ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

д-р техн. наук, профессор		Скобцов Ю.А.
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. № 4132		Н.И. Карпов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

СОДЕРЖАНИЕ

1 Индивидуальное задание

Вариант 6:

6	Berlin52.tsp	Представление соседства

2 Краткие теоретические сведения

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.

Кажется естественным, что представление тура – последовательность (i1, i2, … , in), где (i1, i2, … , in) – числа из множества (1 … n), представляющие определенный город. Двоичное представление городов неэффективно, т.к. требует специального ремонтирующего алгоритма: изменение одиночного бита может повлечь неправильность тура. В настоящее время существует три основных представления пути: соседское, порядковое и путевое. Каждое из этих представлений имеет собственные полностью различные операторы рекомбинации.

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 кроссинговера.

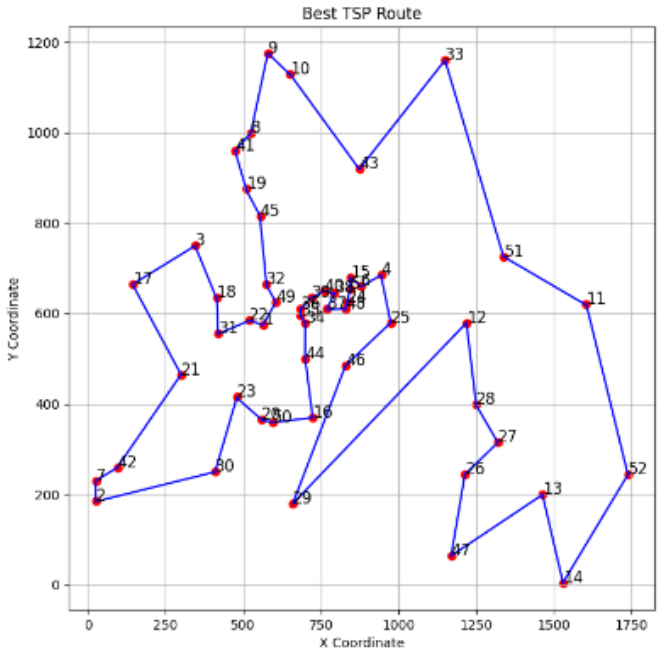
Эвристический кроссинговер (heuristic crossover) строит потомков, выбирая случайный город как стартовую точку для маршрута – потомка.

Потом он сравнивает два соответствующих ребра от каждого из родителей и выбирает более короткое. Затем конечный город выбирается как начальный для выбора следующего более короткого ребра из этого города. Если на каком–то шаге получается замкнутый тур, тур продолжается любым случайным городом, который еще не посещался.

3 Результаты выполнения работы

3.1 Исследование лучшего решения

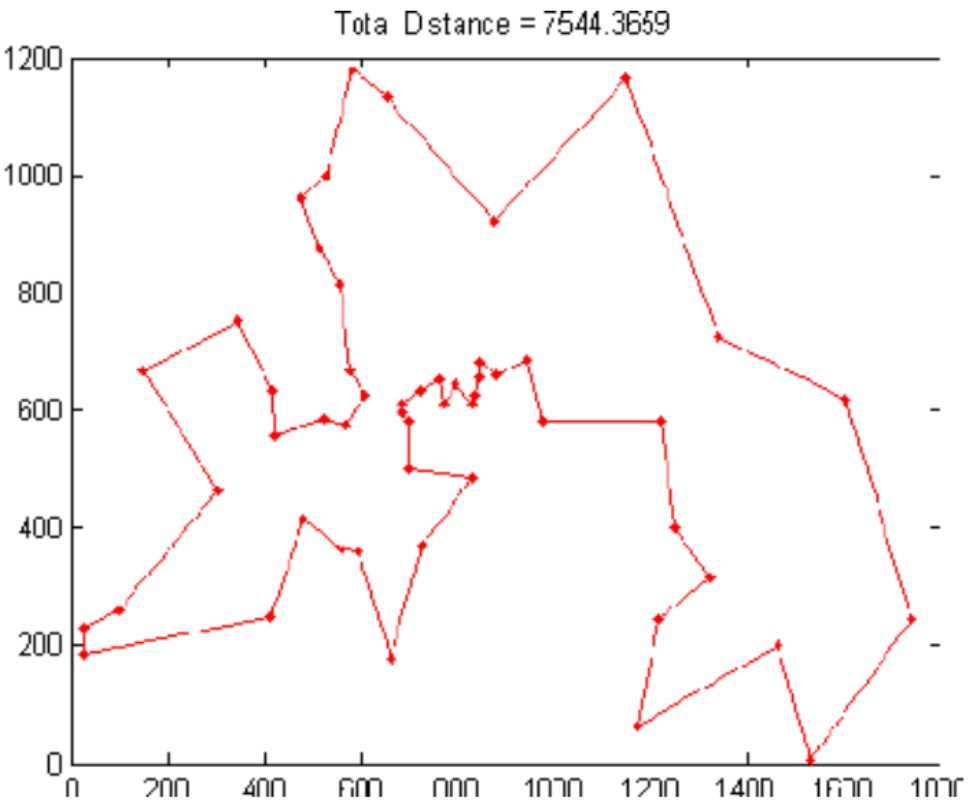
В ходе выполнения программного решения были получены следующие результаты:




```
Generation [96/100]     Min distance = 8006.301
Generation [97/100]     Min distance = 8005.409
Generation [98/100]     Min distance = 8005.409
Generation [99/100]     Min distance = 8098.698

 ============================================================================================
Required time: 30.60s. Found answer: 8098.697679.
 Best solution: 1-22-31-18-3-17-21-42-7-2-30-23-20-50-16-44-34-35-36-39-40-38-37-48-24-5-15-6-4-25-46-29-12-28-27-26-47-13-14-52-11-51-33-43-10-9-8-41-19-45-32-49
```

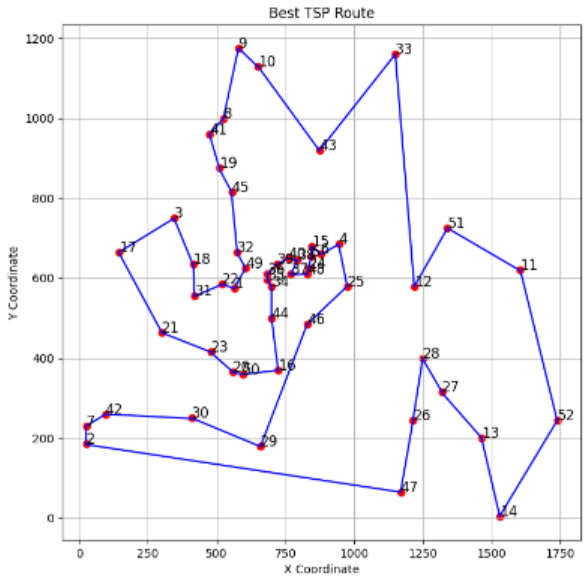
Полученное решение близко к оптимальному решению. Общая дистанция тура составила 8099 метров, в оптимальном решении – 7544 метра:



3.3 Исследование решений при разных параметрах

В ходе работы было установлено, что размер популяции положительно влияет на точность, но негативно влияет на скорость вычислений:

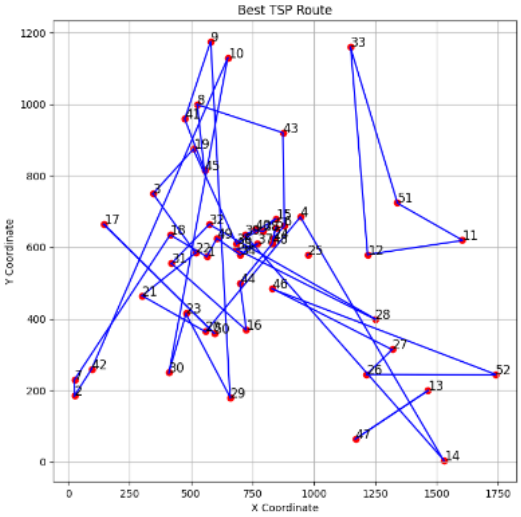
Размер популяции 2000:



```
Generation [97/100]     Min distance = 8386.076
Generation [98/100]     Min distance = 8417.739
Generation [99/100]     Min distance = 8226.591

 ============================================================================================================
Required time: 29.55s. Found answer: 8226.590621.
```

Размер популяции 20:




```
Generation [95/100]     Min distance = 12893.312
Generation [96/100]     Min distance = 13399.181
Generation [97/100]     Min distance = 12877.913
Generation [98/100]     Min distance = 12528.745
Generation [99/100]     Min distance = 12398.480

 ============================================================================================================
Required time: 0.16s. Found answer: 12398.480301.
```

Большая вероятность кроссинговера или мутации уменьшает шанс попасть в локальный минимум, но ухудшает сходимость:

```
Generation [98/100]     Min distance = 9129.960
Generation [99/100]     Min distance = 9129.960

 ============================================================================================
Required time: 1.02s. Found answer: 9129.959751.
 Best solution: 1-39-37-5-15-6-4-48-24-25-12-11-52-13-14-47-26-27-28-51-33-43-10-9-41-8-19-45-31-18-3-17-21-23-42-2-7-30-29-16-44-46-20-50-22-32-49-36-40-38-34-35
 ([39, 7, 17, 48, 15, 4, 30, 19, 41, 9, 52, 11, 14, 47, 6, 44, 21, 3, 45, 50, 23, 32, 42, 25, 12, 27, 28, 51, 16, 29, 18, 49, 43, 35, 1, 40, 5, 34, 37, 38, 8, 2, 10, 46, 31, 20, 26, 24, 36, 22, 33, 13]) Pc = 0.4, Pm = 0.01
```

```
Generation [96/100]     Min distance = 8278.525
Generation [97/100]     Min distance = 8278.525
Generation [98/100]     Min distance = 8278.525
Generation [99/100]     Min distance = 8278.525

 ============================================================================================
Required time: 1.52s. Found answer: 8278.525479.
 Best solution: 1-22-44-39-40-24-48-38-37-34-35-36-50-20-23-31-18-3-17-21-42-7-2-30-29-16-46-25-15-5-6-4-12-28-27-47-26-13-14-52-11-51-33-43-10-9-8-41-19-45-32-49
 ([22, 30, 17, 12, 6, 4, 2, 41, 8, 9, 51, 28, 14, 52, 5, 46, 21, 3, 45, 23, 42, 44, 31, 48, 15, 13, 47, 27, 16, 29, 18, 49, 43, 35, 36, 50, 34, 37, 40, 24, 19, 7, 10, 39, 32, 25, 26, 38, 1, 20, 33, 11]) Pc = 0.9, Pm = 0.01
```

3.3 Листинг программы

```
import genalg

POPULATION_SIZE = 200

CROSSING_OVER_PROBABILITY = 0.9
MUTATION_PROBABILITY = 0.01
MAX_GENERATIONS = 100
FILENAME = "berlin52.tsp"

if __name__ == '__main__':
    genetic_optimizer = genalg.GeneticTSPSolver(FILENAME, MAX_GENERATIONS,
POPULATION_SIZE, CROSSING_OVER_PROBABILITY,
                                                MUTATION_PROBABILITY)

    genetic_optimizer.start()
import matplotlib.pyplot as plt
import numpy as np

from time import time
import random


def count_euclidean_distance(city1, city2):
    """
    Calculates the Euclidean distance between two coordinate points.
    :param city1: first (x, y) coordinates
    :param city2: second (x, y) coordinates
    :return: float number - distance between two points
    """
    return np.linalg.norm(city1 - city2)


def check_nlr_for_cycle(neighbor_list):
    """
    Checks if the is valid neighbor representation list (closed, without
cycles)
    :param neighbor list: list of neighbors (chromosome)
    :return: True if it is valid, False otherwise
    """
    # return True
    n = len(neighbor_list)
    visited = [False] * n  # All visited cities.
    current_city = 0  # Start from city 0.

    for _ in range(n):
        if visited[current_city]:
            # If current city is visited then we got cycle.
            return False
        visited[current_city] = True
        next_city = neighbor_list[current_city] - 1

        # Go to next city.
        current_city = next_city

    # After visiting n cities check if we are in the start city.
    return current_city == 0


def convert_neighbor_list_to_route(neighbor_list, start_city=1):
```

```
    """
    Converts a neighbor list representation into an ordered route.
    :param neighbor_list: list of neighbors (neighbor list representation)
    :param start_city: city to start the route (1-based index)
    :return: list of cities in the order of the route
    """
    n = len(neighbor_list)
    route = [start_city]
    current_city = start_city

    for _ in range(n - 1):
        next_city = neighbor_list[current_city - 1]
        route.append(next_city)
        current_city = next_city

    return route


def format_route(route):
    """
    Formats the given route as a string with cities connected by dashes.
    :param route: list of cities
    :return: string representing the route
    """
    return '-'.join(map(str, route))


class GeneticTSPSolver:
    """
    Solves task of finding maximum of target function (fitness function)
using genetic algorithm.
    :param tsp_filename: file with tsp initial data
    :param population_size: size of the population
    :param max_generations: number of max available generations. For a sit-
uation when it is impossible to find a satisfying optimum
    :param crossover_probability: probability of a crossover
    :param mutation_probability: probability of a mutation
    """

    def __init__(self, tsp_filename: str, max_generations: int, popula-
tion_size: int,
                 crossover_probability: float, mutation_probability:
float):
        self.crossover_probability = crossover_probability
        self.max_generations = max_generations
        self.population_size = population_size
        self.mutation_probability = mutation_probability
        self.cities = []
        self.dimensions = 0

        # Read from file coordinates, cities and number of cities (dimen-
sions).
        self.coords = self.read_tsp_file(tsp_filename)

        # Distances matrix.
        self.distances = np.zeros(shape=(self.dimensions, self.dimensions))
        self.calculate_distances()

        self.population = []
        self.total_distances = []
```

```
        # Initialize population with random shuffled cities arrays
        for i in range(population_size):
            is_invalid_way = True

            while is_invalid_way:
                chromosome = self.cities.copy()
                random.shuffle(chromosome)

                # Check is there random generated way valid.
                is_invalid_way = not check_nlr_for_cycle(chromosome)

                if not is_invalid_way:
                    self.population.append(chromosome)
                    self.total_distances.append(0.)

        # array = [49, 7, 18, 6, 24, 15, 42, 9, 10, 43, 52, 25, 47, 13, 5,
29, 3, 31, 41, 23, 17, 1, 30, 48, 4, 27, 28,
        #          12, 50, 2, 22, 45, 51, 44, 34, 35, 40, 37, 36, 39, 8,
21, 33, 46, 19, 16, 26, 38, 32, 20, 11, 14]
        #
        # self.population = [array, array, array, array]

    def read_tsp_file(self, filename) -> np.array:
        """
        Reads tsp file and save data to self cities, coords and dimensions
(number of dimensions)
        :param filename: name of the tsp file
        """
        with open(filename, 'r') as file:
            lines = file.readlines()

        start_read_coords = False

        coords = []

        for line in lines:
            line = line.strip()

            if line == "NODE_COORD_SECTION":
                start_read_coords = True
                continue

            if line == "EOF":
                break

            if start_read_coords:
                parts = line.split()
                self.cities.append(int(parts[0]))
                x = float(parts[1])
                y = float(parts[2])
                coords.append((x, y))

        self.dimensions = len(self.cities)
        return np.array(coords)


    def calculate_distances(self):
        """Creates Matrix of distances between cities"""
        for i in range(self.dimensions):
            for j in range(self.dimensions):
                self.distances[i][j] =
```

```
count_euclidean_distance(self.coords[i], self.coords[j])

    def start(self):
        generation = 1  # generations counter

        start_time = time()

        while generation < self.max_generations:
            # 1. Evaluate current distances.
            self.evaluate()

            new_population = self.population.copy()

            for i in range(0, self.population_size, 2):
                # 2. Reproduction.
                element1 = self.reproduction()
                element2 = self.reproduction()

                # 3. Crossing.
                if random.random() < self.crossover_probability:
                    element1 = self.crossover(element1, element2)
                    element2 = self.crossover(element2, element1)

                new_population[i] = element1
                new_population[i + 1] = element2

            for i in range(len(new_population)):
                self.population[i] = new_population[i]

            # 4. Mutate.
            self.mutate()

            print(f'Generation [{generation}/{self.max_generations}]\t\t' +
                  f'Min distance = {min(self.total_distances):.3f}\t\t')

            generation += 1

        end_time = time()

        print('\n', '=' * 100)

        best_index = self.total_distances.index(min(self.total_distances))
        best_solution = self.population[best_index]
        route = convert_neighbor_list_to_route(best_solution)
        formatted_route = format_route(route)

        print(f'Required time: {end_time - start_time:.2f}s. Found answer:
{min(self.total_distances):4f}.\n',
              f'Best solution: {formatted_route}\n',
              f'({best_solution})',
              f'Pc = {self.crossover_probability}, Pm =
{self.mutation_probability}')

        self.plot_route(best_solution)

    def evaluate(self):
        """
        Evaluates current total distances for current population.
        """
        for i in range(len(self.population)):
            current_total_distance = 0.
```

```
            # Count total distance for every chromosome in population.
            for j in range(self.dimensions):
                # Use neighbour representation.
                current_city = self.population[i][j]
                prev_city = j + 1

                distance = self.distances[current_city - 1, prev_city - 1]
                current_total_distance += distance

            self.total_distances[i] = current_total_distance

    def reproduction(self, k=3):
        """
        Tournament based reproduction algorithm.
        """
        selected = random.sample(list(zip(self.population,
self.total_distances)), k)
        return min(selected, key=lambda x: x[1])[0]

    def crossover(self, parent1, parent2):
        """
        Heuristic crossover between two parents.
        :param parent1: first parent (chromosome)
        :param parent2: second parent (chromosome)
        :return: child (chromosome)
        """
        child = [-1] * self.dimensions  # Empty child chromosome
        used_cities = set()  # Set of cities that have already been added
to the child

        # Start from a random city from parent1
        current_city = random.randint(1, self.dimensions)
        start_city = current_city
        used_cities.add(start_city)

        for i in range(1, self.dimensions+1):
            # Find the next city from parent1 or parent2 based on the
shortest distance
            next_city_p1 = parent1[current_city - 1]
            next_city_p2 = parent2[current_city - 1]

            # Choose the city with the shortest distance
            if next_city_p1 not in used_cities and next_city_p2 not in
used_cities:
                distance_p1 = self.distances[current_city - 1][next_city_p1
- 1]
                distance_p2 = self.distances[current_city - 1][next_city_p2
- 1]

                if distance_p1 < distance_p2:
                    next_city = next_city_p1
                else:
                    next_city = next_city_p2
            elif next_city_p1 in used_cities and next_city_p2 in
used_cities:
                # If no valid next city, choose a random unvisited city
                remaining_cities = list(set(range(1, self.dimensions + 1))
- used_cities)
                if len(remaining_cities) > 0:
                    next_city = random.choice(remaining_cities)
```

```
                else:
                    # All cities were visited -> next city = start city
                    next_city = start_city
            elif next_city_p2 in used_cities:
                next_city = next_city_p1
            else:
                next_city = next_city_p2

            # Add the chosen city to the child
            child[current_city - 1] = next_city
            used_cities.add(next_city)

            current_city = next_city

        return child

    def select(self):
        """Selects population_size best chromosomes from current_population
into population."""
        while len(self.population) > self.population_size:
            worse_chromosome_index =
self.total_distances.index(max(self.total_distances))
            self.population.pop(worse_chromosome_index)
            self.total_distances.pop(worse_chromosome_index)

    def mutate(self):
        """
        Inversion mutation: inverts the order of a random subsequence of
cities in the route.
        """
        for i in range(self.population_size):
            if random.random() < self.mutation_probability:
                # Randomly select two cities to invert the segment between
them
                city1, city2 = sorted(random.sample(range(self.dimensions),
2))

                # Invert the subsequence between city1 and city2
                self.population[i][city1:city2 + 1] = re-
versed(self.population[i][city1:city2 + 1])

    def plot_route(self, best_route):
        """
        Plots the best found TSP route.
        """
        plt.figure(figsize=(8, 8))

        plt.scatter(self.coords[:, 0], self.coords[:, 1], color='red')

        for i in range(len(best_route)):
            plt.plot([self.coords[i, 0]-1, self.coords[best_route[i]-1,
0]],
                     [self.coords[i, 1]-1, self.coords[best_route[i]-1,
1]], 'b')

        for i, city in enumerate(best_route):
            plt.text(self.coords[city - 1][0], self.coords[city - 1][1],
str(city), fontsize=12)

        plt.title('Best TSP Route')
        plt.xlabel('X Coordinate')
```

```
    plt.ylabel('Y Coordinate')
    plt.grid(True)
    plt.show()
```

4 Вывод

В ходе работы была создана программа для решения задачи коммивояжера с помощью генетических алгоритмов. Полученные решения близки к оптимальному решению, что демонстрирует эффективность генетических алгоритмов в задачах подобного рода.