

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	5
1.	Требования к программе	6
1.1.	Исходные требования к программе	6
2.	План разработки	7
2.1	Параметры генетического алгоритма	7
2.2	План взаимодействия с программой и GUI	12
3.	Реализация	13
3.1	Модель	14
3.1.1	Классы ядра	14
3.1.2	Операторы ГА	16
3.1.3	Исполнители ГА	17
3.2	Графический интерфейс	17
3.2.1	Виджеты	17
4.	Тестирование	0
4.1	Тестирование графического интерфейса	0
4.2	Тестирование кода алгоритма	0
4.3	...	0
	Заключение	0
	Список использованных источников	0
	Приложение А. Исходный код	0

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.

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

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

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.

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

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

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.

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

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

- 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.

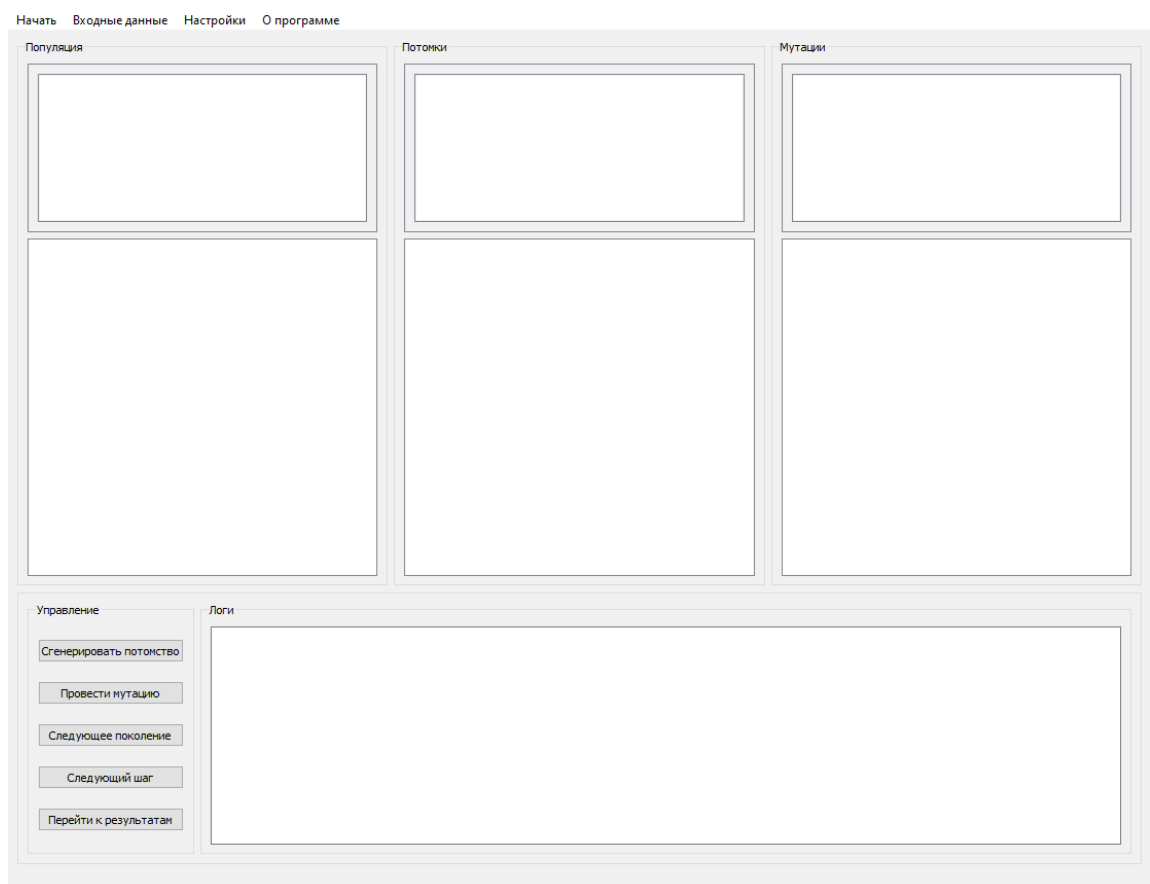


Рисунок 1: скетч GUI

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 Модель

Кодовая база модели задачи условно разделена на:

- Ядро: классы, представляющие точку маршрута, 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.

3.1.1 Классы ядра.

Город: представление точки маршрута (вектор).

```
class Town:
```

```
    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:
```

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

```
class Solution(list[int]):
```

```
    def __init__(self, dists: np.ndarray):
```

- Генерация случайного решения (особи)

```
def rand(length: int, dists: np.ndarray) -> Solution:
```

- Преобразование стандартного списка к решению

```
def list(dists: np.ndarray, lst: list[int]) -> Solution:
```

- Целевая функция

```
def F(self) -> float:
```

- Обратная к целевой (сумма длин всех путей в графе делённая на длину решения).

```
def rF(self) -> float:
```

- Циклический сдвиг последовательности

```
def shift(self, n: int) -> 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]):
```

```
def __init__(self, dists: np.ndarray):
```

- Случайная генерация популяции

```
def rand(dists: np.ndarray, psize: int):
```

- Преобразование стандартного списка к популяции

```
def list(dists: np.ndarray, lst: list[Solution]):
```

- Получение лучшей особи

```
def min(self):
```

- Копия популяции

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

- Сортировка особей по порядку длины решения

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

- Преобразования к строке

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

```
def __repr__(self):
```

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

Операторы отбора родителей: создают пары особей для рекомбинации

```
def panmixon(pop: Population) -> list[tuple[Solution, Solution]]:
```

```
def tournament(tsize: int, pop: Population) -> list[tuple[Solution, Solution]]:
```



```
def roulette(pop: Population) -> list[tuple[Solution, Solution]]:
```

Операторы рекомбинации: создают двух потомков от двух родителей

```
def pmx(p1: Solution, p2: Solution, start: int, size: int) -> list[Solution,  
Solution]:
```

```
def ox(p1: Solution, p2: Solution, start: int, size: int) -> list[Solution, Solution]:
```

```
def cx(p1: Solution, p2: Solution, start=0) -> list[Solution, Solution]:
```

Операторы мутации: производят мутацию особи

```
def swap(o: Solution, start: int = 0, end: int = None) -> Solution:
```

```
def insert(o: Solution, start: int = 0, end: int = None) -> Solution:
```

```
def inverse(o: Solution, start: int = 0, end: int = None) -> Solution:
```

Операторы отбора потомков: производят отбор в новую популяцию

```
def trunc(pop: Population, N: int, threshold: float) -> Population:
```

```
def elite(pop: Population, N: int) -> Population:
```

3.1.3 Исполнители ГА: представляют модификацию генетического алгоритма

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

```
class GA:  
    def __init__(self, towns: list[Town], psize: int):  
    def parentSelect(self):
```

```
def crossover(self)
def mutation(self):
def offspringSelect(self):
def newPopulation(self):
```

Классический ГА: [WIP]

Генитор: [WIP]

Метод прерывистого равновесия: [WIP]

3.2 Графический Интерфейс

3.2.1 Виджеты

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

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

```
class PopulationWidget(QWidget):
```

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

- Слот для отрисовки выбранной особи

```
    def drawSolution(self):
```

- Метод для установки набора особей

```
    def setPopulation(self, pop: Population, towns: list[Town]):
```

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

```
class LoggerWidget(QGroupBox):
```

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

- Печать переданной строки

```
    def print(self, s):
```

ControlWidget: виджет содержащий кнопки управления

```
class ControlWidget(QGroupBox):
```

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

3.2.2 Главное окно

4. ТЕСТИРОВАНИЕ

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

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

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ