Федеральное государственное автономное

образовательное учреждение

высшего образования

«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

институт

Кафедра Информатики

кафедра

**ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ**

Применение генетического алгоритма для решения задачи коммивояжера

тема

Преподаватель \_\_\_\_\_\_\_\_\_ В. В. Тынченко

подпись, дата инициалы, фамилия

Студент КИ19-17/1, 031940152 \_\_\_\_\_\_\_\_\_ П. И. Скрыль

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2021

# Задача

Разработать и применить генетический алгоритм для решения задачи коммивояжера. Разработанная программа должна поддерживать настройку параметров алгоритма и выводить график обучения с наилучшей, средней и наихудшей пригодностью популяции на каждом поколении.

# Задача коммивояжера и способы ее решения

Задача коммивояжера – одна из самых известных задач комбинаторной оптимизации. Коммивояжер – это сбытовой агент, который перемещается от одного пункта назначения к другому с целью сбыта товаров. Необходимо найти самый выгодный маршрут, который соединяет все пункты назначения с последующим возвратом в исходную точку. При этом посещение одного и того же пункта дважды запрещено.

Задача коммивояжера имеет множество различных трактовок и вариаций. Разработанный генетический алгоритм решает задачу нахождения очередности посещения пунктов таким образом, чтобы длина получившегося маршрута была минимальной. Считается, что все пункты соединены между собой.

Существует несколько способов решения поставленной задачи. Самый простой из них – полный перебор. Он заключается в составлении всех возможных маршрутов и выбора наилучшего из них. Он гарантирует нахождение оптимального результата, но крайне неэффективен с точки зрения времени вычисления. Маршрут всего из 15 пунктов имеет более 40 миллиардов комбинаций. Добавление нового пункта в маршрут кратно увеличивает количество возможных комбинаций. Для решения относительно сложной задачи методом полного перебора может потребоваться неразумное количество времени.

Вариацией метода полного перебора является метод случайного перебора. Случайным образом формируется определенное количество маршрутов. Из них выбирается наилучший. Этот метод не гарантирует нахождение оптимального маршрута, но позволяет добиться результата в разумные сроки.

Другим подходом к решению данной задачи является применение жадного алгоритма. Он заключается в принятии локально оптимальных решений на каждом этапе. Для задачи коммивояжера маршрут будет строиться соединением текущей точки маршрута с ближайшей к ней непосещенной точкой. Данный метод также не гарантирует нахождение оптимального результата, но позволяет добиться приближенных значений.

Одним из наиболее эффективных методов решения задачи коммивояжера является метод эластичной сети. Он заключается в отображении окружности на плоскость и растяжении ее в кольцо, проходящее около всех городов. Данный метод позволяет добиваться результатов максимально близких к оптимальному при небольших вычислительных затратах.

Еще одним эффективным методом решения задачи коммивояжера является генетический алгоритм. Генетический алгоритм относится к классу эвристических алгоритмов. Такие алгоритмы не гарантируют нахождение точного результата, но позволяют ускорить решение задачи в случаях, когда точное решение не может быть найдено.

# Описание разработанного генетического алгоритма

Разработанный генетический алгоритм содержит все этапы классического генетического алгоритма: инициализация популяции, отбор части популяции для порождения потомков, генерация потомков оператором скрещивания, мутация потомков, формирования новой популяции на основе текущей и потомков.

Каждый индивид обладает генотипом в виде упорядоченной последовательности точек, которые составляют маршрут. Инициализация популяции происходит случайным образом.

Для отбора индивидов к скрещиванию можно воспользоваться двумя методами. Первый – метод рулетки. Этот метод отбирает индивида пропорционально (чем больше функция пригодности, тем больше у индивида быть отобранным). Второй – метод туров. Из популяции случайным образом отбирается выбранное количество индивидов и из них выбирается наиболее пригодный.

Реализован метод одноточечного скрещивания. Случайным образом выбирается точка разрыва, родительские хромосомы разрываются в этой точке и обмениваются правыми частями. В результате такого скрещивания из двух родительских особей образуются два потомка.

Мутация потомков представляет собой обмен двух элементов в хромосоме.

Формирование новой популяции происходит замещением наименее пригодных индивидов из исходной популяции потомками, образовавшимися на текущей итерации.

Код программы, относящийся к генетическому алгоритму представлен на листингах 1-3.

Листинг 1 – Код файла «point.py»

import math  
  
  
class Point:  
 def \_\_init\_\_(self, x: int, y: int):  
 *"""  
 Инициализация точки  
 :param x: Координата x  
 :param y: Координата y  
 """* self.x = x  
 self.y = y  
  
 def get\_coord(self):  
 *"""  
 Вернуть координаты точки  
 :return: Список из x и y координаты  
 """* return [self.x, self.y]  
  
 def \_\_str\_\_(self):  
 *"""  
 Метод toStr  
 :return: Строковое представление точки  
 """* return f"x: {self.x}, y: {self.y}"  
  
 def \_\_eq\_\_(self, other):  
 *"""  
 Метод equals  
 :param other: Другая точка  
 :return: True если точки одинаковы, False в иных случаях  
 """* if not isinstance(other, Point):  
 return False  
 else:  
 return self.x == other.x and self.y == other.y  
  
 def distance(self, other) -> float:  
 *"""  
 Расстояние до полученной точки  
 :param other: Другая точка  
 :return: Расстояние между точками  
 """* return math.hypot(self.x - other.x, self.y - other.y)

Листинг 2 – Код файла «individual.py»

class Individual:  
 def \_\_init\_\_(self, genome: list):  
 self.genome = genome  
 self.fitness = self.\_\_fitness()  
 self.normalized\_fitness = -1  
  
 def \_\_fitness(self) -> float:  
 *"""  
 Функция пригодности  
 :return: Значение функции пригодности (меньше - лучше)  
 """* curr\_fitness = 0  
 for i in range(len(self.genome)):  
 if i == len(self.genome) - 1:  
 curr\_fitness += self.genome[i].distance(self.genome[0])  
 else:  
 curr\_fitness += self.genome[i].distance(self.genome[i + 1])  
 return curr\_fitness  
  
 def update\_fitness(self):  
 self.fitness = self.\_\_fitness()  
  
 def set\_normalized\_fitness(self, min\_fitness: float):  
 *"""  
 Нормализовать значения функции пригодности  
 :param min\_fitness: Значение худшей пригодности в текущем поколении  
 :return:  
 """* self.normalized\_fitness = -1 \* self.fitness + min\_fitness  
  
 def \_\_eq\_\_(self, other):  
 *"""  
 Метод equals  
 :param other: Другой индивид  
 :return: True если индивиды равны и False в любом другом случае  
 """* if not isinstance(other, Individual):  
 return False  
 if len(self.genome) != len(other.genome):  
 return False  
 for i in range(len(self.genome)):  
 if self.genome[i] != other.genome[i]:  
 return False  
 return True

Листинг 3 – Код файла «genetic\_master.py»

import math  
import random  
import copy  
import numpy as np  
import matplotlib.pyplot as plt  
  
from genetic.individual import Individual  
from genetic.point import Point  
  
  
class GeneticMaster:  
 def \_\_init\_\_(self, population\_size: int, crossover\_count: int, selection\_method: str,  
 mutation\_type: str, generations\_count: int, tour\_size):  
 *# Параметр - размер популяции* self.population\_size = population\_size  
 *# Параметр - кол-во пар участвующих в скрещивании  
 # ! Кол-во пар должно быть не больше половины размера популяции* self.crossover\_count = crossover\_count  
 *# Параметр - метод селекции ("tour" или "roulette")* self.selection\_method = selection\_method  
 *# Параметр - тип мутации ( "weak" или "med" или "strong")* self.mutation\_type = mutation\_type  
 *# Параметр - количество поколений* self.generations\_count = generations\_count  
 *# Параметр - размер тура (имеет смысл только тогда когда selection\_method == "tour")* self.tour\_size = tour\_size  
   
 self.points = []  
 self.super\_genome = []  
 self.metrics = {"max": [], "min": [], "avg": []}  
  
 def \_\_individual\_creator(self, points: list) -> Individual:  
 *"""  
 Создание индивида. Геном формируется случайным образом из списка переданных точек  
 :param points: Список точек  
 :return: Экземпляр класса Individual  
 """* local\_points = points.copy()  
 random.shuffle(local\_points)  
 return Individual(local\_points)  
  
 def population\_creator(self, points: list) -> list:  
 *"""  
 Создание популяции  
 :param points: Список точек для формирования генома  
 :param population\_size: Размер популяции  
 :return: Список экземпляров класса Individual  
 """* return [self.\_\_individual\_creator(points) for \_\_ in range(self.population\_size)]  
  
 def evaluate\_population(self):  
 *"""  
 Оценка данной популяции  
 ! Вызов данного метода обязателен перед проведением операции отбора кандидатов для скрещивания  
 :param population: Популяция  
 :return: Лучшая пригодность, худшая пригодность, средняя пригодность в популяции (абсолютные значения), лучший индивид  
 """  
 # Поиск значений функций пригодности* population = self.population  
 population.sort(key=lambda x: x.fitness)  
 best\_fitness = population[0].fitness  
 worst\_fitness = population[-1].fitness  
 avg\_fitness = np.mean([individual.fitness for individual in population])  
 best\_individual = copy.copy(population[0])  
 *# Обновление значений нормализованной пригодности  
 # Устанавливаются значения относительные для текущей популяции* for individual in population:  
 individual.set\_normalized\_fitness(worst\_fitness)  
 *# print(best\_fitness)  
 # print(worst\_fitness)  
 # print(avg\_fitness)* return best\_fitness, worst\_fitness, avg\_fitness, best\_individual  
  
 def candidate\_selection(self) -> list:  
 *"""  
 Селекция индивидов для скрещивания.  
 :param population: Популяция  
 :param crossover\_count: Количество отбираемых пар  
 :param method: Метод селекции "roulette" или "tour"  
 :return: Список списков пар отобранных индивидов  
 """* def roulette(selecting\_population: list, max\_selecting\_fitness: float) -> Individual:  
 *"""  
 Отбор индивида методом рулетки  
 :param selecting\_population: Популяция  
 :param max\_selecting\_fitness: Сумма нормализованной пригодности популяции  
 :return: Отобранный индивид  
 """* pick = random.uniform(0, max\_selecting\_fitness)  
 current = 0  
 for individual in selecting\_population:  
 current += individual.normalized\_fitness  
 if current > pick:  
 return individual  
  
 def tour(selecting\_population: list, selecting\_tour\_size: int) -> Individual:  
 *"""  
 Отбор индивида методом турнира  
 :param selecting\_population: Популяция  
 :param selecting\_tour\_size: Размер турнира  
 :return: Отобранный индивид  
 """  
 # Важно чтобы tour\_size был меньше размера population* tour\_members = []  
 *# for \_ in range(selecting\_tour\_size):  
 # potential\_member = random.choice(selecting\_population)  
 # while potential\_member in tour\_members:  
 # potential\_member = random.choice(selecting\_population)  
 # tour\_members.append(potential\_member)* tour\_indexes = random.sample(range(0, len(selecting\_population) - 1), selecting\_tour\_size)  
 for index in tour\_indexes:  
 tour\_members.append(selecting\_population[index])  
 tour\_members.sort(key=lambda x: x.normalized\_fitness, reverse=True)  
 return tour\_members[0]  
  
 population = self.population  
 crossover\_count = self.crossover\_count  
 method = self.selection\_method  
 selected\_candidates = []  
 if method == "roulette":  
 max\_fitness = sum(individual.normalized\_fitness for individual in population)  
 for \_ in range(crossover\_count):  
 individual1 = roulette(population, max\_fitness)  
 individual2 = roulette(population, max\_fitness)  
 *# while individual1 == individual2:  
 # individual2 = roulette(population, max\_fitness)  
 # print(individual1.fitness, individual1.normalized\_fitness)  
 # print(individual2.fitness, individual2.normalized\_fitness)* selected\_candidates.append([individual1, individual2])  
 elif method == "tour":  
 tour\_size = self.tour\_size  
 for \_ in range(crossover\_count):  
 individual1 = tour(population, tour\_size)  
 individual2 = tour(population, tour\_size)  
 *# while individual1 == individual2:  
 # individual2 = tour(population, tour\_size)* selected\_candidates.append([individual1, individual2])  
 else:  
 raise Exception("Некорректный способ проведения селекции")  
 return selected\_candidates  
  
 def crossover\_candidates(self, candidates: list) -> list:  
 *"""  
 Одноточечное скрещивание индивидов  
 :param candidates: Список возвращаемый методом candidate\_selection()  
 :return: Список новых созданных индивидов  
 """  
 # Список возвращаемых индивидов* new\_individuals = []  
 for pair in candidates:  
 *# Родительские геномы* parent1 = pair[0].genome  
 parent2 = pair[1].genome  
 child\_genome1 = []  
 child\_genome2 = []  
 *# Выбор точки разбиения геномов случайным образом* cut\_point = random.randint(1, len(parent1) - 1)  
 *# print(cut\_point)* for i in range(len(parent1)):  
 if i < cut\_point:  
 child\_genome1.append(parent1[i])  
 child\_genome2.append(parent2[i])  
 else:  
 if parent2[i] in child\_genome1:  
 for j in range(i, len(parent1)):  
 if parent2[j] not in child\_genome1:  
 child\_genome1.append(parent2[j])  
 break  
 if len(child\_genome1) < i + 1:  
 for j in range(len(parent1)):  
 if parent2[j] not in child\_genome1:  
 child\_genome1.append(parent2[j])  
 break  
 if parent1[i] in child\_genome2:  
 for j in range(i, len(parent1)):  
 if parent1[j] not in child\_genome2:  
 child\_genome2.append(parent1[j])  
 break  
 if len(child\_genome2) < i + 1:  
 for j in range(len(parent1)):  
 if parent1[j] not in child\_genome2:  
 child\_genome2.append(parent1[j])  
 break  
 *# print("Child 1")  
 # for i in range(len(child\_genome1)):  
 # print(child\_genome1[i])  
 # print("Child 2")  
 # for i in range(len(child\_genome2)):  
 # print(child\_genome2[i])* if len(child\_genome1) != len(parent1) or len(child\_genome2) != len(parent1):  
 raise Exception("Ошибка при скрещивании")  
 new\_individuals.append(Individual(child\_genome1))  
 new\_individuals.append(Individual(child\_genome2))  
 return new\_individuals  
  
 def mutation(self, population):  
 *"""  
 Мутация  
 :param mutation\_type: Тип мутации (вероятность, что ген мутирует)  
 :param population: Популяция  
 :return:  
 """* mutation\_type = self.mutation\_type  
 if mutation\_type == "strong":  
 chance = 2 / len(population)  
 elif mutation\_type == "med":  
 chance = 1 / len(population)  
 else: *# mutation\_type == "weak"* chance = 0.5 / len(population)  
 for individual in population:  
 for i in range(len(individual.genome)):  
 if random.random() <= chance:  
 j = random.randint(0, len(individual.genome) - 1)  
 while i == j:  
 j = random.randint(0, len(individual.genome) - 1)  
 individual.genome[i], individual.genome[j] = individual.genome[j], individual.genome[i]  
 individual.update\_fitness()  
 *# print("Got mutation")* def run(self, ax=None):  
 *"""  
 Запуск генетического алгоритма  
 :param canvas: FigureCanvasQTAgg для отображения графика  
 :return:  
 """* self.population = self.population\_creator(self.points)  
 self.super\_genome.clear()  
 for value in self.metrics.values():  
 value.clear()  
  
 super\_best = math.inf  
 *# super\_individual = None* super\_genome = []  
 for \_ in range(self.generations\_count):  
 best, worst, average, ind = self.evaluate\_population()  
 *# print(average)  
 # print(f"best: {best}")* self.metrics["max"].append(best)  
 self.metrics["avg"].append(average)  
 self.metrics["min"].append(worst)  
 if best < super\_best:  
 super\_best = best  
 *# super\_individual = ind* super\_genome = [Point(point.x, point.y) for point in ind.genome]  
 candidates = self.candidate\_selection()  
 legacy = self.crossover\_candidates(candidates)  
 self.population.sort(key=lambda x: x.fitness)  
 self.population = self.population[:-self.crossover\_count\*2]  
 self.mutation(legacy)  
 self.population += legacy  
  
   
 *# Построение графика* if (ax):  
 generations = list(range(1, self.generations\_count+1))  
 ax.plot(generations, self.metrics["avg"], label="avg")  
 ax.plot(generations, self.metrics["min"], label="min")  
 ax.plot(generations, self.metrics["max"], label="max")  
 ax.set\_xlabel("Поколение")  
 ax.set\_ylabel("Пригодность (меньше - лучше)")  
 ax.set\_title("Обучение генетического алгоритма")  
 ax.legend(loc=2)  
   
 *# self.super\_genome = super\_individual.genome* self.super\_genome = super\_genome  
  
 return super\_best, Individual(super\_genome)  
  
 def clear(self):  
 self.points.clear()  
 self.super\_genome.clear()

Рисунке 1 представлен графический интерфейс разработанной программы.

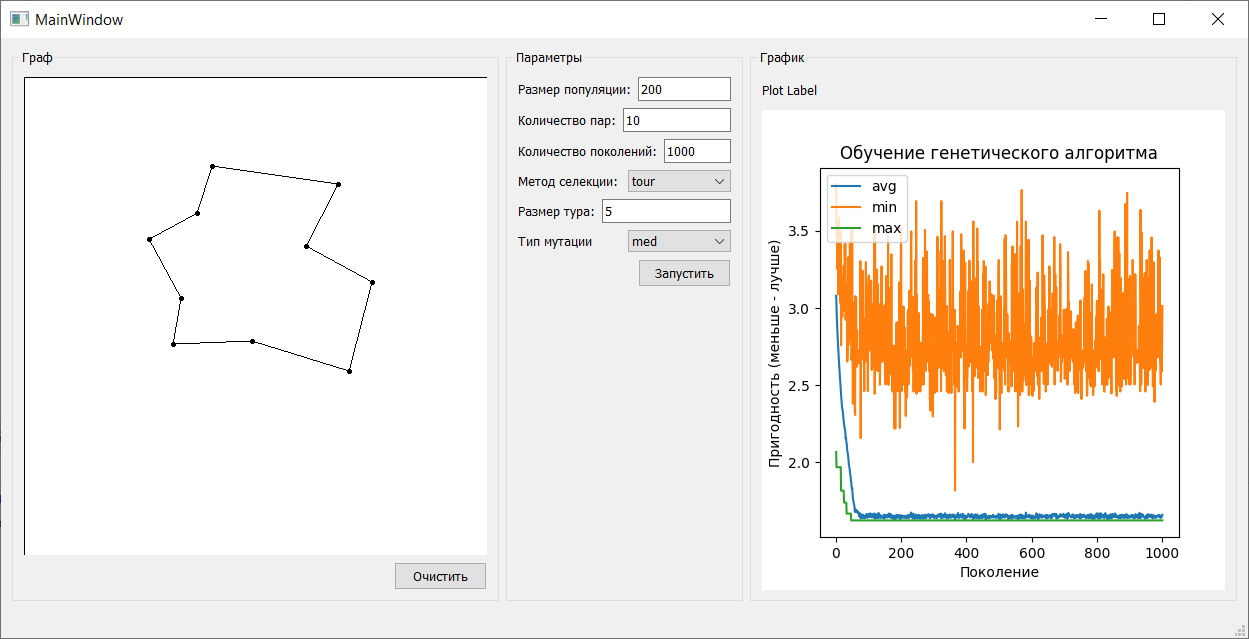
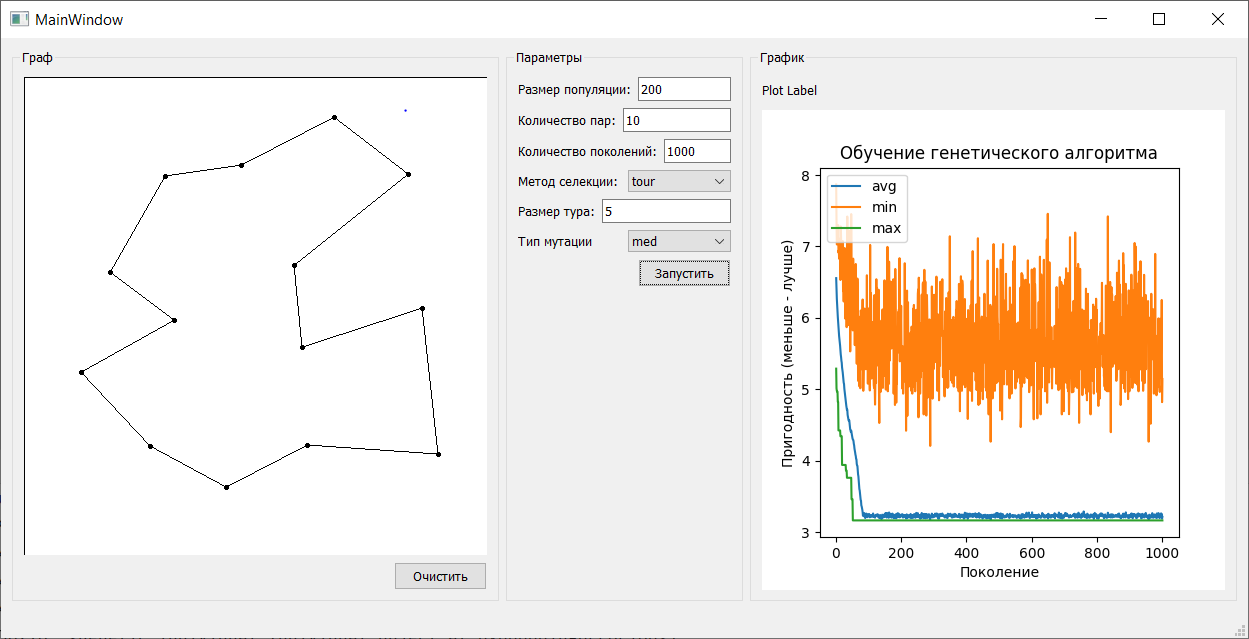


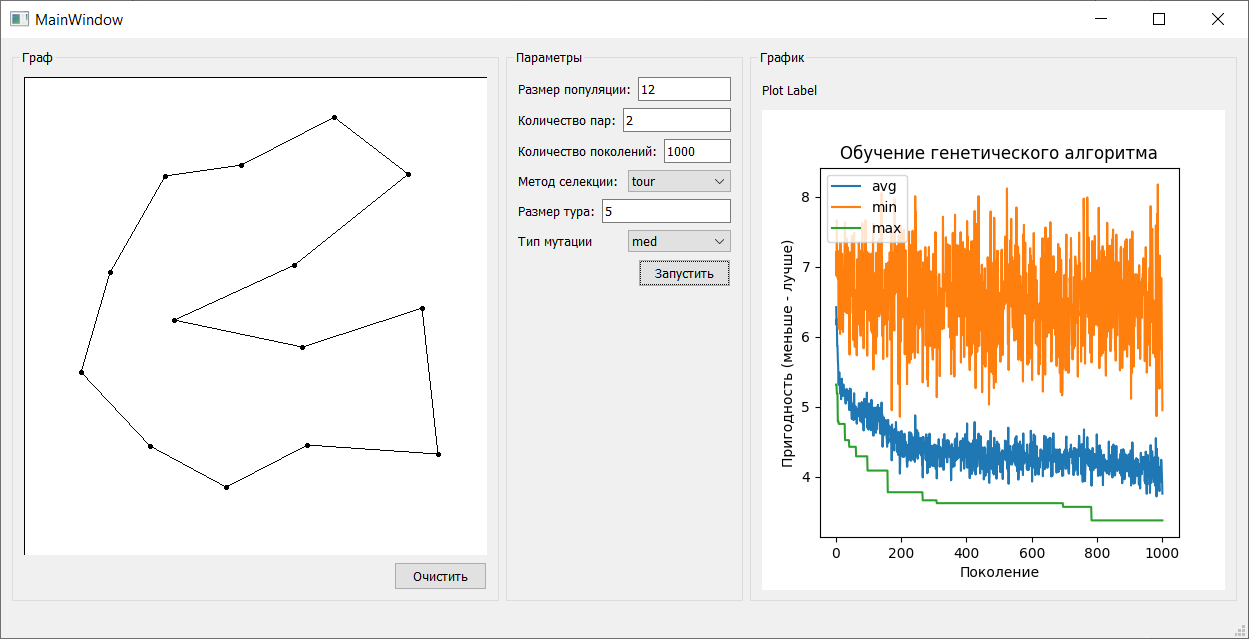
Рисунок 1 – Графический интерфейс программы

# Исследование влияния параметров на работу генетического алгоритма

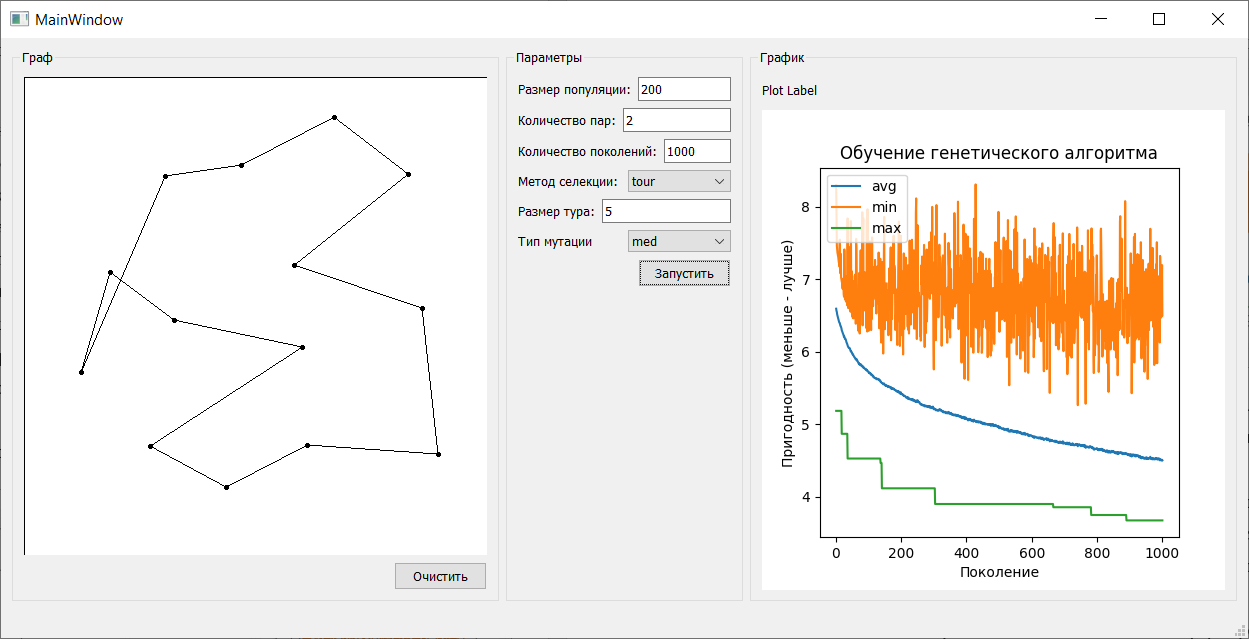
Разработанный генетический алгоритм позволяет настраивать следующие параметры: размер популяции, количество пар участвующих в скрещивании, количество поколений, метод селекции, размер тура (при методе селекции «tour»), тип мутации.

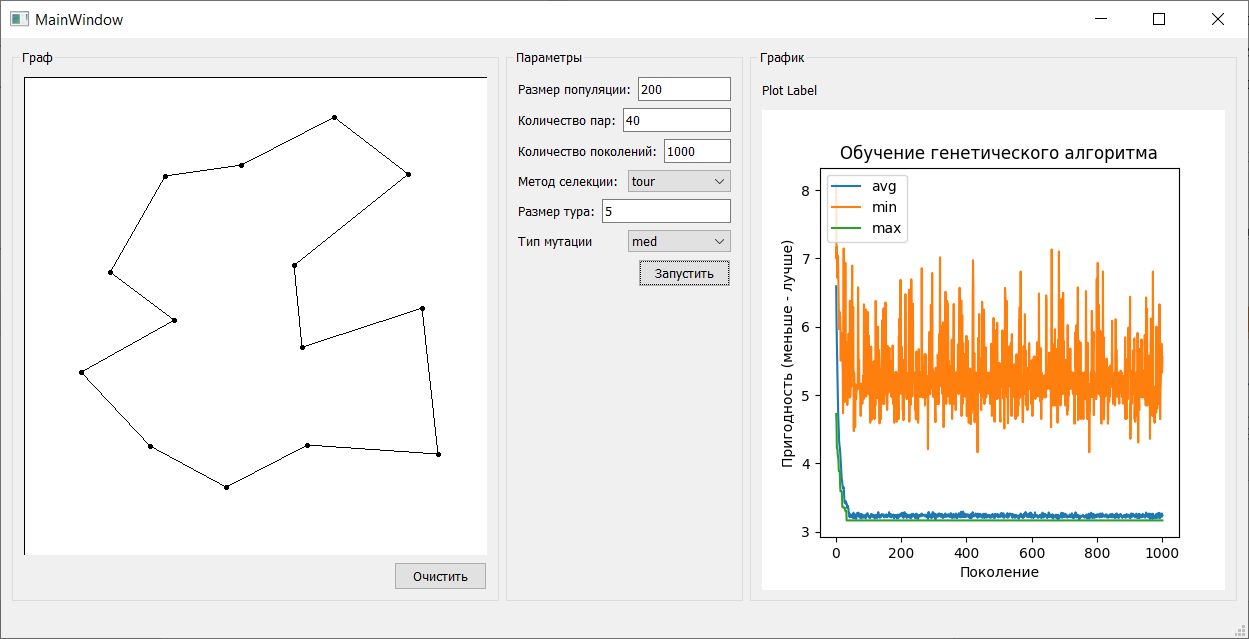
Размер популяции определяет количество индивидов в популяции. Выбор количества индивидов должен определяться количеством точек в составляемом маршруте. Слишком большой размер популяции превращает алгоритм в метод полного перебора. Например, для 5 пунктов можно составить 12 вариантов маршрута. Нет смысла выбирать размер популяции равной или превышающей 12. Слишком малый же размер популяции не позволит просмотреть достаточное количество вариантов маршрутов для нахождения оптимального.



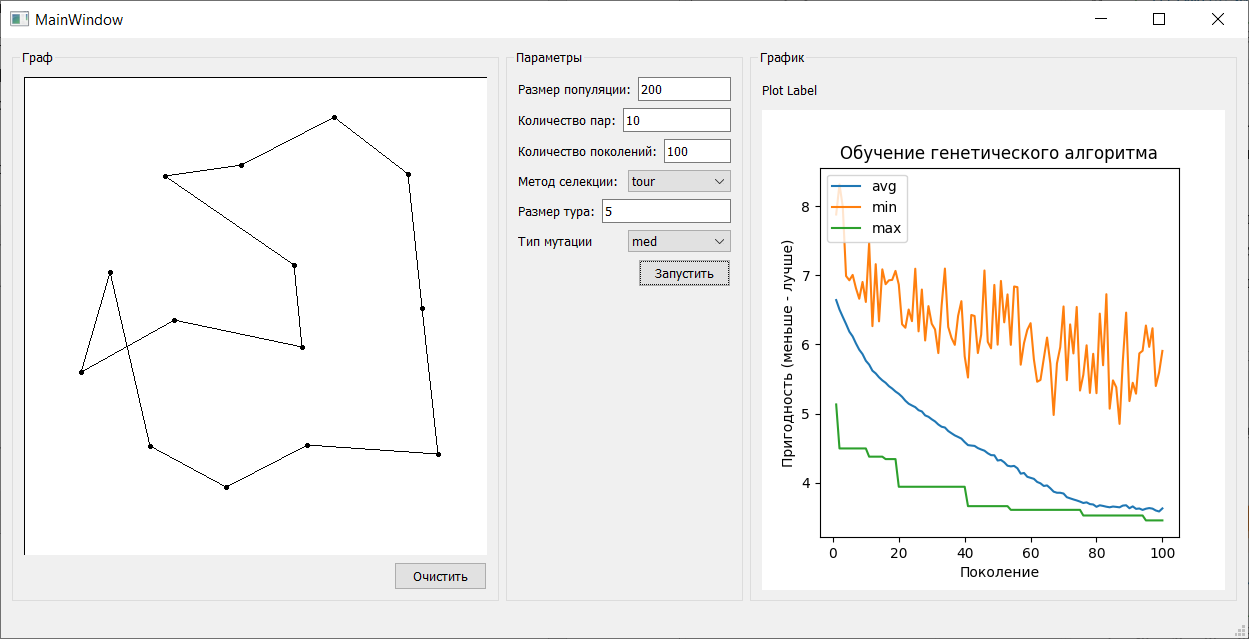


Количество пар участвующих в скрещивании определяет сколько пар индивидов необходимо отобрать для скрещивания и сколько пар потомков образуется в результате скрещивания. Выбор значения этого параметра должен определяться размером популяции. При слишком низком значении параметра популяция будет слабо изменяться, значит будет просмотрено недостаточное количество индивидов. При слишком высоком значении этого параметра популяция окажется неустойчивой и приблизится к методу случайного поиска.



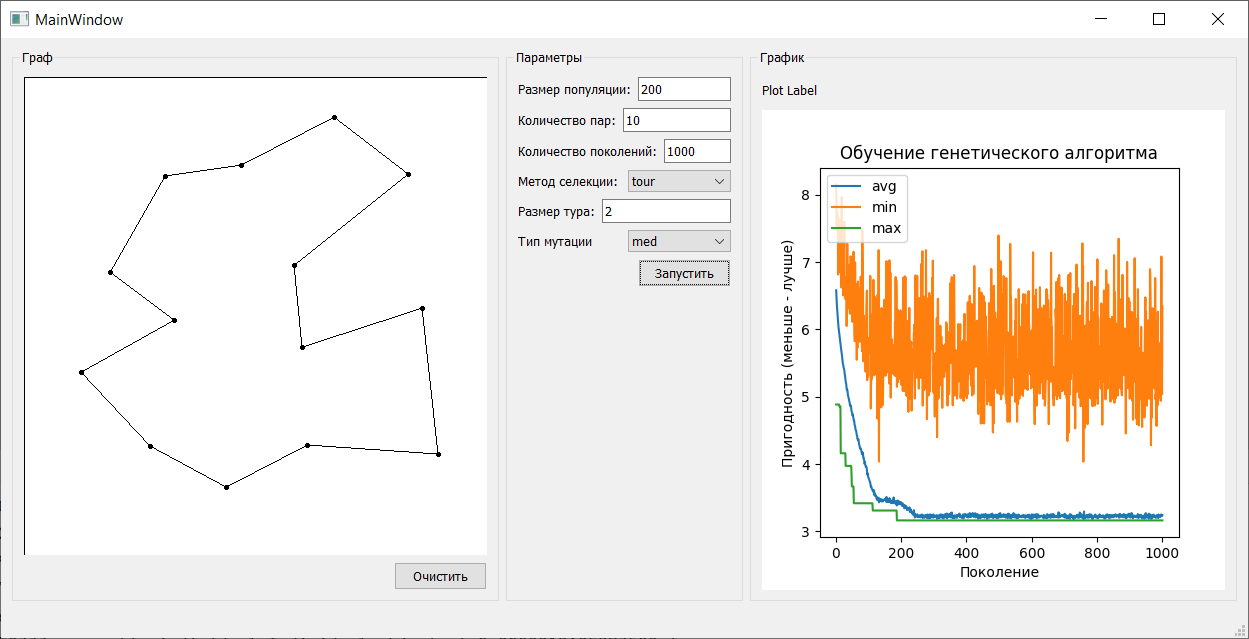


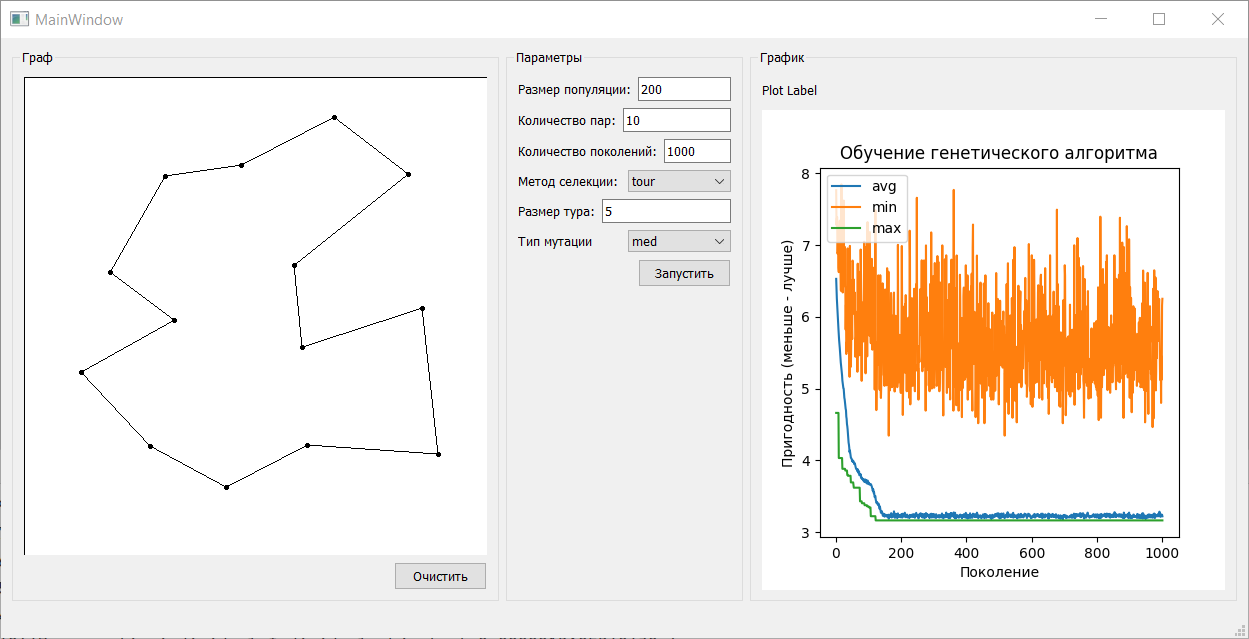
Количество поколений определяет количество циклов формирования и оценки популяции. При слишком малом значении этого параметра оптимальный результат не успеет сформироваться. Слишком большое значение параметра не имеет смысла, т. к. оптимальное решение может быть найдено раньше, а остальное время будет потрачено на бессмысленные вычисления.

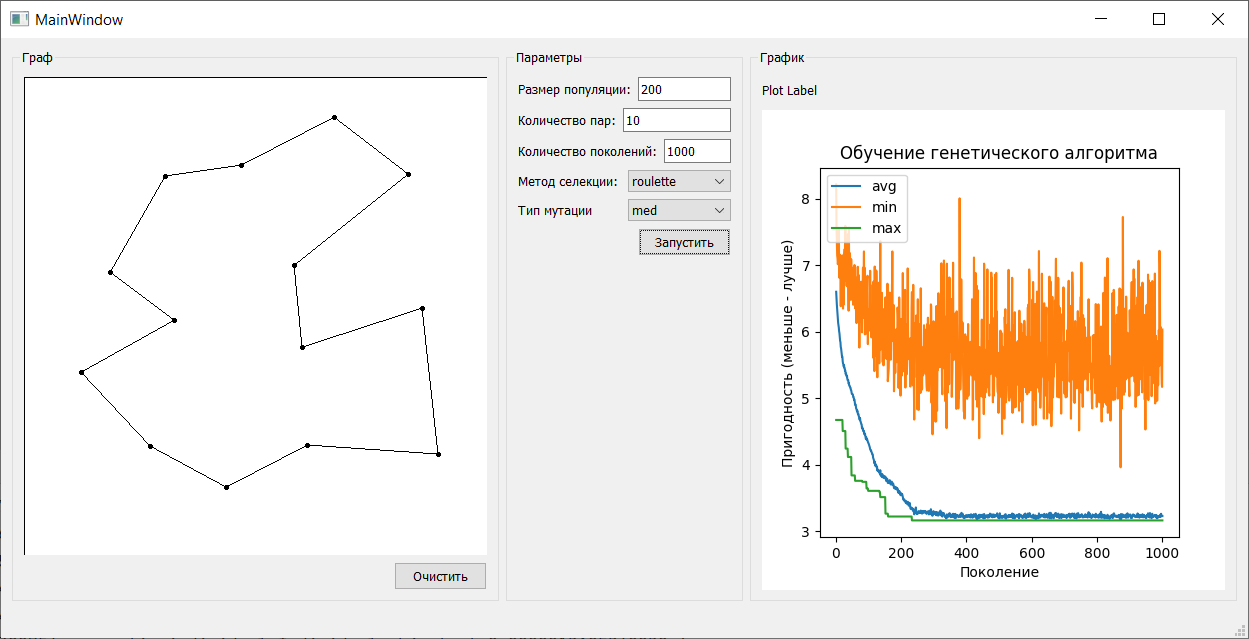


Метод селекции в программе может иметь два значения: «tour» и «roulette». Первое значение определяет селекцию методом тура. Второе – методом рулетки. Данные методы селекции описаны в предыдущем разделе.

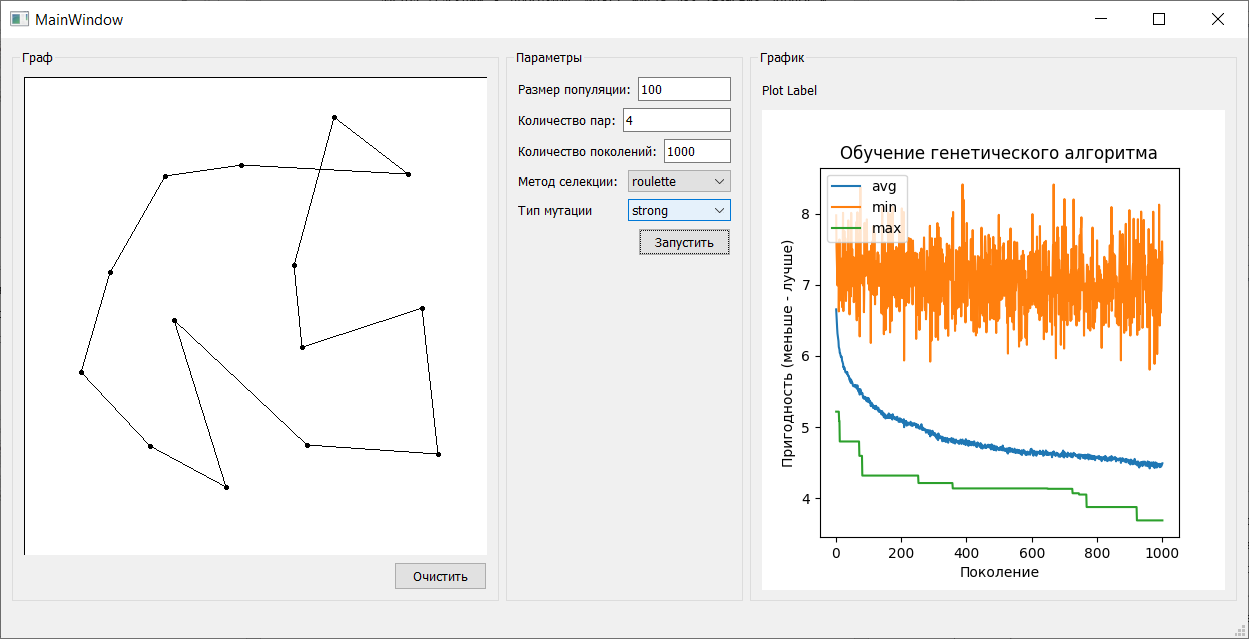
Размер тура – параметр, доступный для настройки, когда выбрана селекция методом тура. Он определяет количество индивидов, которые участвуют в туре для отбора. Размер тура принимается от 2 до размера популяции. Размер тура не стоит выбирать слишком большим, т.к. существует большая вероятность, что тур будет проходить один и тот же индивид с наибольшим значением функции пригодности.

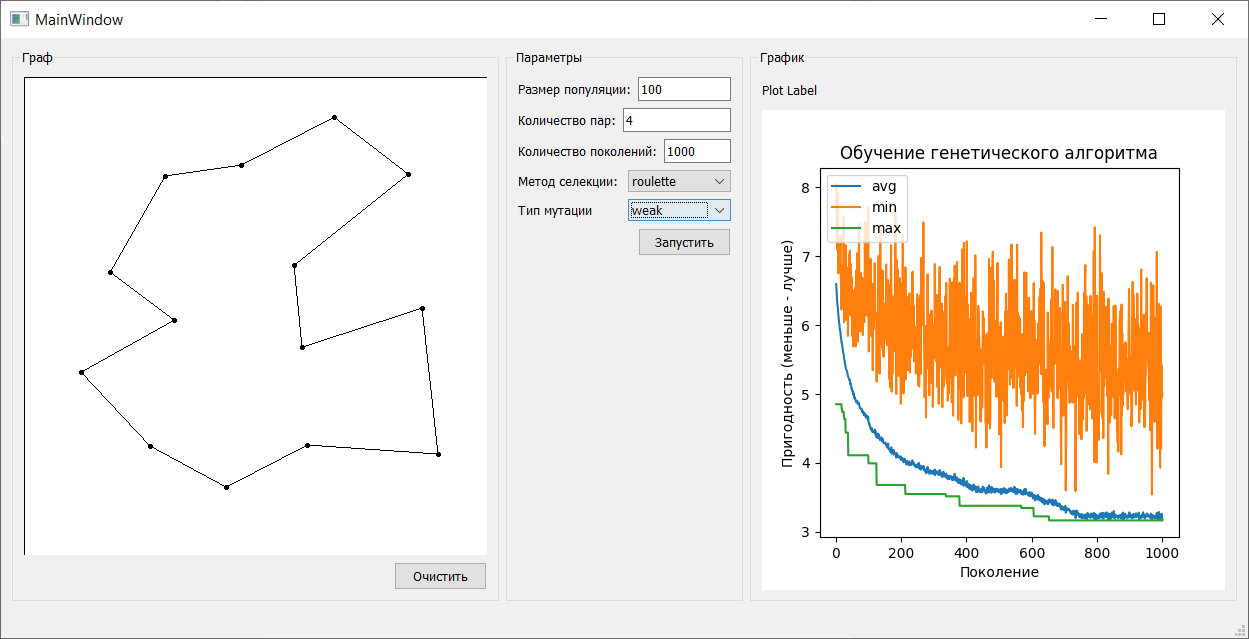






Тип мутации определяет вероятность мутации индивида. Для его настройки доступны три значения: «weak» - слабая мутация, «med» - средняя мутация, «strong» - сильная мутация. Выбор типа мутации должен основываться на других значениях параметров. При сильном типе мутации наследники будут более разнообразными, в то время как при слабом мутация будет слабо изменять наследников.





# Выводы

Получены навыки разработки генетических алгоритмов.