

**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 был описан генетический алгоритм решающий поставленную задачу и написана программа, реализующая решение данной задачи с помощью генетического алгоритма.

## СОДЕРЖАНИЕ

	Введение	6
1.	Требования к программе	7
1.1.	Исходные требования к программе	7
2.	План разработки	8
2.1	Параметры генетического алгоритма	8
2.2	План взаимодействия с программой и GUI	13
3.	Реализация	15
3.1	Утилитарный модуль	15
3.2	Модель	16
3.2.1	Классы ядра	16
3.2.2	Операторы ГА	19
3.2.3	Исполнители ГА	20
3.3	Графический интерфейс	24
3.3.1	Виджеты	24
3.3.2	Диалоговые окна	26
3.3.3	Главное окно	28
4.	Тестирование	30
4.1	Тестирование вспомогательного модуля	30
4.1.1	Логгер	30
4.1.2	Ввод из файла	30
4.2	Тестирование модели	30
4.2.1	Генерация популяции	30
4.2.2	Выбор родителей	31
4.2.3	Рекомбинация	32
4.2.4	Мутации	32
4.2.5	Отбор в новую популяцию	33
4.2.6	Классический ГА	34

4.2.7	Генитор	34
4.2.8	Метод прерывистого равновесия	35
4.3.1	Панель управления	36
4.3.2	Логгер	37
4.3.3	Виджет популяции	37
4.3.4	Диалог ввода	38
4.3.5	Диалог настроек	38
4.3.6	Главное окно	30
4.4	Готовая программа	40
	Заключение	41
	Список использованных источников	42
	Приложение А. Исходный код	42

## **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)

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

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

Создадим детей и вставим аллели, которыми обмениваются родители.

$$O_1 = (\_ \_ \_ \mid 6 \ 5 \ 2 \mid \_ \_)$$

$$O_2 = (\_ \_ \_ \mid 2 \ 7 \ 3 \mid \_ \_)$$

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

$$O_1 = (4 \ 1 \ 8 \mid 6 \ 5 \ 2 \mid \_ \_)$$

$$O_2 = (1 \ \_ \_ \mid 2 \ 7 \ 3 \mid 8 \ 4)$$

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 (бый 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.

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

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

Существует ещё несколько операторов рекомбинации перестановок: **ОХ** (order crossover) и **СХ** (cycle crossover).

- Рассмотрим принцип работы **ОХ** 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.

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

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

Создадим детей.

$$O_1 = (\_ \_ \_ \mid 6 \ 5 \ 2 \mid \_ \_)$$

$$O_2 = (\_ \_ \_ \mid 2 \ 7 \ 3 \mid \_ \_)$$

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

$$P_1 : 5 \ 6 \ 4 \ 1 \ 8 \mid 2 \ 7 \ 3$$

$$P_2 : 8 \ 4 \ 1 \ 3 \ 7 \mid 6 \ 5 \ 2$$

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.

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

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

- Рассмотрим работу оператора **СХ**.

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.

$$O_1 = (1 \ \_ \ \_ \ 6 \ \_ \ 2 \ \_ \ 4) \text{ – продолжим построение цикла.}$$

$$O_1 = (1 \ 3 \ \_ \ 6 \ \_ \ 2 \ \_ \ 4) \text{ – мы получили цикл } 1 \ 4 \ 6 \ 2 \ 3$$

Дополним генами первого родителя

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

В методе **СХ** 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:  $(1\ 2\ 3\ 4\ 5) \rightarrow (1\ 4\ 2\ 3\ 5)$

Либо если сдвигать вправо:  $(1\ 2\ 3\ 4\ 5) \rightarrow (2\ 3\ 1\ 4\ 5)$

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

Например:  $(1\ 2\ 3\ 4\ 5) \rightarrow (1\ 5\ 3\ 4\ 2)$

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

Например:  $(1\ 2\ 3\ 4\ 5) \rightarrow (1\ 4\ 3\ 2\ 5)$

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

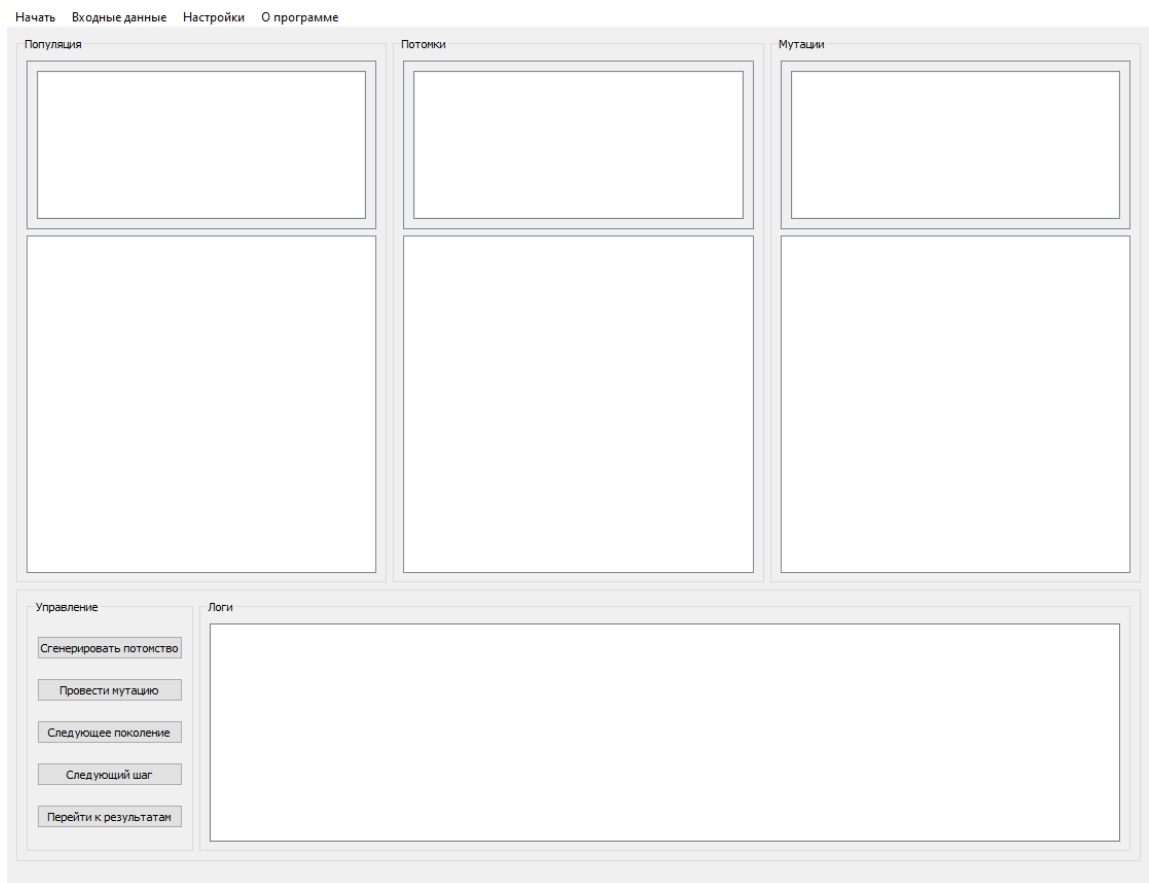


Рисунок 1: скетч GUI

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

В разделе «настройки» пользователь может выбрать модификацию 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 функции.

#### Функция ввода данных из файла

```
def file_input(path: str) -> Region
```

#### Уровни логов

Класс уровня логов, содержит уровни *Debug*, *Info* и *Warn*.

```
class LogLevel(enum.Enum)
```

#### 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, lvl: LogLevel = LogLevel.Debug) -> 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 panmixon(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)  
psize: `int`

- Максимальное поколение (до которого идёт метод результата)  
maxGen: `int`
- Вероятность рекомбинации  
rprob: `float`
- Минимальная доля аллели рекомбинации  
minR: `float`
- Максимальная доля аллели рекомбинации  
maxR: `float`
- Вероятность мутации  
mprob: `float`
- Размер турнира для турнирного отбора  
tsize: `int`
- Пороговое значение для отбора усечением  
threshold: `float`

#### Свойства (передаваемые операторам)

- Начало аллели кроссинговера  
cstart: `int`
- Размера аллели кроссинговера  
csize: `int`
- Первый ген мутации  
ngen1: `int`
- Второй ген мутации  
ngen2: `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,  
    mutator: 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. Реализован с небольшой модификацией: так как классический метод прерывистого равновесия быстро плодит огромные популяции, что приводит к невозможности исполнять алгоритм, модифицированный механизм отбора при росте популяции более чем в  $k$  раз обрезает популяцию до начального размера. Это позволило эффективно использовать данный метод на больших задачах.

```
class InterBalance(ClassicGA)
```

### 3.3 Графический Интерфейс

Элементы графического интерфейса разделены на отдельные компоненты (виджеты и диалоги), что позволяет переиспользовать и агрегировать 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. При установке популяции подсвечивает лучшую особь.

```
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). Также содержит панель с кнопками выбора уровня логов которые будут отображаться.

```
class LoggerWidget(QGroupBox)
```

## Методы

- Конструктор  
`def __init__(self)`
- Печать переданной строки  
`def print(self, s: str, lvl: LogLevel)`

**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, также осуществляет валидацию введенных данных.

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

### Свойства

- Спин-бокс с количеством городов

count: QSpinBox

- Кнопка подтверждения

btn: QPushButton

- Таблица ввода координат

table: QTableWidgetItem

### 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
  - panmixon: QRadioButton
  - tournament: QRadioButton
  - roulette: QRadioButton
- Выбор оператора рекомбинации
  - pmx: QRadioButton
  - ох: QRadioButton
  - сх: 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

- Пределы размеров аллели рекомбинации  
minR: QDoubleSpinBox  
maxR: 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) -> None`

### 3.3.3 Главное окно

Объединяет в себе виджеты в соответствии со скетчем. Виджеты объединены через QSplitter, что позволяет менять размеры каждого элемента. При нажатии получения результата алгоритм будет запущен на заданное количество поколений, при этом каждая  $k$ -ая (настраиваемый параметр) будет отображаться графически.

```
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 логов производится из нескольких потоков.

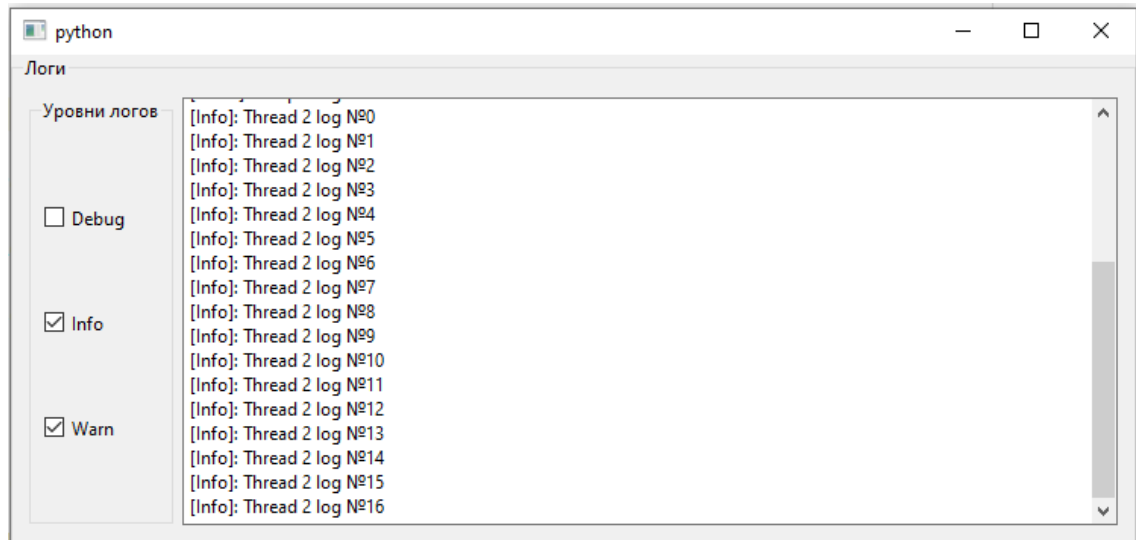


Рисунок 2: Логгер

#### 4.1.2 Ввод из файла. (файл *file\_input.py*)

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 ОХ:

```
{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 ОХ (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 СХ:

```
{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 (файл *classic.py*)

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

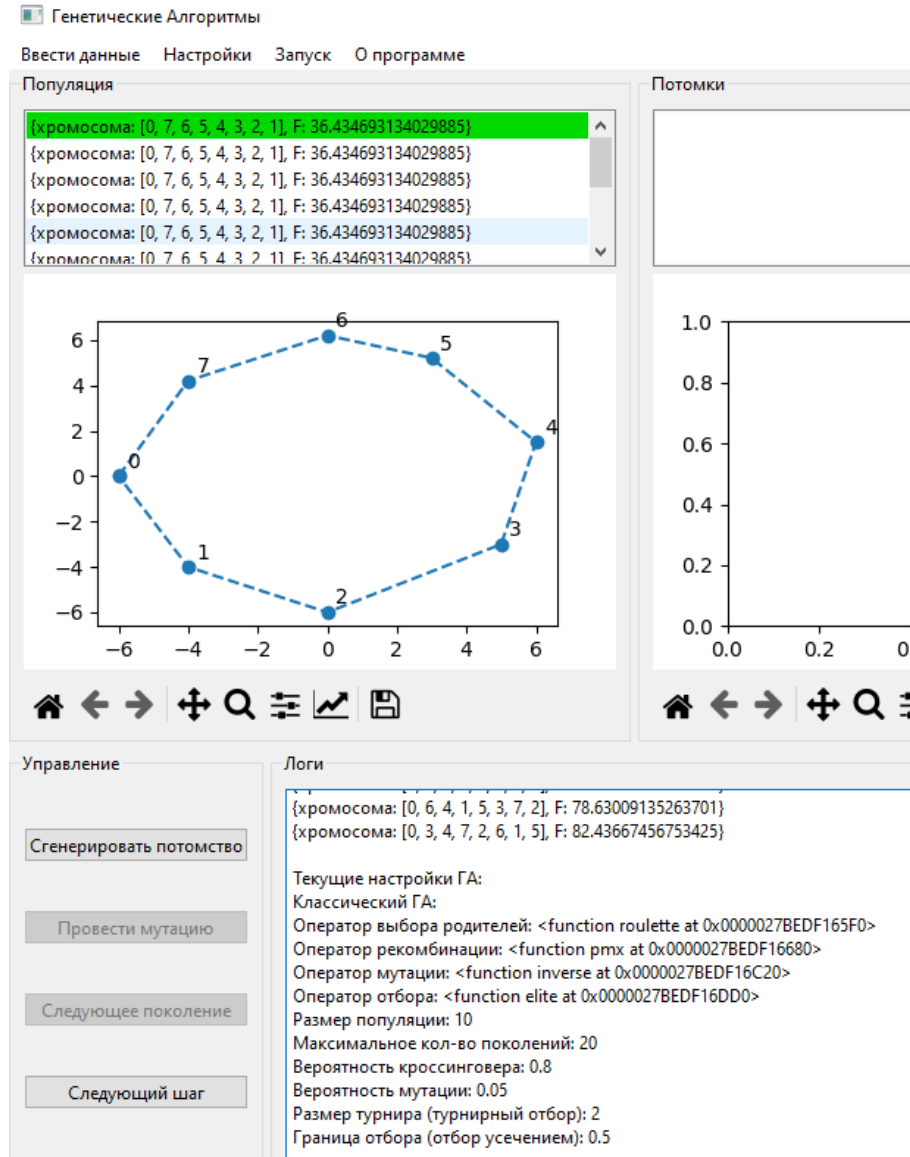


Рисунок 3: классический ГА

#### 4.2.7 Генитор (файл *genitor.py*)

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.

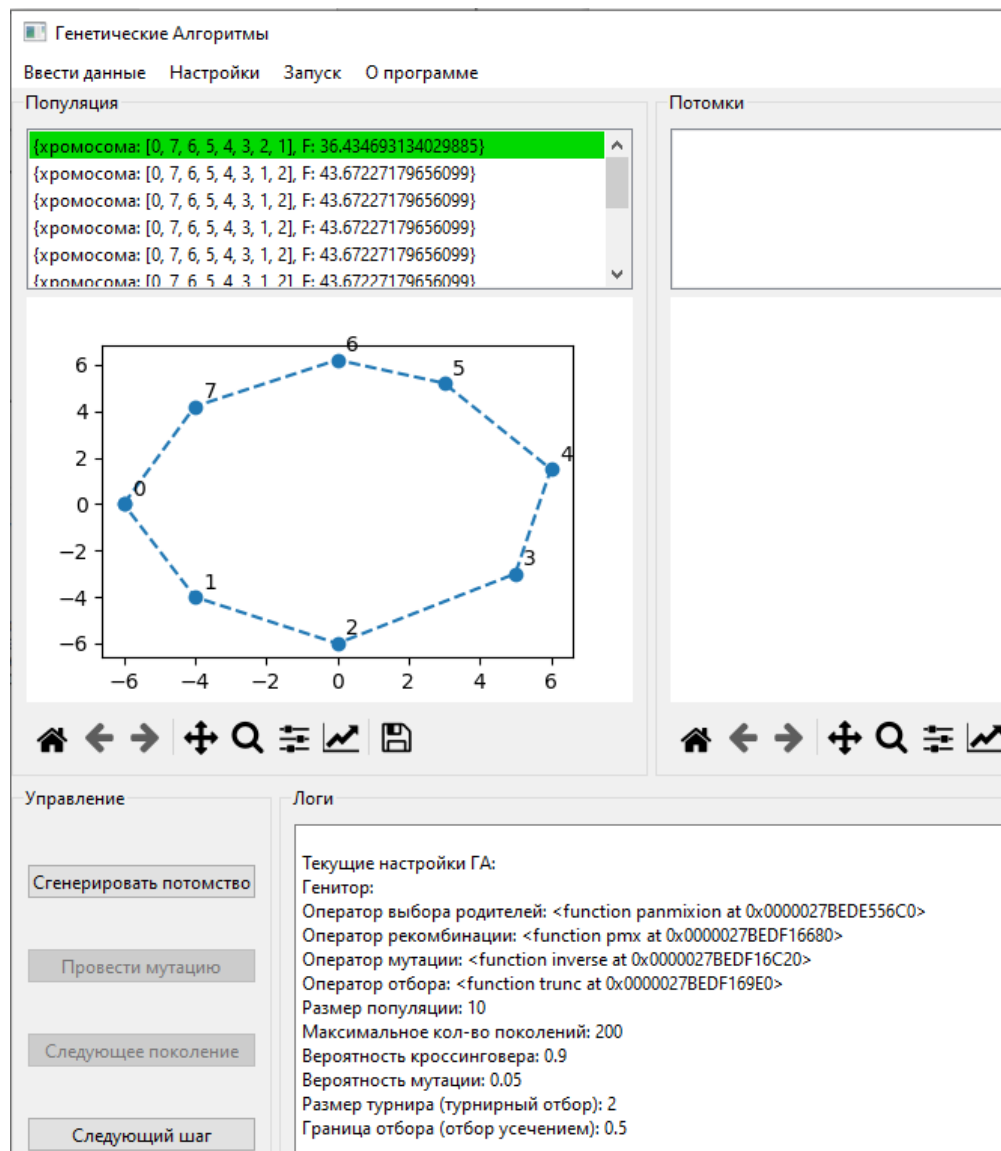


Рисунок 4: генитор

#### 4.2.8 Метод прерывистого равновесия (файл *interbalance.py*)

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.

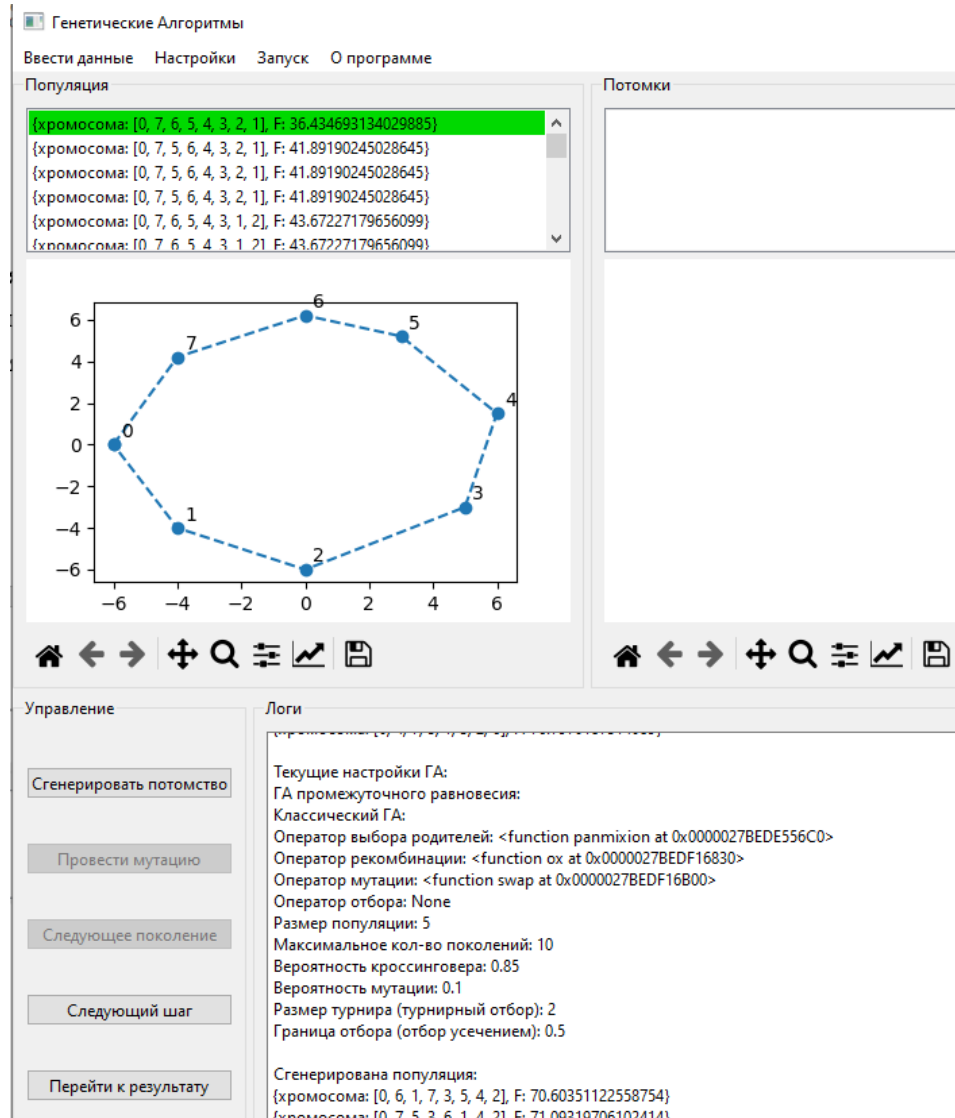


Рисунок 5: метод прерывистого равновесия

## 4.3 Тестирование Графического интерфейса

### 4.3.1 Панель управления (файл *control\_widget.py*)

Предоставляет набор кнопок для управления процессом ГА. Пример на рисунке 6.

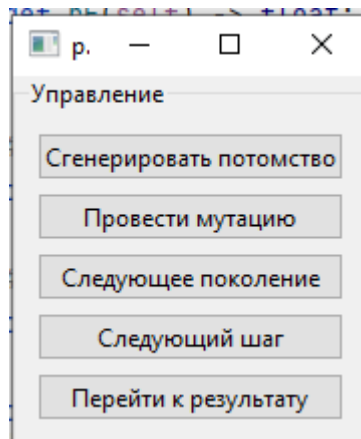


Рисунок 6: панель управления

### 4.3.2 Логгер (файл *logger\_widget.py*)

В разделе 4.1.1

### 4.3.3 Виджет популяции (файл *population\_widget.py*)

Предоставляет список особей с возможностью выбора особи которую нужно визуализировать (путь в графе). Пример представлен на рисунке 8.

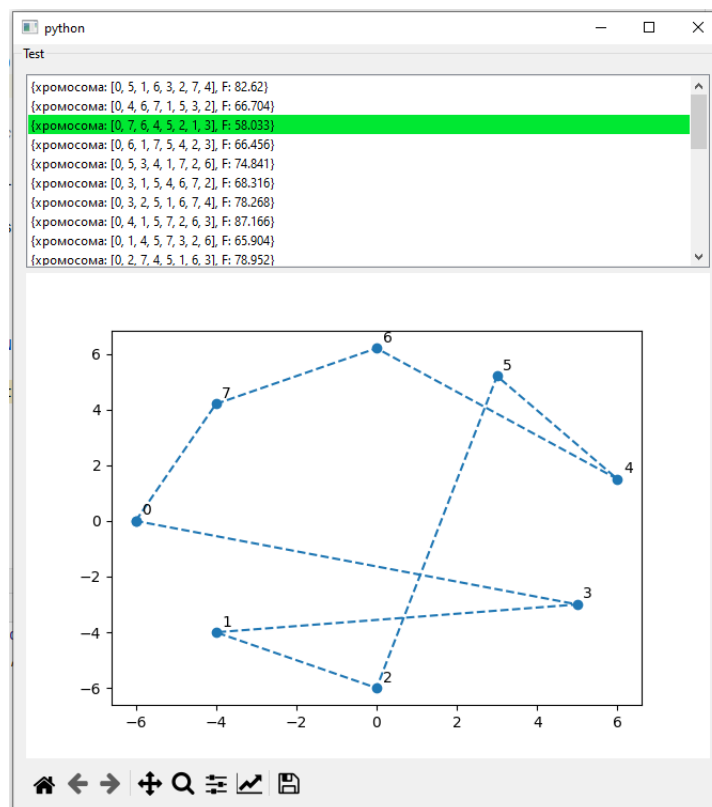


Рисунок 7: виджет популяции

#### 4.3.4 Диалог ввода (файл *input\_dialog.py*)

Ввод данных

Количество городов:  
3

Координаты

|   | x | y |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 3 | 4 |
| 3 | 6 |   |

Подтвердить

Рисунок 8: диалог ввода

#### 4.3.5 Диалог настроек (файл *settings\_dialog.py*)

Настройка ГА

Модификация ГА  
☒ Классический ГА ☐ Генитор ☐ Прер. Равновесия

Оператор отбора родителей  
☐ Панмиксия ☐ Турнир ☒ Рулетка

Оператор рекомбинации  
☐ PMX ☒ OX ☐ CX

Оператор мутации  
☐ Обмен ☒ Вставка ☐ Инверсия

Оператор отбора в новую популяцию  
☐ Отбор усечением ☒ Элитарный отбор

Параметры

Размер популяции  
55

Максимальное поколение  
55

Вероятность кроссинговера (%)  
55,00

Минимальный размер аллели рекомбинации (%)  
0,00

Максимальный размер аллели рекомбинации (%)  
100,00

Вероятность мутации (%)  
55,00

Размер турнира  
55

Порог отбора (%)  
55,00

Промежуток отображения популяции при вычислении результата  
5

Подтвердить

Рисунок 9: диалог настроек

### 4.3.6 Главное окно (файл *main\_window.py*)

Аггрегирует виджеты в соответствии со скетчем (п. 2.1). Пример представлен на рисунке 11.

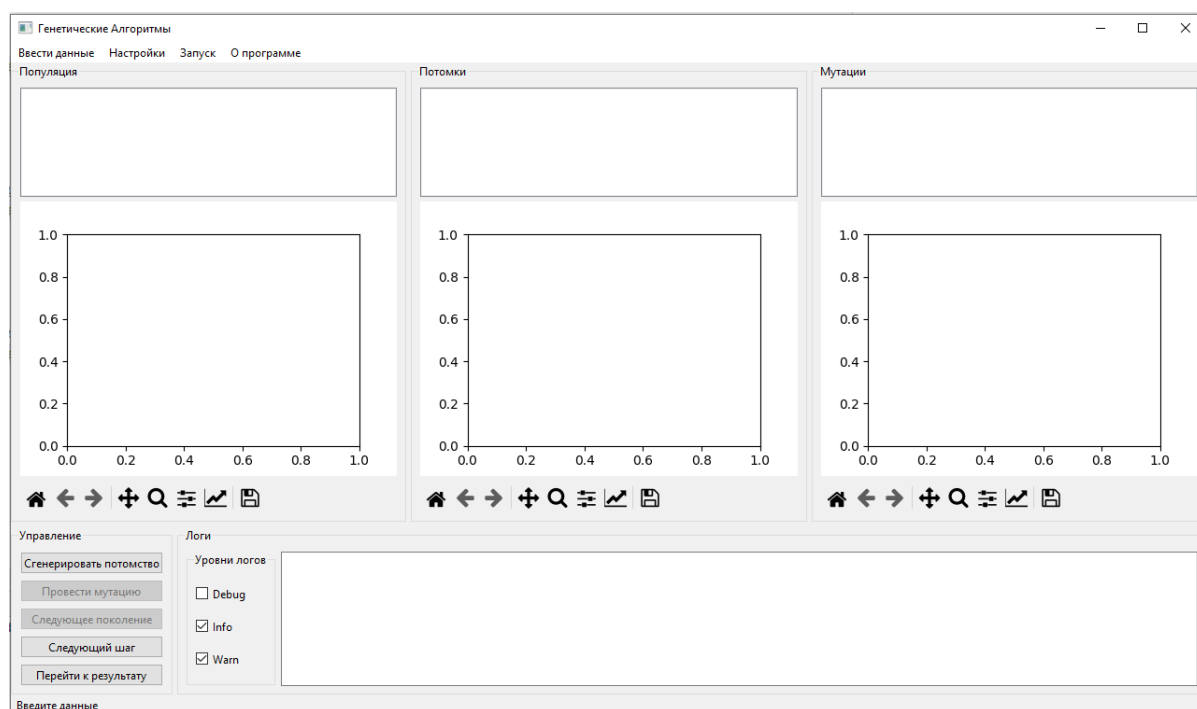


Рисунок 10: главное окно

## 4.4 Готовая программа

Готовая программа показывает сходимость на различных тестах (*naipka regions*) 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 11 изображён процесс работы, на рисунке 12 результат за заданное количество поколений.

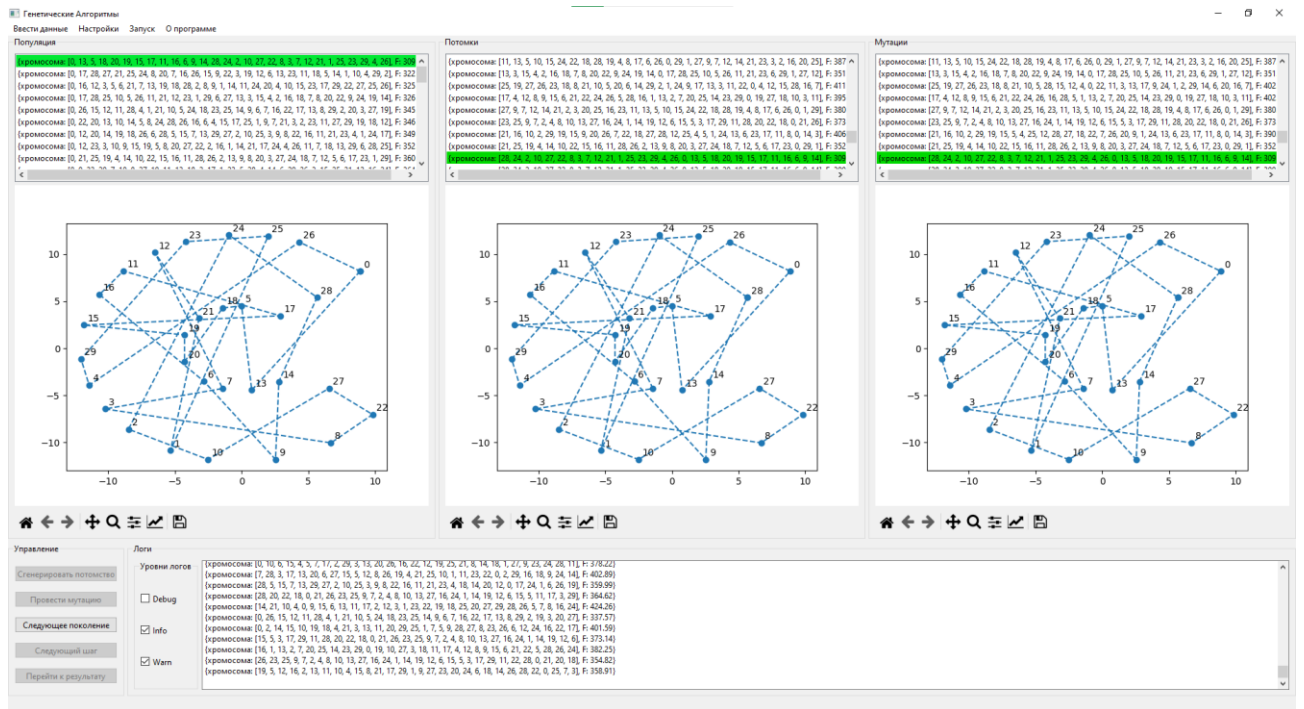


Рисунок 11: Готовая программа

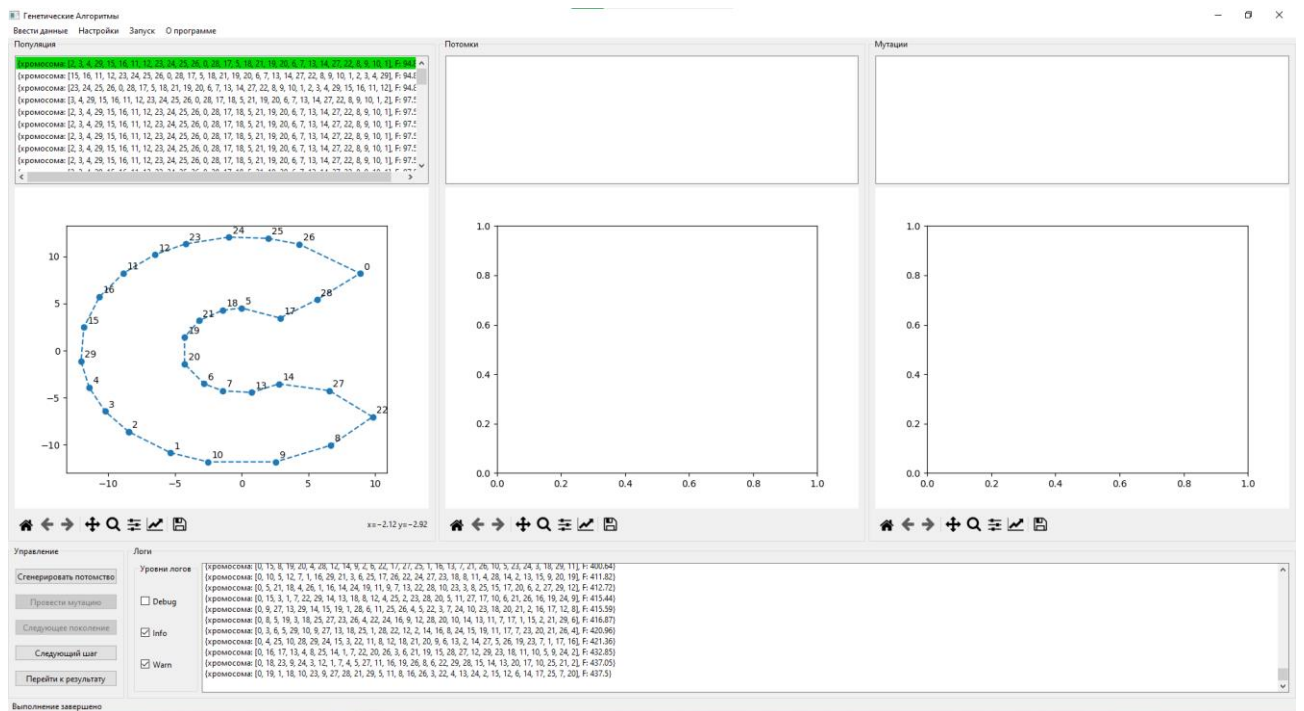


Рисунок 12: Результат за 150 поколений



## **ЗАКЛЮЧЕНИЕ**

В ходе учебной практики был исследован класс оптимизационных методов решения задач – Генетические алгоритмы. Для поставленной задачи (коммивояжёра) была создана модель генетического алгоритма, определены операторы, параметры и модификации. Данная модель была реализована программно с помощью языка python 3.10 и библиотеки PyQt6. Готовая программа позволяет вводить различные данные для задачи и гибко настраивать параметры генетического алгоритма. Сам алгоритм при правильной настройке показывает сходимость и находит оптимальное решение за конечное время на созданных тестах.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Панченко Т.В. «Генетические Алгоритмы»: Издательский дом «Астраханский университет» 2007.
2. Abid Hussain, «Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator», Computational Intelligence and Neuroscience, vol. 2017, Article ID 7430125, 7 pages, 2017. <https://doi.org/10.1155/2017/7430125>
3. Н. М. Ершов, Н. Н. Попова, «Естественные модели параллельных вычислений»: ВМК МГУ 2014.

## ПРИЛОЖЕНИЕ А

### КОД ПРОГРАММЫ

#### Файл *main.py*

```
import sys
from PyQt6.QtWidgets import *
from src.gui import *

if __name__ == "__main__":
    qapp = QApplication.instance()
    if not qapp:
        qapp = QApplication(sys.argv)

    w = MainWindow()
    w.show()

    qapp.exec()
```

#### Файл *file\_input.py*

```
from src.model.core.Town import Town
from src.model.core.Region import Region

def file_input(path: str) -> Region:
    file = open(path)
    towns = []
    for i in file:
        x, y = list(map(float, i.split()))
        towns.append(Town(x, y))
    file.close()
    reg = Region(towns)
    return reg
```

#### Файл *LogeLevel.py*

```
import enum

# УРОВНИ ЛОГОВ
class LogLevel(enum.Enum):
    Debug = 0
    Info = 1
    Warn = 2
```

#### Файл *Logger.py*

```
from __future__ import annotations
from typing import Callable
from PyQt6.QtCore import pyqtSignal, QObject
import threading
from src.util.LogLevel import LogLevel

# Логгер-синглтон
class Logger(QObject):
    _instance = None
    _lock = threading.Lock()
```

```

logSignal = pyqtSignal(str, LogLevel)

def __init__(self):
    if Logger._instance is None:
        super().__init__()

    @staticmethod
    def _get() -> Logger:
        if Logger._instance is None:
            Logger._instance = Logger()
        return Logger._instance

    # Функция логгирования, выпускает сигнал со строкой лога (обработчик получит
    её в слот)
    @staticmethod
    def log(msg: str, lvl: LogLevel = LogLevel.Debug) -> None:
        Logger._lock.acquire()
        Logger._get().logSignal.emit(msg, lvl)
        Logger._lock.release()

    # Присоединение слота обработчиков
    @staticmethod
    def connect(slot: Callable[[str], None]) -> None:
        Logger._lock.acquire()
        Logger._get().logSignal.connect(slot)
        Logger._lock.release()

```

## Файл Town.py

```

from __future__ import annotations

# Представление города (вектор)
class Town:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    # Расстояние между двумя городами
    def dist(self, t: Town) -> float:
        return ((self.x - t.x) ** 2 + (self.y - t.y) ** 2) ** (1 / 2)

    def __str__(self) -> str:
        return f'{{x: {self.x}, y: {self.y}}}'

    def __repr__(self) -> str:
        return str(self)

```

## Файл Region.py

```

import numpy as np
from src.model.core.Town import Town

# Регион городов, содержит список городов и матрицу расстояний
class Region(list[Town]):
    def __init__(self, towns: list[Town]):
        super().__init__()
        N = len(towns)
        self.extend(towns)
        self.dists = np.zeros((N, N))

```

```

        for i in range(N):
            for j in range(i, N):
                self.dists[i][j] = self.dists[j][i] = towns[i].dist(towns[j])

        if self.dists.sum() == 0:
            raise ArithmeticError('Region in one dot')

    def __str__(self) -> str:
        return f'Города: {super().__repr__()} \n' \
               f'Расстояния: \n' + \
               "\n".join(['[' + '\t'.join(['{:5g}'.format(j) for j in i]) + ']'
                           for i in self.dists])

    def __repr__(self) -> str:
        return str(self)

```

## Файл *Solution.py*

```

from __future__ import annotations
from multipledispatch import dispatch
import math
import random
import numpy as np
from src.model.core.Region import Region

# Класс особи
class Solution(list[int]):
    # Стандартный конструктор (пустая особь)
    @dispatch(Region)
    def __init__(self, reg: Region):
        super().__init__()
        self.reg = reg

    # Конструктор со случайной генерацией особи
    @dispatch(Region, int)
    def __init__(self, reg: Region, length: int):
        self.__init__(reg)
        self.extend(list(range(length)))
        random.shuffle(self)

    # Приведение обычного списка к особи
    @dispatch(Region, list)
    def __init__(self, reg: Region, lst: list[int]):
        self.__init__(reg)
        self.extend(lst)

    # Целевая функция (сумма длин весов рёбер подграфа)
    def F(self) -> float:
        return sum([self.reg.dists[self[i - 1]][self[i]] for i in
                    range(len(self))])

    # Обратная к целевой функция (чем она больше тем приспособленнее особь)
    def rF(self) -> float:
        return (np.tril(self.reg.dists).sum() / self.F()) if self.F() != 0 else
        math.inf

    # Рекурсивный сдвиг решения (цикла графа)
    def shift(self, n: int) -> Solution:
        return Solution(self.reg, list(np.roll(self, n)))

    # Сдвиг цикла, чтобы он начинался с нуля

```

```

def normalized(self) -> Solution:
    return self.shift(-self.index(0))

# Копия особи
def copy(self) -> Solution:
    return Solution(self.reg, super().copy())

def __str__(self) -> str:
    return f'{{xпомоща: {super().__repr__()}, ' + 'F:
{:.5g}}}'.format(self.F())

def __repr__(self) -> str:
    return str(self)

```

## Файл *Population.py*

```

from __future__ import annotations
import numpy as np
from multipledispatch import dispatch
from src.model.core.Solution import Solution
from src.model.core.Region import Region

# Популяция особей
class Population(list[Solution]):
    # Стандартный конструктор (пустая популяция)
    @dispatch(Region)
    def __init__(self, reg: Region):
        super().__init__()
        self.reg = reg

    # Конструктор со случайной генерацией
    @dispatch(Region, int)
    def __init__(self, reg: Region, psize: int):
        self.__init__(reg)
        self.extend([Solution(reg, len(reg)) for i in range(psize)])

    # Приведение списка особей к популяции
    @dispatch(Region, list)
    def __init__(self, reg: Region, lst: list[Solution]):
        self.__init__(reg)
        self.extend(lst)

    # Получение лучшей особи (минимального по длине цикла)
    def min(self) -> Solution:
        return min(self, key=lambda x: x.F())

    # Худшая особь
    def max(self) -> Solution:
        return max(self, key=lambda x: x.F())

    # Среднее значение целевой функции
    def meanF(self) -> float:
        return np.mean([x.F() for x in self])

    # Среднее значение обратной к целевой  $\phi$ .
    def meanrF(self) -> float:
        return np.mean([x.rF() for x in self])

    # Нормализация всех особей
    def normalized(self) -> Population:
        return Population(self.reg, [i.normalized() for i in self])

```

```

# Копия популяции
def copy(self) -> Population:
    return Population(self.reg, super().copy())

# Сортировка особей по возрастанию длины
def sorted(self) -> Population:
    return Population(self.reg, sorted(self, key=lambda x: x.F()))

def __str__(self) -> str:
    return '\n'.join([f'{i}' for i in self])

def __repr__(self) -> str:
    return str(self)

```

## Файл *parents\_selectors.py*

```

import random
from src.model.algorithms.GA import GA
from src.model.core.Population import Population
from src.model.core.Solution import Solution

# Панмиксия - каждой особи сопоставляется другая случайная особь (в т.ч. сама особь)
def panmixon(pop: Population, ga: GA = None) -> list[tuple[Solution, Solution]]:
    return [(pop[i], pop[random.randint(0, len(pop) - 1)]) for i in range(len(pop))]

# Турнирный отбор: N раз выбирается лучшая особь из случайных tsize выбранных, далее применяется панмиксия
def tournament(pop: Population, ga: GA) -> list[tuple[Solution, Solution]]:
    ga.checkParam('tsize')
    tsize = ga.params.tsize
    if tsize < 0 or tsize > ga.params.psize:
        raise ValueError('Incorrect tournament size')
    return panmixon(
        Population(
            pop.reg,
            [Population(pop.reg, random.sample(pop, tsize)).min() for i in range(len(pop))]
        )
    )

# Рулетка: N раз выбирается случайная особь (выбор с весами), к полученной выборке применяется панмиксия
def roulette(pop: Population, ga: GA = None) -> list[tuple[Solution, Solution]]:
    weights = [p.rF() for p in pop]
    return panmixon(
        Population(
            pop.reg,
            Population(pop.reg, random.choices(pop, weights, k=len(pop)))
        )
    )

```

### Файл *recombinators.py*

```
from src.model import Solution
from src.model.algorithms.GA import GA
from src.util import *

# Partial Mapped Crossover
def pmx(p1: Solution, p2: Solution, ga: GA) -> tuple[Solution, Solution]:
    for a in ['cstart', 'csize']:
        ga.checkParam(a)

    start = ga.params.cstart
    size = ga.params.csize

    N = len(p1)

    if size < 0 or size > N or start < 0 or start > N - 1:
        raise ValueError('Incorrect start/size parameters')

    Logger.log(f'Рекомбинация. Параметры: начало: {start}, размер: {size}.\n'
               f'Родители: {p1}, {p2}')

    # Хромосомы сдвигаются так чтобы рекомбинируемая аллель была вначале (удобно
    для вычислений)
    p1, p2 = p1.shift(-start), p2.shift(-start)

    o1 = Solution(p1.reg, [-1] * N)
    o2 = Solution(p2.reg, [-1] * N)

    # Обмен аллелями
    o2[:size] = p1[:size]
    o1[:size] = p2[:size]

    Logger.log(f'Аллели после обмена: {o1} ; {o2}\n')

    # Вставка остальных генов
    for p, o in zip([p1, p2], [o1, o2]):
        for i in range(size, N):
            gen = p[i]
            # Пока ген есть в аллеле выбираем тот который он собой заменил
            while gen in o[0:size]:
                gen = p[o.index(gen)]
            o[i] = gen

    return (o1, o2)

# Ordered Crossover
def ox(p1: Solution, p2: Solution, ga: GA) -> tuple[Solution, Solution]:
    for a in ['cstart', 'csize']:
        ga.checkParam(a)

    start = ga.params.cstart
    size = ga.params.csize

    N = len(p1)

    if size < 0 or size > N or start < 0 or start > N - 1:
        raise ValueError('Incorrect start/size parameters')

    Logger.log(f'Рекомбинация. Параметры: начало: {start}, размер: {size}.\n'
               f'Родители: {p1}, {p2}')
```



```

# Хромосомы сдвигаются так чтобы рекомбинируемая аллель была вначале (удобно
для вычислений)
p1, p2, start = p1.shift(-start), p2.shift(-start), 0

o1 = Solution(p1.reg, [-1] * N)
o2 = Solution(p2.reg, [-1] * N)

# Обмен аллелями
o2[:size] = p1[:size]
o1[:size] = p2[:size]

Logger.log(f'Аллели после обмена: {o1} ; {o2}\n')

# Цепочки начинающиеся с гена сразу после аллели
ps1 = p1.shift(-size)
ps2 = p2.shift(-size)

# Удаление присутствующих генов
for i in range(size):
    ps1.remove(o1[i])
    ps2.remove(o2[i])

j = 0
# Добавление оставшихся генов
for i in range(size, N):
    o1[i] = ps1[j]
    o2[i] = ps2[j]
    j += 1

return (o1, o2)

# Cycle Crossover
def cx(p1: Solution, p2: Solution, ga: GA) -> tuple[Solution, Solution]:
    ga.checkParam('cstart')
    start = ga.params.cstart

    N = len(p1)

    if start < 0 or start > N - 1:
        raise ValueError('Incorrect start parameter')

    Logger.log(f'Рекомбинация. Параметры: начало: {start}.\n'
               f'Родители: {p1}, {p2}')

    o1 = Solution(p1.reg, [-1] * N)
    o2 = Solution(p2.reg, [-1] * N)

    logs = []
    for o, p in zip([o1, o2], [[p1, p2], [p2, p1]]):
        g = start
        # Поиск цикла подстановки
        while True:
            # Добавление генов из цикла
            o[g] = p[1][g]
            g = p[1].index(p[0][g])
            if g == start:
                break
        logs.append(f'{o}')
        # Добавление остальных генов из другого родителя
    for i in range(N):
        if o[i] == -1:
            o[i] = p[0][i]

```

```

Logger.log('Аллели после обмена: ' + ' ; '.join(logs) + '\n')

return (o1, o2)

```

## Файл *mutationers.py*

```

import random
from src.model import Solution
from src.model.algorithms.GA import GA
from src.util import *

# Мутация обменом (два гена меняются местами)
def swap(o: Solution, ga: GA) -> Solution:
    for a in ['mgen1', 'mgen2']:
        ga.checkParam(a)
    i, j = ga.params.mgen1, ga.params.mgen2

    N = len(o)
    if i < 0 or j < 0 or i > N - 1 or j > N - 1:
        raise ValueError('Incorrect gens indices')

    Logger.log(f'Мутация. Параметры: i = {i} ; j = {j}\n')

    co = o.copy()
    co[i], co[j] = co[j], co[i]
    return co

# Мутация вставкой (выбранный ген перемещается к другому)
def insert(o: Solution, ga: GA) -> Solution:
    for a in ['mgen1', 'mgen2']:
        ga.checkParam(a)
    i, j = sorted([ga.params.mgen1, ga.params.mgen2])

    N = len(o)
    if i < 0 or j < 0 or i > N - 1 or j > N - 1:
        raise ValueError('Incorrect gens indices')

    Logger.log(f'Мутация. Параметры: i = {i} ; j = {j}\n')

    co = o.copy()
    if i != j:
        dir = bool(random.randint(0, 1))
        if dir:
            n = co[i]
            del co[i]
            co.insert(j - 1, n)
        else:
            n = co[j]
            del co[j]
            co.insert(i + 1, n)

    return co

# Мутация инверсией (инверсия генов между двумя генами)
def inverse(o: Solution, ga: GA) -> Solution:
    for a in ['mgen1', 'mgen2']:
        ga.checkParam(a)
    i, j = sorted([ga.params.mgen1, ga.params.mgen2])

```

```

N = len(o)
if i < 0 or j < 0 or i > N - 1 or j > N - 1:
    raise ValueError('Incorrect gens indices')

Logger.log(f'Мутация. Параметры: i = {i} ; j = {j}\n')

co = o.copy()
co[i:j + 1] = reversed(co[i:j + 1])
return co

```

### Файл *offspring\_selectors.py*

```

import random
from src.model.core.Population import Population
from src.model.algorithms.GA import GA

# Отбор усечением (psize раз выбирается особь из тех кто попал под threshold)
def trunc(pop: Population, ga: GA) -> Population:
    for a in ['psize', 'threshold']:
        ga.checkParam(a)
    threshold = ga.params.threshold

    if threshold < 0 or threshold > 1:
        raise ValueError('Incorrect threshold params')

    return Population(
        pop.reg,
        random.choices(
            pop.sorted()[:int(len(pop) * threshold) + 1], k=ga.params.psize
        )
    )

# Элитарный отбор (отбор psize самых лучших)
def elite(pop: Population, ga: GA) -> Population:
    ga.checkParam('psize')
    return Population(
        pop.reg, pop.sorted()[:ga.params.psize]
    )

```

### Файл *Params.py*

```

# Параметры ГА
class Params:
    def __init__(self):
        # Настраиваемые параметры
        self.psize: int = 0
        self.maxGen: int = 0
        self.rprob: float = 0
        self.minR: float = 0
        self.maxR: float = 0
        self.mprob: float = 0
        self.tsize: int = 0
        self.threshold: float = 0
        # Генерируемые параметры
        self.cstart: int = 0
        self.csize: int = 0
        self.mgen1: int = 0
        self.mgen2: int = 0

```

```

def __str__(self):
    return f'Размер популяции: {self.psize}\n' \
           f'Максимальное кол-во поколений: {self.maxGen}\n' \
           f'Вероятность кроссинговера: {self.rprob}\n' \
           f'Границы размеров аллели: [{self.minR}; {self.maxR}]\n' \
           f'Вероятность мутации: {self.mprob}\n' \
           f'Размер турнира (турнирный отбор): {self.tsize}\n' \
           f'Граница отбора (отбор усечением): {self.threshold}'

```

## Файл GA.py

```

from __future__ import annotations
from typing import Callable
from src.model.core.Region import Region
from src.model.core.Solution import Solution
from src.model.core.Population import Population
from src.model.algorithms.Params import Params
from src.util import *

# Абстрактный ГА
class GA:
    def __init__(
        self,
        pSelector: Callable[[Population, GA], list[tuple[Solution,
Solution]]] = None,
        recombinator: Callable[[Solution, Solution, GA], tuple[Solution,
Solution]] = None,
        mutator: Callable[[Solution, GA], Solution] = None,
        oSelector: Callable[[Population, GA], Population] = None
    ):
        self.pSelector = pSelector
        self.recombinator = recombinator
        self.mutator = mutator
        self.oSelector = oSelector
        self.params = Params()
        self.gen = 0

        self.reg: Region = None
        self.N: int = 0
        self.population: Population = None
        self.parents: list[tuple[Solution, Solution]] = None
        self.children: Population = None
        self.mutChildren: Population = None
        self.tempPop: Population = None
        self.offspring: Population = None

    def start(self, reg: Region) -> None:
        self.reg = reg
        self.N = len(reg)
        self.population = Population(self.reg, self.params.psize).normalized()

    def parentsSelect(self) -> None:
        Logger.log('Выбор родителей:')

    def crossover(self) -> None:
        Logger.log('Скрещивание:')

    def mutation(self) -> None:
        Logger.log('Мутации:')

```

```

def offspringSelect(self) -> None:
    Logger.log('Отбор потомков:')

def newPopulation(self) -> None:
    self.gen += 1
    Logger.log(f'Новое поколение: {self.gen}\n')

def nextGeneration(self) -> None:
    self.parentsSelect()
    self.crossover()
    self.mutation()
    self.offspringSelect()
    self.newPopulation()

# Проверка существования параметра
def checkParam(self, attr: str) -> None:
    if not hasattr(self.params, attr):
        raise AttributeError(f'GA has not parameter {attr}')

def __str__(self) -> str:
    return f'Оператор выбора родителей: {self.pSelector}\n' \
           f'Оператор рекомбинации: {self.recombinator}\n' \
           f'Оператор мутации: {self.mutationer}\n' \
           f'Оператор отбора: {self.oSelector}\n' \
           + str(self.params)

def __repr__(self) -> str:
    return str(self)

```

## Файл *ClassicGA.py*

```

from typing import Callable
import random
from src.model.algorithms.GA import GA
from src.model.core.Population import Population
from src.model.core.Region import Region
from src.model.core.Solution import Solution

# Классический ГА
class ClassicGA(GA):
    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):
        super().__init__(pSelector, recombinator, mutationer, oSelector)

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

    def parentsSelect(self) -> None:
        super().parentsSelect()
        self.parents = self.pSelector(self.population, self)

    def crossover(self) -> None:
        super().crossover()
        self.children = Population(self.reg)
        for p in self.parents:
            if random.random() < self.params.rprob:
                # Генерация начала аллели кроссинговера

```

```

        self.params.cstart = random.randint(0, self.N - 1)
        # Генерация размера аллели (от 1 до N-1, т.к. при значения 0 и N
        изменений не будет)
        self.params.csize = random.randint(int(self.N *
self.params.minR), int(self.N * self.params.maxR))
        self.children.extend(self.recombinator(p[0], p[1], self))
    else:
        self.children.extend(p)

def mutation(self) -> None:
    super().mutation()
    self.mutChildren = Population(self.reg)
    for c in self.children:
        if random.random() < self.params.mprob:
            self.params.mgen1, self.params.mgen2 = sorted([random.randint(0,
self.N - 1) for i in range(2)])
            self.mutChildren.append(self.mutationer(c, self))
        else:
            self.mutChildren.append(c)

def offspringSelect(self) -> None:
    super().offspringSelect()
    self.tempPop = Population(self.reg, self.population + self.mutChildren)
    self.offspring = self.oSelector(self.tempPop, self)

def newPopulation(self) -> None:
    super().newPopulation()
    self.population = self.offspring
    self.parents = self.children = self.mutChildren = self.tempPop =
self.offspring = None

def __str__(self) -> str:
    return 'Классический ГА:\n' + super().__str__()

def __repr__(self) -> str:
    return str(self)

```

## Файл *InterBalance.py*

```

from typing import Callable
from src.model.algorithms.ClassicGA import ClassicGA
from src.model.algorithms.GA import GA
from src.model.core.Population import Population
from src.model.core.Solution import Solution
from src.model.operators.parents_selectors import panmixon
from src.util import *

# ГА с методом промежуточного баланаса (модификация с ограничением)
class InterBalance(ClassicGA):
    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):
    super().__init__(panmixon, recombinator, mutationer, oSelector)
    # Максимальное расширение популяции
    self.maxExpandCoef = 3 ** 3

def offspringSelect(self) -> None:
    Logger.log('Отбор потомков:')

```

```

        self.tempPop = Population(self.reg, self.population + self.mutChildren)
        # Если размер популяции вырос в maxExpandCoef раз, урезаем её до
        # начального размера выбранным оператором отбора
        if len(self.tempPop) > self.params.psize * self.maxExpandCoef:
            self.offspring = self.oSelector(self.tempPop, self)
        else:
            mean = self.tempPop.meanF()
            self.offspring = Population(
                self.reg,
                [i for i in self.tempPop if i.F() <= mean]
            )

    def __str__(self) -> str:
        return 'ГА промежуточного равновесия:\n' + super().__str__()

    def __repr__(self) -> str:
        return str(self)

```

## Файл *Genitor.py*

```

from typing import Callable
import random
from src.model.algorithms.GA import GA
from src.model.core.Population import Population
from src.model.core.Region import Region
from src.model.core.Solution import Solution

# ГА-Генитор
class Genitor(GA):
    def __init__(self, pSelector: Callable[[Population, GA],
list[tuple[Solution, Solution]]] = None,
                recombinator: Callable[[Solution, Solution, GA],
tuple[Solution, Solution]] = None,
                mutator: Callable[[Solution, GA], Solution] = None,
                oSelector: Callable[[Population, GA], Population] = None):
        super().__init__(pSelector, recombinator, mutator, oSelector)

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

    def parentsSelect(self) -> None:
        super().parentsSelect()
        self.parents = [tuple(Solution(self.reg, i) for i in
random.sample(self.population, 2))]

    def crossover(self) -> None:
        super().crossover()
        self.children = Population(self.reg)
        self.params.cstart = random.randint(0, len(self.reg) - 1)
        self.params.csize = random.randint(int(self.N * self.params.minR),
int(self.N * self.params.maxR))
        self.children.append(
            self.recombinator(
                self.parents[0][0], self.parents[0][1], self
            )[random.randint(0, 1)]
        )

    def mutation(self) -> None:
        super().mutation()
        self.mutChildren = Population(self.reg)
        c = self.children[0]

```

```

        if random.random() < self.params.mprob:
            self.params.mgen1, self.params.mgen2 = sorted([random.randint(0,
len(self.reg) - 1) for i in range(2)])
            self.mutChildren.append(self.mutationer(c, self))
        else:
            self.mutChildren.append(c)

    def offspringSelect(self) -> None:
        super().offspringSelect()
        self.offspring = Population(self.reg)
        self.offspring.extend(self.population)
        self.offspring[self.offspring.index(self.offspring.max())] =
self.mutChildren[0]

    def newPopulation(self) -> None:
        super().newPopulation()
        self.population = self.offspring
        self.parents = self.children = self.mutChildren = self.offspring = None

    def __str__(self) -> str:
        return 'Генитоп:\n' + super().__str__()

    def __repr__(self) -> str:
        return str(self)

```

## Файл *InputDialog.py*

```

from PyQt6.QtWidgets import *
from src.model.core.Town import Town
from src.model.core.Region import Region

# Диалог ввода региона
class InputDialog(QDialog):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Ввод данных")

        self.count = QSpinBox(self)
        self.btn = QPushButton(self)
        self.btn.setText("Подтвердить")
        self.btn.clicked.connect(self.ret)
        self.table = QTableWidget(self)
        self.table.setColumnCount(2)
        self.table.setHorizontalHeaderLabels(['x', 'y'])
        self.table.resizeColumnsToContents()

        clbl = QLabel('Количество городов:')
        tlbl = QLabel('Координаты')
        lt = QVBoxLayout()
        lt.addWidget(clbl)
        lt.addWidget(self.count)
        lt.addWidget(tlbl)
        lt.addWidget(self.table)
        lt.addWidget(self.btn)
        self.setLayout(lt)

        self.count.valueChanged.connect(self.table.setRowCount)
        self.count.setMinimum(2)
        self.reg: Region = None

    def ret(self) -> None:

```



```

i = 0
try:
    towns = []
    for i in range(self.table.rowCount()):
        x, y = [self.table.item(i, j) for j in [0, 1]]
        if x is None or y is None:
            raise ValueError()
        towns.append(Town(float(x.text()), float(y.text())))
    self.reg = Region(towns)
except ArithmeticError:
    QMessageBox(
        QMessageBox.Icon(QMessageBox.Icon.Critical),
        'Ошибка',
        f'Некорректный регион'
    ).exec()
    return
except (ValueError, TypeError):
    QMessageBox(
        QMessageBox.Icon(QMessageBox.Icon.Critical),
        'Ошибка',
        f'Некорректные данные в ряду: {i + 1}'
    ).exec()
    return

self.accept()

```

## Файл *SettingsDialog.py*

```

from typing import Type, Callable
from PyQt6.QtWidgets import *
from src.model import *
from src.util import Logger
from src.util import LogLevel

# Диалог настроек параметров ГА
class SettingsDialog(QDialog):
    # Обработка кнопок выбора ГА
    def _onGA(self, state: bool, gaType: Type[GA]) -> None:
        if state:
            self.gaType = gaType

        if gaType is Genitor:
            # Выключаем/включаем уникальность кнопок в группах (операторы
            # выборов родителей и потомков)
            self.psGroup.setExclusive(not state)
            self.osGroup.setExclusive(not state)
            # Включаем кнопки
            if state:
                for rb in [self.panmixon, self.tournament, self.roulette,
self.trunc, self.elite]:
                    if rb.isChecked():
                        rb.setChecked(False)
            # Включаем кнопки по умолчанию
            if not state:
                self.panmixon.toggle()
                self.trunc.toggle()
            # Вкл/Выкл группы кнопок
            self.psBox.setEnabled(not state)
            self.osBox.setEnabled(not state)

        if gaType is InterBalance:

```

```

        if state:
            self.panmixon.setChecked(True)
        self.psBox.setEnabled(not state)

# Инициализация блока кнопок выбора ГА
def _initGA(self) -> None:
    self.classic = QRadioButton('Классический ГА')
    self.genitor = QRadioButton('Генитор')
    self.interbalance = QRadioButton('Прер. Равновесия')

    self.classic.toggled.connect(lambda state: self._onGA(state, ClassicGA))
    self.genitor.toggled.connect(lambda state: self._onGA(state, Genitor))
    self.interbalance.toggled.connect(lambda state: self._onGA(state,
InterBalance))

    self.gaBox = QGroupBox('Модификация ГА')
    lt = QHBoxLayout()
    for r in [self.classic, self.genitor, self.interbalance]:
        lt.addWidget(r)
    self.gaBox.setLayout(lt)
    self.lt.addWidget(self.gaBox)

# Обработка кнопок выбора оператора выбора родителей
def _onPS(self, state: bool, pSelector: Callable[[Population, GA],
list[tuple[Solution, Solution]]]) -> None:
    if state:
        self.pSelector = pSelector
    if pSelector is tournament:
        # Включаем размер турнира если выбран турнирный метод
        self.tsize.setEnabled(state)

# Инициализация блока кнопок выбора оператора выбора родителей
def _initPS(self) -> None:
    self.panmixon = QRadioButton('Панмиксия')
    self.tournament = QRadioButton('Турнир')
    self.roulette = QRadioButton('Рулетка')

    self.panmixon.toggled.connect(lambda state: self._onPS(state,
panmixon))
    self.tournament.toggled.connect(lambda state: self._onPS(state,
tournament))
    self.roulette.toggled.connect(lambda state: self._onPS(state, roulette))

    self.psGroup = QButtonGroup(self)
    self.psBox = QGroupBox('Оператор отбора родителей')
    lt = QHBoxLayout()
    for r in [self.panmixon, self.tournament, self.roulette]:
        lt.addWidget(r)
        self.psGroup.addButton(r)
    self.psBox.setLayout(lt)
    self.lt.addWidget(self.psBox)

# Обработка кнопок выбора оператора рекомбинации
def _onRCMB(self, state: bool, recombinator: Callable[[Solution, Solution,
GA], tuple[Solution, Solution]]) -> None:
    if recombinator is cx:
        for m in [self.minR, self.maxR]:
            m.setEnabled(not state)
    if state:
        self.recombinator = recombinator

# Инициализация блока кнопок выбора оператора рекомбинации
def _initRCMB(self) -> None:

```

```

self.pmx = QRadioButton('PMX')
self.ox = QRadioButton('OX')
self.cx = QRadioButton('CX')

self.pmx.toggled.connect(lambda state: self._onRCMB(state, pmx))
self.ox.toggled.connect(lambda state: self._onRCMB(state, ox))
self.cx.toggled.connect(lambda state: self._onRCMB(state, cx))

self.rcmbBox = QGroupBox('Оператор рекомбинации')
lt = QHBoxLayout()
for r in [self.pmx, self.ox, self.cx]:
    lt.addWidget(r)
self.rcmbBox.setLayout(lt)
self.lt.addWidget(self.rcmbBox)

# Обработка кнопок выбора оператора мутации
def _onMT(self, state: bool, mutationer: Callable[[Solution, GA], Solution])
-> None:
    if state:
        self.mutationer = mutationer

# Инициализация блока кнопок выбора оператора мутации
def _initMT(self) -> None:
    self.swap = QRadioButton('Обмен')
    self.insert = QRadioButton('Вставка')
    self.inverse = QRadioButton('Инверсия')

    self.swap.toggled.connect(lambda state: self._onMT(state, swap))
    self.insert.toggled.connect(lambda state: self._onMT(state, insert))
    self.inverse.toggled.connect(lambda state: self._onMT(state, inverse))

    self.mtBox = QGroupBox('Оператор мутации')
    lt = QHBoxLayout()
    for r in [self.swap, self.insert, self.inverse]:
        lt.addWidget(r)
    self.mtBox.setLayout(lt)
    self.lt.addWidget(self.mtBox)

# Обработка кнопок выбора оператора выбора потомков
def _onOS(self, state: bool, oSelector: Callable[[Population, GA],
Population]) -> None:
    if state:
        self.oSelector = oSelector
    if oSelector is trunc:
        # Включаем порог отбора если выбран отбор усечением
        self.threshold.setEnabled(state)

# Инициализация блока кнопок выбора оператора выбора потомков
def _initOS(self) -> None:
    self.trunc = QRadioButton('Отбор усечением')
    self.elite = QRadioButton('Элитарный отбор')

    self.trunc.toggled.connect(lambda state: self._onOS(state, trunc))
    self.elite.toggled.connect(lambda state: self._onOS(state, elite))

    self.osGroup = QButtonGroup(self)
    self.osBox = QGroupBox('Оператор отбора в новую популяцию')
    lt = QHBoxLayout()
    for r in [self.trunc, self.elite]:
        lt.addWidget(r)
        self.osGroup.addButton(r)
    self.osBox.setLayout(lt)
    self.lt.addWidget(self.osBox)

```

```

def onParam(self, name: str, val: int | float) -> None:
    setattr(self.params, name, val)
    self.maxR.setMinimum(self.minR.value())
    self.minR.setMaximum(self.maxR.value())
    self.tsize.setMaximum(self.psize.value())

# Инициализация блока числовых параметров
def _initParams(self) -> None:
    labels = [
        QLabel('Размер популяции'),
        QLabel('Максимальное поколение'),
        QLabel('Вероятность кроссинговера (%)'),
        QLabel('Минимальный размер аллели рекомбинации (%)'),
        QLabel('Максимальный размер аллели рекомбинации (%)'),
        QLabel('Вероятность мутации (%)'),
        QLabel('Размер турнира'),
        QLabel('Порог отбора (%)'),
    ]
    self.psize = QSpinBox()
    self.maxGen = QSpinBox()
    self.rprob = QDoubleSpinBox()
    self.minR = QDoubleSpinBox()
    self.maxR = QDoubleSpinBox()
    self.mprob = QDoubleSpinBox()
    self.tsize = QSpinBox()
    self.threshold = QDoubleSpinBox()

    self.tsize.setEnabled(False)
    self.threshold.setEnabled(False)

    self.psize.valueChanged.connect(lambda v: self.onParam('psize', v))
    self.maxGen.valueChanged.connect(lambda v: self.onParam('maxGen', v))
    self.rprob.valueChanged.connect(lambda v: self.onParam('rprob', v /
100))
    self.minR.valueChanged.connect(lambda v: self.onParam('minR', v / 100))
    self.maxR.valueChanged.connect(lambda v: self.onParam('maxR', v / 100))
    self.mprob.valueChanged.connect(lambda v: self.onParam('mprob', v /
100))
    self.tsize.valueChanged.connect(lambda v: self.onParam('tsize', v))
    self.threshold.valueChanged.connect(lambda v: self.onParam('threshold',
v / 100))

    for sb in [self.rprob, self.maxR, self.minR, self.mprob,
self.threshold]:
        sb.setMinimum(0)
        sb.setMaximum(100)
    self.maxGen.setMaximum(10 ** 6)
    self.psize.setMaximum(10 ** 3)

    box = QGroupBox("Параметры")
    lt = QVBoxLayout()
    for r, l in zip([self.psize, self.maxGen, self.rprob, self.minR,
self.maxR,
                    self.mprob, self.tsize, self.threshold], labels):
        r.valueChanged.emit(r.value())
        lt.addWidget(l)
        lt.addWidget(r)
    box.setLayout(lt)
    self.lt.addWidget(box)

def __init__(self, parent=None):
    super().__init__(parent)

```

```

self.setWindowTitle("Настройка ГА")

self.btn = QPushButton(self)
self.btn.setText("Подтвердить")
self.btn.clicked.connect(self.ret)

self.gaType = None
self.pSelector = None
self.recombinator = None
self.mutationer = None
self.oSelector = None
self.params = Params()

self.subResult = 0
sbLbl = QLabel('Промежуток отображения популяции при вычислении
результата')
self.subResultSB = QSpinBox()
self.subResultSB.setMaximum(10 ** 6)
self.subResultSB.valueChanged.connect(lambda v: setattr(self,
'subResult', v))

self.lt = QVBoxLayout()
self._initGA()
self._initPS()
self._initRCMB()
self._initMT()
self._initOS()
self._initParams()
self.lt.addWidget(sbLbl)
self.lt.addWidget(self.subResultSB)
self.lt.addWidget(self.btn)
self.setLayout(self.lt)

# Значения по умолчанию (radio button)
self.classic.toggle()
self.panmixion.toggle()
self.pmx.toggle()
self.swap.toggle()
self.trunc.toggle()
# Значения по умолчанию (spin box)
self.psize.setValue(10)
self.maxGen.setValue(50)
self.rprob.setValue(80)
self.mprob.setValue(5)
self.tsize.setValue(2)
self.threshold.setValue(50)
self.subResultSB.setValue(5)
self.minR.setValue(0)
self.maxR.setValue(100)

self.cfgFile = 'config'
# Загрузка настроек из файла (если настройка есть - значение по умолчанию
перезаписывается, иначе оно остаётся)
self.load()

def ret(self) -> None:
    self.save()
    self.accept()

def load(self) -> None:
    try:
        with open(self.cfgFile, 'r') as f:
            for line in f:

```

```

        data = line.split()
        if len(data) == 1:
            name = data[0]
            getattr(self, name).toggle()
        elif len(data) == 2:
            name = data[0]
            val = float(data[1])
            if type(getattr(self, name)) is QSpinBox:
                val = int(val)
            getattr(self, name).set_value(val)
    except Exception as e:
        Logger.log(f'Ошибка чтения настроек: {e}\n', LogLevel.Warn)

def save(self) -> None:
    try:
        with open(self.cfgFile, 'w') as f:
            for box in [
                ['classic', 'genitor', 'interbalance'],
                ['panmixture', 'tournament', 'roulette'],
                ['pmx', 'ox', 'cx'],
                ['swap', 'insert', 'inverse'],
                ['trunc', 'elite'],
            ]:
                for op in box:
                    if getattr(self, op).isChecked():
                        f.write(f'{op}\n')

                for sb in ['psize', 'maxGen', 'rprob', 'maxR', 'minR', 'mprob',
                           'tsize', 'threshold', 'subResultSB']:
                    f.write(f'{sb} {getattr(self, sb).value()}\n')

    except Exception as e:
        Logger.log(f'Ошибка записи настроек: {e}\n', LogLevel.Warn)

```

## Файл *ControlWidget.py*

```

from PyQt6.QtWidgets import *

# Виджет кнопок управления
class ControlWidget(QGroupBox):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setTitle("Управление")
        lt = QVBoxLayout(self)

        self.offspring = QPushButton(self)
        self.offspring.setText("Сгенерировать потомство")

        self.mutate = QPushButton(self)
        self.mutate.setText("Провести мутацию")

        self.next = QPushButton(self)
        self.next.setText("Следующее поколение")

        self.forceNext = QPushButton(self)
        self.forceNext.setText("Следующий шаг")

        self.results = QPushButton(self)
        self.results.setText("Перейти к результату")

        for w in [self.offspring, self.mutate, self.next, self.forceNext,

```

```

self.results]:
    lt.addWidget(w)

    self.setLayout(lt)

```

## Файл *LoggerWidget.py*

```

from PyQt6.QtWidgets import *
from src.util.Logger import *

# Виджет вывода логов
class LoggerWidget(QGroupBox):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setTitle("Логи")

        self.levels = [False] * 3
        box = QGroupBox("Уровни логов")
        boxlt = QVBoxLayout(box)
        self.debug = QCheckBox(LogLevel.Debug.name)
        self.info = QCheckBox(LogLevel.Info.name)
        self.warn = QCheckBox(LogLevel.Warn.name)

        self.debug.toggled.connect(lambda s: self.levelToggled(0, s))
        self.info.toggled.connect(lambda s: self.levelToggled(1, s))
        self.warn.toggled.connect(lambda s: self.levelToggled(2, s))

        for i, w in zip(list(range(3)), [self.debug, self.info, self.warn]):
            boxlt.addWidget(w)

        self.info.toggle()
        self.warn.toggle()
        box.setLayout(boxlt)

        lt = QHBoxLayout(self)

        self.text = QTextEdit()
        self.text.setReadOnly(True)
        lt.addWidget(box)
        lt.addWidget(self.text)
        self.setLayout(lt)

    def levelToggled(self, n: int, state: bool) -> None:
        self.levels[n] = state

# Вывод лога в виджет (можно использовать в качестве слота для логгера из
util.Logger)
    def print(self, s: str, lvl: LogLevel) -> None:
        if self.levels[lvl.value]:
            self.text.append(f'[{lvl.name}]: {s}')

```

## Файл *PopulationWidget.py*

```

from PyQt6.QtGui import QColor
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *

from matplotlib.backends.backend_qtagg import (
    FigureCanvasQTagg as FigureCanvas,
    NavigationToolbar2QT as NavigationToolbar

```

```

)
from matplotlib.figure import Figure
from src.model.core import Population, Town

class PopulationWidget(QGroupBox):
    def __init__(self, name: str, parent=None):
        super().__init__(parent)

        self.list = QListWidget()
        self.canvas = FigureCanvas(Figure())
        # Сам график
        self.plt = self.canvas.figure.subplots()
        # Линия пути
        self.line = None
        self.pop: Population = None

        navbar = NavigationToolbar(self.canvas, self)
        splitter = QSplitter(self)
        splitter.setOrientation(Qt.Orientation.Vertical)
        splitter.addWidget(self.list)
        splitter.addWidget(self.canvas)

        lt = QVBoxLayout(self)
        lt.addWidget(splitter)
        lt.addWidget(navbar)
        self.setLayout(lt)
        self.setTitle(name)

        # Установка слота на сигнал изменения выбранного элемента
        self.list.currentRowChanged.connect(self.drawSolution)

    # Слот отрисовки выбранного решения
    def drawSolution(self) -> None:
        x = [self.pop.reg[i].x for i in self.pop[self.list.currentRow()]]
        y = [self.pop.reg[i].y for i in self.pop[self.list.currentRow()]]

        # Если путь не отрисован, рисуется. Иначе просто меняются точки
        if self.line is None:
            self.line = self.plt.plot(list(x) + [x[0]], list(y) + [y[0]],
linestyle='--', marker='o')[0]
        else:
            self.line.set_data(list(x) + [x[0]], list(y) + [y[0]])

        self.canvas.draw()

    def clear(self) -> None:
        self.list.clear()
        self.canvas.figure.clf()
        self.plt = self.canvas.figure.subplots()
        self.canvas.draw()

    # Установка отображаемой популяции
    def setPopulation(self, pop: Population) -> None:
        self.clear()

        self.pop = pop
        for i in pop:
            self.list.addItem(str(i))

        # Подписи номеров к городам
        for i in range(len(self.pop.reg)):
            self.plt.annotate(

```



```

        i, (self.pop.reg[i].x, self.pop.reg[i].y),
        textcoords="offset points", xytext=(4, 4)
    )
    self.line = None

    # Выделение лучшей особи
    m = pop.index(pop.min())
    self.list.item(m).setBackgroundColor(QColor('lime'))
    self.list.setCurrentRow(m)

```

## Файл *MainWindow.py*

```

from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from src.gui.dialogs.InputDialog import InputDialog
from src.gui.dialogs.SettingsDialog import SettingsDialog
from src.gui.widgets.LoggerWidget import LoggerWidget
from src.gui.widgets.ControlWidget import ControlWidget
from src.gui.widgets.PopulationWidget import PopulationWidget
from src.model import *
from src.util import *

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Генетические Алгоритмы")

        mainSplitter = QSplitter(self)
        popSplitter = QSplitter(self)
        infoSplitter = QSplitter(self)

        self.parents = PopulationWidget('Популяция', self)
        self.offspring = PopulationWidget('Потомки', self)
        self.mutations = PopulationWidget('Мутации', self)
        self.logger = LoggerWidget(self)
        Logger.connect(self.logger.print)
        self.control = ControlWidget(self)

        self.control.mutate.setEnabled(False)
        self.control.next.setEnabled(False)

        for w in [self.parents, self.offspring, self.mutations]:
            popSplitter.addWidget(w)

        infoSplitter.addWidget(self.control)
        infoSplitter.addWidget(self.logger)
        infoSplitter.setStretchFactor(1, 1)

        mainSplitter.addWidget(popSplitter)
        mainSplitter.addWidget(infoSplitter)
        mainSplitter.setOrientation(Qt.Orientation.Vertical)
        self.setCentralWidget(mainSplitter)

        self.inputMenu = QMenu('Ввести данные')
        self.manually = QAction('Ввести вручную')
        self.manually.triggered.connect(self.onInput)
        self.fromFile = QAction('Выбрать файл')
        self.fromFile.triggered.connect(self.onFile)
        self.inputMenu.addAction([self.manually, self.fromFile])

```

```

self.menuBar().addMenu(self.inputMenu)
self.settings = self.menuBar().addAction('Настройки', self.onSettings)
self.start = self.menuBar().addAction('Запуск', self.onStart)
self.info = self.menuBar().addAction('О программе', self.onInfo)

self.statusBar().showMessage('Введите данные')
self.ga: GA = None
self.reg: Region = None
self.subResult = 1

self.control.offspring.clicked.connect(self.onChildren)
self.control.mutate.clicked.connect(self.onMutate)
self.control.next.clicked.connect(self.onNext)
self.control.forceNext.clicked.connect(self.onForceNext)
self.control.results.clicked.connect(self.onResults)

# Проверка на ввод данных и установку настроек
def checkSetup(self) -> bool:
    if self.reg is None:
        QMessageBox(
            QMessageBox.Icon(QMessageBox.Icon.Critical),
            'Ошибка',
            f'Не введены данные'
        ).exec()
        return False
    if self.ga is None:
        QMessageBox(
            QMessageBox.Icon(QMessageBox.Icon.Critical),
            'Ошибка',
            f'Алгоритм не настроен'
        ).exec()
        return False
    return True

# Проверка на то что популяция сгенерирована
def checkGenerated(self) -> bool:
    if self.ga.population is None:
        QMessageBox(
            QMessageBox.Icon(QMessageBox.Icon.Critical),
            'Ошибка',
            f'Популяция не сгенерирована'
        ).exec()
        return False
    return True

def onChildren(self) -> None:
    if not self.checkSetup() or not self.checkGenerated():
        return
    self.ga.parentsSelect()
    Logger.log('Выбраны родители:\n' + '\n'.join(map(str, self.ga.parents))
+ '\n', LogLevel.Info)
    self.ga.crossover()
    self.offspring.setPopulation(self.ga.children)
    for w in [self.control.offspring, self.control.forceNext,
self.control.results]:
        w.setEnabled(False)
    self.control.mutate.setEnabled(True)
    Logger.log(f'Полученные потомки:\n{self.ga.children}\n', LogLevel.Info)

def onMutate(self) -> None:
    self.ga.mutation()
    Logger.log(f'Потомки с мутациями:\n{self.ga.mutChildren}\n',
LogLevel.Info)

```

```

        self.mutations.setPopulation(self.ga.mutChildren)
        self.control.mutate.setEnabled(False)
        self.control.next.setEnabled(True)

    def onNext(self) -> None:
        self.ga.offspringSelect()
        Logger.log(f'Промежуточная популяция:\n{self.ga.tempPop}\n',
LogLevel.Info)
        Logger.log(f'Новая популяция:\n{self.ga.offspring}\n', LogLevel.Info)
        self.ga.newPopulation()
        self.parents.setPopulation(self.ga.population)
        self.offspring.clear()
        self.mutations.clear()
        for b in [self.control.offspring, self.control.forceNext,
self.control.results]:
            b.setEnabled(True)
            self.control.next.setEnabled(False)

    def onForceNext(self) -> None:
        if not self.checkSetup() or not self.checkGenerated():
            return
        self.ga.nextGeneration()
        self.parents.setPopulation(self.ga.population)

    def wait(self) -> None:
        self.setEnabled(False)
        QApplication.setOverrideCursor(QCursor(Qt.CursorShape.WaitCursor))
        self.statusBar().showMessage("Алгоритм выполняется...")

    def resume(self) -> None:
        self.setEnabled(True)
        QApplication.restoreOverrideCursor()
        self.statusBar().showMessage("Выполнение завершено")

    def onResults(self) -> None:
        if not self.checkSetup() or not self.checkGenerated():
            return

        self.wait()
        minSln = self.ga.population.min()
        while self.ga.gen < self.ga.params.maxGen:
            self.ga.nextGeneration()
            QApplication.processEvents()
            if self.ga.gen % self.subResult == 0 and minSln.F() !=
self.ga.population.min().F():
                minSln = self.ga.population.min()
                self.parents.setPopulation(self.ga.population)
                self.statusBar().showMessage(f'Алгоритм выполняется...
Поколение: {self.ga.gen}')

            self.parents.setPopulation(self.ga.population.sorted())
            self.resume()

    def onInput(self) -> None:
        d = InputDialog(self)
        res = d.exec()
        if res:
            self.reg = d.reg
            self.statusBar().showMessage('Данные введены')
            Logger.log(f'Введены данные:\n{self.reg}\n', LogLevel.Info)
            self.setDefault()

    def onFile(self) -> None:

```

```

d = QFileDialog.getOpenFileName(self, 'Выбрать файл с данными')
if d[0] != '':
    try:
        reg = file_input(d[0])
        self.reg = reg
        self.statusBar().showMessage(f'Данные введены из файла {d[0]}')
        Logger.log(f'Введены данные:\n{self.reg}\n', LogLevel.Info)
        self.setDefault()
    except ArithmeticError:
        Logger.log(f'Файл {d[0]} содержит некорректный регион',
LogLevel.Warn)
        QMessageBox(
            QMessageBox.Icon(QMessageBox.Icon.Critical),
            'Ошибка',
            f'Некорректный регион'
        ).exec()
        return
    except FileNotFoundError:
        Logger.log(f'Файл {d[0]} не найден', LogLevel.Warn)
        QMessageBox(
            QMessageBox.Icon(QMessageBox.Icon.Critical),
            'Ошибка',
            f'Файл {d[0]} не найден'
        ).exec()
    except ValueError:
        Logger.log(f'Некорректные данные в файле {d[0]}', LogLevel.Warn)
        QMessageBox(
            QMessageBox.Icon(QMessageBox.Icon.Critical),
            'Ошибка',
            f'Некорректные данные в файле {d[0]}'
        ).exec()

def onSettings(self) -> None:
    d = SettingsDialog(self)
    res = d.exec()
    if res:
        self.ga = d.gaType(d.pSelector, d.recombinator, d.mutationer,
d.oSelector)
        self.ga.params = d.params
        self.subResult = d.subResult
        self.statusBar().showMessage(f'Настройки применены')
        Logger.log(f'Текущие настройки ГА:\n{self.ga}\n', LogLevel.Info)
        self.setDefault()

def onStart(self) -> None:
    if not self.checkSetup():
        return
    self.setDefault()
    self.ga.start(self.reg)
    self.ga.gen = 0
    self.parents.setPopulation(self.ga.population.sorted())
    Logger.log(f'Сгенерирована популяция:\n{self.ga.population.sorted()}\n',
LogLevel.Info)

def setDefault(self) -> None:
    for w in [self.control.offspring, self.control.forceNext,
self.control.results]:
        w.setEnabled(True)
    for w in [self.control.mutate, self.control.next]:
        w.setEnabled(False)
    if self.ga is not None:
        self.ga.population = None
    self.parents.clear()

```

```

self.offspring.clear()
self.mutations.clear()

def onInfo(self) -> None:
    msg = \
        'Программа предоставляет средства для решения задачи коммивояжёра с
        помощью генетического алгоритма.\n\n' \
        'Ввод данных возможен с вручную, либо из файла с координатами точек
        (координаты разделены пробелами, сами ' \
        'точки на разных строках).\n\n' \
        'Для запуска необходимо ввести данные и выбрать настройки алгоритма
        (настройки сохраняются в файле config).' \
        'Далее требуется сгенерировать начальную популяцию (кнопка меню:
        начать). После можно управлять процессом ' \
        'алгоритма с помощью кнопок управления. \n' \
        'С помощью кнопки получить результат алгоритм будет запущен ' \
        'до указанного поколения (максимального), в процессе будут
        отображаться промежуточные результаты, ' \
        'промежуток отображения можно настроить (это влияет на
        производительность: чем меньше промежуток тем больше ' \
        'графиков отрисовывается).\n\n' \
        'Параметры можно настроить различными способами. Однако эмпирически
        оказался успешным следующий подход:\n' \
        '1. Количество особей должно быть равно количеству городов, либо
        немногим больше (где-то на 5-10) \n' \
        '2. Количество поколений в среднем для классического алгоритма
        должно быть в 2-10 раз больше чем количество особей.' \
        ' Для метода прерывистого равновесия это число может быть
        значительно меньше (2-5 раз). Для генитора ' \
        ' наоборот требуется больше поколений, в 100-500 раз больше размера
        популяции.\n' \
        '3. Операторы рекомбинации и мутации в среднем работают примерно
        одинаково и для выявления каких-то отличий ' \
        ' требуется проводить статистические тесты. Однако по результатам
        тестирования алгоритмов на ' \
        'подготовленных регионах сочетание операторов ОХ и Инверсии дают
        чуть лучший результат (субъективно).\n' \
        '4. Вероятность кроссинговера должна быть достаточно высокая (80-
        100%).\n' \
        '5. Размеры аллели кроссинговера влияют на разнообразие популяции.
        Соответственно чем аллель больше, тем ' \
        ' популяция разнообразнее, однако при этом алгоритм может колебаться
        вокруг оптимального решения. ' \
        'В целом если нужна быстрая сходимость этот параметр должен быть
        выше, если же нужно более точное решение, ' \
        'и есть возможность взять большое число поколений, то этот параметр
        можно уменьшить.\n' \
        '6. Вероятность мутации должна быть не слишком большая, и в
        зависимости от выбранных операторов она может ' \
        ' быть в диапазоне (5-50)%. Если разнообразия с другими параметрами
        достаточно, вероятность мутации можно ' \
        'уменьшить, если же наоборот разнообразия не достаточно (например
        если использовать рулетку и элитный отбор), ' \
        'тогда этот параметр можно увеличивать вплоть до 50% (возможно и
        больше).\n' \
        '7. Отбор родителей. Панмиксия даёт достаточное разнообразие, однако
        уменьшает сходимость. Рулетка наоборот ' \
        ' приводит к уменьшению разнообразия, однако и алгоритм сходится к
        оптимумам быстрее. При использовании ' \
        ' панмиксии, другие параметры лучше настраивать на ускорение
        сходимости. А при использовании рулетки, наоборот ' \
        ' остальные параметры рекомендуется настраивать на увеличение
        разнообразия (например размер аллели рекомбинации).' \

```

```
' Турнирный отбор можно гибко настроить с помощью размера турнира,
чем размер турнира больше? тем больше будет ' \
'более приспособленных особей, и тем меньше разнообразия.\n' \
'8. Отбор в новую популяцию. Элитарный отбор аналогично рулетке
уменьшает разнообразие популяции, однако ' \
'обеспечивает более быструю сходимость. Отбор усечением можно
настроить с помощью изменения пороговой величины, ' \
'чем она больше тем популяции разнообразнее и тем медленнее алгоритм
сходится.'
```

```
QMessageBox (
    QMessageBox.Icon (QMessageBox.Icon.Information),
    'Справка',
    msg
).exec ()
```