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

Оптимизация путей на графах с помощью муравьиных алгоритмов

по курсу: Эволюционные методы проектирования программно-информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4132		Н.И. Карпов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

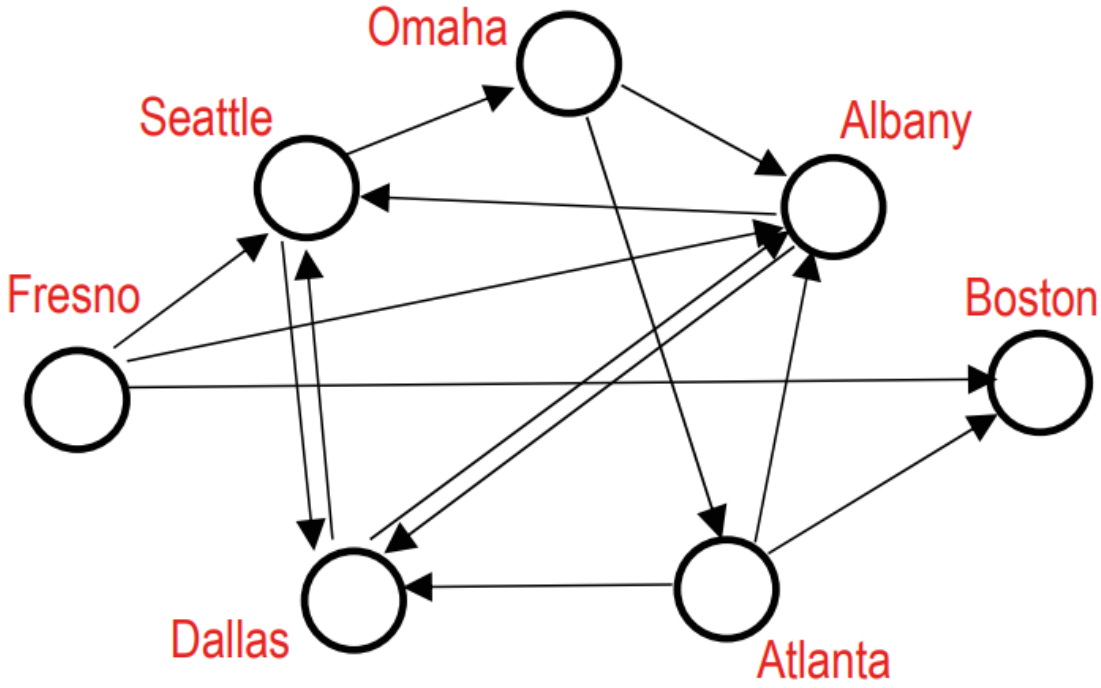
СОДЕРЖАНИЕ

1 Индивидуальное задание 3
2 Краткие теоретические сведения 3
3 Результаты выполнения работы 4
3.1 Поиск Гамильтонова пути 4
3.2 Решение задачи коммивояжера 6
3.4 Листинг программы 10
4 Вывод 15

1 Индивидуальное задание

Вариант 6:

1) Создать программу, использующую МА для решения задачи поиска гамильтонова пути на следующем графе:



2) Создать программу, использующую МА для решения задачи коммивояжера для berlin52.

2 Краткие теоретические сведения

Муравьиные алгоритмы (МА) основаны на использовании популяции потенциальных решений и разработаны для решения задач комбинаторной оптимизации, прежде всего, поиска различных путей на графах. Кооперация между особями (искусственными муравьями) здесь реализуется на основе моделирования. При этом каждый агент, называемый искусственным муравьем, ищет решение поставленной задачи. Искусственные муравьи последовательно строят решение задачи, передвигаясь по графу, откладывают феромон и при выборе дальнейшего участка пути учитывают концентрацию этого фермента. Чем больше концентрация феромона в последующем участке, тем больше вероятность его выбора.

3 Результаты выполнения работы

3.1 Поиск Гамильтонова пути

Сходимость решения к оптимальному ответу при использовании созданного МА довольно быстрая даже при малых размерах популяции:

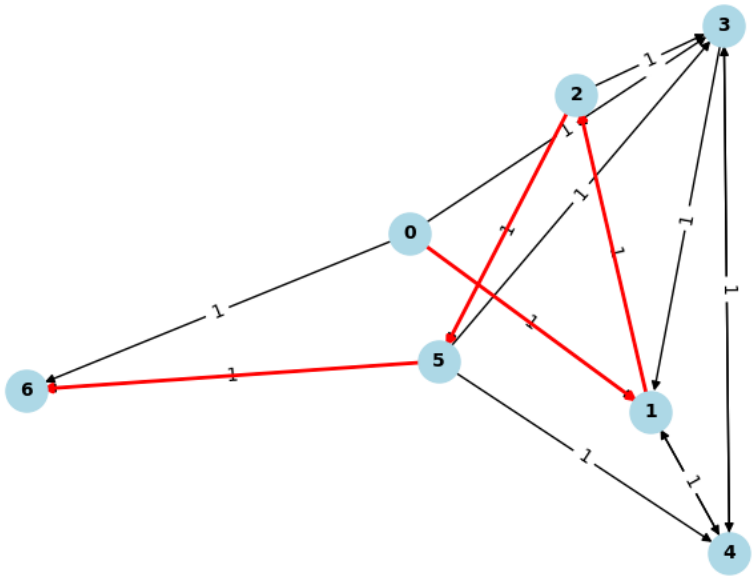


Рисунок 1 – размер колонии – 5, alpha=0.9, rho=0.2, первая итерация

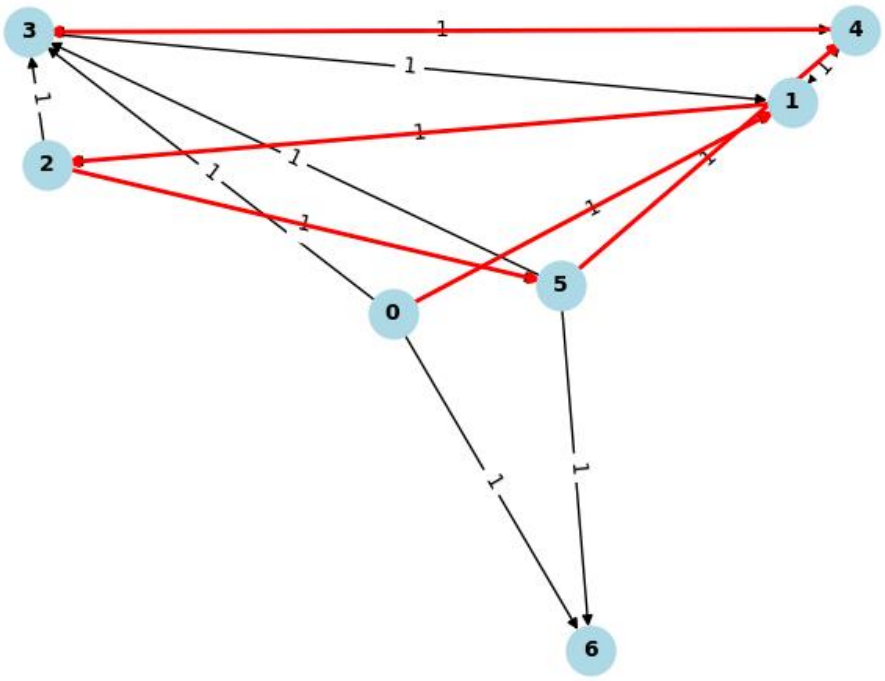


Рисунок 2 – размер колонии – 5, alpha=0.9, rho=0.2, вторая итерация

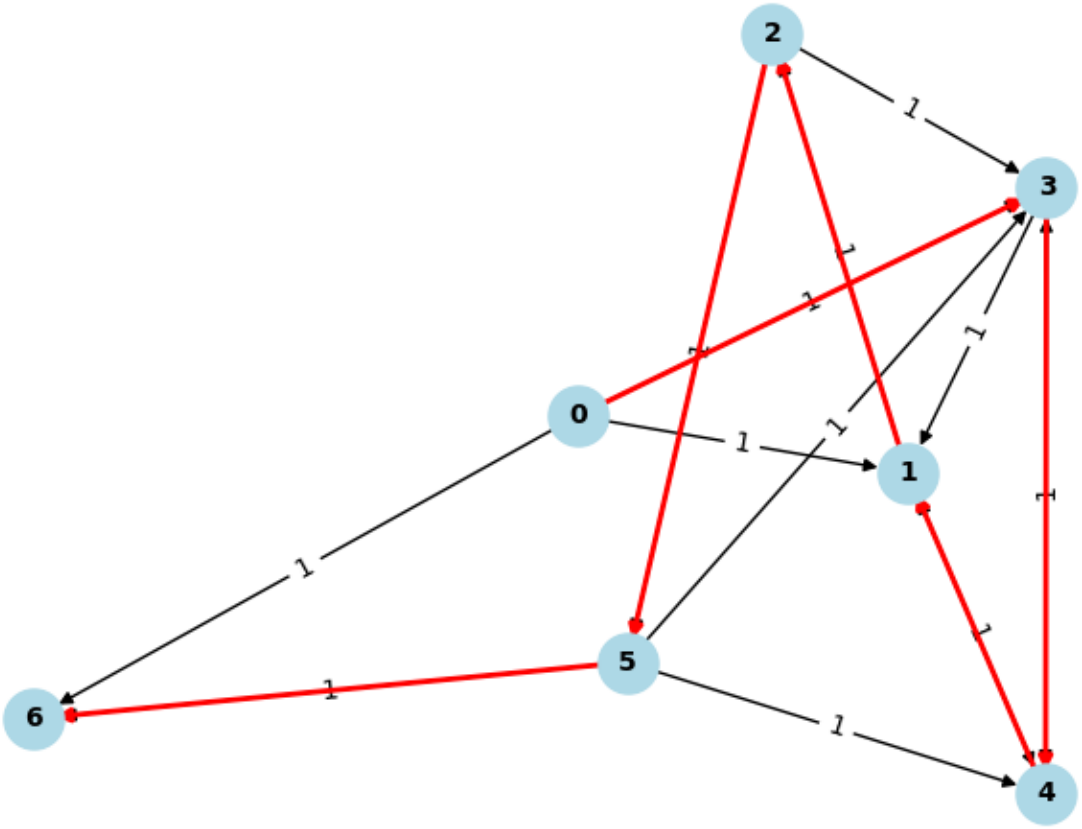


Рисунок 3 – размер колонии – 5, alpha=0.9, rho=0.2, третья итерация

```
C:\Python312\python.exe F:\Coding\py\evolutionary-programming\antAlgorithms\main.py
Iteration [1/10]        Visited nodes = 5        total pheromone = 4.871

Iteration [2/10]        Visited nodes = 6        total pheromone = 6.478

Iteration [3/10]        Visited nodes = 7        total pheromone = 7.799

Iteration [4/10]        Visited nodes = 7        total pheromone = 8.589

Iteration [5/10]        Visited nodes = 7        total pheromone = 8.805

Iteration [6/10]        Visited nodes = 7        total pheromone = 9.461

Iteration [7/10]        Visited nodes = 7        total pheromone = 10.252

Iteration [8/10]        Visited nodes = 7        total pheromone = 10.201

Iteration [9/10]        Visited nodes = 7        total pheromone = 9.828

Iteration [10/10]       Visited nodes = 7        total pheromone = 10.062


 ============================================================================================
Required time: 2.48s. Found answer: [0, 3, 4, 1, 2, 5, 6].  Required generations: 10. Total pheromone: 10.062
```

Всего за 3 эпохи был получен корректный путь, включающий все вершины графа по одному разу.

3.2 Решение задачи коммивояжера

В ходе использования МА для решения задачи коммивояжера, было установлено, что при правильных параметрах (в данном случае – 200 особей, alpha=0.95, pho=0.8) алгоритм быстро сходится к оптимальным решениям (по сравнению с генетическими алгоритмами):

```
Iteration 1/50: Best path length = 17192.720152943304
Iteration 2/50: Best path length = 16377.83228383039
Iteration 3/50: Best path length = 15189.77766994064
Iteration 4/50: Best path length = 11857.814384030462
Iteration 5/50: Best path length = 10576.357250971305
Iteration 6/50: Best path length = 9907.443261957456
Iteration 7/50: Best path length = 9660.319025522596
Iteration 8/50: Best path length = 9173.796237262297
Iteration 9/50: Best path length = 8884.62721166434
Iteration 10/50: Best path length = 8855.668126227338
                                ...
Iteration 48/50: Best path length = 8019.545189926506
Iteration 49/50: Best path length = 8019.545189926506
Iteration 50/50: Best path length = 8019.545189926506
Best path length: 8019.545189926506, Time: 666.68s
```

Рисунок 4 – размер колонии – 200, alpha=0.95, rho=0.8, третья итерация

Решение, полученное в ходе выполнения третьей ЛР, немного уступает по количеству затраченных поколений и качеству ответа, но сильно превосходит в скорости решения:

```
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.
```

Рисунок 5 – лучшее решение из ЛР 3 с использованием ГА (популяция – 1000, вероятность кроссинговера – 0.9, мутации – 0.01)

Графическое отражение решения наглядно показывает, как происходило решение оптимизационной задачи:

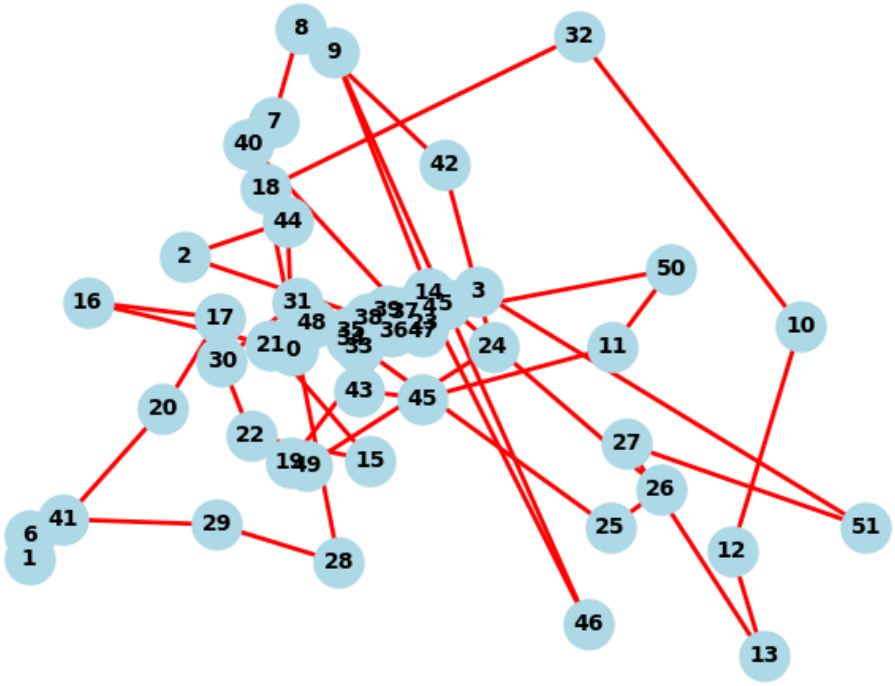


Рисунок 6 – итерация 1, длина пути 17192.72

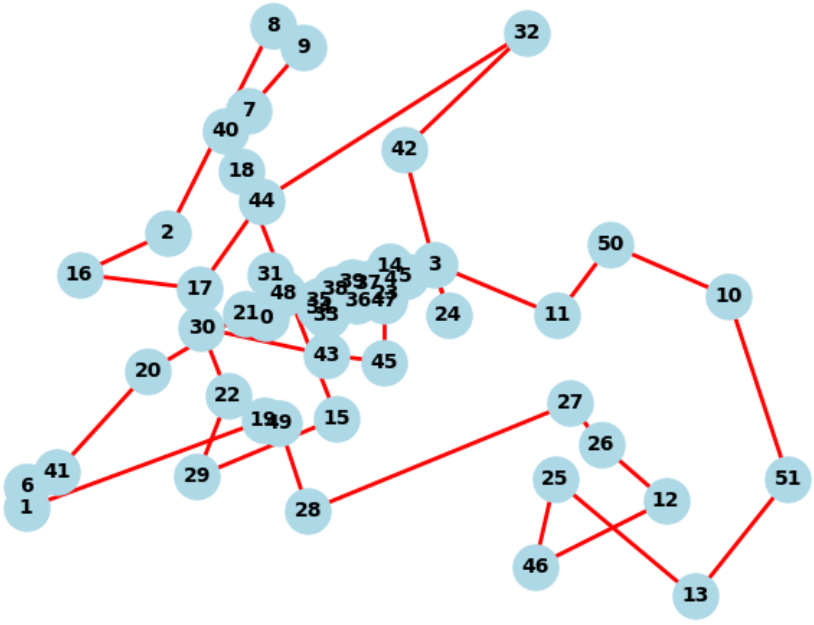


Рисунок 7 – итерация 5, длина пути 10576.35

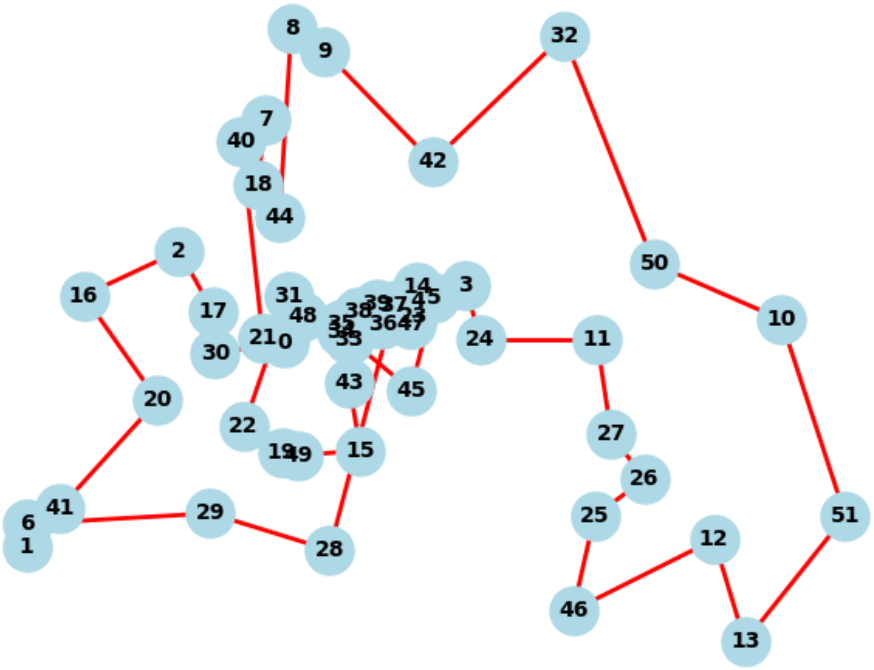


Рисунок 8 – итерация 10, длина пути 8855.66

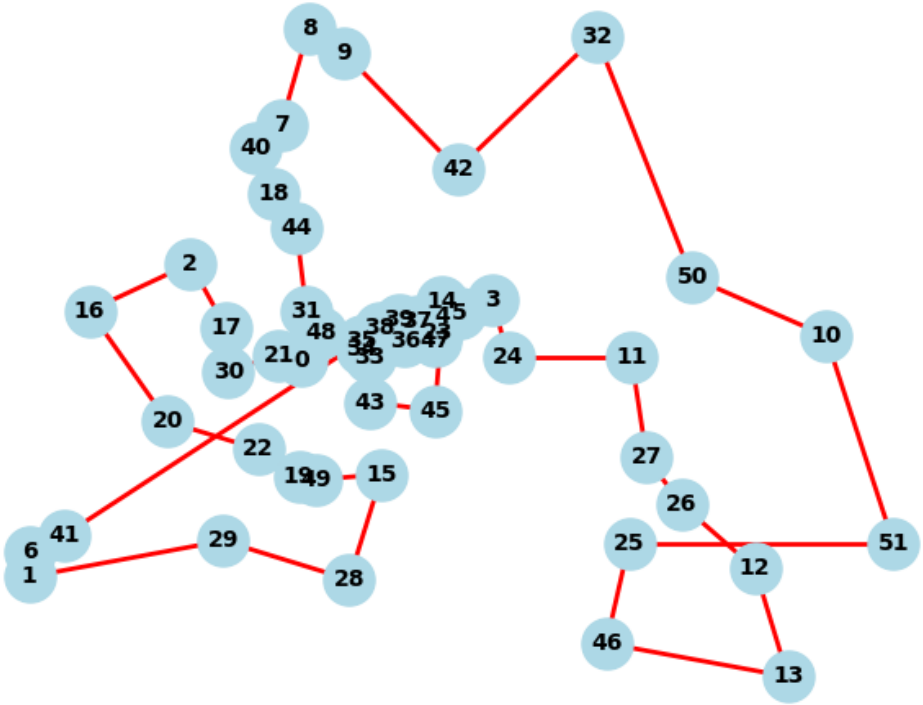


Рисунок 9 – итерация 30, длина пути 8674.40

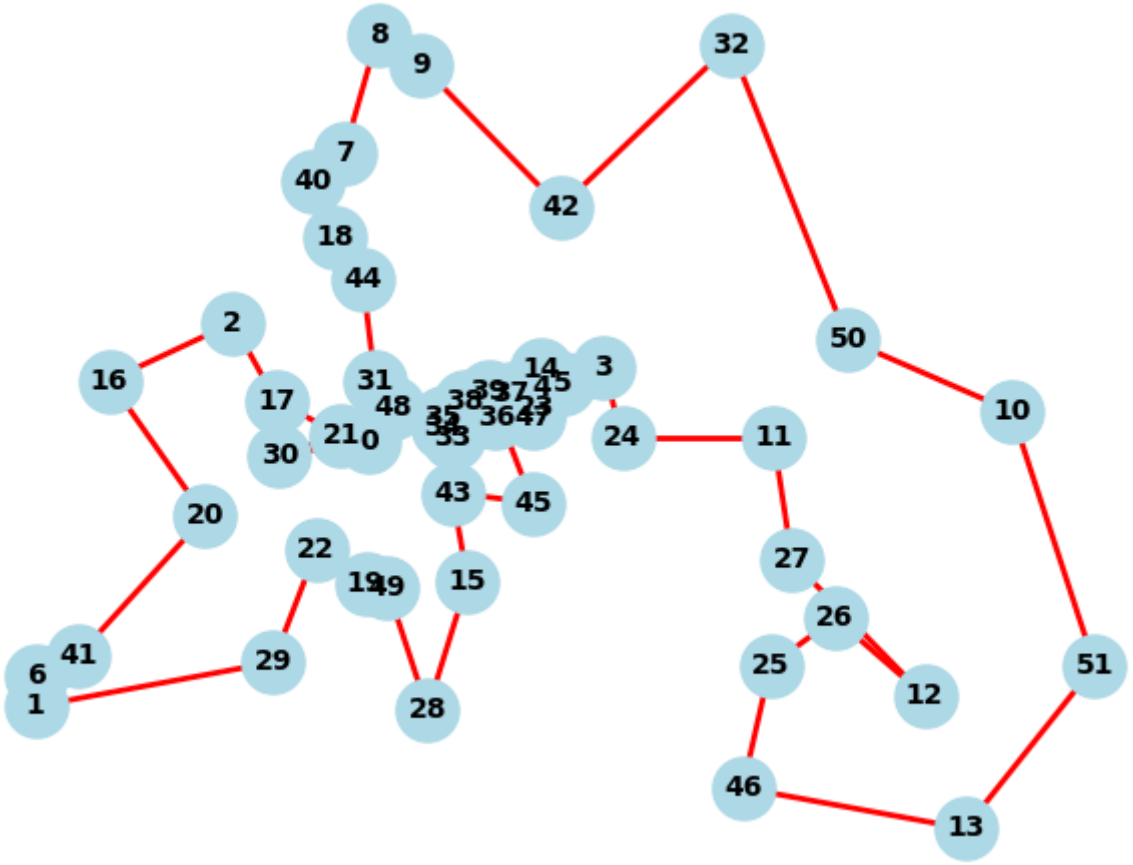
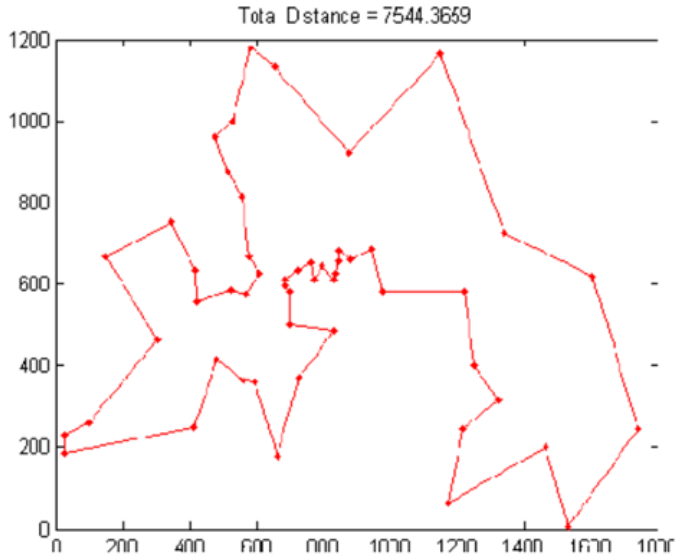


Рисунок 10 – итерация 50, длина пути 8019.54

Оптимальное же решение имеет длину пути 7544.36. Полученный путь достаточно близок к оптимальному и совпадает с ним во многих точках:



3.4 Листинг программы

```
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np

from time import time
import random


class AntColonyTask:
    """
    Solves task of finding Hamilton path in oriented graph.
    :param population_size: size of the population
    :param max_iterations: number of max available iterations
    :param alpha: influence of pheromone
    :param rho: coefficient of pheromone evaporation
    """

    def __init__(self, population_size: int, max_iterations: int, alpha:
float, rho: float,
                 distance_matrix, coords=None):
        self.population_size = population_size
        self.max_iterations = max_iterations
        self.alpha = alpha
        self.rho = rho
        self.distance_matrix = distance_matrix
        self.coords = coords

        self.total_nodes = distance_matrix.shape[0]
        self.pheromone_matrix = np.ones_like(distance_matrix) /
len(distance_matrix)

    def calculate_transition_probabilities(self, current_node, unvisited):
        probabilities = []
        total_tau = 0.

        for next_node in unvisited:
            if self.distance_matrix[current_node][next_node] > 0:
                total_tau +=
(self.pheromone_matrix[current_node][next_node] ** self.alpha)

        for next_node in unvisited:
            probabili-
ties.append(self.pheromone_matrix[current_node][next_node] ** self.alpha
                                    / total_tau if
self.distance_matrix[current_node][next_node] > 0 else 0)

        probabilities = np.array(probabilities)
        return probabilities / probabilities.sum() if probabilities.sum() >
0 else probabilities

    def calculate_transition_probabilities_tsp(self, current_node, unvisit-
ed):
        pheromones = self.pheromone_matrix[current_node, list(unvisited)]
        distances = self.distance_matrix[current_node, list(unvisited)]

        attractiveness = pheromones ** self.alpha / distances
        return attractiveness / attractiveness.sum()

    def start_hamilton(self):
```

```
        iteration = 1  # iterations counter

        best_path = None
        best_unvisited_total = float('inf')
        best_visited_total = 0

        start_time = time()

        for iteration in range(self.max_iterations):
            paths = []
            path_lengths = []

            for ant in range(self.population_size):
                current_node = 0
                path = [current_node]
                unvisited = set(range(self.total_nodes)) - {current_node}

                while current_node != self.total_nodes - 1:
                    probabilities =
self.calculate_transition_probabilities(current_node, unvisited)

                    if np.sum(probabilities) != 1.:
                        break

                    next_node = random.choices(list(unvisited),
weights=probabilities, k=1)[0]
                    unvisited.remove(next_node)
                    path.append(next_node)
                    current_node = next_node

                unvisited_total = len(unvisited)
                visited_total = self.total_nodes - unvisited_total
                path_lengths.append(visited_total)
                paths.append(path)

                # Обновляем лучший путь
                if unvisited_total < best_unvisited_total:
                    best_unvisited_total = unvisited_total
                    best_visited_total = visited_total
                    best_path = path

            # Испарение феромона
            self.pheromone_matrix *= (1 - self.rho)

            # Обновление феромона
            for path, visited_total in zip(paths, path_lengths):
                for i in range(len(path) - 1):
                    self.pheromone_matrix[path[i]][path[i + 1]] += 1 /
(visited_total + 1)

            print(f'Iteration [{iteration + 1}/{self.max_iterations}]\t\t'
+
                  f'Visited nodes = {best_visited_total}\t\t total phero-
mone = {np.sum(self.pheromone_matrix):.3f}\n')

            self.draw_graph(iteration, best_visited_total, best_path)

            iteration += 1

        end_time = time()
```

```
        print('\n', '=' * 100)
        print(f'Required time: {end_time - start_time:.2f}s. Found answer:
{best_path}. ',
              f'Required generations: {iteration}. Total pheromone:
{np.sum(self.pheromone_matrix):.3f}')

    def calculate_path_length(self, path):
        return sum(self.distance_matrix[path[i]][path[i + 1]] for i in
range(len(path) - 1))

    def two_opt(self, path):
        best_path = path
        best_length = self.calculate_path_length(path)
        improved = True

        while improved:
            improved = False
            for i in range(1, len(path) - 2):
                for j in range(i + 1, len(path) - 1):
                    new_path = path[:i] + path[i:j + 1][::-1] + path[j +
1:]
                    new_length = self.calculate_path_length(new_path)
                    if new_length < best_length:
                        best_path = new_path
                        best_length = new_length
                        improved = True
        return best_path

    def start_tsp(self):
        best_path = None
        best_path_length = float('inf')
        start_time = time()

        for iteration in range(self.max_iterations):
            paths = []
            path_lengths = []

            for ant in range(self.population_size):
                current_node = random.randint(0, self.total_nodes - 1)
                path = [current_node]
                unvisited = set(range(self.total_nodes)) - {current_node}

                while unvisited:
                    probabilities =
self.calculate_transition_probabilities_tsp(current_node, unvisited)
                    next_node = random.choices(list(unvisited),
weights=probabilities, k=1)[0]
                    unvisited.remove(next_node)
                    path.append(next_node)
                    current_node = next_node

                path.append(path[0])  # Замыкаем путь

                path = self.two_opt(path)  # Локальная оптимизация
                path_length = self.calculate_path_length(path)

                paths.append(path)
                path_lengths.append(path_length)

                if path_length < best_path_length:
                    best_path_length = path_length
```

```
                    best_path = path

            # Испарение и обновление феромонов
            self.pheromone_matrix *= (1 - self.rho)

            for path, length in zip(paths, path_lengths):
                for i in range(len(path) - 1):
                    self.pheromone_matrix[path[i]][path[i + 1]] += 1 /
length

            print(f"Iteration {iteration + 1}/{self.max_iterations}: Best
path length = {best_path_length}")
            self.draw_with_coords(iteration, best_path_length, best_path)

        end_time = time()
        print(f"Best path length: {best_path_length}, Time: {end_time -
start_time:.2f}s")

    def draw_with_coords(self, iteration, best_len=float('inf'),
best_path=None):
        G = nx.Graph()
        for i, (x, y) in enumerate(self.coords):
            G.add_node(i, pos=(x, y))

        pos = nx.get_node_attributes(G, 'pos')
        nx.draw(G, pos, with_labels=True, node_color='lightblue',
node_size=500, font_size=10, font_weight='bold')

        if best_path:
            path_edges = [(best_path[i], best_path[i + 1]) for i in
range(len(best_path) - 1)]
            nx.draw_networkx_edges(G, pos, edgelist=path_edges,
edge_color='red', width=2)

        plt.title(f"Iteration {iteration}. Length: {best_len:.2f}")
        plt.show()

    def draw_graph(self, iteration, best_len=float('inf'), best_path=None):
        G = nx.DiGraph()
        for i in range(len(self.distance_matrix)):
            for j in range(len(self.distance_matrix)):
                if self.distance_matrix[i][j] > 0:
                    G.add_edge(i, j, weight=self.distance_matrix[i][j])

        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, node_color='lightblue',
node_size=500, font_size=10, font_weight='bold')
        nx.draw_networkx_edge_labels(G, pos, edge_labels={(i, j):
f"{w['weight']}" for i, j, w in G.edges(data=True)})

        if best_path:
            path_edges = [(best_path[i], best_path[i + 1]) for i in
range(len(best_path) - 1)]
            nx.draw_networkx_edges(G, pos, edgelist=path_edges,
edge_color='red', width=2)

        plt.title(f"Iteration {iteration}. Length: {best_len}")
        plt.show()

import numpy as np
import genalg
```

```
POPULATION_SIZE = 200
ALPHA = 0.95
RHO = 0.8
MAX_GENERATIONS = 50

graph_hamilton = np.array([
    # 0  1  2  3  4  5  6
    [0, 1, 0, 1, 0, 0, 1],  # 0
    [0, 0, 1, 0, 1, 0, 0],  # 1
    [0, 0, 0, 1, 0, 1, 0],  # 2
    [0, 1, 0, 0, 1, 0, 0],  # 3
    [0, 1, 0, 1, 0, 0, 0],  # 4
    [0, 0, 0, 1, 1, 0, 1],  # 5
    [0, 0, 0, 0, 0, 0, 0],  # 6
])


def parse_tsp_file(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()

    coords = []
    start = False
    for line in lines:
        if line.startswith("NODE_COORD_SECTION"):
            start = True
            continue
        if start:
            if line.startswith("EOF"):
                break
            parts = line.split()
            coords.append((float(parts[1]), float(parts[2])))  # (x, y)
    return coords


def calculate_distance(coord1, coord2):
    return np.sqrt((coord1[0] - coord2[0]) ** 2 + (coord1[1] - coord2[1])
** 2)


def generate_distance_matrix(coords):
    n = len(coords)
    graph = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i != j:
                graph[i][j] = calculate_distance(coords[i], coords[j])
    return graph


FILENAME = "berlin52.tsp"
coords = parse_tsp_file(FILENAME)
graph_tsp = generate_distance_matrix(coords)


if __name__ == '__main__':
    # genetic_optimizer = genalg.AntColonyTask(POPULATION_SIZE,
MAX_GENERATIONS, ALPHA, RHO, graph_hamilton)
    # genetic optimizer.start hamilton()
```

```
    genetic_optimizer = genalg.AntColonyTask(POPULATION_SIZE,
MAX_GENERATIONS, ALPHA, RHO, graph_tsp, coords)
    genetic_optimizer.start_tsp()
```

4 Вывод

В ходе работы была создана программа для решения задач на графах с использованием муравьиных алгоритмов. Полученное решение является весьма эффективным.