МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБО6РОСТРОЕНИЯ»

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «Решение задачи коммивояжера с помощью генетических алгоритмов» |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  | 13.10.24 |  | А.С. Вилюмсон |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**1. Задание по варианту**

Реализовать с использованием генетических алгоритмов решение задачи

коммивояжера по индивидуальному заданию согласно номеру варианта (см.

таблицу 3.1. и приложение Б.).

Сравнить найденное решение с представленным в условии задачи

оптимальным решением.

Представить графически найденное решение.

Проанализировать время выполнения и точность нахождения результата

в зависимости от вероятности различных видов кроссовера, мутации.

***Вариант 7 — eil51.tsp***

**2. Основные сведения из теории**

Задача коммивояжера (ЗК) считается классической задачей генетических

алгоритмов. Она заключается в следующем: путешественник (или

коммивояжер) должен посетить каждый из базового набора городов и

вернуться к исходной точке. Имеется стоимость билетов из одного города в

другой. Необходимо составить план путешествия, чтобы сумма затраченных

средств была минимальной. Поисковое пространство для ЗК- множество из N

городов. Любая комбинация из N городов, где города не повторяются,

является решением. Оптимальное решение – такая комбинация, стоимость

которой (сумма из стоимостей переезда между каждыми из городов в

комбинации) является минимальной. Представление соседства

В представлении соседства тур является списком из n городов. Город J

находится на позиции I только в том случае, если маршрут проходит из

города I в город J. Например, вектор (2 4 8 3 9 7 1 5 6) представляет

следующий тур: 1-2-4-3-8-5-9-6-7 . Каждый маршрут имеет только одно

соседское представление, но некоторые векторы в соседском представлении

могут представлять неправильный маршрут. Например, вектор (2 4 8 1 9 3 5 7

6) обозначает маршрут 1-2-4-1..., т.е. часть маршрута – замкнутый цикл. Это

представление не поддерживает классическую операцию кроссовера. Три

операции кроссовера были определены и исследованы для соседского

представления: alternating edges (альтернативные ребра), subtour chunks

(куски подтуров), heuristic crossovers (евристический кроссовер).

**3. Листинг программы**

import random  
import matplotlib.pyplot as plt  
import matplotlib  
import math  
  
import numpy as np  
  
matplotlib.use('TkAgg')  
path = [  
 0, 21, 7, 25, 30, 27, 2, 35, 34, 19,  
 1, 28, 20, 15, 49, 33, 29, 8, 48, 9,  
 38, 32, 44, 14, 43, 41, 39, 18, 40, 12,  
 24, 13, 23, 42, 6, 22, 47, 5, 26, 50,  
 45, 11, 46, 17, 3, 16, 36, 4, 37, 10,  
 31  
]  
def euclidean\_distance(coord1, coord2):  
 return math.sqrt((coord2[0] - coord1[0]) \*\* 2 + (coord2[1] - coord1[1]) \*\* 2)  
# Генерация случайного пути (индивида)  
def neighbor\_to\_path(neighbor\_representation):  
 size = len(neighbor\_representation)  
 path = [0]  
 current\_city = 0  
 for \_ in range(1, size):  
 next\_city = neighbor\_representation[current\_city]  
 path.append(next\_city)  
 current\_city = next\_city  
 return path  
  
def path\_to\_neighbor(path):  
 size = len(path)  
 neighbor\_representation = [0] \* size  
 for i in range(size - 1):  
 neighbor\_representation[path[i]] = path[i + 1]  
 neighbor\_representation[path[-1]] = path[0]  
 return neighbor\_representation  
def generate\_individual\_path(cities):  
 individual = cities[:]  
 random.shuffle(individual)  
 return individual  
def generate\_individual(cities):  
 normal\_individual = generate\_individual\_path(cities)  
 return path\_to\_neighbor(normal\_individual)  
  
def fitness(individual, distances):  
 total\_distance = 0  
 curren\_city = 0  
 next\_city = 0  
 for \_ in range(len(individual)):  
 current\_city = next\_city  
 next\_city = individual[current\_city]  
 total\_distance += distances[current\_city][next\_city]  
 return total\_distance  
  
  
# Кроссинговер (Alternating Edges)  
def crossover\_alternating\_edges(parent1, parent2, crossover\_rate):  
 if random.random() > crossover\_rate:  
 return parent1[:]  
 n = len(parent1)  
 offspring = [-1] \* n  
 used = set()  
  
 current\_city = parent1[0]  
 offspring[0] = current\_city  
 used.add(current\_city)  
 used.add(0)  
  
 for i in range(1, n-1):  
 if i % 2 == 1:  
  
 next\_city = parent2[current\_city]  
 else:  
  
 next\_city = parent1[current\_city]  
  
 if next\_city in used:  
 next\_city = random.choice([city for city in parent1 if city not in used])  
  
 offspring[current\_city] = next\_city  
 used.add(next\_city)  
 current\_city = next\_city  
 offspring[offspring.index(-1)]=0  
 return offspring  
  
  
def get\_next\_city(parent, current\_city, child):  
 idx = parent.index(current\_city)  
 next\_city = parent[(idx + 1) % len(parent)]  
  
 if next\_city in child:  
 available\_cities = [city for city in parent if city not in child]  
 return random.choice(available\_cities) if available\_cities else -1  
  
 return next\_city  
  
def crossover\_subtour\_chunks(parent1, parent2, crossover\_rate):  
 if random.random() < crossover\_rate:  
 return parent1[:]  
  
 size = len(parent1)  
 child = [-1] \* size  
  
 start, end = sorted(random.sample(range(size), 2))  
 subtour = parent1[start:end]  
  
 child[start:end] = subtour  
  
 pointer = end  
 for city in parent2:  
 if city not in subtour:  
 if pointer >= size:  
 pointer = 0  
 child[pointer] = city  
 pointer += 1  
  
 return child  
  
  
def crossover\_heuristic(parent1, parent2, distances,crossover\_rate):  
  
 if random.random() < crossover\_rate:  
 return parent1[:]  
 n = len(parent1)  
 used = set()  
  
 offspring = [-1] \* n  
  
  
 current\_city = parent1[0]  
 offspring[0] = current\_city  
 used.add(current\_city)  
 used.add(0)  
  
 for i in range(1, n-1):  
 next\_city1 = parent1[current\_city]  
 next\_city2 = parent2[current\_city]  
  
 if distances[current\_city][next\_city1] < distances[current\_city][next\_city2]:  
 next\_city = next\_city1  
 else:  
 next\_city = next\_city2  
  
 while next\_city in used:  
 next\_city = random.choice([city for city in parent1 if city not in used])  
  
  
 offspring[current\_city] = next\_city  
 used.add(next\_city)  
  
 current\_city = next\_city  
 offspring[offspring.index(-1)] = 0  
 return offspring  
  
  
  
def mutate(individual, mutation\_rate):  
 if random.random() < mutation\_rate:  
 path = neighbor\_to\_path(individual)  
  
 idx1, idx2 = random.sample(range(len(path)), 2)  
  
 path[idx1], path[idx2] = path[idx2], path[idx1]  
  
 individual = path\_to\_neighbor(path)  
 return individual  
  
  
  
def selection(population, fitnesses):  
 tournament = random.sample(list(zip(population, fitnesses)), 3)  
 return min(tournament, key=lambda x: x[1])[0]  
  
  
  
def plot\_path(individual, coordinates, generation, best\_distance):  
 current\_city = 0  
 path = [current\_city]  
 for \_ in range(len(individual) - 1):  
 next\_city = individual[current\_city]  
 path.append(next\_city)  
 current\_city = next\_city  
 x = [coordinates[city][0] for city in path] + [coordinates[path[0]][0]]  
 y = [coordinates[city][1] for city in path] + [coordinates[path[0]][1]]  
  
 plt.figure(figsize=(6, 6))  
 plt.plot(x, y, 'o-', label=f'Generation {generation}')  
 plt.scatter(x[0], y[0], c='red', label='Start/End', zorder=5)  
 for i, city in enumerate(individual):  
 plt.text(coordinates[city][0], coordinates[city][1], str(city+1), fontsize=12, ha='right')  
  
 plt.title(f'Path at Generation {generation} Best fit = {best\_distance:.3f}')  
 plt.xlabel('X')  
 plt.ylabel('Y')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
  
def load\_tsp(filename):  
  
 with open(filename, 'r') as file:  
 lines = file.readlines()  
  
 coordinates = []  
 reading\_coords = False  
  
 for line in lines:  
 if "NODE\_COORD\_SECTION" in line:  
 reading\_coords = True  
 continue  
 if "EOF" in line:  
 break  
 if reading\_coords:  
 parts = line.split()  
 x, y = float(parts[1]), float(parts[2])  
 coordinates.append((x, y))  
  
 return coordinates  
  
def genetic\_algorithm(cities, distances, coordinates, pop\_size=100, generations=500,  
 crossover\_rate=0.8, mutation\_rate=0.05, elitism=True, elite\_size=10, crossover\_method="heuristic",population = None):  
 # Инициализация популяции  
 if population is None:  
 population = [generate\_individual(cities) for \_ in range(pop\_size)]  
 for generation in range(generations):  
 fitnesses = [fitness(ind, distances) for ind in population]  
 # Сортируем популяцию по фитнесу (наименьший фитнес — лучший путь)  
 population\_sorted = sorted(population, key=lambda ind: fitness(ind, distances))  
  
 new\_population = []  
 fitnesses\_sorted = [fitness(ind, distances) for ind in population\_sorted]  
 # Элитарность: сохраняем лучших индивидов (по умолчанию 1)  
 if elitism:  
 new\_population.extend(population\_sorted[:elite\_size])  
  
 # Эволюция нового поколения для оставшихся индивидов  
 while len(new\_population) < pop\_size:  
 parent1 = selection(population, fitnesses)  
 parent2 = selection(population, fitnesses)  
 if crossover\_method == "alternating":  
 child = crossover\_alternating\_edges(parent1, parent2, crossover\_rate)  
 elif crossover\_method == "subtour":  
 child = crossover\_subtour\_chunks(parent1, parent2, crossover\_rate)  
 elif crossover\_method == "heuristic":  
 child = crossover\_heuristic(parent1, parent2, distances, crossover\_rate)  
 child\_safe = child  
 child = mutate(child, mutation\_rate)  
 if child[0]==0:  
 child = child\_safe  
 new\_population.append(child)  
 # Преобразование списка списков в множество кортежей для удаления дубликатов  
 new\_population = list(set(tuple(individual) for individual in new\_population))  
  
 # Преобразование обратно в список списков  
 new\_population = [list(individual) for individual in new\_population]  
 new\_population\_sorted = sorted(new\_population, key=lambda ind: fitness(ind, distances))  
 population = new\_population\_sorted  
 fitnesses = [fitness(ind, distances) for ind in population]  
 best\_fitness = min(fitnesses)  
 best\_individual = min(population, key=lambda ind: fitness(ind, distances))  
 #print(f"BP: {best\_individual}, BF: {best\_fitness}")  
  
  
  
 best\_individual = min(population, key=lambda ind: fitness(ind, distances))  
 return best\_individual, fitness(best\_individual, distances),population  
def run\_genetic\_algorithm\_with\_varying\_parameters():  
 coordinates = load\_tsp("eil51.tsp")  
 cities = list(range(len(coordinates)))  
 distances = [[euclidean\_distance(coordinates[i], coordinates[j]) for j in cities] for i in cities]  
  
 crossover\_methods = ["alternating", "heuristic"]  
 crossover\_rates = [i \* 0.05 for i in range(1, 21)] # От 0.05 до 1.0  
 mutation\_rates = [i \* 0.05 for i in range(1, 11)] # От 0.05 до 0.5  
  
  
 steps\_data = []  
  
 for crossover\_method in crossover\_methods:  
 print(f"Тестирование метода кроссинговера: {crossover\_method}")  
  
 for crossover\_rate in crossover\_rates:  
 for mutation\_rate in mutation\_rates:  
 print(f" Кроссинговер с вероятностью {crossover\_rate:.2f} и мутация с вероятностью {mutation\_rate:.2f}")  
 best\_fitness = float('inf')  
 generation = 0  
 population = None  
  
 while best\_fitness > 450 and generation < 2000:  
 generation += 1  
  
 best\_individual, best\_fitness, population = genetic\_algorithm(  
 cities, distances, coordinates, pop\_size=100, generations=1,  
 crossover\_rate=crossover\_rate, mutation\_rate=mutation\_rate,  
 elitism=True, elite\_size=10, crossover\_method=crossover\_method,  
 population=population  
 )  
 # Записываем количество шагов (итераций)  
 steps\_data.append((crossover\_method, crossover\_rate, mutation\_rate, len(population)))  
  
 # После выполнения всех тестов строим график  
 steps\_data = np.array(steps\_data)  
  
 # Создаем сетку для подграфиков  
 fig, axs = plt.subplots(len(crossover\_methods), 1, figsize=(10, 15))  
  
 for idx, crossover\_method in enumerate(crossover\_methods):  
 ax = axs[idx]  
 method\_data = steps\_data[steps\_data[:, 0] == crossover\_method]  
  
 # Разделяем по вероятности кроссинговера и мутации  
 for mutation\_rate in mutation\_rates:  
 subset = method\_data[method\_data[:, 2] == mutation\_rate]  
 ax.plot(subset[:, 1], subset[:, 3], label=f"Mutation rate = {mutation\_rate:.2f}")  
  
 ax.set\_title(f"{crossover\_method.capitalize()} Crossover")  
 ax.set\_xlabel("Crossover Rate")  
 ax.set\_ylabel("Number of Steps")  
 ax.legend()  
  
 plt.tight\_layout()  
 plt.show()  
  
def run\_genetic\_algorithm\_steps():  
 coordinates = load\_tsp("eil51.tsp")  
 cities = list(range(len(coordinates)))  
 #cities = list(range(20))  
 #coordinates = {i: (random.randint(0, 100), random.randint(0, 100)) for i in cities}  
 distances = [[euclidean\_distance(coordinates[i], coordinates[j]) for j in cities] for i in cities]  
 population = None  
 while True:  
  
  
 generations = int(input("Введите число шагов (итераций) для выполнения (или 0 для выхода): "))  
  
 if generations == 0:  
 print("Выход из программы...")  
 break  
  
  
 best\_solution, best\_distance, population = genetic\_algorithm(  
 cities, distances, coordinates, generations=generations,crossover\_rate=0.6,mutation\_rate=0.2,population=population  
 )  
 print(f"Лучший путь: {best\_solution}, длина пути: {best\_distance}")  
  
 plot\_path(best\_solution, coordinates, generations, best\_distance)  
  
  
  
  
def menu():  
 while True:  
 print("\n=== Меню ===")  
 print("1. Запустить генетический алгоритм с заданным числом шагов")  
 print("2. Исследовать")  
 print("3. Выйти")  
 choice = input("Выберите пункт меню: ")  
  
 if choice == '1':  
 run\_genetic\_algorithm\_steps()  
 elif choice == "2":  
 run\_genetic\_algorithm\_with\_varying\_parameters()  
 elif choice == '3':  
 print("Выход...")  
 break  
 else:  
 print("Неверный ввод. Попробуйте снова.")  
  
  
  
menu()

**4. Исследование**

Я исследовал зависимость достижения в зависимости от типа кроссинговера и вероятности кроссинговера и мутации.

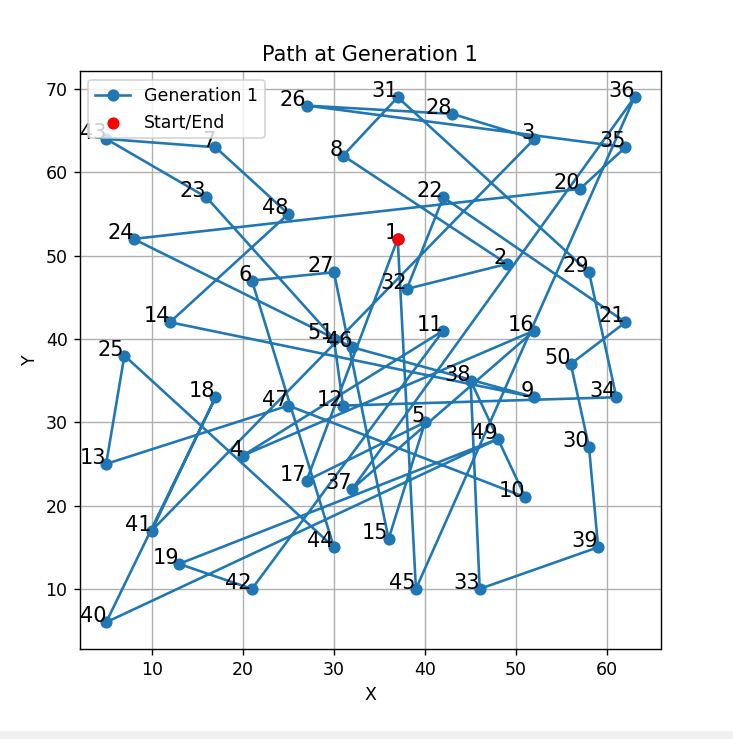
Заметно, что оптимум вероятности мутации находится около 0.2

Заметно что эвристический кроссовер значительно лучше

Заметен оптимум в районе 0.6

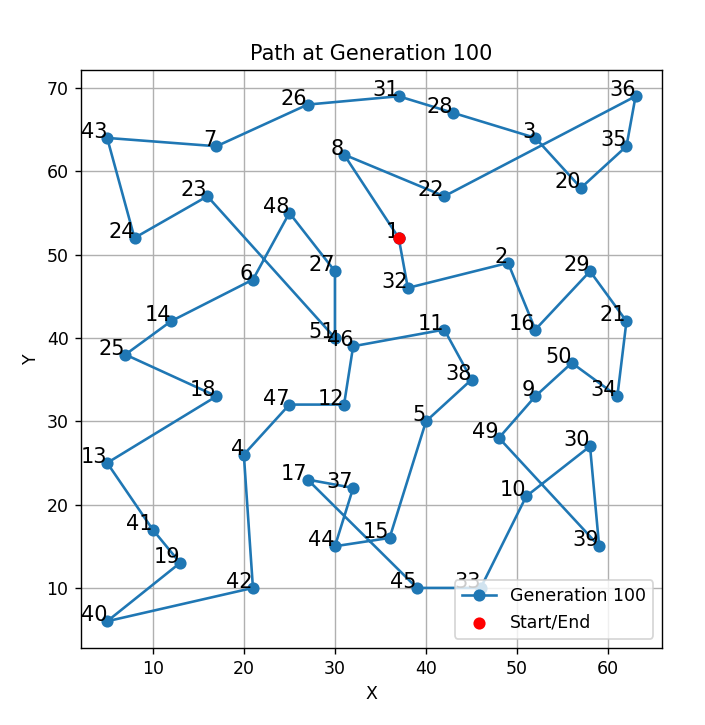
**5. Выполнение симуляции**

**Шаг 1**

****Начальный лучший путь

Длина пути — 1254

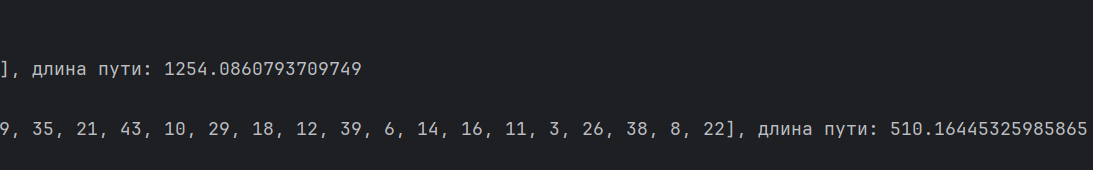
**Шаг 100**

****Популяция спустя 100 шагов

Суммарный путь резко уменьшился, но всё ещё велик

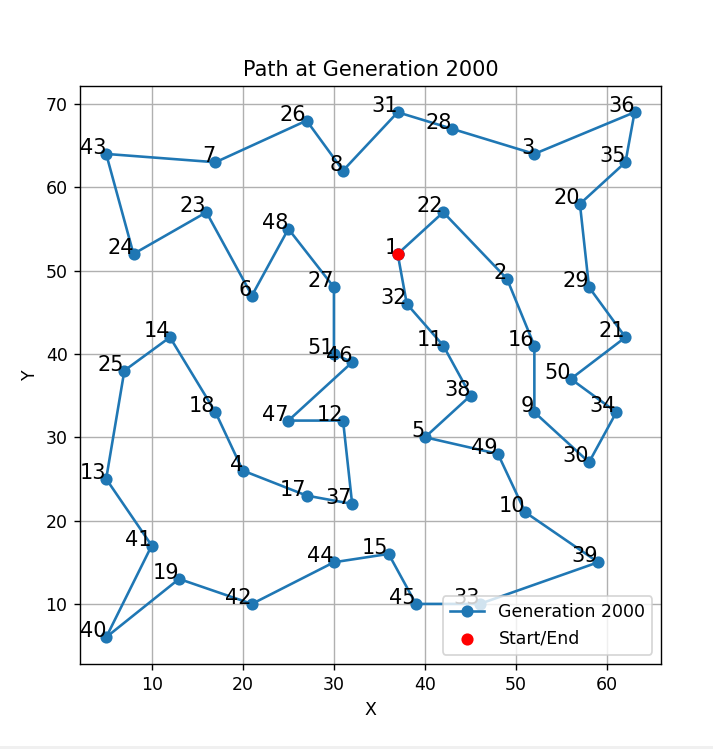
Длина пути — 510

**Шаг 15**

****Популяция спустя 15 шагов

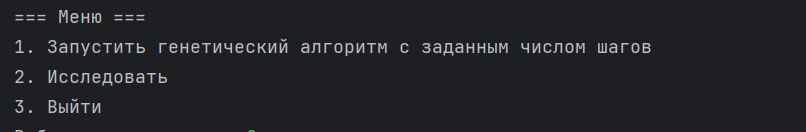
Лучшая точка— (427.932,430.149):-821.218

**Поколение 2000**

****Популяция спустя 2000 шагов

Длина пути — 438

Идеальный путь имеет длину 429, те отличается на 9 от текущего, что незначительно.

  
Скриншот вывода программы во время поиска

**6. Контрольный вопрос**

**Тур в представлении пути, кроссинговеры частично-отображенный**

**(PMX), порядковый (OX), циклический (CX)**

Тур в представлении пути — маршрут представлен последовательностью точек в порядке и посещения, самый интуитивный и эффективный способ заданя тура. Для него уществуют такие операции кроссинговера как:

* **PMX –** выбираем точки разреза на родителях, сегменты кроме центрального меняются местами и если возможно то подставляются. По запол строится «преобразование данных», например 1<-> 8 и заполняются оставшиеся ячейки.
* **OX –** аналогично выбираем две точки разреза, путь между ними копируется потомкам. Оставшиеся части заполняем в том же порядке как было у противоположного родителя.
* **CX –** каждый потомок начинает порождаться от одного родителя. Берётся первый город от первого родителя, затем каждый город берётся из позиции в которой он стоит в противоположном родителе. Когда брать города становится невозможным, оставшиеся берутся из второго родителя.