойства

- Регион в котором решается задача

reg: Region

Методы

- Стандартный конструктор (пустое решение)

def __init__(self, reg: Region)

- Конструктор с генерацией случайного решения

def __init__(self, reg: Region, length: int)

- Конструктор преобразования списка номеров к решению

def __init__(self, reg: Region, lst: list[int])

- Целевая функция

def F(self) -> float

- Обратная к целевой (1 / F)

def rF(self) -> float

- Циклический сдвиг решения

def shift(self, n: int) -> Solution

- Нормализация (сдвиг чтобы решение начиналось в городе 0)

def normalized(self) -> Solution

- Копия особи

def copy(self) -> Solution

- Преобразования к строке

def __str__(self) -> str

def __repr__(self) -> str

Популяция: класс для набора особей (популяции) – наследуется от списка.

class Population(list[Solution])

Свойства

- Регион в котором решается задача

reg: Region

Методы

- Стандартный конструктор (пустая популяция)

def __init__(self, reg: Region)

- Конструктор с генерацией популяции

def __init__(self, reg: Region, psize: int)

- Конструктор преобразования списка особей к популяции

def __init__(self, reg: Region, lst: list[Solution])

- Получение особи с минимальной длиной

```
def min(self) -> Solution
```

- Получение особи с максимальной длиной

```
def max(self) -> Solution
```

- Среднее значение целевой функции

```
def meanF(self) -> float
```

- Среднее значение обратной к целевой функции

```
def meanrF(self) -> float
```

- Нормализация всей популяции

```
def normalized(self) -> Population
```

- Копия популяции

```
def copy(self) -> Population
```

- Сортировка популяции

```
def sorted(self) -> Population
```

- Преобразования к строке

```
def __str__(self) -> str
def __repr__(self) -> str
```

3.2.2 Операторы ГА. (реализация операторов описанных в разделе 2.1)

Операторы помимо своих параметров (популяция/особь), принимают также ссылку на исполнителя ГА, откуда они (операторы которым нужны какие-то параметры) берут переданные им параметры. Это было сделано для универсализации интерфейса операторов, т.е. чтобы любой ГА мог одинаково вызывать любой оператор (передавая одни и те же аргументы)

Операторы отбора родителей: создают пары особей для рекомбинации

```
def panmixion(pop: Population, ga: GA = None) -> list[tuple[Solution, Solution]]
def tournament(pop: Population, ga: GA) -> list[tuple[Solution, Solution]]
def roulette(pop: Population, ga: GA = None) -> list[tuple[Solution, Solution]]
```

Операторы рекомбинации: создают двух потомков от двух родителей

```
def pmx(p1: Solution, p2: Solution, ga: GA) -> tuple[Solution, Solution]
def ox(p1: Solution, p2: Solution, ga: GA) -> tuple[Solution, Solution]
def cx(p1: Solution, p2: Solution, ga: GA) -> tuple[Solution, Solution]
```

Операторы мутации: производят мутацию особи

```
def swap(o: Solution, ga: GA) -> Solution
def insert(o: Solution, ga: GA) -> Solution
def inverse(o: Solution, ga: GA) -> Solution
```

Операторы отбора потомков: производят отбор в новую популяцию

```
def trunc(pop: Population, ga: GA) -> Population
def elite(pop: Population, ga: GA) -> Population
```

3.2.3 Исполнители ГА: представляют модификацию генетического алгоритма

Класс параметров: содержит настраиваемые параметры ГА и параметры, которые он передаёт в операторы

```
class Params
```

Свойства (настраиваемые)

- Размер популяции (в т.ч. начальный)

```
self.psize: int
```

- Максимальное поколение (до которого идёт метод результата)

  self.maxGen: int
- Вероятность рекомбинации

  self.rprob: float
- Вероятность мутации

  self.mprob: float
- Размер турнира для турнирного отбора

  self.tsize: int
- Пороговое значение для отбора усечением

  self.threshold: float

Свойства (передаваемые операторам)

- Начало аллели кроссинговера

  self.cstart: int
- Размера аллели кроссинговера

  self.csize: int
- Первый ген мутации

  self.mgen1: int
- Второй ген мутации

  self.mgen2: int

Абстрактный ГА: создаёт случайную популяцию, задаёт интерфейс для конкретных реализаций ГА

class GA

Свойства

- Привязанный регион

reg: Region

- Текущая популяция

population: Population

- Список пар родителей

parents: list[tuple[Solution, Solution]]

- Популяция дочерних особей

children: Population

- Популяция с мутированными дочерними особями

mutChildren: Population

- Промежуточная популяция

tempPop: Population

- Популяция отобранная из промежуточной

offspring: Population

Методы

- Конструктор принимающий ссылки на все операторы

```
def __init__(
    self,
    pSelector: Callable[[Population, GA], list[tuple[Solution, Solution]]] = None,
    recombinator: Callable[[Solution, Solution, GA], tuple[Solution, Solution]] = None,
    mutationer: Callable[[Solution, GA], Solution] = None,
    oSelector: Callable[[Population, GA], Population] = None
)
```

- Инициализация региона и генерация начальной популяции

```
def start(self, reg: Region) -> None
```

- Выбор родителей (родительских пар)

```
def parentsSelect(self) -> None
```

- Запуск кроссинговера на родительских парах

```
def crossover(self) -> None
```

- Проведение мутации на полученных дочерних особях

```
def mutation(self) -> None
```

- Построение промежуточной популяции и отбор в следующую

```
def offspringSelect(self) -> None
```

- Замена текущей популяции следующей и инкремент поколения

```
def newPopulation(self) -> None
```

- Получение следующего поколения (последовательный вызов всех предыдущих методов)

```
def nextGeneration(self) -> None
```

- Проверка существования параметра (выбрасывает исключение если параметр не найден)

```
def checkParam(self, attr: str) -> None
```

- Преобразования к строке

```
def __str__(self) -> str
def __repr__(self) -> str
```

Классический ГА: работает по классической модели. Проводит отбор родителей, кроссинговер, мутацию, отбор потомков с помощью переданных операторов.

```
class ClassicGA(GA)
```

Генитор: на каждом шаге выбирает двух родителей, создаёт одного потомка, который заменяет самую слабую особь.

```
class Genitor(GA)
```

Метод прерывистого равновесия: отбор родителей осуществляется панмиксией, в следующую популяцию отбираются особи у которых целевая функция меньше средней по популяции. Наследуется от классического ГА, переопределяя лишь метод выбора потомков.

class InterBalance(ClassicGA)

3.3 Графический Интерфейс

Элементы графического интерфейса разделены на отдельные компоненты (виджеты и диалоги), что позволяет переиспользовать и аггрегировать их в другие виджеты и окна.

3.3.1 Виджеты

PopulationWidget: виджет для отображения особей популяции. Содержит список особей с возможностью выбора конкретной и график с маршрутом решения выбранной особи.

class PopulationWidget(QWidget):

Свойства

- Виджет списка особей

list: QListWidget

- Виджет графика

canvas: FigureCanvasQTAgg

- Переданная популяция

pop: Population

Методы

- Конструктор с именем виджета

def __init__(self, name: str)

- Слот для отрисовки выбранной особи

def drawSolution(self) -> None

- Очистка виджета

def clear(self) -> None

- Метод для установки набора особей

def setPopulation(self, pop: Population)

LoggerWidget: виджет для отображения поступающих логов (текстовое поле только для чтения)

class LoggerWidget(QGroupBox):

Методы

- Конструктор

def __init__(self)

- Печать переданной строки

def print(self, s)

ControlWidget: виджет содержащий кнопки управления

class ControlWidget(QGroupBox):

Свойства

- Кнопки управления

offspring: QPushButton

mutate: QPushButton

next: QPushButton

forceNext: QPushButton

results: QPushButton

Методы

- Конструктор

def __init__(self)

3.3.2 Диалоговые окна

Диалог ввода: позволяет ввести список городов вручную

```
class InputDialog(QDialog):
```

Свойства

- Спин-бокс с количеством городов

```
self.count: QSpinBox
```

- Кнопка подтверждения

```
self.btn: QPushButton
```

- Таблица ввода координат

```
self.table: QTableWidget
```

Методы

- Конструктор

```
def __init__(self, parent=None)
```

- Слот подтверждения ввода

```
def ret(self) -> None
```

Диалог настроек: позволяет выбрать настройки ГА. Делает неактивными параметры не соответствующие выбору некоторых ГА/операторов.

```
class SettingsDialog(QDialog):
```

Свойства

- Выбор модификации ГА

```
classic: QRadioButton
genitor: QRadioButton
interbalance: QRadioButton
```

- Выбор оператора отбора родителей

```
panmixion: QRadioButton
tournament: QRadioButton
roulette: QRadioButton
```

- Выбор оператора рекомбинации

pmx: QRadioButton

ox: QRadioButton

cx: QRadioButton

- Выбор оператора мутации

swap: QRadioButton

insert: QRadioButton

inverse: QRadioButton

- Выбор оператора отбора в новую популяцию

trunc: QRadioButton

elite: QRadioButton

- Размер популяции

psize: QSpinBox

- Максимальное число поколений

maxGen: QSpinBox

- Вероятность рекомбинации

rprob: QDoubleSpinBox

- Вероятность мутации

mprob: QDoubleSpinBox

- Размер турнира для турнирного метода

tsize: QSpinBox

- Пороговое число для отбора усечением

threshold: QDoubleSpinBox

Методы

- Конструктор

```
def __init__(self, parent=None):
```

- Слот подтверждения ввода

```
def ret(self):
```

3.3.3 Главное окно

Объединяет в себе виджеты в соответствии со скетчем. Виджеты объединены через QSplitter, что позволяет менять размеры каждого элемента.

```
class MainWindow(QMainWindow):
```

Методы

- Конструктор

```
def __init__(self):
```

- Проверка того что данные введены и ГА настроен

```
def checkSetup(self):
```

- Проверка запущенного ГА

```
def checkGenerated(self):
```

- Слот кнопки генерации потомства

```
def onChildren(self):
```

- Слот кнопки мутации

```
def onMutate(self):
```

- Слот кнопки следующего поколения

```
def onNext(self):
```

- Слот кнопки следующего шага (сразу даёт следующее поколение)

```
def onForceNext(self):
```

- Слот нажатия ввода данных

```
def onInput(self):
```

- Слот нажатия ввода из файла (запускает диалог выбора файла)

```
def onFile(self):
```

- Слот нажатия настроек (запускает диалог настроек)

```
def onSettings(self):
```

- Слот нажатия кнопки запуска

```
def onStart(self):
```

- Слот нажатия кнопки о программе

```
def onInfo(self):
```

4. ТЕСТИРОВАНИЕ

Скрипты с юнит-тестами находятся в папке *tests*.

4.1 Тестирование вспомогательного модуля

4.1.1 Логгер *(находится в тесте с виджетом логгера)*

В данном тесте проверяется работа логгера и соответствующего виджета. Ввод лого производится из нескольких потоков.

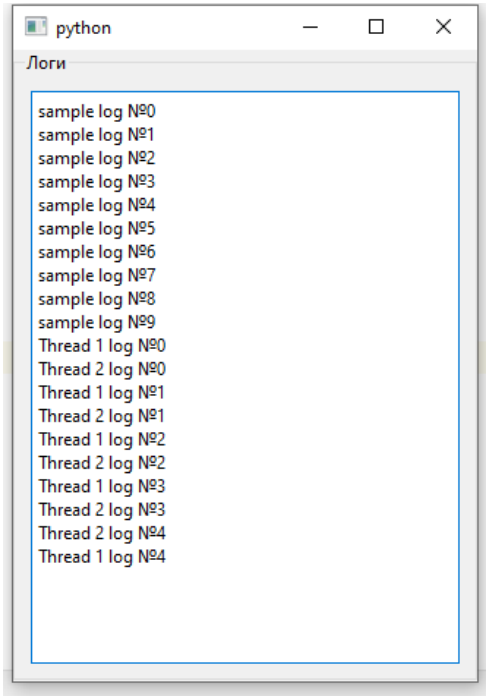

Рисунок 2: Логгер

4.1.2 Ввод из файла.

В данном тесте осуществляется ввод данных о задаче из файла

```
    Города: [{x: 1.0, y: 2.0}, {x: 3.0, y: 4.0}, {x: 5.0, y:
6.0}, {x: 3.14, y: 2.73}, {x: 0.0, y: 0.0}]
    Расстояния:
     [[0.         2.82842712 5.65685425 2.26108381 2.23606798]
     [2.82842712 0.         2.82842712 1.27769323 5.        ]
     [5.65685425 2.82842712 0.         3.76198086 7.81024968]
     [2.26108381 1.27769323 3.76198086 0.         4.16082924]
     [2.23606798 5.         7.81024968 4.16082924 0.        ]]
```

4.2 Тестирование Модели

4.2.1 Генерация популяции *(файл generations.py)*

В данном тесте по списку городов генерируется начальная популяция.

```
Регион:
Города: [{x: 1, y: 0}, {x: 2, y: 2}, {x: 3, y: 4}, {x: 10, y: 5},
{x: 2.73, y: 3.14}]
Расстояния:
 [[ 0.          2.23606798  4.47213595 10.29563014  3.58503835]
 [ 2.23606798  0.          2.23606798  8.54400375  1.35369864]
 [ 4.47213595  2.23606798  0.          7.07106781  0.90138782]
 [10.29563014  8.54400375  7.07106781  0.          7.50416551]
 [ 3.58503835  1.35369864  0.90138782  7.50416551  0.        ]]

Популяция:
{хромосома: [0, 4, 1, 3, 2], F: 25.025944503299343}
{хромосома: [0, 4, 3, 2, 1], F: 22.632407630815543}
{хромосома: [0, 3, 4, 2, 1], F: 23.173319424754396}
{хромосома: [0, 4, 2, 3, 1], F: 22.337565707597463}
{хромосома: [0, 2, 4, 3, 1], F: 23.657761006584714}

Особь с минимальной длиной:  {хромосома: [0, 4, 2, 3, 1], F:
22.337565707597463}
```

4.2.2 Отбор родителей *(файл parents_selections.py)*

В данном тесте из популяции различными способами отбираются родительские пары.

```
Популяция:
{хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}
{хромосома: [0, 2, 1, 4, 3], F: 18.89495124262827}
{хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}
{хромосома: [0, 4, 1, 2, 3], F: 17.408150440650402}
{хромосома: [0, 1, 4, 3, 2], F: 14.986161090931175}

Панмиксия:
({хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}, {хромосома:
[0, 2, 1, 4, 3], F: 18.89495124262827})
({хромосома: [0, 2, 1, 4, 3], F: 18.89495124262827}, {хромосома:
[0, 2, 1, 4, 3], F: 18.89495124262827})
({хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}, {хромосома:
[0, 1, 4, 3, 2], F: 14.986161090931175})
({хромосома: [0, 4, 1, 2, 3], F: 17.408150440650402}, {хромосома:
[0, 4, 1, 2, 3], F: 17.408150440650402})
({хромосома: [0, 1, 4, 3, 2], F: 14.986161090931175}, {хромосома:
[0, 1, 4, 3, 2], F: 14.986161090931175})

Турнирный отбор:
({хромосома: [0, 1, 4, 3, 2], F: 14.986161090931175}, {хромосома:
[0, 4, 1, 2, 3], F: 17.408150440650402})
({хромосома: [0, 4, 1, 2, 3], F: 17.408150440650402}, {хромосома:
[0, 1, 4, 3, 2], F: 14.986161090931175})
```

```
({хромосома: [0, 4, 1, 2, 3], F: 17.408150440650402}, {хромосома:
[0, 4, 1, 2, 3], F: 17.408150440650402})
({хромосома: [0, 4, 1, 2, 3], F: 17.408150440650402}, {хромосома:
[0, 4, 1, 2, 3], F: 17.408150440650402})
({хромосома: [0, 4, 1, 2, 3], F: 17.408150440650402}, {хромосома:
[0, 4, 1, 2, 3], F: 17.408150440650402})

Рулетка:
({хромосома: [0, 2, 1, 4, 3], F: 18.89495124262827}, {хромосома:
[0, 4, 1, 3, 2], F: 17.573150606284717})
({хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}, {хромосома:
[0, 1, 4, 3, 2], F: 14.986161090931175})
({хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}, {хромосома:
[0, 4, 1, 3, 2], F: 17.573150606284717})
({хромосома: [0, 1, 4, 3, 2], F: 14.986161090931175}, {хромосома:
[0, 1, 4, 3, 2], F: 14.986161090931175})
({хромосома: [0, 4, 1, 3, 2], F: 17.573150606284717}, {хромосома:
[0, 4, 1, 3, 2], F: 17.573150606284717})
```

4.2.3 Рекомбинация *(файл recombinations.py)*

В данном тесте проводится рекомбинация генов двух родителей различными операторами рекомбинации.

```
Родители:
{хромосома: [0, 4, 1, 8, 2, 7, 3, 5, 6], F: 83.68121332735672}
{хромосома: [0, 1, 3, 7, 6, 5, 2, 8, 4], F: 62.60655252723025}

Особи полученные с помощью PMX:
{хромосома: [6, 5, 2, 7, 3, 0, 4, 1, 8], F: 65.11702777429367}
{хромосома: [2, 7, 3, 8, 4, 0, 1, 6, 5], F: 75.31414315507716}
Особи полученные с помощью PMX(разрез выходит за пределы
хромосомы):
{хромосома: [8, 4, 0, 6, 1, 5, 2, 7, 3], F: 75.31414315507715}
{хромосома: [5, 6, 0, 1, 3, 7, 4, 8, 2], F: 82.47024102653462}

Особи полученные с помощью OX:
{хромосома: [6, 5, 2, 0, 4, 1, 8, 7, 3], F: 56.876860779752874}
{хромосома: [2, 7, 3, 8, 4, 0, 1, 6, 5], F: 75.31414315507716}
Особи полученные с помощью OX(разрез выходит за пределы
хромосомы):
{хромосома: [8, 4, 0, 1, 2, 7, 3, 5, 6], F: 62.81384716485418}
{хромосома: [5, 6, 0, 1, 3, 7, 2, 8, 4], F: 80.91721919768806}

Особи полученные с помощью CX:
{хромосома: [0, 1, 3, 8, 6, 7, 2, 5, 4], F: 42.62486739000506}
{хромосома: [0, 4, 1, 7, 2, 5, 3, 8, 6], F: 65.2521437323693}
```

4.2.4 Мутации *(файл mutations.py)*

В данном тесте проводится мутация особи различными операторами мутации.

```
     Исходная особь:  {хромосома: [0, 1, 2, 3, 4, 5], F:
25.057622786556678}

     Мутация обменом:  {хромосома: [0, 4, 2, 3, 1, 5], F:
27.654153337558522}

     Мутация вставкой:  {хромосома: [0, 2, 3, 1, 4, 5], F:
27.64934056242943}

     Мутация инверсией:  {хромосома: [0, 4, 3, 2, 1, 5], F:
27.75081435738765}
```

4.2.5 Отбор в новую популяцию *(файл offspring_selections.py)*

В данном тесте проводится отбор особей в новую популяцию различными методами.

```
Популяция:
{хромосома: [0, 1, 2, 4, 3], F: 23.173319424754393}
{хромосома: [0, 4, 1, 3, 2], F: 25.025944503299343}
{хромосома: [0, 4, 3, 1, 2], F: 26.34141154176739}
{хромосома: [0, 4, 3, 2, 1], F: 22.632407630815543}
{хромосома: [0, 4, 3, 2, 1], F: 22.632407630815543}
{хромосома: [0, 3, 2, 4, 1], F: 21.857852386286353}
{хромосома: [0, 4, 2, 1, 3], F: 25.56212803671899}
{хромосома: [0, 2, 4, 1, 3], F: 25.566856297238196}
{хромосома: [0, 2, 3, 4, 1], F: 22.63713589133475}
{хромосома: [0, 1, 4, 3, 2], F: 22.63713589133475}
{хромосома: [0, 3, 2, 1, 4], F: 24.541502921469025}
{хромосома: [0, 3, 4, 1, 2], F: 25.861698220456276}
{хромосома: [0, 1, 4, 2, 3], F: 21.857852386286353}
{хромосома: [0, 1, 3, 2, 4], F: 22.337565707597463}
{хромосома: [0, 4, 1, 2, 3], F: 24.541502921469025}

Отбор усечением:  {хромосома: [0, 1, 3, 2, 4], F:
22.337565707597463}
{хромосома: [0, 4, 3, 2, 1], F: 22.632407630815543}
{хромосома: [0, 1, 4, 3, 2], F: 22.63713589133475}
{хромосома: [0, 1, 4, 2, 3], F: 21.857852386286353}
{хромосома: [0, 2, 3, 4, 1], F: 22.63713589133475}

Элитарный отбор:  {хромосома: [0, 3, 2, 4, 1], F:
21.857852386286353}
{хромосома: [0, 1, 4, 2, 3], F: 21.857852386286353}
{хромосома: [0, 1, 3, 2, 4], F: 22.337565707597463}
{хромосома: [0, 4, 3, 2, 1], F: 22.632407630815543}
{хромосома: [0, 4, 3, 2, 1], F: 22.632407630815543}
```

4.2.6 Классический ГА

Классический ГА находит решение на заданном наборе данных в среднем за несколько десятков поколений. Например, на следующей конфигурации (рис. 3).

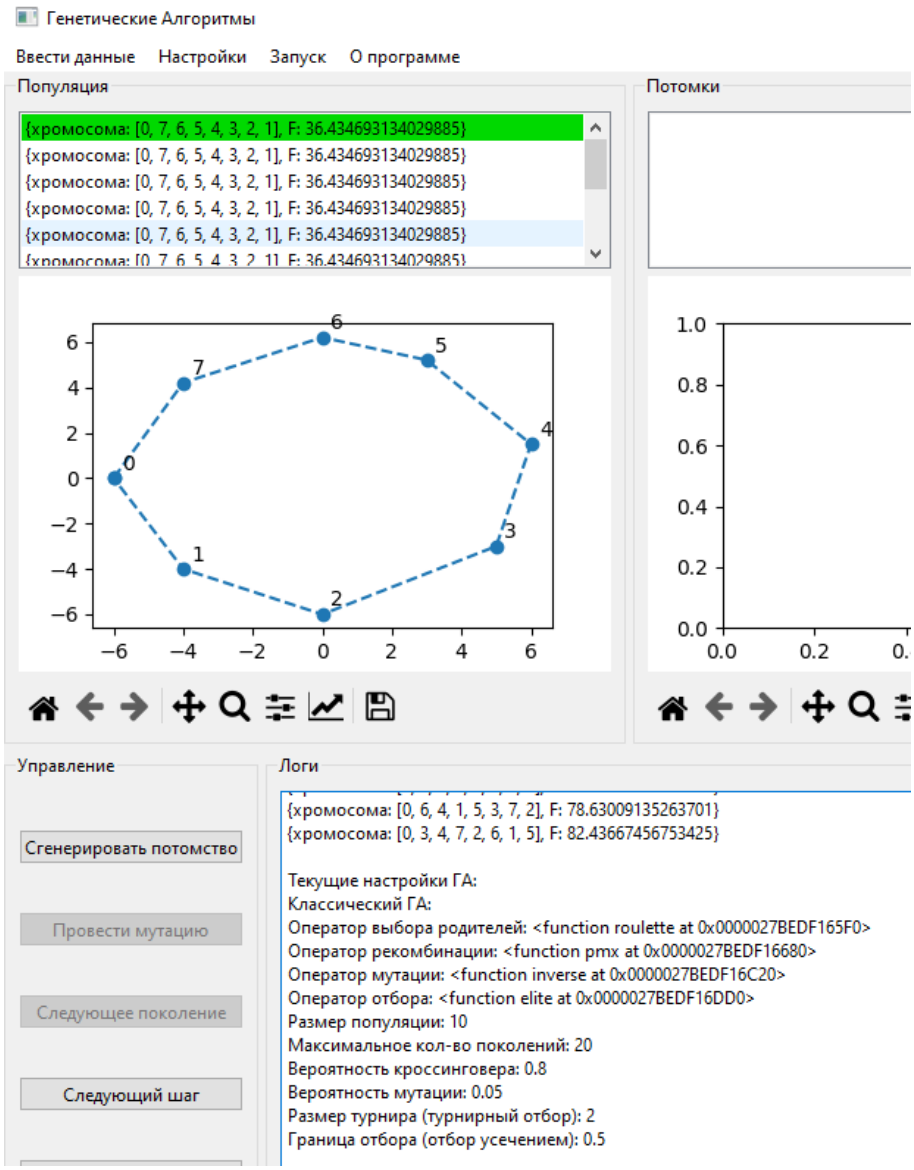

Рисунок 3: классический ГА

4.2.7 Генитор

Генитор требует значительно больше поколений для поиска оптимального решения. Компенсируется это низкой вычислительной сложностью, т.к. за один цикл создаётся лишь один потомок. Пример конфигурации представлен на рисунке 4.

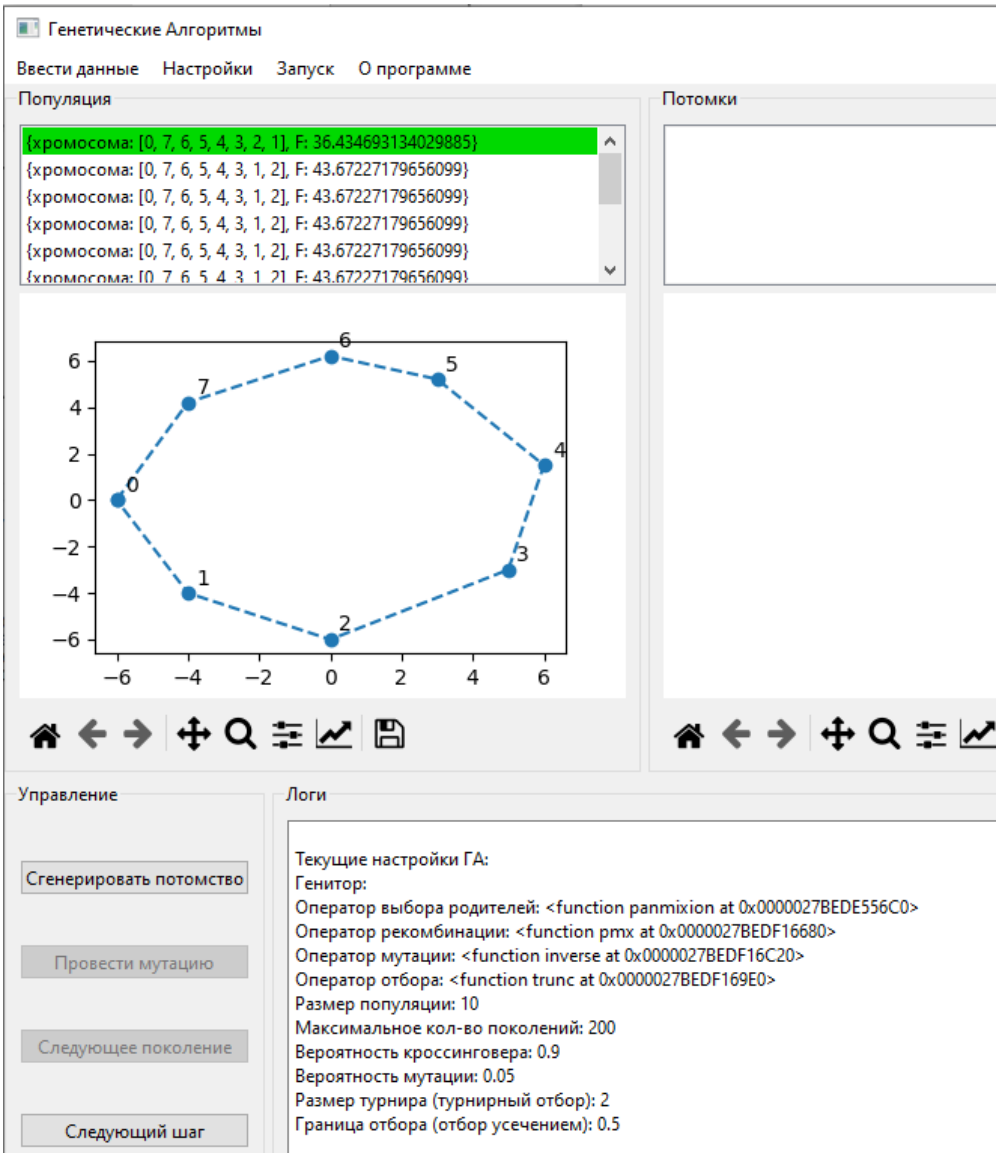

Рисунок 4: генитор

4.2.8 Метод прерывистого равновесия

Данному методу требуется относительно меньшее количество поколений для нахождения решения, однако из-за своего принципа отбора он плодит большое количество особей, что сказывается на сложности алгоритма. Пример конфигурации представлен на рисунке 5.

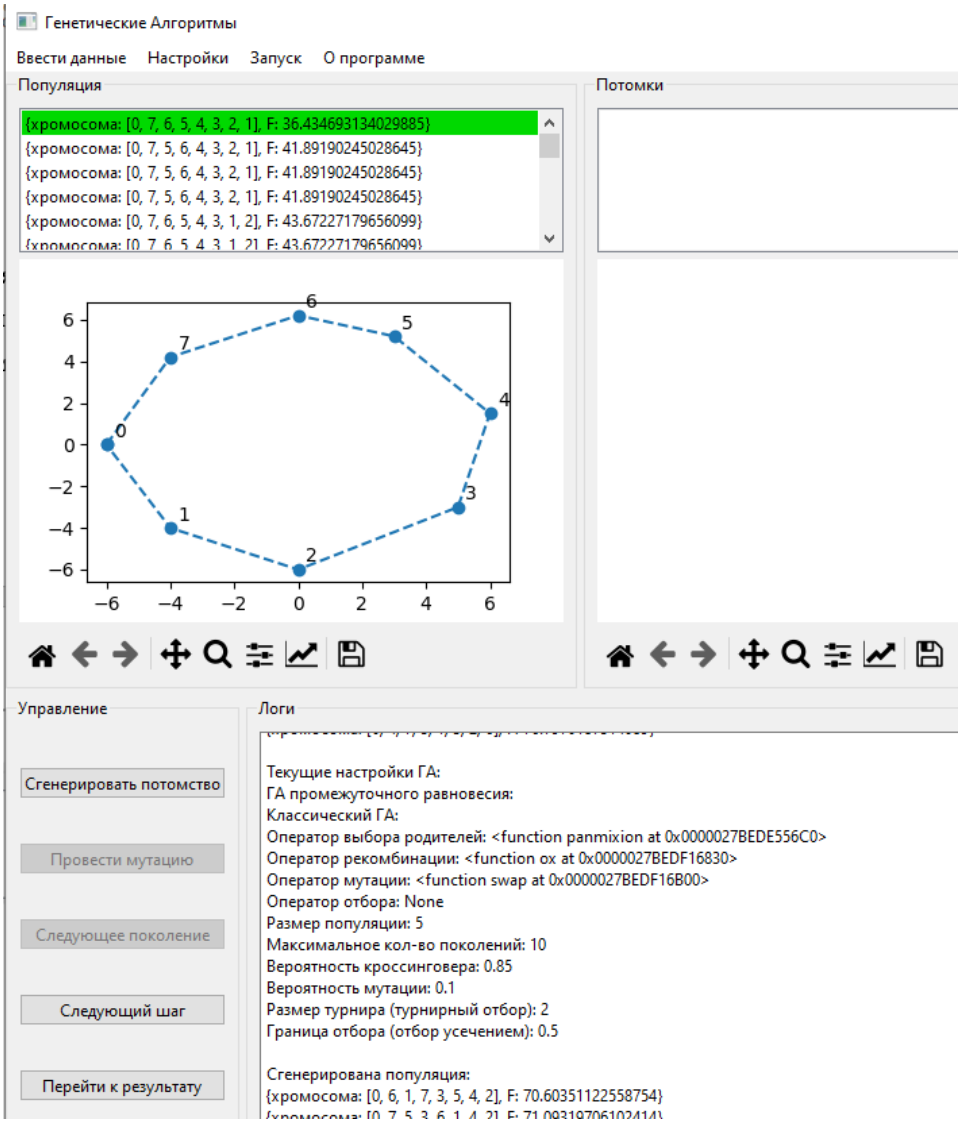

Рисунок 5: метод прерывистого равновесия

4.3 Тестирование Графического интерфейса

4.3.1 Панель управления

Предоставляет набор кнопок для управления процессом ГА. Пример на рисунке 3.

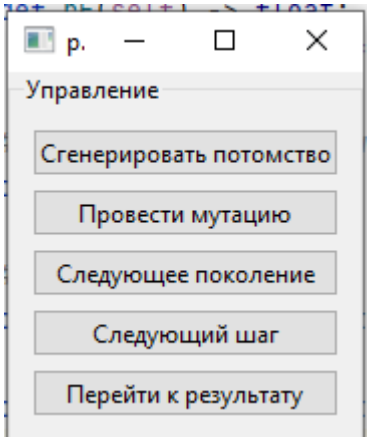

Рисунок 6: панель управления

4.3.2 Логгер

Выводит информацию о ходе алгоритма, пример представлен на рисунке 4.

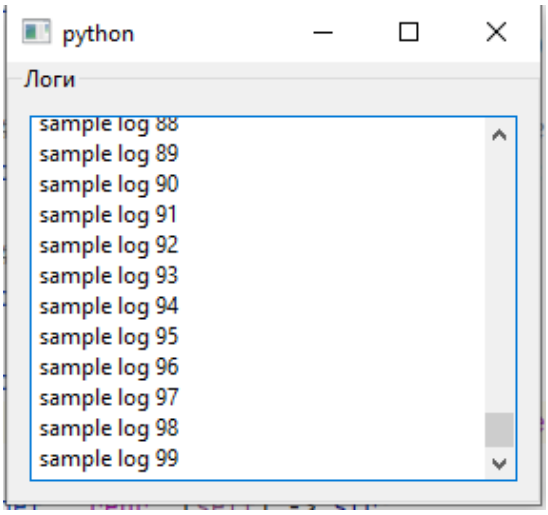

Рисунок 7: логгер

4.3.3 Виджет популяции

Предоставляет список особей с возможностью выбора особи которую нужно визуализировать (путь в графе). Пример представлен на рисунке 4.

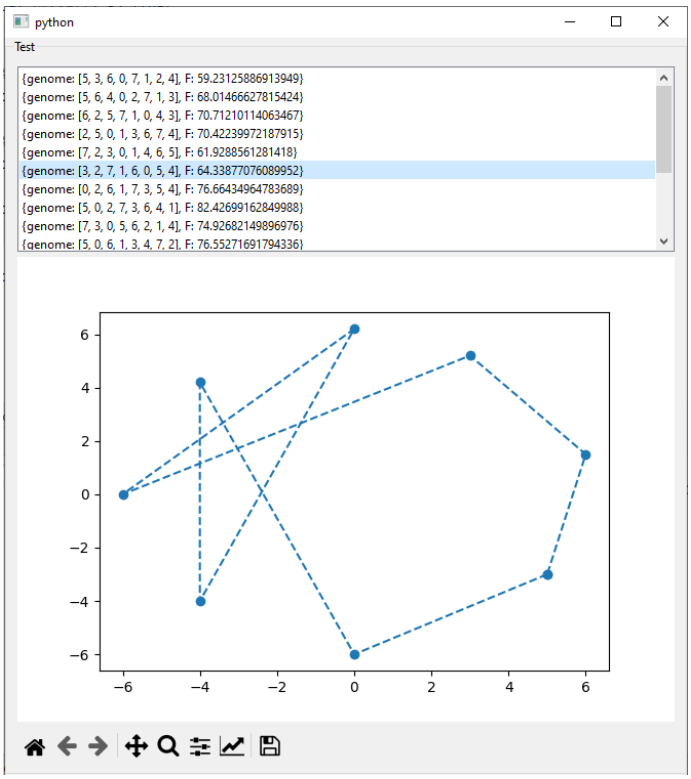

Рисунок 8: виджет популяции

4.3.4 Главное окно

Аггрегирует виджеты в соответствии со скетчем (п. 2.1). Пример представлен на рисунке 5.

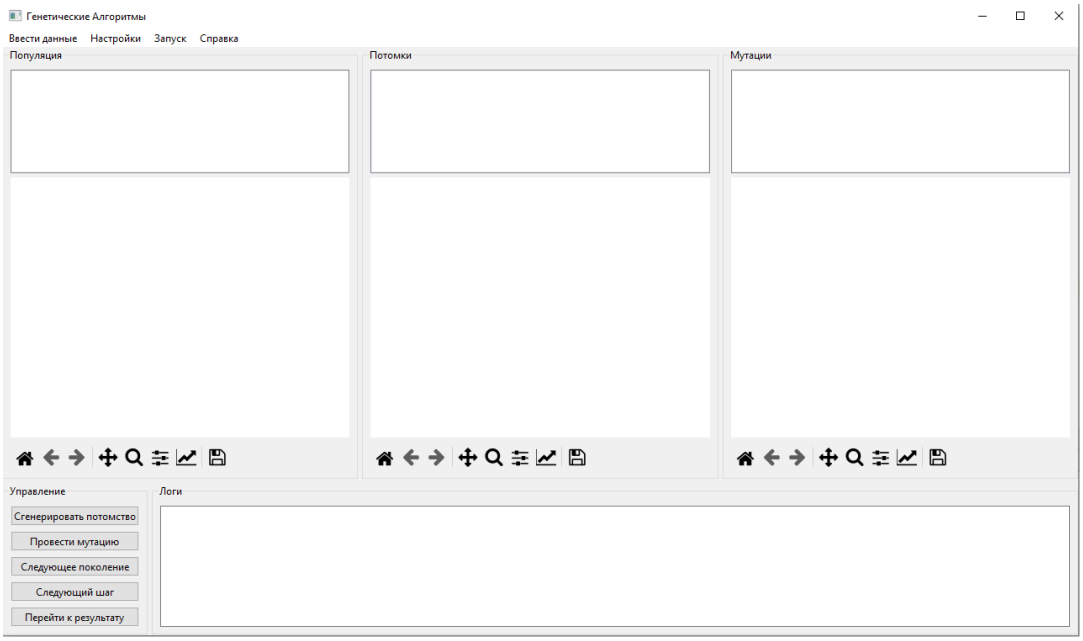

Рисунок 9: главное окно

4.4 Готовая программа

Готовая программа показывает сходимость на заданном тесте *(файл towns.txt)* при различных (в адекватных пределах) параметрах настройки алгоритма. Также программа выдаёт оповещения при нарушениях процедуры работы с программой (например отсутствие данных или настроек). На рисунке 7 изображён процесс работы, на рисунке 8 результат за заданное количество поколений.

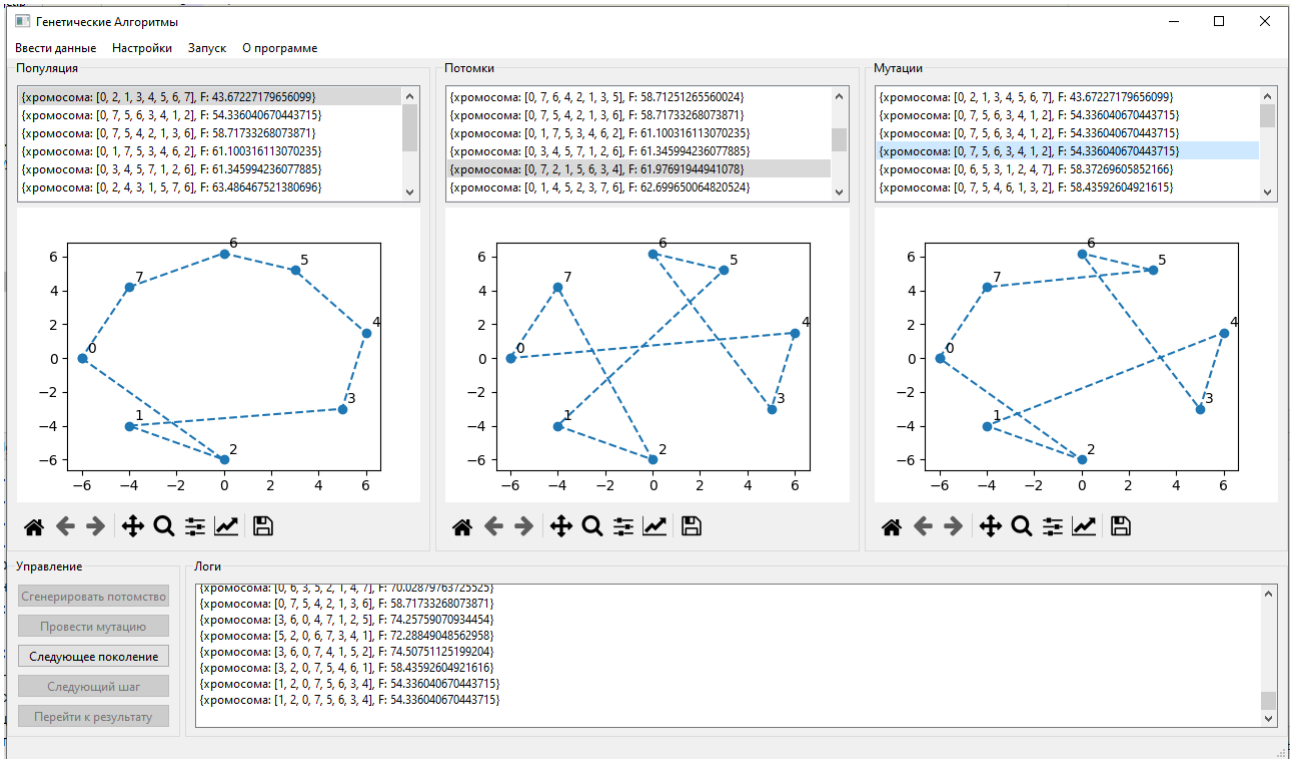

Рисунок 10: Готовая программа

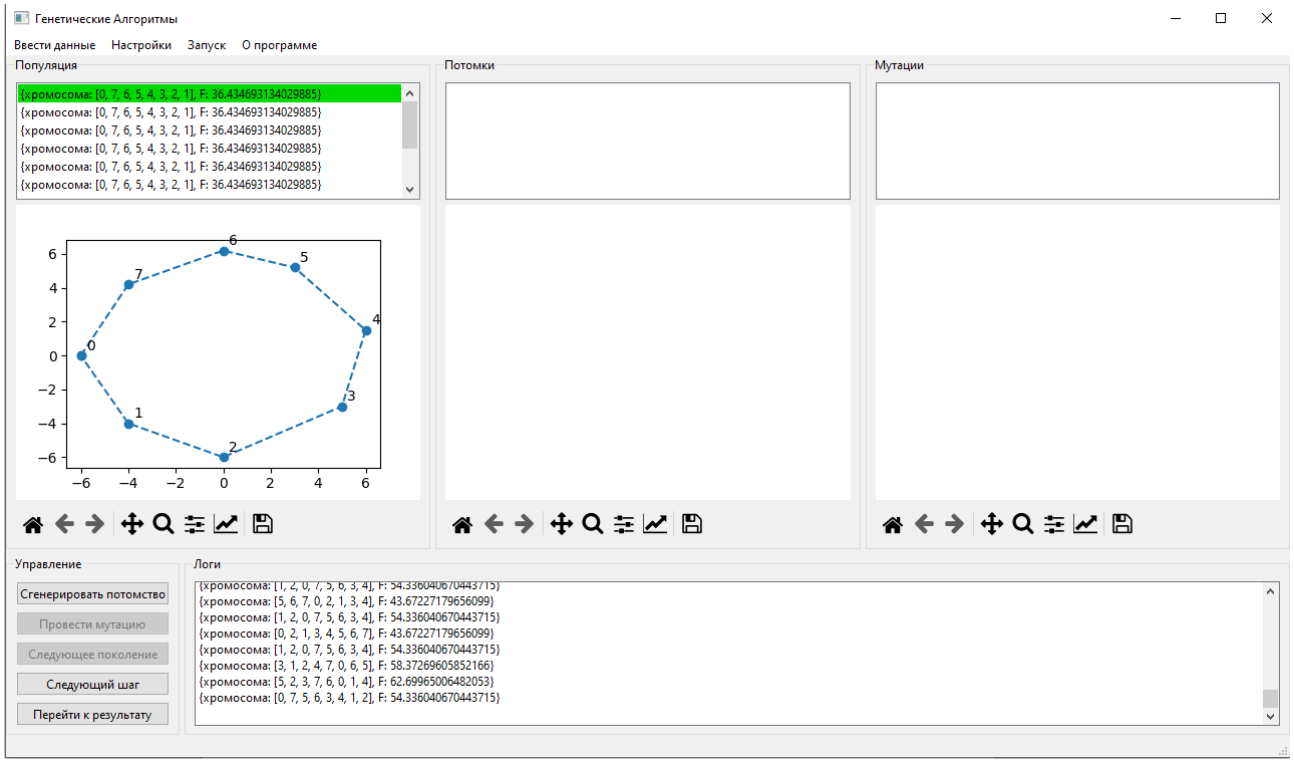

Рисунок 11: Результат за 10 поколений

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Панченко Т.В. «Генетические Алгоритмы»: Издательский дом «Астраханский университет» 2007.

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ